TP 1 : Fonctions de bases sous ROS

Thomas Le Mézo - ENSTA Bretagne

2019

Ce TP est fortement inspiré des tutoriels ros officiels disponibles ici : `wiki.ros.org/ROS/Tutorials/`. Il vous est fortement recommandé d'utiliser un IDE (Qt ROS par exemple).

1 Contexte du TP

Nous allons modéliser et construire l'architecture logicielle d'un bateau équipé d'une nacelle robotisée. Cette dernière pourra accueillir différents capteurs optiques ou acoustiques par exemple. Le robot sera modélisé par le vecteur d'état (x, y, θ_{robot}) et la nacelle par son cap et son élévation (ou tangage) $(\theta_{tourelle}, \phi)$.

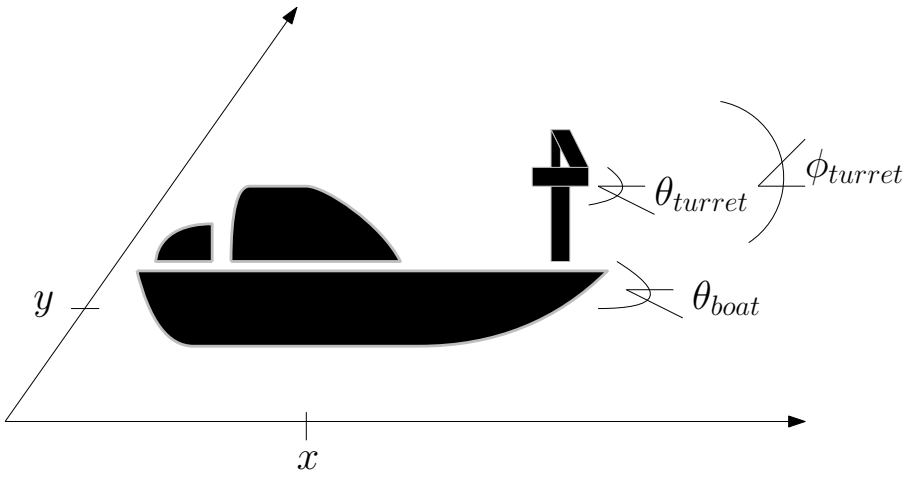

FIGURE 1 – Schéma d'un bateau avec une nacelle robotisée

Au cours des différents TP, nous réaliserons une modélisation et une implémentation d'une partie de ce système sous ROS.

2 Modélisation C2

Question 1 *Proposez, sur papier, une architecture fonctionnelle C2 haut-niveau (i.e. simplifiée) modélisant le système. Quelle est la différence avec une architecture physique de l'électronique du système.*

3 Découverte des topics et services

Dans cette partie, nous allons utiliser les fonctionnalités de ROS en ligne de commande dans un terminal.

D'une manière générale, il est rappelé que la touche **TAB** du clavier permet d'obtenir une auto-complétion très efficace en particulier avec les programmes ROS.

3.1 Les nodes

— Dans un terminal tapez la commande `roscore` qui permet de lancer le *master*.

— Utilisez la commande `rosnode list` dans un autre terminal :
Question 2 *Combien de nodes apparaissent ?*

— Vous pouvez utiliser la commande `rosnode info nom_du_node` pour obtenir des informations sur ce dernier.

