Embedded systems: design, specification, models and architecture
ROS, GenoM, FIACRE, Hippo & Time Petri Net

Félix Ingrand, Janette Cardoso (ROS), Pascal Chauvin (ROS, Gazebo, VMWare)

2021-2022

Contents

1 Introduction, context, models and tools

This **graded** exercice/tutorial aims at a quick repracticing of ROS, the use of GenoM, FIACRE, Hippo etc. Questions are asked as we go along and each student has to answer them on a sheet of paper and hand it in at the end of the session.

The objective is to (re)familiarize oneself with ROS (Robot Operating System), to approach ROS2, but also GenoM (largely seen in class). The simulations will be done using Gazebo. All the necessary tools are installed in a VMWare VM (Virtual Machine) under Ubuntu 20.04 LTS (Focal Fossa).

1.1 Virtual Machine

The operating system is a Linux system (Ubuntu 20.04 LTS), which has the same functionalities as those found on various robots (those presented in class). From your account you will be able to access this virtual machine. To do so, you will use the virtualization software VMWare Player.

You must start the PC under Windows. At the Windows 10 prompt, log in with your credentials. Launch the VMWare software (in demonstration/education mode), and add a new virtual machine. If you are in the second group, the image will probably already be in the list, If not, look for the `Ubuntu 20.04 Up` image which must be on the disk `D:` in the folder: `D:\Images VMWare`. At the first launch, you will be asked if the machine has been moved or copied, answer copied.

Once the VMWare machine is launched and booted, you can connect with the following credentials.

```
login: felix (TP UPSSITECH)
password: souris
```

Once the desktop appears, you can switch the Ubuntu virtual machine to full screen. It is possible that the type of keyboard is not correct (French AZERTY or English QWERTY), in which case, select the right keyboard in the top right menu bar (either `En` or `Fr`).

The whole work is done either with already installed software, or with software that you will install.

All installations/modifications are done in the `~/work` folder.

1.2 ROS - middleware

ROS (Robot Operating System) provides libraries and tools to help software developers create robotic applications. It provides hardware abstraction, device drivers, libraries, data viewers, message passing, packet management, and more. ROS is licensed under an open source, BSD license.

For more information: http://wiki.ros.org/ROS/StartGuide

For tutorial: http://wiki.ros.org/ROS/Tutorials

By default, the ROS installation is located in:

```
/opt/ros/$ROS_DISTRO
```

For Ubuntu 20.04, the recommended ROS version is `noetic` (so in `/opt/ros/noetic`). When you install ROS binary packages, they are installed in the various directories found in this location. The account you are using is already configured to use this version of ROS.

1.3 ROS2 - middleware

There is now a new major version of ROS: ROS2. We have quickly seen it in class, and during this tutorial we will see some of the commands of this new version.

From the user's point of view, the most visible difference is the absence of `roscore` which it is no longer necessary to run before each node. But the basic concepts remain the same (topics, services, actions, etc).

For more information: https://docs.ros.org/en/foxy/index.html

For tutorial: https://docs.ros.org/en/foxy/Tutorials.html

For Ubuntu 20.04, the recommended version of ROS2 is `foxy` (so in `/opt/ros/foxy`).

1.4 Gazebo and RVIZ

Gazebo is a simulator which has an existence independently of ROS (see the simulation of drones in the section 8. It allows to precisely simulate robots (with a physical engine), and is very well interfaced with ROS: https://www.gazebosim.org

RVIZ is a ROS topics viewer. It allows to display in real time many ROS objects: http://wiki.ros.org/rviz, and thus to follow graphically these values (Laser Scan, PCL, odometry, wrench, twist, etc). This is not a simulator...

2 T.P.1 A simple example : Producer Consumer (ROS)

Before doing anything, in a terminal, type:

```
cd
cd work
./be_clean.sh
```

Two versions of ROS will be used in this TP. ROS1 and ROS2, note that the two versions can cohabit on the same machine, providing various environment variables are correctly assigned.

Two files in `~/work`, `choose_ros1` and `choose_ros2` allow you to choose the version of ROS you want to use. These files are not scripts to be executed, but need to be sourced with the command `source`.

```
source choose_ros1
```

ou bien

```
source choose_ros2
```

In a new shell, by default, ROS version 1 is configured.

We will start from an existing model, with two "nodes", a producer, and a consumer, see appendix 11, listing 3 and 2. The objective here is to become familiar with ROS

2.1 Creating a work space and program compilation

Open a terminal, (Ctrl-Alt-T), and type the following command:

```
cd
mkdir -p ~/work/catkin_ws/src
cd ~/work/catkin_ws/src
catkin_init_workspace
git clone git://redmine.laas.fr/laas/users/felix/isae-cnam/prod_cons_package.git
cd ..
catkin_make
```

Listing 1: Create a catkin workspace

2.2 Execution

The compiled nodes are in the `devel/lib/<name_package>` directory of the workspace, and so are the services, launch files, scripts, etc (for this example: `<package_name> =prod_cons_package`).

Scenario 2.1 : launch in order the roscore, `producteur_node, consommateur_node`.

Terminal 1:

```
roscore
```

When launched, you can leave the `roscore` running until the end of this session.

Terminal 2:

```
cd ~/work/catkin_ws
source devel/setup.bash
rosrun <package_name> <node_name>
```

Terminal 3:

```
cd ~/work/catkin_ws
source devel/setup.bash
rosrun <package_name> <node_name>
```

In the consumer terminal, you will see: `[ INFO ] [15991.....] : I heard: [1]`

2.3 More ROS commands

ROS provides some commands to answer the questions below (see ROS cheatsheet, section 9 or the course slides). For each question, list the ROS command and the answer.

Questions for evaluation: R1 (give answers R1.1, R1.2, etc., on the provided sheet)

1. How many ROS nodes does the project have? Give their names. Run the command `rqt_graph` to see the nodes running!
2. What data is being exchanged?
3. What is its types?

4. How often is the data exchanged? Use the `rosbag` command to record the topic `chatter`.
5. Suspend the producer (with Ctrl-Z), and use this same command `rosbag` to replay this topic in place of the producer. Refresh the `rqt_graph` window. What do you notice?
6. What happens if you simultaneously play the ros bag and wake up the producer (command `fg`)?
7. What happens if you kill the roscore program while both nodes are running?

2.4 Some ROS2 commands

In order to familiarize yourself with the new ROS2 version, you will run the same example as the previous section in ROS2 (see ROS2 cheatsheet, section 10).

First switch to ROS2:

```
source ~/work/choose_ros2
```

The `ros2_ws` folder contains some sample ROS2 code, including the consumer/producer. Compile the packages that are there with the command.

```
cd ~/work/ros2_ws
colcon build
```

`colcon` is the new build system used by ROS2.

All ROS2 commands starts with `ros2`, e.g.: `ros2 <command>`.

In a terminal, type the commands:

```
cd ~/work/
source choose_ros2
source ros2_ws/install/setup.bash
ros2 run cpp_pubsub talker
```

In another terminal:

```
cd ~/work/
source choose_ros2
source ros2_ws/install/setup.bash
ros2 run cpp_pubsub listener
```

In a third one:

```
cd ~/work/
source choose_ros2
source ros2_ws/install/setup.bash
ros2 node list
ros2 node info /minimal_publisher
ros2 topic list
ros2 topic info /topic
ros2 topic echo /topic
ros2 topic hz /topic
etc...
```

Stop both nodes, and in two different terminals, type:

```
ros2 run turtlesim turtlesim_node
```

```
ros2 run turtlesim turtle_teleop_key
```

In a third terminal, using the commands `ros2 {node, topic, service, action, interface}` reply to the following questions:

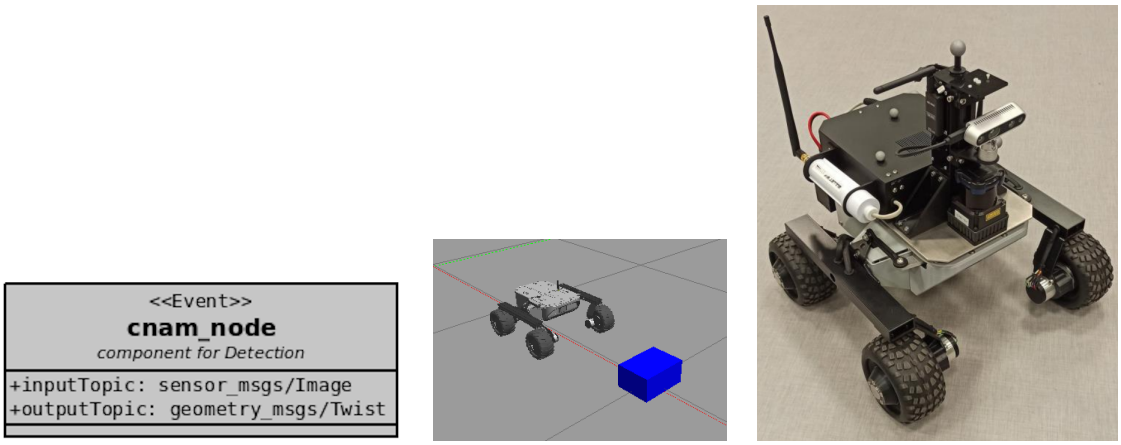
Questions for evaluation: R2 (give answers R2.1, R2.2, etc.)

1. How many ROS nodes does the project have? Give their names.
2. By inspecting the nodes, what is the topic that is used to transmit the speed instruction from one node to another?
3. What is its type?

4. How many ROS services are served by the node `/turtlesim`?
5. An action is defined between the two nodes, what is its name?
6. Which node is the server for this action?
7. What are the goal, result and feedback arguments of this action?

3 T.P. 2: Gazebo - ROS Visual target tracking

Go back to ROS1 (execute `source ~/work/choose_ros1`) and if needed, launch `roscore` again in its own terminal.



a) ROS node b) Gazebo node c) the Leo Rover at l'ISAE

Figure 1: Colored target tracking robot.

The purpose of this part is to show the interaction between the Gazebo simulator and the ROS node, and to understand the commands used to allow this interaction. Gazebo is used to simulate a robot (Leo Rover) and a target represented by a "blue brick" (fig. 1.b). The ROS node called `cnam_package`, represented in fig. 1.a, tracks the target using an image as input. It computes the barycenter of the image, then sends commands to the robot (Leo Rover) to track this target. The Gazebo simulator is seen as a ROS node.

The software architecture of the different existing components is represented on the fig. 2.

