Robot Operating System and its implementation on PC architecture

Stefan Dragićević, Miloš D Jovanović

Abstract — **Prevalent problem in robotic development was that engineers lacked common platform for communication and collaboration. Every time they started a new project they needed to build software and hardware from the beginning. To overcome this problem various robotic platforms were developed and today's leading platform for robotics is Robot Operating System (ROS). ROS is intending to be worldwide useful robotic operation system as well as for industrial applications and scientific research. The paper presents a main concept of the ROS, advantages and comparison of ROS and other frameworks used in robotics. PC implementation and using PC platform as a robotic controller is the main goal of the paper.**

Key words **— operation system, PC platform, robot, ROS.**

I. Introduction

THE Robot Operating System (ROS) is a meta-operating system, something between an operating system and middleware. It is a collection of software frameworks for robot software development, providing operating system-like functionality.

ROS is developed using the permissive BSD open-source license, and gradually has become a widely-used platform in the robotics research community.

Stefan Dragićević, The Faculty of Computer Science, Union University in Belgrade, Trg Republike 2, 11000 Beograd, Serbia, (e-mail: s.v.dragicevic@gmail.com)

Miloš D Jovanović, Institute Mihajlo Pupin, University in Belgrade, Volgina 15, 11000 Beograd, Serbia, (e-mail: milos.jovanovic@pupin.rs)

Paper was also published in proceedings at 2nd International Conference on Electrical, Electronic and Computing Engineering IcETRAN 2015, Silver Lake, Serbia, June 8 – 11, 2015

It provides standard operating system services such as hardware abstraction, contention management and process management, as well as high-level functionalities such as asynchronous and synchronous calls, centralized database, a robot configuration system and other functionalities.

In past engineers would spend significant amount of time designing robot's hardware and software within. Each time they had to redesign and reprogram new robots.

The main idea of ROS is to avoid reinventing the wheel over and over again, and it tries to simplify the task of designing and programing complex and reliable robot behavior across various robotic platforms. This is not an easy task to accomplish because there is a large number of variations of robotic systems.

ROS is built with collaboration in mind because it is hard for a single institution to do all the work. It allows different institutions to work together and to build upon each other's progress.

There are five main principles in ROS [1]:

- Peer-to-Peer
- Tools-based (microkernel)
- Multi-language
- Thin
- Free and open source

Peer to Peer: Complex robot systems consists of several onboard computers or boards that are connected via Ethernet, and sometimes offboard computers for complex computation tasks. Peer to peer architecture together with buffer and lookup systems enables every component to communicate directly with any other, synchronously or asynchronously as required.

Multi-language: ROS is language-neutral, and can be programmed in various languages. The ROS specification works at the messaging layer. Peer-to-peer connections are negotiated in XML-RPC, which exists in various languages. To support a new language, either C++ classes are re-wrapped or classes are written enabling messages to be generated. These messages are described in IDL (Interface Definition Language).

Tools-based: ROS has microkernel which uses a large amount of small tools to build and run various ROS components. Each command used to manipulate nodes and messages in ROS is an executable. This is advantage because problem with one executable will not affect others which makes

ROS more reliable and flexible than systems with centralized runtime environment.

Thin: ROS drivers and algorithms are intended to be stored in standalone executables to combat development of algorithms that are dependent on robotics OS and therefore hard to reuse. This ensures maximum reusability and most importantly keeps ROS's size down. This makes ROS easy to use and it facilitates unit testing. The complexity is moved to libraries.

ROS also uses code from other open source projects.

II. History

ROS was originally developed in the Stanford Artificial Intelligence Laboratory as support for the Stanford AI Robot STAIR project [2].

Since then it was backed up by large amount of researchers by contributing their time and knowledge to core ROS ideas and its elemental packages.

Since 2007 it was mainly developed and maintained by a Californian company, Willow Garage. In 2013 it was transferred to the Open Source Robotics Foundation.

From the beginning, ROS's development was distributed among multiple institutions and for multiple robots. This is one of ROS's strengths since any group can start their own ROS repository on their servers and maintain full control and ownership.

At starts most of ROS users were researchers, but now the ROS community consists of thousands of users around the world, mainly in commercial sector, industrial automation and service robotics, with projects ranging from small home projects to large industrial automation systems.

The community is very active. According to metrics that can be found on www.ros.org it has more than 1,700 members on the ros-users mailing list, more than 4,000 users on the collaborative documentation wiki and around 8,500 users on community driven Q&A website. The wiki has around 14,500 pages which have 30 edits and 31,500 visits per day. The Q&A website has more than 18,000 questions with 70% answer rate. [3]

III. Structure

ROS was designed to be as distributed and modular as possible, so that users can use as much or as little of ROS as they desire.

There is large number of user-created packages that add a lot of functionality on top of the core ROS system. They range from proof-of-concept algorithm implementations to industrial-quality drivers and capabilities.

At the lowest level, ROS offers a message passing interface that provides inter-process communication and is commonly referred to as a middleware.

ROS provides:

- publish/subscribe anonymous message passing
- recording and playback of messages
- request/response remote procedure calls
- distributed parameter system

A communication system is often one of the first needs to arise when implementing a new robot application. ROS's messaging system manages the details of communication between distributed nodes via the anonymous publish/subscribe mechanism.

Another benefit of using a message passing system is that it forces implementation of clear interfaces between the nodes in system, thereby improving encapsulation and promoting code reuse. The structure of these message interfaces is defined in the message IDL (Interface Description Language).

ROS is language-independent. At this point there are three main libraries in ROS that allow programing in C++, Python and LISP which are geared toward UNIX-like systems. Beside those three libraries there are two experimental libraries making it possible to program ROS in Java or Lua.

A. Ros file system

The resources of ROS are organized into a hierarchical structure on disc. Two important concepts stand out [4]:

The package: the fundamental unit within ROS software organization. A package is a directory containing nodes, external libraries, data, configuration files and one xml configuration file called package.xml.

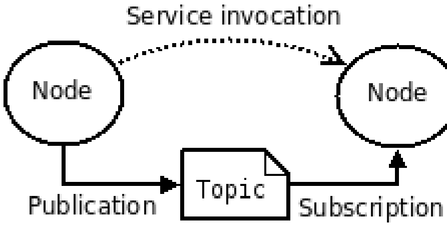
The stack: a collection of packages. It offers a set of functionalities such as navigation, positioning, etc. A stack is a directory containing package directories plus a configuration file called stack.xml.

Message (msg) types - message descriptions that define data structures for messages sent in ROS.

Service (srv) types - service descriptions that define request and response data structures for services in ROS.

B. Ros computation graph

The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways. Simple interaction between nodes, topics and services is presented in Pic. 1 [5].



Pic. 1. ROS Basic concepts

Nodes: In ROS, a node is an instance of an executable. A node may equate to a sensor, motor, processing or monitoring algorithm, and other. Every node that starts running declares itself to the Master. This comes back to the microkernel architecture, whereby each resource is an independent node.

Master: The Master is a node declaration and registration service, which makes it possible for nodes to find each other and exchange data. The Master is implemented via XMLRPC. The Master includes a heavily-used component called the **Parameter Server,** also implemented in the form of XMLRPC, and which is, as the name implies, a kind of centralized database within which nodes can store data and, in doing so, share system-wide parameters.

Data is exchanged asynchronously by means of a topic and synchronously via a service.

Topics: A topic is an asynchronous communication method used for many-to-many communication, data transport system that is based on a subscribe/publish system. One or more nodes are able to publish data to a topic, and one or more nodes can read data on that topic. A topic is, in a way, an asynchronous message bus, a little like an RSS feed. This notion of an asynchronous, many-to-many bus is essential in a distributed system situation. A topic is typed, meaning that the type of data published (the message) is always structured in the same way. Nodes send and receive messages on topics.

Messages: A message is a complex data structure which is combination of primitive types (character strings, Booleans, integers, floating point, etc.) and messages which are recursive structures.

Services: A service is a method for synchronous communication between two nodes. The idea is similar to that of a remote procedure call.

Bags: Bags are formats for storing and playing back message data. This mechanism makes it possible, for example, to collect data measured by sensors and subsequently play it back as many times as desired to simulate real data. It is also a very useful system for debugging a system after the event.

These concepts are used by the system as it is running, whereas the ROS File System is a static concept.

C. Urdf

URDF (Unified Robot Description Format) is an XML format used to describe an entire robot in the form of a standardized file. Robots described in this way can be static or dynamic and the physical and collision properties can be added to it. Besides the standard, ROS offers several tools used to generate, parse or check this format. For example, URDF is used by the Gazebo simulator to represent the robot [4]

IV. ADVANTAGE AND DISADVANTAGES OF ROS

ROS was designed to be as distributed and modular as possible so that people can decide how much of ROS they want to use.

