ROS2: Outils opérationnels

Fichiers de lancement

Laurent Beaudoin & Loïca Avanthey
Epita

Avant propos

Au cours de cette séance, nous allons voir comment automatiser le lancement de plusieurs nœuds composant un programme ROS sans passer par une multitude de terminaux.

1 Lancement automatique d'une séquence de nœuds

1.1 ROS2 Launch

Jusqu'à maintenant, nous avons ouvert **une console** pour chaque lancement de **nœud** via l'appel à la commande `ros2 run`.

Ce mode de fonctionnement, bien qu'utile d'un point de vue pédagogique pour comprendre la notion de nœud, n'est pas vraiment **opérationnel** car la plupart des projets sont composés de nombreux nœuds.

Pour éviter d'avoir à recourir à autant de terminaux qu'il y a de nœuds, ou tout simplement pour **automatiser une séquence de lancement** et maîtriser quoi faire lors de changements d'état d'un nœud (lancé, arrêté, mis en pause, évènement particulier, etc.), ROS dispose d'une procédure dédiée :

- toute la séquence de lancement est décrite dans un **fichier** (*launchfile*) qui a pour extension `.launch` et qui se situe dans le répertoire `launch` à la racine du paquet.

Ce fichier de lancement contient la liste des nœuds à lancer avec les options pour chacun d'eux.

- depuis ROS2, un fichier de lancement peut être écrit en `python`, en `YAML` ou `XML`. Dans le cadre de ce TD, on restera sur du `XML`,
- le lancement de la séquence se fait via la commande `ros2 launch` (ajouter l'option `-v` pour la version bavarde):

```
ros2 launch package_name launch_file_name
```

1.2 Création d'un fichier de lancement basique

La pratique valant mieux que de longs discours, on va compléter notre espace de travail `my_ros2_ws` du TD précédent par toute une série d'exemple de fichiers de lancement.

EXERCICE 1 (*Launch repertory*)

Commençons par créer le répertoire `launch` dans le paquet `my_first_ros_codes`.

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

Ce répertoire contiendra tous les fichiers de lancement que nous allons créer dans la suite de ce TD.

Comme précédemment, on a mis l'espace de travail par défaut à la racine du compte, à vous d'adapter.

EXERCICE 2 (*Informations de compilation*)

Pour que les fichiers de lancement contenus dans le répertoire `launch` soient bien trouvés lors de la compilation, il faut compléter le fichier `CMakeLists.txt` du paquet juste avant le `ament_package()` final par :

```
#Install launch files
install(DIRECTORY
   launch
   DESTINATION share/${PROJECT_NAME}/
)
```

Voyons un premier exemple qui automatise le lancement de notre simulateur `turtlesim` préféré avec le déplacement de la tortue piloté par le nœud `turtle_random_node` de notre paquet `my_first_ros_codes`.

EXERCICE 3 (*First launchfile – XML*)

Créez le fichier de lancement `turtlesim_random_launch.xml` dans le répertoire `launch` et recopiez-y le code suivant :

```
<launch>
  <node pkg="turtlesim" exec="turtlesim_node"/>
  <node pkg="my_first_ros_codes" exec="turtle_random_node"/>
</launch>
```

Le fichier de lancement est un fichier `XML` et donc architecturé en balises ouvrantes `<>` et fermantes `</>`

La balise principale est la balise `launch` qui contient toutes les autres.

la balise `node` correspond à chaque nœud à lancer et a pour champs :

- `pkg="package_name"` (obligatoire) qui est le nom du paquet à qui appartient le nœud.
- `exec="executable_name"` (obligatoire) qui est le nom de l'exécutable (nom du nœud) à lancer.

EXERCICE 4 (*Compile and Launch Turtle random mode !*)

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

```
cd ~/my_ros2_ws/
```

Et on lance l'outils de compilation `colcon` :

```
colcon build
```

Ensuite, retournez à la racine de votre paquet et utilisez votre nouveau *launchfile* pour que le simulateur `turtlesim_node` du paquet `turtlesim` soit lancé et que les déplacements soient générés par le nœud `turtle_random_mode` du paquet `my_first_ros_codes`.

```
ros2 launch my_first_ros_codes turtlesim_random_launch.xml
```

Nouveau terminal ? N'oubliez pas de sourcer votre espace de travail :

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

EXERCICE 5 (*First launchfile – Python*)

On va créer maintenant le même fichier de lancement, mais cette fois-ci écrit en python. Créez le fichier de lancement `turtlesim_random_launch.py` dans le répertoire `launch` et recopiez-y le code suivant :

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    ld = LaunchDescription()
    my_turtlesim_node = Node(
        package="turtlesim",
        executable="turtlesim_node"
    )
   my_turtle_random_node = Node(
        package="my_first_ros_codes",
        executable="turtle_random_node"
    )

    ld.add_action(my_turtlesim_node)
    ld.add_action(my_turtle_random_node)

    return ld
```

Sous ce format, on donne un **nom** à nos nœuds et on retrouve la **description** des deux éléments obligatoires : le **nom du paquet** du noeud et le nom de son **exécutable**. Puis on **liste** le nom des nœuds à lancer dans la description et on renvoie cette description.

EXERCICE 6 (*Compile and Launch Turtle random node ! – Python's way*)

Depuis la racine de l'espace de travail, recompiler avec `colcon` et lancez cette fois-ci le fichier de lancement écrit en python.

```
colcon build
```

```
ros2 launch my_first_ros_codes turtlesim_random_launch.py
```

Vous devriez obtenir le même résultat !

1.3 Options de nœud basiques

Il existe de nombreuses balises optionnelles qui peuvent très utilement compléter la balise `node`, on trouve par exemple les options suivantes :

- `respawn="true"` qui permet de redémarrer le nœud en cas d'arrêt
- `launch-prefix="xterm -e"` permet de lancer un nœud dans un nouveau terminal
- `name="node_name"` permet de changer le nom du nœud en écrasant le nom fixé dans le code par `rclcpp::init`. Cela peut être utile pour personnaliser des nœuds lors d'un débug par exemple.

Voyons des cas d'usage par la pratique.

EXERCICE 7 (*Turtlesim original – XML*)

Dans le répertoire `launch`, créez le fichier `turtlesim_original_launch.xml` qui lance les nœuds `turtlesim_node` et `turtle_teleop_key` du paquet `turtlesim`. Pour fonctionner, le nœud `turtle_teleop_key` devra être lancé dans un nouveau terminal et être redémarré s'il est arrêté, aidez-vous des options vues ci-dessus pour écrire votre fichier de lancement.

Puis à la racine de l'espace de travail, compilez avec `colcon` et lancez votre nouveau fichier de lancement avec `ros2 launch`.

Ça fonctionne ? Normalement vous devriez voir apparaître le simulateur avec la tortue et un terminal qui prend les ordres envoyés par le clavier et les envoie à la tortue.

Fermez maintenant le terminal qui prend les ordres clavier : il devrait ré-apparaître instantanément si vous avez mis les bons paramètres.

Vous pouvez vérifier que le terminal a bien été fermé puis relancé dans votre terminal depuis lequel vous avez lancé le *launchfile* :

```
[INFO] [xterm-2]: process has finished cleanly [pid XXXXX]
[INFO] [xterm-2]: process started with pid [YYYYY]
```

Vous pouvez également ajoutez le paramètre `respawn_delay="3"` qui permettra d'attendre 3 secondes avant de relancer le nœud pour mieux voir le terminal qui se relance.

Vous ne vous en sortez pas ? Normalement, par rapport à votre premier *launchfile*, il vous suffit de remplacer la ligne du deuxième nœud par celle-ci :

```
<node pkg="turtlesim" exec="turtle_teleop_key" launch-prefix="xterm -e"
    respawn="true"/>
```

EXERCICE 8 (*Turtlesim original – Python*)

Testez maintenant le code Python `turtlesim_original_launch.py` qui fait la même chose que le code XML de l'exercice précédent :

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    ld = LaunchDescription()

   my_turtlesim_node = Node(
      package="turtlesim",
      executable="turtlesim_node"
   )
   my_turtle_teleop_key_node = Node(
      package="turtlesim",
      executable="turtle_teleop_key",
      prefix=["xterm -e"],
      respawn=True,
      respawn_delay=3
   )

    ld.add_action(my_turtlesim_node)
    ld.add_action(my_turtle_teleop_key_node)

   return ld
```

Est-ce que vous obtenez bien le même comportement ?

EXERCICE 9 (*Another name for my turtle*)

Modifiez les deux codes de lancement précédents pour donner un nom spécifique à votre nœud `turtlesim_node`, par exemple `GalacticTurtle`. Compilez, lancez les *launchfiles* et vérifiez que le bon nom s'affiche.

EXERCICE 10 (*Ping pong check !*)

Écrire en XML puis en Python les fichiers de lancement `pingpong_launch.xml` et `pingpong_launch.py` qui lance les nœuds `subscribe_node` et `publish_node` du paquet `my_first_ros_codes`. Pour visualiser les messages, on lancera `subscribe_node` dans un nouveau terminal. Vous pouvez donner des noms spécifiques à vos nœuds.

Après avoir compilé et lancé chacun des *launchfiles* est-ce que vous avez bien un terminal qui s'ouvre du nœud qui écoute sur le forum et qui affiche ce qu'il lit et répond à chaque fois et les bons noms associés ?

1.4 Options de nœud avancées

Il existe des balises optionnelles plus avancées pour compléter la balise `node` :

- `namespace="namespace"` permet de rattacher un nœud a un espace particulier, ce qui permet entre autre de lancer plusieurs fois le même exécutable en parallèle sans conflit puisque chacun appartient à des espaces différents

- `remap from="originalTopicName" to="newTopicName"` permet de changer le nom d'un forum sans avoir à toucher au code source du nœud concerné.

1.4.1 Namespace

Pour certaines applications, il peut être nécessaire de lancer plusieurs fois un même nœud d'un même paquet lorsque vous avez besoin d'exécuter une même tâche mais sur des objets différents. Pour qu'il n'y ait aucune confusion possible, il est indispensable que chaque nœud appartienne à un espace d'exécution différent. Dans le cas contraire, le comportement du programme n'est pas garanti.

 C'est une manière polie de dire que Murphy ne va pas vous louper au pire moment.

EXERCICE 11 (*Not One but Two TurtleSims! – XML*)

 Prenons le cas d'exemple où l'on souhaite lancer deux nœuds `turtlesim_node` en parallèle.

 **Pour que ces deux nœuds n'entrent pas en conflit,** il faut qu'ils appartiennent à deux espaces différents.

Pour cela, il suffit de préciser leur espace respectif à leur lancement via l'option `namespace` dans le fichier de lancement comme dans l'exemple suivant:

```
<launch>
  <node pkg="turtlesim" exec="turtlesim_node" namespace="turtlesim1"/>
  <node pkg="turtlesim" exec="turtle_teleop_key" launch-prefix="xterm -e" namespace=
      "turtlesim1"/>
  <node pkg="turtlesim" exec="turtlesim_node" namespace="turtlesim2"/>
  <node pkg="turtlesim" exec="turtle_teleop_key" launch-prefix="xterm -e" namespace=
      "turtlesim2"/>
</launch>
```

Recopiez ce code dans un fichier de lancement nommé `turtlesim_2turtles_launch.xml`. Compilez et lancez-le.

Dans un second terminal, lancez la commande suivante pour vérifier que les deux espaces sont bien séparés et qu'il n'y a pas risque de confusion en ce qui concerne les communications :

```
ros2 node list
```

 **Est-ce que ça fonctionne ?** Si oui, vous devriez obtenir les lignes suivantes :

```
/turtlesim1/teleop_turtle
/turtlesim1/turtlesim
/turtlesim2/teleop_turtle
/turtlesim2/turtlesim
```

EXERCICE 12 (*Not One but Two TurtleSims! – Python*)

 À vous de jouer pour la version python du *launchfile* `turtlesim_2turtles_launch.py` décrit à l'exercice précédent : l'option a le même nom. Compilez, lancez et testez avec :

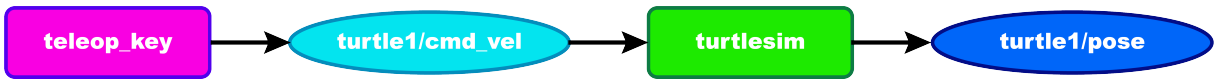
```
ros2 node list
```

 **Tout fonctionne après vérification ?**

1.4.2 Remap

Un cas d'usage courant de l'option `remap` est de pouvoir redéfinir les entrées et sorties d'un nœud.

À titre d'exemple, on va complexifier l'exemple précédent à 2 tortues en faisant en sorte qu'un seul des nœud `teleop_key` contrôle simultanément les tortues des nœuds `turtlesim` définis dans les espaces `turtlesim1` et `turtlesim2`. Posons le problème. Rappelons-nous le fonctionnement (liens entre les nœuds) de `turtlesim` avec une tortue :

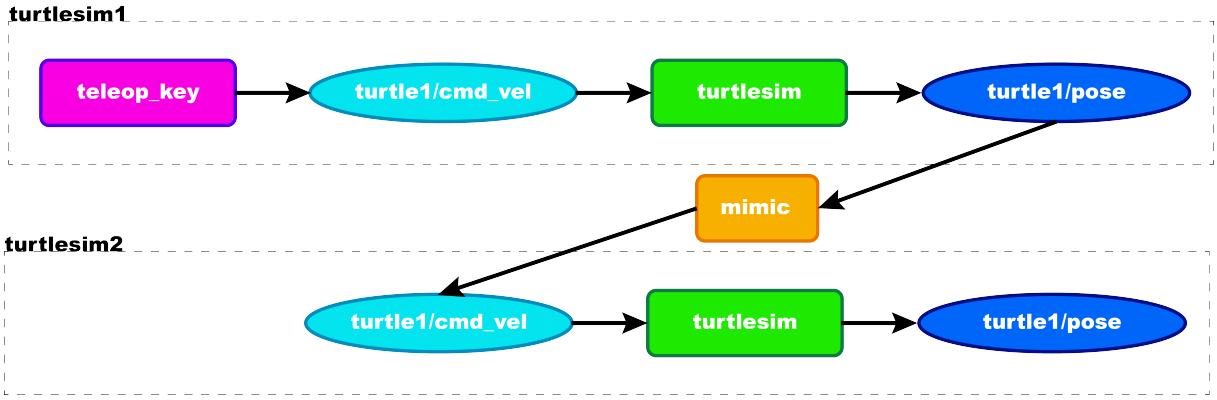


Dans notre cas, ce que l'on souhaite faire, c'est :

- rajouter le nœud `teleop_key` à la tortue de l'espace `turtlesim1`
- faire que la sortie de `turtlesim` de l'espace `turtlesim1` (`turtle1/pose`) devienne l'entrée de de `turtlesim` de l'espace `turtlesim2` (`turtle1/cmd_vel`)

Le problème c'est que les messages publiés sur les forums `turtle1/pose` et `turtle1/cmd_vel` ne sont pas identiques.

Il faut donc construire l'un à partir des données de l'autre. Et c'est ce que fait le nœud `mimic` du paquet `turlesim` ! Ce nœud lit ses données d'entrée sur un forum intitulé `/input/pose` et met les données converties sur le forum `/output/cmd_vel`. Il suffit donc de transformer ces deux noms de forums génériques avec les noms des forums qui nous intéressent dans notre cas grâce à l'utilisation de l'option `remap` ! Voici ce que devient notre schéma si on veut piloter deux tortues simultanément :



EXERCICE 13 (*Piloter deux tortues simultanément*)

Écrivez les fichiers de lancement en XML (`turtlesim_mimic_launch.xml`) puis en Python (`turtlesim_mimic_launch.py`) qui permettent de commander 2 tortues simultanément en suivant le schéma fourni.

Pour information, voici comment s'intègre la balise `remap` en XML :

```
<node pkg="pakage_name" exec="executable_name">
        <remap from="original_topic_name" to="new_topic_name"/>
</node>
```

Si on veut renommer plusieurs forums pour un même nœud, il suffit de mettre plusieurs balises remap à la suite.

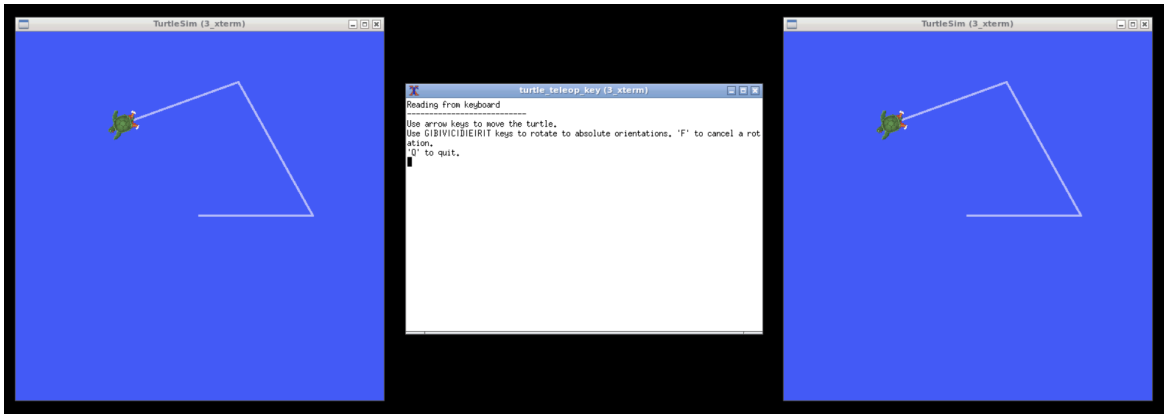
Pour rappel, les noms génériques originaux des forums du nœud mimic sont `"/input/pose"` et `"/output/cmd_vel"`.

Et en Python, voici comment intégrer l'option `remap` à votre code :

```
my_node = Node(
    package="pakage_name",
    executable="executable_name",
    remappings=[
        ("original_topic_name", "new_topic_name")
    ]
)
```

Si vous devez changer plusieurs noms de forums en Python, il vous suffit de les lister les uns à la suite des autres comme cela :

```
    remappings=[
        ("originalTopicName1", "newTopicName1"),
        ("originalTopicName2", "newTopicName2"),
    ]
```



1.5 Paramètres

1.5.1 Paramètres en dur dans un fichier de lancement

La balise `param` permet de définir la valeur d'un paramètre d'un nœud. La syntaxe suit les modèles suivants :

En XML :

```
<param name="param_name" value="value"/>
```

En Python :

```
parameters=[{
   "param_name": value
}]
```

EXERCICE 14 (*Tortues de mer et de sable*)

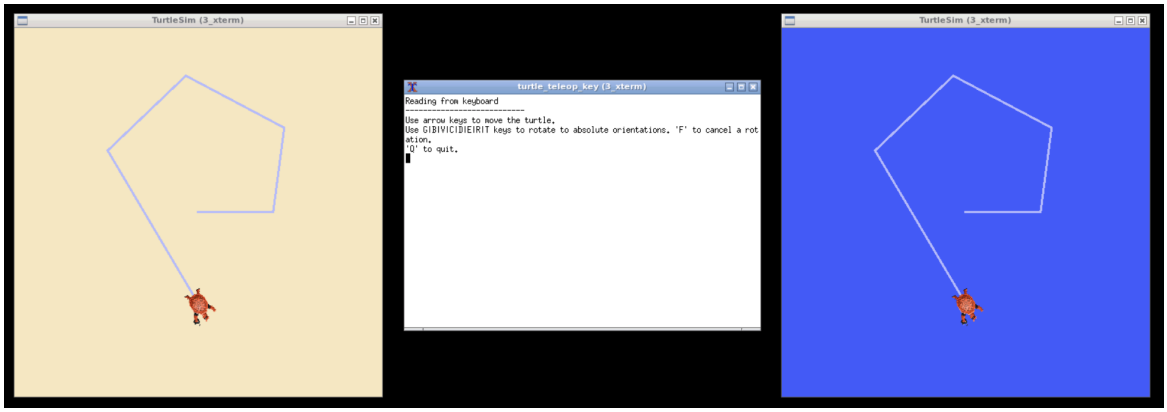
Recopiez le fichier de lancement précédent dans un fichier (`turtlesim_mimic_sand_launch.xml`) et modifiez le code pour que la seconde tortue évolue sur du sable.

Pour rappel, les paramètres qui contrôlent la couleur du fond d'écran du simulateur sont `background_r` (rouge), `background_g` (vert) et `background_b` (bleu)

On a vu dans le précédent TD que la couleur sable s'obtenait en RGB avec les valeurs suivantes : (242, 231, 191).

Pour fixer les valeurs de plusieurs paramètres, on répète plusieurs fois la balise `param` en XML ou on les liste à la suite en Python

Compilez, lancez et vérifiez que les paramètres ont bien été pris en compte. Si tout fonctionne, écrivez et testez une version Python de ce *launchfile* (`turtlesim_mimic_sand_launch.py`).



1.5.2 Paramètres en variables dans un fichier de lancement

Pour définir une variable dans un fichier de lancement en XML, il faut utiliser la balise `arg` avec la syntaxe suivante :

```
<arg name="argument_name" default="default_value"/>
```

`default` est la valeur prise par défaut par la variable.

Vous pouvez changer cette valeur lors du lancement de votre *launchfile* en ajoutant à la fin de la ligne de commande : `argument_name:="new_value"`.

Ensuite, pour récupérer la valeur de l'argument passé en ligne de commande, il faut interpréter la variable en utilisant la commande suivante :

```
$(var argument_name)
```

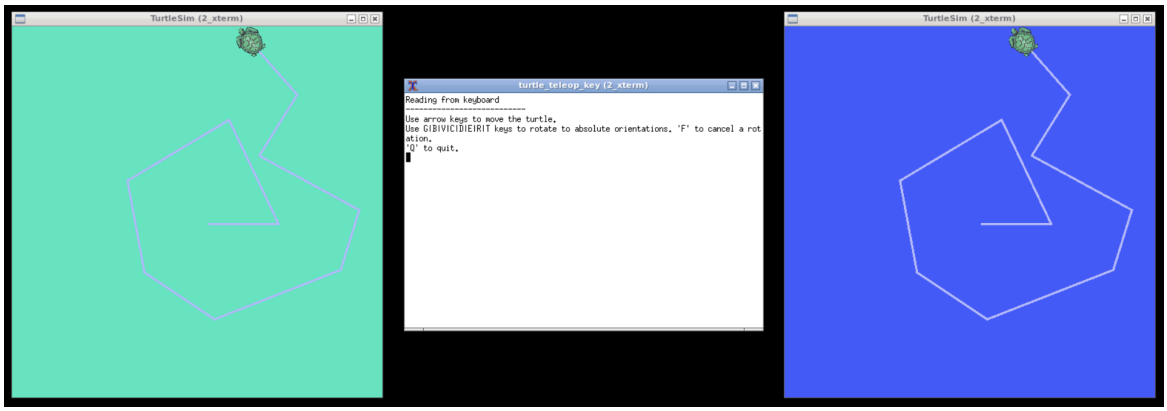
Voici un modèle d'utilisation de ces balises :

```
<launch>
        <arg name="argument_name" default="default_value"/>
        <node pkg="pakage_name" exec="executable_name">
                <param name="param_name" value="$(var argument_name)"/>
        </node>
</launch>
```

EXERCICE 15 (*Tortues de mers d'ici ou d'ailleurs – XML*)

Recopiez votre *launchfile* en XML de l'exercice 14 dans un fichier `turtlesim_mimic_arg_launch.xml`. Modifiez-le en utilisant les variables `red`, `green` et `blue` pour définir la couleur de la fenêtre du simulateur.

Compilez et testez votre nouveau *launchfile* en le lançant en passant en argument 0 pour la couleur rouge afin de faire passer votre tortue des sables sous les tropiques.



Pour Python, la syntaxe est assez similaire. Un argument se déclare de cette manière :

```
my_argument_name = DeclareLaunchArgument(
    "argument_name", default_value="default_value"
)
```

N'oubliez pas ensuite de les ajouter dans la description du lancement !

Et il s'utilise ainsi :

```
my_node = Node(
    package="package_name",
    executable="executable_name",
    parameters=[{
        "argument_name":LaunchConfiguration("argument_name"),
    }]
)
```

Il faudra ajouter les lignes suivantes pour accéder aux nouvelles fonctionnalités :

```
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
```

EXERCICE 16 (*Tortues de mers d'ici ou d'ailleurs – Python*)

Comme précédemment, recopiez votre *launchfile* en Python de l'exercice 14 dans un fichier `turtlesim_mimic_arg_launch.py`. Modifiez-le en utilisant les variables `red`, `green` et `blue` pour définir la couleur de la fenêtre du simulateur.

Compilez et testez votre nouveau *launchfile* en lui passant en argument 0 pour la variable `red`.

Alors ? Votre tortue python est-elle bien elle aussi passée sous les tropiques ?

1.6 Gestion d'événements

La commande `ros2 launch` est chargée de surveiller l'état des processus qu'elle a lancés. Elle peut donc réagir à des changements d'état de ces processus qu'on appelle des événements. Ainsi par exemple, on peut vouloir lancer un nœud que lorsqu'un autre nœud a terminé son exécution. On peut aussi conditioner des actions par rapport à des messages d'entrées / sorties. Python est plus flexible que XML et nous permet d'utiliser facilement ces fonctionnalités dans la description du fichier de lancement.

Voici un exemple générique d'utilisation pour lancer un nœud au moment où un autre termine son exécution :

```
waiting_node_1 = RegisterEventHandler(
   event_handler=OnExecutionComplete(
      target_action=my_node_name_1,
        on_completion=[my_node_name_2],
   )
)
```

Parmi les types de gestion d'événements, on retrouve : `OnProcessStart`, `OnProcessIO`, `OnExecutionComplete`, `OnProcessExit` ou `OnShutdown`

Il ne faut pas oublier d'ajouter `waiting_node_1` à la description du lancement avant d'y ajouter `node_1`.

Il faudra ajouter les lignes suivantes pour accéder aux nouvelles fonctionnalités :

```
from launch.actions import RegisterEventHandler
from launch.event_handlers import OnExecutionComplete
```

EXERCICE 17 (*Two Turtles one after the other – Python*)

Recopiez votre *launchfile* en Python de l'exercice 11 dans un fichier `turtlesim_2turtles_event_launch.py`. Retirez les nœuds de pilotage pour simplifier le tout, puis ajouter un gestionnaire d'événement qui permet de lancer la deuxième tortue uniquement quand le processus de la première se termine.

Compilez et lancez votre nouveau *launchfile*.

Alors ? Vous avez bien qu'une seule fenêtre qui s'affiche ? Dans un autre terminal, tappez la commande `ros2 node list` pour vérifier qu'il s'agit bien de la première :

```
user:~$ ros2 node list
/turtlesim1/turtlesim
```

Fermez la fenêtre de la tortue : une autre doit se rouvrir immédiatement. Relancez la commande `ros2 node list` dans votre second terminal pour vérifier maintenant qu'il s'agit bien de la deuxième tortue :

```
user:~$ ros2 node list
/turtlesim2/turtlesim
```

1.7 Outils pour structurer les *launchfiles* des gros projets

1.7.1 Imbrication de *launchfiles*

La balise `include` permet d'inclure un fichier de lancement dans un autre, ce qui est particulièrement utile dans un gros projet. La syntaxe en XML est :

```
<include file="$(find-pkg-share include_package_name)/launch_directory_name/launchfile_name"/>
```

Et en Python (sans oublier de l'ajouter à la description du lancement) :

```
my_launchfile_1 = IncludeLaunchDescription(
   PythonLaunchDescriptionSource([FindPackageShare("launch_package_name"),
      "/launch_directory_name",
      "/launchfile_name.py"])
)
```

Avec les lignes suivantes pour accéder aux fonctionnalités :

```
from launch.actions import IncludeLaunchDescription
from launch_ros.substitutions import FindPackageShare
from launch.launch_description_sources import PythonLaunchDescriptionSource
```

EXERCICE 18 (*Two Turtle in different worlds – XML*)

 Reprenez votre *launchfile* en XML de l'exercice 11 et mettez les éléments de la première tortue dans un fichier de lancement à part entière (`turtlesim_2turtles_include1_launch.xml`), puis dans le fichier de lancement de la seconde tortue (`turtlesim_2turtles_include2_launch.xml`) incluez le fichier de la première. Compilez et testez votre fichier.

EXERCICE 19 (*Two Turtle in different worlds – Python*)

 Refaites le même exercice mais en Python, avec vos deux fichiers de lancement `turtlesim_2turtles_include1_launch.py` et `turtlesim_2turtles_include2_launch.py`. Compilez et testez votre fichier.

1.7.2 Groupement d'actions pour différents éléments

La balise `group` permet de regrouper des actions dans un fichier de lancement, comme des accolades dans un programme. Dans l'exemple suivant en XML, on définit par exemple une seule fois le nom de l'espace de travail pour tous les nœuds qui sont compris dans la balise `group` et pas à chaque nœud comme on l'a fait à la section 1.4.1.

```
<group>
      <push-ros-namespace namespace="namespace_name"/>
      <node pkg="package_name" exec="executable_name1"/>
      <node pkg="package_name" exec="executable_name2"/>
</group>
```

La même chose en Python :

```
my_group_action = GroupAction(
   actions=[
      PushRosNamespace("namespace_name"),
      my_node_1,
      my_node_2
   ]
)
```

Avec les lignes suivantes pour accéder aux fonctionnalités :

```
from launch.actions import GroupAction
from launch_ros.actions import PushRosNamespace
```

Attention, comme les actions sont groupées, il suffit d'ajouter le groupe à la description à la fin du fichier de lancement.

EXERCICE 20 (*Two turtles with groups – XML*)

Reprenez votre *launchfile* en XML de l'exercice 11, renommez-le `turtlesim_2turtles_group_launch.xml` et modifiez-le pour ne préciser qu'une fois chaque *namespace* en groupant les nœuds correspondants. Compilez et testez votre fichier.

EXERCICE 21 (*Two turtles with groups – Python*)

Même exercice, mais en Python (`turtlesim_2turtles_group_launch.py`). Compilez et testez votre fichier.

Fin du troisième TD

Nous avons découvert dans ce TD comment coder écrire des fichiers de lancement automatique de nos nœuds. Dans le TD suivant, nous allons nous intéresser à la simulation de nos robots.