ROS: an open-source Robot Operating System

Morgan Quigley*, Brian Gerkey†, Ken Conley†, Josh Faust†, Tully Foote†,
Jeremy Leibs‡, Eric Berger†, Rob Wheeler†, Andrew Ng*

*Computer Science Department, Stanford University, Stanford, CA
†Willow Garage, Menlo Park, CA
‡Computer Science Department, University of Southern California

***Abstract*— This paper gives an overview of ROS, an open-source robot operating system. ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating systems of a heterogenous compute cluster. In this paper, we discuss how ROS relates to existing robot software frameworks, and briefly overview some of the available application software which uses ROS.**

I. INTRODUCTION

Writing software for robots is difficult, particularly as the scale and scope of robotics continues to grow. Different types of robots can have wildly varying hardware, making code reuse nontrivial. On top of this, the sheer size of the required code can be daunting, as it must contain a deep stack starting from driver-level software and continuing up through perception, abstract reasoning, and beyond. Since the required breadth of expertise is well beyond the capabilities of any single researcher, robotics software architectures must also support large-scale software integration efforts.

To meet these challenges, many robotics researchers, including ourselves, have previously created a wide variety of frameworks to manage complexity and facilitate rapid prototyping of software for experiments, resulting in the many robotic software systems currently used in academia and industry [1]. Each of these frameworks was designed for a particular purpose, perhaps in response to perceived weaknesses of other available frameworks, or to place emphasis on aspects which were seen as most important in the design process.

ROS, the framework described in this paper, is also the product of tradeoffs and prioritizations made during its design cycle. We believe its emphasis on large-scale integrative robotics research will be useful in a wide variety of situations as robotic systems grow ever more complex. In this paper, we discuss the design goals of ROS, how our implementation works towards them, and demonstrate how ROS handles several common use cases of robotics software development.

II. DESIGN GOALS

We do not claim that ROS is the best framework for all robotics software. In fact, we do not believe that such a framework exists; the field of robotics is far too broad for a single solution. ROS was designed to meet a specific set of challenges encountered when developing large-scale service robots as part of the STAIR project [2] at Stanford University[1] and the Personal Robots Program [3] at Willow Garage,[2] but the resulting architecture is far more general than the service-robot and mobile-manipulation domains.

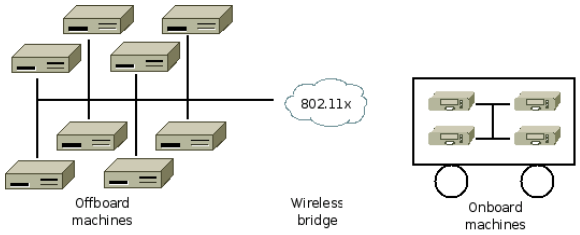


Fig. 1. A typical ROS network configuration

The philosophical goals of ROS can be summarized as:

- Peer-to-peer
- Tools-based
- Multi-lingual
- Thin
- Free and Open-Source

To our knowledge, no existing framework has this same set of design criteria. In this section, we will elaborate these philosophies and shows how they influenced the design and implementation of ROS.

A. Peer-to-Peer

A system built using ROS consists of a number of processes, potentially on a number of different hosts, connected at runtime in a peer-to-peer topology. Although frameworks based on a central server (e.g., CARMEN [4]) can also realize the benefits of the multi-process and multi-host design, a central data server is problematic if the computers are connected in a heterogenous network.

For example, on the large service robots for which ROS was designed, there are typically several onboard computers connected via ethernet. This network segment is bridged via wireless LAN to high-power offboard machines that are running computation-intensive tasks such as computer vision or speech recognition (Figure 1). Running the central server either onboard or offboard would result in unnecessary

[1] http://stair.stanford.edu
[2] http://pr.willowgarage.com

traffic flowing across the (slow) wireless link, because many message routes are fully contained in the subnets either onboard or offboard the robot. In contrast, peer-to-peer connectivity, combined with buffering or "fanout" software modules where necessary, avoids the issue entirely.

The peer-to-peer topology requires some sort of lookup mechanism to allow processes to find each other at runtime. We call this the *name service*, or *master*, and will describe it in more detail shortly.