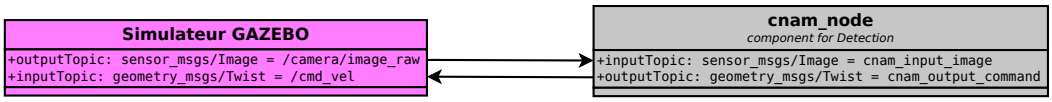


Figure 2: Gazebo (simulation) + ROS

The same ROS node `cnam_package` is used with the Gazebo simulator (fig. 2.a) and with the (real) robot Leo Rover (fig. 2.b). It is thus possible to test and validate this component with the simulator, then to use it with the real robot.

3.1 Analyse des composants de l'architecture : noeud `cnam_package`

The component code (node `cnam_package`, fig. 1.a.) is shown in the Listing 4 in the section 11. Get this component and compile it with the following commands:

```
cd ~/work/leo_rover_ws/src
git clone git://redmine.laas.fr/laas/users/felix/isae-cnam/cnam_package.git
cd ..
catkin_make
```

Make sure that `roscore` is still active (if not, restart it in its own terminal window), then launch the experiment with the bash script:

```
source devel/setup.bash
./src/cnam_package/start.sh
```

This should launch a window that is divided into three sub-windows: one for the Gazebo simulation, one for the ros node, and one for the ROS node which allows you to move the blue brick on the screen with the following commands (it is better to use the strafing mode).

```
Reading from the keyboard and Publishing to Twist!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit
```

Now look at the code of this node (Listing 4) and find this information; in particular check:

- sensor declaration (line 6), output topic (line 11),
- the codels (line 14),
- publication (line 19, 97, 116, 51) and subscription (line 112, 97).

Questions for evaluation: R3 answer each R3.1 R3.2, etc

1. How many ROS nodes does the project have? Give their names. Run the command `rqt_graph` to see the nodes running!
2. What data is exchanged?
3. What are their types?
4. How often is the data exchanged?
5. Which mechanism allows the component to react to new topic?
6. How is it declared?
7. What is the command to execute before being able to publish a topic?

4 T.P.3: GenoM: Visual tracking

The goal of this exercise is to use a GenoM module (using the ROS middleware template) to control the robot and to track the colored target (just like you did in the previous exercise).

We use exactly the same setup as the previous exercise (Gazebo simulator), only the ROS node will be replaced by a GenoM module (see Figure 3). The module `CT_robot` will read the `/camera/image_raw` topic (with the GenoM port `ImagePort`) and will produce the `/cmd_vel` topic (with the GenoM port `CmdPort`).

4.1 GenoM `CT_robot` component

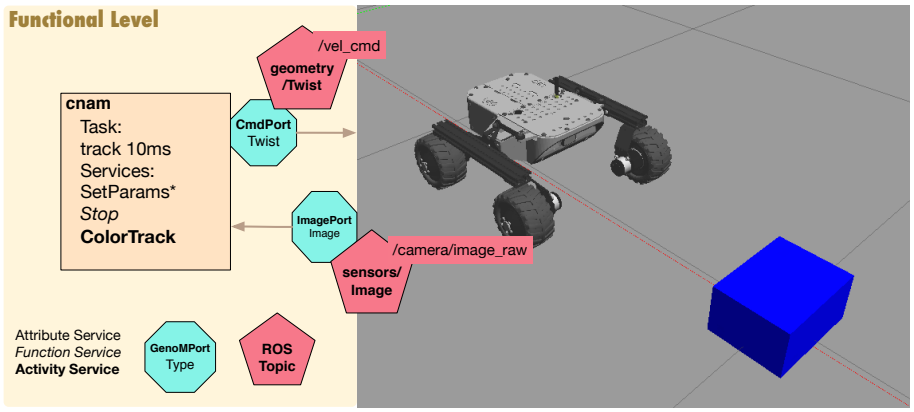


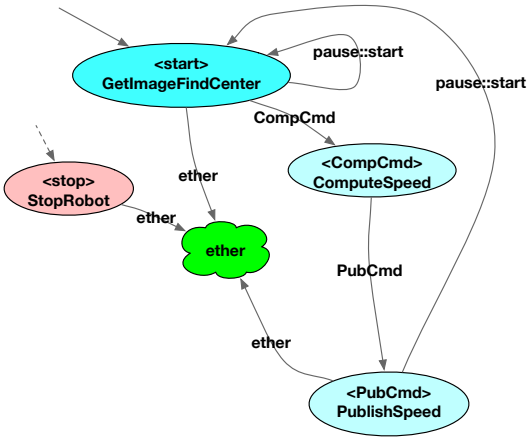
Figure 3: GenoM component connected to the Gazebo robot simulation.

The `CT_robot` component on which you intervene must be downloaded . In a terminal (ctrl + alt + T), go in the `/home/felix/work` directory and enter the commands:

```
$ cd ~/work
$ git clone git://redmine.laas.fr/laas/users/felix/isae-cnam/isae-cnam-be.git isae-upssitech-be
$ cd isae-upssitech-be
$ more CT_robot.gen
```

You will find a copy of the contents of the module specification in the file `CT_robot.gen` (see Listing 5 in the appendix 11 page 25).

Examine this file in detail... Go through the specifications of the `ids` (each of the variables), the ports (`ImagePort` and `CmdPort`) (which will be respectively mapped in the ROS topics, `/camera/image_raw` and `/cmd_vel`), the task `track`, its period, the setters (`attribute` services which allow to change the values of the variables of the `ids`), of the `function` services, of the `activity` service `ColorTrack` and of its automaton (below).



The speed reference is first computed and stored in the `cmd` field of the `IDS` and then its is exported into the `CmdPort` port.

For the `ColorTrack` activity service automaton, inspect each line `codel <status> call_function_C_C++(...) yield s` and examine the parameters and their direction. For example:

```
codel <start> GetImageFindCenter(port in ImagePort, ids in my_r, ids in my_g, ids in my_b,
          ids in my_seuil, ids out x, ids out y, ids out width,
          ids out height, ids in verbose)
           yield pause::start, // no new image, wait next cycle
                 CompCmd,     // found it
                 ether;       // in case of error.
```

The `ImagePort` port, and the `ids`, `my_r`, `my_b`, `my_seuil` and `verbose` the IDS fields are taken as input (`in`), and are not modifiable by this codel, whereas the `x`, `y`, `width` and `height` from the IDS are modified (`out`) by this codel. Note that the specification of the prefix `port`, `ids` or `local` (to indicate where the parameter is located) is, if

unambiguous, optional. Note that the code `GetImageFindCenter` can lead (`yield`) to three different states `start` (with a `pause`), `CompCmd` and `ether` (in case of exception).

You will also find the `codels` folder which respectively contains the files `CT_robot_codels.cc` and `CT_robot_track_codel` containing the codels of `CT_robot.gen`: those defined respectively in the control task, and those defined for the services (i.e. `ColorTrack`) of the execution task `track`). Browse these two files in order to identify the C/C++ functions corresponding to the codels, the call arguments (which are automatically synthesized), etc. Browse the automaton of `ColorTrack`, and go identify and analyze the corresponding codels in `CT_robot_track_codels.cc`.

Questions for evaluation: G1 (answer on the sheet G1.1, G1.2, etc.).

1. What is the direction (`in` or `out`) of the port `CmdPort`?
2. What is the direction of the `ImagePort` port?
3. What is the frequency of the `track` task?
4. Which service is interrupted by the `function Stop ()` service?
5. Which C/C++ codel and function is then called in this service? Where is it defined? What does it do?
6. At which frequency will the codels of the activity `ColorTrack` service be called when it is active?
7. Should/could we have several `ColorTrack` services running simultaneously? why?
8. Which state(s) can lead to the `start` state?
9. What is/are the `out` parameter(s) of the `ComputeSpeed` codel?
10. Where are located the `out` parameter(s) (`ids`, `port`, or `local`) of the `ComputeSpeed` codel?

4.2 GenoM: synthesis and compilation of the component

Synthesize the module with the ROS templates. In the`isae-upssitech-be` directory, execute:

```
$ autoreconf -vif
$ mkdir build
$ cd build
$ ../configure --prefix=/home/felix/work --with-templates=ros/server,ros/client/c,ros/client/ros
```

Compile and install with:

```
$ make install
```

4.3 Launching the experiment

Inspect the content of the `start.sh` bash script file in the `isae-upssitech-be` directory.

```
#!/bin/bash
mkdir -p log
genomixd -v -v > log/genomixd-‘date +"%Y%m%d-%H%M%S"‘.log &
tmux \
    new-session "roslaunch leo_gazebo leo_gazebo_brick.launch" \; \
    split-window -p 80 "script -f -c ’CT_robot-ros /CT_robot/CmdPort:=/cmd_vel; bash’ log/CT_robo
t-ros-‘date +"%Y%m%d-%H%M%S"‘.log" \; \
    split-window -p 75 "script -f -c \"eltclsh ; ./end.sh\" log/eltclsh-‘date +"%Y%m%d-%H%M%S"‘.l
og" \; \
    split-window -p 66 "rosrun teleop_twist_keyboard teleop_twist_keyboard.py /cmd_vel:=/brick_cm
d_vel" \; \
    selectp -t 2
```

Note the call to the GenoM ROS component `CT_robot-ros` with the remapping of the `/CT_robot/CmdPort` topic to `/cmd_vel`.

Check the content of the `start.tcl` file:

```
package require genomix
genomix::connect
genomix1 rpath /home/felix/work/lib/genom/ros/plugins/
genomix1 load CT_robot

```

```
proc init {} {
    CT_robot::SetVerbose 1
    CT_robot::Set_my_r 3
    CT_robot::Set_my_g 2
    CT_robot::Set_my_b 105
    CT_robot::Set_my_seuil 40
    CT_robot::connect_port {local ImagePort remote /camera/image_raw}
}
```

Note the `init` TCL procedure which calls the various services `Set_...` with their arguments as well as the service `connect_port` which connects the `ImagePort` port to the `/camera/image_raw` ROS port/topic.

