AA274A: Principles of Robot Autonomy I
Course Notes

Sept 26, 2019

2 Introduction to Robot Operating System (ROS)

In this section, we discuss Robot Operating System (ROS), a popular framework for writing robot software. Unlike what its name appears to suggest, ROS is not an operating system (OS). Rather, ROS is a "middleware" that encompasses tools, libraries and conventions to operate robots in a simplified and consistent manner across a wide variety of robot platforms.

Before we start our discussion on ROS, we start with specific challenges in robot programming. Afterwards, we will look at brief history of ROS, which will shed some light on motivation to develop a brand new robot software platform from scratch. After looking into the key characteristics of ROS that made the framework so popular today, we will discuss 'pub/sub model' of ROS in greater detail. Lastly, we will explore other ROS environment features such as `catkin` and Gazebo that can make robot software development much easier.

2.1 Challenges in Robot Programming

This section is largely an excerpt from [Jos18]. Robot programming is a subset of computer programming, but it differs greatly from other software programming. Above all, robot software needs to manage a multitude of sensors and actuators on board. In other words, robot software needs to not only run the "brain" of the robot to make decisions, but also to handle multiple input and output devices at the same time. Therefore, the following features are needed for robot programming.

- *Multitask.* Given a number of sensors and actuators on a robot, robot software needs to multitask and work with different input/output devices in different threads at the same time. Each thread needs to be able to communicate with other threads to exchange data.

- *Low level device control.* Robot software needs to be compatible with a wide variety of input and output devices: GPIO (general purpose input/output) pins, USB, SPI among others. C, C++ and Python all work well with low-level devices, so robot software needs to support either of these languages, if not all.

- *High level Object Oriented Programming (OOP).* In OOP, codes are encapsulated, inherited, and reused as multiple instances. Ability to reuse codes and develop programs in independent modules makes it easy to maintain code for complex robots.

- *Availability of 3rd party libraries and community support* Ample third-party libraries and community support not only expedite software development, but also facilitate efficient software implementation.

2.2 Brief History of ROS

Until the advent of ROS, it was impossible for various robotics developers to collaborate or share work among different teams, projects or platforms. In 2007, early versions of ROS started to be conceived at Stanford AI Robot (STAIR) project with the following vision:

- The new robot development environment should be free and open-source for everyone, and need to remain so to encourage collaboration of community members.

- The new platform should make core components of robotics – from its hardware to library packages – readily available for anyone who intends to launch a robotics project.

- The new software development platform should integrate seamlessly with existing frameworks (OpenCV, SLAM, Gazebo, etc).

Development of ROS started to gain tangible traction when Scott Hassan, a software architect and entrepreneur, and his startup Willow Garage took over the project later that year to develop standardized robotics development platform. While mostly self-funded by the founder Scott Hassan himself, ROS really satiated the dire needs for a standardized robot software development environment at the time. In 2009, ROS 0.4 was released, and a working ROS robot with a mobile manipulation platform called PR2 was developed. Eleven PR2 platforms were awarded to eleven universities across the country for further collaboration on ROS development, and in 2010, ROS 1.0 was released. Many of the original features from ROS 1.0 are still in use. In 2012, Open Source Robotics Foundation (OSRF) started to supervise the future of ROS by supporting development, distribution, and adoption of open software and hardware for use in robotics research, education, and product development. In 2014, this was the first long-term support (LTS) release, ROS Indigo Igloo, became available.

In this class, we're using ROS Kinetic Kame, which is the second LTS version of ROS released in 2016. Today, ROS has been around for 12 years, and the platform has become what is closest to the "industry standard" in robotics.

2.2.1 Characteristics of ROS

Given the initial mission of ROS first conceived by STAIR project and unique challenges persistent in robot programming, ROS framework provides the following capabilities:

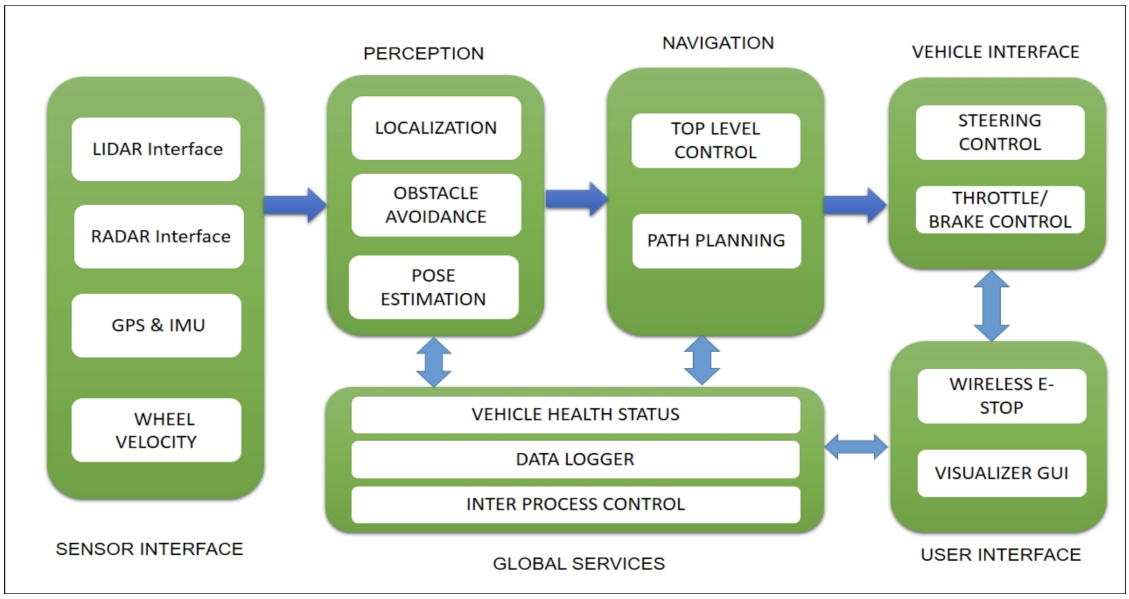

Figure 1: Modular software architecture designed to handle complexity of robot programming

- *Modularity.* ROS handles complexity of a robot through modularity: Each robot component that performs separate functions can be developed independently in units called nodes (Figure 1). Each node can share data with other nodes, and acts as the basic building blocks of ROS. Different functional capabilities on a robot can be developed in units called packages. Each package has source code, configuration files, or data files for a specific task. These packages can be distributed and installed on other computers.

- *Message passing.* ROS provides a message passing interface that allows two programs to communicate with each other. For example, a camera detects edges in the image, then the edges can be sent to the object recognition module, which can send a list of detected obstacles to a navigation module.

- *Built-in algorithms.* A lot of popular robotics algorithms are already built-in and available as off-the-shelf packages: PID (`http://wiki.ros.org/pid`), SLAM (`http://wiki.ros.org/gmapping`); and path planners such as A*, Dijkstra (`http://wiki.ros.org/global_planner`) are just a few examples. These built-in algorithms can significantly reduce time needed to prototype a robot.

- *A wide range of third-party libraries and community support.* The ROS framework is developed with pre-existing third-party libraries in mind, and most popular libraries such as OpenCV `https://opencv.org` and PCL `http://pointclouds.org` integrate simply with a couple lines of codes. In addition, ROS is supported by active developers all over the world to answer questions (ROS Answers, `https://answers.ros.org/questions/`) or to discuss various topics and public ROS-related news (ROS Discourse,

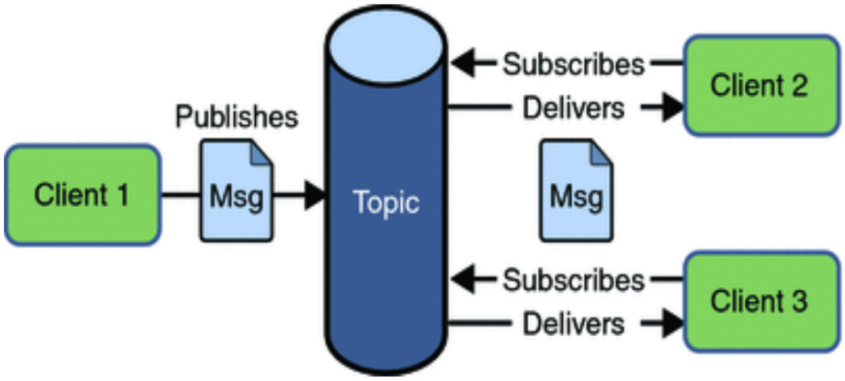

Figure 2: The pub/sub model.

`https://discourse.ros.org`).

2.3 Programming with ROS

In this section, we will discuss how to use the built-in functions in ROS to write a simple robot software. Before we look at the actual codes, we first need to understand 'Pub/Sub model' in ROS, which requires us to discuss four main components of communication in ROS: Nodes, Master, Topics and Messages.

