GVSC-GVR FY20

Ground Vehicle Robotics

U.S. Army CCDC Ground Vehicle Systems Center

ROS-M

Evaluation of ROS2 Eloquent

Matthew Schickler

March 2020

THIS PAGE IS INTENTIONALLY LEFT BLANK

Evaluation of ROS2 Eloquent

Matthew Schickler

FLIR Unmanned Ground Systems
19 Alpha Road
Chelmsford, MA 01824

Prepared for U.S. Army CCDC GVSC
Warren, MI 48397-5000

Executive Summary

The Robot Operating System (ROS) has proven to be an invaluable framework and package repository for the development of robot software applications. It is used heavily by the research community and there is a strong desire to also use it to produce finished products for industry. Despite its success, there are fundamental weaknesses in its architecture that have been a barrier to wide-spread adoption for production. The latest version of ROS, called ROS2, is a clean sheet design that aims to achieve significant improvements in reliability and security over the original ROS. These improvements are of great interest to the United States Army Combat Capabilities Development Command (CCDC) Ground Vehicle Systems Center (GVSC) and members of the National Advanced Mobility Consortium (NAMC) because these features are necessary to achieve the requirements of future unmanned ground vehicles (UGV). In the interest of moving the state of the art forward on a common, shared software platform, NAMC is examining ways to expedite the adoption of ROS2 (and more specifically, the military version of ROS, called ROS-M) as the de facto standard for UGV software. This report contains a survey of ROS2 features and identifies gaps and remaining work needed to bring ROS2 to parity with the existing ROS1 feature set.

Table of Contents

1 Overview

ROS1 has been under active development by an enthusiastic community of robotics researchers and software developers for over 12 years and provides an extremely diverse and rich set of core and auxiliary features. ROS2 is a ground-up redesign of ROS1 that was announced in 2014. Starting in 2017, four official distributions have been delivered with each release incrementally adding back capabilities originally offered by ROS1. As of the Eloquent distribution released in November 2019, most of the remaining feature gaps between ROS1 and ROS2 have been filled. The next distribution, Foxy, will be released in May 2020 and is planned for long-term support.

As of the end of 2019, there are almost 2500 separate packages available in ROS1 Melodic. It is therefore not the intention of this paper to provide a complete, detailed analysis of how the current ROS1 and ROS2 distributions differ. Instead, we focus on a survey of the core ROS2 infrastructure and a subset of community developed packages that are of particular interest to those building UGVs.

In the survey, the software features of ROS are broken down into the following major categories:

- Build System
- Core
- Algorithms
- Device Drivers
- Tools

Most aspects of the build system, core, and tools are of interest to any ROS developer, including those developing for ROS-M projects.

Within the algorithms category, ROS-M developers are expected to have particular interest in at least some of the following sub-categories:

- Diagnostics
- Controllers
- State Estimation
- Simultaneous Localization and Mapping (SLAM)
- Navigation
- Perception
- Manipulation

Within the device drivers category, ROS-M developers are expected to have particular interest in at least some of the following sub-categories:

- CAN
- GPS
- IMU
- LiDAR
- Cameras

After presenting a survey of ROS2 features, this paper identifies remaining gaps and areas of future focus that will facilitate faster migration to ROS2.

2 ROS2 Survey

This section presents a survey of the current ROS2 feature set as of the November 2019 release of Eloquent and compares and contrasts with ROS1 Melodic.

2.1 Build System

ROS1 uses a package called catkin to extend the built-in capabilities of CMake. The catkin tool provides a comprehensive mechanism for building ROS1 packages and managing their dependencies. ROS2 continues to use CMake at its core, but introduces a new package called ament which is a major redesign and evolution of catkin. In addition, a higher level tool called colcon has been introduced that has the ability to build both catkin and ament packages. Colcon is now the preferred method for performing a ROS2 build.

In ament, the contents of the CMakeLists.txt and package.xml files are changed significantly. Also, there is no longer a "devel" directory at the top level of the workspace. Build products such as libraries and executables are output into a package-specific subdirectory under the "build" directory. This arrangement aligns more closely with what is expected from a standard CMake build. To support quick development iterations there is an option called *–symlink-install*, which creates symbolic links to installed files instead of copies. This allows, for example, changes to a launch file in the source directory to take effect immediately without requiring a re-build.

From a developer's perspective, ament offers the same basic functionality as catkin. Many of the advantages of ament over catkin are related to its underlying design.

2.2 Core

The ROS core functions consist of the middleware, services, utilities, and programming APIs that make up the infrastructure used by all ROS packages.

2.2.1 Nodes

In ROS1, a node was more of a concept than a programmatic entity. A ROS1 node was simply an executable that used ROS topics or services to communicate with other ROS1 nodes. In ROS2, a ROS node is implemented by deriving from a Node base class. There is also no assumption that there is a one-to-one mapping between a node and an executable. A single executable can contain multiple nodes and the activities of these nodes are orchestrated through objects called executors. Executors can be single threaded or multi-threaded.

An additional benefit of deriving from the Node base class is that it provides a standard design framework from within which the rest of the code operates. The main function of your process simply instantiates your node objects and an optional executor and then calls spin. The operation of your node object is then completely event driven through timer, subscriber, and service callbacks.

In addition to supporting the same level of functionality as ROS1 nodes, ROS2 also supports the concept of a managed lifecycle for a node. This managed lifecycle consists of a set of well-known states and transitions. When a transition is requested, callbacks are made in order to give the node a chance to take appropriate actions. For example, a node might read its configuration parameters and allocate resources appropriately when transitioning from the "Unconfigured" to the "Inactive" state. The lifecycle is fully integrated with command line tools and the ROS2 launch system. This makes it possible to manually request state transitions or to sequence state transitions based on launch events.

