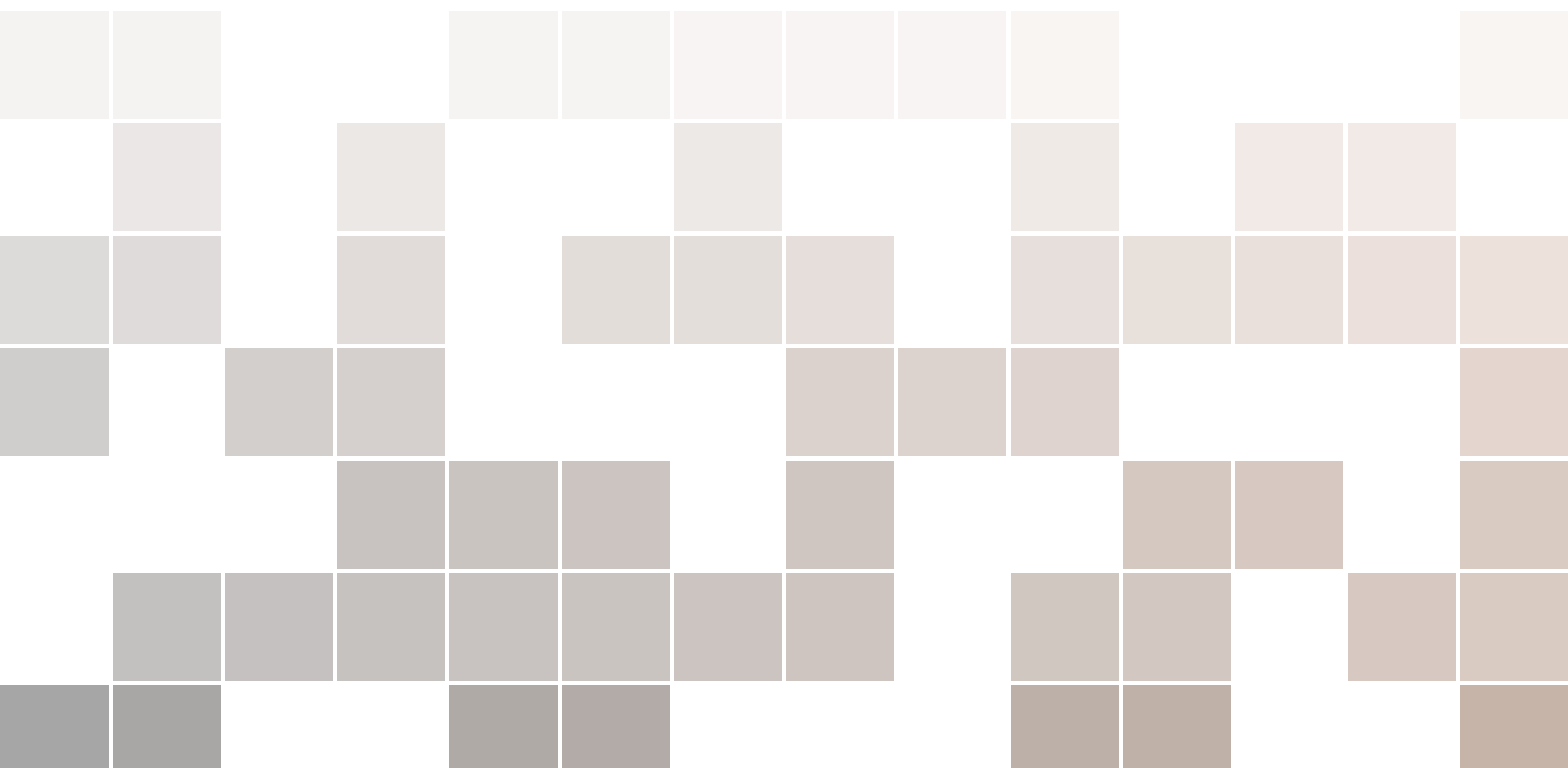
Robot Operating System

Introduction

Olivier Stasse



Copyright © 2022 Olivier Stasse

PUBLISHED BY LAAS-CNRS

HTTPS ://HOMEPAGES.LAAS.FR/ OSTASSE

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at `http://creativecommons.org/licenses/by-nc/3.0`. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Edition, 2022/03/18

