Chair of Cyber-Physical Systems in Production Engineering
Department of Mechanical Engineering
Technical University of Munich

ROS introduction, template and exercices

denis.hoornaert@tum.de

August 19, 2021

Objective

The objective in the present lab is to introduce you to the key concepts of ROS to enable you to use it as fast as possible. The section will go through the basic concepts and the basic vocabulary, provide a simple example boasting the three types of nodes you will encounter and some ready to use templates with explanations of the different instructions used.

Disclaimer

The purpose of this laboratory (exercises and lectures) is not to teach ROS. Despite being an extremely interesting tool, in the present laboratory, ROS is simply used as a tool to accomplish a task.

3.1 What is ROS and why?

The Robot Operating System (or ROS) is a *middleware* designed for Linux systems that aims at providing a large set of useful and functional libraries targeting robotic applications. In addition to providing sensor and actuator drivers working out of the box, ROS offers a clean and user-friendly message passing system, enabling easy to use inter-process communication (or IPC).

However, note that ROS also has many drawbacks. In fact, since it sits on top of Linux, ROS is not suited for highly critical Hard Real-Time systems and applications. Specifically, the lack of task prioritisation and the big overheads in communication are the main drawbacks. This is why a second iteration of ROS named ROS2 is being developed.

3.2 Nodes

A node in ROS is basically a process. In other words, it is just like any other program in Linux, it can be ran, it can be assigned to multiple processors or constrained to just one, have multiple threads, be assigned a priority and so on. However, unlike in Real-Time Operating Systems (or RTOS), a node is not a Task spawning Jobs throughout the system runtime. ROS nodes must rather be seen as recurrent task in the sense that they are continuously active at runtime thanks to a `while` loop performing recurrently an operation. A good practice is to design each node of a ROS project with a specific and dedicated purpose (i.e. avoid spaghetti program/project).

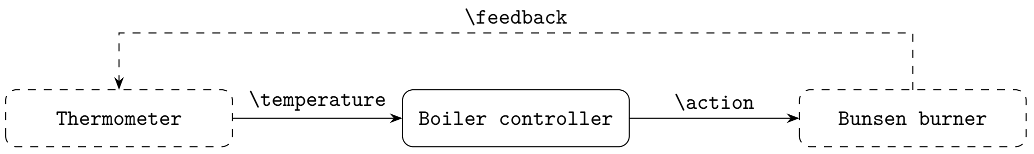

Figure 1: Simple example of a boiler system.

3.3 Topics and messages

As mentioned earlier, ROS provides easy-to-use inter-process communication that enables the previously introduced nodes composing a ROS project to communicate. Communication in ROS follows the publisher-subscriber pattern.

As its name suggests it, this pattern defines two elements: a *publisher* and a *subscriber*. The former is any process/node producing a message of a given type and publishing it on a named medium (called a *topic*). While the latter has subscribed to this named *topic* and can perceive and receive the message posted on the said *topic*. Note that, both the publishers and the subscribers only know the existence of a *topic*. Consequently, a publisher can actually publish messages on a *topic* with no subscribers. Similarly, subscribers can follow a *topic* where no messages will be posted as there is no publishers.

Such message passing systems are not specific to ROS (see dds) and are already widely used as they enable inter-process communication between programs that can be written by different programmers and/or in different languages. For instance, in ROS, inter-process communication with *topics* works seamlessly with nodes written in `C++` or `python`. Moreover, we will see in this lab that by only knowing the appropriate *topics*, we can communicate with the simulator eventhough we do not know how the latter works or has been implemented.

Remark:

All the message types provided by ROS can be found:

- `http://docs.ros.org/melodic/api/std_msgs/html/index-msg.html`
- `http://docs.ros.org/melodic/api/sensor_msgs/html/index-msg.html`

The messages that matters for our use case are listed and further detailed later.

3.4 Basic functions

This section is a list of the basic functions and instructions your nodes (publishers, subscribers or a combination of the two) must or might need.