2.2.2 Communication

ROS1 nodes use topics, services, and actions for communication. Topics are an asynchronous, many-to-many message passing mechanism based on the publish/subscribe pattern. The association between a publisher (writer of data) and a subscriber (reader of data) is created indirectly through a named topic. Publishers typically write to a topic periodically, but may also write new data immediately as changes occur. In addition, data for a topic may be "latched" so that new subscribers can always read the data that was most recently written.

Services are a synchronous mechanism whereby one node sends a request and then waits for another node to produce a response.

Actions are like services but the transaction includes a set of intermediate feedback that can report on the progress of a long-running process that was initiated by the request. An action client sends a goal request to initiate an action and then has the option of either blocking and waiting for a response or not blocking and registering callbacks that will asynchronously handle the feedback and result responses.

ROS2 supports all of the communication mechanisms supported by ROS1. In addition, actions, which were an add-on package in ROS1 are now a part of the core ROS2 API. The interfaces for managing all three types of communication are included in the Node base class.

It should be mentioned that there is a difference in how services are implemented in ROS1 and ROS2. In ROS1, a service client blocks until it times out or receives a response. In ROS2, a service client is always asynchronous, meaning that the response from the server must be handled in a callback function. This is a good design choice since a blocking service call would not be real-time safe. Note that while there is no API for a synchronous client, it is still possible, when appropriate, to implement it using a combination of an asynchronous client and a spinner.

2.2.3 Middleware

ROS1 uses a custom transport protocol and a central master process to coordinate the setup of communication channels between publishers and subscribers. ROS2 is based on Distributed Data Services (DDS) middleware and does not require a central master. It should be noted that a fully distributed solution does not come without drawbacks. In ROS2, endpoint discovery must occur every time a process is executed which incurs a significant delay. This problem is mitigated through the use of an optional ROS Daemon process that caches discovery information for short-lived executables such as command line tools.

The DDS implementation used by ROS2 also offers important security and quality of service features. Some of these features are not supported by ROS1.

ROS2 currently benefits from the following DDS security capabilities that are not supported in ROS1:

- Privacy: Strong symmetric key encryption (e.g. AES) of messages
- Authentication: Private key cryptography and X.509 certificates
- Authorization: Read/write permissions on a per-node, per-topic basis
- Authorization configuration files are write-protected using cryptographic signature

ROS2 currently exposes the following options for implementing quality of service:

- Best effort or reliable transport. This is similar to the choice of ROSUDP or ROSTCP in ROS1 but in ROS2 this choice can be made on a per-topic basis. Also, unlike ROSTCP, reliable DDS transport supports multicast for increased network efficiency.
- Transmit and receive history settings. These are similar to setting the queue depth in ROS1.
- Volatile or "transient local" durability. The latter attempts to send "historical" messages to newly started subscribers and behaves like a latched topic in ROS1 if the topic history depth is also set to 1.
- Deadlines. Monitors the timeliness of periodic messages and notifies if updates are not sent or received within the expected timeframe. This is a new capability in ROS2.
- Lifespan. Prevents delivery of stale data by throwing out messages that have expired timestamps. This is a new capability in ROS2.
- Liveliness. Monitors the liveliness of a publisher and notifies if liveliness is lost. Similar monitoring of liveliness is accomplished in ROS1 using bonds (see section 2.2.9).

Note that features that would be helpful for supporting determinism are not yet exposed through the ROS2 API. These features include:

- Prioritization: higher priority messages should be serviced from queues before lower priority messages and when a queue is full, lower priority messages should be dropped before higher priority messages
- Time Sensitivity: real-time applications should have the ability to identify the age of messages and use that information to determine a processing order that can be used to guarantee deadlines

Some further discussion on proposed enhancements to the ROS2 Middleware for supporting deterministic execution can be found at [1].

Because of the open nature and standardization of DDS, system developers can choose from a range of free DDS implementations or use a licensed, commercial implementation. Table 1 lists the current DDS implementations that may be used with ROS2.

Table 1. ROS2 middleware options.

Product	License	Notes
eProsima Fast RTPS	Apache2	ROS2 default middleware
ADLINK Opensplice	Apache2	support will be discontinued in favor of Cyclone [2]
RTI Connext	Commercial	Limited to v5.3.1 [3], must be licensed
Eclipse Cyclone	EPL 2.0	Missing security and QoS modes

The default DDS implementation for ROS2 is Fast RTPS from eProsima. The company has published test results in [4] showing their implementation has the lowest latency and highest throughput of the non-commercial options.

eProsima announced in February 2019 that Apex.AI, a company dedicated to using ROS to build a self-driving car, had selected Fast RTPS for one of its two backbone middlewares [5]. On the other hand, Rover Robotics published guidance in July 2019 on choosing a DDS implementation based on their direct experience with the Fast RTPS software. After having difficulties with discovery in Fast RTPS, the authors recommended using Opensplice [6].

A performance analysis comparing OpenSplice, RTI Connext, and the ROS1 middleware was written in 2016 [7]. At that time, the authors found that

DDS latency was about the same, or in some cases worse, than latency in ROS1. An updated version of this study including Fast RTPS, OpenSplice, and Connext would be very helpful to understanding the current middleware performance characteristics of ROS2.

A security analysis of ROS2 with Fast RTPS published in Sep 2018 concluded that "After analyzing the default DDS middleware with ROS 2 we found that it did not conform to the security specification by OMG [the Object Management Group, the consortium responsible for the DDS specification]. This can cause a [Cyber-Physical System] to be left in vulnerable situations". Also, due to rapid development of ROS2, they note "Continuously monitoring the implementation of ROS 2 to check violations to the relevant security specifications and logical errors, which can be linked to potential vulnerabilities is greatly beneficial to enhance the security of ROS 2" [8].