Table des matières

1 Introduction 9

1.1 Motivations 9

1.1.1 Complexité des systèmes robotiques 9

1.1.2 Application Robotique 10

1.2 Concepts de ROS 10

1.2.1 Plomberie 10

1.2.2 Outils 11

1.2.3 Capacités 11

1.2.4 Exemples 12

1.3 Ecosystème 12

1.3.1 Historique 12

1.3.2 Fonctionnalités 12

1.3.3 Systèmes supportés 14

1.4 Exemple : Asservissement visuel sur TurtleBot 2 15

I Introduction à ROS-2

2 Graphe d'application avec ROS-2 21

2.1 Turtlesim et rqt 21

2.1.1 Utilisation de turtlesim 21

2.1.2 Installer rqt 23

2.1.3 Utiliser rqt 23

2.1.4 Essayer le service spawn 23

2.1.5 Essayer le service **set_pen** 24

2.1.6 Redirection 25

2.1.7 Arrêt de turtlesim 26

2.2 Node 26
2.2.1 Définition d'un node . . . 26
2.2.2 ros2 run . . . 26
2.2.3 ros2 node list . . . 26
2.2.4 Redirection . . . 27
2.2.5 ros2 node info . . . 27
2.2.6 rqt_graph . . . 28
2.2.7 Topic . . . 28
2.3 Service 32
2.3.1 ros2 service list . . . 32
2.3.2 ros2 service type (service) . . . 33
2.3.3 ros2 service list -t . . . 33
2.3.4 ros2 interface show . . . 33
2.3.5 ros2 service call . . . 33
2.4 Paramètres 34
2.4.1 ros2 param list . . . 35
2.4.2 ros2 param dump . . . 35
2.4.3 ros2 param load . . . 36
2.5 Actions 36
2.5.1 Introduction . . . 36
2.5.2 Préparation . . . 36
2.5.3 Utilisation des actions . . . 37
2.5.4 ros2 node info . . . 37
2.5.5 ros2 action list . . . 38
2.5.6 ros2 action info . . . 38
2.5.7 ros2 interface show . . . 39
2.5.8 ros2 action send_goal . . . 39
2.5.9 Résumé . . . 40
2.6 rqt_console 40
2.6.1 rqt . . . 40
2.6.2 rqt_console . . . 40
2.6.3 Niveaux d'enregistrement . . . 41
2.7 Bag : fichier de données 41
2.8 Launch : démarrer une application 42
2.8.1 Introduction . . . 42
2.8.2 Un exemple . . . 42
2.8.3 Préparer son package . . . 43
2.9 Création de messages et de services 44
2.9.1 Introduction . . . 44
2.9.2 Création d'un msg . . . 45
2.9.3 Création d'un srv . . . 45
2.9.4 CMakeList.txt . . . 46
2.9.5 package.xml . . . 46
2.9.6 Génération des codes . . . 46
2.9.7 Vérifier la création du fichier msg et srv . . . 46
3 Les paquets ROS . . . 47
3.1 Configuration de l'environnement 47
3.1.1 Initialiser les variables d'environnement . . . 47
3.1.2 Navigation dans le système de paquets . . . 47

3.2 Structure générale des paquets ROS 48
3.3 Création d'un paquet 48
3.4 Description des paquets : le fichier package.xml 48
3.5 Compilation des paquets humble 51
3.6 Chaînage de workspace 51

4 Ecrire des nodes ROS-2 53
4.1 Topics 53
4.1.1 Création d'un paquet 53
4.1.2 Emetteur 54
4.1.3 Souscripteur 59
4.1.4 Lancer les nodes 61
4.2 Services 63
4.2.1 Création d'un paquet 63
4.2.2 Serveur 64
4.2.3 Client 66
4.2.4 Lancer les nodes 69

II Travaux Pratiques

5 TurtleBot3 73
5.1 Démarrage 73
5.1.1 Introduction Modèle du TurtleBot3 73
5.1.2 Installation 73
5.2 Gazebo 73
5.2.1 Téléopérer le turtlebot 74
5.3 Construction de cartes et navigation 74
5.3.1 Construction de cartes 74
5.3.2 Navigation 75
5.3.3 RVIZ 75

III Modèles et Simulation

6 Universal Robot Description Format (URDF) 79
6.1 Capteurs - Sensors 79
6.1.1 Norme courante 79
6.1.2 Limitations 81
6.2 Corps - Link 81
6.2.1 Attributs 81
6.2.2 Elements 81
6.2.3 Résolution recommandée pour les mailles 82
6.2.4 Elements multiples des corps pour la collision 82
6.3 Transmission 83
6.3.1 Attributs de **transmission** 83
6.3.2 Eléments de **transmission** 83
6.3.3 Notes de développement 84

6.4 **Joint** 84
6.4.1 Elément **<joint>** . . . 84
6.4.2 Attributs . . . 84
6.4.3 Elements . . . 84
6.5 **Gazebo** 85
6.5.1 Eléments pour les corps/links . . . 85
6.5.2 Eléments pour les joints . . . 85
6.6 **model_state** 85
6.6.1 Elément <model_state> . . . 86
6.6.2 Model State . . . 86
6.7 **model** 86

7 **Simulation** . . . 87
7.1 **Introduction** 87
7.2 **Algorithme général d'un simulateur système** 88
7.3 **Algorithme général d'un moteur dynamique** 88
7.3.1 Problème 1 : Realité . . . 88
7.3.2 Problème 2 : Flexibilité logicielle . . . 88

8 **Gazebo - Ignition** . . . 89
8.1 **Introduction** 89
8.2 **SDF format** 89
8.2.1 Quand utiliser le format SDF ou le format URDF ? . . . 90
8.3 **Plugin** 90
8.3.1 Plugin Gazebo . . . 90
8.3.2 Plugin : exemple de base . . . 90
8.4 **Modèle de robot en SDF** 92
8.4.1 Introduction . . . 92
8.4.2 Modèle du soleil . . . 93
8.4.3 Modèle du monde pour notre robot . . . 93
8.4.4 Modèle du robot turtlebot3_waffle modifié . . . 94
8.4.5 Plugin du capteur solaire . . . 102