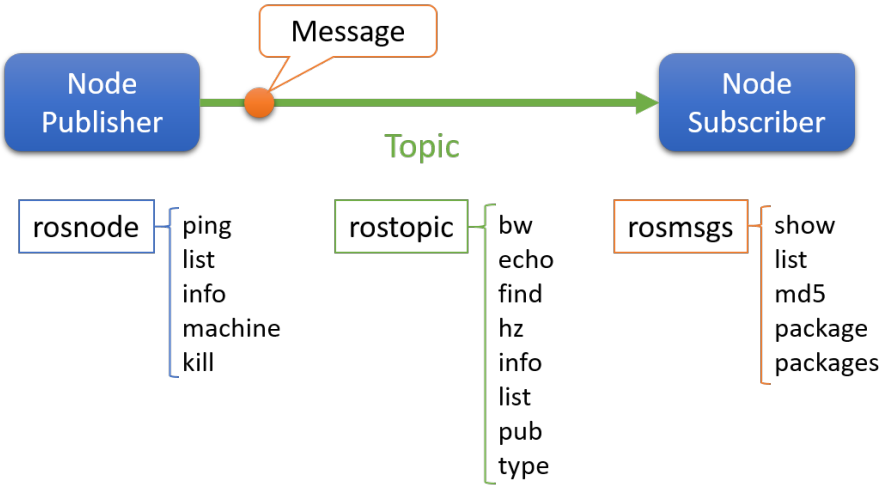

FIGURE 2 – Topics

Question 3 *Quelles sont le(s) topic(s) de publication du node rosout ?*
D'une manière générale, la commande `rosnode [option]` permet d'obtenir des informations sur les nodes.

3.2 Les topics

Vous allez maintenant publier et souscrire à votre premier *topic*.
Dans un premier terminal tapez la commande suivante pour publier un message sur le topic *boat_name* :

```
rostopic pub -r 10 /boat_name std_msgs/String hello
```

Et dans un deuxième terminal tapez la commande suivante pour écouter les messages arrivant sur le topic *boat_name* :

```
rostopic echo /boat_name
```

Question 4 *Avec l'aide de la cheatsheet ou de l'aide dans la console (option -h des commandes), changez la féquence, le texte publié et le nom du topic.*

ROS a prévu un certain nombre de type de base pour les messages. Le package `std_msgs` regroupe les types les plus couramment utilisés.
La commande `rospack find std_msgs` permet de localiser sur votre ordinateur ce package. Dans le répertoire *msg* du package `std_msgs` vous trouverez la liste des messages standards. La structure est normalement de la forme suivante pour les messages :

```
type_de_la_variable1 nom_de_la_variable1
type_de_la_variable2 nom_de_la_variable2
...
```

Dans les messages standard, la variable s'appelle *data*.
Pour les services, le fichier est en deux partie : la demande, trois tirets "---", puis la réponse. Un même fichier sert à définir l'échange. Les messages standards sont regroupés dans le package `std_srvs`.
Vous pouvez utiliser les commandes suivantes pour obtenir des informations sur la structure d'un message ou d'un service.

```
rosmsg show nom_message
rossrv show nom_service
```

Question 5 *Donnez la structure des messages de type "Float64" et "Imu" ainsi que du service de type "Trigger" ainsi que le nom du package de ce dernier.*

Question 6 *Essayez maintenant de publier le cap du bateau que vous supposerez fixe et égal à 90.0 degrés sur le topic "/cap".*

3.3 Les Services

De manière identique, il existe des commandes pour accéder aux services : nous les étudierons un peu plus tard dans le TP.

4 Workspace et package ROS

4.1 Workspace

Nous allons maintenant créer notre workspace ROS. La création du workpackage est assurée par un script de l'utilitaire **catkin**. Catkin est une surcouche de cmake créée par les développeurs de ROS qui permet d'automatiser un certain nombre de tâches dans les workspaces (compilation, création de nouveaux paquets etc.)

Pour créer votre workspace, qui vous servira pour l'ensemble du cours, exécutez les commandes suivantes :

```
mkdir -p ~/workspaceRos/src
cd ~/workspaceRos
catkin_make
```

Nous allons maintenant compiler pour la première fois notre workspace (pour l'instant il n'y a aucun fichier de code à l'intérieur) avec `catkin_make` qui est la commande de base pour compiler un package ROS.
ATTENTION 1 (erreur classique) : Contrairement à Python, les codes en C++ doivent systématiquement être compilés pour pourvoir être exécutés.
ATTENTION 2 (erreur classique) : Placez-vous à la racine de votre workspace pour lancer la commande de compilation `catkin_make` sinon cela ne fonctionnera pas !