Make sure that `roscore` is still active (if not, restart it in its own terminal window), then launch the experiment with the command

```
$ ./start.sh
```

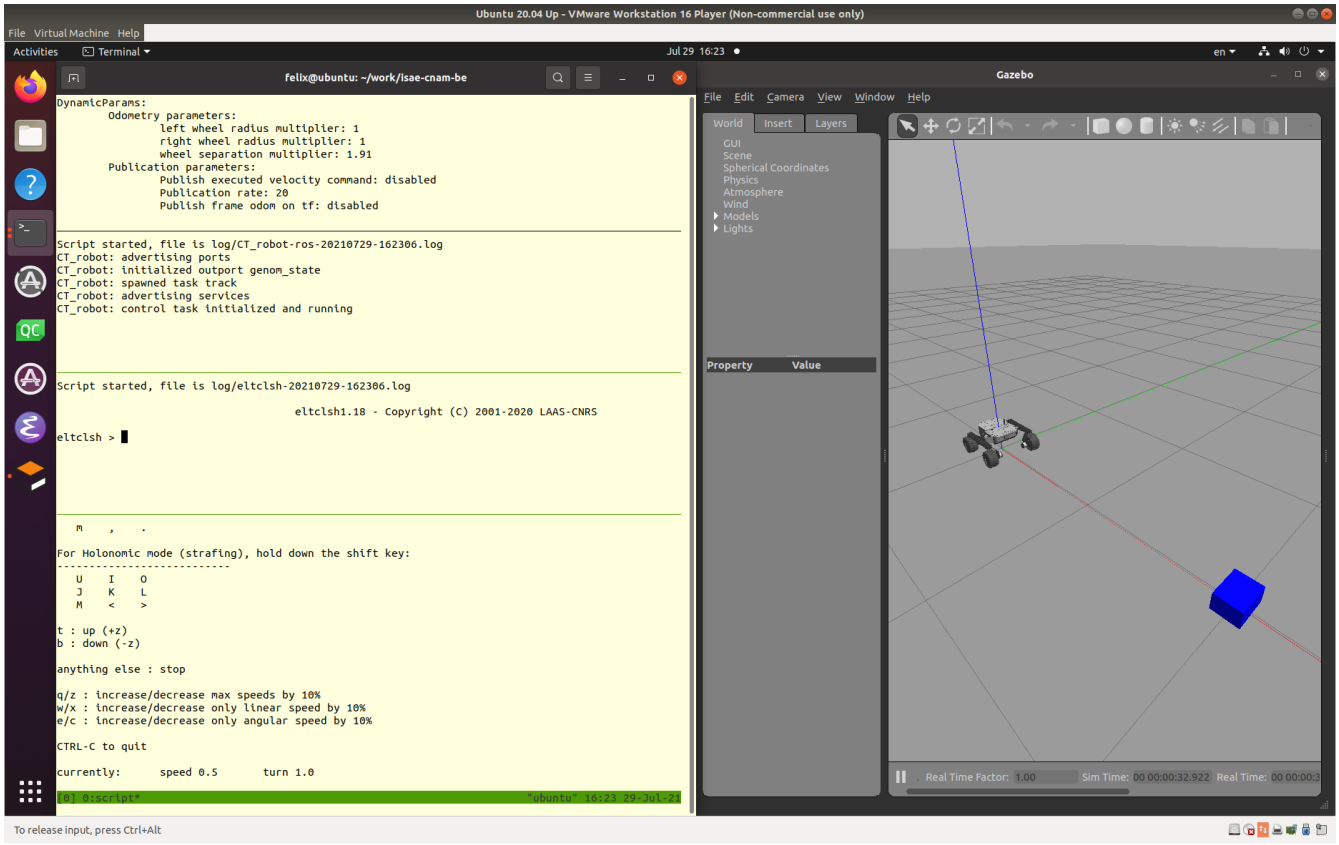


Figure 4: Ecran après lancement de l'expérimentation CT_robot

Your terminal is divided in four panes, 1) the simulation, 2) the module itself 3) a shell `eltclsh` which allows you to control the `CT_robot` module and 4) the ROS node to control the movement of the blue brick.

In the `eltclsh` pane, enter the following commands[1]:

```
eltclsh > source start.tcl
eltclsh > init
eltclsh > CT_robot::ColorTrack &
eltclsh >
```

The experiment is then active, two small windows allow to visualize what the simulated camera sees, and the result of the color filtering. Just like in the previous tutorial, you can move the brick with the arrows of your keyboard (forward, back, left, right), and watch the robot follow the blue brick.

Observe the ROS topics present, the services, and the actions: `rostopic list`, `rosservice list`, `rosaction list`, `rostopic echo`, etc.

Questions for evaluation: G2, (give the answer on a free sheet (G2.1, G2.2, etc). When the module is active:

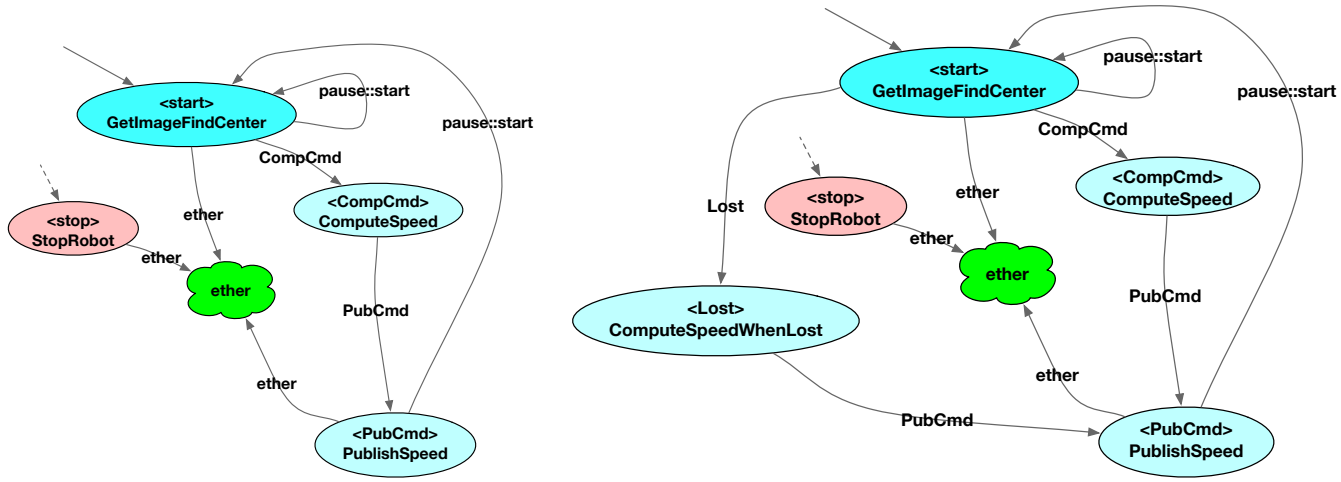
[1]The `eltclsh` shell supports command history (up and down arrows) and command completion (`tab`)

1. What is the refresh rate of the/`CT_robot/CmdPort` port/ROS topic, (remapped to `/cmd_vel`)
2. Is this consistent with the frequency of the `track` task?
3. Why isn't it consistent? (hint: observe the refresh rate of the topic: `/camera/image/raw`)
4. What happens if you move the brick quickly and it goes out of the camera's field (to the left or to the right)?
5. What part of the code leads to this behavior?
6. What happens if you move the brick back into the camera's field of view with the arrows?
7. Using the commands: `rostopic list`, `rosservice list`, `rosaction list`, what happened to the ports, the services of attributes/functions and services activities of GenoM... in ROS.

When you want to quit the simulation, type `<Ctrl-D>` in the `eltclsh` window, and you return to the terminal.

4.4 GenoM: modification of the component, regenerating, compilation, test

You will edit (the `.gen`) file to modify the automata of the ColorTrack service (fig. 5.a) to add a new state `Lost`, when the robot has lost the target. The resulting automata of the ColorTrack service should be the one shown in fig. 5.b. We will then command the robot to turn on itself with a linear speed `vx=0.2` and angular speed `wz=0.2` until it finds the target.



a) Initial version b) Adding a `Lost` state

Figure 5: Automata of the `ColorTrack` activity service.

What should be, in the specification of the `.gen` file, the parameters of the new codel (`codel <Lost> ComputeSpeedWhenL` to just produce a "fixed" speed (inspired by the parameters of `ComputeSpeed`)?

After the modification of the file `CT_robot.gen`, check that there is no error. In the folder `isae-upssitech-be`), type the command:

```
$ genom3 -n CT_robot.gen
```

Correct any errors and then regenerate the codel files in interactive mode (which will result in inserting in `codels/CT_robot_track_codels.cc` the call of the new codel `ComputeSpeedWhenLost` that you specified in the `.gen`. **Beware, when calling `genom3 skeleton`, the command proposes modifications to the files `.cc` and `Makefile.am` . Accept those of the `.cc`, but, if necessary, refuse those of the `Makefile.am`**

```
$ genom3 skeleton -i -l c++ CT_robot.gen
```

Edit the file `codels/CT_robot_track_codels.cc` where the code for calling your new codel (the one called in the `Lost` state whenthe blue brick was not found in the image). It is important to let the previous command synthesize the correct codel call in order to avoid the risk of a double definition (which is allowed in C++ if the arguments are different). So you will notice that GenoM has inserted the code for calling the codel `ComputeSpeedWhenLost`, all you have to do is fill it in and to insert the code which is appropriate (there also, inspire you of the codel `ComputeSpeed`, notice the use of `CT_robot_cmd_s *cmd`). It is also necessary to change the codel `GetImageFindCenter` that it returns the good state according to the situation `CT_robot_CompCmd` or `CT_robot_Lost`, as well as the codel `ComputeSpeed` to remove the now useless code.

Return to the `build` directory, redo `make install`.

```
$ cd build
$ make install
```

Test your new version, and check that the new behavior is correct.

Evaluation: G3: show the result to the professors present, so that they validate it (that your robot turns on itself when it loose the target, and resume tracking when it finds it again).

4.5 GenoM: second modification of the component

We want to allow the user to modify the patrol speeds: when the robot searches for the target (instead of fixed speeds, linear `vx=0.2` and angular `wz=0.2`). For this purpose:

- define a variable `cmd_patrouille` in the `IDS` (of the same type as `cmd`).
- define a service `attribute Set_Patrouille_Speed` to be able to give it a value.
- modify (in the file `CT_robot.gen`) the specification of the codel `ComputeSpeedWhenLost` to add this new parameter.
- regenerate the codel files in interactive mode: `genom3 skeleton -i -l c++ CT_robot.gen`
- modify (in the file `codels/CT_robot_track_codels.cc`) the codel `ComputeSpeedWhenLost` whose call has been updated to take into account the new parameter `cmd_patrouille`.
- re-compile and install: `cd build; make install`
- test (it is advised to give a value to `cmd_patrouille` with `Set_Patrouille_Speed` before launching the tracking, otherwise you run the risk of sending the robot undefined speed values... if it loses the target). [2]
  `CT_robot::Set_Patrol_Speed {cmd_patrol {vx 0 wz 1}}`
  after the `init` but before the `CT_robot::ColorTrack &`.

Evaluation: G4: show the result to the professors present, so they validate it (that you can dynamically change the patrol speed).

5 T.P.4: GenoM: FIACRE formal model synthesis and Petri Net

Go again in the `isae-upssitech-be` directory and create a new subdirectory:

```
$ cd ~/work/isae-upssitech-be
$ mkdir build-tina
$ cd build-tina/
```

Check that your GenoM install provides the proper templates (the result may vary but the `fiacre/model` template should be in the list):

```
$ genom3 -l
fiacre/pocolibs
fiacre/ros
fiacre/model
pocolibs/server
pocolibs/client/c
ros/server
ros/client/c
ros/client/ros
skeleton
example
mappings
interactive
```

Synthesize the model with the file containing, for each codel, the Worst Case Execution Time (WCET), `CT_robot.wcet`, and a "simple" client `CT_robot.client.fcr`. The `-s` specifies the multiplication factor (frequency) to be apply to the time values (WCET and task periods).

