Why ROS 2.0?

This article captures the reasons for making breaking changes to the ROS API, hence the 2.0.

Original Author: **Brian Gerkey**

We started work on ROS in November 2007. A lot has happened since then and we believe that it is now time to build the next generation ROS platform. In this article we will explain why.

How we got here

ROS began life as the development environment for the Willow Garage PR2 robot. Our primary goal was to provide the software tools that users would need to undertake novel research and development projects with the PR2. At the same time, we knew that the PR2 would not be the only, or even the most important, robot in the world, and we wanted ROS to be useful on other robots. So we put a lot of effort into defining levels of abstraction (usually through message interfaces) that would allow much of the software to be reused elsewhere.

Still, we were guided by the PR2 use case, the salient characteristics of which included:

- **a single robot;**
- workstation-class computational resources on board;
- no real-time requirements (or, any real-time requirements would be met in a special-purpose manner);
- excellent network connectivity (either wired or close-proximity high-bandwidth wireless);
- applications in research, mostly academia; and
- maximum flexibility, with nothing prescribed or proscribed (e.g., “we don’t wrap your main()”).

It is fair to say that ROS satisfied the PR2 use case, but also overshot by becoming useful on a surprisingly wide variety of robots. Today we see ROS used not only on the PR2 and robots that are similar to the PR2, but also on wheeled robots of all sizes, legged humanoids, industrial arms, outdoor ground vehicles (including self-driving cars), aerial vehicles, surface vehicles, and more.

In addition, we are seeing ROS adoption in domains beyond the mostly academic research community that was our initial focus. ROS-based products are coming to market, including manufacturing robots, agricultural robots, commercial cleaning robots, and others. Government agencies are also looking more closely at ROS for use in their fielded systems; e.g., NASA is expected to be running ROS on the Robonaut 2 that is deployed to the International Space Station.

With all these new uses of ROS, the platform is being stretched in unexpected ways. While it is holding up well, we believe that we can better meet the needs of a now-broader ROS community by tackling their new use cases head-on.

New use cases

Of specific interest to us for the ongoing and future growth of the ROS community are the following use cases, which we did not have in mind at the beginning of the project:

- **Teams of multiple robots**: while it is possible to build multi-robot systems using ROS today, there is no standard approach, and they are all somewhat of a hack on top of the single-master structure of ROS.
- **Small embedded platforms**: we want small computers, including “bare-metal” micro controllers, to be first-class participants in the ROS environment, instead of being segregated from ROS by a device driver.
- **Real-time systems**: we want to support real-time control directly in ROS, including inter-process and inter-machine communication (assuming appropriate operating system and/or hardware support).
- **Non-ideal networks**: we want ROS to behave as well as is possible when network connectivity degrades due to loss and/or delay, from poor-quality WiFi to ground-to-space communication links.
- **Production environments**: while it is vital that ROS continue to be the platform of choice in the research lab, we want to ensure that ROS-based lab prototypes can evolve into ROS-based products suitable for use in real-world applications.
- Prescribed patterns for building and structuring systems: while we will maintain the underlying flexibility that is the hallmark of ROS, we want to provide clear patterns and supporting tools for features such as life cycle management and static configurations for deployment.

New technologies

At the core of ROS is an anonymous publish-subscribe middleware system that is built almost entirely from scratch. Starting in 2007, we built our own systems for discovery, message definition, serialization, and transport. The intervening seven years have seen the development, improvement, and/or widespread adoption of several new technologies that are relevant to ROS in all of those areas, such as:

- Zeroconf;
- Protocol Buffers;
- ZeroMQ (and the other MQs);
- Redis;
- WebSockets; and
- DDS (Data Distribution Service).

It is now possible to build a ROS-like middleware system using off-the-shelf open source libraries. We can benefit tremendously from this approach in many ways, including:

- we maintain less code, especially non-robotics-specific code;

- we can take advantage of features in those libraries that are beyond the scope of what we would build ourselves;
- we can benefit from ongoing improvements that are made by others to those libraries; and
- we can point to existing production systems that already rely on those libraries when people ask us whether ROS is “ready for prime time”.

API changes

A further reason to build ROS 2.0 is to take advantage of the opportunity to improve our user-facing APIs. A great deal of the ROS code that exists today is compatible with the client libraries as far back as the 0.4 “Mango Tango” release from February 2009. That’s great from the point of view of stability, but it also implies that we’re still living with API decisions that were made several years ago, some of which we know now to be not the best.

So, with ROS 2.0, we will design new APIs, incorporating to the best of our ability the collective experience of the community with the first-generation APIs. As a result, while the key concepts (distributed processing, anonymous publish/subscribe messaging, RPC with feedback (i.e., actions), language neutrality, system introspectability, etc.) will remain the same, you should not expect ROS 2.0 to be API-compatible with existing ROS code.

But fear not: there will be mechanisms in place to allow ROS 2.0 code to coexist with existing ROS code. At the very least, there will be translation relays that will support run-time interactions between the two systems. And it is possible that there will be library shims that will allow existing ROS code to compile/run against ROS 2.0 libraries, with behavior that is qualitatively similar to what is seen today.

Why not just enhance ROS 1

In principle, the changes described above could be integrated into the existing core ROS code. E.g., new transport technologies could be added to `roscpp` and `rospy`. We considered this option and concluded that, given the intrusive nature of the changes that would be required to achieve the benefits that we are seeking, there is too much risk associated with changing the current ROS system that is relied upon by so many people. We want ROS 1 as it exists today to keep working and be unaffected by the development of ROS 2. So ROS 2 will be built as a parallel set of packages that can be installed alongside and interoperate with ROS 1 (e.g., through message bridges).

ROS on DDS http://design.ros2.org/articles/ros_on_dds.html

What are the Benefits of DDS?

http://www.prismtech.com/vortex/technologies/what-are-benefits-dds

The Data Distribution Service for Real-Time Systems (DDS) has many benefits.

DDS makes it much easier to develop distributed applications: DDS provides a completely de-centralized architecture that enables loosely coupled systems. Applications communicate in a true peer-to-peer way, there are no intermediate services or message brokers that can introduce single points of failure or performance bottlenecks. DDS systems are dynamic and can support "plug-and-play" for new application components making it easy to extend or evolve a system.

DDS applications are completely decoupled from each other and an application can still publish information even if there are no subscriber applications active. DDS automatically decides which subscriber applications should receive information and can ensure that the right data is delivered