Another key advantage of ROS is its community. ROS has grown a lot in last several years and now has worldwide community. The community is very active and people are contributing with various packages, ports and other contributions.

Good thing about ROS is that it is under standard three-clause BSD license, which is very permissive open license and people to reuse ROS in commercial and private projects.

Ros is also scalable, it can be implemented in systems with ARM CPU up to the XEON clusters.

Other advantages of ROS, as mentioned before, are that ros is peer-to-peer, multi-language, tool-based and thin.

ROS is not yet truly cross-platform. While there are libraries to use ROS with Windows, OSX and Android they are not currently fully supported.

Many of disadvantages are addressed in ROS 2.0. Some of current needs are [6]:

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: small computers, including "bare-metal" microcontrollers, should be first-class participants in the ROS environment, instead of being segregated from ROS by a device driver.

Real-time systems: real-time control should be directly supported in ROS, including inter-process and inter-machine communication.

Non-ideal networks: ROS should behave as good as possible when network connectivity degrades due to loss and/or delay, from poor-quality Wi-Fi 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, ROS-based lab prototypes should be able to evolve into ROS-based products suitable for use in real-world applications.

Prescribed patterns for building and structuring systems: ROS should provide clear patterns and supporting tools for features such as life cycle management and static configurations for deployment.

V. PORTS AND INTEGRATIONS

Ros is ported and integrated into many robots and systems, some of which are [3]:

- ABB, Adept, Motoman, and Universal Robots are supported by ROS-Industrial
- Baxter at Research Robotics, Inc.
- HERB developed at Carnegie Mellon University in Intel's personal robotics program
- Husky A200 robot developed (and integrated into ROS) by Clearpath Robotics
- PR2 personal robot being developed at Willow Garage

- rosbridge protocol and server developed by Brown University to enable any robot or computing environment to integrate with ROS using JSON-based messaging, such as for common web browsers, Matlab, Microsoft Windows, OS X, and embedded systems
- Shadow Hand – A Fully dexterous humanoid hand.
- STAIR I and II robots developed in Andrew Ng's lab at Stanford
- SummitXL - Mobile robot developed by Robotnik, an engineering company specialized in mobile robots, robotic arms and industrial solutions with ROS architecture.
- Nao humanoid: University of Freiburg's Humanoid Robots Lab developed a ROS integration for the Nao humanoid based on an initial port by Brown University
- UBR1 developed by Unbounded Robotics, a spin-off of Willow Garage.

A. Integration with other libraries

Ros provides seamless integration with various open source projects, some of which are [7]:

Gazebo - a 3D indoor and outdoor multi-robot simulator, complete with dynamic and kinematic physics, and a pluggable physics engine. Integration between ROS and Gazebo is provided by a set of Gazebo plugins that support many existing robots and sensors.

OpenCV - the premier computer vision library, used in academia and in products around the world. OpenCV provides many common computer vision algorithms and utilities that you can use and build upon.

PCL - the Point Cloud Library - a perception library focused on the manipulation and processing of three-dimensional data and depth images. PCL provides many point cloud algorithms, including filtering, feature detection, registration, kd-trees, octrees, sample consensus, and more

MoveIt! - a motion planning library that offers efficient, well-tested implementations of state of the art planning algorithms that have been used on a wide variety of robots, from simple wheeled platforms to walking humanoids.

VI. OTHER MULTI-PLATFORM OPERATING SYSTEMS AND MIDDLEWARE

Beside ROS there are few other operating systems or robot middleware worth mentioning [4]:

Microsoft Robotics Developer Studio: a multiplatform system, created by Microsoft. It is free and provides a simulation tool; however, it is only compatible with Windows and is programmed with a managed .NET language [8].

NAOQ: open source robotics system produced for the NAO robot by Aldebaran Robotics, and programmed in C++ or Python.

URBI: produced by the French company Gostai. URBI is a good multi-platform, open source and offers its own scripting language (URBIScript), although it is also programmed in C++.

VII. Implementation

At the moment only Ubuntu is officially supported, but there are also experimental solutions for OS X, Arch Linux and other platforms.

Newest release of ROS is *Jade Turtle*. There is also a LTS version, *Indigo Igloo*, which is supported for five years.

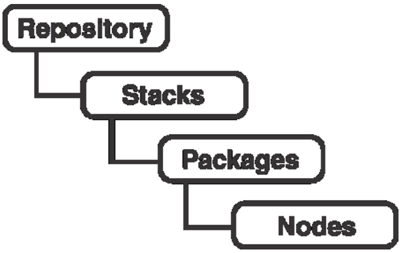
ROS can be installed in several different ways. Supported ways of installation are installing ROS from Debian packages and from source code.

The packages are more efficient than source-based builds and they are preferred way of installing ROS.

ROS can also be installed as virtual machine. There is Nootrix built VM which is Ubuntu 14.04 LTS with ROS Indigo Igloo preinstalled. It is in .ova format so it can be opened in VirtualBox or other virtualization software.

There is also a two-line installation script for installing ROS Indigo on desktop PCs. The script is tested in Ubuntu 14.04 LTS and Ubuntu 13.10 [3].

ROS code is separated hierarchically [3] (Pic. 2.).



Pic. 2. ROS code hierarchy

Repository is collection of code from certain development group.

ROS follows federated repository model, so instead of having one place for all ROS packages, users and developers are encouraged to host their own ROS packages.
Each repository can be managed and licensed by the respective maintainer who retains direct ownership and control over code.

Stack contains code of a particular device.

Stacks in their most basic form:

```
stack_name
stack_name/package_name_1
stack_name/package_name_n
stack_name/stack.xml
```

Stack is a cluster of nodes that does something useful. ROS is able to instantiate a cluster of nodes with a single command, once the cluster is described in an XML file. Sometimes multiple instantiations of a cluster are desired. For example, humanoid robots will need to instantiate two identical arm controllers. To accomplish this ROS pushes nodes and entire roslaunch cluster-description files into a child namespace, which ensures that there will not be name collisions. Basically ROS prepends a string (the namespace) to all node, topic, and service names, without requiring any modification to the code of the node or cluster. *Stack* is atomic unit of "releasing", a collection of packages for distribution [3].

Packages are separate modules that provide different services.

Packages in their most basic form:

```
package_name
package_name/Makefile
package_name/CMakeLists.txt
package_name/package.xml
```

Package is atomic unit of building and can contain anything: nodes, messages, tools, launch files, etc. The goal of packages is to provide code reusability. Package is a directory that has package manifest in it [3].

Package manifest is an XML file called package.xml which must be included. This file defines properties about the package such as package name, version, description, maintainer license and dependencies on other packages.

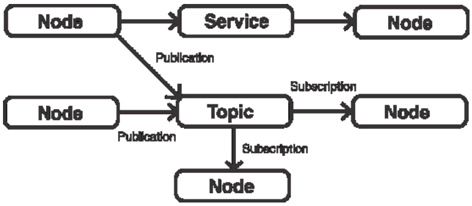
Metapackages – they are specialized packages that references one or more related packages which are loosely grouped together.

Message (msg) types – ROS uses simplified message description language to describe messages that nodes publish. A .msg files, which are stored in /msg subdirectory of a package, have two parts: fields and constants. Fields are part of data that is sent inside message while constants define useful data for interpreting those fields.

Service (srv) types **-** ROS uses simplified service description language to describe services.

It is built upon the **msg** format to enable request/response communication between nodes. Service descriptions are stored in .srv files which are located in the srv/ directory of a package.

Nodes are executables that exist in each model and they perform computations.



Pic. 3. Interconnection of nodes and their communication

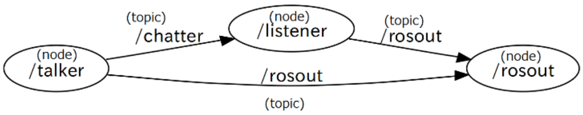
Nodes are interconnected in a graph and communicate using streaming topics, RPC services and the Parameter Server (Pic. 3.).

They can be located on different machines. ROS has several benefits from using nodes. They give additional fault tolerance because crashes are isolated to a particular node. The code complexity is also reduces and implementation details are well hidden since nodes expose minimal API to the rest of the nodes.

Nodes have a graph resource name which acts as unique identifier that can be addressed by rest of the system.

They also have a node type which are package resource names that contain node's package name and name of the node itself [3].

rxgraph is a command-line tool that is used for displaying connection graph, i.e. the ROS nodes that are currently running and also ROS topics that connect them (Pic. 4.).



Pic. 4. Connection graph of simple publisher and subscriber

Following is the information about rxgraph shown in (Pic. 4.).

Nodes:

/talker - "Hello" – sends message

/listener - "Hello" – receive message

/rosout - processes logs

Topics:

/chatter - used when talker publishes message for listener

/rosout - used to log messages to /rosout node

Result example:

```
data: hello world 290790
…
```

A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

Nodes can communicate using two protocols [3]:

-*Topics* are named buses over which nodes exchange messages and they are used for asynchronous communication.

They have anonymous publish/subscribe semantics which separates information production from its use. Nodes do not know who they communicate with. Nodes that want to acquire data subscribe to a particular topic, while nodes that generate data publish to a particular topic.

Topics are typed by Ros Message (message.msg). Directory used for topics discovery is ROS Master. Topic supports concurrent publishers and subscribers. The order of publish/subscribe messages is irrelevant.

-*Services* are defined pairs of messages, one is used for request and one for the reply.

Services are used for synchronous communication. Client calls service by sending request and then waiting for the reply. This is presented as remote procedure call. A client can establish a persistent connection to a service which enables higher performance but with the cost of less robustness to service provider changes.

Services are typed by a Ros Service (service.srv). Directory used for service discovery is ROS Master. There can be multiple simultaneous clients connected to a service.

The Roscore consists of rosmaster, parameter server and log aggregator [3].

Rosmaster provides naming and registration services to the rest of nodes. It is used for tracking publishers and subscribers to topics and services. His role is to enable individual nodes to find one another. When nodes are connected thy communicate peer-to-peer. Rosmaster uses XMLRPC API.

Parameter server is centralized parameter repository. It provides parameter access to all nodes. It is visible globally so tools can inspect configuration state of the system and modify if necessary. The Parameter server is implemented using XMLRPC and runs inside ROS Master so his API is accessible by normal XMLRPC libraries.

A. Ros networking

ROS has certain requirements of the network configuration:

- There must be complete, bi-directional connectivity between all pairs of machines, on all ports.
- Each machine must advertise itself by a name that all other machines can resolve. This is done by setting ROS_HOSTNAME environmental variable of each machine with its own name. Also, a URI of the master node should be provided. There should be only one master node, so its URI should be replicated on all machines by setting ROS_MASTER_URI environmental variable.

A virtual network needs to be created if there are firewalls, or other obstacles, between machines that use ROS to connect them. Use of Wi-Fi network is more convenient for controlling mobile robots.

VIII. Conclusion

ROS operation system provides a powerful infrastructure for developing robotic applications. It is backed by great community which contributes every day. It has potential to grow and, while it is good as it is, it has a plenty of space for improvement. It is growing fast and aims to become standard in robotic application development. Even though people are working on making ROS available on various platform, only Ubuntu is officially supported so ROS is not yet cross-platform. Future of ROS is bright and there is already work on ROS2.0 by Open Source Robotics Foundation which will address some of disadvantages of current ROS, implement new design decisions and expand ROS functionality.

ACKNOWLEDGMENT

The research in the paper is funded by the Serbian Ministry of Education Science and technological development under the grants TR-35003, III-44008. The paper is partially supported by FLIRT HORIZON 2020, funded by the European Commission's Research and Innovation Framework Program H-2020 (2014-2020) by the Marie Skłodowska-Curie actions Directorate-General for Education and Culture. European Commission under Grant Agreements No. 633398, 633369 and 633376; the paper is partially supported by the project named „High speed and high precision robots-path planning, dynamics and control (HIGH-SP ROBOTS)", Serbian-Chinese Science & Technology cooperation, Institute Mihailo Pupin and University of Anhui, School of Mechanical Engineering.

REFERENCES

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Ng "ROS: an open-source Robot Operating System" http://ai.stanford.edu/~ang/papers/icraoss09-ROS.pdf

[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] http://www.ros.org

[4] http://www.generationrobots.com/en/content/55-ros-robot-operating-system

[5] http://library.isr.ist.utl.pt/docs/roswiki/ROS(2f)Concepts.html

[6] http://design.ros2.org

[7] http://www.willowgarage.com/pages/software/ros-platform

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