B. Multi-lingual

When writing code, many individuals have preferences for some programming languages above others. These preferences are the result of personal tradeoffs between programming time, ease of debugging, syntax, runtime efficiency, and a host of other reasons, both technical and cultural. For these reasons, we have designed ROS to be language-neutral. ROS currently supports four very different languages: C++, Python, Octave, and LISP, with other language ports in various states of completion.

The ROS specification is at the messaging layer, not any deeper. Peer-to-peer connection negotiation and configuration occurs in XML-RPC, for which reasonable implementations exist in most major languages. Rather than provide a C-based implementation with stub interfaces generated for all major languages, we prefer instead to implement ROS natively in each target language, to better follow the conventions of each language. However, in some cases it is expedient 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 simple, language-neutral interface definition language (IDL) to describe the messages sent between modules. The IDL uses (very) short text files to describe fields of each message, and allows composition of messages, as illustrated by the complete IDL file for a point cloud message:

```
Header header
Point32[] pts
ChannelFloat32[] chan
```

Code generators for each supported language then generate native implementations which "feel" like native objects, and are automatically serialized and deserialized by ROS as messages are sent and received. This saves considerable programmer time and errors: the previous 3-line IDL file automatically expands to 137 lines of C++, 96 lines of Python, 81 lines of Lisp, and 99 lines of Octave. Because the messages are generated automatically from such simple text files, it becomes easy to enumerate new types of messages. At time of writing, the known ROS-based codebases contain over four hundred types of messages, which transport data ranging from sensor feeds to object detections to maps.

The end result is a language-neutral message processing scheme where different languages can be mixed and matched as desired.

C. Tools-based

In an effort to manage the complexity of ROS, we have opted for a microkernel design, where a large number of small tools are used to build and run the various ROS components, rather than constructing a monolithic development and runtime environment.

These tools perform various tasks, e.g., navigate the source code tree, get and set configuration parameters, visualize the peer-to-peer connection topology, measure bandwidth utilization, graphically plot message data, auto-generate documentation, and so on. Although we could have implemented core services such as a global clock and a logger inside the *master* module, we have attempted to push everything into separate modules. We believe the loss in efficiency is more than offset by the gains in stability and complexity management.

D. Thin

As eloquently described in [5], most existing robotics software projects contain drivers or algorithms which *could* be reusable outside of the project. Unfortunately, due to a variety of reasons, much of this code has become so entangled with the middleware that it is difficult to "extract" its functionality and re-use it outside of its original context.

To combat this tendency, we encourage all driver and algorithm development to occur in standalone libraries that have no dependencies on ROS. The ROS build system performs modular builds inside the source code tree, and its use of CMake makes it comparatively easy to follow this "thin" ideology. Placing virtually all complexity in libraries, and only creating small executables which expose library functionality to ROS, allows for easier code extraction and reuse beyond its original intent. As an added benefit, unit testing is often far easier when code is factored into libraries, as standalone test programs can be written to exercise various features of the library.

ROS re-uses code from numerous other open-source projects, such as the drivers, navigation system, and simulators from the Player project [6], vision algorithms from OpenCV [7], and planning algorithms from OpenRAVE [8], among many others. In each case, ROS is used only to expose various configuration options and to route data into and out of the respective software, with as little wrapping or patching as possible. To benefit from the continual community improvements, the ROS build system can automatically update source code from external repositories, apply patches, and so on.

E. Free and Open-Source

The full source code of ROS is publicly available. We believe this to be critical to facilitate debugging at all levels of the software stack. While proprietary environments such as Microsoft Robotics Studio [9] and Webots [10] have many commendable attributes, we feel there is no substitute for a fully open platform. This is particularly true when hardware and many levels of software are being designed and debugged in parallel.

ROS is distributed under the terms of the BSD license, which allows the development of both non-commercial and commercial projects. ROS passes data between modules using inter-process communications, and does not require that modules link together in the same executable. As such, systems built around ROS can use fine-grain licensing of their various components: individual modules can incorporate software protected by various licenses ranging from GPL to BSD to proprietary, but license "contamination" ends at the module boundary.

III. Nomenclature

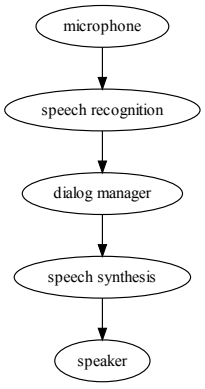
The fundamental concepts of the ROS implementation are nodes, messages, topics, and services,

Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale: a system is typically comprised of many nodes. In this context, the term "node" is interchangable with "software module." Our use of the term "node" arises from visualizations of ROS-based systems at runtime: when many nodes are running, it is convenient to render the peer-to-peer communications as a graph, with processes as graph nodes and the peer-to-peer links as arcs.

