Mémo ROS

UV 5.8 – 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/kinetic/setup.bash

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

Remplacer *{workspace}* par le chemin vers votre workspace actuel de travail
Remplacer *kinetic* 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 **ROS_PACKAGE_PATH**

```
~ $ echo $ROS_PACKAGE_PATH
/opt/ros/kinetic/share:/home/remi/ROS/ensta_ws/src
```

Bashrc

Utiliser ROS avec plusieurs machines implique de définir un **ROS Master**, c'est la machine arbitre qui connait toute l'infrastructure et notamment la répartition des nœuds sur les ordinateurs du réseau, et la répartition des topics à travers les noeuds.

Pour configurer cela il faut définir les variables d'environnement **ROS_HOSTNAME** et **ROS_MASTER_URI** dans le fichier bashrc:

```
# Configuring ROS Master
export ROS_HOSTNAME={LOCAL_IP}
export ROS_MASTER_URI=http://{ROS_MASTER_IP}:11311
```

Remplacer *{LOCAL_IP}* par l'identité réseau de votre ordinateur
Remplacer *{ROS_MASTER_IP}* par l'identité réseau du ROS Master
Dans le cas d'une architecture locale, les deux valeurs peuvent être identiques

L'identité réseau d'un ordinateur peut être son **adresse IP**, son **hostname** si il est déclaré dans le réseau, ou ***'localhost'*** lorsque tout est lancé sur la même machine

La compilation

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

```
~ $ cd ROS/ensta_ws/
~/ROS/ensta_ws $ catkin_make
```

En cas d'erreur de compilation inexpliquée, la solution est souvent de supprimer les fichiers compilés et de recommencer la compilation:

```
~/ROS/ensta_ws $ rm -r build/ devel/
~/ROS/ensta_ws $ catkin_make
```

Les packages

Il est possible de créer un package à l'aide de la commande `catkin_create_pkg {name}`

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

```
my_awesome_package/
    config/             # Contient les fichiers de configuration
        my_config.yaml
    launch/             # Contient les fichiers launch
        my_launch.launch
    msg/                # Contient les messages
        my_message.msg
    scripts/            # Contient les scripts
        my_script.py
    CMakeLists.txt      # Permet de configurer les options de compilation
    package.xml         # Permet de configurer le paquet ROS
```

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.

Les fichiers Launch

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

```xml
<?xml version="1.0"?>
<launch>

    <!-- Arguments -->
    <arg name="start_paused" default="true"/>
    <arg name="start_rviz" default="false"/>

    <!-- Param -->
    <param name="robot_description" command="$(find xacro)/xacro --inorder $(find robot_description)/urdf/robot.xacro"/>

    <!-- Include -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <!-- Override arguments of empty_world.launch -->
        <arg name="world_name" value="worlds/willowgarage.world"/>
        <!-- Override arguments with local argument -->
        <arg name="paused" value="$(arg start_paused)"/>
    </include>

    <!-- Node -->
    <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-urdf -model robot -param robot_description"/>

    <!-- Node group with condition -->
    <group if="$(arg start_rviz)">
        <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">
            <!-- Node param -->
            <param name="publish_frequency" type="double" value="50.0" />
        </node>
        <node name="rviz" pkg="rviz" type="rviz"/>
    </group>

</launch>
```

Exécution

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

- Via la commande `rosrun {package} {script}`
  - Nécessite que le ROS Master ait démarré un **roscore**
  - Démarre un unique nœud ROS

- Via la commande `roslaunch {package} {file.launch} [arg:=value]`
  - Démarre un **roscore** automatiquement aucun n'est démarré
  - Permet d'associer des valeurs à des arguments
  - Exécute un ou plusieurs nœud(s) ROS

Ces commandes peuvent être exécuté 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 `rviz`
  - 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.

Ce nœud peut être lancé de deux manières:

- Avec un rosrun

```
rosrun robot_state_publisher robot_state_publisher
```

- Dans un fichier launch

```
<node pkg="robot_state_publisher" type="robot_state_publisher"
name="rsp"/>
```

Les commandes pratiques

Quelques commandes pratiques:

- La liste des topics `rostopic list`
- Les infos sur un topic `rostopic info /{topic}`
- Les données publiées sur un topic `rostopic echo /{topic}`
- Le chemin vers un package `rospack find {package}`
- Se déplacer dans le dossier d'un package `roscd {package}`
- Vérifier un fichier URDF `check_urdf file.urdf`
- Afficher le graphe des nœuds `rqt_graph`

MODÉLISATION & GAZEBO

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>
```

Le param robot_description

Par convention, le param **robot_description** contient la description en URDF du robot. Pour associer ce **param** dans un fichier launch il y a deux possibilités:

- Pour un fichier URDF

```
<param name="robot_description"
       textfile="$(find {package})/urdf/file.urdf"/>
```

- Pour un fichier Xacro

```
<param name="robot_description"
       command="$(find xacro)/xacro --inorder $(find {package})/urdf/file.xacro"/>
```

Gazebo Model Spawner

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

- Avec un rosrun

```
rosrun gazebo_ros spawn_model -urdf -model {name} -x 0 -y 0 -z 0
-param robot_description
```

- Dans un fichier launch

```
<node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf"
args="-urdf -model {name} -x 0 -y 0 -z 0 -param
robot_description"/>
```

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

Liens utiles

Tutoriels officiels ROS: http://wiki.ros.org/ROS/Tutorials

Syntaxe des fichiers launch: http://wiki.ros.org/roslaunch/XML/launch

Syntaxe des fichiers URDF: http://wiki.ros.org/urdf/XML

Syntaxe du Xacro: http://wiki.ros.org/xacro