Electronic Design

By NEIL PUTHUFF, Senior Automotive Applications Engineer, RTI

ROS and DDS: Making the Most Out of Your Software Framework

ROS 2 has become popular beyond robotics, but it has significant technical limitations. Learn how to break past these limits using the powerful framework on which ROS 2 is built.

The introduction and ongoing releases of the robot operating system, ROS 2 (now built on top of the DDS framework), has expanded its use beyond its original focus on robotic research. ROS 2 comes bundled with application packages and visualization tools, so it facilitates making robotics systems that can sense, map, and navigate their surroundings.

About DDS

Data Distribution Service (DDS) is an open-standard, data-centric communications software framework with more than a dozen commercial and open-source implementations. It provides low latency, extreme reliability, and a rich set of Quality of Service (QoS) controls to enable robust peer-to-peer communications in the most challenging of environments: contested battlefields, noisy industrial settings, wide-area networks, and remote systems with intermittent connectivity.

DDS has been used in thousands of critical systems, hundreds of autonomous-vehicle programs, and dozens of other frameworks and standards including ROS 2, AUTOSAR, and FACE.

About ROS

The free, open-source ROS project is a one-stop shop for quickly creating robotics applications and systems. First released in 2010, the original ROS rapidly became popular in academia, eventually turning into the dominant framework for robotics researchers and educators.

The main value in ROS is in its tools and pre-built packages. Its main disadvantage was the middleware, which prevented it from being used in critical or constrained systems, in multi-robot swarms, or in applications with real-time constraints. As a result, ROS remained largely in academia for more than a decade.

ROS + DDS = ROS 2

ROS 2 is a redesign of the original ROS that should help solve emerging challenges in robotics. Built on top of the DDS framework, ROS 2 seeks to operate in constrained systems, multi-robot swarms, and production-grade platforms—an ideal marriage that combines the outstanding tools and packages of ROS with the "works everywhere" capabilities of DDS. ROS 2 has helped ROS break out of academia.

The success of ROS 2 has led to the retirement of the original ROS following the 2020 "Noetic" release. All future ROS development is now on ROS 2, and all communications within ROS 2 flow via DDS. So, in effect, all ROS 2 applications are also DDS applications, and the ROS 2 ecosystem is a part of the DDS ecosystem. Read on to learn why this is an important distinction.

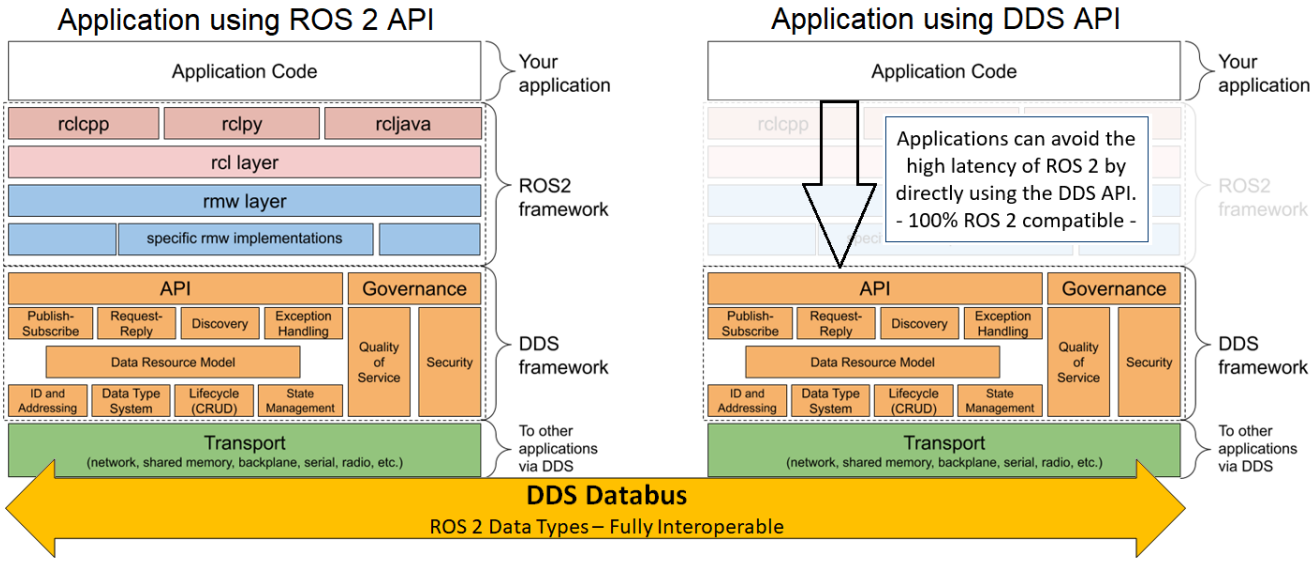
Is ROS 2 the Sum of its Parts?