Nodes communicate with each other by passing **messages**. A message is a a strictly typed data structure. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types and constants. Messages can be composed of other messages, and 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 that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence.

The simplest communications are along pipelines:



However, graphs are usually far more complex, often containing cycles and one-to-many or many-to-many connections.

Although the topic-based publish-subscribe model is a flexible communications paradigm, its "broadcast" routing scheme is not appropriate for synchronous transactions, which can simplify the design of some nodes. In ROS, we call this 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 URIs and have request and response documents of well-defined types. Note that, unlike topics, only one node can advertise a service of any particular name: there can only be one service called "classify_image", for example, just as there can only be one web service at any given URI.

IV. Use Cases

In this section, we will describe a number of common scenarios encountered when using robotic software frameworks. The open architecture of ROS allows for the creation of a wide variety of tools; in describing the ROS approach to these use cases, we will also be introducing a number of the tools designed to be used with ROS.

A. Debugging a single node

When performing robotics research, often the scope of the investigation is limited to a well-defined area of the system, such as a node which performs some type of planning, reasoning, perception, or control. However, to get a robotic system up and running for experiments, a much larger software ecosystem must exist. For example, to do vision-based grasping experiments, drivers must be running for the camera(s) and manipulator(s), and any number of intermediate processing nodes (e.g., object recognizers, pose detectors, trajectory planners) must also be up and running. This adds a significant amount of difficulty to integrative robotics research.

ROS is designed to minimize the difficulty of debugging in such settings, as its modular structure allows nodes undergoing active development to run alongside pre-existing, well-debugged nodes. Because nodes connect to each other at runtime, the graph can be dynamically modified. In the previous example of vision-based grasping, a graph with perhaps a dozen nodes is required to provide the infrastructure. This "infrastructure" graph can be started **and left running** during an entire experimental session. Only the node(s) undergoing source code modification need to be periodically restarted, at which time ROS silently handles the graph modifications. This can result in a massive increase in productivity, particularly as the robotic system becomes more complex and interconnected.

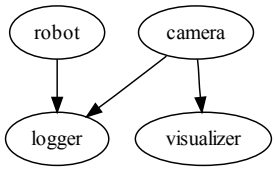
To emphasize, altering the graph in ROS simply amounts to starting or stopping a process. In debugging settings, this is typically done at the command line or in a debugger. The ease of inserting and removing nodes from a running ROS-based system is one of its most powerful and fundamental features.

B. Logging and playback

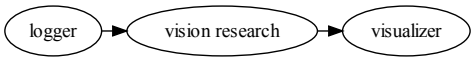
Research in robotic perception is often done most conveniently with logged sensor data, to permit controlled

comparisons of various algorithms and to simplify the experimental procedure. ROS supports this approach by providing generic logging and playback functionality. Any ROS message stream can be dumped to disk and later replayed. Importantly, this can all be done at the command line; it requires no modification of the source code of any pieces of software in the graph.

For example, the following network graph could be quickly set up to collect a dataset for visual-odometry research:



The resulting message dump can be played back into a different graph, which contains the node under development:

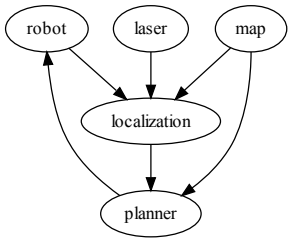


As before, node instantiation can be performed simply by launching a process; it can be done at the command line, in a debugger, from a script, etc.

To facilitate logging and monitoring of systems distributed across many hosts, the `rosconsole` library builds upon the Apache project's `log4cxx` system to provide a convenient and elegant logging interface, allowing `printf`-style diagnostic messages to be routed through the network to a single stream called `rosout`.