[2] Note that for a given scenario, this is the kind of formal property one might want to show. In In this case, the simplest thing to do is to specify that `ColorTrack` can only be executed after `Set_Patrouille_Speed` by using the field `after` of the declaration of the `ColorTrack` activity.

```
$ genom3 fiacre/model -s 1000 -w ../CT_robot.wcet -c ../CT_robot.client.fcr ../CT_robot.gen
creating tina/CT_robot_tina.fcr
creating tina/Makefile.am
creating tina/configure.ac
creating tina/depend/input.d
```

Consult the file `tina/CT_robot_tina.fcr`. For example, look for the model "fiacre" corresponding to the activity service `ColorTrack`.

Moreover, look at the end of the file for the properties: `CT_robot_unfinished` and `CT_robot_track_shedulable`.

You can synthesize the Petri nets corresponding to the component `CT_robot`. This takes a few seconds (but normally less than a minute CPU).

In the folder `tina`:

```
$ autoreconf -vif
$ ./configure --prefix=/home/felix/work
$ make
```

We have thus synthesized the space of reachable states of your component.

```
$ cd CT_robot.tts
$ nd CT_robot.net
```

The program `nd` window appears on the screen. From the **Edit** menu of the program, select **Draw** and then **OK**. You should obtain various RoP similar to those shown in Figures 6, 7 and 8:

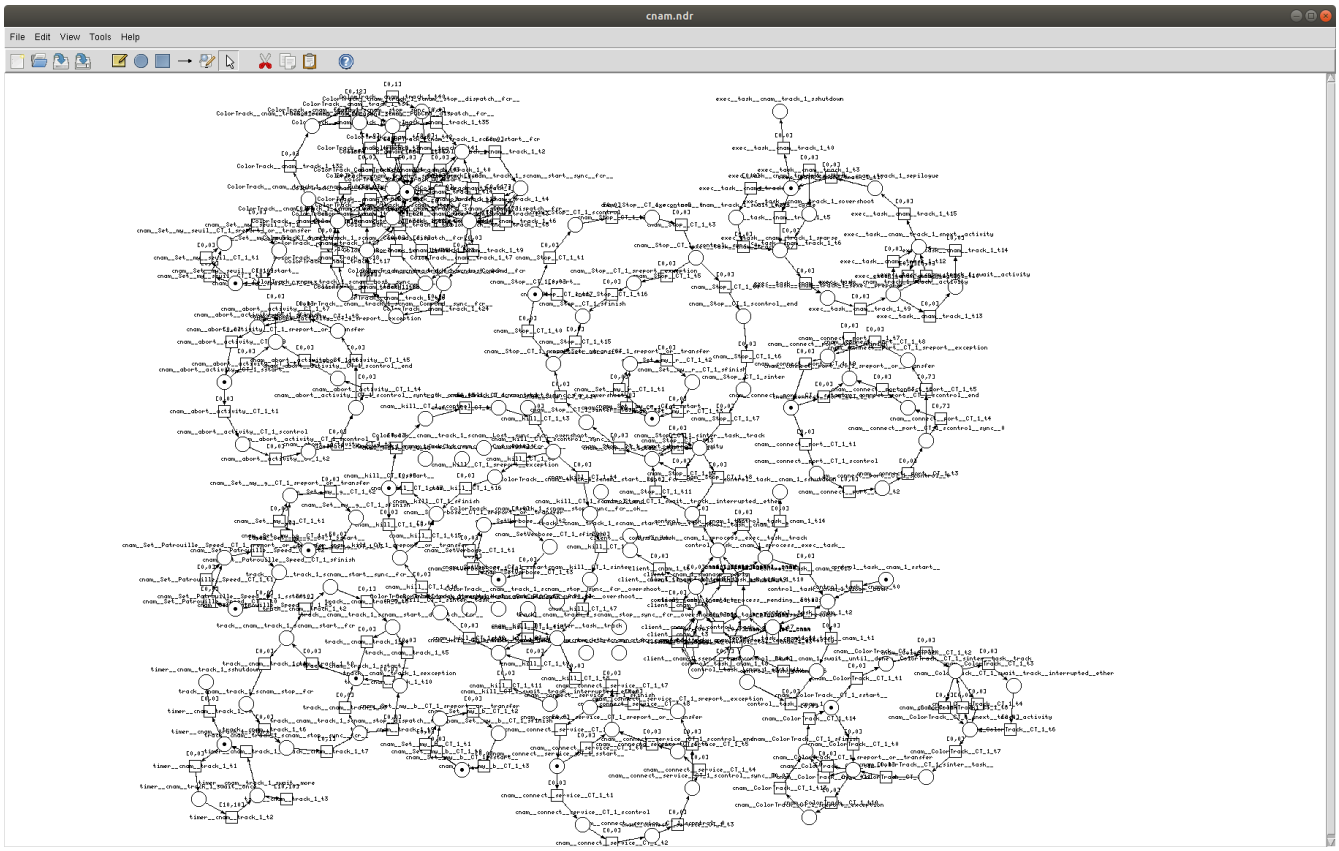


Figure 6: Petri net of the `CT_robot.gen` component

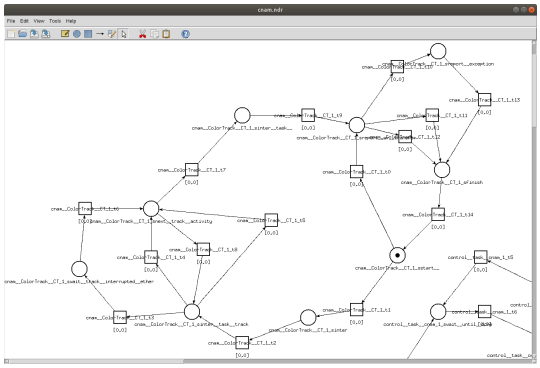


Figure 7: The Petri net of the control task part of the `ColorTrack` activity from the `CT_robot.gen` component.

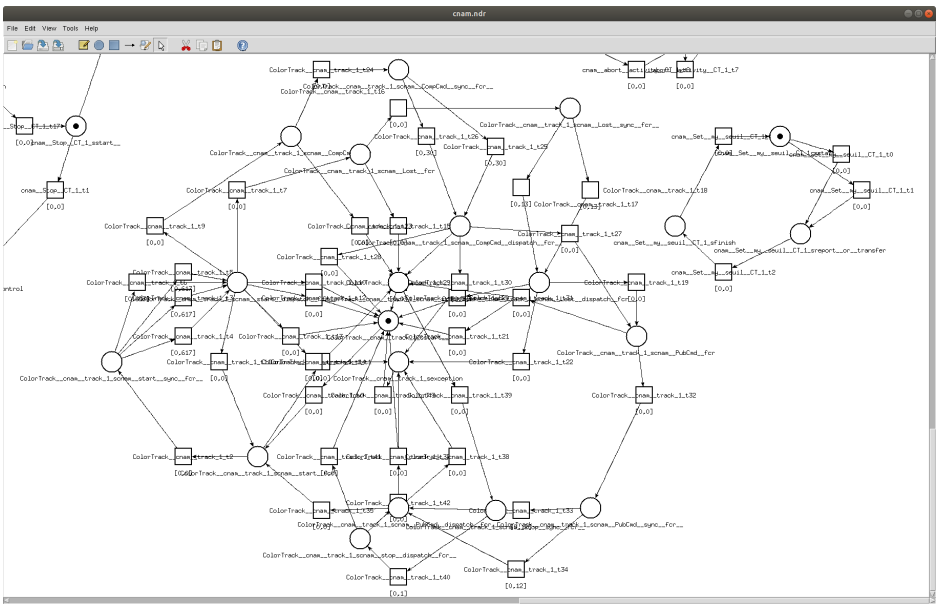


Figure 8: The Petri net of the automata part of the `ColorTrack` activity from the `CT_robot.gen` component.

We will now examine the formal properties `CT_robot_unfinished` and `CT_robot_track_shedulable`:

`CT_robot_unfinished` checks if it is impossible to reach the `finish` state of the client. A priori, we would like it to be to be false, i.e. that in all cases, the client terminates.

`CT_robot_track_shedulable` checks if the task `track` is schedulable or not, taking into account the period of the task, the values declared in the file `CT_robot.wcet` , and the possible executions of the component. This is generally a desirable property, otherwise it means that the system has found an execution trace where the period of the task has been exceeded.

Saisir la commande suivante :

```
selt -stats CT_robot.tts/CT_robot.ktz CT_robot.tts/CT_robot.ltl | more
```

Questions for Evaluation: T1:

1. Is the property `CT_robot_unfinished` true or false?
2. Is the property `CT_robot_track_shedulable` true or false?
3. What did `selt` give you to convince you?

   Let's imagine now that you optimize the codel `GetImageFindCenter` and that its wcet which is currently 6 ms is after optimization of 4 ms (modify the file `CT_robot.wcet` accordingly), redo a `make` in the folder `tina` and redo the command `selt` as above.
4. After this change, is the property `CT_robot_track_shedulable` true or false?
5. Show the result to the teachers present, that they validate it.

6 T.P.5: GenoM: H-FIACRE formal model synthesis and execution with HIPPO

We will now synthesize the H-FIACRE model of your component.

Go back to the `isae-upssitech-be` directory and create a new subdirectory:

```
$ cd ~/work/isae-upssitech-be
$ mkdir build-hippo
$ cd build-hippo
```

```
$ genom3 fiacre/ros -s 1000 -w ../CT_robot.wcet ../CT_robot.gen
```

In the `fiacre-ros` directory:

```
$ cd fiacre-ros
$ autoreconf -vif
$ ./configure --prefix=/home/felix/work
$ make install
```

launch again the experiment

```
$ cd ~/work/isae-upssitech-be
$ ./start-hippo.sh
```

You notice that the second window does not present the GenoM module `CT_robot` anymore, but the HIPPO engine which executes the H-FIACRE model of `CT_robot`.