Not quite. ROS 2 is a big improvement over the original version of ROS, but when ROS 2 is viewed from the field-proven perspective of DDS, it has some significant limitations.

However, those limitations can be overcome by directly accessing the DDS framework upon which ROS 2 is built, enabling system developers to get the full benefit of DDS while maintaining compatibility with ROS 2. Let's examine these limitations and how they can be resolved using this approach.

Performance

Because ROS 2 is layered on top of DDS, any data sent or received within ROS 2 must travel through these layers



1. Applications that directly use the DDS APIs beneath ROS 2 eliminate the data latency of the ROS 2 framework, while maintaining full ROS 2 compatibility.

before reaching the underlying DDS framework. This takes a substantial amount of time.

By designing critical application components to directly use the underlying DDS API, engineers can eliminate many performance bottlenecks. For example, during the 2021 Indy Autonomous Challenge, a ROS 2 LiDAR device driver was modified to directly use the DDS API. This cut latency by up to 90% compared with a driver that used ROS API but didn't affect full interoperability with ROS 2 *(Fig. 1)*.

This approach was employed to dramatic effect by the winning team of the Indy Autonomous Challenge, a $1.5M competition to autonomously race full-size Dallara IL-15 vehicles at the famous Indianapolis Motor Speedway. Not only did the approach eliminate the bulk of data latency, it also freed up two-thirds of the memory used by the ROS 2 driver, while maintaining drop-in compatibility with ROS 2.

Scalability

ROS 2 systems can quickly run into scalability issues because they create large numbers of data topics (unique, discoverable data flows) to support the implementation of ROS Parameters, Services, and Actions in ROS 2.

In operation, every ROS 2 application node creates more than a dozen unique topics for Parameters (even if you don't use them), and more unique topics are created for every ROS 2 Service, Message, and Action used in your applications. Consequently, it doesn't take long for a system to have hundreds or thousands of topics competing for space on the network.

An equivalent DDS system can avoid this overhead. First, common data types may be implemented using Keyed Topics, which enables large numbers of unique data sources to share common data flows while retaining their unique identity, easing congestion in large-scale systems.

Second, DDS can be used to create topic gateways to partition your large ROS 2 system into a tiered hierarchy, permitting only the necessary data to flow to other parts of your system. This comes in handy in situations such as creating a system of many mobile robots connected over a common wireless network.

Third, any applications implemented directly on DDS will not create the dozen-or-more topics created by ROS 2 for its parameter system, which directly reduces the number of topics on the network. The result is a system with reduced network traffic, faster startup, and greater scalability than the equivalent ROS 2 implementation.

Quality of Service (QoS)

QoS plays the primary role in assuring dependable system communications under real-world conditions. While DDS itself has a comprehensive set of QoS capabilities, the majority aren't accessible from within ROS 2. As a comparison:

ROS 2 has support for eight QoS categories, with basic on/off and sizing control for History, Depth, Reliability, Durability, Deadline, Lifespan, Liveliness, and Lease Duration.

