The transition from ROS 1 to ROS 2

By Vincenzo DiLuoffo, William R. Michalson, Berk Sunar

Worcester Polytechnic Institute (WPI)
{vdiluoffo, sunar, wrm} @wpi.edu

Date:06/05/17

Table of Contents

Introduction....3
What is ROS 1....4
What is ROS 2....5
Considerations for the transition....7
Conclusion....7
References....8
Appendix A: Hello World C++ example....9

Table of Figures

Figure 1: NASA’s Valkyrie Robonaut 5 robot. (Courtesy of NASA)....4
Figure 2: ClearPath Turtlebot 2....4
Figure 3: Decomposed Turtlebot 2 Logic....4
Figure 4: ROS Messaging using topics....5
Figure 5: ROS 2 Software layers....5
Figure 6: Data Distributed Service (DDS) Software Model....6

Introduction

Since the early days of robots custom built software was developed to enable simple tasks and the evolution of robots has created the need for frameworks to support complex tasks. In recent years the pub-sub paradigm has been adopted by industry and the common functions of robots to be implemented in a way that makes them portable. The Robot Operating System (ROS) capitalizes on this paradigm shift and the desire for cross-platform multiprocessing. It also provides a framework for building robot software that is portable and extensible (albeit at a cost). ROS is an open source project maintained by the Open Source Robotic Foundation (OSRF) and has expanded into different areas like industrial, military, agriculture and automotive. Other proposals are underway for hardware base models to connect building blocks together at the physical level and be ROS compliant.

As of July 2016, ROS has over 14 million lines of code and 2,477 authors [1]. The number of robots using ROS is over 100, but others aren't publicly stating their use for ROS [2]. Robot categories range from classical industrial, to mobile and complex humanoid types. With the popularity, maturity and generalized usage of ROS, a new version is being introduced called ROS 2 with the same paradigm principles as ROS 1. As robots increase their capabilities and expand into different markets, the question of security needs to be addressed. One of the first papers related to robotics, security, and privacy experimented with toys [3]. As robots take on human roles or enhance the human capability, they pose a potential security risk on valued targets. One example of a valued target is surgical robots and potential risks [4]. ROS 2 attempts to address the security concerns with a security extension, a first to incorporate with robotic systems. This paper will cover the transition from ROS 1 to ROS 2 and the motivation to focus on security and performance.

What is ROS 1

ROS is used in many robotic platforms from NASA's Valkyrie a complex humanoid as shown in Figure 1[5] to research and experimental platforms like Turtlebot as shown in Figure 2 [6]. A number of sensors, controllers and motors must be able to communicate and send data from one to the other and receive data in order to perform tasks. A decomposed Turtlebot logic is shown in Figure 3. A sensor, the 3 D camera captures live streaming data which is sent to a navigation and visualization module to calculate the next step of where the robot must go. The locomotion module takes the instruction of where to go and converts that to mobile base commands (move here and how fast). In this example ROS is being utilized to send/receive the data for each of these modules.

Figure 1: NASA's Valkyrie Robonaut 5 robot. (Courtesy of NASA).

Figure 2: ClearPath Turtlebot 2

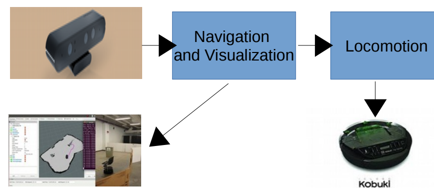
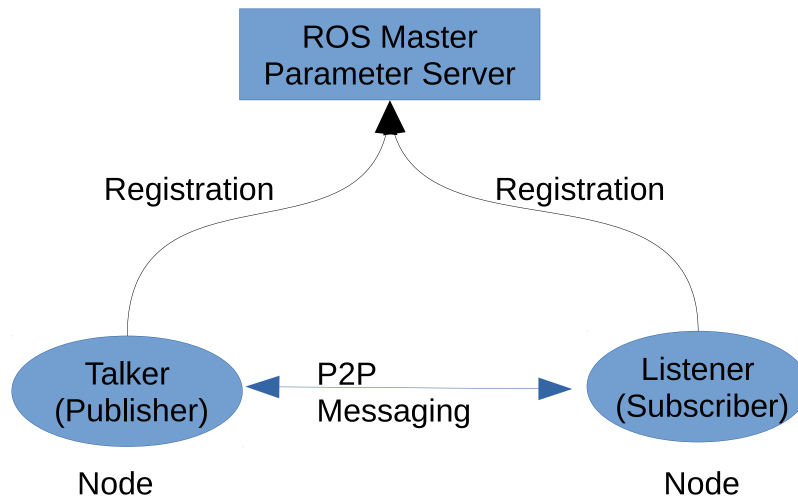