IV **Appendices**

9 **Mots clefs pour les fichiers launch** . . . 109
9.1 **<launch>** 109
9.1.1 Attributs . . . 109
9.1.2 Elements . . . 109

10 **Rappels sur le bash** . . . 111
10.1 **Lien symbolique** 111
10.1.1 Voir les liens symboliques . . . 111
10.1.2 Créer un lien symbolique . . . 111
10.1.3 Enlever un lien symbolique . . . 111
10.2 **Gestion des variables d'environnement** 112
10.3 **Fichier .bashrc** 112

11 Utiliser docker pour ROS 113
11.1 Préparer votre ordinateur 113
11.2 Utiliser l'image docker 113
11.2.1 Lancer un bash 113
11.2.2 Stopper, redémarrer une image 115
11.2.3 Copier un fichier de et vers un docker 115
11.3 Installer les images docker de ROS 115
11.3.1 Mise à jour de l'image 115
11.3.2 Erreurs 115
11.4 Construire l'image docker 116
11.5 Mac 116
11.5.1 Interface X11 sous Mac 116

12 Mémo 117

Bibliography 121
Books 121
Articles 121
Chapitre de livres 121
Autres 121

1. Introduction

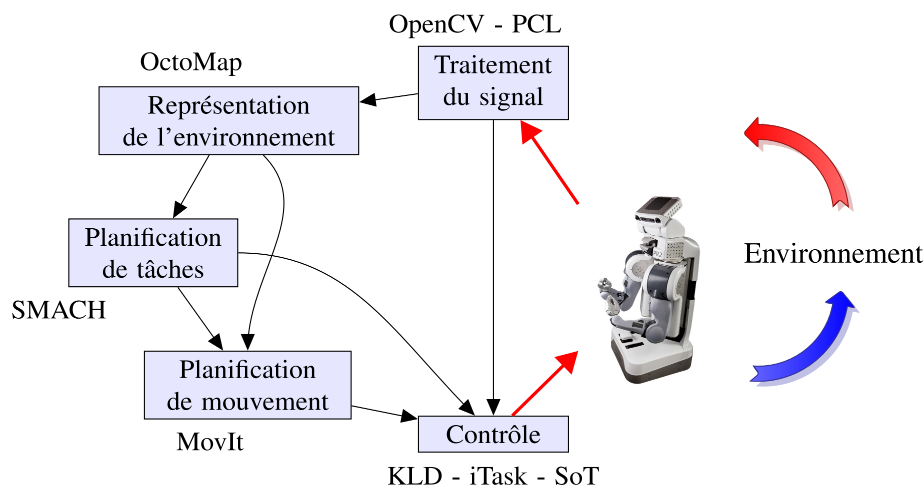


FIGURE 1.1 – Structure d'une application robotique

1.1 Motivations

1.1.1 Complexité des systèmes robotiques

Les systèmes robotiques font appel à de nombreuses compétences telles que la mécanique, l'électrotechnique, le contrôle, la vision par ordinateur, et de nombreux pans de l'informatique comme l'informatique temps-réel, le parallélisme, et le réseau. Souvent les avancées en robotique dépendent de verrous technologiques résolus dans ces champs scientifiques. Un système robotique peut donc être complexe et faire intervenir de nombreuses personnes. Afin de pouvoir être efficace dans la conception de son système robotique il est donc impératif de pouvoir réutiliser au maximum des outils existants et de pouvoir collaborer efficacement avec d'autres équipes.

En physique, des systèmes à grande échelle comme le *Large Hadron Collider* ont vu le jour grâce à des collaborations internationales à grande échelle. En informatique des projets comme le système d'exploitation Linux n'existe qu'à travers une collaboration internationale. Celle-ci peut impliquer des individus, mais on trouve également le support de sociétés qui ne souhaitent pas s'engager dans le développement complet

d'un système d'exploitation mais qui souhaiteraient de nouvelles fonctionnalités. En effet cette approche est souvent moins coûteuse car le développement d'un système d'exploitation est évaluer à 100 années pour un seul homme.

Le système ROS pour Robotics Operating System est le premier projet collaboratif robotique à grande échelle qui fournit un ensemble d'outils informatique permettant de gagner du temps dans la mise au point d'un robot, ou d'un système robotique.

1.1.2 Application Robotique

Une application robotique suit en général la structure représentée dans la figure 1. Elle consiste à générer une commande afin qu'un robot puisse agir sur un environnement à partir de sa perception de celui-ci. Il existe plusieurs boucles de *perception-action*. Elles peuvent être :

- $40\ Khz$ très rapide par exemple pour le contrôle du courant,
- $1\ Khz - 200\ Hz$ rapide, par exemple pour le contrôle du corps complet d'un robot humanoïde,
- $1\ s - 5\ mn$ lente, par exemple dans le cadre de la planification de mouvements réactifs dans des environnements complexes.
- $2 - 72h$ trés lente, par exemple dans le cadre de planification à grande échelle, ou sur des systèmes avec de trés grands nombre de degrés de libertés.

