Laurent LEQUIEVRE – Ingénieur CNRS - Institut Pascal UMR6602

Equipe Maccs http://ip.univ-bpclermont.fr/index.php/fr/maccs

Juan Antonio CORRALES RAMON - Youcef MEZOUAR

Notions de ROS nodes, messages, topics, services, rosparam, roslaunch

Un 'node' est un exécutable qui se trouve dans un paquet ROS. Il utilise ROS afin de communiquer avec d'autres 'nodes'. Plus exactement, il utilise une bibliothèque 'client' ROS afin de communiquer. Cette bibliothèque est écrite en C++ 'roscpp' mais aussi en python 'rospy'.
Cette communication se fait via des messages qui sont envoyés par des 'topics'. Les messages sont en quelque sorte des types de données de ROS, les 'topics' pouvant être assimilés à des bus de communication.
On dit que les 'nodes' publient sur des 'topics', mais ils peuvent aussi souscrire à des 'topics' pour recevoir leurs messages.
L'envoie et la réception de messages se fait de manière asynchrone.

Un 'service' ROS est une autre manière qu'ont les 'nodes' de communiquer. Ce coup-ci un 'node 1' peut envoyer une demande ('request') à un autre 'node 2', le 'node 1' attendra une réponse ('response') du 'node 2'. C'est en quelque sorte un appel 'remote' de fonctionnalités d'un 'node' à partir d'un autre 'node'.

1- Créer un simple node en C++ qui affichera la valeur d'un compteur incrémenté.

Dans votre 'workspace', créer un paquet nommé 'my_first_node' avec une dépendance à 'roscpp'.
Dans le dossier 'src' du paquet créer un fichier C++ 'my_node.cpp' contenant ceci :

```
/**
 ** Simple ROS Node
 **/
#include <ros/ros.h>


int main(int argc, char* argv[])
{
   // This must be called before anything else ROS-related
   ros::init(argc, argv, "my_name_node");

   // Create a ROS node handle
   ros::NodeHandle node;

   // Set the rate at which we print out our message (1Hz)
   ros::Rate loop_rate(1.0);

   // A simple counter for the number of times we iterate through the loop
   int count = 0;

   // Loop through until the ROS system tells the user to shut down
   while(ros::ok()) {
      // Print out a message
      ROS_INFO_STREAM("We've gone through " << count << " times.");
      ++count;
      // Wait the stated duration
      loop_rate.sleep();
   }

   // Exit the program.
   return 0;
}
```

ros::init(argc, argv, "my_node") :
permet d'initialiser le 'node' avec les paramètres ligne de commande 'argv'. On lui passe aussi le nom du 'node', ici 'my_name_node', qui doit être unique.
ros::NodeHandle node :
A la première création d'un 'NodeHandle', cela lance implicitement ***ros::start()***, ce qui permet de démarrer le 'node' (ici son nom est 'my_name_node').
A la destruction du dernier 'NodeHandle', cela arrête le 'node' en lançant implicitement ***ros::shutdown()***. 'NodeHandle' gère pour cela un compteur interne.

L'objet 'NodeHandle' ainsi créé, permet un point d'accès au système ROS (par exemple la communication avec d'autres 'nodes' via des 'topics', ou encore récupérer des paramètres stockés dans le 'ros parameter server').

ros::Rate loop_rate(1.0) et **loop_rate.sleep()** :
Permet de définir une fréquence d'exécution. La méthode 'sleep' prend en compte le temps d'exécution des instructions dans le 'while' afin de garantir la même fréquence.