C. Packaged subsystems

Some areas of robotics research, such as indoor robot navigation, have matured to the point where "out of the box" algorithms can work reasonably well. ROS leverages the algorithms implemented in the Player project to provide a navigation system, producing this graph:



Although each node can be run from the command line, repeatedly typing the commands to launch the processes could get tedious. To allow for "packaged" functionality such as a navigation system, ROS provides a tool called `roslaunch`, which reads an XML description of a graph and instantiates the graph on the cluster, optionally on specific hosts. The end-user experience of launching the navigation system then boils down to

```
roslaunch navstack.xml
```

and a single `Ctrl-C` will gracefully close all five processes. This functionality can also significantly aid sharing and reuse of large demonstrations of integrative robotics research, as the set-up and tear-down of large distributed systems can be easily replicated.

D. Collaborative Development

Due to the vast scope of robotics and artificial intelligence, collaboration between researchers is necessary in order to build large systems. To support collaborative development, the ROS software system is organized into **packages**. Our definition of "package" is deliberately open-ended: a ROS package is simply a directory which contains an XML file describing the package and stating any dependencies.

A collection of ROS packages is a directory tree with ROS packages at the leaves: a ROS package repository may thus contain an arbitrarily complex scheme of subdirectories. For example, one ROS repository has root directories including "nav," "vision," and "motion_planning," each of which contains many packages as subdirectories.

ROS provides a utility called `rospack` to query and inspect the code tree, search dependencies, find packages by name, etc. A set of shell expansions called `rosbash` is provided for convenience, accelerating command-line navigation of the system.

The `rospack` utility is designed to support simultaneous development across multiple ROS package repositories. Environment variables are used to define the roots of local copies of ROS package repositories, and `rospack` crawls the package trees as necessary. Recursive builds, supported by the `rosmake` utility, allow for cross-package library dependencies.

The open-ended nature of ROS packages allows for great variation in their structure and purpose: some ROS packages wrap existing software, such as Player or OpenCV, automating their builds and exporting their functionality. Some packages build nodes for use in ROS graphs, other packages provide libraries and standalone executables, and still others provide scripts to automate demonstrations and tests. The packaging system is meant to partition the building of ROS-based software into small, manageable chunks, each of which can be maintained and developed on its own schedule by its own team of developers.

At time of writing, several hundred ROS packages exist across several publicly-viewable repositories, and hundreds more likely exist in private repositories at various institutions and companies. The ROS core is distributed as its own package repository in Sourceforge:

`http://ros.sourceforge.net`

However, the ros repository includes only the base ROS communications infrastructure and graph-management tools. Software which actually builds robotic systems using ROS is provided in a second repository, also on Sourceforge:

http://personalrobots.sourceforge.net

This repository contains many useful tools and libraries, such as those discussed in this paper.

E. Visualization and Monitoring

While designing and debugging robotics software, it often becomes necessary to observe some state while the system is running. Although printf is a familiar technique for debugging programs on a single machine, this technique can be difficult to extend to large-scale distributed systems, and can become unwieldly for general-purpose monitoring.

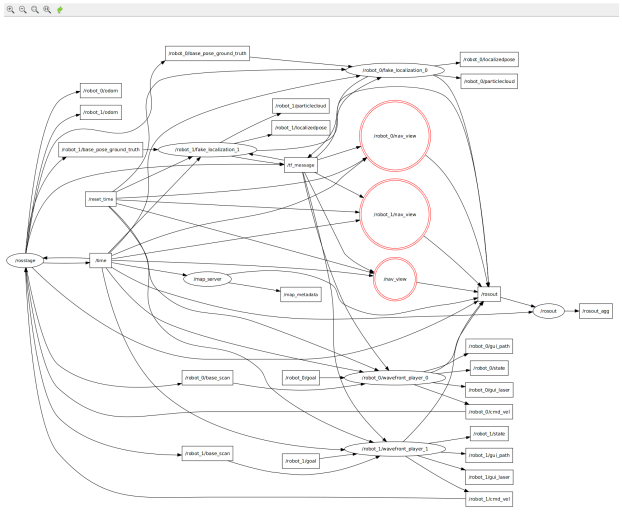
Instead, ROS can exploit the dynamic nature of the connectivity graph to "tap into" any message stream on the system. Furthermore, the decoupling between publishers and subscribers allows for the creation of general-purpose visualizers. Simple programs can be written which subscribe to a particular topic name and plot a particular type of data, such as laser scans or images. However, a more powerful concept is a visualization program which uses a plugin architecture: this is done in the rviz program, which is distributed with ROS. Visualization panels can be dynamically instantiated to view a large variety of datatypes, such as images, point clouds, geometric primitives (such as object recognition results), render robot poses and trajectories, etc. Plugins can be easily written to display more types of data.