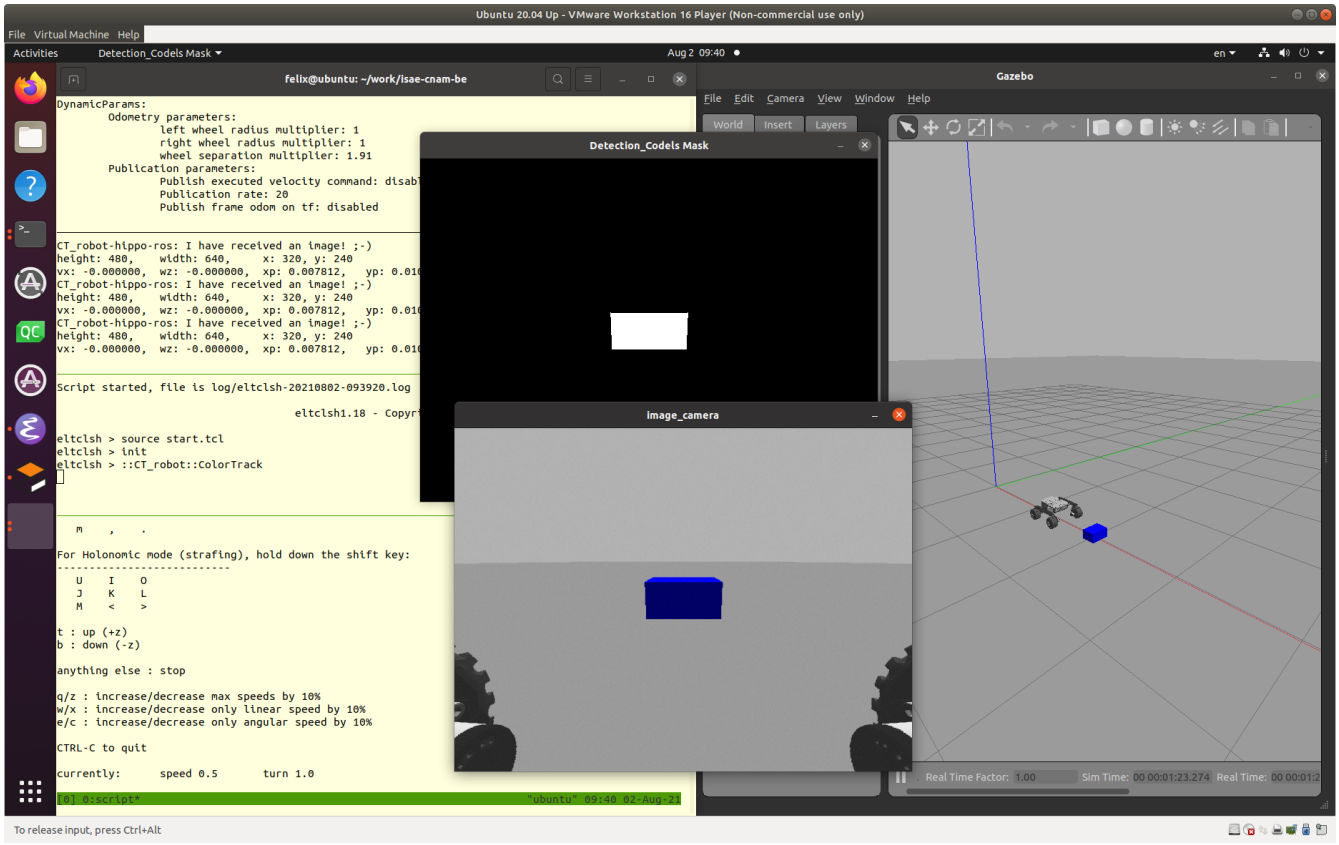


Figure 9: Écran après lancement de l'expérimentation CT_robot with Hippo

In the `eltclsh` pane, enter the following commands:

```
eltclsh > source start.tcl
eltclsh > init
eltclsh > CT_robot::ColorTrack &
eltclsh >
```

Evaluation: H1: show the result to one of the professors present.

7 T.P.6: Experiment with a complex mobile robot (OSMOSIS)

The objective of this section is to illustrate the approach presented in class on a complex mobile robot. It is described in detail on this page: https://redmine.laas.fr/projects/osmosis/gollum/index

All the software used is already installed on the virtual machine. The same modules are used on the Minnie robot from LAAS, and Robotnik from ONERA. A HIPPO version of this experimentation is also available and functional (in simulation AND on the robot Minnie robot).

Make sure that `roscore` is still active (if not, restart it in a separate window), then launch the experimentation with the commands: (enlarge your window so that it takes up the right half of your screen)

```
$ cd ~/work/osmosis/scripts
$ ./start-simu.sh -r robotnik -e airport
```

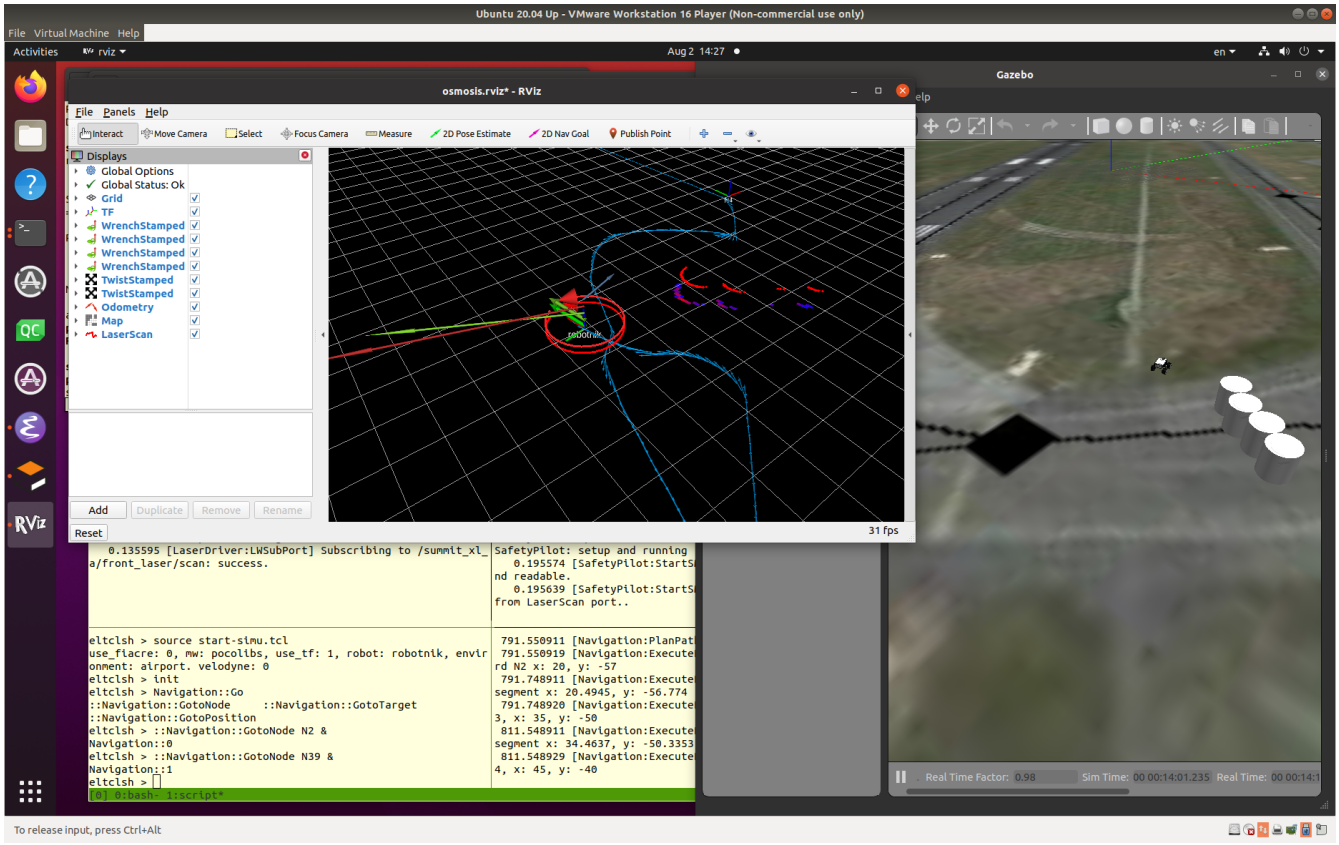


Figure 10: Ecran après lancement de l'expérimentation OSMOSIS (quelques obstacles ont été ajoutés).

Apart from your window which has been divided into several sub-windows, corresponding to the different GenoM modules and the eltclsh interface, two new windows have appeared:

Gazebo is the simulator (with a physical engine). Familiarize yourself with the zoom and the movements to locate the robot (it is initially in the same place as on the (it is initially in the same place as on the screenshot).

Rviz is a tool to visualize different ROS topics.

When the initialization is finished, in the `eltclsh` window, enter the commands:

```
eltclsh > source start-simu.tcl
eltclsh > init
eltclsh > Navigation::GotoNode N2 &
```

You can add obstacles on the path of the robot, and see the navigation avoiding them...
Question for evaluation: O1: show the result to the teachers present so they validate it.

8 T.P.7: Experimentation with a drone

The objective of this section is to illustrate the approach presented in class on a drone. It is described in details on this page: https://redmine.laas.fr/projects/drone-v-v/gollum/index

All the software used is already installed. The same modules are used on the LAAS drones. A Hippo version of this experimentation is also available and functional (in simulation and on LAAS drones).

Make sure that `roscore` is still active (if not, restart it in a separate window), then launch the experimentation with the command

```
$ cd ~/work/drone/scripts
$ ./start-simu.sh
```

Again, your window is divided into several subwindows, corresponding to the different GenoM modules and the eltclsh interface, and two windows, Gazebo and Rviz, have appeared.

If your machine does not seem powerful enough, you can then quit the RVIZ that is launched to reduce the load. But it is not sure that this will be enough...

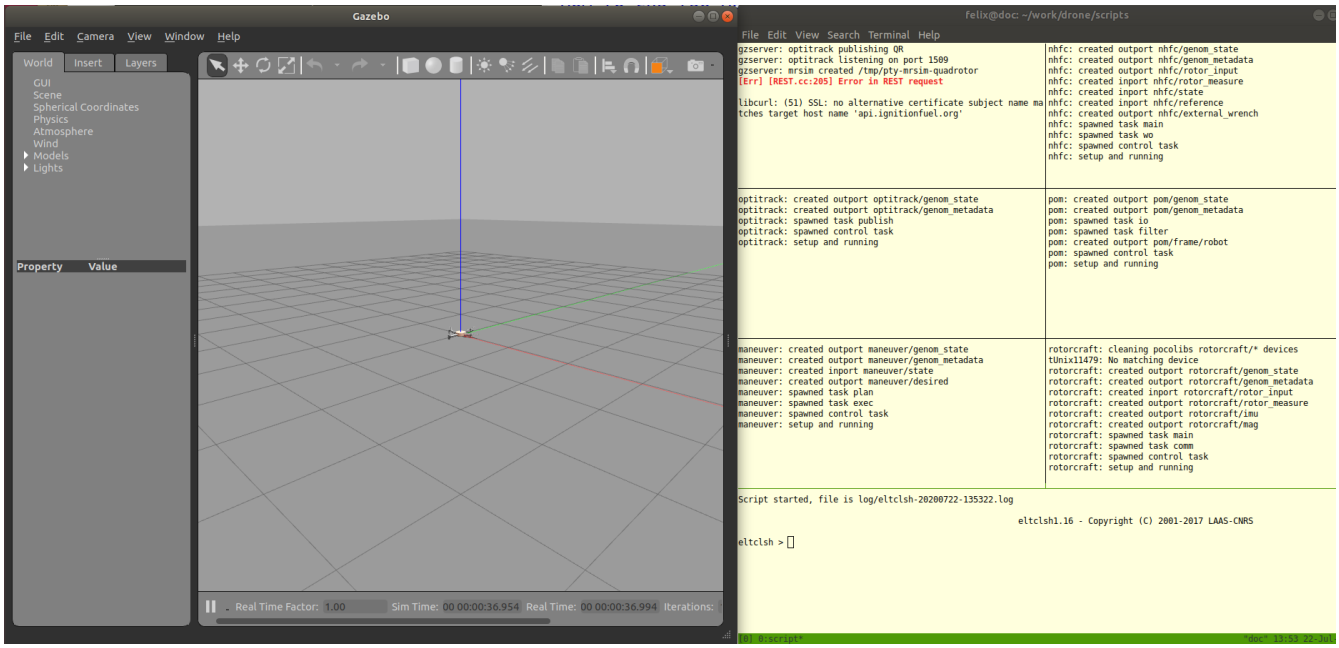


Figure 11: Ecran après lancement de l'expérimentation drone.

When the initialization is finished, in the eltclsh pane, type:

```
eltclsh > source start.tcl
eltclsh > init
eltclsh > setup
eltclsh > ::maneuver::goto {x 1 y 1 z 1.7 yaw 2 duration 0}
eltclsh > carre
eltclsh > fly_around
```

`carre` makes the drone fly to the four corners of a square. `fly_around` makes the drone navigate randomly in a cube of $10x10x10$ (you can find the definition of the function `fly_around` in the file `start.tcl`).

Question for evaluation: D1: show the result to the teachers present so they can validate it.

9 ROS cheatsheet

ROS Indigo Cheatsheet

Filesystem Management Tools

rospack	A tool for inspecting packages.
rospack profile	Fixes path and pluginlib problems.
roscd	Change directory to a package.
rospd/rosd	Pushd equivalent for ROS.
rosls	Lists package or stack information.
rosed	Open requested ROS file in a text editor.
roscp	Copy a file from one place to another.
rosdep	Installs package system dependencies.
roswtf	Displays a errors and warnings about a running ROS system or launch file.
catkin_create_pkg	Creates a new ROS stack.
wstool	Manage many repos in workspace.
catkin_make	Builds a ROS catkin workspace.
rqt_dep	Displays package structure and dependencies.

Usage:

```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rospd [package[/subdir] | +N | -N]
$ rosd
$ rosls [package[/subdir]]
$ rosed [package] [file]
$ roscp [package] [file] [destination]
$ rosdep install [package]
$ roswtf or roswtf [file]
$ catkin_create_pkg [package_name] [depend1]..[dependN]
$ wstool [init | set | update]
$ catkin_make
$ rqt_dep [options]
```

Start-up and Process Launch Tools

roscore

The basis nodes and programs for ROS-based systems. A roscore must be running for ROS nodes to communicate.

Usage:

```
$ roscore
```

rosrun

Runs a ROS package's executable with minimal typing.

Usage:

```
$ rosrun package_name executable_name
```

Example (runs turtlesim):

```
$ rosrun turtlesim turtlesim_node
```

roslaunch

Starts a roscore (if needed), local nodes, remote nodes via SSH, and sets parameter server parameters.

Examples:

Launch a file in a package:

```
$ roslaunch package_name file_name.launch
```

Launch on a different port:

```
$ roslaunch -p 1234 package_name file_name.launch
```

Launch on the local nodes:

```
$ roslaunch --local package_name file_name.launch
```

Logging Tools

rosbag

A set of tools for recording and playing back of ROS topics.

Commands:

rosbag record	Record a bag file with specified topics.
rosbag play	Play content of one or more bag files.
rosbag compress	Compress one or more bag files.
rosbag decompress	Decompress one or more bag files.
rosbag filter	Filter the contents of the bag.

Examples:

Record select topics:

```
$ rosbag record topic1 topic2
```

Replay all messages without waiting:

```
$ rosbag play -a demo_log.bag
```

Replay several bag files at once:

```
$ rosbag play demo1.bag demo2.bag
```

Introspection and Command Tools

rosmsg/rossrv

Displays Message/Service (msg/srv) data structure definitions.

Commands:

rosmsg show	Display the fields in the msg/srv.
rosmsg list	Display names of all msg/srv.
rosmsg md5	Display the msg/srv md5 sum.
rosmsg package	List all the msg/srv in a package.
rosmsg packages	List all packages containing the msg/srv.

Examples:

Display the Pose msg:

```
$ rosmsg show Pose
```

List the messages in the nav_msgs package:

```
$ rosmsg package nav_msgs
```

List the packages using sensor_msgs/CameraInfo:

```
$ rosmsg packages sensor_msgs/CameraInfo
```

rosnode

Displays debugging information about ROS nodes, including publications, subscriptions and connections.

Commands:

rosnode ping	Test connectivity to node.
rosnode list	List active nodes.
rosnode info	Print information about a node.
rosnode machine	List nodes running on a machine.
rosnode kill	Kill a running node.

Examples:

Kill all nodes:

```
$ rosnode kill -a
```

List nodes on a machine:

```
$ rosnode machine aqy.local
```

Ping all nodes:

```
$ rosnode ping --all
```

rostopic

A tool for displaying information about ROS topics, including publishers, subscribers, publishing rate, and messages.

Commands:

rostopic bw	Display bandwidth used by topic.
rostopic echo	Print messages to screen.
rostopic find	Find topics by type.
rostopic hz	Display publishing rate of topic.
rostopic info	Print information about an active topic.
rostopic list	List all published topics.
rostopic pub	Publish data to topic.
rostopic type	Print topic type.

Examples:

Publish hello at 10 Hz:

```
$ rostopic pub -r 10 /topic_name std_msgs/String hello
```

Clear the screen after each message is published:

```
$ rostopic echo -c /topic_name
```

Display messages that match a given Python expression:

```
$ rostopic echo --filter "m.data=='foo'" /topic_name
```

Pipe the output of rostopic to rosmsg to view the msg type:

```
$ rostopic type /topic_name | rosmsg show
```

rosparam

A tool for getting and setting ROS parameters on the parameter server using YAML-encoded files.

Commands:

rosparam set	Set a parameter.
rosparam get	Get a parameter.
rosparam load	Load parameters from a file.
rosparam dump	Dump parameters to a file.
rosparam delete	Delete a parameter.
rosparam list	List parameter names.

Examples:

List all the parameters in a namespace:

```
$ rosparam list /namespace
```

Setting a list with one as a string, integer, and float:

```
$ rosparam set /foo "['1', 1, 1.0]"
```

Dump only the parameters in a specific namespace to file:

```
$ rosparam dump dump.yaml /namespace
```

rosservice

A tool for listing and querying ROS services.

Commands:

rosservice list	Print information about active services.
rosservice node	Print name of node providing a service.
rosservice call	Call the service with the given args.
rosservice args	List the arguments of a service.
rosservice type	Print the service type.
rosservice uri	Print the service ROSRPC uri.
rosservice find	Find services by service type.

Examples:

Call a service from the command-line:

```
$ rosservice call /add_two_ints 1 2
```

Pipe the output of rosservice to rossrv to view the srv type:

```
$ rosservice type add_two_ints | rossrv show
```

Display all services of a particular type:

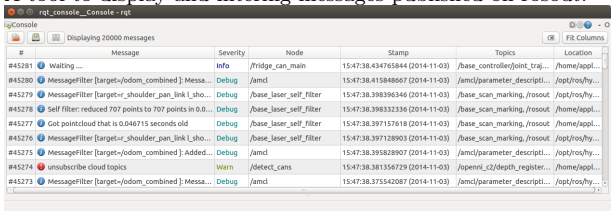
```
$ rosservice find rospy_tutorials/AddTwoInts
```

ROS Indigo Cheatsheet

Logging Tools

rqt_console

A tool to display and filtering messages published on rosout.

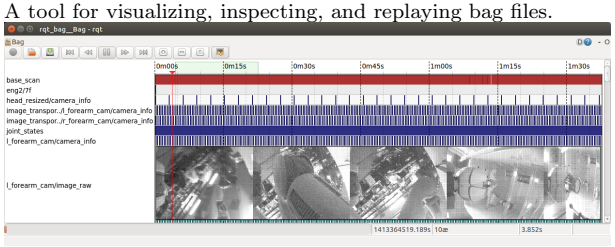


Usage:

```
$ rqt_console
```

rqt_bag

A tool for visualizing, inspecting, and replaying bag files.



Usage, viewing:

```
$ rqt_bag bag_file.bag
```

Usage, bagging:

```
$ rqt_bag *press the big red record button.*
```

rqt_logger_level

Change the logger level of ROS nodes. This will increase or decrease the information they log to the screen and rqt_console.

Usage:

```
viewing $ rqt_logger_level
```

Introspection & Command Tools

rqt_topic

A tool for viewing published topics in real time.

Usage:

```
$ rqt
Plugin Menu->Topic->Topic Monitor
```

rqt_msg, rqt_srv, and rqt_action

A tool for viewing available msgs, srvs, and actions.

Usage:

```
$ rqt
Plugin Menu->Topic->Message Type Browser
Plugin Menu->Service->Service Type Browser
Plugin Menu->Action->Action Type Browser
```

rqt_publisher, and rqt_service_caller

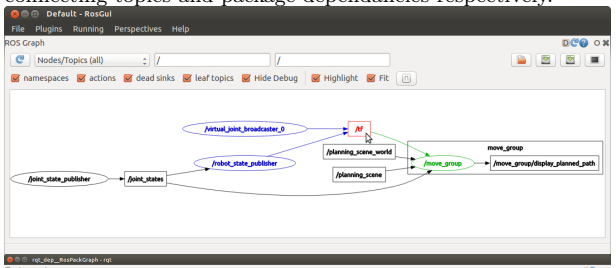
Tools for publishing messages and calling services.

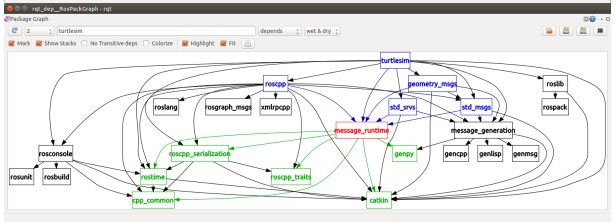
Usage:

```
$ rqt
Plugin Menu->Topic->Message Publisher
Plugin Menu->Service->Service Caller
```

rqt_graph, and rqt_dep

Tools for displaying graphs of running ROS nodes with connecting topics and package dependancies respectively.





Usage:

```
$ rqt_graph
$ rqt_dep
```

rqt_top

A tool for ROS specific process monitoring.

Usage:

```
$ rqt
Plugin Menu->Introspection->Process Monitor
```

rqt_reconfigure

A tool for dynamically reconfiguring ROS parameters.

Usage:

```
$ rqt
Plugin Menu->Configuration->Dynamic Reconfigure
```

Development Environments

rqt_shell, and rqt_py_console

Two tools for accessing an xterm shell and python console respectively.

Usage:

```
$ rqt
Plugin Menu->Miscellaneous Tools->Shell
Plugin Menu->Miscellaneous Tools->Python Console
```

Data Visualization Tools

tf_echo

A tool that prints the information about a particular transformation between a source_frame and a target_frame.

Usage:

```
$ rosrun tf tf_echo <source_frame> <target_frame>
```

Examples:

To echo the transform between /map and /odom:

```
$ rosrun tf tf_echo /map /odom
```

view_frames

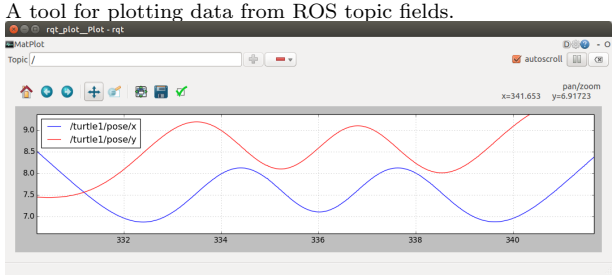
A tool for visualizing the full tree of coordinate transforms.

Usage:

```
$ rosrun tf2_tools view_frames.py
$ evince frames.pdf
```

rqt_plot

A tool for plotting data from ROS topic fields.



Examples:

To graph the data in different plots:

```
$ rqt_plot /topic1/field1 /topic2/field2
```

To graph the data all on the same plot:

```
$ rqt_plot /topic1/field1,/topic2/field2
```

To graph multiple fields of a message:

```
$ rqt_plot /topic1/field1:field2:field3
```

rqt_image_view

A tool to display image topics.

Usage:

```
$ rqt_image_view
```

ROS Indigo Catkin Workspaces

Create a catkin workspace

Setup and use a new catkin workspace from scratch.

Example:

```
$ source /opt/ros/hydro/setup.bash
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Checkout an existing ROS package

Get a local copy of the code for an existing package and keep it up to date using wstool.

Examples:

```
$ cd ~/catkin_ws/src
$ wstool init
$ wstool set tutorials --git git://github.com/ros/ros_tutorials.git
$ wstool update
```

Create a new catkin ROS package

Create a new ROS catkin package in an existing workspace with catkin create package. After using this you will need to edit the CMakeLists.txt to detail how you want your package built and add information to your package.xml.

Usage:

```
$ catkin_create_pkg <package_name> [depend1] [depend2]
```

Example:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg tutorials std_msgs rospy roscpp
```

Build all packages in a workspace

Use catkin_make to build all the packages in the workspace and then source the setup.bash to add the workspace to the ROS_PACKAGE_PATH.

Examples:

```
$ cd ~/catkin_ws
$ ~/catkin_make
$ source devel/setup.bash
```

10 ROS2 cheatsheet

ROS 2 Cheats Sheet

Command Line Interface

All ROS 2 CLI tools start with the prefix 'ros2' followed by a command, a verb and (possibly) positional/optional arguments.

For any tool, the documentation is accessible with,

```
$ ros2 command --help
```

and similarly for verb documentation,

```
$ ros2 command verb -h
```

Similarly, auto-completion is available for all commands/verbs and most positional/optional arguments. E.g.,

```
$ ros2 command [tab][tab]
```

Some of the examples below rely on:
ROS 2 demos package.

---

action Allows to manually send a goal and displays debugging information about actions.

Verbs:

info	Output information about an action.
list	Output a list of action names.
send_goal	Send an action goal.
show	Output the action definition.

Examples:

```
$ ros2 action info /fibonacci
$ ros2 action list
$ ros2 action send_goal /fibonacci \
 action_tutorials/action/Fibonacci "order: 5"
$ ros2 action show action_tutorials/action/Fibonacci
```

---

bag Allows to record/play topics to/from a rosbag.

Verbs:

info	Output information of a bag.
play	Play a bag.
record	Record a bag.

Examples:

```
$ ros2 info <bag-name>
$ ros2 play <bag-name>
$ ros2 record -a
```

---

component Various component related verbs.

Verbs:

list	Output a list of running containers and components.
load	Load a component into a container node.
standalone	Run a component into its own standalone container node.
types	Output a list of components registered in the ament index.
unload	Unload a component from a container node.

Examples:

```
$ ros2 component list
$ ros2 component load /ComponentManager \
 composition composition::Talker
$ ros2 component types
$ ros2 component unload /ComponentManager 1
```

---

daemon Various daemon related verbs.

Verbs:

start	Start the daemon if it isn't running.
status	Output the status of the daemon.
stop	Stop the daemon if it is running

---

doctor A tool to check ROS setup and other potential issues such as network, package versions, rmw middleware etc.

Alias: **wtf** (where's the fire).

Arguments:

--report/-r	Output report of all checks.
--report-fail/-rf	Output report of failed checks only.
--include-warning/-iw	Include warnings as failed checks.

Examples:

```
$ ros2 doctor
$ ros2 doctor --report
$ ros2 doctor --report-fail
$ ros2 doctor --include-warning
$ ros2 doctor --include-warning --report-fail
```

or similarly,

```
$ ros2 wtf
```

---

extension_points List extension points.

---

extensions List extensions.

---

interface Various ROS interfaces (actions/topics/services)-related verbs. Interface type can be filtered with either of the following option, '--only-actions', '--only-msgs', '--only-srvs'.

Verbs:

list	List all interface types available.
package	Output a list of available interface types within one package.
packages	Output a list of packages that provide interfaces.
proto	Print the prototype (body) of an interfaces.
show	Output the interface definition.

Examples:

```
$ ros2 interface list
$ ros2 interface package std_msgs
$ ros2 interface packages --only-msgs
$ ros2 interface proto example_interfaces/srv/AddTwoInts
$ ros2 interface show geometry_msgs/msg/Pose
```

---

launch Allows to run a launch file in an arbitrary package without to 'cd' there first.

Usage:

```
$ ros2 launch <package> <launch-file>
```

Example:

```
$ ros2 launch demo_nodes_cpp add_two_ints.launch.py
```

---

lifecycle Various lifecycle related verbs.

Verbs:

get	Get lifecycle state for one or more nodes.
list	Output a list of available transitions.
nodes	Output a list of nodes with lifecycle.
set	Trigger lifecycle state transition.

---

msg (deprecated) Displays debugging information about messages.

Verbs:

list	Output a list of message types.
package	Output a list of message types within a given package.
packages	Output a list of packages which contain messages.
show	Output the message definition.

Examples:

```
$ ros2 msg list
$ ros2 msg package std_msgs
$ ros2 msg packages
$ ros2 msg show geometry_msgs/msg/Pose
```

---

multicast Various multicast related verbs.

Verbs:

receive	Receive a single UDP multicast packet.
send	Send a single UDP multicast packet.

---

node Displays debugging information about nodes.

Verbs:

info	Output information about a node.
list	Output a list of available nodes.

Examples:

```
$ ros2 node info /talker
$ ros2 node list
```

---

param Allows to manipulate parameters.

Verbs:

delete	Delete parameter.
describe	Show descriptive information about declared parameters.
dump	Dump the parameters of a given node in yaml format, either in terminal or in a file.
get	Get parameter.
list	Output a list of available parameters.
set	Set parameter

Examples:

```
$ ros2 param delete /talker /use_sim_time
$ ros2 param get /talker /use_sim_time
$ ros2 param list
$ ros2 param set /talker /use_sim_time false
```

---

pkg Create a ros2 package or output package(s)-related information.

Verbs:

create	Create a new ROS2 package.
executables	Output a list of package specific executables.
list	Output a list of available packages.
prefix	Output the prefix path of a package.
xml	Output the information contained in the package xml manifest.

Examples:

```
$ ros2 pkg executables demo_nodes_cpp
$ ros2 pkg list
$ ros2 pkg prefix std_msgs
$ ros2 pkg xml -t version
```

---

run Allows to run an executable in an arbitrary package without having to 'cd' there first.

Usage:

```
$ ros2 run <package> <executable>
```

Example:

```
$ ros2 run demo_node_cpp talker
```

---

security Various security related verbs.

Verbs:

create_key	Create key.
create_permission	Create keystore.
generate_artifacts	Create permission.
list_keys	Distribute key.
create_keystore	Generate keys and permission files from a list of identities and policy files.
distribute_key	Generate XML policy file from ROS graph data.
generate_policy	List keys.

Examples (see sros2 package):

```
$ ros2 security create_key demo_keys /talker
$ ros2 security create_permission demo_keys /talker \
 policies/sample_policy.xml
$ ros2 security generate_artifacts
$ ros2 security create_keystore demo_keys
```

---

service Allows to manually call a service and displays debugging information about services.

Verbs:

call	Call a service.
find	Output a list of services of a given type.
list	Output a list of service names.
type	Output service's type.

Examples:

```
$ ros2 service call /add_two_ints \
 example_interfaces/AddTwoInts "a: 1, b: 2"
$ ros2 service find rcl_interfaces/srv/ListParameters
$ ros2 service list
$ ros2 service type /talker/describe_parameters
```

---

srv (deprecated) Various srv related verbs.

Verbs:

list	Output a list of available service types.
package	Output a list of available service types within one package.
packages	Output a list of packages which contain services.
show	Output the service definition.

---

test Run a ROS2 launch test.

---

topic A tool for displaying debug information about ROS topics, including publishers, subscribers, publishing rate, and messages.

Verbs:

bw	Display bandwidth used by topic.
delay	Display delay of topic from timestamp in header.
echo	Output messages of a given topic to screen.
find	Find topics of a given type type.
hz	Display publishing rate of topic.
info	Output information about a given topic.
list	Output list of active topics.
pub	Publish data to a topic.
type	Output topic's type.

