Robotique/V2X UTAC Développement d'un nouveau mode conduite dans un convoi autonome

Joseph LOUVILLE

1^{er} rapport bimensuel de stage.

Tuteurs :

Mme Bouchemal
M. Jun Kim

Fait à Paris, le 28/04/2022

Le constat actuel est que :
— Les 4 robots sont codés individuellements et non de manière générique.
— La version de ROS est le ROS 1 kinetic qui a été publié en 2016.
— La version de Gazebo "8.6.0-1"
— Catkin_tools 0.6.1 (Python 2.7.12)
— Essaie de faire fonctionner le projet, impossible de lancer le **roslaunch ecebot ecebot_turtle3.launch** pour lancer la simulation ⇒ possible problème au niveau du package créé.

On peut créer un package grâce à la commande **catkin_create_pkg** dans le fichier dossier src.

Pour ce qui est de la compilation des paquets, une seule commande est nécessaire à partir du répertoire du workspace :

```
cd ~/catkin_ws/
catkin_make
```

`https://roboticsbackend.com/ros-include-cpp-header-from-another-package/` est un site qui explique comment mettre en place sur ROS des headers pour les fichiers Cpp.

`https://homepages.laas.fr/ostasse/Teaching/ROS/rosintro.pdf` site tuto mis en place du catkin workspace.

Kinetic Kame (May 2016 - May 2021)

Required Support for :
— Ubuntu Wily (15.10)
— Ubuntu Xenial (16.04)
Recommended Support for :
— Debian Jessie
— Fedora 23
— Fedora 24
Minimum Requirements :
— C++11
— GCC 4.9 on Linux, as it's the version that Debian Jessie ships with
— Python 2.7
— Python 3.4 not required, but testing against it is recommended
— Lisp SBCL 1.2.4
— CMake 3.0.2
— Debian Jessie ships with CMake 3.0.2
— Boost 1.55
— Debian Jessie ships with Boost 1.55

Exact or Series Requirements :
— Ogre3D 1.9.x
— Gazebo 7
— PCL 1.7.x
— OpenCV 3.x
— Qt 5.3.x
— PyQt5
Build System Support :
— Same as Indigo

Problème d'exécution lors de la compilation **catkin_make** . L'erreur renvoyé par la console est : **fatal error custom_msg.h : no such file or directory**. Déplacement du workspace : pas de résultat.

essaie de l'utilisation de **catkin_make_isolated** : meilleure compilation, mais incomplète.

J'ai fait appelle à Alexandre Ségarat, l'un des créateurs du PFE, pour qu'il puisse m'aider. Il m'a dit qu'il se mettrait en contact avec les anciens de son groupe pour répondre à la question du problème de compilation du projet ROS.

J'ai vu que le problème venait dans la génération des messages à partir des fichiers .msg. J'ai vu qu'il existait des cas similaires, mais dont les solutions sont déjà utilisé dans le projet.

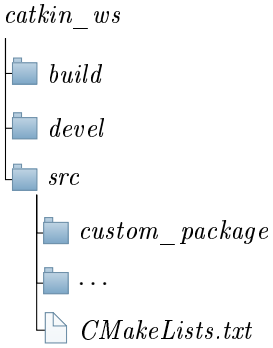
J'ai finalement trouvé la solution à mon problème. Il semblerait que la génération de messages lors de la compilation ne se faisait pas. Je pense que le problème est du au téléchargement depuis Github, qui a pu corrompre les fichiers de type .msg, ou alors le problème viendrait du catkin, possiblement parce que les dossiers auraient été fait avec une version plus ancienne. Je conseil donc de regénérer un environnement catkin et de transférer les fichiers contenant du code de l'ancien environnement vers le nouveau.

L'une des solutions proposés sur des forums étaient d'ajouter des dependencies dnas le fichier CMakeLists.txt afin que le package compile d'abord les .msg avant les fichiers .cpp et .h.

```
add_dependencies(custom_package custom_msg_generate_messages_cpp)
```

Aussi plusieurs packages manquait lors du lancement du programme.
Packages à installer par apt-get :
— curl
— turtlesim (ros-kinetic-turtlesim)
— robot-state-publisher (ros-kinetic-robot-state-publisher)
— turtlebot3 (ros-kinetic-turtlebot3-*)
— std_msgs (ros-std-msgs)(utiliser le paramètre -y pour cette installation)
— xacro (ros-kinetic-xacro)

Revenons sur le design de catkin :
Le workspace a cette aspect initialement :



catkin_ws va être l'espace où le code va être compiler et générer les programmes d'exécution vers les dossiers **build** et **devel**. Pour créer nos packages et mettre notre code ROS, il faut aller dans src. pour créer un package, il faut utiliser la commande :

```
~$catkin_create_pkg[nom_du_package] dependance_1 dependance_2 ...
```

Dans le package, on trouve :

Les dossiers **include** et **src** sont utilisés pour intégrer les fichiers sous format .h .cpp et .py, qui vont contenir les formules mathématiques et les commandes ROS. Pour ce qui est des messages, il est recommandé de faire un dossier spécialisé pour stocker les fichiers .msg. Cette chaîne youtube parle très bien du développement et de la création de projet ROS `https://www.youtube.com/channel/UC5ZQinPsJ4C8YiauoT8xZUg`

Afin de faire compiler le projet il faut appeler la fonction **catkin_make** depuis le dossier root, ici dans notre exemple **catkin_ws**. Si les dossiers build et devel ont été importés ou déplacés, il est nécessaire de les supprimés avant de compiler le projet.

A partir de ce point normalement, on doit désigner comme source de projet le fichier **setup.bash** dans le dossier **devel** afin de pouvoir faire appel des fonctions de ROS dans la fenêtre de commande et qu'elles s'appliquent au projet compilé.

Les fonctions ROS intéressantes que j'ai pu voir sont **roslaunch [launch_package] fichier_launch.launch**, qui permet de lancer la simulation d'un package qui possède un fichier .launch associé, **rosrun nom_de_node**, qui permet de faire fonctionner une node (nœud).

Afin de lancer le projet il est indispensable que toutes les nodes soient lancées. Donc pour lancer le projet il existe plusieurs possibilité : soit faire l'appel de tous les nodes dans le fichier .launch, c'est à dire :

```
<!--in ecebot ecebot_turtle.launch-->

<node pkg="ecebot" name="path_turtle" type="path_turtle.py" output="screen"/>
<node pkg="vehicles" name="voiture_1" type="voiture_1" output="screen"/>
<node pkg="vehicles" name="voiture_2" type="voiture_2" output="screen"/>
<node pkg="vehicles" name="voiture_3" type="voiture_3" output="screen"/>
<node pkg="vehicles" name="voiture_4" type="voiture_4" output="screen"/>
<node pkg="controler" name="controler" type="controler" output="screen"/>
<node pkg="light" name="light" type="light" output="screen"/>
<node pkg="ecebot" name="following_turtle_2"
                    type="following_turtle_2.py"args="2" output="screen"/>
<node pkg="ecebot" name="following_turtle_3"
                    type="following_turtle_3.py" args="3" output="screen"/>
<node pkg="ecebot" name="following_turtle_4"
                    type="following_turtle_4.py" args="4" output="screen"/>
```

Cela permet juste d'appeler un **roslaunch** afin de lancer le projet. L'autre façon de faire est d'appeler chaque node via des rosrun.

```
roslaunch ecebot ecebot_turtle.launch
rosrun vehicles voiture_1
rosrun vehicles voiture_2
rosrun vehicles voiture_3
rosrun vehicles voiture_4
rosrun light light
```

Après avoir appelé ces commandes, Gazebo se lance et il est possible de faire fonctionner la simulation en appuyant sur le bouton play.

0.1 VM

Projet du PFE n°2026 composé de Margot IRLINGER, David OLIVARES, Emma PALFI, Jean PROUVOST, Alexandre SEGERAL, Jimmy VUONG.

Commande à utiliser dans un terminal :

```
$ cd /Bureau/essai_ws
$ source devel/setup.bash
$ roslaunch ecebot ecebot_turtle.launch
```

Il faut ensuite appuyer sur play sur Gazebo pour lancer la simulation.

Projet de la valorisation du groupe composé de Adrien Mailly, Timothé Bruckert, Florent Fonsalas, Benjamin Hubau, Rémi Breton, Valentin Martins.

Commande à utiliser dans un terminal :

```
$ cd /Bureau/essai_ws
$ source devel/setup.bash
$ roslaunch ecebot ecebot_turtle.launch
```