3.4.1 Imports

Importing the proper libaries is essential. Quite obviously, you always need to import ROS to have access to all the following functions.

```
#include "ros/ros.h"
```

Alongside this, you also need to import the definitions of the ROS messages you will need. In the present lab, we will need the following imports. Make sure not to import unnecessary message types!

```
#include "std_msgs/String.h"
#include "std_msgs/Header.h"
#include "ackermann_msgs/AckermannDriveStamped.h"
#include "ackermann_msgs/AckermannDrive.h"
#include "sensor_msgs/LaserScan.h"
```

3.4.2 Node setup

The very first line of your main function should be the one here under. Its aim is to inform ROS that the program is a ROS node. This function takes as arguments the `argc` and `argv` of the main function and a String setting the node's name. You have to make sure here that you respect the lab's naming convention (i.e. `username_node_name`)!

```
ros::init(argc, argv, "Node name");
```

Just after, you should also instanciate a `NodeHandle` object that will enable you to create subscribers and publishers.

```
ros::NodeHandle node;
```

3.4.3 Recurrent task

In order to make recurrent tasks release jobs at a periodic rate. ROS enables you to specify that rate thanks to the Rate obejct (here: `loop_rate`). The loop rate takes in parameters the frequency at which the task will be executed.

```
ros::Rate loop_rate(frequency);
```

The `sleep` method of the `Rate` object forces the program to sleep until the period derived from the frequency set in `loop_back` is reached. Note that, the computation time required by the task is also taken into account such that the real period is applied rather than the execution time plus the period.

```
loop_rate.sleep();
```

If your node does not produce any data and just receives messages from its subscriptions, then, you want it to stay alive without having to create an actively waiting node (i.e. a while-true-if). In this situation, you will call the `spin` function at the end of your main such that the node can still receive messages and act upon their reception by calling the appropriate callback function(s).

```
ros::spin();
```

The `ros::ok()` function is generally used to check the state of the program. It enables ROS to see a kill signal (that you might produce with `CTRL+C`) and terminate. This is why this function is often called within the condition block of the `while(true)` loop (see line 15 in publisher code later on). Without this function, you might not be able to exit a program with an infinite loop.

```
ros::ok()
```

In the case of a recurrent task where you have a loop with the `ok` function, you want to spin to check whether callback functions need to be called (because the node has received messages). The difference between `spin` and `spinOnce` is that the former is a blocking instruction whereas the latter only looks once for the potention callbacks.

```
ros::spinOnce();
```

3.4.4 Subscribing

For a single node, one can instantiate multiple subscribers and attach them to a `Nodehandle`. A subscription called `sub` can be instantiated with the instructions here under. Where `"topic"` is the name of the topic it subscribed to, the topic is 10 slots long and, upon the reception of a message, the callback function called `callback` is executed.

```
ros::Subscriber sub = node.subscribe("topic", 10, callback);
```

Defining a callback function is as easy as defining any C/C++ function. However, there are few requirements. The first being that the function cannot return any value (i.e. it returns `void`). If needed, a simple work around is to store the value you wish to return in a global variable (or a class attribut if the callback function is a method of the class). Secondly, the parameter of the callback function must be constant pointer reference of the topic type. In our example, the topic type being used is `std_msgs::String`.

```
void callback(const std_msgs::String::ConstPtr& msg) {
    ROS_INFO("Recv: %s", msg->data.c_str());
}
```

3.4.5 Publishing

For a single node, one can instantiate multiple publishers and attach them to a `Nodehandle`. For instance, the instruction here under instantiates a publisher called `pub`, that will publish `String` messages in a 10 slots long topic called `topic`. Declaring publishers should be done either globally, as a class attribute or at the beginning of the main if not needed somewhere else.

```
ros::Publisher pub = node.advertise<std_msgs::String>("topic",
  ↪ 10);
```

For every scope where the publisher `pub` is visible, you can send a message using the method `publish()`. The message passed in parameters must be of the same type (or a children class) as the one set in the publisher decalration. Here, `msg` is a `std_msgs::String` object.

```
pub.publish(msg);
```

3.4.6 Tools

ROS provides an alternative way to print data on the terminal. The `ROS_INFO` function is one of them. This function works exactly like `printf` would in ANSI C. For instance:

```
ROS_INFO("Message received: %s", msg); // prints a C string
ROS_INFO("Person age: %i", msg); // prints an integer
```

3.5 Publisher-subscriber with ROS - example

Let us consider this adapted example coming from the ROS website (see subsection 3.7 "Useful links"). The present case is extremelly simple: there is one publisher sending strings on a topic called `"topic"`. On the other side, there is a subscriber in charge of printing the messages posted on the topic.

3.5.1 Publisher

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv) {
    ros::init(argc, argv, "Publisher");
    ros::NodeHandle node;

    ros::Publisher pub = node.advertise<std_msgs::String>("topic", 10);
    ros::Rate loop_rate(5);

    unsigned int i = 0;
    char message[50];
    std_msgs::String msg;

    while (ros::ok()) {
        printf("Enter name: ");
        scanf("%s", message);
        msg.data = message;
        ROS_INFO("%s", msg.data);
        pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        i++;
    }

    return 0;
}
```

3.5.2 Subscriber

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"

void callback(const std_msgs::String::ConstPtr& msg) {
    ROS_INFO("Recv: %s", msg->data.c_str());
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "subscriber");
    ros::NodeHandle node;
    ros::Subscriber sub = node.subscribe("topic", 10, callback);
    ros::spin();
    return 0;
}
```

3.6 Template

Based on the two example before, we can draw the templates for both publisher nodes and subscriber nodes.

3.6.1 Publisher

```
#include "ros/ros.h"
...
include the relevant ros message libraries
...

int main(int argc, char **argv) {
    ros::init(argc, argv, "Publisher name"); # declare the node
    ros::NodeHandle node; # create a node handle

    ...
    Declare publishers and/or subscribers
    ...

    ros::Rate loop_rate(5); # set the rate

    ...
    Declare variables
    ...

    while (ros::ok()) {
        ...
        Do stuff: compute, publish, and so on
        ...
        ros::spinOnce(); # look if any callback has to be called
        loop_rate.sleep(); # wait for the end of the period
    }

    return 0;
}
```

3.6.2 Subscriber

```
#include "ros/ros.h"
...
include the relevant ros message libraries
...

...
Define all your callback functions
...

void callback(const std_msgs::String::ConstPtr& msg) {
    ...
    Act on the message reception
    ...
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "subscriber name"); # declare the node
    ros::NodeHandle node; # create a node handle
    ...
    Do stuff
    ...
    ros::Subscriber sub = node.subscribe("topic_name", 10, callback); # define the subscribers
    ros::spin(); # keep the node alive and look after callback
    return 0;
}
```

3.7 Useful links

- http://wiki.ros.org/ROS/Tutorials
- http://docs.ros.org/melodic/api/std_msgs/html/index-msg.html
- http://docs.ros.org/melodic/api/sensor_msgs/html/index-msg.html

---

Setting up a ROS project

3.7.1 Creating a ROS project/work-space

Creating a ROS project can be done in few simple steps. These steps must only be followed once per project creation.

1. Source the ROS environments and tools with the following command. (it must be done at each system restart)

```
source /opt/ros/lunar/setup.bash
```

2. Create a ROS workspace folder (line 1) and initialized the project (line 3). Note that the project does not have to be located in the root folder (line 1).

```
1 mkdir -p ~/cps-lab-workspace/src
2 cd ~/cps-lab-workspace/
3 catkin_make
```

3. Source the workspace environment variables and tools.

```
source devel/setup.bash
```

4. Check that both ROS and package are setup correctly. If this command does not print anyhting, repeate the previous manipulations.

```
echo $ROS_PACKAGE_PATH
```

3.7.2 Creating a package, creating nodes and launching them

Remark:

Note that, in the following commands, `<package>` and `<node>` are parameters that the student must replace when creating new projects. Typically, `<package>` denotes the name of the package and `<node>` the name of the node.

1. Create the package.

```
cd src/
catkin_create_pkg <package> std_msgs rospy roscpp
```

2. Adapt the `CMakeList.txt` to your project

```
cmake_minimum_required(VERSION 2.8.3)
project(<package>)

find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs
    ↪  genmsg)

generate_messages(DEPENDENCIES std_msgs)

catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})

# IMPORTANT: you must copy and adapt the following lines for
    ↪ each node that you want!
add_executable(<node> src/<node>.cpp)
target_link_libraries(<node> ${catkin_LIBRARIES})
add_dependencies(<node> <package>_generate_messages_cpp)
```

3. Re-build project with the new package.

```
cd ../
catkin_make
```

4. Place any node source code in the package's `src/` folder. For instance:

```
cp publisher.cpp src/<package>/src/
```

5. Create a launch folder in the target package.

```
cd src/<package>/
mkdir launch/
```

6. Create a launch file and open it with, for instance, `nano`

```
nano launch/<node>.launch
```

7. In the launch file just created, write the description of the node following this template

```
<launch>
  <group ns="<package>">
    <node pkg="<package>" name="<node>" type="<node>" output=
       ↪ "screen"/>
  </group>
</launch>
```

8. Launch the node created with the follwing line

```
roslaunch <package> <node>.launch
```

Exercices

The following set of exercices aims at helping the student to better understand how the publisher-subscriber communication scheme implemented in ROS works. As presented previously in this document, the communication scheme allows for a wide range of network topology and independence between the nodes. This set of excercices will provide training for the most common connections.

Disclaimer:

The following exercices only have a pedagogical vocation. In other words, they represent interesting but not realistic use-cases. Better and more optimized topologies can be designed!

3.8 Basics, none-to-1 and 1-to-none

After having read carefully the code examples provided in Section 3.5, copy the content of listing 3.5.1 in a file called `publisher.cpp` and copy the content of listing 3.5.2 in a file called `subscriber.cpp`. Thereafter, using the following commands, launch the two nodes.

```
roslaunch exercice1 publisher.launch
roslaunch exercice1 subscriber.launch
```

Does the outcome matches your expectations?

Once the two nodes have been started together successfully, terminate both and experiment the following:

1. Launch the subscriber node (i.e., `subscriber.launch`) alone. What happens? Is anything being displayed? Start the publisher node (i.e., `publisher.launch`). How is the subscriber node behaving?
2. Perform the opposite manipulation. In other words, start the publisher node alone, observe its behaviour and then, launch the subscriber node.
3. How do these two mamipulations differ?

3.9 1-to-1

Create one new publisher node and one new subscriber node (respectivelly called `receiver.cpp` and `emitter.cpp`). Link this two nodes with a topic called `odd` of type `integer` (see `http://docs.ros.org/melodic/api/std_msgs/html/index-msg.html` for more information). Program the `emitter` such it send the integer 42 every seconds. The `receiver` must continously wait for new messages being posted on the topic and print the number when received.

3.9.1 Altering the behaviour

Alter the behaviour of the `emitter` such that it generates and sends a randomly generated integer every 0.5 second if and only if the integer is odd.

3.9.2 Expanding the node relationship

Change the behaviour of the `receiver` such that it counts the amount of messages received and prints this count each time a new message is received. Additionally, apply the following changes:

- change the type of the topic `odd` to boolean
- create a new topic called `even` of type boolean
- alter the `emitter` behaviour such that it sends *True* on the `even` topic when the randomly generated numbers that are even. Do the same for odd numbers.
- the `receiver` must print the count of odd and even numbers everytime a message is received on one of the two topics

3.9.3 Optimization

Can you think of any optimization? If yes, which one and what will it change?

3.10 1-to-many

Create a publisher node called `user_input.cpp` that will be in charge of waiting for the user inputs through the keyboard. The node must accept integers from the user and must send the integer to a topic called `user_integer`. In addition to the publisher node, we ask the student to write a subscriber node (called `arith_avg.cpp`) that will be in charge of receiving messages posted on the topic and compute the **arithmetic** average (see Equation 1) of all the integers received. Each time a new integer is received, the subscriber must compute the average and print it.

3.10.1 Multiple subscribers

Using the `arith_avg` node as a model, create three extra nodes in charge of computing respectivelly the **geometric** average (see Equation 2), the **harmonic** average (see Equation 3), and the meadian. They should all subscribe to the `user_input` topic and print their formula outcome upon the reception of a new integer!

As a reminder, the formulas to compute the different averages are the following:

$$\bar{x} = \frac{1}{N}\left(\sum_{i=0}^{N} x_i\right) \quad (1) \qquad \bar{x} = \left(\prod_{i=0}^{N} x_i\right)^{\frac{1}{N}} \quad (2) \qquad \bar{x} = \frac{N}{\sum_{i=0}^{N} \frac{1}{x_i}} \quad (3)$$

3.11 1-to-many-to-1

Buidling on top of the previous exercice (i.e., 3.10), create for every subscribing node a new topic on which the node will publish its result instead of printing it. In addition, create a subscriber node that will subscribe to all the newly created topics. This node must print the latest values posted on each topic upon the reception of any message.

3.11.1 Asynchronous subscriber

Alter the behaviour of the subscriber such that it prints the latest values received every 0.5 seconds.