SEMINAR

ROS2: lignes de commande

Découverte des nœuds et de leurs interactions

Laurent Beaudoin & Loïca Avanthey
Epita

Avant propos

Au cours de cette séance, nous allons nous découvrir les principes fondateurs de ROS2 (Robot Operating System 2) (les noeuds, les forums, les services, les actions et les paramètres) et comment interagir avec ROS2 en lignes de commande (Command Line Interface – CLI).

1 Vous avez dit ROS ?

ROS, pour *Robot Operating System*, est un **environnement de développement logiciel** (*middleware*) dédié au prototypage rapide, fiable et performant, pour des applications robotiques.

Il est composé d'un ensemble de **bibliothèques** et d'**outils** qui permettent de gérer en haut niveau les couches matérielles (le matériel est géré comme une abstraction), tout en permettant le contrôle bas niveau de nombreux composants (du moins s'ils ne sont pas trop exotiques). Comme OpenCV, ROS a connu un **développement remarquable** ces dernières années et s'impose aujourd'hui comme **l'un des principaux middlewares en robotique**. De très nombreux contributeurs proposent des **solutions ROS** sur des **domaines** très **variés** : de la navigation automatique en environnment complexe à l'asservissement visuel, en passant par le contrôle-commande de bras manipulateurs par exemple.

Un bref historique

ROS est né en 2007 au sein du *Stanford Artificial Intelligence Laboratory*, puis a été développé entre 2008 et 2013 par le laboratoire de recherche ***Willow Garage***, de Scott Hassan (Google), ce qui explique pourquoi ROS a une bonne compatibilité avec OPENCV (aussi portée à l'époque par *Willow Garage*). Depuis 2013, c'est ***Open Robotics*** (anciennement ***Open Source Robotics Fundation*** qui supporte ROS. Le *middleware* est distribué sous la licence BSD, ce qui laisse de bonnes perspectives de développement, quelles soient libres, privées ou commerciales.

ROS vs ROS2 ?

ROS est la **version historique**. Elle est testée et garantie pour Ubuntu et supportée par la communauté pour d'autres versions de Linux et Mac-OS. Au fil des années, l'équipe de développement a identifié une liste d'**améliorations** et de **fonctionnalités manquantes** importantes pour être compatible avec les **applications industrielles**. Cela comprend notamment le temps réel, la sûreté d'utilisation, la certification et la cybersécurité. Mais l'ajout de ces modifications aurait nécessité des **changements en profondeur** et aurait rendu ROS assez **instable**. Ainsi, **ROS2** a été développé à partir de zéro et a

pour vocation à **remplacer ROS** (qui ne sera plus supporté à partir de 2025). Ce cours utilisera donc ROS2.

Python vs C++ ?

Le python et le C++ sont les **principaux langages** de programmation supportés par ROS. Le python présente l'avantage de **développer rapidement** des programmes grâce à ses **abstractions** de haut niveau. C'est un bon langage **pédagogique**, ce qui explique pourquoi la plupart des livres ou tutoriaux sont en Python. Le C++ lui est plus proche du bas niveau : le temps de développement (la longueur des codes) est donc plus long. Mais le C++ offre un **gain en performance** de l'ordre de 10 à 100 par rapport au Python. Il offre ainsi plus d'intérêt pour le **monde professionnel**. Dans ce cours, on va donc programmer en C++.

Documentation

Il existe de nombreuses ressources pour apprivoiser ROS ou trouver de la documentation. Les ressources en ligne de référence sont :

- `http://www.ros.org` pour les distributions officielles
- `http://docs.ros.org` pour les tutoriaux et la documentation officielle

Et bien sûr, il y a aussi les **livres** ! Voici quelques références qui pourraient vous être utiles :

- ROS Robotics by example, C. Fairchild, T.L. Harman, Packt publishing, 2017.
- Learning ROS for Robotics Programming, A. Martinez, E. Fernadez, Pakt publishing, 2013.
- A gentle introduction to ROS, J. O'Kane, 2014.

2 Installation et paramètres

Pour ce qui suit, on supposera être sur une **distribution Linux**.

EXERCICE 1 (*Installation*)

Installez la dernière version stable des paquets binaires de ROS à partir des sites cités en ressources (https://www.ros.org/blog/getting-started/). Pour la suite de ce cours, on suppose que vous avez fait une installation sous une version native d'Ubuntu (pas en machine virtuelle).

Pour fonctionner, il est nécessaire que des **variables d'environnement** propres à ROS soient correctement **initialisées**.

Pour cela, il faut lancer un **script** dans chaque nouveau terminal utilisant ROS. Afin de se simplifier la vie, on va rendre cette étape **automatique** en l'inscrivant dans le fichier `.bashrc` (script qui est exécuté chaque fois qu'un utilisateur ouvre un nouveau terminal).

EXERCICE 2 (*Variables d'environnement*)

Tappez la ligne suivante dans votre terminal :

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

 Si votre distribution ROS n'est pas `humble`, pensez à adapter la commande !

Enfin, il vous faut configurer un paramètre (identifiant de domaine) pour que tous les programmes ROS que l'on lance puisse communiquer entre eux.

 Cet identifiant est un nombre qui doit être compris entre 0 et 100.

Pour l'instant, fixons ce paramètre à 0. Comme précédemment, on va rendre cette étape automatique en l'ajoutant une fois pour toute à notre fichier `.bashrc`.

EXERCICE 3 (*Identifiant de domaine ROS*)

 Tappez la ligne suivante dans votre terminal :

```
echo "export ROS_DOMAIN_ID=0" >> ~/.bashrc
```

EXERCICE 4 (*Vérification*)

 Pour vérifier si les variables d'environnement spécifiques à ROS ont bien été correctement initialisées, lancez la commande :

```
printenv | grep -i ROS
```

Et vérifiez que vous avez un résultat comparable à :

```
ROS_VERSION=2
ROS_PYTHON_VERSION=3
ROS_DOMAIN_ID=0
ROS_LOCALHOST_ONLY=0
ROS_DISTRO=humble
```

Installation terminée !

Si tout s'est bien passé, on va pouvoir commencer à découvrir ROS. Pour cela, on utilisera l'interface en ligne de commande (CLI) et des robots simulés.

3 Découverte des nœuds et de l'architecture ROS

3.1 Un robot-tortue simulé

Lançons pour commencer un programme de simulation appelé *turtlesim* et directement inspiré de ce qui se faisait il y a longtemps avec la tortue en langage Pascal.

EXERCICE 5 (*Premier contact avec Turtlesim*)

 Dans un premier terminal, lancez la commande suivante :

```
ros2 run turtlesim turtlesim_node
```

 Vous devriez voir apparaître une fenêtre contenant une tortue ! C'est notre robot simulé.

Sous ROS, ce que nous venons de lancer s'appelle un **nœud**. Les nœuds sont les **briques de construction de base** pour nos programmes ROS. Un nœud correspond a une **tâche** spécifique à réaliser sur le système robotique (contrôler un moteur par exemple, ou récupérer les données d'un capteur).

Les nœuds créés sont regroupés en "paquets" (*package*) et on lance un nœud en indiquant le nom de son paquet et son propre nom :

```
ros2 run package_name node_name
```

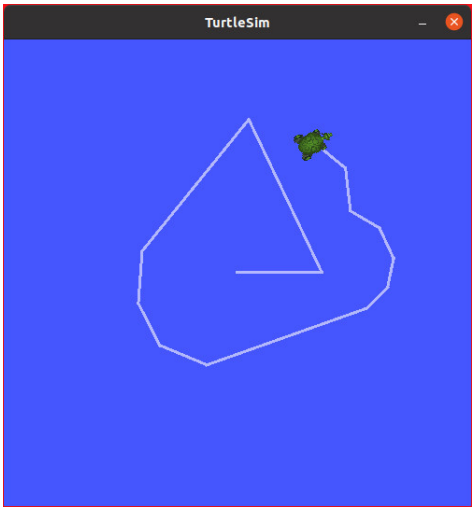
Chaque nœud est un **exécutable à part entière**, indépendant des autres, que l'on peut lancer pour les **ajouter dynamiquement** au programme en cours.

EXERCICE 6 (*Une deuxième nœud pour téléopérer la tortue*)

Laissez tournez le premier terminal, et ouvrez-en un second dans lequel vous écrivez la commande suivante pour lancer un deuxième nœud :

```
ros2 run turtlesim turtle_teleop_key
```

Vous pouvez maintenant utiliser les flèches de votre clavier pour déplacer la tortue (gauche et droite pour tourner, haut et bas pour avancer ou reculer).

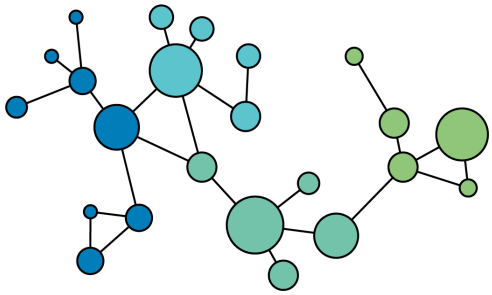


Vous laisserez tourner les deux terminaux jusqu'à la fin de ce TD.

3.2 ROS : une architecture distribuée

Concrètement, comment ça marche ?

Un programme ROS est organisé sous la forme d'**un graphe**, où chaque nœud est **connecté** avec un ou plusieurs autres nœuds (il ne peut pas y avoir de nœuds isolés). L'ensemble fonctionne sur une **architecture distribuée** DDS (Data Distribution Service).



Ce choix d'architecture permet d'éviter que le processus global se retrouve bloqué si un nœud meurt ou buggue (ce qui est quand même rassurant lorsque l'on travaille avec des drones par exemple) et permet de reconfigurer le robot à chaud (comme faire redémarrer un dispositif matériel par exemple).

Dans notre programme ROS `Turtlesim`, nous utilisons deux nœuds, `turtlesim_node` et `turtle_teleop_key` qui communiquent entre-eux :

- Le premier nœud que nous avons lancé, `turtlesim_node`, correspond au simulateur : il affiche la fenêtre avec la tortue et attend des ordres de déplacements.
- Le second nœud, `turtle_teleop_key`, permet un contrôle du déplacement de la tortue via le clavier : il lit les touches du clavier, traduit cela en ordre de déplacements et passe ces ordres au premier nœud.

Les fonctionnalités de contrôle par clavier ont été rassemblées dans un **nœud à part** car elles **peuvent ne pas être nécessaires** en fonction du **mode de contrôle choisi** pour la tortue (on peut vouloir un déplacement autonome par exemple).

Les deux nœuds que nous avons lancés appartiennent au même paquet : `turtlesim`. ROS fait tourner en parallèle des exécutables qui vont **interagir entre eux** : il s'agit donc avant tout d'une **plateforme de commandes en ligne**, mais contrairement à Linux, les exécutables ne peuvent pas être lancés directement sans être identifiés. C'est là qu'interviennent les paquets : l'identification d'un nœud passe par son appartenance à un groupe de nœuds.

Un paquet est un groupe de nœuds et chaque nœud appartient à un seul paquet.

Chaque paquet a une vocation particulière et cette organisation rend impossible la confusion entre nœuds, même s'ils portent le même nom (ils appartiendront forcément à des paquets différents) ! L'intégration de toute contribution, même à très large échelle, se fait donc sans aucun problème!

EXERCICE 7 (*Lister les nœuds en cours d'exécution*)

Ouvrez un troisième terminal et lancez la commande suivante pour afficher tous les noeuds en cours d'exécution de votre programme ROS :

```
ros2 node list
```

Est-ce que vous retrouvez bien nos deux nœuds `/teleop_turtle` et `/turtlesim` ?

Les commandes ROS sur les noeuds que nous avons abordées dans cette partie sont résumées dans l'annexe A page 17.

Communication entre nœuds

Les nœuds **communiquent entre eux** et envoient ou reçoivent des données d'autres nœuds **via différents moyens de communication** que nous allons découvrir dans ce qui suit (forums, services et actions). Nous découvrirons également les paramètres associés à un nœud.

4 Échange d'information entre les nœuds : forums (*topics*) et messages

4.1 Découvrons les forums

Une des méthodes pour que des nœuds d'un programme ROS interagissent entre eux est l'échange d'information sur des forums. Le principe est très simple et ressemble à l'échange d'information sur les réseaux sociaux. **Des contributeurs** vont **publier** (*publish*) de l'information sur **un forum** (*topics*) et tous **les abonnés** du forum vont **recevoir** (*subscribe*) cette information.

Un même nœud peut donc être contributeur en publiant sur un ou plusieurs forums et peut aussi être abonné à un ou plusieurs autres forums.

EXERCICE 8 (*Afficher les forums du programme Turtlesim*)

Lancez la commande suivante dans un terminal pour affocher tous les forums de `Turtlesim` et le type de message échangé sur chacun des forums (indiqué entre crochets) :

```
ros2 topic list -t
```

Vous devriez avoir 5 forums actifs.

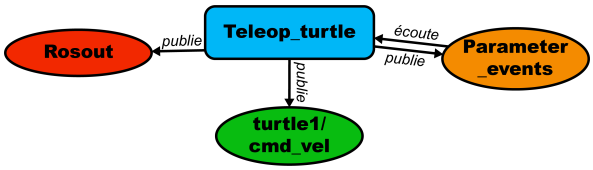
Maintenant que l'on sait visualiser les nœuds et les forums actifs, il faut comprendre comment ces différents composants interagissent (i.e. leur graphe d'interaction) où plus concrètement quel nœud publie ou écoute sur quel forum et quel est le type des informations échangées (message).

EXERCICE 9 (*Informations d'un nœud*)

Dans un terminal, lancez la ligne de commande suivante pour afficher les informations du nœud `teleop_turtle` et voir sur quel(s) forum(s) il écoute (*subscribers*) ou publie (*publishers*).

```
ros2 node info /teleop_turtle
```

Vous devriez retrouver les informations suivantes :

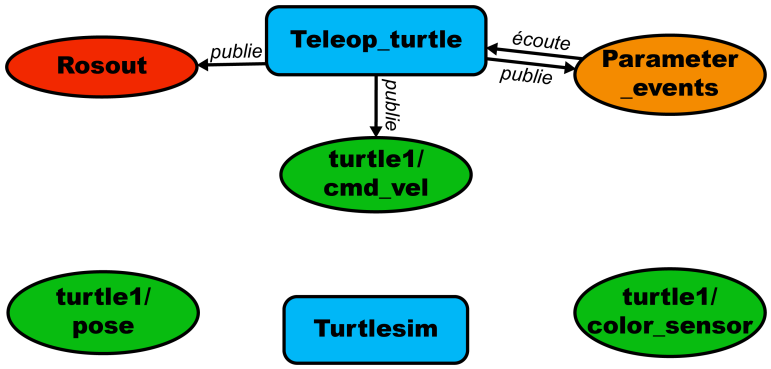


Le type des messages utilisés pour chaque forums est précisé après les deux points de chaque ligne. On retrouve par exemple pour les publications sur le forum `turtle1/cmd_vel` le format de message `geometry_msgs/msg/Twist` que nous avait indiqué la commande de l'exercice précédent.

Pensez à utiliser la touche `TAB` pour compléter efficacement les commandes sans risque de faute de frappe.

EXERCICE 10 (*Informations d'un nœud – 2*)

Lancez la même commande pour notre second nœud, `turtlesim` et compléter le schéma :



EXERCICE 11 (*Informations d'un forum*)

Lancez la commande suivante pour afficher cette fois-ci les informations relatives à un forum et retrouvez les informations sur le type de message, les noeuds qui publient sur ce forum et ceux qui écoutent :

```
ros2 topic info -v /turtle1/cmd_vel
```

Est-ce que vous obtenez bien quelque chose de conforme à votre schéma de l'exercice précédent ?

Modes de communication

La communication entre les nœuds via un forum peut se faire d'un nœud vers un nœud (mode *point-to-point*, un publieur et un écouteur) mais aussi d'un nœud vers plusieurs nœuds (mode *one-to-many*, un publieur et plusieurs écouteurs) ou de manière encore plus générale de plusieurs nœuds vers plusieurs nœuds (mode *many-to-many*, plusieurs publieurs et plusieurs écouteurs).

4.2 Manipulons les messages

L'étape d'après consiste à identifier de manière concrète quels sont les champs qui compose un type de message donné. Intéressons-nous par exemple au message `geometry_msgs/msg/Twist` qui était utilisé par nos deux nœuds à travers le forum `turtle1/cmd_vel`

EXERCICE 12 (*Identifier les champs d'un message*)

Lancez la commande suivante pour afficher le format d'un message de type `geometry_msgs/msg/Twist` :

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

Il vous est indiqué que ce type de message est composé de 2 vecteurs, chacun eux-mêmes composés de 3 réels de type `float64`.

- Le vecteur `linear` donne la vitesse 3D de déplacement avec `x` en avant.

- Le vecteur `angular` donne la vitesse angulaire 3D de rotation dans le sens trigonométrique en radian/seconde.


La tortue ne peut pas se déplacer en crabe ni décoller donc les valeurs `y` et `z` de `linear` seront toujours nulles dans cette simulation.


De même, la tortue n'a pas de roulis ni de tangage, elle ne se déplace que dans le plan horizontal (changement de cap, dit "lacet"), les valeurs `x` et `y` de `angular` seront donc toujours nulles dans cette simulation.

EXERCICE 13 (*On vérifie ? (Visualiser le contenu des messages publiés)*)


Lancez la commande suivante pour afficher le contenu des messages publiés sur le forum `/turtle1/cmd_vel` et déplacez la tortue à l'aide des flèches du clavier :

```
ros2 topic echo /turtle1/cmd_vel
```


Est-ce que vous avez bien le `x` du vecteur `linear` a ± 2.0 et tout le reste à 0.0 quand vous avancez ou reculez ?


Est-ce que vous avez bien le `z` du vecteur `angular` a ± 2.0 et tout le reste à 0.0 quand vous tournez à gauche ou à droite ?

Seuls les champs `linear: x` ou `angular : z` sont actifs.
Pour le premier, une valeur positive correspond à un déplacement en avant, et négatif en arrière. Pour le second, une valeur positive tourne la tortue dans le sens direct (sens anti-horaire / à gauche), et négative dans le sens indirect (sens horaire / à droite).

Maintenant que l'on connait le contenu du message qui permet de faire se déplacer la tortue, on va pouvoir publier directement des messages de ce type sur le forum, sans passer par le clavier et le nœud `turtle_teleop_key`.


Relancez le nœud `turtlesim_node` pour avoir un écran vierge.

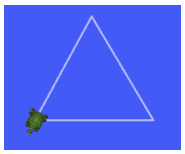
EXERCICE 14 (*Publication sur un forum : trajectoire triangulaire*)


Faire suivre à la tortue une trajectoire dessinant un triangle équilatéral en lançant trois fois les deux lignes de commandes suivantes dans un terminal (sans la dernière rotation finale, 5 lignes à lancer au total donc) :

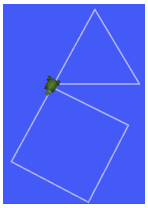
1.
```
ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 4.0, y: 0.0,
    z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
```

2.
```
ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0.0, y: 0.0,
    z: 0.0}, angular: {x: 0.0, y: 0.0, z: 2.09}}"
```


On avance de 4, on tourne ensuite de 120 ° (2.09 en radians), puis on avance de 4, on tourne à nouveau de 120 ° et enfin, on avance une dernière fois de 4.



EXERCICE 15 (*Publication sur un forum : trajectoire carrée*)



On va poursuivre la trajectoire sur un carré.

En vous inspirant des lignes précédentes, lancez les commandes pour avancer de 4, puis pour tourner à 90 ° (1.57 en radians) et ce, 4 fois pour former les quatre côté du carré.

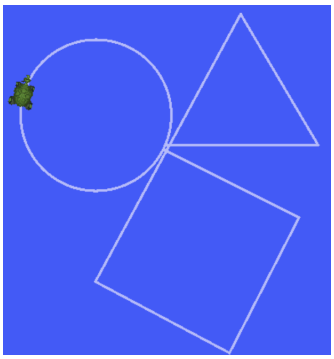
EXERCICE 16 (*Publication sur un forum : trajectoire circulaire*)

On veut terminer le cheminement de la tortue par une trajectoire circulaire.

Pour faire un cercle, il faut en même temps avancer et tourner d'un angle constant.

Donc on utilisera qu'une seule ligne de commande avec les deux valeurs, l'une dans `linear : x` et l'autre dans `angular : z` que l'on va répéter. Pour cela, retirez le `--once` de votre ligne de commande. Vous pourrez prendre 4 en linéaire et -2 radians en angle.

Plus l'**angle** est **grand**, plus le **cercle** sera **petit**.



4.3 Fréquence de publication des messages

Les messages qui sont publiés en continu sur un forum (sans l'option `--once`) le sont à une certaine fréquence. On peut afficher la valeur de cette fréquence et la modifier si on le souhaite.

EXERCICE 17 (*Afficher la fréquence d'une publication*)

Dans un premier terminal, relancez la commande qui publie le message de trajectoire circulaire en continu :

```
ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 4.0, y: 0.0, z: 0.0},
    angular: {x: 0.0, y: 0.0, z: -2.0}}"
```

Puis dans un autre terminal, lancez la commande suivante pour connaître le nombre de message par seconde émis en moyenne sur le forum :

```
ros2 topic hz /turtle1/cmd_vel
```

Est-ce que vous trouvez bien 1 Hz ?

EXERCICE 18 (*Modifier la fréquence d'une publication*)

Lancez maintenant la commande suivante :

```
ros2 topic pub --rate 2 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 4.0, y: 0.0,
    z: 0.0}, angular: {x: 0.0, y: 0.0, z: -2.0}}"
```

Affichez de nouveau la fréquence de publication et vérifiez qu'elle a bien doublé (2 hz) !

Les commandes ROS sur les forums et les messages que nous avons abordées dans cette partie sont résumées dans l'annexe B page 17.

5 Échange d'information entre les nœuds : services et types

5.1 Découvrons les services

Le système de communication par forum fonctionne en permanence : les nœuds écoutent et publient en continu, ce qui est assez coûteux en terme de bande passante et peut être sur-dimensionné pour certains besoins.

Les services sont une autre possibilité pour échanger de l'information entre les nœuds.

- **Les services sont bi-directionnels** : un nœud envoie une requête à un autre nœud et attend une réponse. La réponse revient uniquement au nœud qui a émis la requête, contrairement au forum où la même information est partagée à tous les nœuds écoutants. La communication se fait en mode *one-to-one*.
- **La bande passante** n'est donc sollicitée qu'au moment de la transmission de la requête et de la transmission de la réponse. Ce type d'échange est donc bien adapté pour des échanges ponctuels. Les forums eux sont particulièrement bien adaptés pour des flux d'échange d'information en continu.
- **Chaque service a un type** qui décrit la structure de la requête et de la réponse.

EXERCICE 19 (*Afficher les services actifs de Turtlesim*)

Lancez cette commande dans un terminal pour afficher tous les services actifs de `Turtlesim` avec leur type entre crochets :

```
ros2 service list -t
```

Nous nous intéresserons dans cette partie aux services qui ne contiennent pas le mot `parameter` : ces derniers seront traités dans la partie suivante.

EXERCICE 20 (*Afficher les services actifs utilisant un type donné*)

Affichez maintenant la liste de tous les services actifs qui utilisent par exemple le type `std_srvs/srv/Empty` avec la ligne de commande suivante :

```
ros2 service find std_srvs/srv/Empty
```

Est-ce que vous retrouvez bien `clear` et `reset` ?

5.2 Manipuler les types

Le type d'un service décrit la structure de l'information qui s'échange entre le nœud qui fait une requête et celui qui lui répond.

Cette structure est divisée en deux parties : celle qui décrit la structure de la requête et celle qui décrit la structure de la réponse du serveur.

EXERCICE 21 (*Afficher le détail d'un type*)

Affichez le détail du type `std_srvs/srv/Empty` vu à l'exercice précédent avec la ligne de commande suivante :

```
ros2 interface show std_srvs/srv/Empty
```

`---` ? Il n'y a rien avant `---` (le séparateur entre requête et réponse) et rien après, cela signifie que le service est sollicité avec une **requête** et une **réponse** qui ne contiennent **pas d'information** (d'où son nom : "*Empty*").

Modifier votre ligne de commande pour afficher le détail d'un autre type : `turtlesim/srv/Spawn`.

Cette fois-ci, vous pouvez observer que la **requête** passe comme information une **position** composée de deux valeurs (x et y), une **orientation** et un **champ texte optionnel**. La **réponse** renvoie un **champ texte** qui correspond à celui envoyé ou un champ texte généré de manière unique si le champ n'avait pas été renseigné.

EXERCICE 22 (*Appel d'un service*)

Dans un terminal, appelez le service `clear` en lançant la commande suivante :

```
ros2 service call /clear std_srvs/srv/Empty
```

Vous devriez voir disparaître le tracé fait par la tortue dans la fenêtre.

Appelez maintenant le service `spawn` en lançant la ligne suivante :

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.79, name: 'MyNewTurtle'
    }"
```

Vous devriez voir apparaître une nouvelle tortue aux coordonnées indiquées (le coin en bas à gauche est l'origine du repère) et avec l'orientation donnée (45 °) !

QUESTION 1 (*Contrôle des deux tortues ?*)

Que se passe-t-il si vous retrounez sur le terminal du nœud `turtle_teleop_key` et que vous jouez avec les flèches du clavier ?

La nouvelle tortue reste immobile car le nœud `turtle_teleop_key` ne contrôle que la première.

QUESTION 2 (*Contrôle des deux tortues : 2^e essai*)

Et si vous lancez un nouveau nœud `turtle_teleop_key` dans un autre terminal ?

Malheureusement, c'est un échec. Le nouveau nœud contrôle toujours la tortue originale ! Cela veut dire qu'on n'est pas sur le bon forum, n'est-ce pas ?

EXERCICE 23 (*Contrôle des deux tortues : enquête*)

Affichez la liste des forums actifs (voir page 6 ou page 17 si vous avez oublié la commande !).

Une partie des forums ont été dupliqués : il y a ceux pour `turtle1` notre tortue originale, et ceux pour notre nouvelle tortue `MyNewTurtle`. Le forum qui nous intéresse pour diriger la tortue, comme nous l'avons vu dans la partie précédente, est `/MyNewTurtle/cmd_vel`.

EXERCICE 24 (*Contrôle des deux tortues : 3e essai*)

Dans un nouveau terminal, lancez la commande suivante avec l'argument `--remap` qui permet d'effectuer un changement de paramètre par défaut au moment du lancement d'un nouveau nœud :

```
ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=MyNewTurtle/
    cmd_vel
```

Et maintenant, quand vous activez les flèches du clavier, est-ce que ça fonctionne ?

Les commandes ROS sur les services et les types que nous avons abordées dans cette partie sont résumées dans l'annexe C page 18.

6 Échange d'information entre les nœuds : actions

6.1 Découvrons les actions

Les actions sont une autre façon de communiquer entre des nœuds. Elles sont définies pour des taches qui prennent du temps à être réaliser et pour lesquelles on souhaite avoir un *feedback* de là où on en est en cours d'exécution.

Ainsi, une action est composée d'un objectif à atteindre, d'un *feedback* de l'exécution et du résultat final atteint.

Plus en détails, une action est lancée par un nœud et exécutée par un autre. L'architecture d'une action est composée d'un serveur client sur le nœud qui lance l'action et d'un serveur action sur le nœud qui exécute la tâche.

- Le serveur client émet l'objectif, reçoit la mise à jour des *feedbacks* et le résultat final atteint.
- Le serveur action reçoit l'objectif, émet le *feedback* lors de l'exécution de la tache et envoie le résultat final atteint.

Une action utilise donc des services pour émettre l'objectif et recueillir le résultat atteint et un forum pour échanger des messages sur le *feedback*.

EXERCICE 25 (*Afficher les actions de Turtlesim*)

Voyons ça de manière un peu plus concrète. Comme précédemment, on va commencer par afficher la liste des actions et de leurs types (entre crochets) que l'on a à notre disposition en lançant la commande suivante dans un terminal :

```
ros2 action list -t
```

On remarque qu'on a une action qui s'appelle `rotate_absolute`.

EXERCICE 26 (*Testons l'action de rotation absolue*)

Si vous regardez dans le terminal où a été lancé `turtle_teleop_key`, vous verrez l'information suivante :

```
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
```

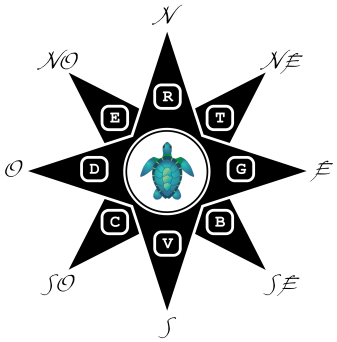
Voici où se trouve notre action `rotate_absolute` !

Avec ces touches, on peut ordonner à la tortue de prendre une orientation absolue. Essayez-les !

Vous avez vu ? La tortue mets du temps à rejoindre l'orientation absolue donnée : on est bien dans une action !

Orientez vous au Nord par exemple et attendez que la tortue atteigne cette orientation. Puis lancer l'action pour l'orienter plein Sud. Avant que la tortue n'y arrive, appuyez sur la touche `F`.

Les actions peuvent s'interrompre (contrairement aux services qui sont séquentiels et ne peuvent pas être interrompus)



EXERCICE 27 (*Informations sur une action*)

Lancez la ligne de commande suivante dans un terminal pour obtenir plus d'information sur l'action `rotate_absolute` :

```
ros2 action info /turtle1/rotate_absolute
```

6.2 Manipulons les types

EXERCICE 28 (*Afficher le détail d'un type*)

Affichez le détail du type `rotate_absolute` vu à l'exercice précédent avec la ligne de commande suivante :

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

Vous devriez avoir la consigne (angle theta), le résultat (angle delta, l'écart à la consigne) et le *feedback* (l'angle *remaining*).

EXERCICE 29 (*Lancer manuellement une action*)

Lancer la commande suivante qui permet de déclencher manuellement une action (comme si on était un nœud client) :

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta:
    1.57}"
```

EXERCICE 30 (*Lancer manuellement une action en affichant le feedback*)

Relancer la ligne de commande précédente en modifiant la consigne (2.57 par exemple) et en ajoutant `--feedback` à la fin.

Les commandes ROS sur les services et les types que nous avons abordées dans cette partie sont résumées dans l'annexe E page 19.

7 Paramètres d'un nœud

7.1 Découvrons les paramètres

Chaque nœud est configuré lors de son lancement et durant toute sa durée de vie par un ensemble de paramètres. Il est possible de changer la valeur de certains de ces paramètres lors de son lancement (comme on l'a fait dans l'exercice précédent pour changer le forum de publication par défaut) ou en cours d'exécution sans avoir à changer le code du nœud, de le recompiler et de le relancer.

C'est donc une méthode complémentaire des forums et services de passer de l'information à un nœud.

Elle est bien adaptée pour des informations qui n'ont pas vocation à changer souvent, comme les paramètres de configuration que l'on ne change que de manière exceptionnelle.

EXERCICE 31 (*Afficher les paramètres des nœuds actifs*)

Dans un terminal, lancez la commande suivante pour afficher la liste de tous les paramètres des nœuds actifs :

```
ros2 param list
```

Est-ce que vous avez identifié par exemple les paramètres relatifs à l'échelle des commandes envoyées par le nœud `turtle_teleop_key` ou ceux relatifs à la couleur du fond du nœud `turtlesim` ?

EXERCICE 32 (*Information sur un paramètre*)

Maintenant, lancez la ligne de commande suivante pour afficher la valeur d'un paramètre :

```
ros2 param get /turtlesim background_b
```

La commande nous renvoie le type de ce paramètre (un entier) et sa valeur actuelle. Il s'agit de la valeur B (bleue) du triplet (R,G,B) qui décrit la couleur du fond de la fenêtre du simulateur.

Affichez maintenant les valeurs R et G et vérifiez que c'est bien concordant avec ce que vous voyez !

7.2 Manipulons les paramètres

EXERCICE 33 (*Modifier des paramètres*)

Lancez la ligne de commande suivante pour modifier la valeur de notre paramètre :

```
ros2 param set /turtlesim background_g 255
```

Vous êtes bien passé dans les tropiques ?

Sur le même principe, modifiez les paramètres pour que la tortue ne se déplace plus dans la mer mais sur le sable (en RGB : 242, 231, 191).

EXERCICE 34 (*Afficher tous les paramètres d'un nœud*)

Lorsque l'on souhaite afficher toutes les valeurs des paramètres d'un nœud, c'est un peu fastidieux d'utiliser la commande précédente sur chacun des paramètres. Il existe une commande spécifique pour ça, lancez-la dans un terminal pour connaître toutes les valeurs des paramètres du nœud `turtlesim` :

```
ros2 param dump /turtlesim
```

EXERCICE 35 (*Sauvegarder les paramètres d'un nœud*)

Lancez la commande suivante dans un terminal pour sauvegarder dans un fichier au format YAML tous les paramètres du nœud `turtlesim` :

```
ros2 param dump /turtlesim > turtlesim.yaml
```

EXERCICE 36 (*Modifier au lancement plusieurs paramètres via un fichier*)

Modifiez le fichier `turtlesim.yaml` précédent pour que la couleur du fond de la fenêtre soit de couleur sable (RGB : 242, 231, 191) et fermez toutes les fenêtres `turtlesim` actives.

Lancez ensuite la commande suivante sur un terminal pour démarrer le nœud `turtlesim` avec comme paramètres d'initialisation ceux du fichier `turtlesim.yaml` :

```
ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml
```

La fenêtre s'est-elle bien ouverte avec la bonne couleur ? Si oui, on a réussi à changer 3 paramètres en même temps au lancement du nœud !

EXERCICE 37 (*Modifier en live plusieurs paramètres via un fichier*)

Sauvegardez maintenant dans un fichier `turtlesim_bg_sea.yaml` uniquement les paramètres de fond de fenêtre avec les valeurs mer (RGB : 69, 86, 255). Vous garderez la hiérarchie de ces paramètres, c'est-à-dire les lignes suivantes :

```
/turtlesim:
  ros__parameters:
```

Puis dans un nouveau terminal, lancez la commande suivante pour modifier à chaud les trois valeurs de la couleur du fond à partir du fichier :

```
ros2 param load /turtlesim turtlesim_bg_sea.yaml
```

La couleur de la fenêtre a-t-elle bien été modifiée ? Si oui, on a réussi à changer 3 paramètres en même temps au cours de l'exécution du nœud !

Seuls les paramètres qui ne sont pas en lecture seule peuvent être modifiés à chaud !

En vous basant sur la ligne de commande précédente, testez de lancer le fichier complet de paramètres `turtlesim.yaml` en cours d'exécution du nœud.

Les paramètres qui ne sont pas en lecture seule seront bien actualisés (la tortue repasse dans le sable) et il sera mentionné `successful` pour ceux-la dans le retour de la commande. En revanche pour ceux en lecture seule, il vous sera indiqué `failed: parameter XXX cannot be set because it is read-only`.

Les commandes ROS sur les paramètres que nous avons abordées dans cette partie sont résumées dans l'annexe D page 18.

Fin du premier TD

Nous avons découvert dans ce TD les briques de bases d'un programme ROS, la manière dont elles s'architecturent ainsi que les commandes élémentaires pour les manipuler. Dans le TD suivant, nous allons nous intéresser à comment les coder.

Annexe A: Commandes sur les nœuds

Commande	Action
`ros2 run package_name node_name`	lance l'exécution du nœud `node_name` du paquet `package_name` (`ctrl-c` pour arrêter l'exécution du nœud)
`ros2 node list`	liste tous les nœuds en cours d'exécution
`ros2 node info node_name`	donne les informations liées au nœud `node_name` comme le type de message émis ou lus, sur quels forums, les connexions aux autres nœuds...
`ros2 run package_name node_name --ros-args --remap foo:=bar`	remplace le paramètre par défaut `foo` par l'argument `bar` au moment du lancement du nœud `node_name` du package `package_name`

Table 1 – *Commandes liées aux nœuds.*

Annexe B: Commandes sur les forums et les messages

Commande	Action
`ros2 topic list -t`	affiche la liste des forums et entre crochets le type de message publié sur chaque forum
`ros2 topic info -v topic_name`	donne les informations liées au forum `topic_name` comme le type de message échangé, les nœuds publiants ou abonnés ...
`ros2 interface show msg_type`	donne la structure du message `msg_type` (i.e les champs et leur type)
`ros2 topic echo topic_name`	permet d'afficher en temps réel les messages publiés sur le topic `topic_name` (`ctrl-c` pour arrêter)
`ros2 topic pub topic_name msg_type "args"`	publie en boucle sur le forum `topicName` le message `msgType` d'arguments `args` au format YAML (`crtl-c` pour arrêter)
`ros2 topic pub --once topic_name msg_type "args"`	publie une seule fois sur le forum `topic_name` le message `msg_type` d'arguments `args` au format YAML
`ros2 topic hz topic_name`	permet de mesurer en unité de message par seconde la fréquence d'émission des messages sur le forum `topic_name`
`ros2 topic pub --rate fhz topic_name msg_type "args"`	publie en boucle à la fréquence `fhz` sur le forum `topic_name` le message de type `msg_type` et d'arguments `args` au format YAML

Table 2 – *Commandes liées aux forums et aux messages.*

Annexe C: Commandes sur les services et les types

Commande	Action
`ros2 service list`	liste tous les services actifs (i.e. disponibles pour être appelés)
`ros2 service list -t`	liste tous les services actifs et leur type entre crochets
`ros2 service type service_name`	affiche le type du service `service_name`
`ros2 service find type_name`	affiche tous les services actifs qui utilisent le type `type_name`
`ros2 interface show type_name`	affiche le détail du type `type_name`. La partie requête est au-dessus du `---` et la partie réponse en dessous.
`ros2 service call service_name service_type "args"`	appelle le service `service_name` avec le type `service_type` et les arguments `"args"`

Table 3 – *Commandes liées aux services.*

Annexe D: Commandes sur les paramètres

Commande	Action
`ros2 param list`	liste les paramètres des nœuds actifs
`ros2 param get node_name parameter_name`	affiche le type et la valeur du paramètre `parameter_name` du nœud `node_name`
`ros2 param set node_name parameter_name value`	modifie la valeur du paramètre `parameter_name` du nœud `node_name` en lui assignant la valeur `value`
`ros2 param dump node_name`	affiche tous les paramètres et leur valeur du nœud `node_name`
`ros2 param dump node_name > file_name`	sauvegarde au format YAML tous les paramètres et leur valeur du nœud `node_name` dans le fichier `file_name`
`ros2 run package_name node_name --ros-args --params-file file_name`	lance l'exécution du nœud `node_name` du paquet `package_name` avec comme valeurs de paramètres celles enregistrées dans le fichier `file_name`
`ros2 param load node_name file_name`	modifie à chaud les valeur des paramètres du nœud `node_name` à partir du fichier `file_name`

Table 4 – *Commandes liées aux paramètres.*

Annexe E: Commandes sur les actions

Commande	Action
`ros2 action list -t`	liste les actions des nœuds actifs et leur type entre crochets
`ros2 action info action_name`	précise quels sont les nœuds qui hébergent le service client et le service action
`ros2 interface show action_type`	détaille le type de l'objectif `---` le type du résultat final `---` le type du feedback
`ros2 action send_goal action_name action_type "args"`	lance manuellement l'action `action_name` de type `action_type` et d'arguments `args`
`ros2 action send_goal action_name action_type "args" --feedback`	lance manuellement l'action `action_name` de type `action_type` et d'arguments `args` en affichant tous les feedbacks

Table 5 – *Commandes liées aux actions.*