2.3.1 Node

Definition 2.1 (Node)**.** *A node is a process that performs computation. Nodes are combined together to communicate with one another using streaming topics, RPC services, and the Parameter Server. [rosb]*

Node is the basic building block of ROS that enables object-oriented robot software development. Each robot component is developed as an individual encapsulated unit of *nodes*, which is later reused and inherited. So a robot control system typically comprise many nodes. Complexity of codes can be greatly reduced with nodes in comparison to monolithic systems.

Nodes are responsible for **pub**lishing or **sub**scribing to certain pieces of information that are shared within a virtual "chat room" (hence the "Pub/Sub model", Figure 2). In ROS, the pieces of information are called "messages", and the virtual chat room, "topics". For example, we can build an `edge detection` node to read images from a camera and detect edges from the images. The `Edge detection` node then can publish a topic called `edges`. Under the `edges` topic, a message with a `Line` object type can be published, which is could be a list of tuples containing the beginning and the end coordinates of detected lines.

2.3.2 Topics

Definition 2.2 (Topics)**.** *Topics are named units over which nodes exchange messages. [rosc]*

This means that any nodes can subscribe to a topic of interest, as much as any nodes can publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

Topics are for unidirectional, streaming communication, so communication that demands a response (i.e. a service routine) is not suitable to be implemented in topics.

As noted above, a publisher does not send messages until the topic is actually subscribed to. Use `rostopic` command line tool to monitor the messages. Below we list three most common rostopic commands:

- `rostopic list`: lists all active topics
- `rostopic echo < topic >`: prints messages received on topic
- `rostopic hz < topic >`: measures topic publishing rate

The last command is particularly useful in debugging responsiveness of an application.

2.3.3 Messages

Definition 2.3 (Messages)**.** *Nodes communicate with each other by publishing simple data structures to topics. These data structures are called messages. [rosa]*

A message comprises fields with field types and field names. The field type refers to the message type, which defines the type of information the message stores. This is easier to see in an example. Suppose we have a sensor packet node that publishes sensor data as an array of `SensorData` object. This message, called `SensorPacket`, will have the following fields:

```
myp_ros/msg/SensorPacket.msg
time            stamp
SensorData[]    sensors
uint32          length
```

Note that an empty bracket `[]` is appended to the field type `SensorData` to indicate that field is an array of `SensorType` objects.

Field types can be either the standard primitive types (integer, floating point, boolean, etc.), as well arrays of primitive types. Messages can include arbitrarily nested structures and arrays as well, as shown in the example above. Primitive message types available in ROS are listed below in Table 1. The first column contains the message type, the second column contains the serialization type of the data in the message and the third column contains the numeric type of the message in Python. [msg]

A publisher and a subscriber must send and receive the same data type to communicate. `rostopic type <topic>` can be used to view the type of message being published to a topic.

Primitive Type	Serialization	Python
bool (1)	unsigned 8-bit int	bool
int8	signed 8-bit int	int
uint8	unsigned 8-bit int	int (3)
int16	signed 16-bit int	int
uint16	unsigned 16-bit int	int
int32	signed 32-bit int	int
uint32	unsigned 32-bit int	int
int64	signed 64-bit int	long
uint64	unsigned 64-bit int	long
float32	32-bit IEEE float	float
float64	64-bit IEEE float	float
string	ascii string (4)	str
time	secs/nsecs unsigned 32-bit ints	rospy.Time

Table 1: Built-in Messages

2.3.4 Master

Definition 2.4 (Master)**.** *The master is a process that can run on any piece of hardware to track publishers and subscribers to topics as well as services in the ROS system.*

Master is responsible for assigning network addresses and enabling individual ROS nodes to locate one another, even if they are running on different computers. Once these nodes have located each other, the communication will be peer-to-peer, i.e., the master will not send nor receive the messages.

In any given ROS system, there is exactly one master running at any time. A unique feature of the master is that master does not need to exist within the robot's hardware. The master can be facilitated remotely, on a much larger and more powerful computer.

2.4 Writing a Simple Publisher and Subscriber

This section is mostly a direct excerpt from [pub].

Below is a code that creates a simple publisher. We will first introduce the code, and explain each component in more detail:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    rospy.init_node('talker', anonymous=True)
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rate = rospy.get_param('~rate',1)
```

```
    ros_rate = rospy.Rate(rate)

    rospy.loginfo("Starting ROS node talker...")

    while not rospy.is_shutdown():
        msg= "Greetings humans!"

        pub.publish(msg)
        ros_rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

We start from the first line:

```
#!/usr/bin/env python
```

Every Python ROS Node will have this declaration at the top. This line makes sure your script is executed as a Python script.

```
import rospy
from std_msgs.msg import String
```

You need to import `rospy` if you are writing a ROS Node. The `std_msgs.msg` import is so that we can reuse the `std_msgs/String` message type (a simple string container) for publishing.