Figure 3: Decomposed Turtlebot 2 Logic

ROS 1 was developed as a distributed architecture using publisher/subscriber messaging between nodes. Each node can have a single/multi topics or services in this architecture that are decoupled and reusable. A topic is a name for content where a producer is a publisher and consumer is a subscriber. An example of a publisher is a 3D camera producing streamed image data and one that consumes that data is a subscriber. A service is a client/server model where a request/respond messaging is used. An example of a request is image data to be formatted into another format/compressed, the response will be converted/compressed data.

In Figure 4 is an example of a ROS messaging between two nodes using topics Talker as a publisher and Listener as a subscriber. The ROS Master sets up the topics to find each other, this is called a central message broker model where communications initialization are started and then each of the topics can communicate directly with each other. This simple example is the same principles as above in the Turtlebot 2 logic of how each of the modules need to send and receive data to perform tasks. The parameter server is a service run on the Master to support global value storage. An example of global values are often configuration parameters where they can be set or retrieved from the ROS network.

Figure 4: ROS Messaging using topics



What is ROS 2

ROS 2 was introduced in 2014, but was first available as alpha code in 3Q2015. ROS 2 has taken a different approach in its messaging layer and now employs the industry standard called Data Distributed Services (DDS), from the Object Management Group (OMG). A new DDS security specification extension was released later in 2016. The coupling of a realtime transport, Quality of Service (QoS) and security is being formulated for the robotic industry. ROS 2 is still in its early stages of development with Beta 1 released in Dec 2016 and being supported by RTI and eProsima vendors. Many more releases are expected before the first production release is available.

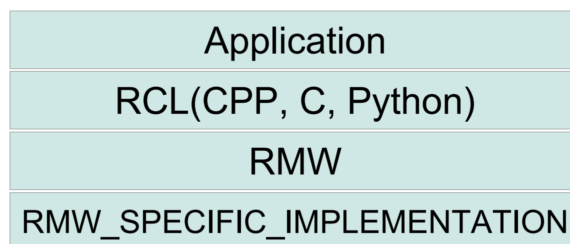
Since Beta 1 release the C++ is more mature than Python or C ROS clients as shown in Figure 5. The added abstraction layer ROS middleware interface provides the logic to convert ROS messages into Interface Definition Language (IDL) messages, so that DDS specific vendor implementations can be supported. DDS provides a number of benefits related to supported vendor products, lossy communication support using QoS profiles, the ability to scale, performance gains, realtime support and security.

Figure 5: ROS 2 Software layers

ROS Client

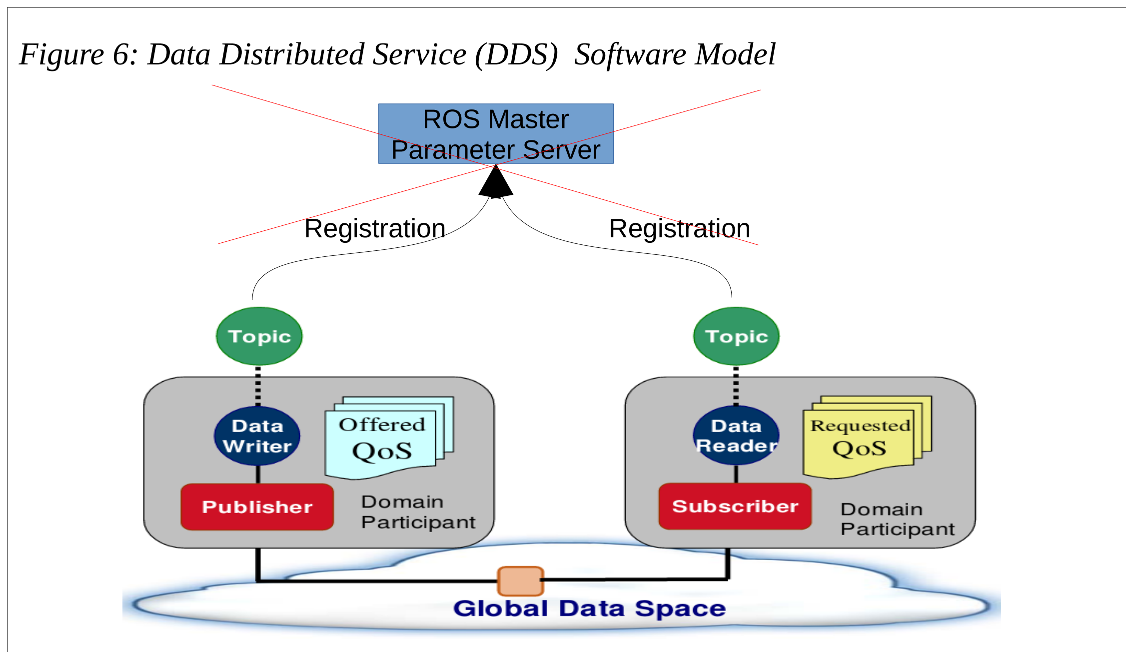
ROS Middleware Interface

DDS Implementation



In Figure 6 [7] a similar model to ROS 1, but no central broker model component. DDS supports a set of discovery services that allows publishers and subscribers to dynamically discover each other without the name server bottleneck. A domain participant allows an application to join the global data space and each topic is a string that address the objects in the same space. Each object is identified by a key where data writer and data reader are supported by pools of resources called publisher and subscriber. A data writer declares the intent to publish a topic and provides type safe operations to write or send data. A data reader declares the intent to subscribe to a topic and provides type safe operations to read or receive data. DDS uses a Real-Time Publish Subscribe (RTPS) protocol for its data transportation. The RTPS protocol is designed to be able to run over multicast and connectionless best-effort transports such as UDP/IP. The use of QoS profiles allows the RTPS communications layer to provide reliability in a lossy environment like wireless/cellular and support for realtime environments where critical processes are under time constraints to complete.

Figure 6: Data Distributed Service (DDS) Software Model



The new DDS Security extension provides ROS 2 with the capability to protect data in motion. DDS security extension defines two policies called Domain Governance and Participant. The first policy defines how the domain is controlled and respectively, the second policy defines what domains can be joined and control over reads and writes on topics. As part of the extension five plug-ins are defined authentication, access control, cryptographic library, logging and data tagging. Once the overall security polices are defined and each participant has their credentials, the system can run in a secure manner using the plug-ins.

Considerations for the transition

In order to migrate or start new in the ROS 2 environment a learning curve must be acquired. Since Beta 1 was released at the end of 2016, little documentation is available. As stated earlier the C++ client is more mature than Python and C client isn't available yet. Other client development on GitHub maybe happening, but not currently public. ROS 2 has a different build environment, more builtin functions are being utilized in C++ 11 and higher and support for Python 3 only. A migration guide is being updated with new content from conversion experiences [8].

The use of QoS profiles is being used with every publisher and subscriber, so this will need to be understood as to which type to use. Security is a new component for the ROS 2 environment and consideration will need to be taken in defining the policy, credentials and architecture. ROS 2 provides many advantages over ROS 1 where a mature DDS standard is being used for its underlining communications layer and a new security extension. A simple conversion example of the hello world is listed below for ROS 1[9] and ROS 2[10]. Please see Appendix A. Highlights are shown in the ROS 2 code for header file changes, replacement of boost functions with C++ builtins, different invoking instructions and ways to access data.