2- Modifier le fichier 'CMakelists.txt' en ajoutant ceci après 'include_directories' :

```
## Declare a cpp executable
add_executable(my_node_exe src/my_node.cpp)

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

la macro **cmake add_executable** permet de spécifier les fichiers sources C++ nécessaires à la construction de l'exécutable (ici 'my_node_exe').
La macro **target_link_libraries** permet de définir les librairies qui seront liées à l'exécutable 'my_node_exe' (ici la variable **catkin_LIBRARIES** contient la liste de ces librairies).

(http://wiki.ros.org/catkin/CMakeLists.txt)

3- Compiler le workspace avec la commande 'catkin_make'.

4- Lancer 'roscore' et lancer le 'node' avec la commande 'rosrun' (fait partie de 'rosbash') dans 2 fenêtres terminal séparées.

La commande : 'rosrun nom_du_paquet nom_du_node' permet de lancer le 'node'.

```
rosrun my_first_node my_node_exe
```

'roscore' (http://wiki.ros.org/roscore) lance une collection de 'nodes' et de programmes nécessaires au bon fonctionnement de ROS, il doit toujours être lancé une seule fois.
Il permet notamment la communication entre 'nodes', mais aussi la sauvegarde de paramètres partagés entre 'nodes'.
Pour permettre cela, 'roscore' lance 'ros Master' (http://wiki.ros.org/Master) et 'ros parameter server' (http://wiki.ros.org/Parameter%20Server)

On le lance comme ceci :

```
roscore
```

Remarques :
- le 'node' et 'roscore' peuvent-être arrêtés avec un Ctrl+C.
- l'exécutable 'my_node_exe' se trouve dans le dossier 'my_catkin_ws/devel/lib/my_first_node'
- Ne pas confondre le nom de l'exécutable 'my_node_exe' que l'on peut exécuter via la commande 'rosrun' et le nom du 'node' donné lors de son initialisation 'my_name_node'.

5- Les paramètres serveur ROS avec 'rosparam' (http://wiki.ros.org/rosparam).

La commande 'rosparam' permet de définir des paramètres/variables au niveau du serveur ROS, qui pourront être partagés par tous les 'nodes'. Ceci permet par exemple de spécifier un contexte global avant de lancer des 'nodes'. Un 'node' peut accéder à ces paramètres, lire leur contenu, les supprimer.

Définir un paramètre 'my_param' avec comme contenu le chiffre 123.45 dans un autre terminal :

```
rosparam set my_param 123.45
rosparam list
```

Attention : 'roscore' doit être lancé !

Modifier le code C++ du node 'my_node.cpp', en ajoutant dans la boucle 'while' ceci :

```
if (node.hasParam("my_param"))
{
	float my_value_param;
	bool return_get = node.getParam("my_param", my_value_param);
	ROS_INFO_STREAM("We've get param value " << my_value_param);
}
```

Ici on vérifie si le param est bien définit en se servant du 'NodeHandle' nommé 'node', on récupère sa valeur et on l'affiche.

Compiler le 'workspace', lancer le 'node' et essayer de supprimer le 'param' ou de le recréer quand le 'node' est lancé.

```
rosparam delete my_param
```

6- Utiliser un fichier au format YAML (YAML Ain't Markup Language **http://www.yaml.org/)** pour définir plusieurs paramètres d'un seul coup.

Dans le paquet 'my_first_node' créer un nouveau dossier nommé 'yaml'. Créer un nouveau fichier 'my_params.yaml' dont le contenu est ceci :

```
my_string: 'Hello'
my_integer: 1234
my_float: 1234.5
my_boolean: true
my_list: ["1.0", "mixed list"]
my_dictionary: {a: b, c: d}
```

La syntaxe est très simple :
nom_paramètre : valeur

Lancer la commande 'rosparam' pour charger ce fichier yaml :

```
rosparam load my_params.yaml
```

Observer le résultat avec un 'rosparam list' et essayer des 'rosparam get nom_paramètre'.

Modifier le code C++ du node 'my_node.cpp', en ajoutant dans la boucle 'while' ceci :

```
if (node.hasParam("my_list"))
{
	std::vector<std::string> my_list_param;
	bool return_get = node.getParam("my_list", my_list_param);
	if (return_get)
	{
		ROS_INFO_STREAM("We've get 'list' param value ");
		for (std::vector<std::string>::iterator it = my_list_param.begin() ; it != my_list_param.end(); ++it)
			ROS_INFO("--> %s",(*it).c_str());
	}
	else
		ROS_INFO_STREAM("We've NOT get list param value ");

}

if (node.hasParam("my_dictionary"))
{
	std::map<std::string,std::string> my_dic;
	bool return_get = node.getParam("my_dictionary", my_dic);
	if (return_get)
	{
		ROS_INFO_STREAM("We've get 'dic' param value ");
		for (std::map<std::string,std::string>::iterator it=my_dic.begin(); it!=my_dic.end(); ++it)
			ROS_INFO("--> %s : %s", (it->first).c_str(), (it->second).c_str());


	}
	else
		ROS_INFO_STREAM("We've NOT get dic param value ");

}
```

Compiler le 'workspace', lancer le 'node' et essayer de supprimer des paramètres ou de les recréer quand le 'node' est lancé.

7- Notion de 'namespace' dans les 'params' et les 'nodes'.

Lorsque l'on a plusieurs paramètres ou 'nodes' de même nom, on peut les associer à des 'namespace' différents.
Pour cela, il faut mettre au début du nom de 'param' ou du 'node', le nom du 'namespace'. Par défaut le 'namespace' est '/'.

Essayer ceci :

```
rosparam set /left/my_param 12
rosparam set /right/my_param 13
rosparam list
rosparam get /left
rosparam get /right/my_param
```

Lancer le 'node' dans un autre 'namespace' définit avec la variable spéciale '__ns' :

```
rosrun my_first_node my_node_exe __ns:=left
```

ou bien

```
rosrun my_first_node my_node_exe __ns:=right
```

Pour obtenir les 'node' lancés en cours et plus d'infos sur un 'node', lancer ceci dans une autre fenêtre terminal :

```
rosnode list
rosnode info my_name_node
```

rosnode info /left/my_name_node **ou bien** rosnode info /right/my_name_node

'rosnode list' donnera les noms des nodes (avec leurs éventuels namespaces), ici dans notre cas il s'agit du nom que l'on a spécifié dans 'ros::init' → 'my_name_node'.

8- Les 'params' relatifs (private) au 'node' courant :

On peut définir des 'params' privés au 'node' courant, ils seront donc relatifs au namespace du 'node' courant et commenceront (par exemple) par '/my_name_node/…' si le namespace du node courant est '/'.

Modifier le fichier 'my_node.cpp' an ajoutant un '**NodeHandle private**' symbolisé par le nom '~' et la création de 'params' comme ceci :

```
ros::NodeHandle node_private("~");
node_private.setParam("/global_param", 7);
node_private.setParam("relative_param", "my_string");
node_private.setParam("bool_param", false);
```

Compiler, Exécuter le 'node' et afficher la liste des 'params'. Observer que 'global_param' reste global avec le 'namespace' par défaut '/'.

Lancer le 'node' et observer la liste des 'params'.

Lancer le 'node' avec un autre 'namespace', observer la liste des 'params'.

```
rosrun my_first_node my_node_exe __ns:=toto
rosparam list
```

9- Utilisation de 'roslaunch' (http://wiki.ros.org/roslaunch).

On souhaite pouvoir positionner des paramètres, de lancer un ou des 'node', de définir des 'namespace' et ceci avec une seule commande.

Pour réaliser cela on devra utiliser la commande 'roslaunch', mais aussi définir la liste des commandes à exécuter dans un fichier 'launch' (écrit dans un format type 'xml').

Il est à noter que 'roslaunch' lance automatiquement 'roscore'.

Créer dans le paquet 'my_first_node' un dossier nommé 'launch' qui contiendra un fichier 'my_first_node.launch' ressemblant à ceci :

```
<launch>
  <group ns="first">
        <node pkg="my_first_node" type="my_node_exe" name="my_other_name_node"
output="screen">
          <param name="testparameter" value="a" />
          <param name="show_windows" value="true" />
          <rosparam file="$(find my_first_node)/yaml/my_params.yaml" command="load"/>
        </node>
         <rosparam file="$(find my_first_node)/yaml/my_params.yaml" command="load"/>
  </group>
</launch>
```

On commence par définir un 'namespace' global nommé 'first'. C'est à dire que le 'node' sera lancé dans ce 'namespace'. La commande 'rosnode list' nous donnera '/first/my_other_name_node'. (ici le 'node' est renommé 'my_other_name_node')

Puis, on décrit le 'node' à lancer entre les balises '<node …>' et '</node>' :
→ 'pkg' correspond au nom du paquet, ici 'my_first_node'
→ 'type' correspond au nom de l'exécutable du 'node', ici 'my_node_exe'
→ 'name' correspond au nom du 'node', ici renommé en 'my_other_name_node'
→ 'output' permet de définir les affichages du 'node', ici une sortie écran.

Les 'params' : 'testparameter', 'show_windows' ou le chargement du fichier 'yaml' se feront dans le 'namespace' du 'node', c'est à dire '/first/my_other_name_node'. Pour trouver le chemin du fichier 'my_params.yaml' on se sert de la commande 'find' suivie du nom du paquet.

Le chargement du fichier 'yaml' définit en dehors du 'node' se fera dans le namespace global 'first'.

Lancer le fichier launch :

```
roslaunch my_first_node my_first_node.launch
```

Observer la liste des params et leurs 'namespace', les infos sur le 'node' :

```
rosparam list
rosnode list
rosnode info /first/my_other_name_node
```

Modifier le fichier launch pour lancer le même 'node' 'my_node_exe' dans 2 'namespace' différents.

10- Création d'un 'node' qui publie des messages dans un 'topic' ROS (http://wiki.ros.org/msg).

Créer un nouveau paquet nommé 'my_first_topic' dans le 'workspace'.

```
catkin_create_pkg my_first_topic roscpp
```

L'objectif est de créer un 'node' qui publie des messages via un 'topic'. Le message contiendra la position 2D d'un robot, c'est à dire un angle et une coordonnée (x,y).

La première chose à faire est de définir le contenu du message. Pour cela, on va créer un fichier 'position.msg' dans un nouveau dossier 'msg' du paquet 'my_first_topic'.

Le contenu du message sera :

```
# A 2D Position: X,Y, and angle
Header header
float64 x
float64 y
float64 angle
```

Les types de données des messages peuvent être :
→ Boolean: bool
→ Integer: int8,int16,int32,int64
→ Unsigned Integer: uint8,uint16,uint32,uint64
→ Floating Point: float32, float64
→ String: string
→ Fixed length arrays: bool[16]
→ Variable length arrays: int32[]
→ Other: for more complex data structure

Dans notre cas, on utilise le type primitif 'float64' et un type complexe 'Header' qui fait partie du paquet ROS 'std_msgs'. (http://wiki.ros.org/std_msgs)
Le type 'Header' contient notamment un champ nommé 'stamp' qui permettra de positionner la date (plus exactement le temps) ou a été créé le message.

Pour utiliser ce message dans ROS, il faut demander à ROS de générer quelques fichiers.

Modifier le fichier 'package.xml' en dé-commentant ces deux lignes :

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Modifier le fichier 'CMakeLists.txt' en ajoutant/dé-commentant ces lignes :

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  message_generation
  std_msgs
)
## Generate messages in the 'msg' folder
add_message_files(
   FILES
   position.msg
)

## Generate added messages and services with any dependencies listed here
generate_messages(
   DEPENDENCIES
   std_msgs  # Or other packages containing msgs
)

catkin_package(
 # INCLUDE_DIRS include
#  LIBRARIES my_first_topic
  CATKIN_DEPENDS
    roscpp
    message_runtime
#  DEPENDS system_lib
)
```

Lancer la compilation du paquet.

```
catkin_make my_first_topic
ou bien
catkin_make
```

Remarque : L'outil 'gencpp' qui fait partie du paquet 'message_generation' a créé le fichier c++ 'my_catkin_ws/devel/include/my_first_topic/position.h'. C'est ce fichier qu'il faudra inclure dans le code du 'node' qui publiera ce type de message.
Une version python se trouve dans : 'my_catkin_ws/devel/lib/python2.7/dist-packages/my_first_topic/msg/_position.py'
Une version lisp se trouve dans : 'my_catkin_ws/devel/share/common-lisp/ros/my_first_topic/msg/position.lisp'

Utiliser la commande 'rosmsg' (http://wiki.ros.org/rosmsg) pour obtenir le contenu du message 'position' :

```
rosmsg show my_first_topic/position
```

Utiliser la commande 'rosmsg' pour obtenir la liste de tous les messages d'un paquet :

```
rosmsg package my_first_topic
```

Ajouter le fichier c++ du 'publisher' nommé 'simple_publisher.cpp' dans le dossier 'src' contenant :

```
/**
 ** Simple ROS Publisher Node
 **/
#include <ros/ros.h>
#include <my_first_topic/position.h>


int main(int argc, char* argv[])
{
   ros::init(argc, argv, "simple_publisher");
   ros::NodeHandle node;

   // Create a 1Hz update rate
   ros::Rate loop_rate(1.0);

   // Advertise that we're publishing the topic "my_position", of type 'position'
   ros::Publisher pub = node.advertise<my_first_topic::position>("/my_position", 1000);

   // The angle counter
   int angle = 0;

   // The angle increment
   int angle_increment = 10;

   ROS_INFO("Starting publisher");

   while(ros::ok()) {
      // Create a message, and set the fields appropriately
      my_first_topic::position msg;
      msg.header.stamp = ros::Time::now();
      msg.angle = angle * 3.141592 / 180.0;
      msg.x = 100.0 * cos(msg.angle);
      msg.y = 100.0 * sin(msg.angle);

      // Update angle
      angle = (angle + angle_increment) % 360;

      // Publish the message, give ROS an opportunity to run
      pub.publish(msg);

      // Handles the callback events and returns immediately
      ros::spinOnce();

      ROS_INFO("Published message %.1f, %.1f, %.1f", msg.x, msg.y, msg.angle * 180.0 / 3.141592);

      // Wait 1 second to publish again
      loop_rate.sleep();
   }

   ROS_INFO("Publisher done.");
   return 0;
}
```

Ici on prévient ROS que l'on veut publier sur un topic nommé 'my_position' avec des messages de type 'my_first_topic::position'. 1000 étant la taille de son buffer.

```
ros::Publisher pub = node.advertise<my_first_topic::position>("/my_position", 1000);
```

Ici on publie le message.

```
pub.publish(msg);
```

Ici on permet à ROS de traiter d'autres appels en attente, par exemple dans le cas ou 'simple_publisher' reçoit des messages en attente dans un buffer, provenant d'autres 'nodes'.

```
ros::spinOnce();
```

Modifier le 'CMakeLists.txt' pour ajouter la compilation de ce fichier :

```
## Declare a C++ executable
add_executable(simple_publisher src/simple_publisher.cpp)

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

Compiler et lancer le 'node' (attention il faut penser à lancer 'roscore') :

```
catkin_make
rosrun my_first_topic simple_publisher
```

Remarques :
- L'exécutable du 'node simple_publisher' se trouve dans le dossier 'devel/lib/my_first_topic/'.
- L'instruction 'add_dependencies' permet d'ajouter une dépendance à la génération de messages pour la compilation de ce 'node'. Ce qui force à générer le message avant de lancer la compilation du 'node'.
- La variable cmake ${PROJECT_NAME} contient le nom du projet définit par :
project(my_first_topic)

Dans un autre terminal afficher les infos et le contenu du topic '/my_position' avec la commande 'rostopic' (http://wiki.ros.org/rostopic) :

```
rostopic list
rostopic type /my_position
rostopic info /my_position
rostopic echo /my_position
```

11- Création d'un 'node' qui s'abonne à un 'topic' pour recevoir des messages ROS.

L'objectif étant de construire un 'node' qui s'abonne au topic '/my_position' et qui affiche le contenu des messages reçus.

Ajouter le fichier c++ nommé 'simple_subscriber.cpp' contenant :

```
/**
 ** Simple ROS Subscriber Node
 **/
#include <ros/ros.h>
#include <my_first_topic/position.h>

void positionCallback(const my_first_topic::position& msg)
{
   ROS_INFO("New position: %.1f,%.1f,%.1f", msg.x, msg.y, msg.angle * 180.0 / 3.141592);
}

int main(int argc, char* argv[])
{
   ros::init(argc, argv, "simple_subscriber");
   ros::NodeHandle node;

   // Subscribe to the topic '/my_position' with a buffer length of 1000
   // and a callback function 'positionCallback'.
   ros::Subscriber subscriber = node.subscribe("/my_position", 1000, positionCallback);

   // Block and process ROS messages and callback until the node has been shutdown
   ros::spin();

   return 0;
}
```

Ici on s'abonne au topic '/my_position' avec un buffer de 1000 et surtout à chaque nouveau message reçu, on appelle la fonction 'positionCallback'.

```
ros::Subscriber subscriber = node.subscribe("/my_position", 1000, positionCallback);
```

Ici le 'node' est bloqué (c'est à dire ne sort pas), ce qui lui permet de traiter tous les messages reçus.

```
ros::spin();
```

Modifier le fichier 'CMakelists.txt' pour prendre en compte le 'node' 'simple_subscriber'.
Compiler et exécuter 'simple_publisher' et 'simple_subscriber' dans des terminaux différents.
Essayer aussi d'afficher le contenu du topic '/my_position'.

Lancer la commande 'rqt_graph' dans un autre terminal pendant que les 2 'nodes' s'exécutent.

```
rosrun rqt_graph rqt_graph
```

Cet outil graphique dessine dynamiquement les liaisons entre les 'node' et les 'topics' associés.

11 bis- Utilisation de 'rosbag' (http://wiki.ros.org/rosbag)

L'outil 'rosbag' permet d'enregistrer le contenu des messages publiés sur un 'topic' par un 'node', afin de pouvoir les rejouer ultérieurement (très utile pour des tests).

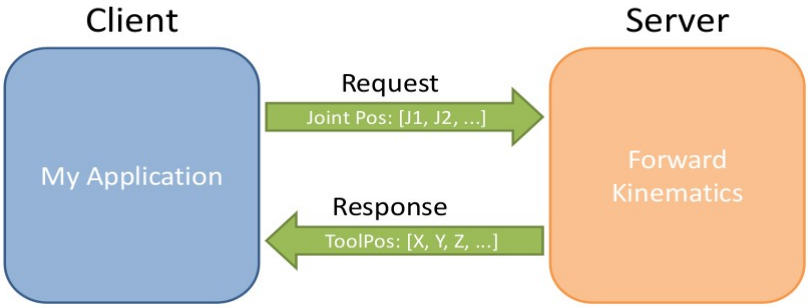
Lancer le 'node' 'simple_publisher', puis dans une autre fenêtre terminal lancer ceci :
```
rosbag record /my_position -o ~/my_bag_for_my_position_topic
```

Vous pouvez arrêter l'enregistrement du 'rosbag' et le 'simple_publisher' par un 'ctrl+c'.
L'option '-o' indique le dossier et nom du fichier 'bag' (qui aura l'extension '.bag').

Arrêter le 'simple_publisher', lancer le 'simple_subscriber', lancer la lecture du 'bag' avec ceci :
```
rosbag play ~/my_bag_for_my_position_topic_2016-01-15-13-12-04.bag
```

L'outil 'rqt_bag' (http://wiki.ros.org/rqt_bag) permet de faire la même chose en mode graphique.
```
rqt_bag
```

12- Création d'un 'service' ROS (http://wiki.ros.org/rosservice).



Un service ROS peut être vu comme l'appel d'une fonction d'un serveur par un client.
L'appel de ce service bloque le client jusqu'à la réponse du serveur.
Un service est donc fait d'une requête (request) envoyée par le client et reçue par le serveur et d'une réponse (response) générée par le serveur en envoyée vers le client.

L'objectif est de créer un service permettant de faire la somme de deux entiers et de retourner le résultat.

Créer un nouveau paquet nommé 'my_first_service'.
Créer un fichier nommé 'addTwoInts.srv' dans un nouveau dossier 'srv' contenant :
```
int64 a
int64 b
---
int64 sum
```

Ce fichier définit les paramètres de requête et de réponse du service. Ici deux entiers 'int64' en requête et un 'int64' en réponse. Observer les trois tirets séparant le tout.

Comme pour les messages, ajouter ceci dans le fichier 'package.xml' :
```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Modifier le 'CMakelists.txt' pour prendre en compte ce service :

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  message_generation
)

## Generate services in the 'srv' folder
add_service_files(FILES addTwoInts.srv)

## Generate added messages and services with any dependencies listed here
generate_messages(DEPENDENCIES std_msgs)

catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES my_first_service
  CATKIN_DEPENDS
            roscpp
            message_runtime
#  DEPENDS system_lib
)
```

Lancer la compilation du paquet.
Observer la création de plusieurs '.h' dans le dossier 'devel/include/my_first_service/', dont le fichier 'addTwoInts.h' qu'il faudra inclure dans les 'nodes'.

Créer deux fichiers C++ nommés 'add_two_ints_client.cpp' et 'add_to_ints_server.cpp'.
Il s'agira d'un 'node client' qui demandera la somme de deux entiers et attendra la réponse, mais aussi d'un 'node serveur' qui fera la somme en retournera le résultat.

Code de 'add_two_ints_client.cpp' :

```
#include "ros/ros.h"
#include "my_first_service/addTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<my_first_service::addTwoInts>("add_two_ints");
  my_first_service::addTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

Ici on créé le service client 'add_two_ints' définit par la structure 'my_first_service::addTwoInts' :

```
ros::ServiceClient client = n.serviceClient<my_first_service::addTwoInts>("add_two_ints");
```

Ici on envoie une requête client :

```
client.call(srv)
```

Code de 'add_two_ints_server.cpp' :

```
#include "ros/ros.h"
#include "my_first_service/addTwoInts.h"

bool add(my_first_service::addTwoInts::Request  &req,
         my_first_service::addTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}

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

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

Ici on créé le service serveur 'add_two_ints' associé à une fonction 'add' et on prévient ROS :

```
ros::ServiceServer service = n.advertiseService("add_two_ints", add);
```

Modifier le fichier 'CMakeLists.txt' :

```
## Declare a C++ executable
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server ${PROJECT_NAME}_generate_messages_cpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client ${PROJECT_NAME}_generate_messages_cpp)
```

Lancer les deux 'node' dans des terminaux différents :

```
rosrun my_first_service add_two_ints_server
rosrun my_first_service add_two_ints_client 18 98
```

Observer la liste des services et leurs informations :

```
rosservice list
rosservice info /add_two_ints
rosservice type '/add_two_ints'
rosservice args '/add_two_ints'
```

Appeler le service '/add_two_ints' directement avec 'rosservice' :

```
rosservice call '/add_two_ints' 45 67
```

13- On souhaite pouvoir changer la valeur d'incrément d'angle ('angle_increment') dans le node 'publisher' du paquet 'my_first_topic' et ceci à volonté.
C'est à dire qu'une fois le 'node' lancé, on doit pouvoir modifier cette valeur d'incrément.