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

This section of code defines the talker's interface to the rest of ROS.

- `pub = rospy.Publisher("chatter", String, queue_size=10)` declares that your node is publishing to the chatter topic using the message type String. String here is actually ROS datatype (`std_msgs.msg.String`), not python's. `queue_size` argument limits the amount of queued messages if any subscriber is not receiving them fast enough.
- `rospy.init_node(NAME, ...)` tells `rospy` the name of your node – until `rospy` has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker. NOTE: the name must be a base name, i.e. it cannot contain any slashes /".

- `anonymous=True` flag tells `rospy` to generate a unique name for the node, since ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one. This is so that malfunctioning nodes can easily be kicked off the network. This makes it easy to run multiple talker.py nodes.
- `anonymous = True` ensures that your node has a unique name by adding random numbers to the end of NAME.

```
rate = rospy.get_param('~rate',1)
ros_rate = rospy.Rate(rate)
```

`rospy.get_param(param_name, default_value)` reads a private parameter (indicated by '~') `rate` in `rospy`, and this value is used to create a `Rate` object `ros_rate` in the second line. With the help of its method `sleep()`, this offers a convenient way for looping at the desired rate. For example, if `rate` is 10, we should expect ROS to go through the loop 10 times per second (as long as our processing time does not exceed 1/10th of a second!)

```
rospy.loginfo("Starting ROS node talker...")
```

This line performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to `rosout`. `rosout` is a handy for debugging: you can pull up messages using `rqt_console` instead of having to find the console window with your Node's output.

```
while not rospy.is_shutdown():
    msg = "Greetings humans!"
    pub.publish(msg)
    ros_rate.sleep()
```

This loop is a fairly standard rospy construct of checking the `rospy.is_shutdown()` flag and then doing work. You have to check `is_shutdown()` to see if your program should exit (e.g. if there is a Ctrl-C or otherwise). In this case, the "work" is a call to `pub.publish(msg)` that publishes a string to our chatter topic. The loop calls `ros_rate.sleep()`, which sleeps just long enough to maintain the desired rate through the loop. Note that you may also run across `rospy.sleep()` which is similar to `time.sleep()`, except that it works with simulated time, not the robot time.

We now make a `talker` node to subscribe to a published topic.

```
!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(msg):
```

```
    rospy.loginfo("Received: %s", msg.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)

    rospy.spin()

if __name__ == '__main__':
    listener()
```

The code for `listener.py` is similar to `talker.py`, except we've introduced a new callback-based mechanism for subscribing to messages.

```
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
```

This declares that your node subscribes to the chatter topic which is of type `std_msgs.msgs.String`. When new messages are received, callback is invoked with the message as the first argument.

```
    rospy.spin()
```

This line simply keeps your node from exiting until the node has been shutdown.

Once we have both `talker.py` and `listener.py` ready, we can use the `catkin` build system to compile our new codes and see both nodes in action. By running the following commands

```
   $ cd  /catkin_ws
$ catkin_make
```

And starting both nodes with

```
$ python talker.py
$ python listener.py.
```

2.5 Other Features in ROS Development Environment

2.5.1 Launch files

A big portion of this section is a direct excerpt from [lau].
As a robot project grows in scale, the number of nodes and configuration files grow very quickly. *Launch file* provides a convenient way to start up multiple nodes and a master, as well as set up other configurations.

Definition 2.5. *Launch files are .launch files with a specific XML format that can be placed anywhere within a package directory to initialize multiple nodes, configuration files, and a master.*

A common practice is to create a `launch` folder inside the workspace directory to organize all your launch files.

Launch files need to start and end with a pair of launch tags:
`<launch> ... </launch>`.
This is required for all launch files.

The following syntax is used to start a node:
`<node name="..." pkg="..." type="..."/>`

- **pkg** points to the package associated with the node that is to be launched.
- **type** refers to the name of the node executable file.
- you can overwrite the name of the node with **name** argument. This will take priority over the name that is given to the node in the code.

For example,
`<node name="bar1" pkg="foo_pkg" type="bar"` launches `bar` node with a new name `bar1` from the foo_pkg package, whereas
`<node name="listener1" pkg="rospy_tutorials" type="listener.py" args="--test" respawn="true" />` launches the `listener1` node using the `listener.py` executable from the `rospy_tutorials` package with the command-line argument `--test`. If the node dies, it will automatically be respawned.

There are other attributes that can be set when starting a node, while we introduced only `args` and `respawn` in this note. `http://wiki.ros.org/roslaunch/XML/node` is a great resource on other available parameters for `<node>` tag.

2.5.2 Catkin Workspace

`catkin` is a build system that compiles ROS packages. While `catkin`'s workflow is very similar to CMake's, `catkin` adds support for automatic 'find package' infrastructure and building multiple, dependent projects at the same time, as well as supporting both C and Python. [cat]

Fun fact: `catkin` refers to the tail-shaped flower cluster on willow trees – a reference to Willow Garage where `catkin` was created.

We run catkin whenever we start a new project, or if there are any addition to packages. We do this by creating a directory called `catkin_ws`, and run the compile command, i.e.:

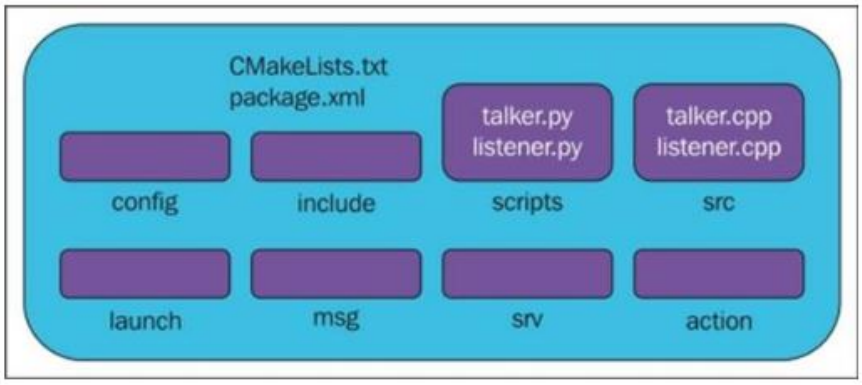

Figure 3: Components of a typical ROS package in a `catkin` workspace.

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws
catkin_make
```

Once `catkin` workspace is compiled, your directory automatically contains `CMakeLists.txt` and `package.xml`. CMakeLists file and a package.xml. There are other sub-folders in `catkin_ws` as well as shown in Figure 3, which can be changed as needed.

2.5.3 Debugging

Robot programming requires a lot of debugging. There are a few ways to debug your robot software, including (but not limited to):

- `rostopic` monitors ROS topics in the command line
- `rospy.loginfo()` starts a background process that writes ROS messages to a ROS logger, viewable through a program such as `rqt_console`.
- `rosbag` provides a convenient way to record a number of topics for playback
- `pdb` is extremely useful in debugging python scripts.

2.5.4 Gazebo

Gazebo (Figure 4) is a popular 3D dynamic simulator with the ability to accurately and efficiently simulate robots in complex environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs. Gazebo can be used in any stage of robot development, from testing algorithms to run regression testing in realistic scenarios. [gaz]
Gazebo features physics engines that enable physics in simulation to be mostly accurate, a rich off-the-shelf library of robot models, environments and sensors. [gaz] Integration of Gazebo is possible via `gazebo_ros_pkgs` package, which will be covered later in AA274A.

Figure 4: Screenshot of a scene in Gazebo

References

[cat] catkin/conceptual_overview - ros wiki. `http://wiki.ros.org/catkin/conceptual_overview`. (Accessed on 09/30/2019).

[gaz] Gazebo : Tutorial : Beginner: Overview. `http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b1`. (Accessed on 09/30/2019).

[Jos18] Lentin Joseph. *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Apress, 2018.

[lau] Launch files — ros tutorials 0.5.1 documentation. `http://www.clearpathrobotics.com/assets/guides/ros/Launch%20Files.html`. (Accessed on 09/30/2019).

[msg] Messages and topics: Communicating between nodes · fp-robotics/myp_ros wiki. `https://github.com/fp-robotics/myp_ros/wiki/Messages-and-Topics:-Communicating-between-nodes`. (Accessed on 09/30/2019).

[pub] rospy_tutorials/tutorials/writingpublishersubscriber - ros wiki. `http://wiki.ros.org/rospy_tutorials/Tutorials/WritingPublisherSubscriber`. (Accessed on 09/30/2019).

[rosa] Messages - ros wiki. `http://wiki.ros.org/Messages`. (Accessed on 09/30/2019).

[rosb] Nodes - ros wiki. `http://wiki.ros.org/Nodes`. (Accessed on 09/30/2019).

[rosc] Topics - ros wiki. `http://wiki.ros.org/Topics`. (Accessed on 09/30/2019).