Conclusion

The motivation for moving from ROS 1 to ROS 2 has many benefits related to realtime, QoS and security being supported in the underlining DDS implementations. ROS has the capability to now shift from research to production environments with mature DDS products. The migration from ROS 1 to ROS 2 still perseveres the pub-sub paradigm. This paper has covered ROS 1, ROS 2, the difference in implementation and the transition from one to the other. The simple hello world example was provided to show the differences that need to be considered for migration or taking a first look at ROS 2 code structure. ROS 2 and DDS Security will still continue to have intermediate releases before final. As the robotics market continues to grow and expand its presences into different segments, security will be an area for future research.

References

[1] T. Foote, “Celebrating 9 Years of ROS, the Robot Operating System,” *IEEE Spectrum: Technology, Engineering, and Science News*, 09-Jan-2017. [Online]. Available: http://spectrum.ieee.org/automaton/robotics/robotics-software/celebrating-9-years-of-ros.

[2] K. Conley and T. Foote, “metrics-report-2016-07.pdf,” *http://wiki.ros.org/Metrics*. [Online]. Available: http://download.ros.org/downloads/metrics/metrics-report-2016-07.pdf.

[3] T. Denning, C. Matuszek, K. Koscher, J. R. Smith, and T. Kohno, “A spotlight on security and privacy risks with future household robots: attacks and lessons,” in *Proceedings of the 11th international conference on Ubiquitous computing*, 2009, pp. 105–114.

[4] T. Bonaci, J. Herron, T. Yusuf, J. Yan, T. Kohno, and H. J. Chizeck, “To make a robot secure: An experimental analysis of cyber security threats against teleoperated surgical robots,” *ArXiv Prepr. ArXiv150404339*, 2015.

[5] E. Kisliuk, “Valkyrie,” *NASA*, 23-Sep-2015. [Online]. Available: http://www.nasa.gov/feature/valkyrie.

[6] “TurtleBot 2 | Clearpath Robotics.” [Online]. Available: https://store.clearpathrobotics.com/products/turtlebot-2.

[7] Pardo-Castellote, Gerardo, “Distributed Data Management,” *http://www.omg.org/news/meetings/workshops/Real-time_WS_Final_Presentations_2008/Tutorials/00-T1_Pardo-Castellote.pdf*. [Online]. Available: http://www.omg.org/news/meetings/workshops/Real-time_WS_Final_Presentations_2008/Tutorials/00-T1_Pardo-Castellote.pdf.

[8] D. Thomas, “Migration Guide from ROS 1,” *GitHub*. [Online]. Available: https://github.com/ros2/ros2/wiki/Migration-Guide.

[9] D. Thomas, “ros/ros_tutorials,” *GitHub*. [Online]. Available: https://github.com/ros/ros_tutorials.

[10] W. Wood, “ros2/demos,” *GitHub*. [Online]. Available: https://github.com/ros2/demos.

Appendix A: Hello World C++ example

ROS 1 Talker in C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
/**
 * This tutorial demonstrates simple sending of messages over the ROS system.
 */
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can perform
   * any ROS arguments and name remapping that were provided at the command line.
   * For programmatic remappings you can use a different version of init() which takes
   * remappings directly, but for most command-line programs, passing argc and argv is
   * the easiest way to do it.  The third argument to init() is the name of the node.
   *
   * You must call one of the versions of ros::init() before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "talker");
  /**
   * NodeHandle is the main access point to communications with the ROS system.
   * The first NodeHandle constructed will fully initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;
  /**
   * The advertise() function is how you tell ROS that you want to
   * publish on a given topic name. This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing. After this advertise() call is made, the master
   * node will notify anyone who is trying to subscribe to this topic name,
```

```
 * and they will in turn negotiate a peer-to-peer connection with this
 * node.  advertise() returns a Publisher object which allows you to
 * publish messages on that topic through a call to publish().  Once
 * all copies of the returned Publisher object are destroyed, the topic
 * will be automatically unadvertised.
 *
* The second parameter to advertise() is the size of the message queue
 * used for publishing messages.  If messages are published more quickly
 * than we can send them, the number here specifies how many messages to
 * buffer up before throwing some away.
 */
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
ros::Rate loop_rate(10);
/**
 * A count of how many messages we have sent. This is used to create
 * a unique string for each message.
 */
int count = 0;
while (ros::ok())
{
  /**
   * This is a message object. You stuff it with data, and then publish it.
   */
  std_msgs::String msg;
  std::stringstream ss;
  ss << "hello world " << count;
  msg.data = ss.str();
  ROS_INFO("%s", msg.data.c_str());
  /**
   * The publish() function is how you send messages. The parameter
   * is the message object. The type of this object must agree with the type
   * given as a template parameter to the advertise<>() call, as was done
   * in the constructor above.
   */
  chatter_pub.publish(msg);
  ros::spinOnce();
```

```
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

ROS 2 Talker in C++

```
#include <iostream>
#include <memory>
#include "rclcpp/rclcpp.hpp"                                Need to change headers and naming convection
#include "std_msgs/msg/string.hpp"
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);                                 Initialization, followed by create object
  auto node = rclcpp::node::Node::make_shared("talker");
  rmw_qos_profile_t custom_qos_profile = rmw_qos_profile_default;
  custom_qos_profile.depth = 7;
  auto chatter_pub = node->create_publisher<std_msgs::msg::String>("chatter", custom_qos_profile);       Create publisher with QoS profile
  rclcpp::WallRate loop_rate(2);
  auto msg = std::make_shared<std_msgs::msg::String>();     creating outgoing message
  auto i = 1;
  while (rclcpp::ok()) {
    msg->data = "Hello World: " + std::to_string(i++);      accessing the data in the pointer
    std::cout << "Publishing: '" << msg->data << "'" << std::endl;
    chatter_pub→publish(msg);                               publish the message
    rclcpp::spin_some(node);
    loop_rate.sleep();
  }
  Return 0;
}
```

ROS 1 Listener in C++

```
#include "ros/ros.h"
#include "std_msgs/String.h"
/**
```

```
 * This tutorial demonstrates simple receipt of messages over the ROS system.
 */
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
int main(int argc, char **argv)
{
  /**
   * The ros::init() function needs to see argc and argv so that it can perform
   * any ROS arguments and name remapping that were provided at the command line.
   * For programmatic remappings you can use a different version of init() which takes
   * remappings directly, but for most command-line programs, passing argc and argv is
   * the easiest way to do it.  The third argument to init() is the name of the node.
   *
   * You must call one of the versions of ros::init() before using any other
   * part of the ROS system.
   */
  ros::init(argc, argv, "listener");
  /**
   * NodeHandle is the main access point to communications with the ROS system.
   * The first NodeHandle constructed will fully initialize this node, and the last
   * NodeHandle destructed will close down the node.
   */
  ros::NodeHandle n;
  /**
   * The subscribe() call is how you tell ROS that you want to receive messages
   * on a given topic.  This invokes a call to the ROS
   * master node, which keeps a registry of who is publishing and who
   * is subscribing.  Messages are passed to a callback function, here
   * called chatterCallback.  subscribe() returns a Subscriber object that you
   * must hold on to until you want to unsubscribe.  When all copies of the Subscriber
   * object go out of scope, this callback will automatically be unsubscribed from
   * this topic.
   *
   * The second parameter to the subscribe() function is the size of the message
```

```
   * queue.  If messages are arriving faster than they are being processed, this
   * is the number of messages that will be buffered up before beginning to throw
   * away the oldest ones.
   */
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  /**
   * ros::spin() will enter a loop, pumping callbacks.  With this version, all
   * callbacks will be called from within this thread (the main one).  ros::spin()
   * will exit when Ctrl-C is pressed, or the node is shutdown by the master.
   */
  ros::spin();
  return 0;
}
```

ROS 2 Listener in C++

```
#include <iostream>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"                    Need to change the header files and use C++ 11 builtins
void chatterCallback(const std_msgs::msg::String::SharedPtr msg)
{
  std::cout << "I heard: [" << msg->data << "]" << std::endl;
}
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);                         Initialization and create the object. Callback is using QoS profiles
  auto node = rclcpp::Node::make_shared("listener");
  auto sub = node->create_subscription<std_msgs::msg::String>(
    "chatter", chatterCallback, rmw_qos_profile_default);      Subscribe to message using Topic string chatter
  rclcpp::spin(node);
  return 0;
}
```