Dans tous les cas, il est nécessaire de transporter l'information entre des composants logiciels largement hétérogènes et faisant appel à des compétences très différentes. Chacun de ces composants logiciels a une structure spécifique qui dépend du problème à résoudre et de la façon dont il est résolu. Chacun est encore très actif du point de vue de la recherche scientifique, ou demande des niveaux de technicité relativement avancés. La réprésentation de l'environnement par exemple a connu récémment beaucoup de changements, et on trouve maintenant des prototypes industriels permettant de construire en temps réel des cartes denses de l'environnement sur des systèmes embarqués [4]. Il est donc nécessaire de pouvoir facilement tester des structures logiciels implémentant des algorithmes différents pour une même fonctionalité sans pour autant changer l'ensemble de la structure de l'application. Par exemple on veut pouvoir changer la boîte "Représentation de l'environnement" sans avoir à modifier les autres boîtes. Ceci est réalisé en utilisant des messages standards et les mêmes appels de service pour accéder aux logiciels correspondants à ces boîtes. Enfin, on souhaiterait pouvoir analyser le comportement du robot à différents niveaux pour investiguer les problèmes potentiels ou mesurer les performances du système. Dans le paragraphe suivant, les concepts de ROS permettant de répondre à ces challenges sont introduits.

1.2 Concepts de ROS

La clef de ROS est la mise en place de 4 grands types de mécanismes permettant de construire une application robotique comme celle décrite dans le paragraphe précédent. Ces mécanismes, décrits dans la figure. 1.2.1, sont :

- la *Plomberie*, c'est à dire la connection des composants logiciels entre eux quelque soit la répartition des noeuds de calcul sur un réseau,
- les *Outils*, c'est à dire un ensemble de logiciels permettant d'analyser, d'afficher et de débugger une application répartie,
- les *Capacités*, c'est à dire des librairies qui implémentent les fonctionnalités telles que la planification de tâches (SMACH), de mouvements (OMPL), la construction d'un modèle de l'environnement (Octomap),
- l'*Ecosystème*, c'est à dire un nombre suffisant d'utilisateurs tels que ceux-ci ont plus intérêt à collaborer plutôt qu'à reconstruire les mêmes outils.

1.2.1 Plomberie

La plomberie, représentée dans la figure 1.2.1, est implémentée grâce à un middleware. Un middleware fournit un bus logiciel qui permet à des objets localisés sur différents ordinateurs d'interagir. L'interaction s'effectue en transmettant des données sous une forme normalisée (les *messages*) via ce que pourrait voir comme des post-its (les *topics*). Un noeud de calcul (un *node*) produisant des données peut ainsi inscrire des données qu'il produit sur un post-it en utilisant le nom de celui-ci. Un autre noeud lui lira les données sur le post-it sans savoir quel noeud a produit ces données. La normalisation des données est réalisée grâce à un langage de description d'interface. Des programmes permettent de générer automatiquement la transcription

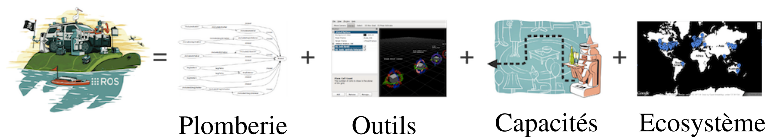


FIGURE 1.2 – Concepts généraux sous-jacents à ROS

entre ce langage de description et des langages de programmation comme le C++, python ou java. Afin de pouvoir communiquer entre les objets, le système fournit un annuaire sur lequel viennent s'enregistrer chacun des noeuds de calcul. Enfin un noeud peut demander un service à un autre noeud en suivant un mécanisme de d'appel de service à distance. Ces mécanismes très généraux existent aussi chez d'autre middlewares comme Corba, ou YARP un autre middleware robotique. En général un middleware fournit les mécanismes d'appels pour différents langagues et différents systèmes d'exploitation. On peut également stockés des paramètres qui correspondent à des données spécifiques à une application, à un robot ou des données qui d'une manière générale n'évoluent pas au cours de l'exécution d'une application.

1.2.2 Outils

Les outils sont une des raisons du succés de ROS. Nous trouvons ainsi :

— *rviz* : Une interface graphique permettant d'afficher les modèles des robots, des cartes de navigation reconstruites par des algorithmes de SLAM, d'interagir avec le robot, d'afficher des images, des points 3D fournis par des caméras 3D.
— *rqt_graph* : Une interface graphique permettant d'analyser le graphe d'applications et les transferts de données via les topics.
— *rosbag* : Un programme permettant d'enregistrer et de rejouer des séquences topics. La fréquence, la durée les topics à enregistrer peuvent être spécifiés.
— *rqt* : Une interface de contrôle incrémentale. Elle se base sur le système de plugins de la bibliothèque de GUI Qt.
— *catkin* : Un système de gestion de paquets, de génération de code automatique et compilation.

Chacun de ces outils a également ses propres limites, et de nombreux alternatives existent dans les paquets fournis sur ROS.