Question 7 *Observez la structure de fichiers créée par catkin : retrouvez-vous celle décrite dans le cours ?*

Lors de la compilation, ROS génère une série de fichiers dans les répertoires *build* et *devel*. Pour permettre aux autres programmes de savoir où se trouvent ces fichiers sur votre ordinateur, il est nécessaire d'utiliser un mécanisme qui fournit le chemin d'accès (ie le *path*) à ces fichiers.

Le script `setup.bash` crée dans le répertoire *devel* de votre workspace au moment de la compilation a justement ce rôle : il permet d'ajouter aux variables d'environnement de votre terminal, l'endroit où se trouvent vos fichiers.

Pour automatiser l'exécution du script "setup.bash", il suffit d'ajouter une ligne dans le fichier `~/.bashrc`. Le script `~/.bashrc` est exécuté à chaque lancement d'un terminal.

```
echo "source ~/workspaceRos/devel/setup.bash" >> ~/.bashrc
```

Question 8 *Ouvrez votre fichier `.bashrc` et vérifier que la ligne de commande permettant de faire un source*[1] *du setup.bash a bien été ajoutée.*

ATTENTION 3 (erreur classique) : Après chaque compilation, pour pouvoir exécuter vos programmes ROS, il faut indiquer à votre terminal où se trouvent les fichiers exécutables. Suivant les cas, il peut être nécessaire de (re)faire un `source` du fichier `setup.bash`.

4.2 Premier package

Nous allons créer notre premier package. Une commande de **catkin** permet d'automatiser la création et les dépendance du package. Pour ce premier package que nous appellerons `tp1`, nous allons annoncer à catkin que nous voulons utiliser le package `std_msgs` pour avoir accès aux types de bases et `roscpp` pour pouvoir coder et compiler en C++.
On définit ainsi les dépendances de notre package à des packages pré-existants. Ce mécanisme de dépendance est une des bases de ROS qui permet de réutiliser du code déjà disponible.

```
cd ~/workspaceRos/src
catkin_create_pkg tp1 std_msgs roscpp
```

Pour programmer en Python, il faudrait également ajouter `rospy` à la liste.

Question 9 *Vérifiez que la structure du package est identique à celle présentée dans le cours.*

Note : Vous pouvez également remplir le fichier `package.xml` en renseignant entre autre les auteurs et le type de licence du code. Ce type d'informations est utile lorsque vous diffusez votre code.

1. `https://en.wikipedia.org/wiki/Source_(command)`

5 Coder son premier Publisher

5.1 Fichier Cpp

Créez un nouveau fichier `talker.cpp` dans le répertoire `src` du package `tp1`.
En vous inspirant du code suivant, re-codez dans un node, l'exemple de la question 6.

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
// Attention à bien inclure chaque type de message !

int main(int argc, char **argv){
        // Initialisation du node : le troisième argument est son nom
        ros::init(argc, argv, "nom_du_node");

        // Connexion au master et initialisation du NodeHandle qui permet d'avoir accès aux topics et services
        ros::NodeHandle n;

        // Création du publisher avec
        // - le type du message
        // - le nom du topic
        // - la taille du buffer de message à conserver en cas de surchage
        ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);

        // La durée de la pause (voir le sleep)
        ros::Rate loop_rate(10);

        // Boucle tant que le master existe (ros::ok())
        while (ros::ok()){
                // création d'un message de type String
                std_msgs::String msg;

                // affectation la valeur "hello" au champ data
                msg.data = "hello";

                // publication du message
                chatter_pub.publish(msg);

                // fonction utile seulement dans le cas de l'utilisation d'un subscriver ou d'un server
                ros::spinOnce();

