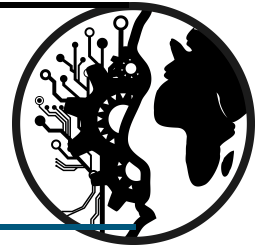


# ROS2: programmation

## Coder les briques élémentaires

Laurent Beaudoin & Loïca Avanthey  
Epita



### Avant propos

Comme on l'a vu dans le TD précédent, un programme ROS doit respecter une architecture particulière pour pouvoir fonctionner (organisation en nœuds, communication via des forums, services, actions, etc.). Dans le TD précédent, nous avons utilisé des programmes écrits par d'autres pour nous familiariser avec cette architecture. Maintenant, nous allons écrire nos propres programmes.

## 1 Fondamentaux de la programmation ROS : *workspace*, paquet & nœud

Il y a un certain nombre de règles à respecter pour l'organisation des codes sources d'un programme ROS. La première d'entre-elles stipule que vos codes doivent appartenir à un espace de travail (*working space*).

### 1.1 L'espace de travail

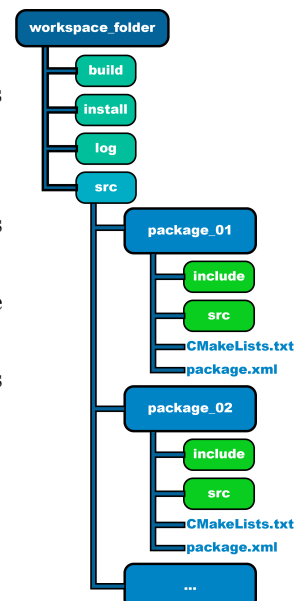
C'est l'endroit où se trouvent tous vos codes personnels relevant d'un même projet global. Vos contributions doivent être regroupées en paquets. Un paquet est généralement thématique c'est-à-dire dédié à un robot ou à un simulateur (comme `turtlesim`), à un capteur particulier ou à une fonctionnalité particulière (perception et reconstruction de l'environnement en SLAM, contrôle / commande spécifique, etc.). Un paquet regroupe tous les programmes nécessaires à sa thématique.

**i** Un paquet peut être totalement autonome ou au contraire s'interfacer avec d'autres paquets pour réaliser des tâches encore plus complexes.

L'espace de travail est organisé en un répertoire racine qui contient tous les paquets sur lesquels on code. Il est organisé avec les répertoires suivant :

- `src` où sont mis tous les code sources organisés en paquets,
- `build` où se trouvent tous les fichiers nécessaires à la compilation de tous les paquets ainsi que les exécutables générés,
- `install` qui contient tous les fichiers de configuration propres à votre espace de travail,
- `log` qui contient tous les fichiers de log générés lors de la compilation des paquets de l'espace de travail.

**✓** En tant que développeur, vous n'aurez à intervenir que dans le répertoire `src`, les autres répertoires seront automatiquement remplis par les outils que l'on utilisera lors des phases de création et de compilation.



## EXERCICE 1 (*Création d'un espace de travail*)



Commençons par créer un répertoire pour l'espace de travail (qu'on nommera `my_ros2_ws`) et celui qui contiendra les sources des paquets (`my_ros2_ws/src`). On a mis par défaut l'espace à la racine du compte utilisateur, à vous d'adapter :

```
mkdir -p ~/my_ros2_ws/src
```

Rendez-vous dans votre nouveau *workspace* :

```
cd ~/my_ros2_ws/
```

On va ensuite lancer la commande `colcon` qui permet de compiler tous les paquets en un seul appel. Dans notre cas, on n'a pas encore de paquet à compiler, mais `colcon` va créer à cette itération les répertoires et fichiers qui lui faudra pour la suite (`build`, `install`, `log`, etc.).

```
colcon build
```



**Colcon est une sorte de super `cmake`**

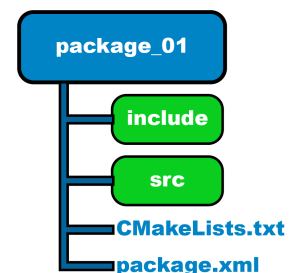


N'oubliez pas que vous pouvez retrouver toutes ces informations sur la documentation en ligne <https://docs.ros.org> (sélectionnez la version de ROS que vous avez installée) !

## 1.2 Les paquets

Un paquet est une collection de répertoires et fichiers contenant les codes sources, les instructions de compilation et de configuration propres au paquet. Tout paquet doit être placé dans le répertoire `src` de l'espace de travail. Un paquet est au minimum composé :

- du fichier `CMakeLists.txt` qui permettra de créer automatiquement tous les fichiers nécessaires à la compilation des codes du paquet,
- du fichier `package.xml`, qu'on appelle "manifeste" et qui précise plusieurs points importants de configuration comme le nom du package (pour ROS), son numéro de version, le contributeur, et surtout ses dépendances,
- des répertoires `src` et `include` qui contiennent le code source du paquet.



## EXERCICE 2 (*Création d'un paquet*)



On va créer le paquet `my_first_ros_codes` dans le répertoire `src` de l'espace de travail `my_ros2_ws` avec les commandes suivantes :

```
cd ~/my_ros2_ws/src  
ros2 pkg create --build-type ament_cmake my_first_ros_codes
```



Vérifiez en affichant le contenu du répertoire du paquet `my_first_ros_codes` que la commande a bien créé à l'intérieur les deux répertoires `src` et `include` ainsi que les deux fichiers de configuration `package.xml` (le manifeste) et `CMakeLists.txt`.



**On a créé notre premier paquet !**

On va pouvoir coder notre premier nœud maintenant !!!


### 1.3 Premier programme : nœud Hello ROS!


Un exemple valant plus qu'un long discours, voici un premier code ROS, `hello_world.cpp` :


```
#include "rclcpp/rclcpp.hpp"

int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv); // Launch the necessary initializations
    auto node = std::make_shared<rclcpp::Node>("hello_world_node"); // Create a new node
    RCLCPP_INFO(node->get_logger(), "Hello, world!"); // Send a message to the console
    rclcpp::spin(node); // Handle callback functions
    rclcpp::shutdown(); // Close the node properly
    return 0;
}
```

Dans ce programme, après avoir lancé les initialisations, on crée un nœud qui porte comme nom `"hello_world_node"` puis on affiche un message sur la console en unique action de ce nouveau nœud avant de le déconstruire.


 La première ligne permet d'inclure le fichier *header* `"rclcpp.hpp"` (pour "ROS Client Library version C++") des fonctionnalités de base de ROS.

 Notre nœud est créé à partir de la classe `Node` (tous nos autres nœuds le seront également), c'est une application du *Design Pattern Composite*. Ce concept de *composition* va nous permettre de gérer un ensemble d'objets en tant qu'un seul et même objet. Ainsi un nœud pourra créer des publieurs, des écouteurs, des serveurs ou des clients par exemple, qui seront également issus de la classe `Node`.

 La méthode `RCLCPP_INFO` est une sorte d'équivalent de `echo` ou `printf`, on indique l'identifiant du nœud qui écrit le message dans la console (`node->get_logger()`).

 On reviendra sur la notion de *callbacks* (`rclcpp::spin(node)`) un peu plus tard.

#### EXERCICE 3 (Création du nœud Hello ROS!)

 Recopiez le code précédent dans un fichier que vous nommerez `hello_world.cpp` et que vous placerez dans le répertoire `src` de votre paquet `my_first_ros_codes`.

### 1.4 Compilation & exécution

Maintenant, on va vouloir compiler notre premier programme pour pouvoir ensuite l'exécuter. Pour cela, nous allons devoir :

- Déclarer les dépendances,
- Déclarer les exécutable,
- Puis lancer l'outil de compilation.

## EXERCICE 4 (*Déclaration des dépendances*)



Affichez le contenu du manifeste (`package.xml`) de votre paquet et ajoutez grâce à la balise `<depend>` les dépendances avec les autres paquets juste après la balise `<buildtool_depend>`. Dans notre cas, nous avons une dépendance à `rclcpp`, donc nous devons ajouter la ligne suivante :

```
<depend>rclcpp</depend>
```

Enregistrez et fermez le manifeste puis ouvrez le fichier `CMakeLists.txt` de votre paquet pour indiquer qu'il faut rechercher ces dépendances avec la fonction `find_package`. Vous rajouterez donc la ligne suivante :

```
find_package(rclcpp REQUIRED)
```

Enregistrez le fichier `CMakeLists.txt` (mais ne le fermez pas).

## EXERCICE 5 (*Déclaration des exécutables*)



Il faut ensuite déclarer les exécutables à compiler dans le fichier `CMakeLists.txt`. Pour cela, nous allons utiliser la fonction `add_executable`. Dans notre cas, nous en avons un seul, donc vous ajouterez la ligne suivante, juste après les lignes `find_package` :

```
add_executable(hello_node src/hello_world.cpp)
```



Le nom que l'on donne à l'exécutable peut être différent du nom du fichier : ici on l'a appelé "hello\_node"

Ensuite, on s'assure de gérer les dépendances de nos exécutables lors de la compilation grâce à la fonction `ament_target_dependencies`. Nous ajouterons donc après notre dernière ligne la ligne suivante :

```
ament_target_dependencies(hello_node rclcpp)
```



Notre nœud ne dépend pour l'instant que de `rclcpp`, si on avait d'autres dépendances, on les aurait énumérées séparées par des espaces.

Et pour finir, on ajoute le nom de notre exécutable (un nom d'exécutable par ligne) à la liste des cibles (TARGETS) de la fonction `install`, avant l'appel à `ament_package()` qui termine le fichier :

```
install(TARGETS
  hello_node
  DESTINATION lib/${PROJECT_NAME})
```

Et c'est tout ! Enregistrez et fermez le fichier `CMakeLists.txt`.

Si tout va bien, votre fichier `package.xml` modifié devrait ressembler à ceci (avec deux trois trucs en plus) :

```
<?xml version="1.0"?>
<package format="3">
  <name>my_first_ros_codes</name>
  <version>0.0.0</version>
  <description>My first ROS codes</description>
  <maintainer email="me@todo.todo">me</maintainer>
  <license>Best license</license>

  <buildtool_depend>ament_cmake</buildtool_depend>
  <depend>rclcpp</depend>

  <test_depend>ament_lint_auto</test_depend>
```

```

<test_depend>ament_lint_common</test_depend>

<export>
  <build_type>ament_cmake</build_type>
</export>
</package>

```

Et votre fichier CMakeLists.txt modifié devrait ressembler à ceci (avec deux trois trucs en plus) :

```

cmake_minimum_required(VERSION 3.8)
project(my_firt_ros_codes)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)

add_executable(hello_node src/hello_world.cpp)
ament_target_dependencies(hello_node rclcpp)

install(TARGETS
  hello_node
  DESTINATION lib/${PROJECT_NAME})

ament_package()

```

## EXERCICE 6 (*Compilation*)



Pour compiler, on se replace d'abord dans l'espace de travail :

```
cd ~/my_ros2_ws/
```

Puis on vérifie que l'on n'a pas oublié des dépendances avec l'outil **rosdep** :

```
rosdep install -i --from-path src --rosdistro humble -y
```

Et enfin, la compilation a proprement parler avec l'outil **colcon** :

```
colcon build
```

Si tout s'est bien passé, vous devriez avoir affiché dans le terminal les lignes suivantes :

```

Starting >>> my_first_ros_codes
Finished <<< my_first_ros_codes [0.12s]

Summary: 1 package finished [0.28s]

```

## EXERCICE 7 (*On lance notre nœud ?*)



On vient donc de compiler notre premier nœud ! On essaye de le lancer ? On reprend nos bonnes habitudes :

```
ros2 run my_first_ros_codes hello_node
```



**Package not found !** Ça ne fonctionne pas car ROS2 ne trouve pas notre espace de travail !



Notre espace de travail ne fait pas parti pour l'instant des endroits où **ros2** va chercher les paquets et leurs exécutable.

Pour compléter les endroits explorés par `ros2`, il faut "sourcer" un nouveau fichier qui contient toutes les informations relatives à notre espace de travail. Ce fichier, généré automatiquement par `colcon`, s'appelle `local_setup.bash` et se trouve dans le répertoire `install` à la racine de l'espace de travail.



**Votre espace de travail, et donc vos paquets, sont prioritaires par rapport aux autres.** En conséquence, si vous écrivez dans l'espace de travail un nœud dans un paquet qui portent tous deux le même nom que ceux classiques, ce sera votre nœud qui sera lancé et pas le nœud classique.

### EXERCICE 8 (*Sourcer le workspace*)



Depuis la racine de votre espace de travail, lancez la commande suivante :

```
source install/local_setup.bash
```



**Pour que les toutes les variables d'environnements propres à votre espace de travail soient correctement initialisées**, il faut impérativement "sourcer" le fichier `install/local_setup.bash` à la racine de l'espace de travail à chaque fois que vous utilisez un nouveau terminal.

### EXERCICE 9 (*On (re)lance notre nœud ?*)



Retentez votre commande :

```
ros2 run my_first_ros_codes hello_node
```



**Ça fonctionne !!!** Vous devriez voir apparaître une ligne de ce genre s'afficher dans le terminal (CTRL+C pour stopper le programme) :

```
[INFO] [1674835235.003781340] [hello_world_node]: Hello, world!
```

- ⇒ [INFO] signifie que le message est de type informatif (on verra plus tard les autres type de message possible)
- ⇒ [1674835....] c'est la date en secondes depuis le 01/01/1970
- ⇒ [hello\_world\_node] c'est le nom du nœud qui émet le message
- ⇒ Hello, world! c'est le contenu du message, celui qu'on a écrit dans notre nœud !

## 1.5 Personnaliser un nœud

On peut avoir besoin de rajouter des attributs ou des méthodes spécifiques à nos nœuds. Par exemple, si on veut publier le message précédent de manière périodique, on veut ajouter un *timer*. Pour cela, on réimplémente la classe `Node`.

```
#include "rclcpp/rclcpp.hpp"





class MyNode : public rclcpp::Node
{
public:
    MyNode() : Node("hello_world_custom_node")
    {
        timer_ = this->create_wall_timer(
            std::chrono::milliseconds(500),
            std::bind(&MyNode::timerCallback, this));
    }
private:
    void timerCallback()
```

```

{
    RCLCPP_INFO(this->get_logger(), "Hello, world!");
}
rclcpp::TimerBase::SharedPtr timer_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<MyNode>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}

```

-  Le nom du nœud, "hello\_world\_custom\_node", est initialisé au moment de l'appel à son constructeur.
-  `rclcpp::TimerBase::SharedPtr timer_` est un attribut privé permettant de rattacher un minuteur à notre nœud.
-  `timerCallback` est un *callback*. C'est une fonction privée qui sera exécutée automatiquement à chaque fois que le temps du minuteur sera écoulé.
-  `create_wall_timer` permet d'initialiser le minuteur au moment de l'appel du constructeur avec comme arguments le temps du minuteur et la fonction de *callback* à lancer.

## EXERCICE 10 (Nœud personnalisé)



Recopiez le code précédent dans un fichier `hello_world_custom_node.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud "hello\_custom\_node". Si tout s'est bien passé, exécutez votre nouveau nœud :

```
ros2 run my_first_ros_codes hello_custom_node
```



Vous devriez voir apparaître une ligne similaire à notre précédent nœud qui s'affiche cette fois-ci en boucle, selon une période de 500 ms avec une précision de  $10^{-5}$  s.

## 2 Les forums (*topics*)

### 2.1 Publier sur un forum

Pour créer un nœud publieur, on va reprendre notre classe personnalisée, conserver le *timer* qui va nous permettre de publier un message de manière périodique sur un forum et décrire le contenu du message :

```

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class MyPublisher : public rclcpp::Node
{
public:
    MyPublisher() : Node("my_publisher_node"), count_(0)
    {
        publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
        timer_ = this->create_wall_timer(
            std::chrono::milliseconds(500),

```

```

        std::bind(&MyPublisher::timer_callback, this));
    }

private:
    void timer_callback()
    {
        auto message = std_msgs::msg::String();
        message.data = "Ping " + std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
        publisher_>publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MyPublisher>());
    rclcpp::shutdown();
    return 0;
}

```



**Au moment de l'appel au constructeur**, on peut non seulement initialiser le nom du nœud, mais aussi passer d'autres arguments, comme ici un compteur.



**La méthode `create_publisher`** permet de créer un nœud qui publie avec comme argument le nom du forum où publier et le nombre maximal de messages stockés dans la file d'attente. L'envoi du message sur le forum se fait ensuite via la méthode `publish`.



**Pour notre message à publier**, on a choisi le type `std_msgs::msg::String`.



**C'est une bonne pratique** de prendre l'habitude d'afficher chaque message émis sur un forum via `RCLCPP_INFO`.

### EXERCICE 11 (*Publication périodique sur un forum*)



Recopiez le code précédent dans un fichier `publish_node.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud "`publish_node`".



**Deux dépendances** : attention, cette fois-ci on a une nouvelle dépendance à ajouter, celle à `std_msgs`.

Lancez la compilation, puis si tout s'est bien passé, exécutez votre nouveau nœud :

```
ros2 run my_first_ros_codes publish_node
```



Vous devriez voir apparaître à une période de 500 *ms* (avec une précision de  $10^{-5}$  s) des lignes similaires à ce qui suit avec un compteur qui s'incrémente indéfiniment :

```
[INFO] [1675103101.140736994] [my_publisher_node]: Publishing: 'Ping 0'
[INFO] [1675103101.640724414] [my_publisher_node]: Publishing: 'Ping 1'
[INFO] [1675103102.140722366] [my_publisher_node]: Publishing: 'Ping 2'
```



**Ne stoppez pas encore l'exécution de votre nœud !**



## EXERCICE 12 (*Publication ? Vérifions !*)



Pour l'instant, ce que nous avons vu, ce sont les messages envoyés sur la console. Nous allons nous assurer que notre nœud publie bien sur le forum. On avait appelé notre forum `"topic"`. Vérifions dans un premier temps sa présence dans un nouveau terminal avec la ligne que nous avons vu dans le précédent TD :

```
ros2 topic list -t
```



**Est-ce que vous le voyez bien dans la liste avec le bon type de message ?**

Si oui, on va pouvoir afficher les messages qu'il reçoit, encore une fois en réutilisant une ligne de commande que nous avons vu précédemment :

```
ros2 topic echo /topic
```



**Si tout fonctionne**, vous devriez obtenir quelque chose de similaire à ça :

```
data: Ping 0
---
data: Ping 1
---
```



**Mettons maintenant à profit** ce nouveau savoir-faire dans un cadre plus complexe : reprenons `turtlesim` et codons un nœud qui dirige la tortue !

## EXERCICE 13 (*The drunken turtle step (mouvements aléatoires) – Code*)



Dupliquez le fichier précédent en un fichier `turtle_random_node.cpp` dans le dossier `src` de votre paquet `my_first_ros_codes` et modifiez le code pour que le nœud publie à une période de 500 ms une vitesse linéaire aléatoire entre 0 et 1 et une vitesse angulaire aléatoire entre -1.5 et 1.5 radians sur le forum `turtle1/cmd_vel` utilisé par le programme `turtlesim`.



**Type de message** : ajoutez la dépendance `geometry_msgs/msg/twist.hpp` pour pouvoir utiliser les messages de type `Twist`.



**Adaptez en conséquence** : dans le constructeur et la *callback* modifiez le type de message (`<geometry_msgs::msg::Twist>`), son contenu et le nom du forum (`"turtle1/cmd_vel"`) sur lequel publier.



**Pour rappel**, un message de type `Twist` contient deux champs `"linear"` et `"angular"`, chacun contenant eux-même trois champs pour `x`, `y` et `z`. Exemple pour modifier la vitesse linéaire en `x`  $\Rightarrow$  `message.linear.x = 2.0;`.



`double(rand())/RAND_MAX` permet d'obtenir un nombre aléatoire entre 0 et 1 et `3.*double(rand())/RAND_MAX-1.5` un nombre aléatoire entre -1.5 et 1.5. Souvenez-vous que seuls la vitesse linéaire en `x` et la vitesse angulaire en `z` peuvent être non nuls.



**Pour afficher les valeurs directement** (sans les convertir en chaîne de caractères), on utilise le pattern `%.3f`.

```
RCLCPP_INFO(this->get_logger(), "Publishing: linear=%.3f, angular=%.3f", message.linear.x, message.angular.z);
```



**Pensez à initialiser la graine aléatoire** avec `srand(time(0))` dans le `main`.

## EXERCICE 14 (*The drunken turtle step (mouvements aléatoires)* – *Run*)



Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud `"turtle_random_node"`.



**N'oubliez pas la nouvelle dépendance !**

Lancez la compilation et si tout s'est bien passé, lancez dans un premier terminal le simulateur :

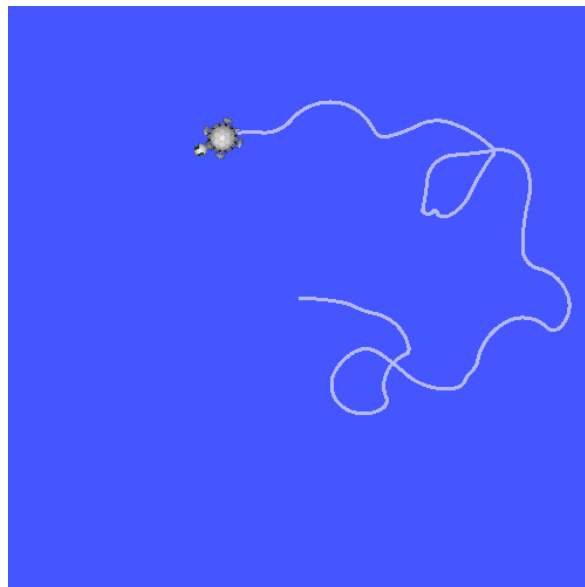
```
ros2 run turtlesim turtlesim_node
```

Et dans un second terminal, lancez votre nœud publiant qui contrôle la tortue avec des ordres aléatoires (sans oublier de sourcer l'espace de travail en premier lieu) :

```
ros2 run my_first_ros_codes turtle_random_node
```



**Profitez du spectacle !**



## 2.2 Lire sur un forum

Le code suivant permet de créer un nœud qui va écouter ce qui est publié sur un forum et réagir à chaque nouveau message :

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;

class MySubscriber : public rclcpp::Node
{
public:
    MySubscriber() : Node("my_subscriber_node")
    {
        subscription_ = this->create_subscription<std_msgs::msg::String>(
            "topic", 10, std::bind(&MySubscriber::topic_callback, this, _1));
    }
};
```


```

    }

private:
    void topic_callback(const std_msgs::msg::String & msg) const
    {
        RCLCPP_INFO(this->get_logger(), "Receiving: '%s' Answering : Pong !",
                    msg.data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MySubscriber>());
    rclcpp::shutdown();
    return 0;
}

```

 **Il n'y a pas de minuteur (timer)** car la fonction de rappel est lancée dès qu'un message est posté sur le forum.

 **Attention à bien être vigilant** à avoir le même type de message entre le nœud qui publie et celui qui écoute !

### EXERCICE 15 (*Lecture dans un forum*)



Recopiez le code précédent dans un fichier `subscribe_node.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud "`subscribe_node`".

Lancez la compilation, puis si tout s'est bien passé, exécutez dans un premier terminal votre nouveau nœud :

```
ros2 run my_first_ros_codes subscribe_node
```

Et dans un deuxième terminal, votre nœud précédent :

```
ros2 run my_first_ros_codes publish_node
```

 **Ça marche ?** Si oui, vous devriez avoir quelque chose de similaire à ces lignes :

 **Dans le terminal de l'émetteur :**

```

[INFO] [1675177000.863048151] [my_publisher_node]: Publishing: 'Ping 0'
[INFO] [1675177001.363055906] [my_publisher_node]: Publishing: 'Ping 1'
[INFO] [1675177001.863055868] [my_publisher_node]: Publishing: 'Ping 2'
[INFO] [1675177002.363029746] [my_publisher_node]: Publishing: 'Ping 3'
[INFO] [1675177002.863025308] [my_publisher_node]: Publishing: 'Ping 4'


```

 **Dans le terminal de l'écouteur :**

```

[INFO] [1675177000.863707210] [my_subscriber_node]: Receiving: 'Ping 0' Answering : Pong !
[INFO] [1675177001.363505862] [my_subscriber_node]: Receiving: 'Ping 1' Answering : Pong !
[INFO] [1675177001.863277248] [my_subscriber_node]: Receiving: 'Ping 2' Answering : Pong !
[INFO] [1675177002.363488603] [my_subscriber_node]: Receiving: 'Ping 3' Answering : Pong !
[INFO] [1675177002.863539340] [my_subscriber_node]: Receiving: 'Ping 4' Answering : Pong !

```

 **Quel écart de temps trouvez-vous en millisecondes entre les affichages Ping et Pong** (soustraire les deux timestamp et multiplier par 1000 pour passer des secondes en millisecondes) ?

## 2.3 Créer son propre type de message

Jusqu'à présent nous avons utilisé des types de messages déjà existants dans d'autres paquets. Voyons maintenant comment créer ses propres types de messages.



Les messages sont tous regroupés dans un paquet dédié aux interfaces, que ce soit des messages échangés sur des forums, ou entre un client et un serveur dans le cadre d'un service.

### EXERCICE 16 (*Paquet dédié aux interfaces*)



Créez le nouveau paquet `my_first_ros_interfaces` dans le répertoire `src` de l'espace de travail `my_ros2_ws`. Reportez-vous pour cela à la ligne de commande que nous avons vu dans l'exercice 2 page 2.

Rendez-vous dans ce nouveau répertoire, créez-y un nouveau répertoire `msg`, puis supprimez les répertoires `include` et `src` qu'il contient (ils ne nous seront pas utiles car on n'écrit pas de nœud dans ce paquet) :

```
cd my_first_ros_interfaces
mkdir msg
rm -rf ./include ./src
```



**Assurez-vous d'être vraiment au bon endroit** avant de lancer la commande de suppression (un petit `pwd` peut-être pour être sûr ?) !



**On se retrouve avec trois éléments** : le répertoire `msg` et les deux fichiers `CMakeLists.txt` et `package.xml`.

### EXERCICE 17 (*Paramétrage global du paquet dédié aux interfaces*)



Modifiez le manifeste (`package.xml`) pour lui indiquer que ce paquet s'occupe des interfaces en y ajoutant les lignes suivantes avant la balise `<test_depend>` :

```
<build_depend>roscpp</build_depend>
<exec_depend>roscpp</exec_depend>
<member_of_group>roscpp</member_of_group>
```

Puis modifiez le `CMakeLists.txt` en ajoutant les dépendances correspondantes à la suite de la première ligne `find_package` :

```
find_package(roscpp REQUIRED)
```

Et ajoutez également la liste des cibles à produire (vide pour le moment) avant l'appel à `ament_package()` qui termine le fichier :

```
ament_add_executable(${PROJECT_NAME}
  src/main.cpp
)
ament_target_dependencies(${PROJECT_NAME}
  roscpp
)
ament_package()
```



**On a notre paquet paramétré !**  
Maintenant, nous pouvons créer un nouveau type de message !!!



Tout nouveau type de message sera placé dans le dossier `msg` du paquet dédié aux interfaces.

Les types de messages sont des fichiers au format texte, qui ont pour extension `.msg` et pour nom le nom du type de message avec une majuscule (`Twist.msg` par exemple). Dans le TD précédent, on avait utilisé la commande suivante pour voir le contenu d'un fichier message (prenons `Twist` comme exemple) :

```
ros2 interface show geometry_msgs/msg/Twist
```

Cela nous affiche les lignes suivantes :

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3 linear
  float64 x
  float64 y
  float64 z
Vector3 angular
  float64 x
  float64 y
  float64 z
```



Sur chaque ligne du fichier, on a la description d'un champ avec son type et son nom.



Les lignes indentées décrivent sur le même principe (type et nom) les champs qui composent l'objet de la ligne principale : ce n'est pas à nous de les décrire, elles seront automatiquement récupérées de la description des objets existants.

### EXERCICE 18 (Nouveau type de message *Circle*)



Créez le fichier `Circle.msg` qui contient les champs suivants :

```
geometry_msgs/Point center
float64 radius
```

### EXERCICE 19 (Compilation du message de type *Circle*)



Complétez maintenant le manifeste avec la dépendance à `geometry_msgs` en ajoutant la ligne suivante comme on le faisait jusqu'à maintenant, juste avant la ligne `<build_depend>roscpp</build_depend>` :

```
<depend>geometry_msgs</depend>
```

Puis complétez le `CMakeLists.txt` avec la ligne pour rechercher la nouvelle dépendance, toujours comme on le faisait jusqu'à maintenant :

```
find_package(geometry_msgs REQUIRED)
```

Et enfin, ajoutez la nouvelle cible (comme on le faisait jusqu'à maintenant : une cible par ligne) tout en précisant la liste des dépendances utilisées par les cibles citées (on n'utilise pas `ament_target` dans le cas des interfaces) :

```
roscpp_generate_interfaces(${PROJECT_NAME}
  "msg/Circle.msg"
  DEPENDENCIES geometry_msgs # Add packages the messages depend on
)
```



On teste ? Lancez la compilation (`colcon build`) à la racine de l'espace de travail, sourcez et utilisez la commande `ros2 interface show my_first_ros_interfaces/msg/Circle` pour visualiser votre nouveau type.

## EXERCICE 20 (*Utilisation du message Circle : nœud publieur*)



Dupliquez le fichier `turtle_random_node.cpp` en un fichier `publish_circle_node.cpp` dans le dossier `src` de votre paquet `my_first_ros_codes` et modifiez le code pour que le nœud publie sur le forum `topic_circles` à une période de `500 ms` des messages de type `Circle` où le centre du cercle est placé de manière aléatoire entre 0 et 4 en `x` et en `y` et où le rayon est aléatoirement compris entre 0 et 1.



**Dépendances** : ajoutez la ligne `#include "my_first_ros_interfaces/msg/circle.hpp"` pour utiliser votre type de message `Circle`. Le fichier `.hpp` a été automatiquement créé par `colcon` (d'où l'importance de respecter les conventions sur le nom des fichiers).



**N'oubliez pas** d'adapter l'ensemble du code pour votre nouveau type de message et pour le forum voulu.



`message.center.x = 4.*double(rand())/RAND_MAX` permet d'avoir un nombre aléatoire entre 0 et 4 (idem pour `.y` et `.z` sera nul).



`message.radius = double(rand())/RAND_MAX` permet d'avoir un nombre aléatoire entre 0 et 1.

## EXERCICE 21 (*Utilisation du message Circle : nœud écouteur*)



Dupliquez le fichier `subscribe_node.cpp` en un fichier `subscribe_circle_node.cpp` dans le dossier `src` de votre paquet `my_first_ros_codes` et modifiez le code pour que le nœud écoute sur le forum `topic_circles` et affiche le contenu des messages reçus.



**Affichez** les valeurs des coordonnées `x`, `y` et `z` du centre ainsi que la valeur du rayon.



**N'oubliez pas** d'adapter l'ensemble du code pour votre nouveau type de message et pour le forum voulu.

## EXERCICE 22 (*Compile and run*)



Modifiez les fichiers de configuration pour pouvoir compiler les deux nouveaux nœuds que vous avez créé depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on nommera les exécutables "`publish_circle_node`" et "`subscribe_circle_node`".



Nos nouveaux nœuds dépendent du paquet `my_first_ros_interfaces`, c'est ce dernier qu'on ajoute dans le `CMakeLists.txt` (`find_package`, `ament_target_dependencies`).



Inutile dans `ament_target_dependencies` de préciser la dépendance à `geometry_msgs` : elle se fait via notre paquet `my_first_ros_interfaces`.

Lancez la compilation, puis si tout s'est bien passé, exécutez vos deux nouveaux nœud, chacun dans un terminal :

```
ros2 run my_first_ros_codes subscribe_circle_node
```

```
ros2 run my_first_ros_codes publish_circle_node
```



Est-ce que vous avez bien les mêmes valeurs et le z des coordonnées du centre à 0.0 ?

```
[INFO] [1675431588.735813095] [my_publisher_node]: Publishing: Center:x:0.922 center:y:3.072 radius:0.561
[INFO] [1675431589.235849645] [my_publisher_node]: Publishing: Center:x:0.632 center:y:0.407 radius:0.646
[INFO] [1675431589.735824445] [my_publisher_node]: Publishing: Center:x:2.511 center:y:0.754 radius:0.354
```

```
[INFO] [1675431588.736221766] [my_subscriber_node]: Receiving: Circle center x:0.922 y:3.072 z:0.000 and radius:0.561 Answering : Ok!
[INFO] [1675431589.236082238] [my_subscriber_node]: Receiving: Circle center x:0.632 y:0.407 z:0.000 and radius:0.646 Answering : Ok!
[INFO] [1675431589.736196026] [my_subscriber_node]: Receiving: Circle center x:2.511 y:0.754 z:0.000 and radius:0.354 Answering : Ok!
```

## 3 Services

Les services sont liés à la notion de client & serveur : un client envoie une requête à un serveur et attend que celui-ci lui renvoie la réponse.



Pour illustrer concrètement les notions de requête, client et serveur, on va construire un service qui va additionner les entiers passés en requête et retourner leur somme comme réponse.

Pour cela, on commence par définir le type de service, c'est-à-dire le type du message échangé entre le client et le serveur.

### 3.1 Créer son propre type de service



L'information qui s'échange entre le client et le serveur doit être définie par un type de service.

Vous pouvez soit utiliser des types de service déjà existants dans des paquets ROS, soit créer les vôtres. Créer son propre type de service est très similaire à créer son propre message de forum. On peut les ranger dans le même paquet que ces derniers d'ailleurs.

### EXERCICE 23 (*Complément du paquet dédié aux interfaces*)



À la section 2.3 page 12, vous avez créé le paquet `my_first_ros_interfaces`. Rendez-vous dans ce dernier et créez-y un nouveau répertoire `srv` (au même niveau donc que le répertoire `msg` que l'on avait créé pour les messages de forums).



**En configuration**, on ajoutera les dépendances dans le manifeste et les nouveaux types de service à générer (et leurs dépendances) dans la liste des cibles `rosidl_generate_interfaces` (avec les messages de forum).



**Tout nouveau type de service** sera placé dans le dossier `srv` du paquet dédié aux interfaces.

Les types de service, comme les messages de forums, sont des fichiers au format texte, qui ont pour extension `.srv` et pour nom le nom du type de service avec une majuscule (`Spawn.srv` par exemple). Dans le TD précédent, on avait utilisé la commande suivante pour voir le contenu d'un fichier de type de service (prenons `Spawn` comme exemple) :

```
ros2 interface show turtlesim/srv/Spawn
```

Cela nous affiche les lignes suivantes :

```
float32 x
float32 y
float32 theta
string name # Optional. A unique name will be created and returned if this is empty
---
string name
```



**Sur chaque ligne du fichier**, on a la description d'un champ avec son type et son nom.



**---** Indique la séparation entre le format de la requête du client (au-dessus) et le format de la réponse du serveur (en dessous).

### EXERCICE 24 (*Nouveau type de service AddTwoInts*)



Créez le fichier `AddTwoInts.srv` qui contient les champs suivants :

```
int64 a
int64 b
---
int64 sum
```

### EXERCICE 25 (*Compilation du type de service AddTwoInts*)



Complétez le `CMakeLists.txt` en ajoutant la nouvelle cible (comme on le faisait jusqu'à maintenant : une cible par ligne). Nous n'utilisons pas de dépendances, donc pas besoin de modifications sur ce point.

Puis lancez la compilation (`colcon build`) à la racine de l'espace de travail. Si tout s'est bien passé, sourcez et utilisez la commande suivante pour visualiser votre nouveau type de service :

```
ros2 interface show my_first_ros_interfaces/srv/AddTwoInts
```



**Est-ce que ça correspond bien ?**



**Votre nouveau type de service existe désormais !** Mais pour l'utiliser, vous devez définir dans chaque paquet qui l'utilise comment émettre une requête et comment y répondre.



## 3.2 Recevoir et traiter une requête (serveur)



Dans notre exemple de service qui va additionner les entiers passés à travers une requête et retourner leur somme comme réponse, commençons par la partie serveur (celle qui fait la somme).

Le code suivant permet de créer un nœud qui peut recevoir une requête de type `AddTwoInts`, de réaliser l'addition des deux entiers reçus et de renvoyer le résultat en retour :

```
#include "rclcpp/rclcpp.hpp"
#include "my_first_ros_interfaces/srv/add_two_ints.hpp"
using namespace std::placeholders;

class MyServerNode : public rclcpp::Node
{
public:
    MyServerNode() : Node("add_two_ints_server_node")
    {
        service_ = this->create_service<my_first_ros_interfaces::srv::AddTwoInts>("add_two_ints",
            std::bind(&MyServerNode::addCallback, this, _1, _2));
    }

private:
    void addCallback(
        const std::shared_ptr<my_first_ros_interfaces::srv::AddTwoInts::Request> request,
        std::shared_ptr<my_first_ros_interfaces::srv::AddTwoInts::Response> response)
    {
        response->sum = request->a + request->b;
        RCLCPP_INFO(this->get_logger(), "Incoming request\na: %ld" " b: %ld",
            request->a, request->b);
        RCLCPP_INFO(this->get_logger(), "sending back response: [%ld]",
            (long int)response->sum);
    }
    rclcpp::Service<my_first_ros_interfaces::srv::AddTwoInts>::SharedPtr service_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
    rclcpp::spin(std::make_shared<MyServerNode>());
    rclcpp::shutdown();
}
```



**Dépendances :** ajoutez la ligne `#include "my_first_ros_interfaces/srv/add_two_ints.hpp"` pour utiliser votre type de service `AddTwoInts`. Le fichier `.hpp` a été automatiquement créé par `colcon` (d'où l'importance de respecter les conventions sur le nom des fichiers).



`std::placeholders` nous permet de définir les arguments qui restent variables (ie. qui ne sont pas fixés par défaut) lors de l'utilisation de la fonction `bind`. Ces arguments variables seront nommés `_1`, `_2`, `_3`, etc., pour être utilisés, chacun correspondant à l'ordre des paramètres de la fonction redéfinie.



`create_service` permet de créer le service et de le lier avec la fonction de rappel privée `addCallback`. Cette fonction sera lancée par le serveur lorsque le service est appelé. C'est dans cette dernière que l'on exécute la requête (ici on additionne deux entiers) et où on écrit la réponse.

## EXERCICE 26 (Nœud serveur pour additionner deux entiers)



Recopiez le code précédent dans un fichier `add_two_ints_server.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud `"add_two_ints_server_node"`.



**Dépendances :** on a déjà rajouté la dépendance au paquet `my_first_ros_interfaces` dans les fichiers de configuration au cours des exercices précédents. On a juste besoin de préciser que ce code utilise cette dépendance via `ament_target_dependencies`.

Lancez la compilation, puis si tout s'est bien passé, exécutez votre nouveau nœud :

```
ros2 run my_first_ros_codes add_two_ints_server_node
```



Voyez-vous bien apparaître à l'écran une ligne qui vous dit que le serveur est prêt et attend une requête ?

```
[INFO] [1675520270.201278494] [rclcpp]: Ready to add two ints.
```



Le service est maintenant en attente de sollicitation par un programme client.



Laissez tourner le serveur pour le moment.

### 3.3 Émettre une requête (client)

Le code suivant permet de créer un nœud qui peut envoyer une requête de type `AddTwoInts` composée de deux entiers, attend que le serveur traite sa demande, réceptionne la réponse du serveur et l'affiche :

```
#include "rclcpp/rclcpp.hpp"
#include "my_first_ros_interfaces/srv/add_two_ints.hpp"

using namespace std::placeholders;

class MyClientNode : public rclcpp::Node
{
public:
    MyClientNode() : Node("add_two_ints_client_node")
    {
        client_ = this->create_client<my_first_ros_interfaces::srv::AddTwoInts>("add_two_ints");
    }

    void call_service(char *argv[])
    {
        while (!client_->wait_for_service(std::chrono::milliseconds(1000))) {
            if (!rclcpp::ok()) {
                RCLCPP_ERROR(this->get_logger(), "Interrupted while waiting for the service. Exiting.");
                rclcpp::shutdown();
                std::exit(1);
            }
            RCLCPP_INFO(this->get_logger(), "Service not available, waiting again...");
        }
        auto request = std::make_shared<my_first_ros_interfaces::srv::AddTwoInts::Request>();
        request->a = atoll(argv[1]);
        request->b = atoll(argv[2]);
        RCLCPP_INFO(this->get_logger(), "Sending request");
        auto future = client_->async_send_request(request);
```

```

    if ( rclcpp::spin_until_future_complete(this->get_node_base_interface(), future) ==
        rclcpp::FutureReturnCode::TIMEOUT )
    {
        client_->remove_pending_request(future);
        RCLCPP_ERROR(this->get_logger(), "Timeout : failed to receive service response");
        rclcpp::shutdown();
        std::exit(1);
    }
    auto response = future.get();
    RCLCPP_INFO(this->get_logger(), "Result of add_two_ints: %ld", response.get()->sum);
}

private:
    rclcpp::Client<my_first_ros_interfaces::srv::AddTwoInts>::SharedPtr client_;
};

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    if (argc != 3) {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client_node X Y");
        rclcpp::shutdown();
        return 1;
    }
    auto node = std::make_shared<MyClientNode>();
    node->call_service(argv);
    rclcpp::shutdown();
    return 0;
}

```



`create_client` permet de créer le nœud client (et donc de savoir qu'il va manipuler une requête et une réponse). Contrairement aux exemples précédents, il n'y a pas de fonction de rappel associée pour ce nœud.



**La méthode publique** `call_service` lance la requête et traite la réponse, elle est appelée depuis le main, après la création du nœud, au moment où on a besoin du service.



`wait_for_service` permet de s'assurer que le client a bien eu le temps de déterminer quels sont les services et forums disponibles. Si on ne fait pas cette étape et que l'on tente de lancer la requête directement, il y a un fort risque d'être dans une attente de réponse infinie.



**Les champs de la requête** (les deux entiers à additionner) sont initialisés à partir des valeurs passées sur la ligne de commande lors du lancement du nœud (`argv`).



`async_send_request` est une méthode qui lance la requête, et `future` est un objet créé immédiatement et qui contiendra dans un futur qu'on espère proche la réponse à la requête (d'où son nom).



`spin_until_future_complete` est la méthode qui permet au nœud d'attendre que la réponse à sa requête arrive.

## EXERCICE 27 (Nœud client pour additionner deux entiers)



Recopiez le code précédent dans un fichier `add_two_ints_client.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud "`add_two_ints_client_node`".

Lancez la compilation, puis si tout s'est bien passé, exécutez votre nouveau nœud dans un nouveau terminal (n'oubliez pas de sourcer) :

```
ros2 run my_first_ros_codes add_two_ints_client_node 3 4
```



**N'oubliez pas que votre nœud serveur doit être en train de tourner avant de lancer le nœud client.**



**N'oubliez pas les deux entiers en arguments de votre ligne de commande !**



**Côté serveur, vous devez obtenir un résultat comparable à :**

```
[INFO] [1675703843.235959439] [rclcpp]: Incoming request a: 3 b: 4
[INFO] [1675703843.236034098] [rclcpp]: sending back response: [7]
```



**Et côté client, un résultat comparable à :**

```
[INFO] [1675703843.235264510] [add_two_ints_client_node]: Sending request
[INFO] [1675703843.236500639] [add_two_ints_client_node]: Result of add_two_ints: 7
```



**Combien de temps s'écoule-t-il entre l'émission de la requête, la réception par le serveur et le retour de la réponse ?**



**N'hésitez pas à tester avec d'autres entiers !**



**Entre l'émission et la réception de la requête,**

il s'est écoulé de l'ordre de 0,77 *ms*, le traitement par le serveur a pris environ 0,08 *ms* et le temps entre l'émission par le serveur et la réception puis l'affichage par le client est de l'ordre de 0,47 *ms*. En tout, l'ensemble du processus a pris environ 1,24 *ms*.

## 4 Paramètres

Comme on l'a vu lors de la découverte des commandes en ligne de ROS2, les paramètres permettent de passer des informations à un nœud de manière très exceptionnelle et de façon très économe en ressources.



**Par exemple,** ils sont utilisés pour changer des variables initialisées au moment du lancement du nœud comme par exemple la couleur de fond de la fenêtre de `turtlesim`.

### 4.1 Créer des paramètres pour un nœud

Pour comprendre comment créer et manipuler des paramètres, reprenons notre premier code `hello_world_custom_node.cpp` qui affichait en boucle son message dans la console et ajoutons lui un paramètre : une chaîne de caractère initialisée par défaut à "`world`".

```
#include "rclcpp/rclcpp.hpp"

class MyNode : public rclcpp::Node
{
public:
    MyNode() : Node("hello_world_parameter_node")
    {
        this->declare_parameter("target_parameter", "world");
        timer_ = this->create_wall_timer(std::chrono::milliseconds(1000), std::bind(&MyNode::
            timerCallback, this));
    }

private:
    void timerCallback()
    {
        std::string target_param = this->get_parameter("target_parameter").get_parameter_value().
            get<std::string>();
        RCLCPP_INFO(this->get_logger(), "Hello %s!", target_param.c_str());
    }
    rclcpp::TimerBase::SharedPtr timer_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<MyNode>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

- ✓ `declare_parameter` permet de créer le paramètre `target_parameter` du nœud `hello_world_parameter_node` et de l'initialiser avec la valeur `"world"`.
- i `get_parameter` et `get_parameter_value` permettent de récupérer la valeur du paramètre. Les méthodes `get<>` et `c_str()` permettent d'obtenir le type `char *` attendu.

### EXERCICE 28 (Nœud Hello world avec paramètre)



Recopiez le code précédent dans un fichier `hello_world_parameter.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud `"hello_world_parameter_node"`.

Lancez la compilation, puis si tout s'est bien passé, exécutez votre nouveau nœud :

```
ros2 run my_first_ros_codes hello_world_parameter_node
```



**Voyez-vous bien apparaître à l'écran en boucle l'affichage que fait le nœud ?**

```
[INFO] [1675786016.127521337] [hello_world_parameter_node]: Hello world!
[INFO] [1675786017.127424678] [hello_world_parameter_node]: Hello world!
[INFO] [1675786018.127473893] [hello_world_parameter_node]: Hello world!
```



**Laissez tourner ce nœud pour le moment.**

## EXERCICE 29 (*Changer le paramètre du nœud Hello world*)



Dans un autre terminal, après avoir sourcé, commencer par lister les paramètres disponibles avec la commande que nous avons vue dans le TD précédent :

```
ros2 param list
```



**Est-ce que vous voyez bien apparaître pour notre nœud `hello_world_parameter_node` notre nouveau paramètre `target_parameter` dans la liste ?**

Alors on peut le changer, toujours en utilisant la commande que nous avons vue dans le TD précédent :

```
ros2 param set /hello_world_parameter_node target_parameter earth
```



**Avez-vous bien le message `Set parameter successful` qui s'affiche dans la console ?** Alors basculez sur le terminal où tourne le nœud pour vérifier qu'il utilise bien la nouvelle valeur du paramètre !

```
[INFO] [1675786057.127524735] [hello_world_parameter_node]: Hello world!
[INFO] [1675786058.127497827] [hello_world_parameter_node]: Hello earth!
[INFO] [1675786059.127501115] [hello_world_parameter_node]: Hello earth!
```



**On sait maintenant ajouter un paramètre à un nœud et le modifier depuis l'interface de commande en ligne, mais peut-on le modifier depuis un nœud ?**

## 4.2 Modifier des paramètres depuis un nœud

Les paramètres d'un nœud peuvent être directement modifiés depuis d'autres nœuds. Pour illustrer cette méthode, reprenons notre exemple et créons un deuxième nœud qui aura pour tâche d'aller modifier le paramètre `target_parameter` du nœud `hello_world_parameter_node` :






```
#include "rclcpp/rclcpp.hpp"

class MyNode : public rclcpp::Node
{
public:
    MyNode() : Node("change_hello_world_parameter_node")
    {
    }
    void changeParameter(char *argv[]){
        auto parameters_client = std::make_shared<rclcpp::SyncParametersClient>(this, "
            hello_world_parameter_node");
        while (!parameters_client->wait_for_service(std::chrono::milliseconds(1000))) {
            if (!rclcpp::ok()) {
                RCLCPP_ERROR(this->get_logger(), "Interrupted while waiting for the service. Exiting.
                ");
                rclcpp::shutdown();
                std::exit(1);
            }
            RCLCPP_INFO(this->get_logger(), "Service not available, waiting again...");
        }
        parameters_client->set_parameters({rclcpp::Parameter("target_parameter", argv[1])});
    }
private:
};
```

```

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    if (argc != 2) {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: change_hello_world_paramater
        newStringName ");
        return 1;
    }
    auto node = std::make_shared<MyNode>();
    node->changeParameter(argv);
    rclcpp::shutdown();
    return 0;
}

```

-  **Les paramètres sont gérés par un serveur**, donc le nœud qui veut les modifier doit être un client de ce serveur.
-  **La classe `rclcpp::SyncParametersClient`** permet de se connecter au serveur de paramètre du nœud `hello_world_parameter_node` et d'accéder ainsi aux paramètres de celui-ci. Ici, la connexion est synchrone (c'est-à-dire bloquante tant qu'elle n'est pas faite). Elle peut l'être de manière asynchrone avec la classe `rclcpp::AsyncParametersClient`.
-  **Comme pour tout service client**, il faut commencer par s'assurer que la connexion avec le service est bien opérationnelle avec `wait_for_service`.
-  **La méthode `set_parameters`** permet de changer la valeur du paramètre.
-  **Il n'y a pas de fonction `spin`** mais seulement un appel à la méthode publique `changeParameter`.

### EXERCICE 30 (Modifier un paramètre depuis un autre nœud)



Recopiez le code précédent dans un fichier `change_hello_world_parameter.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de notre nouveau nœud "`change_hello_world_parameter_node`".

Lancez la compilation, puis si tout s'est bien passé, exécutez dans un premier terminal le nœud `change_hello_world_parameter_node` :

```
ros2 run my_first_ros_codes hello_world_parameter_node
```

Et dans un second terminal, après avoir sourcé, votre nouveau nœud `change_hello_world_parameter_node` auquel vous passerez "me" en argument :

```
ros2 run my_first_ros_codes change_hello_world_parameter_node me
```



**Voyez-vous bien apparaître le changement d'affichage dans le premier terminal ?**

```

[INFO] [1675788659.427837224] [hello_world_parameter_node]: Hello world!
[INFO] [1675788660.427724250] [hello_world_parameter_node]: Hello me!
[INFO] [1675788661.427795711] [hello_world_parameter_node]: Hello me!

```

### 4.3 Ajout d'un *callback* lors de la modification d'un paramètre

Enfin, voici un dernier exemple de cas d'usage qui consiste à déclencher une fonction de rappel (*callback*) quand un paramètre est mis à jour :

```
#include "rclcpp/rclcpp.hpp"

class MyNode : public rclcpp::Node
{
public:
    MyNode() : Node("hello_world_parameter_node")
    {
        this->declare_parameter("target_parameter", "world");
        timer_ = this->create_wall_timer(std::chrono::milliseconds(1000), std::bind(&MyNode::
            timerCallback, this));

        param_subscriber_ = std::make_shared<rclcpp::ParameterEventHandler>(this);
        auto cb = [this](const rclcpp::Parameter & p) {
            RCLCPP_INFO(
                this->get_logger(), "parameter \"%s\" of type %s update with: \"%s\"",
                p.get_name().c_str(),
                p.get_type_name().c_str(),
                p.as_string().c_str()
            );
        };
        cb_handle_ = param_subscriber_->add_parameter_callback("target_parameter", cb);
    }
private:
    void timerCallback()
    {
        std::string target_param = this->get_parameter("target_parameter").get_parameter_value().
            get<std::string>();
        RCLCPP_INFO(this->get_logger(), "Hello %s!", target_param.c_str());
    }
    rclcpp::TimerBase::SharedPtr timer_;
    std::shared_ptr<rclcpp::ParameterEventHandler> param_subscriber_;
    std::shared_ptr<rclcpp::ParameterCallbackHandle> cb_handle_;
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<MyNode>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```



La classe `rclcpp::ParameterEventHandler` permet de se surveiller quand un paramètre est changé.



Le rattachement à une fonction de rappel (de classe `rclcpp::ParameterCallbackHandle` quand un changement est détecté sur le paramètre se fait par la méthode `add_parameter_callback`.



### EXERCICE 31 (*Ajouter une fonction de rappel*)



Modifiez le code de votre fichier `hello_world_parameter.cpp` pour lui ajouter la fonction de rappel. Compilez et relancez les deux nœuds comme à l'exercice précédent.



**Est-ce que vous voyez bien le message affiché par la fonction de *callback* lors du changement de paramètre ?**

```
[INFO] [1675850593.204666662] [hello_world_parameter_node]: Hello world!
[INFO] [1675850593.885572132] [hello_world_parameter_node]: parameter "
    target_parameter" of type string update with: "me"
[INFO] [1675850594.204690652] [hello_world_parameter_node]: Hello me!
```

## 5 Actions

Les actions sont, comme on l'a vu, basées sur une communication client-serveur mais avec en plus un retour intermédiaire sur la progression de la réalisation (une barre ou un pourcentage de progression est une bonne image d'un type de retour intermédiaire).



**L'objectif à atteindre est fixé par la requête du client et la réalisation est menée par le serveur** qui rend compte au client par des retours intermédiaires réguliers puis par la réponse finale qui contient le résultat réel atteint (ie. l'objectif fixé plus ou moins une marge d'erreur tolérée).

Comme pour la communication client-serveur, pour lancer sa propre action, il faut définir son propre type d'action, le programme serveur et le programme client. Comme exemple, on va créer une action de type minuteur qui va durer le temps précisé en requête, afficher le temps écoulé depuis le début de la requête comme résultat intermédiaire et confirmer le temps final réellement écoulé.



**Ce type de minuteur** peut être intéressant en robotique car il faut souvent attendre qu'une action (un déplacement physique par exemple) ait réellement le temps de se faire avant de passer à l'action suivante.

### 5.1 Créer son propre type d'action

Un type d'action, comme un type de message ou de service, sert d'interface entre les nœuds. C'est donc en toute logique que l'on va créer notre type d'action en suivant le même protocole que précédemment.

### EXERCICE 32 (*Complément du paquet dédié aux interfaces*)



À la section 2.3 page 12, vous avez créé le paquet `my_first_ros_interface`. Rendez-vous dans ce dernier et créez-y un nouveau répertoire `action` (au même niveau donc que les répertoires `msg` et `srv` que l'on avait créé pour les types de messages de forums et de services).



**Il faut ajouter dans le manifeste la dépendance suivante : `<depend>action_msgs</depend>`** car la définition d'une action nécessite une métadonnée (l'identifiant de l'objectif) accessible uniquement via l'utilisation de ce paquet.



**En configuration**, on ajoutera les dépendances dans le manifeste et les nouveaux types de d'action à générer (et leurs dépendances) dans la liste des cibles `rosidl_generate_interfaces` (avec les messages des forums et des services).



Tout nouveau type d'action sera placé dans le dossier `action` du paquet dédié aux interfaces.

Les types d'actions, comme les messages de forums et de services, sont des fichiers au format texte, qui ont pour extension `.action` et pour nom le nom du type d'action avec une majuscule (`RotateAbsolute.action` par exemple). Dans le TD précédent, on avait utilisé la commande suivante pour voir le contenu d'un fichier de type d'action (prenons `RotateAbsolute` comme exemple) :

```
ros2 interface show turtlesim/action/RotateAbsolute
```

Cela nous affiche les lignes suivantes :

```
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```



Sur chaque ligne du fichier, on a la description d'un champ avec son type et son nom.



--- Indique les séparations entre le format de la requête (l'objectif à atteindre, au-dessus), le format de la réponse (le résultat réellement atteint, au milieu) et le *feedback* (les retours intermédiaires, en dessous).

### EXERCICE 33 (*Nouveau type d'action Timer*)



Créez le fichier `Timer.action` qui contient les champs suivants :

```
float32 timer_target
---
float32 timer_final
---
float32 timer_elapsed
```

### EXERCICE 34 (*Compilation du type d'action Timer*)



Complétez le `CMakeLists.txt` en ajoutant la nouvelle cible (comme on le faisait jusqu'à maintenant : une cible par ligne). Nous n'utilisons pas de dépendances spécifiques dans ce type, donc pas besoin de modifications sur ce point.

Puis lancez la compilation (`colcon build`) à la racine de l'espace de travail. Si tout s'est bien passé, sourcez et utilisez la commande suivante pour visualiser votre nouveau type d'action :

```
ros2 interface show my_first_ros_interfaces/action/Timer
```



Est-ce que ça correspond bien ?



**Vous avez créé votre nouveau type d'action !**

Pour l'utiliser, commençons par le côté serveur (le nœud qui effectue l'action).

## 5.2 Recevoir et effectuer une action (serveur)

Comme on l'a vu pour les services, il est d'usage de séparer le côté interface (définition du type d'action) du côté utilisation du type d'action. Pour ce qui suit, on va donc aller dans le répertoire `src` du paquet `my_first_ros_codes`. Le programme suivant présente un serveur action dédié au minuteur :

```
#include "my_first_ros_interfaces/action/timer.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

using namespace std::placeholders;

class TimerActionServer : public rclcpp::Node
{
public:
    TimerActionServer() : Node("timer_action_server_node")
    {
        this->action_server_ = rclcpp_action::create_server<my_first_ros_interfaces::action::Timer>
            (>(this,
            "timer", std::bind(&TimerActionServer::handle_goal, this, _1, _2), std::bind(&
            TimerActionServer::handle_cancel, this, _1), std::bind(&TimerActionServer::
            handle_accepted, this, _1));
    }
private:
    rclcpp_action::Server<my_first_ros_interfaces::action::Timer>::SharedPtr action_server_;
    rclcpp_action::GoalResponse handle_goal(const rclcpp_action::GoalUUID & uuid, std::shared_ptr
        <const my_first_ros_interfaces::action::Timer::Goal> goal)
    {
        RCLCPP_INFO(this->get_logger(), "Received goal request %.2f s", goal->timer_target);
        (void)uuid;
        return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
    }

    rclcpp_action::CancelResponse handle_cancel(const std::shared_ptr<rclcpp_action::
        ServerGoalHandle<my_first_ros_interfaces::action::Timer>> goal_handle)
    {
        RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
        (void)goal_handle;
        return rclcpp_action::CancelResponse::ACCEPT;
    }

    void handle_accepted(const std::shared_ptr<rclcpp_action::ServerGoalHandle<
        my_first_ros_interfaces::action::Timer>> goal_handle)
    {
        // we need a quick return to avoid blocking the executor, so spin up a new thread
        std::thread{std::bind(&TimerActionServer::execute, this, _1), goal_handle}.detach();
    }

    void execute(const std::shared_ptr<rclcpp_action::ServerGoalHandle<my_first_ros_interfaces::
        action::Timer>> goal_handle)
    {
        RCLCPP_INFO(this->get_logger(), "Executing goal");
        float timer_step = 4.0; // timer step in hertz
        rclcpp::Rate loop_rate(timer_step);
        const auto goal = goal_handle->get_goal();
        auto feedback = std::make_shared<my_first_ros_interfaces::action::Timer::Feedback>();
        auto result = std::make_shared<my_first_ros_interfaces::action::Timer::Result>();
        float elapsed_time = 0.;
        for (int num_step = 0; (num_step/timer_step < goal->timer_target) && rclcpp::ok();
            num_step++)
        {
```

```

        elapsed_time = elapsed_time + 1./timer_step;
        // Check if there is a cancel request
        if (goal_handle->is_canceling()) {
            result->timer_final = elapsed_time;
            goal_handle->anceled(result);
            RCLCPP_INFO(this->get_logger(), "Goal canceled");
            return;
        }
        feedback->timer_elapsed = elapsed_time;
        // Publish feedback
        goal_handle->publish_feedback(feedback);
        RCLCPP_INFO(this->get_logger(), "Publish feedback");
        loop_rate.sleep();
    }
    // Check if goal is reached
    if (rclcpp::ok()) {
        result->timer_final = elapsed_time;
        goal_handle->succeed(result);
        RCLCPP_INFO(this->get_logger(), "Goal succeeded : result = %.2f s", result->timer_final);
    }
}
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Action server Timer ready.");
    rclcpp::spin(std::make_shared<TimerActionServer>());
    rclcpp::shutdown();
}

```



**Dépendances :** `#include "rclcpp_action/rclcpp_action.hpp"` permet d'avoir la classe liée aux actions qui n'est pas incluse par défaut dans `rclcpp`.



`rclcpp_action::create_server` permet de créer le serveur action. Il faut préciser le nœud auquel est rattaché le serveur, le type d'action, et les fonctions de rappel liées respectivement à la réception et la définition de l'objectif (`handle_goal`), à l'annulation de l'action en cours de réalisation de celle-ci car il y a un nouvel objectif qui rend l'actuel obsolète par exemple (`handle_cancel`) et à son acceptation et sa réalisation (`handle_accepted`).



`std::thread` permet d'éviter de bloquer le processus en créant un nouveau process qui réalise l'exécution proprement dite de l'action.



`get_goal()` permet de récupérer l'objectif, `Feedback` permet de gérer les retours intermédiaires (publiés par `publish_feedback`) et `Result` le résultat final atteint (renvoyé au client par `succeed`).

## EXERCICE 35 (Nœud serveur pour réaliser l'action timer)



Recopiez le code précédent dans un fichier `timer_action_server.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud `"timer_action_server_node"`.



**Dépendances :** on a déjà rajouté la dépendance au paquet `my_first_ros_interfaces` dans les fichiers de configuration au cours des exercices précédents. Par contre, il faut rajouter celle à `rclcpp_action` dans les deux fichiers de configuration et de préciser que ce code utilise ces deux dépendances supplémentaires via `ament_target_dependencies`.

Lancez la compilation, puis si tout s'est bien passé, exécutez votre nouveau nœud :

```
ros2 run my_first_ros_codes timer_action_server_node
```



Voyez-vous bien apparaître à l'écran une ligne qui vous dit que le serveur d'action est prêt et attend une requête ?

```
[INFO] [1676292394.062526974] [rclcpp]: Action server Timer ready.
```



Il nous faut maintenant un programme client qui sollicite notre serveur en lui demandant de réaliser une action.



Laissez tourner le serveur d'action pour le moment.

## 5.3 Émettre une requête d'action (client)

Passons maintenant au code du client qui lance la requête d'action de type Timer :

```
#include "my_first_ros_interfaces/action/timer.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
using namespace std::placeholders;

class TimerActionClient : public rclcpp::Node
{
public:
    TimerActionClient() : Node("Timer_action_client_node")
    {
        this->client_ptr_ = rclcpp_action::create_client<my_first_ros_interfaces::action::Timer>(
            this, "timer");
    }






    void send_goal(char argv[])
    {
        if (!this->client_ptr_->wait_for_action_server()) {
            RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
            rclcpp::shutdown();
        }
        auto goal_msg = my_first_ros_interfaces::action::Timer::Goal();
        goal_msg.timer_target = std::atof(argv);
        RCLCPP_INFO(this->get_logger(), "Sending goal");
        auto send_goal_options = rclcpp_action::Client<my_first_ros_interfaces::action::Timer>::
            SendGoalOptions();
        send_goal_options.goal_response_callback = std::bind(&TimerActionClient::
            goal_response_callback, this, _1);
```

```

        send_goal_options.feedback_callback = std::bind(&TimerActionClient::feedback_callback,
            this, _1, _2);
        send_goal_options.result_callback = std::bind(&TimerActionClient::result_callback, this,
            _1);
        this->client_ptr->async_send_goal(goal_msg, send_goal_options);
    }
private:
    rclcpp_action::Client<my_first_ros_interfaces::action::Timer>::SharedPtr client_ptr_;
    void goal_response_callback(const rclcpp_action::ClientGoalHandle<my_first_ros_interfaces::
        action::Timer>::SharedPtr & goal_handle)
    {
        if (!goal_handle) {
            RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
        } else {
            RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
        }
    }
    void feedback_callback(rclcpp_action::ClientGoalHandle<my_first_ros_interfaces::action::Timer
        >::SharedPtr, const std::shared_ptr<const my_first_ros_interfaces::action::Timer::
        Feedback> feedback)
    {
        RCLCPP_INFO(this->get_logger(), "Time elapsed : %.2f", feedback->timer_elapsed);
    }
    void result_callback(const rclcpp_action::ClientGoalHandle<my_first_ros_interfaces::action::
        Timer>::WrappedResult & result)
    {
        switch (result.code) {
            case rclcpp_action::ResultCode::SUCCEEDED:
                break;
            case rclcpp_action::ResultCode::ABORTED:
                RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
                return;
            case rclcpp_action::ResultCode::CANCELED:
                RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
                return;
            default:
                RCLCPP_ERROR(this->get_logger(), "Unknown result code");
                return;
        }
        RCLCPP_INFO(this->get_logger(), "Success : final time elapsed %.2f", result.result->
            timer_final);
        rclcpp::shutdown();
    }
};

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    if (argc != 2) {
        RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: timer_action_client_node X (seconds)");
        rclcpp::shutdown();
    }
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Action client Timer ready.");
    auto node = std::make_shared<TimerActionClient>();
    node->send_goal(argv[1]);
    rclcpp::spin(node);
    rclcpp::shutdown();
}

```

-  `rclcpp_action::create_client` permet de créer le client action. Il faut préciser le nœud auquel est rattaché le client et le type d'action.
-  Une fois créé (dans le `main`), l'envoi du type d'action est fait par un appel explicite à la méthode publique `send_goal` (vous êtes libre de lui donner le nom que vous voulez). Dans cet exemple, la durée du timer est passée en argument lors du lancement du nœud.
-  `wait_for_action_server()` permet de s'assurer que le serveur est bien disponible, comme on l'a vu pour les services.
-  `Goal` et `SendGoalOptions` permettent de gérer l'envoi de la requête et les fonctions de rappel à activer pour l'envoi, le *feedback* et le succès.
-  Contrairement au cas du client pour le service, il est impératif de lancer le nœud par un `spin` après l'envoi de la requête pour traiter son retour par le serveur, que ce soit pour le *feedback* ou le résultat final.




### EXERCICE 36 (Nœud client d'action pour lancer un timer)



Recopiez le code précédent dans un fichier `timer_action_client.cpp` que vous mettrez dans le dossier `src` de votre paquet `my_first_ros_codes`. Modifiez les fichiers de configuration pour pouvoir compiler votre code depuis l'espace de travail `my_ros2_ws` (reprendre les exercices 4 page 4, 5 page 4 et 6 page 5) : on appellera l'exécutable de ce nouveau nœud `"timer_action_client_node"`.

Lancez la compilation, puis si tout s'est bien passé, exécutez votre nouveau nœud dans un nouveau terminal (n'oubliez pas de sourcer) :

```
ros2 run my_first_ros_codes timer_action_client_node 3
```

-  N'oubliez pas que votre nœud serveur d'action doit être en train de tourner avant de lancer le nœud client d'action.
-  N'oubliez pas l'entier en argument de votre ligne de commande (le temps du timer) !
-  N'hésitez pas à tester avec d'autres entiers !

 Côté serveur, vous devez obtenir un résultat comparable à :

```
[INFO] [1676292401.121943585] [timer_action_server_node]: Received goal request 3.00 s
[INFO] [1676292401.122089726] [timer_action_server_node]: Executing goal
[INFO] [1676292401.122153722] [timer_action_server_node]: Publish feedback
[INFO] [1676292401.372416755] [timer_action_server_node]: Publish feedback
[INFO] [1676292401.622420419] [timer_action_server_node]: Publish feedback
[INFO] [1676292401.872348186] [timer_action_server_node]: Publish feedback
[INFO] [1676292402.122440626] [timer_action_server_node]: Publish feedback
[INFO] [1676292402.372357739] [timer_action_server_node]: Publish feedback
[INFO] [1676292402.622532076] [timer_action_server_node]: Publish feedback
[INFO] [1676292402.872375718] [timer_action_server_node]: Publish feedback
[INFO] [1676292403.122408804] [timer_action_server_node]: Publish feedback
[INFO] [1676292403.372401007] [timer_action_server_node]: Publish feedback
[INFO] [1676292403.622539996] [timer_action_server_node]: Publish feedback
[INFO] [1676292403.872371918] [timer_action_server_node]: Publish feedback
[INFO] [1676292404.122479721] [timer_action_server_node]: Goal succeeded : result = 3.00 s
```



Et côté client, un résultat comparable à :

```
[INFO] [1676292401.110137991] [rclcpp]: Action client Timer ready.
[INFO] [1676292401.121794065] [Timer_action_client_node]: Sending goal
[INFO] [1676292401.122063181] [Timer_action_client_node]: Goal accepted by server, waiting for
result
[INFO] [1676292401.122180465] [Timer_action_client_node]: Time elapsed : 0.00
[INFO] [1676292401.372697679] [Timer_action_client_node]: Time elapsed : 0.25
[INFO] [1676292401.622730986] [Timer_action_client_node]: Time elapsed : 0.50
[INFO] [1676292401.872650831] [Timer_action_client_node]: Time elapsed : 0.75
[INFO] [1676292402.122777973] [Timer_action_client_node]: Time elapsed : 1.00
[INFO] [1676292402.372688849] [Timer_action_client_node]: Time elapsed : 1.25
[INFO] [1676292402.622848783] [Timer_action_client_node]: Time elapsed : 1.50
[INFO] [1676292402.872665271] [Timer_action_client_node]: Time elapsed : 1.75
[INFO] [1676292403.122699402] [Timer_action_client_node]: Time elapsed : 2.00
[INFO] [1676292403.372705229] [Timer_action_client_node]: Time elapsed : 2.25
[INFO] [1676292403.622853539] [Timer_action_client_node]: Time elapsed : 2.50
[INFO] [1676292403.872637126] [Timer_action_client_node]: Time elapsed : 2.75
[INFO] [1676292404.122738794] [Timer_action_client_node]: Success : final time elapsed 3.00
```



### Fin du deuxième TD

Nous avons découvert dans ce TD comment coder les briques de bases d'un programme ROS. Dans le TD suivant, nous allons nous intéresser à des actions un peu plus avancées.