

Robot Operating System (ROS)

Introduction to ROS



What is ROS?

- **Robot Operating System**
- software platform / middleware: a set of software tools and libraries to build robot applications

Definition (from: <https://wiki.ros.org/ROS/Introduction>)

ROS is an open-source, **meta-operating system** for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

What is ROS?

The 4 building blocks of ROS:

- **plumbing**: message-passing system to ensure communication between distributed nodes via an anonymous publish/subscribe pattern
- **tools** for debugging, logging, launching, introspection, visualization, etc.
- **capabilities**: broad collection of drivers and algorithms for many use cases
- **community/ecosystem**: large, diverse and global community developing and improving well documented ROS packages

ROS

=



plumbing

+



tools

+



capabilities

+



community

- **Peer-to-peer:** programs (ROS nodes) communicate over defined APIs (ROS Messages, ROS services)
- **Distributed:** ROS nodes can run on multiple devices and communicate via network
- **Multi-lingual:** existing client libraries for different languages (C++, Python, Java, Matlab, ...)
- **Free and open-source**

Filesystem Level

Covers ROS resources on disk.

- Packages
- Package Manifests
- Message types
- Service types
- ...

Computation Graph Level

Peer-to-peer network components of ROS processes.

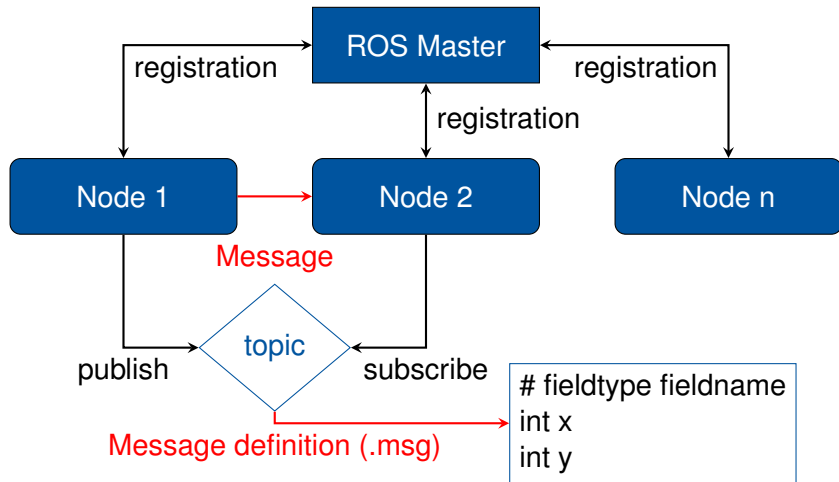
- Master
- Nodes
- Topics
- Messages
- Services
- ...

Community Level

ROS resources that enable separate communities to exchange software and knowledge.

- Distributions
- Repositories
- ROS Wiki
- ...

ROS Graph Concept



Filesystem Level Structure

```
catkin_ws
├── build
├── devel
├── src
│   ├── package_#1
│   │   ├── CMakeLists.txt
│   │   ├── package.xml
│   │   ├── src
│   │   ├── scripts
│   │   └── ...
│   ├── package_#2
│   └── package_#n
```

Now we can **start Docker** to **run** the provided **image**. The instructions how to start the group of containers can be found in the [00_GettingStarted](#) folder in the courses repository on gitlab.

During this course we work with two containers:

- **ros**: main container with ROS **distribution**: ROS **Melodic** Morenia
→ **OS: Ubuntu 18.04**
- **gui**: auxiliary container to work with graphical user interfaces (**GUI**)

The ROS Workspace Environment:

- Defines the context for the environment.
- Check which ROS environment variables are set with the following command:

```
$ printenv | grep ROS
```

- To change the workspace to a different distribution or for initial set up (here already done):

```
$ source /opt/ros/melodic/setup.bash
```

- To set the workspace permanently → write to bashrc file

```
$ echo 'source /opt/ros/melodic/setup.bash' >> ~/.bashrc
```

The ROS Workspace Environment:

- Inspect bashrc file

```
$ cat ~/.bashrc
```

- Component of the computation graph
- **Manages** the **communication** between nodes by providing naming and registration services
- When a node gets started → registers at master
- Master gets **started** with the **command**:

```
$ roscore
```

ROS Nodes

- Process that performs computation → executable program
- Single-purpose → reduces code complexity and increases fault tolerance
- Organized in packages

Running a node:

```
$ rosrun <package_name> <node_name>
```

Example (Ensure that in another terminal a ROS Master is running!):

```
$ rosrun turtlesim turtlesim_node
```

To see the opened GUI connect to port 8080
<http://localhost:8080/vnc.html> and click connect.