                // Pause
                loop_rate.sleep();
                // Il est également possible d'utiliser des Timers qui fonctionnent par interruption
                // http://wiki.ros.org/roscpp_tutorials/Tutorials/Timers
        }
        return 0;
}
```

REMARQUE : Il est possible d'avoir plusieurs *publishers* ou *subscrivers* dans un même *node*.

5.2 Compilation du publisher

La première étape pour compiler notre node est d'ajouter les informations suivante dans le `CMakeLists.txt` :

- Pour compiler le node (ajouter la ligne après les commentaires dans le fichier, lignes 132 et 147) :

```
## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
# add_executable(${PROJECT_NAME}_node src/tp1_node.cpp)
add_executable(talker src/talker.cpp)
```

- Pour linker le node (idem) :

```
## Specify libraries to link a library or executable target against
# target_link_libraries(${PROJECT_NAME}_node
#   ${catkin_LIBRARIES}
# )
target_link_libraries(talker
        ${catkin_LIBRARIES}
)
```

Ces deux opérations devrons être systématiquement effectuées avant de compiler un nouveau node. Vous pouvez alors compiler votre workspace :

```
cd ~/workspaceRos
catkin_make
```

5.3 Test

Pour lancer votre node, vérifiez que roscore est toujours lancé, puis lancez la commande suivante :

```
rosrun tp1 talker
```

Question 10 *Utilisez les outils de la partie précédente pour écouter votre node (`rostopic echo`) et vérifiez que votre node apparait bien (`rosnode list`).*

Question 11 *La fonction suivante `ros::Time::now().toSec()` retourne le temps sous forme de double. Reprenez votre node précédent pour que le cap publié suive une sinusoïde fonction du temps.*

6 Coder son premier Subscriber

La particularité de ROS, concernant les Subscrivers, est l'utilisation des fonctions de callback. Le principe est le suivant :

1. Le node vérifie si de nouveaux messages sont disponibles sur le topic (fonction `ros::spinOnce()`)
2. Pour chaque message reçu, ROS exécute une fonction dit de *callback* associée au topic et à son type. Cela permet d'accéder facilement aux données du message et de déclencher un traitement associé à sa réception.

Question 12 *De manière identique à la question de la partie précédente, codez un node subscriber au cap du bateau en vous inspirant du code ci-dessous.*

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

// Fonction de callback appelée lorsqu'un message est reçu
// Le paramètre d'entrée dépend du type de message sur le topic
// il est systématiquement de la forme :
// const package::type_du_message::ConstPtr&
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
        // ROS_INFO permet d'afficher des chaines de caractère dans la console.
        // Cette fonction est très utile pour le débuggage.
        // Il en existe d'autre type en fonction de la gravité de l'information
        // (ROS_DEBUG, ROS_WARN, ROS_FATAL, ... )
        // http://wiki.ros.org/roscpp/Overview/Logging
        ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
        ros::init(argc, argv, "listener");
        ros::NodeHandle n;
```

```
        // On ajoute la fonction de callback par rapport au Publisher
        // Le type du message est déduit du paramètre d'entrée de la fonction de callback.
        ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

        // Vérifie si de nouveaux messages sont arrivés
        // spin est une boucle SANS FIN qui vérifie si de nouveaux messages sont arrivés
        // (contrairement à spinOnce qui exécute seulement l'action une fois)
        ros::spin();
        return 0;
}
```

7 Coder son premier Service

Dans cette partie nous allons coder notre premier service. Nous utiliserons les services standards sous ROS (voir la liste des types de services disponibles avec `rossrv package std_srvs`).

Nous allons utiliser le type `Trigger` de `std_srvs` qui n'a pas de variable pour la question et répond un booléen et une chaine de caractère (voir `rossrv show std_srvs/Trigger`).

Question 13 *En vous inspirant du code suivant, ajoutez au node publisher codé précédemment un service qui renvoie le nom du bateau et l'état du moteur (on ou off).*

Les services sont toujours formés d'une partie "*Request*" et d'une partie "*Response*". Une fonction de callback est appelée à chaque demande de service pour remplir la partie *Response* à partir de la *Request*.