OMG Standard DDS has 22 QoS categories. Here's a few highlights of what's not in ROS 2:

- *Entities and Keys:* These enable vast scale-up by sharing common data topics using unique "Keys" for each data source. This makes the system start faster and run more ef-

ROS 2 / C++

```cpp
#include <chrono>
#include <functional>
#include <memory>
#include <string>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

class MinimalPublisher : public rclcpp::Node
{
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
    timer_ = this->create_wall_timer(
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }

private:
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = "Hello, world! " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};

int main(int argc, char* argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

DDS / C++

```cpp
#include <iostream>
#include <dds/pub/ddspub.hpp>
#include <rti/util/util.hpp> // for sleep()
#include "string.hpp"

void publisher_main(int domain_id)
{
  dds::domain::DomainParticipant participant (domain_id);
  dds::topic::Topic<std_msgs::msg::dds_::String_> topic (participant, "rt/topic");
  dds::pub::DataWriter<std_msgs::msg::dds_::String_> writer(dds::pub::Publisher(participant), topic);
  std_msgs::msg::dds_::String_ sample;
  for (int count = 0; ; count++) {
    sample.data_(std::string("Hello, world! " + std::to_string(count)));
    std::cout << "Publishing " << sample.data_() << std::endl;
    writer.write(sample);
    rti::util::sleep(dds::core::Duration(0, 500000000));
  }
}