- Nodes exchange messages via ROS Topics
- Nodes can subscribe or or publish to topics
- usually one publisher and n subscribers
- Get a list of active topics:

```
$ rostopic list
```

- Get the contents of a topic

```
$ rostopic echo /<topic_name>
```

- Get topic information (Type, Publishers, Subscribers)

```
$ rostopic info /<topic_name>
```

- Messages get published by nodes to topics
- Messages define the data structure as
 - primitive types (integer, floating point, string, boolean),
 - or arrays
- Defined in *.msg files
- Get the type of a topic

```
$ rostopic type /<topic_name>
```

- Publishing a message to a topic

```
$ rostopic pub /<topic_name> <type> <arguments>
```

Example:

- Publishing a message to the topic Twist

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist \
-- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

- Using a node to teleoperate the turtle

```
$ rosrn turtlesim turtle_teleop_key
```

- To display the relations graphically

```
$ rosrn rqt_graph rqt_graph
```

Effective command-line interface (CLI) commands:

- **rospack**: package management tool

```
$ rospack find <package name>
$ rospack depends <package name>
$ rospack depends-on <package name>
$ rospack export <package name>
```

- **roscd**: change directory to a package w/o knowing the path

```
$ roscd <package name>
```

- **rosls**: view contents of a package

```
$ rosls <package name>
```

Inspect a message - Example

- Change directory into the *turtlesim* package

```
$ roscd turtlesim
```

- Get the content of the directory

```
$ ls msg
```

- Change into the *msg* folder

```
$ cd msg
```

- Display the content of the *Pose.msg*

```
$ cat Pose.msg
```

Setting up your catkin workspace

- *catkin* is a ROS build system to generate executables, libraries and interfaces
- We will use the *Catkin Command Line Tools*.
- To build your workspace we use the *catkin build* command

```
$ cd ~/catkin_ws  
$ catkin build
```

Setting up your catkin workspace

- Inspect the created folder and file structure in your workspace by using the *ls* command
- Overlay this workspace on top of your ROS environment and source your *bashrc* file

```
$ echo "source ~/catkin_ws/devel/setup.bash"  
>> ~/.bashrc  
$ source ~/.bashrc
```

Building a Package

Now that your *catkin* workspace is set up, you can build your first package in it.

- Change into the *src* folder of your *catkin* workspace

```
$ cd ~/catkin_ws/src
```

- The tool to build a package is *catkin_create_pkg*

```
$ catkin_create_pkg <package name> <dependencies>
```

- We name our first package *exercise_1* and use it to implement our first nodes to get familiar with concept of publishing/subscribing to a topic.

```
$ catkin_create_pkg exercise_1 std_msgs rospy roscpp
```

Building a Package

- Since we'll use the python programming language, it's good and common practice to store the python scripts into a separate *scripts* folder and not in the *src* folder.

```
$ cd ~/catkin_ws_<your student id>/src/exercise_1  
$ mkdir scripts
```

- Following the creation of the package, change directory into the *catkin_ws* to build your packages.

```
$ catkin build
```

Implementation of a Node

We'll implement a node that publishes a message to the *cmd_vel* topic of the turtle from the turtlesim package.

- Create the empty *turtle_go.py* file in the following location:
/catkin_ws_/src/exercise_1/scripts
- Copy paste the source code from the following slide into the *turtle_go.py* file.

Implementation of a Node

source code

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

def turtle_go():
    pub = rospy.Publisher('/turtle1/cmd_vel', Twist,\
                           queue_size=10)
    rospy.init_node('turtle_go', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    twist = Twist()
    twist.linear.x = 1.0
    while not rospy.is_shutdown():
        pub.publish(twist)
        rate.sleep()

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