2.2.4 Components

ROS1 includes an add-on feature that allows developers to restructure their nodes into objects, called nodelets, that can be dynamically loaded and run within a single process. The main advantage of this technique is highly efficient zero-copy message passing between nodelets in the same process. This same capability is implemented in ROS2 through Components, which are essentially dynamically loadable ROS2 nodes. Since ROS2 already fundamentally supports running multiple nodes in the same process and zero-copy message passing occurs between those nodes using DDS, equivalent functionality to nodelets is achieved. An interesting and potentially useful characteristic of ROS2 components is that they can be dynamically composed into processes using command line tools. Since it only requires a few extra lines of code in your package, it is recommended that all ROS2 nodes are implemented as components for maximum flexibility [9].

2.2.5 Launch

ROS1 uses XML files to describe which nodes to run and how they should be configured. Launch files can include other launch files so that the recipes for launching nodes can be reused and customized for different situations or platforms. Configuration parameters can be specified as children of the XML tags that specify the execution of a node or they may be loaded out of YAML files. The XML also allows for some limited decision making. For example, "if" and "unless" attributes can be added to the XML tags to control

whether or not the tag should have an effect based on the value of a launch argument. While easy to learn and to use, the launch XML has a few drawbacks:

- The order in which nodes are launched cannot be controlled
- Nodes tend to all be launched at once without regard to sequencing or dependencies
- Decisions more complicated than checking the value of a single launch argument are cumbersome or potentially not even possible

ROS2 introduces Python-based launch files that increase flexibility and allow for the sequencing of launch steps. The python launch API implements launch actions based on ROS2 node life cycle events. For example, a life cycle node entering the active state may trigger a launch action that executes additional nodes or transitions another node to a new state.

An XML launch format has also recently been released. It is based on the underlying Python launch APIs and is similar to but not exactly the same as the ROS1 XML format. A complete guide for migrating ROS1 XML to ROS2 XML is available at [10].

The following ROS1 XML features are not yet supported in ROS2:

- Respawn and respawn_delay attributes in the node tag. There is an open enhancement request for this capability [11].
- Machine attribute and tag. These are missing because the remote launch feature is not yet implemented.

2.2.6 Parameters

ROS1 uses a global parameter server that maintains a single tree of parameters for the entire system. The drawback of this approach is that, in general, nothing has ownership of a parameter.

ROS2 uses local parameters that are directly associated with a node. The lifetime of a parameter is implicitly tied to the node's lifetime. Also, the

node has control over how the parameter can be set. This allows nodes to provide parameter constraint information to clients and validate requests to modify parameters. Through the use of callbacks in a ROS2 node, all of the dynamic reconfigure functionality provided by ROS1 in an add-on package is now directly supported by the ROS2 client library. Parameters can be dynamically set through the ROS2 command line tools or through the RQt GUI.

Transitioning from an architecture that uses a global parameter server to one that uses local parameters owned by individual nodes is a significant change that will likely complicate the porting of some existing ROS1 code. In cases where a group of nodes must share parameters, it is recommended that a specific node with the responsibility of managing that cohesive set of parameters is added to the design. While it is possible to add a generic, global parameter server node that would completely replace the ROS1 parameter server [12], this approach violates the principle of least privilege (POLP) [13] and weakens the overall security of the system [14].

2.2.7 Plugins

Plugins allow applications to be extended without the need for recompilation. Specific use of plugins in both ROS1 and ROS2 include:

- planning algorithms in the navigation stack
- data display types in the RViz visualization tool
- kinematics, collision detection, and planning algorithms in MoveIt!

The ROS1 pluginlib package has been ported to ROS2 and is available at [15].

2.2.8 Logging

Both ROS1 and ROS2 provide the ability to write log messages to the console. Each message is assigned one of the following severities which are used to categorize and filter messages by level:

- DEBUG - low-level tracing for development or troubleshooting

- INFO - normal operation
- WARN - indicates there might be a problem
- ERROR - a serious, but recoverable problem has occurred
- FATAL - something unrecoverable has occurred

The following ROS1 logging features are available in ROS2:

- Logs are automatically written to files on a per-node basis
- Use of C-style format string or C++ IO stream for messages
- Throttling to limit the frequency of a particular message
- Log only when a particular condition is true (called "expressions" in ROS2)
- Log once

The following logging features have been added in ROS2:

- Log based on the boolean result of a general function
- Skip first log message

The following ROS1 logging features are missing in ROS2:

- Throttle with an initial delay.
- Filter based on formatted log message contents.
- A generalized mechanism for changing the level of individual loggers at runtime. This is a powerful debugging tool that gives developers fine-grained control over which logs they see while troubleshooting an issue. Currently specific code needs to be added to each node in order to control its logger level.

2.2.9 Transforms

ROS1 includes packages containing support for calculating the forward kinematics of a robot model and converting between its coordinate frames. These packages have been carried forward to ROS2 [16].

2.2.10 Bonding

Bonding allows a pair of processes to monitor each other's liveliness. The ROS1 bond package has been ported to ROS2 and is available at [17].

2.3 Algorithms

ROS contains many support packages contributed by the community that implement algorithms used in the field of robotics.

2.3.1 Diagnostics

Table 2 contains packages that support system diagnostics in ROS2.

Table 2. Diagnostics packages.

Package	Description	Eloquent Status	Repo
diagnostic_updater	tools for easily updating the diagnostics topic	Debian exists [1]	[18]

ROS2 currently does not support:

- diagnostic_aggregator: a node that uses analyzer plugins to process and categorize diagnostics data
- rqt_robot_monitor: a plug-in that allows viewing of diagnostic status through the RQt GUI

Both of these missing ROS1 features are useful for making sense of the diagnostic data being published by the updated.