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

// Fonction exécutée lorsque le Service est appelé
bool service_callback(std_srvs::Trigger::Request  &req,
                      std_srvs::Trigger::Response &res)
{
        res.success = true;
        res.message = "petit bateau";
        return true; // Valide que le service s'est bien exécuté (seulement utile en local pour le node)
}

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

        ros::ServiceServer service = n.advertiseService("service_name", service_callback);
        ros::spin();

        return 0;
}
```

Question 14 *Testez votre server avec les lignes de commandes disponibles dans la Cheatsheet.*

8 Coder son premier Client

Question 15 *De la même manière que dans la partie précédente, inspirez vous du code suivant pour interroger votre server de la partie précédente.*

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

int main(int argc, char **argv)
{
```

```
        ros::init(argc, argv, "client_node");
        ros::NodeHandle n;
        ros::ServiceClient client = n.serviceClient<std_srvs::Trigger>("service_name");

        std_srvs::Trigger srv;
        // La partie request du trigger étant vide,
        // il n'est pas nécessaire de la remplir
        // dans le cas contraire, on peut accéder à srv.request.data = ...

        if (client.call(srv)){ // Appel du service
                ROS_INFO("message: %s", srv.response.message.c_str());
                ROS_INFO("success: %d", srv.response.success);
        }
        else{
                // Dans le cas où le service ne répond pas
                // l'appel est bloquant, cela permet de continuer
                // l'exécution si le seriuce est indisponible
                ROS_ERROR("Failed to call service");
        }
        return 0;
}
```

9 Personnaliser ses messages et services

Lorsque les messages de base ne sont plus suffisants, vous pouvez créer vos propres messages et services. Pour cela, il suffit d'ajouter un fichier `votre_msg.msg` ou `votre_srv.srv` dans un répertoire `msg`, respectivement `srv` de votre package que vous aurez préalablement créé.

— pour les messages

```
type nom_de_la_variable
```

— pour les services (la question et la réponse sont séparés par trois tirets "---")

```
# Question
type nom_de_la_variable
---
# Réponse
type nom_de_la_variable
```

Vous devez indiquer à **catkin** qu'il faut compiler ces fichiers de définition dans le `CMakeList.txt` et le `package.xml` de votre package. La dépendance à d'autres packages doit également être indiquée (par exemple lorsque vous imbriquez des messages dans d'autres, voir la remarque plus bas) :

```
find_package(
        message_generation # À ajouter
        std_msgs # Ou autre si il existe une dépendance
)

###########
add_message_files(
        FILES
        votre_msg.msg
)

add_service_files(
        FILES
        votre_service.msg
)

generate_messages(
        DEPENDENCIES
```

```
        std_msgs # Ou autre si il existe une dépendance
)

###########
catkin_package(
        CATKIN_DEPENDS
                roscpp
                std_msgs # Ou autre
                message_runtime # A ajouter
)
```

Dans le fichier `package.xml`, vérifiez que les lignes suivantes sont dé-commentées pour préciser que vous compilez des messages dans ce package et que vous les utiliserez pendant l'exécution.

```
<build_depend>message_generation</build_depend>
<depend>message_runtime</depend>

<depend>std_msgs</depend>
```

Vous pouvez vous inspirer de la page suivante `http://wiki.ros.org/msg` pour composer vos message : elle recense les types de base que vous pouvez utiliser (attention aux majuscules : par convention la première lettre du fichier est en majuscule !).
Remarque : il est possible d'utiliser les types définis dans d'autres packages mais vous devrez bien spécifier le nom du package dans les dépendances. Par exemple, vous pouvez jeter un coup d'œil aux messages du package `sensors_msgs` de ros : `http://docs.ros.org/api/sensor_msgs/html/index-msg.html`

Question 16 *Créez un node publiant un message personnalisé contenant le cap et l'élévation de la nacelle.*

Remarque : pour pouvoir utiliser les messages crées, il faut penser à inclure le message au début de votre code.

```
#include "tp1/Votre_message.h"
// Et systématiquement "tp1::Votre_message" dans la déclaration des topics, services etc.
```