A native ROS port is provided for Python, a dynamically-typed language supporting introspection. Using Python, a powerful utility called rostopic was written to filter messages using expressions supplied on the command line, resulting in an instantly customizable "message tap" which can convert any portion of any data stream into a text stream. These text streams can be piped to 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 through the use of Python introspection and expression evaluation.

F. Composition of functionality

In ROS, a "stack" of software is a cluster of nodes that does something useful, as was illustrated in the navigation example. As previously described, ROS is able to instantiate a cluster of nodes with a single command, once the cluster is described in an XML file. However, sometimes *multiple* instantiations of a cluster are desired. For example, in multi-robot experiments, a navigation stack will be needed for each robot in the system, and robots with humanoid torsos will likely need to instantiate two identical arm controllers. ROS supports this by allowing nodes and entire roslaunch cluster-description files to be pushed into a child **namespace**, thus ensuring that there can be no name collisions. Essentially, this prepends a string (the namespace) to all node, topic, and service names, without requiring any modification to the code of the node or cluster. The following graph shows a hierarchical multi-robot control system constructed by simply instantiating multiple navigation stacks, each in their own namespace:



The previous graph was automatically generated by the rxgraph tool, which can inspect and monitor any ROS graph at runtime. Its output renders nodes as ovals, topics as squares, and connectivity as arcs.

G. Transformations

Robotic systems often need to track spatial relationships for a variety of reasons: between a mobile robot and some fixed frame of reference for localization, between the various sensor frames and manipulator frames, or to place frames on target objects for control purposes.

To simplify and unify the treatment of spatial frames, a transformation system has been written for ROS, called tf. The tf system constructs a dynamic *transformation tree* which relates all frames of reference in the system. As information streams in from the various subsystems of the robot (joint encoders, localization algorithms, etc.), the tf system can produce streams of transformations between nodes on the tree by constructing a path between the desired nodes and performing the necessary calculations.

For example, the tf system can be used to easily generate point clouds in a stationary "map" frame from laser scans received by a tilting laser scanner on a moving robot. As another example, consider a two-armed robot: the tf system can stream the transformation from a wrist camera on one robotic arm to the moving tool tip of the second arm of the robot. These types of computations can be tedious, error-prone, and difficult to debug when coded by hand, but the tf implementation, combined with the dynamic messaging infrastructure of ROS, allows for an automated, systematic approach.

V. Conclusion

We have designed ROS to support our philosophy of modular, tools-based software development. We anticipate that its open-ended design can be extended and built upon by others to build robot software systems which can be useful to a variety of hardware platforms, research settings, and runtime requirements.

Acknowledgements

We thank the fledgeling ROS user community for their feedback and contributions, especially Rosen Diankov (author of the ROS Octave library) and Bhaskara Marthi (author of the ROS LISP library).

References

[1] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007.

[2] M. Quigley, E. Berger, and A. Y. Ng, "STAIR: Hardware and Software Architecture," in *AAAI 2007 Robotics Workshop, Vancouver, B.C, August*, 2007.

[3] K. Wyobek, E. Berger, H. V. der Loos, and K. Salisbury, "Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2008.

[4] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) Toolkit," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, Las Vegas, Nevada, Oct. 2003, pp. 2436–2441.

[5] A. Makarenko, A. Brooks, and T. Kaupp, in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, Nov. 2007.

[6] R. T. Vaughan and B. P. Gerkey, "Reusable robot code and the Player/Stage Project," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts on Advanced Robotics, D. Brugali, Ed. Springer, 2007, pp. 267–289.

[7] G. Bradski and A. Kaehler, *Learning OpenCV*, Sep. 2008.

[8] R. Diankov and J. Kuffner, "The robotic busboy: Steps towards developing a mobile robotic home assistant," in *Intelligent Autonomous Systems*, vol. 10, 2008.

[9] J. Jackson, "Microsoft robotics studio: A technical introduction," in *IEEE Robotics and Automation Magazine*, Dec. 2007, http://msdn.microsoft.com/en-us/robotics.

[10] O. Michel, "Webots: a powerful realistic mobile robots simulator," in *Proc. of the Second Intl. Workshop on RoboCup*. Springer-Verlag, 1998.