[1] this means that this package can be installed directly from the ROS2 package repository at http://packages.ros.org/ros2/ubuntu

2.3.2 Controllers

ROS1 includes packages that implement a wide range of controllers for robot hardware [19] [20]. These packages have not yet been ported to ROS2 but as of January 2020 there was interest by Amazon and PAL Robotics to form a working group with the goal of doing a conversion [2]. Table 3 contains a list of new packages that are a redesign of a subset of the ROS1 controls packages aimed at fully leveraging ROS2 concepts.

Table 3. Controller packages.

Package	Description	Eloquent Status	Repo
ros_control	controller manager, robot hardware interface	Build from source	[21]
ros_controller	useful controllers such as joint trajectory, etc.	Build from source	[22]

Code for performing differential drive and ackermann vehicle control are missing from the new ROS2 controls packages. Since ground vehicles typically employ one of these two drive schemes, it would be useful for these features to be added.

2.3.3 State Estimation

State estimation uses a combination of sensors such as GPS, IMU, and encoders to determine the pose of a robot.

Table 4 contains packages that support state estimation in ROS2.

Table 4. State Estimation packages.

Package	Description	Eloquent Status	Repo
robot_localization	non-linear state estimation through sensor fusion	Build from source	[23]

2.3.4 SLAM

Simultaneous Localization and Mapping (SLAM) is a technique for mapping an environment and determining the robot's pose within that environment.

Table 5 contains packages that support SLAM in ROS2.

Table 5. SLAM packages.

Package	Description	Eloquent Status	Repo
slam_toolbox	Lifelong mapping and localization	Build from source, will be default SLAM for ROS2	[24]
cartographer	Cartographer SLAM	Debian exists	[25]

Although Cartographer is a very capable SLAM implementation it has not been chosen by the ROS2 technical steering committee as the default ROS2 SLAM because it is no longer actively supported by Google. That said, Cartographer will likely still be of interest to projects that require 3D SLAM until an alternative has been identified.

There has also been discussion at with regard to porting the newer LaMa SLAM algorithm to ROS2 [26].

2.3.5 Navigation

Navigation involves planning collision-free paths through an environment (global planning) and controlling a robot's motion to achieve the waypoints along that path (local planning). The default navigation stack in ROS1 was called *move_base*. This has been replaced in ROS2 by *navigation2*. Major features of this new navigation stack include:

- Task coordination using behavior trees
- *bt_navigator* replaces *move_base* at top level
- Cost map and planner implementations are still plugins
- Planners and recovery behaviors are executed using ROS2 actions
- All nodes use the ROS2 node life cycle model
- Generalized world model framework for providing a more flexible set of cost map outputs
- Global planning using Dijkstra or A*
- Local planning using the Dynamic Window Approach algorithm (DWA)
- Adaptive Monte Carlo Localization (AMCL)

A recent real-world experiment demonstrated that a robot controlled by the Navigation2 software could move autonomously through a crowded space for an extended period of time while avoiding collisions [27]. One important limitation that was identified during the experiment was that "the A* planner used does not create feasible paths for non-circular non-holonomic robots." This limitation is planned to be addressed in a future extension of the software.

As a separate project, a port of the Timed Elastic Band (TEB) local planner plugin is in progress.

Table 6 contains packages that support navigation in ROS2.

Table 6. Navigation packages.

Package	Description	Eloquent Status	Repo
navigation2	Global and local planner for navigation	Build from source	[28]
teb_local_planner	Timed Elastic Band Local Planner	Build from source	[29]

2.3.6 Perception

Perception involves the extraction of information about an environment from raw sensor data.

Table 7 contains packages that support perception in ROS2.

Table 7. Perception packages.

Package	Description	Eloquent Status	Repo
image_pipeline	Camera calibration, distortion removal, stereo, depth	Build from source	[30]
perception_pcl	Data structures and algorithms for working with point clouds	Debian exists	[31]
vision_opencv	Data structures and algorithms for computer vision	Debian exists	[32]
tensorflow	ROS nodes for using tensorflow machine learning	Python scripts in repo	[33]
object_analytics	Object tracking and 3D localization	Build from source	[34]

2.3.7 Manipulation

Manipulation involves the planning of collision-free paths to place an articulated arm or end effector in a particular pose or to grasp an object.

Table 8 contains packages that support manipulation in ROS2.

Table 8. Manipulation packages.

Package	Description	Eloquent Status	Repo
moveit2	Joint motion planning	Build from source, beta version	[35]
grasp	Grasp detection and planning	Build from source	[36]

2.4 Drivers

The ROS1 community has contributed a large number of packages to support various hardware devices used in robots. For example, according to [37] there are well over 100 different supported sensors. A full survey of ROS1 drivers and whether or not they have been ported to ROS2 is beyond the

scope of this report. Instead, focus is placed on major hardware categories and specific devices that are most likely to be used in ROS-M projects.

2.4.1 CAN

Controller Area Network (CAN) is a robust communication bus technology used for motor control and automotive drive-by-wire (DBW) systems.

Table 9 contains packages that support the CAN bus in ROS2.

Table 9. CAN packages.

Package	Description	Eloquent Status	Repo
ros_canopen	speak to devices using the CANopen protocol	Build from source	[38]

Note that Dataspeed Inc. provides ROS DBW support for the Lincoln MKZ as well as (Fiat Chrysler) FCA platforms at [39]. This support does not appear to be ported to ROS2 at this time.

2.4.2 GPS

GPS allows a robot to determine its global position relative to an earth frame.

Table 10 contains packages that support GPS devices in ROS2.

Table 10. GPS packages.

Package	Description	Eloquent Status	Repo
novatel_gps_driver	support for NavAtel GPS / GNSS receivers	Build from source	[40]
gps_tools	convert raw GPS data into ROS odometry	Build from source	[41]

Drivers for GPS devices from Microstrain (GX4/GX5) [42] and ublox [43] have not yet been ported to ROS2.

The gps_tools package is ported from the ROS1 gps_common package.

2.4.3 IMU

Inertial Measurement Units (IMU) measure acceleration and orientation of the device. They typically also output a gravity vector and a magnetic compass heading.

Table 11 contains IMU device drivers that were supported in ROS1 and have been ported to ROS2.

Table 11. IMU packages supported in ROS2.

Device	Eloquent Status	Repo
MicroStrain 3DM-GX2	Build from source	[44]
PhidgetSpatial 3/3/3	Debian exists	[45]

ROS1 includes drivers for the MicroStrain GX4/GX5 [42], Bosch BNO055 [46], and the DSP-3000 from KVH [47]. These drivers have not yet been ported to ROS2.

ROS1 also includes some IMU support packages like *imu_filter_madgwick* and *imu_complementary_filter* that do not appear to have been ported to ROS2. These packages fuse angular velocity and linear acceleration data and produce a device orientation as an output. Most IMU devices support some level of on-board sensor fusion so this functionality may not be required in many applications.

2.4.4 LiDAR

Light Detection and Ranging (LiDAR) is a range-finding technique based on illuminating a target with a laser and measuring the reflected light. LiDAR driver nodes publish scan data as either a LaserScan or a PointCloud2 message.

Table 12 contains LiDAR device manufacturers that have some level of ROS2 support.

Table 12. LiDAR support in ROS2.

Device	Eloquent Status	Repo
Velodyne	Build from source	[48]
Hokuyo	Build from source	[49]
SICK	Build from source	[50]
Ouster	Debian exists	[51]

2.4.5 Cameras

Cameras are sensors that capture image data in both the visible and non-visible light spectrums. In some cases, the extrinsic (e.g. position and orientation relative to the robot) and intrinsic (e.g. lens distortion) properties of the cameras are important for making sense of the image data. For example, stereo cameras compare the images from the two different cameras that are offset in space.

Table 13 contains packages supporting camera calibration in ROS2.

Table 13. Camera calibration packages.

Package	Description	Eloquent Status	Repo
image_common	calibration management and image transport	Debian exists	[52]

Table 14 contains stereo camera device manufacturers that have some level of ROS2 support.

Table 14. Stereo camera support in ROS2.

Device	Eloquent Status	Repo
Intel RealSense	Build from source	[53]
StereoLabs Zed	Build from source	[54]

Support for popular GigE Vision monocular cameras (e.g. AVT Prosilica, FLIR Blackfly, Basler Ace) has not yet been ported to ROS2.

2.5 Tools

2.5.1 Command Line

Core ROS1 command line tools such as roslaunch, rosrun, rostopic, etc. have been consolidated under a single ROS2 command line tool that supports the following subcommand functions:

- Run Launch Files
- Run Executables
- List and Describe Nodes
- List, Describe, Set, and Get Parameters
- Publish, Subscribe, and Describe Topics
- Execute and Describe Services
- Play and Record Bags

A very nice "cheat sheet" for the ROS2 CLI has been contributed by Ubuntu [55].

2.5.2 RQt

ROS1 includes RQt, a comprehensive set of tools based on PyQt, that provide GUI support for interaction, introspection, monitoring, data collection and visualization.

There are separate repositories for the RQt framework and available RQt plugins that can be found at [56].

Table 15 contains a description of the RQt plugins that have so far been ported to ROS2.

Table 16 contains a description of useful, non-experimental RQt plugins from ROS1 that have not been ported to ROS2.

Table 15. RQt plugins in ROS2.

Plugin	Description	Notes
Action Type Browser	Describe known action types	
Dynamic Reconfigure	Set parameters at runtime	
Node Graph	Display nodes and communication paths	
Process Monitor	Display list of running nodes	
Console	Display and filter log messages	Only displays default node loggers and filtering by node does not work
Python Console	Provides a panel to access a Python shell	
Shell	Provides a panel to access an operating system shell	
Robot Steering	Simple widgets for commanding velocity	
Service Caller	Invoke service calls	
Service Type Browser	Describe known service types	
Message Publisher	Write messages to topics	
Message Type Browser	Describe known message types	
Topic Monitor	List topics and their current values	
Image View	Render an image topic	
Plot	Plot data being published to topics	

Table 16. RQt plugins not in ROS2.

Plugin	Description
Package Graph	Displays graphs of ROS package dependencies
Bag	Record, play, and inspect bag files
Web	Provides a panel to access a web URL
Diagnostics Viewer	Display raw diagnostics
Logger Levels	Dynamically set verbosity on a per logger basis
Runtime Monitor	Display current, categorized diagnostic status
Navigation Viewer	Display maps and plans
Pose View	Visualize the orientation described by a pose topic
TF Tree	Visualize the transform tree of a robot

2.5.3 RViz

RViz is a GUI tool for visualizing data that is published by ROS nodes. Some examples of display types handled by RViz are:

- Robot Model
- TF
- Axes
- Point Cloud
- Odometry

Almost all of the default display plugins are supported by RViz in ROS2 Eloquent. Only DepthCloud and Effort, which do not seem to be widely used, are missing [57].

2.5.4 Gazebo

Gazebo is the standard physics simulation environment and visualizer for ROS1. Work was completed over the summer of 2019 to port the bulk of Gazebo to ROS2 [58]. A complete list of Gazebo plugins and their ROS2 migration status is available at [59].

Ignition Gazebo is an updated and restructured version of the classic Gazebo software with improved performance and rendering capabilities. Although Ignition Gazebo uses its own middleware layer, called Ignition Transport, for inter-process communication, a ROS2-to-Ignition Transport bridge is available at [60].

3 Migration

This section gives a brief summary of existing and planned ROS1 to ROS2 migration techniques.

3.1 Porting from ROS1 to ROS2

To illustrate the type of effort needed to port code from ROS1 to ROS2, the following is a list of some of the major tasks to be undertaken during the conversion process:

- Convert CMakeLists.txt file to use ament_cmake [61]
- Convert package.xml file from format 1 to at least format 2 [62]
- Convert time and duration types in message definition files to the new versions of the types defined by the builtin_interfaces package
- Convert topic, service, and action code to use new API
- Convert usages of ros::Time to std::chrono
- Convert nodelets to components
- Convert dynamic reconfiguration code to use parameter change callbacks
- Convert any shared, global parameters to local parameters (see section 2.2.5)
- Switch from Boost to standard C++ library when possible

For a more detailed conversion guide, see [63].

3.2 Techniques

This section describes common techniques for facilitating the migration of existing ROS1 systems to ROS2.

3.2.1 ROS1 API Shim

Over the past few years, the possibility of a ROS1 to ROS2 API shim layer has been discussed and some prototyping has been undertaken. This layer would allow ROS1 code to be run directly on top of a ROS2 infrastructure. This would theoretically allow ROS1 code to immediately benefit from some advantages of ROS2 like the DDS middleware and eliminate the need to run the ROS1 and ROS2 infrastructure side-by-side during a piecewise migration. ROS1 packages would then be gradually migrated to be native ROS2 and eventually the shim layer would no longer be needed. As of today, there is no known viable implementation of this shim layer.

3.2.2 Automated Tools

There has been some discussion around creating tools to automate the conversion process. An active project with the goal of providing automatic conversion tools is at [64].

3.2.3 ROS1 Bridge

The ROS1 Bridge is the only currently implemented technique for doing a piecewise migration to ROS2. The bridge essentially translates the ROS1 messaging protocol to DDS and vice versa. Some of the drawbacks to using a bridge are:

- bridge is a single point of failure
- extra CPU load due to message translation
- increased message passing latency

In a performance study executed in 2016 it was found that ROS2 bridge latency was around half a millisecond [7]. Since the ROS1 bridge figures to be an important part of the ROS2 migration process, it would be useful to execute a new study that characterizes latency, throughput, and relative CPU load.

4 Training

A relatively comprehensive set of textbooks, tutorials, references, and course material exist for ROS1. ROS2 currently lags behind ROS1 on nearly all of these fronts, mostly due to the rapid development pace of ROS2.

There are number of introductory texts for ROS1 such as [65], [66], [67], and [68]. There is currently no equivalent book coverage for ROS2.

Based on release notes and commit history, the amount of ROS2 demos and tutorials are increasing over time. They can be found at [69]. ROS2 reference material, including API documentation, is available at [70].

There are online introductory courses available for ROS2 at [71] and [72]. The content is geared toward beginners and only covers the basics of the build system, nodes, topics, and services.

5 Conclusion

There are still areas of concern related to the adoption of and migration to ROS2 for ROS-M projects. These concerns fall into the following major categories:

- Feature Gaps
- Quality
- Compliance
- Training

5.1 Feature Gaps

There are still areas where feature gaps exist with respect to ROS1. Some of the gaps observed during the survey of Eloquent include:

- generalized mechanism for setting level of individual loggers at runtime
- bridging ROS1 actions to ROS2 actions
- remote launch
- missing control algorithms (e.g. differential drive, ackermann, see section 2.2.10)
- missing or incomplete drivers (e.g. IMU, GPS, see section 2.3.7)
- missing or incomplete RQt plugins (e.g. console, logger levels, bag, diagnostics, TF tree, see section 2.5.1)

Note that there is already an experimental implementation of ROS action bridging at [73].

5.2 Quality

Feature set parity is a necessary but insufficient criteria for successful adoption of ROS2. Software quality is also a major concern. Much of the ROS2 software is relatively new when compared to ROS1. It will take a significant amount of testing and bug fixing by the ROS community to bring ROS2 to an acceptable and trusted level of quality. Using a standard DDS middleware (with the option of licensing a field-proven implementation like RTI Connext) helps in this regard, but the middleware is only a small, albeit core, subset of the overall software. When it comes to open source software, quality typically comes through real-world use that leads to the reporting and fixing of bugs.

5.3 Compliance

By virtue of adopting DDS as its middleware layer, ROS2 realizes improvements over ROS1 in terms of meeting certain security and reliability goals of ROS-M. However, DDS is not a "silver bullet" in this regard. Not all DDS implementations are created equal and will vary in terms of performance and compliance to the DDS specification. Projects should be careful to choose the DDS implementation that meets their specific objectives.

Similarly, future projects may require real-time determinism in order to meet reliability and safety requirements. Enabling DDS quality of service features is not enough. An operating system with real-time capabilities (e.g. priority-based preemption) and tuning (e.g. eliminating page faults) needs to be used. Also, the ROS2 framework and the application software that runs on it must be carefully designed such that time critical functions are guaranteed to complete before their deadlines. More study is required in this area. See [74] for an initial discussion of real-time support in ROS2 and [75] for current status of the ROS2 Real-time Working Group as of February 2020.

5.4 Training

Training is another aspect that could delay migration to ROS2. Current course offerings are not sufficient for an intensive "boot camp" style training. A more comprehensive training would require more coverage of advanced topics like constructing launch files and effectively using Rviz/Gazebo. Also, hands-on lab training involving more advanced features like Navigation 2

and some of the perception packages could go a long way to showing developers how ROS2 can be used to solve real world problems in robotics.

5.5 Recommendations

The survey of ROS2 Eloquent shows that the core ROS2 infrastructure is approaching parity with ROS1. The Foxy release in May 2020 will be a long-term support (LTS) release and there is discussion in the ROS2 TSC to "back off to annual releases" starting with that release [76]. This seems to indicate that there is a feeling that Foxy will represent what could be considered a reasonably complete core feature set for ROS2. Thus, it appears Foxy may be the point at which ROS2 could be seriously considered for projects that implement their own feature packages and only require the core ROS2 infrastructure. A follow-on analysis of Foxy should be performed when it is available to determine if this is the case.

The survey also shows that while many projects related to drivers and algorithms are currently in progress, this development lags significantly behind the core infrastructure. More experience with community provided packages in these areas is needed before it is possible to determine which specific packages are at a level where they could be combined to form the foundation of a working UGV system.

In order to fill gaps in ROS2, these follow-on efforts are proposed:

- Help port the packages identified as feature gaps
- Execute an updated middleware performance study that includes all DDS implementations integrated by ROS2 and compares and contrasts throughput, latency, and relative CPU load of ROS1, ROS2, and ROS1-ROS2 bridged communication.
- Execute an updated DDS security analysis with respect to DoD information assurance and safety guidelines (e.g. DoDI 8500.2, AR 25-2, and MIL-STD-882-E).
- Create a document that provides guidance related to configuring security on a ROS2 system.
- Closely monitor the ongoing discussions in the ROS community regarding real-time support, participate in ROS2 real-time working group meet-

ings, and provide requirements and feedback as appropriate. See section 5.2.

- Execute a targeted set of projects to migrate existing ROS1 code bases to ROS2. The lessons learned from these projects will provide valuable guidance for future projects. One output of these projects would be a migration guide that provides detailed porting instructions and assistance in determining level of effort for a particular porting project.
- Until a viable alternative is identified for 3D SLAM, help support the maintenance of Cartographer for ROS2.
- Identify and support projects that use drivers and algorithm packages supplied by the ROS community with the goal of evaluating the suitability of those packages for ROS-M projects.

References

[1] Improvements to rmw for deterministic execution. January 2020. https://github.com/ros2/design/issues/259.

[2] ROS 2 Technical Steering Committee Meeting Minutes. January 2020. https://discourse.ros.org/t/ros-2-tsc-meeting-minutes-2020-01-16/12382.

[3] ROS 2 Technical Steering Committee Meeting Minutes. March 2020. https://discourse.ros.org/t/ros-2-tsc-meeting-minutes-2020-03-18/13313.

[4] Fast-RTPS vs Cyclone DDS vs OpenSplice DDS, December 2019. https://www.eprosima.com/index.php/resources-all/performance/125-fast-rtps-vs-cyclone-dds-vs-opensplice-dds.

[5] APEX.AI Adopts eProsima Fast RTPS for its Autonomous Driving Framework. February 2019. https://www.eprosima.com/index.php/company-all/news/103-apex-ai-adopts-eprosima-fast-rtps-for-its-autonomous-driving.

[6] ROS2 - Is it time to switch? December 2019. https://blog.roverrobotics.com/ros-2-is-it-time-to-switch-tutorial-included.

[7] Y. Maruyama, S. Kato, and T. Azumi. Exploring the performance of ros2. October 2016.

[8] J. Kim, J. Smereka, C. Cheung, S. Nepal, and M. Grobler. Security and performance considerations in ros 2: A balancing act. September 2018.

[9] Composing multiple nodes in a single process. https://index.ros.org/doc/ros2/Tutorials/Composition.

[10] Migrating launch files from ROS1 to ROS2. https://index.ros.org/doc/ros2/Tutorials/Launch-files-migration-guide/.

[11] Support for respawn in 'ExecuteProcess'. https://github.com/ros2/launch/issues/287.

[12] Parameter Blackboard Demo. https://github.com/ros2/demos/blob/master/demo_nodes_cpp/src/parameters/parameter_blackboard.cpp.

[13] Principle of Least Privilege. https://en.wikipedia.org/wiki/Principle_of_least_privilege.

[14] ROS2 "Global Parameter Server" Status. https://discourse.ros.org/t/ros2-global-parameter-server-status/10114.

[15] ROS2 Plugin Repository. https://github.com/ros/pluginlib/tree/eloquent.

[16] ROS2 Geometry2 Repository. https://github.com/ros2/geometry2.

[17] ROS2 Bond Repository. https://github.com/ros/bond_core/tree/ros2.

[18] ROS2 Diagnostic Updater Repository. https://github.com/ros/diagnostics/tree/eloquent/diagnostic_updater.

[19] ROS1 Controls Repository. https://github.com/ros-controls/ros_control.

[20] ROS1 Controllers Repository. https://github.com/ros-controls/ros_controllers.

[21] ROS2 Controls Repository. https://github.com/ros-controls/ros2_control/tree/dashing_update.

[22] ROS2 Controller Repository. https://github.com/ros-controls/ros2_controllers/tree/dashing_update.

[23] ROS2 Robot Localization Repository. https://github.com/cra-ros-pkg/robot_localization/tree/dashing-devel.

[24] ROS2 SLAM Toolbox Repository. https://github.com/SteveMacenski/slam_toolbox/tree/eloquent-devel.

[25] ROS2 Cartographer Repository. https://github.com/ros2/cartographer.

[26] Default SLAM(s) For ROS2. January 2020. https://github.com/ros-planning/navigation2/issues/1389.

[27] S. Macenski, R. White, J. Clavero, and F. Martin. The marathon 2: A navigation system. March 2020.

[28] ROS2 Navigation 2 Repository. https://github.com/ros-planning/navigation2/tree/eloquent-devel.

[29] ROS2 Timed Elastic Band Local Planner Repository. https://github.com/rst-tu-dortmund/teb_local_planner/tree/eloquent-devel.

[30] ROS2 Image Pipeline Repository. https://github.com/ros-perception/image_pipeline/tree/ros2.

[31] ROS2 Point Cloud Library Repository. https://github.com/ros-perception/perception_pcl/tree/dashing-devel.

[32] ROS2 OpenCV Machine Vision Repository. https://github.com/ros-perception/vision_opencv/tree/ros2.

[33] ROS2 TensorFlow Machine Learning Repository. https://github.com/alsora/ros2-tensorflow.

[34] ROS2 Object Analytics Repository. https://github.com/intel/ros2_object_analytics.

[35] ROS2 MoveIt! 2 Repository. https://github.com/ros-planning/moveit2.

[36] ROS2 Grasp Repository. https://github.com/intel/ros2_grasp_library.

[37] ROS Sensors Wiki. December 2019. http://wiki.ros.org/Sensors.

[38] ROS2 CANopen Repository. https://github.com/ros-industrial/ros_canopen/tree/dashing.

[39] Dataspeed Inc DBW Repository. https://bitbucket.org/DataspeedInc/profile/repositories.

[40] ROS2 NovAtel GPS Driver Repository. https://github.com/swri-robotics/novatel_gps_driver/tree/dashing-devel.

[41] ROS2 GPS Tools Repository. https://github.com/swri-robotics/gps_umd/tree/dashing-devel/gps_tools.

[42] ROS1 Microstrain GX4/GX5 Driver Repository. https://github.com/ros-drivers/microstrain_mips.

[43] ROS1 ublox Driver Repository. https://github.com/KumarRobotics/ublox.

[44] ROS2 Microstrain GX2 IMU Driver Repository. https://github.com/ros-drivers/microstrain_3dmgx2_imu/tree/dashing-devel.

[45] ROS2 Phidget IMU Driver Repository. https://github.com/ros-drivers/phidgets_drivers/tree/dashing.

[46] ROS1 Bosch IMU Driver Repository. https://github.com/dheera/ros-imu-bno055.

[47] ROS1 KVH IMU Driver Repository. https://github.com/ros-drivers/kvh_drivers.

[48] ROS2 Velodyne Driver Repository. https://github.com/ros-drivers/velodyne/tree/dashing-devel.

[49] ROS2 Hokuyo Driver Repository. https://github.com/bponsler/urg_node/tree/ros2-devel.

[50] ROS2 SICK Driver Repository. https://github.com/SICKAG/sick_scan2.

[51] ROS2 Ouster Driver Repository. https://https://github.com/SteveMacenski/ros2_ouster_drivers.

[52] ROS2 Camera Calibration and Image Transport Repository. https://github.com/ros-perception/image_common/tree/dashing.

[53] ROS2 Intel RealSense Camera Driver Repository. https://github.com/intel/ros2_intel_realsense.

[54] ROS2 StereoLabs Zed Camera Driver Repository. https://github.com/stereolabs/zed-ros2-wrapper.

[55] ROS2 CLI Cheat Sheet. https://github.com/ubuntu-robotics/ros2_cheats_sheet.

[56] ROS2 Visualization Tools. https://github.com/ros-visualization.

[57] ROS2 RViz README. November 2019. https://github.com/ros2/rviz/blob/ros2/README.md.

[58] Migrating Gazebo plugins to ROS2. August 2019. https://discourse.ros.org/t/migrating-gazebo-plugins-to-ros2/10433.

[59] ROS2 Gazebo Plugins. https://github.com/ros-simulation/gazebo_ros_pkgs/wiki.

[60] ROS2 Ignition Bridge Repository. https://github.com/osrf/ros2_ign.

[61] ament cmake User Documentation. https://index.ros.org/doc/ros2/Tutorials/Ament-CMake-Documentation.

[62] Package Manifest Format Two Specification. https://www.ros.org/reps/rep-0140.html.

[63] ROS2 Conversion Guide. https://index.ros.org/doc/ros2/Contributing/Migration-Guide.

[64] ROS2 Automated Conversion. https://github.com/awslabs/ros2-migration-tools.

[65] M. Quigley. *Programming Robots with ROS*. O'Reilly, 2015.

[66] W. Newman. *A Systematic Approach to Learning Robot Programming with ROS*. CRC Press, 2018.

[67] C. Fairchild. *ROS Robotics By Example*. Packt Publishing, 2017.

[68] L. Joseph. *Mastering ROS for Robotics Programming*. Packt Publishing, 2018.

[69] ROS2 Tutorials. https://index.ros.org/doc/ros2/Tutorials.

[70] ROS2 Reference Guide. http://docs.ros2.org.

[71] ROS2 How To: Discover Next Generation ROS. https://www.udemy.com/course/ros2-how-to.

[72] ROS2 Basics in 5 Days. https://www.theconstructsim.com/robotigniteacademy_learnros/ros-courses-library/ros2-basics-course.

[73] ROS2 Bridging of ROS1 Actions Repository. https://github.com/ipa-hsd/action_bridge.

[74] ROS2 and Real-time. https://discourse.ros.org/t/ros-2-and-real-time/8796.

[75] ROS2 Real-time Working Group Online Meeting 10. https://discourse.ros.org/t/ros-2-real-time-working-group-online-meeting-10-feb-5-2020-meeting-minutes/12809.

[76] ROS2 Technical Steering Committee Meeting Minutes. December 2019. https://discourse.ros.org/t/ros-2-tsc-meeting-minutes-2019-12-19/12069.