Autonomous Navigation – Ros & Rviz

**Summary**: ROS (Robot Operating System) is an open-source robotic software framework that provides a set of tools and libraries for developing robot software. It is not an operating system in the traditional sense of process management and scheduling; instead, it offers a built-in communication layer above the hosting operating system's computing cluster. This article addresses how ROS is related to robotics software frameworks and provides an overview of some available options for ROS-based application development.

ROS is essentially a collection of robotic middleware software packages. While not an operating system itself, ROS provides services intended for heterogeneous computing clusters, such as hardware abstraction, low-level device control, common functionality implementation, inter-process message passing, and package management. Executing sets of ROS-based processes are represented in a graph architecture where processing occurs in nodes that can receive, publish, sensor data, control, state, planning, actuate, and other messages. Despite the importance of responsiveness and low latency in robot control, ROS is not a real-time operating system (RTOS). However, ROS can be integrated with real-time code. The lack of real-time system support was addressed by creating a large-scale version of the ROS API that leverages modern libraries and technologies for ROS core functionality while adding support for real-time code and embedded systems.

Software development for robots is challenging, especially given the scale and scope of the growing robotics field. Robots can have highly variable hardware, and code reuse and reusability are not trivial. Additionally, the sheer size of the required code can be daunting, as it must encompass low-level device drivers, abstract thinking software at the device manager level, and beyond. Since the required expertise goes well beyond what a single researcher can possess, robotics software architectures need to support large-scale software integration efforts. To meet these challenges, robotics researchers have created a variety of frameworks for managing complexity and enabling rapid software prototyping. As a result, many robotic software systems are primarily used in industry and academia. Each of these frameworks was designed for a specific purpose, often in response to perceived weaknesses in existing frameworks, and/or with an emphasis on aspects seen as important in the design process. ROS, the framework described here, is also a product of trade-offs and prioritizations made during its design cycle. It can be said that the emphasis on integrability at a large scale has been a focal point of robotics research and is applicable in a wide range of situations as robotic systems grow, scale, and become more complex. Here, the emphasis is on the design goals of ROS and how its implementation addresses various use cases in common robotics software development.

**Design Goals**: It is not claimed that ROS is the best framework for all robotics software. In fact, the field of robotics is too broad for a single, solitary solution. ROS is designed to address a specific set of challenges encountered during large-scale development, such as slow, flowing movement over a wireless link, as many message paths are fully included in both the robot's secondary networks or outside of it.

In contrast, peer-to-peer connectivity, combined with gating or "fanout" software modules as needed, completely avoids the issue. The peer-to-peer topology requires some discovery mechanism that allows processes to find each other at runtime, that is a machine (service or master).

**Multilingual**: When writing code, many people have preferences for certain programming languages over others. These preferences are a result of personal trade-offs between programming time, ease of debugging, syntax, efficiency at runtime, and a myriad of technical and cultural reasons. For these reasons, ROS is designed to be language neutral. It currently supports four different languages: C++, Python, Octave, and LISP, with other languages coming out and in various stages of completion.

The ROS specification is in the messaging layer, not deeper within each one. Negotiation and configuration of peer-to-peer connections occur via XML-RPC, for which there are reasonable implementations in most major languages. Rather than providing a C-based embedding with stub interfaces generated for all major languages, the preference is to implement ROS natively in each target language, to better follow the conventions of each language. However, in some cases, it is indeed beneficial to add support for a new language by wrapping an existing library: the Octave client is implemented by wrapping the ROS C++ library. To support cross-language development, ROS uses a language-neutral Interface Definition Language (IDL) to describe the messages sent between modules. The IDL uses short text files to describe the fields of each message and allows message nesting, as demonstrated by a full IDL file for points in a cloud message:

ROS is distributed under the terms of the BSD license, which allows both non-commercial and commercial project development. It transfers data between modules using inter-process communication and does not require that the modules connecting together be in the same executable. As such, systems built around ROS can take nuanced licensing of their various components: individual modules can combine software protected by different licenses, ranging from GPL to BSD.

Code generators are then created for each supported language, creating native applications that "feel" like local objects and also automatically serialize and deserialize as ROS messages are sent and received. The three-line IDL file from earlier automatically expands to 137 lines of C++, 96 of Python, 81 lines of Lisp, and 99 lines of Octave. This is because messages are automatically generated from simple text. At the time of writing, known ROS-based codebases contain more than four hundred message types, transmitting data ranging from sensor inputs to object detection to maps. The end result is a language-neutral message processing scheme in which different languages can be mixed and matched.

**Terminology**: (NOMENCLATURE) The basic concepts of ROS application are nodes, messages, topics, and services. Nodes are processes that perform computations. ROS is designed to be modular at a fine scale: a complex system typically consists of many nodes. In this context, the term "node" can be replaced with "software module". The use of the term "node" arises from the analogy of ROS-based systems. At runtime: when multiple nodes are running, it's convenient for communication processing in a peer-to-peer graph, with processes as graph nodes and peer-to-peer links as edges.