int main(int argc, char *argv[])
{
  try {
    publisher_main(0);
  } catch (const std::exception& ex) {
    std::cerr << "Exception in publisher_main(): " << ex.what() << std::endl;
    return -1;
  }
  return 0;
}
```

2. Here's a comparison of C++ source code for a "Hello, World!" application written in ROS 2 vs. code written directly in the underlying DDS (RTI Connext) used by ROS 2. (Source: Real-Time Innovations)

ficiently, and it reduces the load on the network.

• *Instance Management:* This allows applications to manage open-ended collections of objects (e.g., robot swarms) where all objects in the collection publish the same Topics but they're distinguished by an Instance Key (typically the identifier of the object in the collection, e.g., a robot_id). Subscribers subscribe to the "collection" topic and are notified as new objects join or leave the collection or if any object ceases to be active. The data published by each object in the collection is maintained in a separate sub-stream (identified by the Instance Key) and the QoS is applied to each sub-stream separately.

• *Data Cache:* This allows the application to access the data cache of the subscriber (DataReader in DDS terms) directly. The cache provides zero-copy views of the data received synchronously or asynchronously with data reception. The data can be accessed multiple times (e.g., to perform aggregation or sensor fusion) without requiring the application to keep extra copies.

• *Ownership and Strength:* These provide for automatic failover when using redundant components. This also enables zero-downtime during system upgrades, maintenance, and testing.

• *Transport Priority:* This allows a system to give preference to a transport when it's available. For example, a mobile system using cellular communications can be made to automatically switch over to Wi-Fi when available.

• *Partitions:* These enable a network to be divided into isolated logical sections. Data flows within these sections can't interact with outside elements, thus providing system-scale encapsulation of related areas.

Vendor-Extended DDS offers even more capabilities to meet real-world networking challenges. For example, RTI Connext DDS has options for wide-area-network (WAN) connectivity, small sample batching, content filtering and querying, compression and bandwidth reduction, time-sensitive networking (TSN), and more.

Each DDS implementation can choose where to focus their efforts in these QoS categories, from simple on/off controls to fully tunable parameters, or to omit the category entirely. These will have a direct impact on whether your system can run in a particular environment or not. Simply put, more QoS means more adaptability. A capable DDS implementation will be able to operate in countless environments where ROS 2 cannot.

Interoperating with Non-ROS Systems

As DDS is used in thousands of critical systems and in standards such as AUTOSAR and FACE, a ROS 2 system also may need to interoperate with these non-ROS environments based on DDS.

While DDS provides standards-based interoperability, the implementation within ROS 2 imposes a set of rules and restrictions on the names of data types and topics that it will accept. Any data that falls outside of these rules is ignored or unusable. This has a direct impact on ROS 2's ability to interoperate with non-ROS systems built on DDS, so a separate bridge application must be written to translate these non-conforming data types into something that ROS 2 can accept.

However, an application written to directly use the DDS API can easily interoperate with any DDS-based system, such as ROS 2, AUTOWARE, FACE, and many others. In this role, DDS can be thought of as something like a "Hypervisor for Distributed Systems," capable of directly integrating many different types of DDS-based applications (ROS 2, AUTOSAR, etc.) into a high-performance system.

Getting the Best of DDS and ROS 2

DDS maintains a very stable, standards-based software API, and ROS 2 has a very stable data model and is itself based on DDS. However, ROS has a history of introducing changes to its software API with each major release, which creates an ongoing burden for developers who must update and version-stamp their applications to keep pace with ROS.

There's a way to get the full benefits of DDS while retaining the benefits of ROS 2 and eliminating its drawbacks: Implementing critical components directly in DDS using the ROS 2 data types.

This hybrid combination has some compelling benefits:

- Significantly improved performance. Eliminating the ROS 2 software layers can eliminate more than 90% of the latency of data communications. This approach also enables full access to advanced capabilities of DDS implementations that aren't supported in ROS 2, such as zero-copy transfers, reduced encoding, compression, etc.
- Scalability unbounded. By using keyed topics and DDS-enabled hierarchical design, your system can scale to massive proportions, as was done with the Constellation control system at the Kennedy Space Center (the largest SCADA system in the world).
- Access to all DDS QoS. Quality of Service enables your systems to work in the most demanding environments, where ROS 2 cannot venture.
- Fully interoperable with ROS 2, and with non-ROS DDS systems.
- Supported on production-grade hardware and operating systems, and available in safety-certifiable versions.

While this might seem like a risky introduction of new technology to a ROS 2 development, it's actually the opposite. Developers using ROS 2 are already working with DDS; this change merely gives them a more direct connection to DDS. Let's now look at how that affects the software.

Examining the Software API

ROS and DDS are both data-centric, publish/subscribe technologies with very similar design patterns. Taking a closer look at the ROS 2 API shows that the bulk of the API functions cover data communications, which are provided under the hood by the DDS API. The remainder are mostly common system-level functions (files, timers, callbacks, etc.) that can be found in standard libraries.

Therefore, the API patterns of creating applications directly in DDS should feel very familiar to developers accustomed to ROS 2. But what about the source code? *Figure 2* illustrates a comparison of a simple "Hello World!" application written using the ROS 2 and DDS APIs.

Implementing systems directly in DDS can be a natural progression for developers using ROS 2. It follows similar design patterns but provides far greater control over system communications, while eliminating many layers of software that hinder debugging.

Implementing the Improvements

A typical pattern for this hybrid use case is to replace critical system components written in ROS 2 with their native DDS equivalents. Because of the standards-based interoperability of DDS, the replacements can be dropped in without disruption to the remainder of the ROS 2 system. An example might be a signal-processing module for camera or LiDAR that needs to operate at minimum latency, or a gateway application to communicate with many peer units over a radio transport.

The system developer can implement these improvements strategically, replacing only critical components while leaving the remainder of the system in ROS 2, or as part of an overall system transition to native DDS. Interoperability with the ROS 2 ecosystem can be maintained continuously.

Wrapping It All Up

ROS 2 is a popular way to quickly create robotics systems using distributed applications, but it may struggle with scalability, performance, and operation in challenging environments—all places where DDS excels.

Fortunately, ROS 2 is implemented on top of DDS, meaning that system developers can freely intermix ROS 2 and DDS applications to solve specific ROS 2 shortcomings. Or they can migrate their entire system to a higher performing, open-standard DDS option without losing interoperability with ROS 2 and its excellent ecosystem of tools and packages. Truly, the best of both ROS 2 and DDS is available now for builders of robotic and autonomous systems.

To learn more, visit https://community.rti.com/ros.

Neil Puthuff is a senior automotive applications engineer at Real-Time Innovations with a focus on automotive, ROS, and grid modernization. Prior to joining RTI, he created processor probes and replay debugging products at Green Hills Software. Neil is a named inventor on more than a dozen U.S. patents.

Captions: