Mémo ROS 2

Ingénierie Système et Modélisation Robotique

Rémi RIGAL

RAPPELS ROS

Bashrc

Le fichier *bashrc* a pour chemin `~/.bashrc`

Il est exécuté à chaque fois qu'un terminal est ouvert, il s'agit donc de mettre en place les variables d'environnement nécessaires au fonctionnement de ROS

```
# ROS setup
source /opt/ros/foxy/setup.bash

# Workspace setup
source ~/{workspace}/install/setup.bash
```

Remplacer *{workspace}* par le chemin vers votre workspace actuel de travail
Remplacer *foxy* par le nom de votre distribution ROS si elle est différente

ROS va chercher les packages dans les chemins définis dans la variable d'environnement **AMENT_PREFIX_PATH**

```
~ $ echo $AMENT_PREFIX_PATH
/opt/ros/foxy:/home/remi/ensta_ws/install/my_package
```

Bashrc

La variable **ROS_DOMAIN_ID** permet de définir des domaines ROS différents au sein d'une même machine ou d'un même réseau local. Deux machines ou deux processus peuvent communiquer uniquement si leur **Domain ID** est identique.

Pour configurer cela il faut exporter la variable d'environnement dans le fichier *bashrc* en lui donnant une valeur entière:

```
# Configuring ROS Domain ID
export ROS_DOMAIN_ID=42
```

Il est également possible de contraindre ROS à la machine locale uniquement grâce à la variable d'environnement **ROS_LOCALHOST_ONLY**:

```
# Force localhost
export ROS_LOCALHOST_ONLY=1
```

La compilation

La compilation d'un workspace se fait à l'aide de la commande `colcon build`

```
~ $ cd ROS/ensta_ws
~/ROS/ensta_ws $ colcon build
```

Pendant la phase de développement il est conseillé d'utiliser l'option `--symlink-install`

Cette option créé des liens symboliques pour tous les fichiers non-compilés (Python, XML, YAML, Rviz...) plutôt que de les copier. Il n'est alors pas nécessaire de recompiler le *workspace* à chaque fois que l'un de ces fichiers est modifié. Il est cependant nécessaire de recompiler le *workspace* lorsque l'un de ces fichiers est crée, supprimé ou renommé.

```
~ $ cd ROS/ensta_ws
~/ROS/ensta_ws $ colcon build --symlink-install
```

Les packages C++

Il est possible de créer un package C++ à l'aide de la commande `ros2 pkg create {name}`

Par **convention**, les packages C++ sont nommés en *Snake Case* (mots en minuscule séparés par des underscores "_") et présentent l'arborescence suivante:

```
my_awesome_package/
    config/                 # Fichiers de configuration
        my_config.yaml
    launch/                 # Fichiers launch
        my_launch.launch.py
    msg/                    # Messages
        my_message.msg
    include/                # Headers C++
        my_awesome_package/
            some_script.h
    src/                    # Sources C++
        some_script.cpp
    CMakeLists.txt          # Configuration de la compilation
    package.xml             # Configuration du paquet ROS
```

Les packages Python

Il est possible de créer un package Python à l'aide de la commande:

```
ros2 pkg create --build-type ament_python {name}
```

Par **convention**, les packages Python sont nommés en *Snake Case* (mots en minuscule séparés par des underscores "_") et présentent l'arborescence suivante:

```
my_awesome_package/
    config/                 # Fichiers de configuration
        my_config.yaml
    launch/                 # Fichiers launch
        my_launch.launch.py
    msg/                    # Messages
        my_message.msg
    resource/
        my_awesome_package  # Nécessaire à ROS pour trouver le paquet
    my_awesome_package/     # Sources Python
        some_script.py
    setup.py                # Configuration du paquet Python
    setup.cfg               # Configuration du paquet ROS
    package.xml             # Configuration du paquet ROS
```

Les fichiers Launch

Les fichiers *launch* permettent de configurer le lancement de plusieurs nœuds en même temps:

```python
import os
from launch import LaunchDescription
from launch.substitutions import Command
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():
    pkg_share = FindPackageShare("my_package").find("my_package")
    model_file = os.path.join(pkg_share, "urdf", "robot.urdf.xacro")
    rviz_config_file = os.path.join(pkg_share, "config", "config.rviz")
    robot_state_publisher_node = Node(
        package="robot_state_publisher", executable="robot_state_publisher",
        parameters=[{"robot_description": Command(["xacro", " ", model_file])}]
    )
    rviz_node = Node(
        package="rviz2", executable="rviz2",
        arguments=["-d", rviz_config_file]
    )
    return LaunchDescription([
        robot_state_publisher_node,
        rviz_node
    ])
```

Les nœuds Python

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
from geometry_msgs.msg import Twist, Vector3

class MyNode(Node):

    def __init__(self):
        super().__init__("my_node")
        self.cmd_vel_publisher = self.create_publisher(Twist, "cmd_vel", 10)
        self.str_subscriber = self.create_subscription(String, "command", self.str_callback, 10)

    def str_callback(self, msg):
        if msg.data == "stop":
            self.cmd_vel_publisher.publish(Twist())
        elif msg.data == "start":
            self.cmd_vel_publisher.publish(Twist(linear=Vector3(x=1.0)))
        self.get_logger().info(f"Received message: {msg.data}")

if __name__ == "__main__":
    rclpy.init(args=args)
    node = MyNode()
    rclpy.spin(node)
```

Exécution

Il y a plusieurs manières d'exécuter des nœuds ROS:

- Via la commande `ros2 run {package} {node}`
  - Démarre un unique nœud ROS

- Via la commande `ros2 launch {package} {file.launch.py} [arg:=value]`
  - Permet d'associer des valeurs à des arguments
  - Exécute un ou plusieurs nœud(s) ROS

Ces commandes peuvent être exécutées depuis n'importe quel emplacement, il n'y a pas besoin de se trouver dans le dossier du *package*.

Visualisation

De nombreux outils sont à disposition pour visualiser l'état du robot, l'état de l'infrastructure ROS ou toute information difficilement lisible en texte brut (matrices, vecteurs, images…)

- Rviz via la commande `rviz2`
  - Permet la visualisation d'informations en 3D
  - Affichage des données de tous les types de capteurs

- RQT via la commande `rqt`
  - Fenêtre configurable au besoin avec des plugins
  - Affichage du graphe des nœuds ROS, des topics, de la console…
  - Création de courbes temporelles
  - Publication de données dans des topics

Robot state publisher

Le **robot state publisher** est un nœud ROS qui à partir d'un fichier **URDF** et de l'état des joints publié dans le topic */joint_states* recrée l'arborescence géométrique du robot. En d'autres termes, il calcule les translations et rotations entre les repères de tous les links en temps réel. Il publie également la description du robot dans le topic */robot_description*.

Ce nœud peut être lancé de deux manières:

- Avec un rosrun

```
ros2 run robot_state_publisher robot_state_publisher
```

- Dans un fichier launch

```
robot_state_publisher_node = Node(
    package="robot_state_publisher", executable="robot_state_publisher",
    parameters=[{"robot_description": Command(["xacro", " ", model_file])}]
)
```

Joint state publisher

Le **joint state publisher** est un nœud ROS qui à partir d'un fichier **URDF** publie des valeurs d'angle fictives pour chaque *joint* non fixe du robot dans le topic */joint_states*. Il permet de vérifier l'intégrité d'un robot lorsque ces valeurs ne sont pas disponibles.

Ce nœud peut être lancé de deux manières:

- Avec un rosrun

```
ros2 run joint_state_publisher joint _state_publisher
```

- Dans un fichier launch

```
joint_state_publisher_node = Node(
    package="joint_state_publisher", executable="joint_state_publisher"
)
```

Commandes utiles

Quelques commandes utiles:

- La liste des topics `ros2 topic list`
- Les infos sur un topic `ros2 topic info /{topic}`
- Les données publiées sur un topic `ros2 topic echo /{topic}`
- Diagnostique global `ros2 doctor`
- La liste des nœuds actifs `ros2 node list`
- Vérifier un fichier URDF `check_urdf file.urdf`
- Afficher le graphe des nœuds `rqt_graph`

MODÉLISATION & GAZEBO

Rappels sur le XML

Le langage XML est utilisé pour la syntaxe de nombreux fichiers dans ROS comme les fichiers URDF, launch, package.xml etc…

Pour rappel, une balise XML est définie comme suit:

```
<tag attribute="value">
        content
</tag>
```

ou

```
<tag attribute="value"/>
```

Elle possède forcément un **tag**, peut posséder un ou plusieurs **attribut(s)** et peut avoir un **contenu** qui peut lui même être une ou plusieurs balise(s) XML.

L'indentation et les sauts de ligne ne sont pas importants.

URDF

Les fichiers **URDF** permettent la description physique du robot à l'aide de **links** (éléments rigides du robot) et de **joints** (articulations entre les links). Ils sont rédigés en XML.

```xml
<?xml version="1.0"?>
<robot name="robot">

  <link name="root_link">
    <visual>
      <geometry><box size="0.4 0.2 0.1"/></geometry>
    </visual>
  </link>

  <joint name="root_to_sphere_joint" type="fixed">
    <parent link="root_link"/>
    <child link="sphere_link" />
    <origin xyz="0 0 1" rpy="0 0 0"/>
  </joint>

  <link name="sphere_link">
    <visual>
      <geometry><sphere radius="0.2"/></geometry>
    </visual>
  </link>

</robot>
```

Xacro

Le **Xacro** est identique à l'URDF mais rajoute la possibilité d'évaluer des expressions mathématiques, de créer des macros et d'inclure d'autres fichiers Xacro ou URDF.

```
<?xml version="1.0"?>
<robot name="robot" xmlns:xacro="http://ros.org/wiki/xacro">

  <xacro:include filename="other_file.xacro"/>

  <xacro:macro name="box_geometry" params="sizeX sizeY sizeZ">
    <geometry>
      <box size="${sizeX} ${sizeY} ${sizeZ}"/>
    </geometry>
  </xacro:macro>

  <link name="cool_box_link">
    <visual>
      <xacro:box_geometry sizeX="1.0" sizeY="1.0" sizeZ="0.2"/>
    </visual>
    <collision>
      <xacro:box_geometry sizeX="1.0" sizeY="1.0" sizeZ="0.2"/>
    </collision>
  </link>

</robot>
```

Gazebo Spawn Entity

Spawn Entity est un script du package *gazebo_ros* permettant d'instancier des éléments dans Gazebo, on peut l'utiliser de deux manières:

- Avec un rosrun

```
ros2 run gazebo_ros spawn_entity.py –entity {name} –x 0 –y 0 –z 0 –topic /robot_description
```

- Dans un fichier launch

```
spawn_entity_node = Node(
    package='gazebo_ros', executable='spawn_entity.py',
    arguments=['-entity', 'robot_name', '-topic', '/robot_description']
)
```

Dans les deux cas les arguments *x*, *y* et *z* sont optionnels, ils déterminent la position où sera instanciée le modèle.