Examples:

```
$ ros2 topic bw /chatter
$ ros2 topic echo /chatter
$ ros2 topic find rcl_interfaces/msg/Log
$ ros2 topic hz /chatter
$ ros2 topic info /chatter
$ ros2 topic list
$ ros2 topic pub /chatter std_msgs/msg/String \
 'data: Hello ROS 2 world'
$ ros2 topic type /rosout
```

11 Annexe : Listings with the code

Listing 2: producteur

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
int main(int argc, char **argv)
{
  ros::init(argc, argv, "producteur");
  ros::NodeHandle nh;
  ros::Publisher chatter_pub =
    nh.advertise<std_msgs::Int32>("chatter", 1000);
  ros::Rate loop_rate(10);
  std_msgs::Int32 val;
  while (ros::ok())
  {
    val.data++;
    chatter_pub.publish(val);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Listing 3: consommateur

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
void chatterCallback(const std_msgs::Int32::ConstPtr& val
    )
{
  ROS_INFO("I heard: [%d]", val->data);
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "consommateur");
  ros::NodeHandle nh;
  ros::Subscriber sub =
    nh.subscribe("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

Listing 4: Composant `cnam_detection.cpp`

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "std_msgs/Int32.h"

//add include for manipulate sensor_msgs::Image type:
#include "sensor_msgs/Image.h"
#include <cv_bridge/cv_bridge.h>
#include "opencv2/opencv.hpp"

//TODO add #include for output topic type (geometry_msgs/Twist)
#include "geometry_msgs/Twist.h"

//for binarisation function:
#include "DetectionCnam_codels.hpp"

static const std::string OPENCV_WINDOW = "Image window";

//TODO declare publisher topic for command output:
ros::Publisher pub;

int32_t _r;
int32_t _g;
int32_t _b;
int32_t _seuil;

ros::NodeHandle *nh;

void cnam_image_Callback(const sensor_msgs::Image::ConstPtr& msg)
{
  ros::Time begin = ros::Time::now();

  //necessary for transform ros image type into opencv image type
  cv_bridge::CvImagePtr cv_ptr;
  try
  {
    cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    ROS_INFO("I have received image! ;-)");
  }
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("cv_bridge exception: %s", e.what());
    return;
  }
#if CV_VERSION_MAJOR == 4
  IplImage _ipl_img=cvIplImage(cv_ptr->image);
#else
```

```
  IplImage _ipl_img=cv_ptr->image;
#endif
  IplImage *ptr_ipl_img= &_ipl_img;

  geometry_msgs::Twist cmd;

  //declare a CvPoint
  CvPoint coord;

  nh->getParam("my_r", _r);
  nh->getParam("my_g", _g);
  nh->getParam("my_b", _b);
  nh->getParam("my_seuil", _seuil);

  //call binarisation method!
  coord = binarisation(ptr_ipl_img, _b, _g, _r,_seuil);

  if (coord.x > -1)
  {

  //print with ROS_INFO coordinate points.x points.y
  ROS_INFO("width: %d",ptr_ipl_img->width);
  ROS_INFO("height: %d",ptr_ipl_img->height);
  ROS_INFO("coord X: %d",coord.x);
  ROS_INFO("coord Y: %d",coord.y);

  ROS_INFO("_r: %d", _r);
  ROS_INFO("_g: %d", _g);
  ROS_INFO("_b: %d", _b);
  ROS_INFO("_seuil: %d", _seuil);


  // calculate command to send:

  float cmd_x_pixel_value= 5.0 / ptr_ipl_img->width;
  float cmd_y_pixel_value= 5.0 / ptr_ipl_img->height;

  cmd.angular.z= - (coord.x - ptr_ipl_img->width/2) * cmd_x_pixel_value;
  cmd.linear.x = - (coord.y - ptr_ipl_img->height/2) * cmd_y_pixel_value;

  ROS_INFO("cmd.linear.x: %f",cmd.linear.x);
  ROS_INFO("cmd.angular.z: %f",cmd.angular.z);
  }
  else
  {
    ROS_INFO("Lost image");
    cmd.angular.z=0.0;
    cmd.linear.x=0.0;
  }
  //TODO send command in output topic.
  pub.publish(cmd);
  ros::Time end = ros::Time::now();

  ROS_INFO("begin: %lu",begin.toNSec());
  ROS_INFO("end: %lu",end.toNSec());
  ROS_INFO("duration: %lu", (end.toNSec() - begin.toNSec()));

}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "cnam_detection");

  nh = new ros::NodeHandle;

  ros::Subscriber sub = nh->subscribe("cnam_input_image", 1, cnam_image_Callback);

```

```
  //instantiate publisher topic for command output (geometry_msgs/Twist):
  //must be global for visibility in callback function.
  pub = nh->advertise<geometry_msgs::Twist>("cnam_output_command", 1);

  nh->setParam("my_r", 3);
  nh->setParam("my_g", 2);
  nh->setParam("my_b", 105);
  nh->setParam("my_seuil", 40);

  ros::spin();

  return 0;
}
```

Listing 5: Spécification du module dans le fichier `CT_robot.gen`

```
/*
 * Copyright (c) 2019-2021 LAAS/CNRS
 *
 * Author: Felix Ingrand - LAAS/CNRS
 *
 * Permission to use, copy, modify, and/or distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */

#include "geometry.idl" // Twist definition ROS masquerade geometry/Twist
#include "sensor.idl"   // Image definition ROS masquerade sensor/Image

/* -------------------------- MODULE DECLARATION -------------------------- */
component CT_robot {
  version "1.0";
  email   "felix@laas.fr";
  lang    "c";
  doc     "This module illustrates a simple GenoM module for the CT_robot ISAE UPSSITECH BE.";

  codels-require "roscpp,geometry_msgs,nav_msgs,opencv4,cv_bridge";

  exception bad_image_port, bad_cmd_port, opencv_error, e_mem;

  struct cmd_s{
      double vx; // The internal speed struct declaration
      double wz;
  };

  ids {
    long x,y;      // Position of the center of orange object in the image
    long width,height;  // Size of the image
    long verbose;  // For logging verbosity
    cmd_s cmd;     // Internal speed command computed

    long my_r;     // Various values used by the image analysis algo.
    long my_g;
    long my_b;
    long my_seuil;
  };

  /* ------------- DEFINITION OF PORTS -------------- */
  port in sensor::Image ImagePort {
```

```
  doc "The port ImagePort containing the image from the camera.";
};

port out geometry::Twist CmdPort { // CmdPort is the speed command port
                                   // (cmd (see above), in lower case, is the ids field)
  doc "The port CmdPort in which we put the speed at which we drive the robot.";
};

/* ------------------ TASK DEFINITION -------------------- */
task track {
  period   10 ms; // fast, but we only process the image when it is new.
  codel <start> InitIDS(port out CmdPort, ids out cmd, ids out x, ids out y) yield ether;
  codel <stop> CleanIDS(port out CmdPort) yield ether;
};

/* ------------------ SERVICES DEFINITION: The attributes -------------------- */
attribute SetVerbose(in verbose = 0 : "Verbose level")
{
  doc  "Set the verbose level.";
};

attribute Set_my_r(in my_r);
attribute Set_my_g(in my_g);
attribute Set_my_b(in my_b);
attribute Set_my_seuil(in my_seuil);

/* ------------------ SERVICES DEFINITION: The Functions -------------------- */
function Stop()
{
  doc  "Stop the tracking.";
  codel StopTrack(in verbose); // This codel does not do anything... just here as an example.

  interrupts ColorTrack; // This field will force the transition to the stop codel in the
       // ColorTrack activity automata
};

/* ------------------ SERVICES DEFINITION: The activities -------------------- */

activity ColorTrack () {
  doc    "Produce a twist so the robot follow the colored object.";

  task   track; // The task in which ColorTrack will execute

  // Automata syntax
  // codel <state>  c_function({{ids|port|local}? {in|out|inout} arg_k,}*)
  //                     yield {pause::}?<state_i> {, {pause::}?<state_j>}*;
  // - ids/port/local is optional if arg_k name is not ambiguous,
  // - start, stop and ether are predefined states,
  // - yield pause::state means transition will wait the next task cycle to lead to state.

  codel <start>   GetImageFindCenter(port in ImagePort, ids in my_r, ids in my_g, ids in my_b,
          ids in my_seuil, ids out x, ids out y,
          ids out width, ids out height, ids in verbose)
      yield pause::start, // no new image, wait next cycle of the exec task
           CompCmd,     // found the image
           ether;       // in case of error.
  codel <CompCmd> ComputeSpeed(ids in x, ids in y, ids in width, ids in height,
          ids out cmd, ids in verbose)
                          yield PubCmd;
  codel <PubCmd> PublishSpeed(ids in cmd, port out CmdPort)
      yield pause::start, // Loop back at the start in the next cycle
           ether;       // in case of error.
  codel <stop> StopRobot(ids out cmd, port out CmdPort) // stop is a predefined state in GenoM
      yield ether; // ColorTrack execution will jump to this state when the
            //service is interrupted

  throw  bad_cmd_port, bad_image_port, opencv_error; // Possible errors in the codels.
```

```
                    // Any will force execution to ether
    interrupts   ColorTrack; // Only one ColorTrack service running at a time
  };
};
```

Evaluation sheet BE UPSSITECH 3A SRI

(Return this sheet to the teacher at the end of the TP)

Name: **First name:** **Binome number:** **Grade:**

The question are in the text of the BE, the text on this page is just a guide to the location of the answer.

R1

1. # node, names:
2. exchanged data:
3. types:
4. frequency:
5. report on rosbag:
6. report if ros bag and producer:
7. finding if you kill roscore:

R2

1. how many nodes, and their names:
2. the topic:
3. its type:
4. number of services served by `/turtlesim`:
5. action name:
6. name of the server node:
7. goal:
   result:
   feedback:

R3

1. # node, names:
2. exchanged data:
3. types:
4. frequency:
5. mechanism:
6. declaration:
7. command before publish:

G1

1. direction `CmdPort`:
2. direction `ImagePort`:
3. frequency of `track`:

4. service interrupted by `Stop`:
5. C/C++ function called, where and role:
6. frequency codels of `ColorTrack`:
7. multiple services `ColorTrack`? why:
8. states leading to `start`:
9. parameter/s `out` codel `ComputeSpeed`:
10. where are they (`ids`, `port`, `local`)?

G2

1. frequency of `/CT_robot/CmdPort`:
2. consistent with frequency of task `track`:
3. if inconsistent why:
4. lost brick:
5. code:
6. brick put back in field:
7. ports:
   attributes/functions:
   activities:

G3
teacher validation:

G4
teacher validation:

T1

1. the property `CT_robot_unfinished` is:
2. the property `CT_robot_track_shedulable` is:
3. selt provides:
4. after optimization, the property `CT_robot_track_shedulable` is:
5. teacher validation:

H1 teacher validation:

01 teacher validation:

D1 teacher validation: