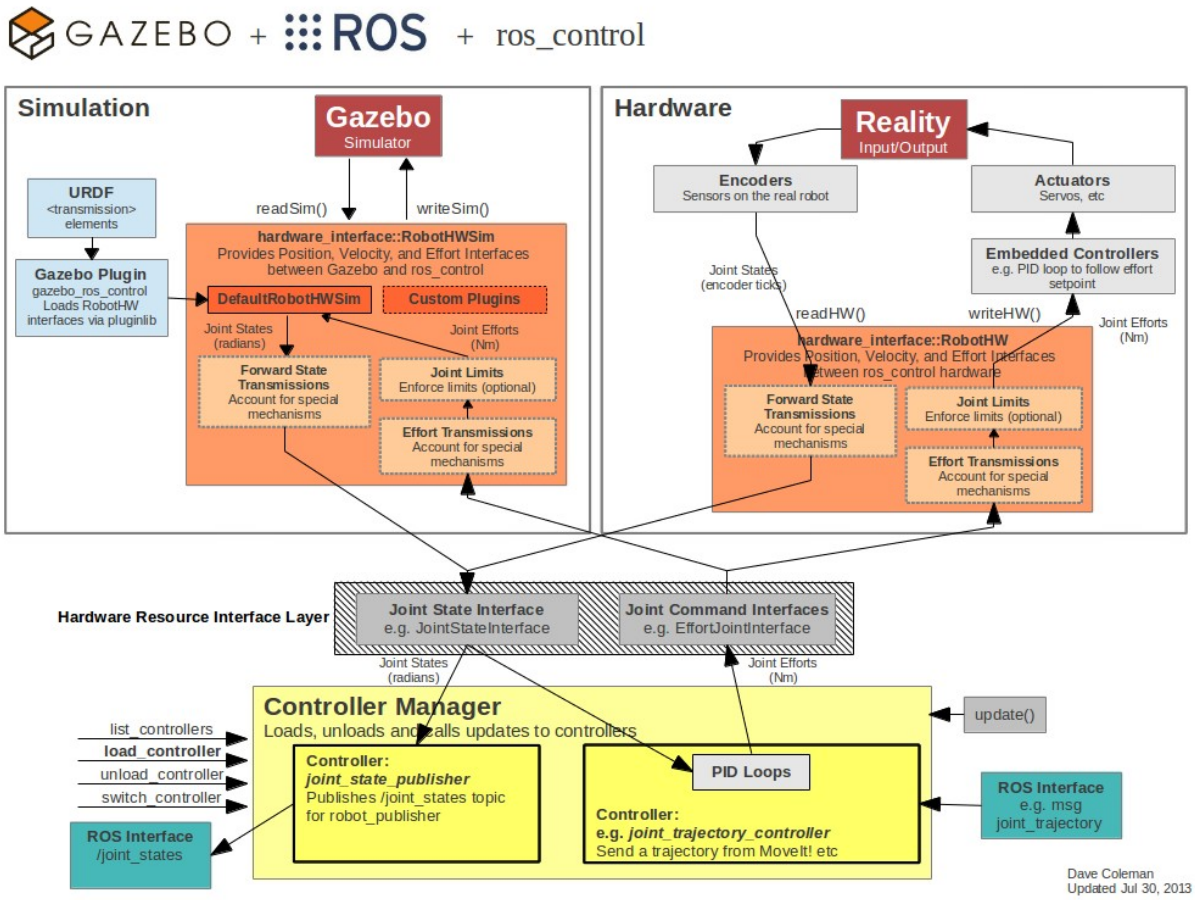
Laurent LEQUIEVRE – Institut Pascal UMR6602

Notions de contrôleur ROS : 'ros_control', 'Controller Manager' relié à 'gazebo'

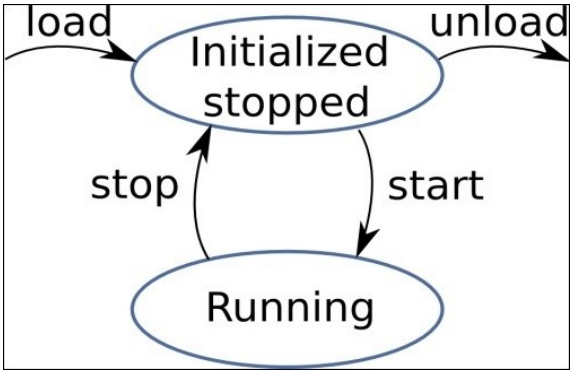


Dans cette partie, on va développer de 'A' à 'Z' un contrôleur ROS pour un bras 'kuka lwr'. Ce contrôleur devra être commandé via un 'topic' contenant un groupe de valeurs articulaires à atteindre (une valeur articulaire pour chaque 'joint' du bras).

Par exemple comme ceci :
rostopic pub -1 /kuka_lwr/my_kuka_group_position_controller/command std_msgs/Float64MultiArray "data: [0.0,0.0,0.0,0.0,0.0,0.0,0.0]"

Dans cette commande, le nom du contrôleur est '**my_kuka_group_position_controller**' et se trouve dans le 'namespace' '**kuka_lwr**'. On lui passe en paramètre un message de type '**std_msgs/Float64MultiArray**' qui peut contenir un tableau de 'float'.

Ce contrôleur devra être chargé et initialisé par le '**controller manager**' de ROS. Après son initialisation, on peut le lancer et l'arrêter à volonté. Un contrôleur arrêté reste toujours chargé. Une fois le contrôleur 'déchargé', il se ne sera plus utilisable.



Pour développer ce contrôleur nous allons devoir passer par plusieurs étapes :
- Créer un nouveau paquet ROS
- Rédiger le code C++ du contrôleur (une classe C++)
- Déclarer cette classe comme un 'plugin' (qui sera chargé par le 'controller manager')
- Modifier les fichiers 'package.xml' et 'CMakeLists.txt' du paquet
- Compiler le tout
- Modifier le fichier 'launch' pour prendre en compte le chargement de ce nouveau contrôleur.
- Tester le tout avec 'gazebo'

1- Créer un nouveau paquet nommé '**kuka_lwr_controllers**' avec une dépendance à '**roscpp**' dans votre 'workspace'. Ajouter deux sous dossiers '**include**' et '**src**'.

2- Créer un fichier vide (pour l'instant) nommé '**group_command_controller.cpp**' dans le dossier 'src'. Mais aussi un autre fichier nommé '**group_command_controller.h**' dans le dossier 'include'. Ces fichiers contiendront le code C++ du contrôleur.

2bis- Définir les bonnes interfaces des transmissions.

Le contrôleur va commander le bras kuka en position angulaire, il faut donc utiliser l'interface '**PositionJointInterface**' pour chaque 'joint'.
Modifier en conséquence le fichier '**kuka_lwr_transmission.xacro**' qui se trouve dans le paquet '**kuka_lwr_description**'.

3- Modifier le fichier 'group_command_controller.h' comme ceci :

```
#ifndef GROUP_COMMAND_CONTROLLER_BASE_H
#define GROUP_COMMAND_CONTROLLER_BASE_H

// ros controller interface
#include <controller_interface/controller.h>

// ros hardware interface (contains definition of PositionJointInterface)
#include <hardware_interface/joint_command_interface.h>

// ros tools
#include <ros/node_handle.h>
#include <ros/ros.h>
#include <std_msgs/Float64MultiArray.h>
#include <control_toolbox/pid.h>
#include <urdf/model.h>

// boost smart pointer
#include <boost/scoped_ptr.hpp>
```

```
namespace kuka_lwr_controllers
{
	class GroupCommandController:
	public controller_interface::Controller<hardware_interface::PositionJointInterface>
	{
		public:
			GroupCommandController();
			~GroupCommandController();

			bool init(hardware_interface::PositionJointInterface *robot, ros::NodeHandle &n);
			void starting(const ros::Time& time);
			void stopping(const ros::Time& time);
			void update(const ros::Time& time, const ros::Duration& period);

	};
}

#endif
```

On définit une classe nommée **'GroupCommandController'** dans le 'namespace' **'kuka_lwr_controllers'**. Cette classe hérite de '**controller_interface::Controller**' paramétrée par '**hardware_interface::PositionJointInterface**'.

Tout nouveau contrôleur doit respecter un cadre définit par la classe 'Controller' qui hérite de 'ControllerBase' :
(http://docs.ros.org/indigo/api/controller_interface/html/c++/classcontroller__interface_1_1Controller.html).

Le cadre à respecter est l'ajout de 4 méthodes ('init','starting','stopping', 'update').
La méthode **'init'** est lancée lors du chargement du contrôleur par le 'controller manager'.
La méthode **'starting'** est lancée au démarrage du contrôleur.
La méthode **'stopping'** pour son arrêt.
La méthode **'update'** est appelée régulièrement par le 'controller manager' afin de calculer la commande (à une fréquence de 1000 Hz).

L'interface '**hardware_interface::PositionJointInterface'** utilisée ici correspond à celle définit dans le fichier URDF des 'transmissions'.

4- Modifier le fichier 'group_command_controller.cpp' comme ceci :

```
#include <group_command_controller.h>

// For plugin export
#include <pluginlib/class_list_macros.h>

// For angles calculation
#include <angles/angles.h>

namespace kuka_lwr_controllers
{
	GroupCommandController::GroupCommandController() {}
	GroupCommandController::~GroupCommandController() {}

	bool GroupCommandController::init(hardware_interface::PositionJointInterface *robot_hw, ros::NodeHandle &n)
	{

		return true;
	}

	void GroupCommandController::starting(const ros::Time& time)
	{
```

```
	}

	void GroupCommandController::stopping(const ros::Time& time)
	{

	}

	void GroupCommandController::update(const ros::Time& time, const ros::Duration& period)
	{

	}
}

PLUGINLIB_EXPORT_CLASS(kuka_lwr_controllers::GroupCommandController, controller_interface::ControllerBase)
```

On retrouve dans ce fichier le corps vide des 4 méthodes définies dans le fichier '.h'.

On a aussi ajouté l'appel d'une macro 'PLUGINLIB_EXPORT_CLASS' (http://wiki.ros.org/pluginlib), qui indique que cette classe pourra être chargée dynamiquement comme un 'plugin'.
Dans notre cas on indique que la classe '**kuka_lwr_controllers::GroupCommandController**' est un '**controller_interface::ControllerBase**'.

Pour info : la classe '**controller_interface::Controller<T>**' hérite de '**controller_interface::ControllerBase**'.

5- Modifier le fichier 'yaml' des paramètres des contrôleurs.

Ajouter ceci à la fin du fichier 'kuka_lwr_control.yaml' (il se trouve pour rappel dans le dossier 'config' du paquet 'kuka_lwr_description') :

```
# Joint Group Position Controller ----------------------------
 my_kuka_group_position_controller:
   type: kuka_lwr_controllers/GroupCommandController
   joints:
      - kuka_lwr_0_joint
      - kuka_lwr_1_joint
      - kuka_lwr_2_joint
      - kuka_lwr_3_joint
      - kuka_lwr_4_joint
      - kuka_lwr_5_joint
      - kuka_lwr_6_joint
```

Ici on ajoute les paramètres d'un nouveau contrôleur nommé '**my_kuka_group_position_controller**'.
Ce contrôleur est de type 'kuka_lwr_controllers/GroupCommandController ', ce qui correspond au 'namespace' 'kuka_lwr_controllers' de la classe 'GroupCommandController'.
Le paramètre 'joints' définit les noms des articulations concernées par ce contrôleur.

6- Ajouter des attributs 'private' dans le fichier 'group_command_controller.h' comme ceci :

```
private:
	ros::NodeHandle nh_;
	std::vector< std::string > joint_names_;
	std::vector< hardware_interface::JointHandle > joints_;
	std::vector<double> commands_buffer_;
	unsigned int n_joints_;
	ros::Subscriber sub_command_;
```

```
	std::vector<boost::shared_ptr<const urdf::Joint> > joints_urdf_;
```

Le vecteur 'joint_names_' contiendra les noms des 'joints'.
Le vecteur 'joints_' contiendra les interfaces d'accès aux 'joints'.
Le vecteur 'commands_buffer_' contiendra les commandes à envoyer aux 'joints'.
Le vecteur 'joints_urdf_' contiendra le contenu 'urdf' pour chaque 'joint'.

7- Modifier la méthode 'init' dans le fichier 'group_command_controller.cpp' comme ceci :

```
bool GroupCommandController::init(hardware_interface::PositionJointInterface *robot_hw, ros::NodeHandle &n)
{
	//ROS_INFO("***** START GroupCommandController::init ************");

	// List of controlled joints
	std::string param_name = "joints";
	if(!n.getParam(param_name, joint_names_))
	{
		ROS_ERROR_STREAM("Failed to getParam '" << param_name << "' (namespace: " << n.getNamespace() <<
").");
		return false;
	}

	n_joints_ = joint_names_.size();
	//ROS_INFO("***** GroupCommandController:: nb joints = %d ************", n_joints_);

	if (n_joints_ == 0) {
		ROS_ERROR_STREAM("List of joint names is empty.");
		return false;
	}

	commands_buffer_.resize(n_joints_);
	joints_.resize(n_joints_);

	for (unsigned int i=0; i<n_joints_; i++)
	{
		try
		{
			//ROS_INFO("***** GroupCommandController:: get joint name = %s ************",
joint_names_[i].c_str());
			joints_[i]= robot_hw->getHandle(joint_names_[i]);
			commands_buffer_[i] = 0.0;
		}
		catch (const hardware_interface::HardwareInterfaceException& e)
		{
			ROS_ERROR_STREAM("Exception thrown: " << e.what());
			return false;
		}
	}


	urdf::Model urdf;
	if (!urdf.initParam("robot_description"))
	{
		ROS_ERROR("Failed to parse urdf file");
		return false;
	}

	joints_urdf_.resize(n_joints_);

	for (unsigned int i=0; i<n_joints_; i++)
	{
		joints_urdf_[i] = urdf.getJoint(joint_names_[i]);
		if (!joints_urdf_[i])
```

```
                {
                        ROS_ERROR("Could not find joint '%s' in urdf", joint_names_[i].c_str());
                        return false;
                }
        }

        sub_command_ = n.subscribe<std_msgs::Float64MultiArray>("command", 1, &GroupCommandController::commandCB,
this);

        //ROS_INFO("***** FINISH GroupCommandController::init ************");

        return true;
}
```

Cette méthode commence par récupérer le nom des articulations via le paramètre 'joints' définit dans le fichier 'kuka_lwr_control.yaml' et le stocke dans le vecteur 'joint_names_' :

```
// List of controlled joints
std::string param_name = "joints";
if(!n.getParam(param_name, joint_names_))
{
        ROS_ERROR_STREAM("Failed to getParam '" << param_name << "' (namespace: " << n.getNamespace() << ").");
        return false;
}
```

Puis pour chaque articulation, récupère son interface et le stocke dans le vecteur 'joints_' et initialise sa commande à '0.0' dans le vecteur 'commands_buffer_' :

```
for (unsigned int i=0; i<n_joints_; i++)
{
        try
        {
                joints_[i]= robot_hw->getHandle(joint_names_[i]);
                commands_buffer_[i] = 0.0;
        }
        catch (const hardware_interface::HardwareInterfaceException& e)
        {
                ROS_ERROR_STREAM("Exception thrown: " << e.what());
                return false;
        }
}
```

Puis toujours pour chaque articulation récupère sa description 'URDF' via le paramètre général 'robot_description' :

```
urdf::Model urdf;
if (!urdf.initParam("robot_description"))
{
        ROS_ERROR("Failed to parse urdf file");
        return false;
}

joints_urdf_.resize(n_joints_);

for (unsigned int i=0; i<n_joints_; i++)
{
        joints_urdf_[i] = urdf.getJoint(joint_names_[i]);
        if (!joints_urdf_[i])
        {
                ROS_ERROR("Could not find joint '%s' in urdf", joint_names_[i].c_str());
                return false;
        }
}
```

Le contenu 'URDF' de chaque articulation sera utile, par la suite, pour récupérer leurs valeurs limites 'upper' et 'lower'.

Enfin, on souscrit à un 'topic' nommé 'command'. La méthode 'commandCB' de la classe étant chargée de traiter le message reçu (pour rappel, un tableau de valeurs articulaires) :

```
sub_command_ = n.subscribe<std_msgs::Float64MultiArray>("command", 1, &GroupCommandController::commandCB, this);
```

8- Ajouter la méthode '**private**' '**commandCB**' dans la classe :

Dans le fichier '.h' :

```
void commandCB(const std_msgs::Float64MultiArrayConstPtr& msg);
```

Dans le fichier '.cpp' :

```
void GroupCommandController::commandCB(const std_msgs::Float64MultiArrayConstPtr& msg)
{
  if (msg->data.size()!=n_joints_)
  {
    ROS_ERROR_STREAM("Dimension of command (" << msg->data.size() << ") does not match number of joints (" << n_joints_
<< ")! Not executing!");
    return;
  }
 commands_buffer_ = msg->data;
}
```

Cette méthode consiste à récupérer le tableau de valeurs articulaires et de l'affecter au vecteur 'commands_buffer_'.
Le message passé au 'topic' est de type 'std_msgs::Float64MultiArray'.

9- Ajouter une méthode 'private' 'enforceJointLimits' afin de contraindre la commande articulaire dans la limite 'upper' et 'lower' du 'joint'.

Dans le fichier '.h' :

```
void enforceJointLimits(boost::shared_ptr<const urdf::Joint> & joint_urdf_,double &command);
```

Dans le fichier '.cpp' :

```
void GroupCommandController::enforceJointLimits(boost::shared_ptr<const urdf::Joint> & joint_urdf_,double &command)
{
	// Check that this joint has applicable limits
	if (joint_urdf_->type == urdf::Joint::REVOLUTE || joint_urdf_->type == urdf::Joint::PRISMATIC)
	{
		if( command > joint_urdf_->limits->upper ) // above upper limnit
		{
			command = joint_urdf_->limits->upper;
		}
		else if( command < joint_urdf_->limits->lower ) // below lower limit
		{
				command = joint_urdf_->limits->lower;
		}
	}
}
```

Si le 'joint' est de type 'revolute' ou 'prismatic' on cherche à contraindre la commande articulaire passée en paramètre.

10- Modifier la méthode 'starting' :

On souhaite au démarrage du contrôleur :
- récupérer la position initiale du 'joint' et la contraindre dans ses limites par sécurité
- initialiser le vecteur de commande avec cette position initiale

```
void GroupCommandController::starting(const ros::Time& time)
{
	double pos_command;

	for (unsigned int i=0; i<n_joints_; i++)
	{
		pos_command=joints_[i].getPosition();
		// Make sure joint is within limits if applicable
		enforceJointLimits(joints_urdf_[i], pos_command);
		commands_buffer_[i] = pos_command;
		//ROS_INFO("***** position -> joints_[%d] = %f ************", i,commands_buffer_[i]);
	}
}
```

11- Modifier la méthode 'update' :

```
void GroupCommandController::update(const ros::Time& time, const ros::Duration& period)
{
	double command_position;

	std::vector<double> & commands = commands_buffer_;
	for(unsigned int i=0; i<n_joints_; i++)
	{
		command_position = commands_buffer_[i];

		// Make sure joint is within limits if applicable
		enforceJointLimits(joints_urdf_[i], command_position);

		joints_[i].setCommand(command_position);
	}
}
```

Pour chaque articulation, on envoie la commande (une valeur angulaire) contrainte dans ses limites par sécurité.
Les valeurs envoyées correspondent à celles récupérées dans le message du 'topic'.

12- Créer le fichier descriptif du 'plugin'.

Ajouter à la racine du paquet, un fichier nommé 'kuka_lwr_controllers_plugins.xml' contenant ceci :

```
<library path="lib/libkuka_lwr_controllers">

<class name="kuka_lwr_controllers/GroupCommandController" type="kuka_lwr_controllers::GroupCommandController"
base_class_type="controller_interface::ControllerBase">
   <description>
    Info...
   </description>
 </class>

</library>
```

Le contrôleur sera physiquement sous forme de librairie dynamique, un fichier nommé 'libkuka_lwr_controllers.so' (ce qui correspond au nom du projet) et se trouvera dans le dossier 'devel/lib' de votre 'workspace'.

Il est sous forme de 'plugin', c'est à dire une sorte de module d'extension qui peut être chargé ou déchargé dynamiquement et qui offre de nouvelles fonctionnalités au système.
La gestion des 'plugin' sous ROS se fait par 'pluginlib' (http://wiki.ros.org/pluginlib)

La classe utilisée pour créer ce 'plugin' est 'GroupCommandController' dans le 'namespace' 'kuka_lwr_controllers'. Cette classe hérite d'une classe de base 'controller_interface::ControllerBase'.

13- Modifier le fichier 'package.xml'

Modifier le fichier 'package.xml' du paquet comme ceci :

```
<?xml version="1.0"?>
<package>
  <name>kuka_lwr_controllers</name>
  <version>0.0.0</version>
  <description>The kuka_lwr_controllers package</description>

  <maintainer email="laurent.lequievre@univ-bpclermont.fr">Laurent LEQUIEVRE</maintainer>

  <license>BSD</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>controller_interface</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>controller_interface</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
    <controller_interface plugin="${prefix}/kuka_lwr_controllers_plugins.xml"/>
  </export>
</package>
```

On a classiquement ajouté les 'build_depend' et 'run_depend' indiquant les paquets nécessaires au 'build' et au 'run' (le paquet 'controller_interface' contient les classes 'Controller' et 'ControllerBase').

Mais surtout, on a 'exporté' notre 'plugin', en indiquant le nom du fichier 'xml' contenant son descriptif.
Dans notre cas, 'controller_interface' correspond au nom du paquet de la classe de base (controller_interface::ControllerBase) utilisée pour créer le 'plugin'.
Le variable ${prefix} contient le chemin on se trouve le fichier 'package.xml'.

13- Modifier le fichier 'CMakeLists.txt' du paquet.

Modifer le contenu du fichier 'CMakeLists.txt' comme ceci :

```
cmake_minimum_required(VERSION 2.8.3)
project(kuka_lwr_controllers)

find_package(catkin REQUIRED COMPONENTS
  controller_interface)

catkin_package(
  CATKIN_DEPENDS
    controller_interface
    std_msgs
  INCLUDE_DIRS include
  LIBRARIES ${PROJECT_NAME}
)

include_directories(include ${Boost_INCLUDE_DIR} ${catkin_INCLUDE_DIRS})

link_directories(${catkin_LIBRARY_DIRS})

add_library(${PROJECT_NAME}
```

```
  src/group_command_controller.cpp
)

target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})
```

Le nom du projet est 'kuka_lwr_controllers' (est contenu dans la variable ${PROJECT_NAME}).
Ce 'cmake' génère une librairie dynamique qui se nommera par défaut 'lib' suivie du nom du projet suivi de '.so' ('libkuka_lwr_controllers.so').
Cette lib est créée à partir du fichier source 'group_command_controller.cpp'.

14- Compiler le 'workspace' avec 'catkin_make'

15- Modifier le fichier '**platform.launch**' qui se trouve dans le paquet '**kuka_lwr_description**'

Il faut le contrôleur '**my_kuka_group_position_controller**' dans un nouveau paramètre 'group_controller' par exemple :

```
<arg name="group_controller" default="my_kuka_group_position_controller" />
```

Et l'utiliser comme paramètre du 'spawner' :

```
<node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="true" output="screen"
args="joint_state_controller $(arg group_controller)"/>
```

16- Tester le tout

Pour lancer la simulation avec le nouveau contrôleur :

```
roslaunch kuka_lwr_description platform.launch
```

Pour envoyer des commandes angulaires via un 'topic' :

```
rostopic pub -1 /ns_kuka_lwr/my_kuka_group_position_controller/command
std_msgs/Float64MultiArray "data: [1.8,1.2,1.9,1.8,0.76,0.9,1.6]"
```

17- Utiliser les 'services' du 'controller_manager'

Les contrôleurs peuvent être chargés, déchargés, démarrés, arrêtés par l'intermédiaire d'appels de 'services' au 'controller_manager'

Consulter la liste des 'services' :

```
rosservice list
```

Consulter la liste des contrôleurs (avec détail sur leur état, interfaces …) :

```
rosservice call /ns_kuka_lwr/controller_manager/list_controllers
```

Arrêter un contrôleur :

```
rosservice call /ns_kuka_lwr/controller_manager/switch_controller "{stop_controllers:
[my_kuka_group_position_controller]}"
```

Vérifier s'il est bien arrêté :

```
rosservice call /ns_kuka_lwr/controller_manager/list_controllers
```

Puis essayer de publier un message sur le 'topic' pour le faire bouger ??

Démarrer un contrôleur :

```
rosservice call /ns_kuka_lwr/controller_manager/switch_controller "{start_controllers:
[my_kuka_group_position_controller]}"
```

Décharger un contrôleur :

```
rosservice call /ns_kuka_lwr/controller_manager/unload_controller "name :
'my_kuka_group_position_controller'"
```

(Attention : Il faut l'arrêter avant)

Charger un contrôleur :

```
rosservice call /ns_kuka_lwr/controller_manager/load_controller "name :
'my_kuka_group_position_controller'"
```