Nodes communicate with each other by transmitting messages. A message is a strictly typed data structure. Primitive types (integer, floating point, boolean, etc.) are supported, as well as arrays of fixed and primitive types. Messages can be composed of other messages, and of arrays of other messages, nested arbitrarily deep.

A node sends a message by publishing it to a given topic, which is simply a string such as "odometry" or "map". A node interested in a certain type of data will subscribe to the appropriate topic. There may be multiple simultaneous publishers and subscribers to a single topic and a node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each other's existence.

The simplest communication is along pipelines: Microphone -> speech recognition -> dialog manager -> speech synthesis -> speaker. However, graphs are usually much more complex, often containing cycles and one-to-many or many-to-many connections. Although the publish-subscribe model provides a flexible communication paradigm, its "broadcast" routing does not suit synchronous transactions, which can simplify the design of some nodes. In ROS, this is called a service, defined by a string name and a pair of strictly typed messages: one for the request and one for the response. This is analogous to web services, which are defined by a URI and have well-defined request and response documents. Note that unlike topics, only one node can publish a service with any given name: there can only be one service called "image segmentation", for example, just like there can only be one web service at any given URI.

In four common cases, we will describe situations and scenarios encountered when using robotic software frameworks. ROS's open architecture allows for the creation of a wide range of tools; in describing ROS's approach to these use cases, we will also introduce a number of tools intended for use with ROS.

(a) **Debugging a single node**: When conducting robotics research, the scope of investigation is often limited to a well-defined area of the system, such as a node performing some form of planning, perception, or control. However, to obtain a functioning robotic system for experiments, a much larger ecosystem of software needs to exist. For example, to perform vision-based grasping experiments, drivers must be operational for cameras and manipulators, and any number of intermediate processing nodes (such as object detection, localization, path planners) must also be running.

This adds a significant amount of difficulty to integrative robotics research. ROS is designed to minimize the difficulty of debugging in such setups, as its modular structure allows nodes to be actively developed while running alongside existing well-developed nodes. Since nodes connect to each other at runtime, the graph can be dynamically changed.

In the previous example of vision-based grasping, perhaps thirteen nodes are required to provide the infrastructure. This infrastructure graph can be started and left running throughout an entire experimental session. Only nodes that undergo changes in the source code need to be restarted periodically, and during this time, ROS quietly handles the graph modifications. This can result in a substantial increase in productivity, especially as the robotic system becomes more complex and interconnected.

To emphasize, graph modification in ROS simply amounts to starting or stopping processes. In debugging setups, this is typically done through command-line or debugging tool interventions. The ease of adding and removing nodes from a ROS-based system is one of its most powerful and fundamental features.

(b) **Logging and playback**: Robotics research is often facilitated by the use of recorded sensor data, allowing for comparative control of different algorithms and simplifying the experimental process. ROS supports this approach by providing generic logging and playback functionality. Any ROS stream of messages can be recorded to disk and played back later.

(c) **Embedded systems**: In certain areas of robotics research, such as indoor navigation robots, progress has been made to a point where "box algorithms" can function reasonably well. ROS harnesses the algorithms implemented in the Player project to provide a navigation system that generates this graph. While each node can be individually activated from the command line, input/output from the navigation system, ROS provides a tool called "roslaunch" that reads an XML description of the graph and displays the graph on the console, optionally on specific hosts. The end-user experience of launching the navigation system then simply involves executing "roslaunch navstack.xml" and pressing Ctrl-C will cleanly shut down all five processes. This functionality can also greatly assist in the collaborative and reusable use of large robotic research frameworks, as the definition and decomposition of complex distributed systems can be easily replicated.

(d) **Collaborative Development**: Due to the immense scope of robotics and artificial intelligence, collaboration among researchers is essential for building large-scale systems. To support collaborative development, the ROS software system is organized into packages. The definition of a "package" is intentionally open: a ROS package is essentially a library that contains an XML file describing the package and specifying all dependencies. A collection of ROS packages forms a directory tree with ROS packages at the leaves. A ROS package repository is typically composed of a complex hierarchy of sub-libraries. For example, a ROS repository may have root libraries such as "nav," "vision," and "motion planning," each containing multiple sub-libraries. ROS provides a utility tool called rospack for querying and inspecting the code tree, searching for dependencies, finding packages by name, and more.

The open nature of ROS packages allows for great versatility in their structure and purpose. Some ROS packages wrap existing software programs, such as Player or OpenCV, automating their builds and exposing their functionality. Some packages build nodes for use in ROS graphs, while others provide libraries and standalone executable files, and others provide scripts to automate demonstrations and tests. The packaging system aims to divide the structure of ROS-based software into manageable components, so that each component can be maintained and developed on its own schedule and by its own team of developers.

**Visualization and Monitoring:** During the design and debugging of robotics software, it is often necessary to observe the system's state while it is running. While printf is a familiar technique for debugging programs on a single machine, this technique can become challenging to scale for large distributed systems and can be cumbersome for general-purpose monitoring. Instead, ROS can leverage the dynamic nature of the connectivity graph to "tap into" any stream of system messages. Furthermore, the decoupling between publishers and subscribers allows for the creation of general-purpose visualizations. Simple programs can be written that subscribe to specific topics and visualize specific types of data, such as laser scans or images. However, a more powerful concept is the user-programmable visualization using an architecture add-on, which is done through the rviz program, distributed with ROS. Dynamic visualization panels can be created to display a wide range of data types, such as images, point clouds, geometric primitives (e.g., object recognition results), robot poses and trajectories, and more.

Plugins can be easily written to display more types of data. The native ROS distribution provides support for Python, a dynamically typed language that supports introspection. A powerful utility tool called rostopic is written to filter messages using expressions provided on the command line, making it "hard on the message" that can be immediately customized to convert any part of any data stream into a text stream. These text streams can be piped into other UNIX command-line tools such as grep, sed, and awk to create complex monitoring tools without writing any code. Similarly, a tool called rxplot provides the functionality of a virtual oscilloscope, plotting any variable in real-time as a time series, again leveraging the introspection capabilities of Python and expression evaluation.

**Composition of functionality**: In ROS, a "stack" of software is a collection of nodes that together perform some useful task, as demonstrated in the navigation example. As described earlier, ROS is capable of creating a cluster of nodes with a single command, where the cluster is described in an XML file. However, there are times when multiple instances of a cluster are desired. For example, in multi-robot experiments, there may be a need for a navigation stack for each robot in the system, and robots with humanoid upper bodies may require two identical arm controllers. ROS supports this by allowing the instantiation of nodes and clusters from complete description files that are pushed into the namespace, ensuring there are no naming collisions. Topics and service names can be remapped without any modifications to the code of the node or cluster. The following graph shows a hierarchical multi-robot control system built simply by creating multiple navigation stacks, each with its own namespace. Topics and service names can be remapped, allowing for easy reuse of code without any modifications. The previous graph was automatically generated by the rxgraph tool, which can inspect and visualize any ROS graph at runtime. Its output turns nodes into ellipses, topics into squares, and connections into edges.

**Transformations**: Robotic systems often need to track spatial relationships for various reasons, such as between a mobile robot and multiple fixed frames for localization, between different sensor frames and manipulator frames, and to establish a frame of reference for object manipulation purposes. To simplify and unify the handling of spatial frames, a transformation system called tf was developed for ROS. The tf system builds a dynamic transformation tree that links all the frames of reference in the system. When information flows internally from various subsystems of the robot (shared encoders, localization algorithms, etc.), the tf system can generate streams of transformations between desired nodes on the tree by constructing a path between them and performing the necessary calculations. For example, the tf system can be used to easily create point clouds within a static "map" frame from laser scans obtained by a laser scanner mounted on a moving robot. Another example is a dual-arm robot: the tf system can stream the transformation from the palm root camera to one robotic arm to the end-effector of the second arm of the robot. Such calculations can be laborious, error-prone, and difficult to debug when hard-coded, but the application of tf, in combination with the dynamic message infrastructure of ROS, allows for an automatic and systematic approach.

**Summary**: It can be said that the design of ROS aims to support the perception of modular software development based on tools. As described above, it is expected that further expansion and development of additional tools will be possible by factors within the field and others, with the goal of building robotic software systems that can be useful for a variety of platforms such as hardware, research, and more.

The main advantages of using ROS can be summarized as follows:

1. **Simplified Hardware Integration**: Developers are only responsible for developing the code that is linked to the application, as ROS provides a high level of hardware abstraction. Most commonly used functions are already developed, eliminating the need to reinvent the wheel.

2. **Flexibility Across Different Hardware Platforms**: Different nodes can operate on different hardware platforms within a single ROS system, providing great flexibility. Nodes developed for a specific application can be easily adapted for another application without any code changes. Only the required software packages for the application need to be installed, without the need for the entire ROS package.

3. **Simulation Capabilities**: ROS allows for simulations using digital twins, and the programming code used in these simulations can be used to operate with real robots. This enables the debugging of code without any risk before moving on to testing with real robots.

4. **Compatibility with Various Hardware Platforms:** ROS is compatible with a wide range of hardware platforms, even though they may differ significantly in terms of hardware. This allows the use of ROS on the appropriate platform. For example, in industries such as autonomous mobile platforms capable of autonomously moving heavy loads, ROS can be utilized without exposing workers to hazardous or harmful tasks. Smart logistic warehouses can be cited as an example.

Overall, ROS offers developers a modular software development framework, simplifying hardware integration and providing flexibility across different platforms. It also allows for simulations and compatibility with various hardware platforms, enabling the development of robotic systems for a wide range of applications.