1.2.3 Capacités

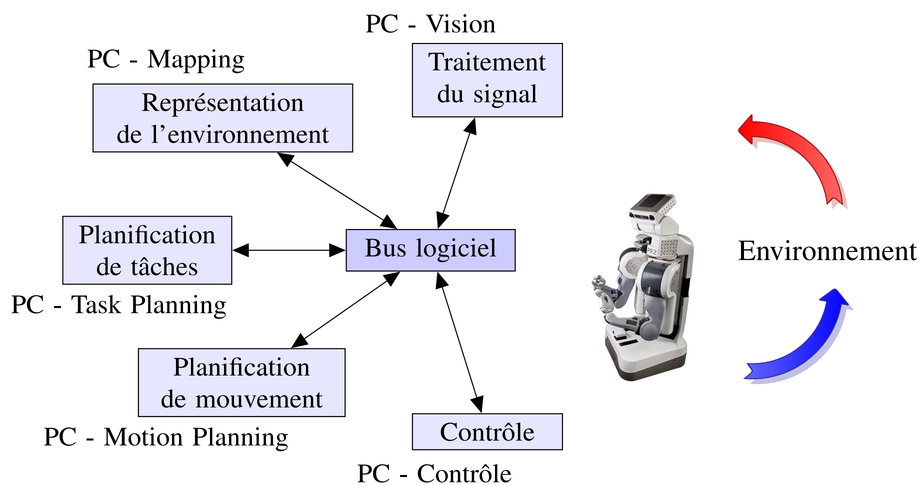


FIGURE 1.3 – Structure d'une application robotique

Les différents fonctionnalités décrites dans la figure 1 sont implémentés par des librairies spécialisés décrites dans la figure 1.2.3. Chacune est issue de plusieurs décennies de recherche en robotique et souvent encore l'objet de nombreuses avancées. Parmi celles-ci on peut trouver :

— Pour la vision :

  - La point cloud library
  - La librairie de vision par ordinateur.
- Pour la reconstruction d'environnements :
  - En 2D : La pile de navigation pour ROS-1 et la https ://navigation.ros.org/pile de navigation ROS-2
  - En 3D : ICP Mapping.
- Pour la planification de mouvements :
  - MoveIt!
- Pour la planification de mission :
  - SMASH avec de la documentation disponible ici
  - Mission Planner pour les drônes
- Pour le contrôle :
  - iTask
  - ControlIt
  - Stack-Of-Tasks

1.2.4 Exemples

ROS-1 a été utilisé comme support pour des challenges robotiques comme le DARPA Robotics Challence et le challenge robotique de la NASA. Dans le cade du DARPA Robotics Challenge qui a pris place en Juin 2015, de nombreuses équipes ont du faire face à des défis d'intégration. ROS-1 a été une des briques fondamentales pour permettre aux équipes utilisant ATLAS de pouvoir fonctionner.

1.3 Ecosystème

1.3.1 Historique

ROS-1 a été écrit à partir de 2008 par la société Willow Garage fondée par Larry Page, également fondateur de Google. En plus de ROS-1, cette société a construit le PR-2, un robot très sophistiqué équippé de deux bras, d'une base mobile et avec des moyens de calcul importants. Les implémentations effectués par cette société ont été nombreux et ont permis d'atteindre une masse critique d'utilisateurs sans commune mesure avec les actions concertées précédentes. Pour cette raison on retrouve dans les documentations des paquets historiques souvent la référence à Willow Garage. Cette société a cependant été fermée en 2013 et a fait place à l'Open Source Robotics Foundation financé par plusieurs société. Un des faits récents les plus notables est le don de 50 millions de dollars effectués par Toyota, suivi d'un autre investissement de 50 millions de dollars pour construire une structure privée adossée à l'OSRF.

L'historique des releases de ROS et ROS-2 sont résumé dans les tableaux 1.1 et 1.2. Si on note au démarrage un manque de régularité, la politique actuelle consiste à caler les releases de ROS sur celles d'Ubuntu en Long Time Support (LTS) et celles plus courtes des années impaires. D'une manière générale on constate un décalage d'au moins une année entre la release de ROS et celle de la mise à jour des applications robotiques. C'est en général le temps qu'il faut pour résoudre les erreurs des releases en LTS et porter le code des applications robotiques.

1.3.2 Fonctionnalités

Il existe un certain nombre de logiciels très bien supportés par des projets et des communautés qui offre des fonctionnalités très intéressantes pour la robotique. Certains existent par ailleurs ROS et d'autres sont supportés directement par l'OSRF. En voici une liste non-exhaustive :

- **Définition de messages standards pour les robots**.
  ros_comm est le projet qui définit et implémente le mécanisme de communication initial et les types initiaux. De nombreux paquets viennent complèter les types nécessaires dans les applications robotiques.
- **Librairie pour la géométrie des robots**.
  La librarie KDL est utilisée pour implémenter le calcul de la position de tous les corps du robot en fonction de ces actionneurs et du modèle des robots.
- **Un language de description des robots**.
  URDF pour Universal Robot Description Format est un spécification en XML décrivant l'arbre cinématique d'un robot, ses caractéristiques dynamiques, sa géométrie et ses capteurs.

2008	Démarrage de ROS par Willow Garage
2010 - Janvier	ROS 1.0
2010 - Mars	Box Turtle
2010 - Aout	C Turtle
2011 - Mars	Diamondback
2011 - Aout	Electric Emys
2012 - Avril	Fuerte
2012 - Décembre	Groovy Galapagos
2013 - Février	Open Source Robotics Fundation poursuit la gestion de ROS
2013 - Aout	Willow Garage est absorbé par Suitable Technologies
2013 - Aout	Le support de PR-2 est repris par Clearpath Robotics
2013 - Septembre	Hydro Medusa (prévu)
2014 - Juillet	Indigo Igloo (EOL - Avril 2019)
2015 - Mai	Jade Turtle (EOL - Mai 2017)
2016 - Mai	Kinetic Kame (EOL - Mai 2021)
2017 - Mai	Lunar Loggerhead (EOL - Mai 2019)
2018 - Mai	Melodic Morenia (EOL - Mai 2023)
2020 - Mai	Noetic Ninjemys (EOL - Mai 2025)

Tableau 1.1 – Historique de ROS et des releases - REP 3

2017 - Dec	Ardent Apalone (EOL - Dec 2018)
2018 - June	Bouncy Bolson (EOL - June 2019)
2018 - Dec	Crystal Clemminys (EOL - Dec 2019)
2019 - Mai	Dashing Diademata (EOL - Mai 2021)
2019 - Nov	Eloquent Elusor (EOL - Mai 2020)
2020 - May	Foxy Fitzroy (EOL - Mai 2023)
2021 - May	Galactic Geochelone (EOL - Nov 2022)
2022 - May	Humble Hawksbill (EOL - Mai 2027)
2020 - June	Rolling Ridley (EOL - Ongoing)

Tableau 1.2 – Historique des releases de ROS 2 - REP 2000

— **Un système d'appel de fonctions à distance interruptible**.
— **Un système de diagnostiques**.
  Il s'agit ici d'un ensemble d'outils permettant de visualiser des données (rviz), d'afficher des messages (roslog/rosout), d'enregister et de rejouer des données (rosbag), et d'analyser la structure de l'application (rosgraph)
— **Des algorithmes d'estimation de pose**.
  Des implémentations de filtre de Kalman, des systèmes de reconnaissance de pose d'objets basé sur la vision (ViSP).
— **Localisation - Cartographie - Navigation**.
  Des implémentations d'algorithmes de SLAM (Self Localization and Map building) permettent de construire une représentation 2D de l'environnement et de localiser le robot dans cet environnement.
— **Une structure de contrôle**.
  Afin de pouvoir normaliser l'interaction entre la partie haut-niveau d'une application robotique que sont la planification de mouvements, la planification de missions, avec partie matérielle des robots, le framework ros-control a été proposé.

1.3.3 Systèmes supportés

ROS-1 ne supporte officiellement que la distribution linux Ubuntu de la société Canonical. La raison principale est qu'il est très difficile de porter l'ensemble de ROS-1 sur tous les systèmes d'exploitation cela nécessiterait de la part de tous les contributeurs de faire l'effort de porter tous les codes sur plusieurs systèmes. Certains logiciels sont très spécifiques à des plateformes particulières et ont peu de sens en dehors de leur cas de figure. A l'inverse le coeur de ros constitués des paquets roscomm a été porté sur différents systèmes d'exploitations comme OS X, Windows, et des distributions embarquées de Linux (Angstrom, OpenEmbedded/Yocto).

ROS-2 était iniatlement prévu pour être supporté sur Ubuntu, Windows 10 et MacOSX, mais il a été décidé récemment que le support de MacOS allait être moins prioritaire du à des difficultés de portage et une faible utilisation.

Les releases de ROS-1 sont spécifiées par la REP (ROS Enhancement Proposal) : `http://www.ros.org/reps/rep-0003.html`. Les releases de ROS-2 sont spécifiées par la REP (ROS Enhancement Proposal) : `http://www.ros.org/reps/rep-2000.html`.

Ceci donne pour les dernières releases :

— Noetic Ninjemys (Mai 2020 - Mai 2025)
  — Ubuntu Focal Fossa (20.04 LTS)
  — Debian Buster
  — Fedora 32
  — C++14, Boost 1.71/1.69/1.67, Lisp SBCL 1.4.16, Python 3.8, CMake 3.16.3/3.13.4/3.17
  — Ogre3D 1.9.x, Gazebo 9.0.0, PCL 1.10/1.9.1, OpenCV 4.2/3.2, Qt 5.13.2/5.12.5/5.11.3, PyQt5
— Melodic Melusa (Mai 2019 - Mai 2021)
  — Ubuntu Xenial (18.04 LTS)
  — Ubuntu Yakkety (17.10)
  — Debian Stretch
  — Fedora 28
  — C++14, Boost 1.62/1.65.1/1.66, Lisp SBCL 1.3.14, Python 2.7, CMake 3.7.2/3.9.1/3.10.2
  — Ogre3D 1.9.x, Gazebo 8.3.0/9.0.0, PCL 1.8.0/1.8.1, OpenCV 3.2, Qt 5.7.1/5.9.5/5.10.0, PyQt5
— Lunar Loggerhead (Mai 2017 - Mai 2019)
  — Ubuntu Xenial (16.04 LTS)
  — Ubuntu Yakkety (16.10)
  — Ubuntu Zesty (17.04)
  — C++11, Boost 1.58/1.61/1.62/, Lisp SBCL 1.2.4, Python 2.7, CMake 3.5.1/3.5.2/3.7.2
  — Ogre3D 1.9.x, Gazebo 7.0/7.3.1/7.5, PCL 1.7.2/1.8.0, OpenCV 3.2, Qt 5.5.1/5.6.1/5.7.1, PyQt5
— Kinetic Kame (Mai 2016 - Mai 2021)
  — Ubuntu Wily (15.10)
  — Ubuntu Xenial (16.04 LTS)
  — C++11, Boost 1.55, Lisp SBCL 1.2.4, Python 2.7, CMake 3.0.2
  — Ogre3D 1.9.x, Gazebo 7, PCL 1.7.x, OpenCV 3.1.x, Qt 5.3.x, PyQt5

et pour ROS2 :

— Humble (Mai 2022 - May 2027)
  — Ubuntu Jammy (22.04 LTS), Windows 10 Tier 1
  — Ubuntu Focal (20.04 LTS), Debian Bullseye, macOS Tier 3
  — C++17, Python 3.6, CMake 3.22.1
  — Ogre3D 1.12.1, Gazebo 11, OpenCV 4.5.4, Qt 5.15.3,
  — Cyclone DDS 0.9.x, Fast-DDS 2.6.x, Connext DDS 6.0.1, Gurum DDS 2.7.x

1.4 Exemple : Asservissement visuel sur TurtleBot 2

Afin d'illustrer le gain à utiliser ROS considérons l'application illustrée dans la Fig.1.4. On souhaite utiliser un robot Turtlebot 2 équipé d'une Kinect et d'une base mobile Kobuki. Le but est de contrôler le robot de telle sorte à ce qu'il reconnaisse un objet par sa couleur et qu'il puisse génére une commande à la base mobile de telle sorte à ce que l'objet soit centré dans le plan image de la caméra du robot. Pour cela on doit extraire une image I du robot et générer un torseur cinématique $\begin{pmatrix} v \\ w \end{pmatrix}$ pour la base mobile.

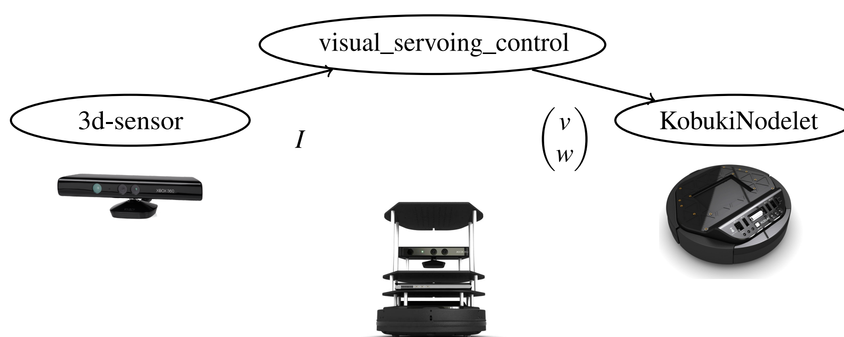


FIGURE 1.4 – Transfert d'une image I vers un node de contrôle qui calcule un torseur cinématique pour contrôler la base mobile du robot.

L'implémentation de cette structure logicielle s'effectue de la façon suivante sous ROS. En premier on utilise un noeud de calcul qui s'occupe de l'extraction de l'image d'une caméra Kinect. Dans la Fig.1.5 il est représenté par l'ellipse nommée **3d-sensor**. L'image I est alors publiée dans un topic appelé **/camera/image_color** qui peut-être vu comme un post-it. Celui-ci peut-être lu par un autre noeud de calcul appelé **visual_servoing_control** qui produit le torseur cinématique $\begin{pmatrix} v \\ w \end{pmatrix}$. Ce torseur est publié sur un autre topic appellé **/mobilebase/command/velocity**. Finalement celui-ci est lu par un troisième noeud de calcul appellé **KobukiNodelet**. Il est en charge de lire le torseur donné par rapport au centre de la base et le transformer en commande pour les moteurs afin de faire bouger la base mobile dans la direction indiquée.

Il est intéressant de noter que chacun des noeuds de calcul fournissent des fonctionnalités qui peuvent être indépendantes de l'application considérée. On peut par exemple utilise le noeud pour la Kinect sur un autre robot qui utiliserait le même capteur. Le programmateur de robot n'est ainsi pas obligé de réécrire à chaque fois un programme sensiblement identique. De même la base mobile peut-être bougée suivant une loi de commande différente de celle calculée par le noeud **visual_servoing_control**. Si un noeud est écrit de façon générique on peut donc le mutualiser à travers plusieurs applications robotiques, voir à travers plusieurs robots. Enfin chacun de ces noeuds peut-être sur un ordinateur différent des autres noeuds. Cependant il est souvent nécessaire de pouvoir changer des paramètres du noeud afin de pouvoir l'adapter à l'application visée. Par exemple, on souhaiterait pouvoir contrôler les gains du noeud **visual_servoing_control**. Il est alors possible d'utiliser la notion de paramétres pour pouvoir changer les gains proportionnels K_p et dérivés K_d du contrôleur implémenté dans le noeud **visual_servoing_control** (cf Fig.1.6).

Comme l'illustre la Fig.1.7 dans ROS, les programmes permettant à tous les noeuds d'interagir ensemble soit à travers des topics soit à travers des services sont **roscomm** (pour la communication) et **rosmaster** (pour les références entre noeuds). Un autre programme sert de serveur de paramètres **rosparam**. Le **rosmaster** enregistrent les nodes qui publient et qui souscrivent aux topics **rostopic**. Les publishers de topics sont informés de l'adresse des souscripteurs/subscribers par le **rosmaster**, et les communications entre les nodes

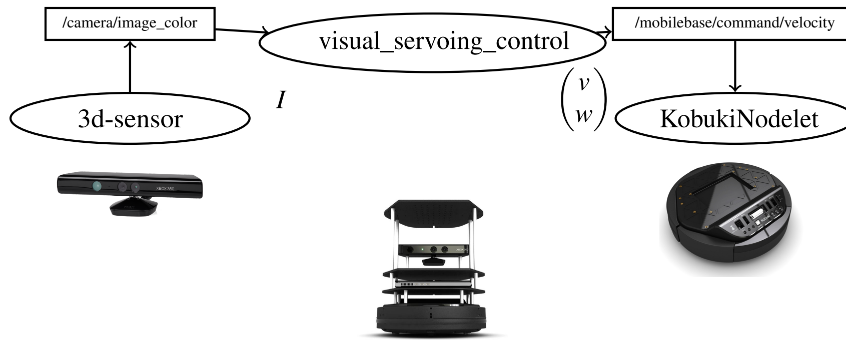


FIGURE 1.5 – Graphe d'une application ROS. Les ellipses représentent les noeuds de calcul ROS, et les rectangles les post-its où sont stockés des données disponibles pour tous les noeuds. Les flčhes indiquent les flux d'informations échangées dans l'application.

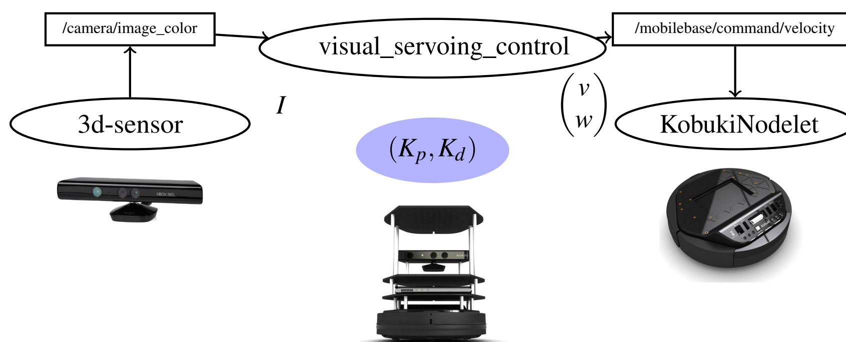


FIGURE 1.6 – Utilisation des paramètres pour moduler le comportement d'une loi de commande. Ici les paramètres K_p et K_d correspondent respectivement au gain proportionnel et au gain dérivé

par les topics s'effectuent directement. Une description technique détaillée est disponible à cette adresse : `https://wiki.ros.org/ROS/TechnicalOverview`.

Une représentation plus générale du concept est donnée dans la figure Fig.1.8.

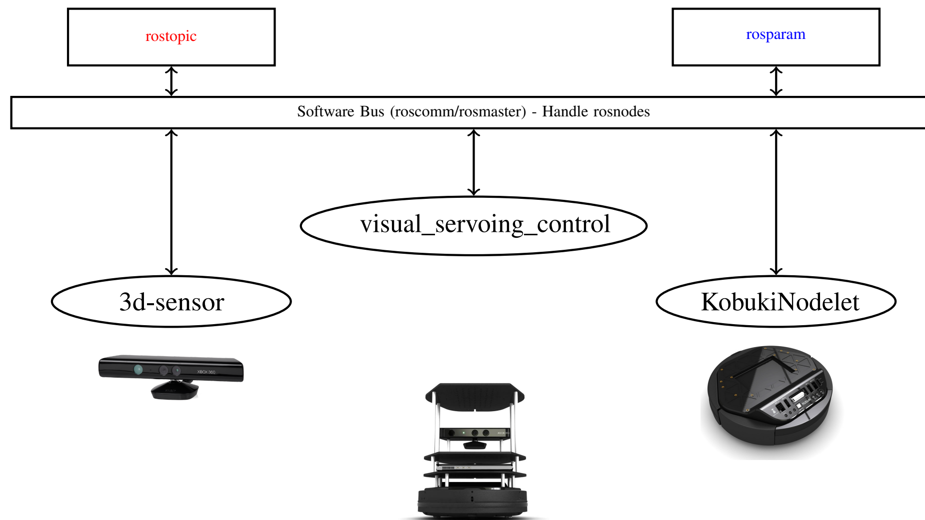


FIGURE 1.7 – Organisation interne de ROS pour implémenter les différents concepts

FIGURE 1.8 – Echange de données correspondant au mécanisme de service et de topic

FIGURE 1.9 – Echange de données correspondant au mécanisme d'action

Introduction à ROS-2

2 Graphe d'application avec ROS-2 21
2.1 Turtlesim et rqt
2.2 Node
2.3 Service
2.4 Paramètres
2.5 Actions
2.6 rqt_console
2.7 Bag : fichier de données
2.8 Launch : démarrer une application
2.9 Création de messages et de services

3 Les paquets ROS 47
3.1 Configuration de l'environnement
3.2 Structure générale des paquets ROS
3.3 Création d'un paquet
3.4 Description des paquets : le fichier **package.xml**
3.5 Compilation des paquets humble
3.6 Chaînage de workspace

4 Ecrire des nodes ROS-2 53
4.1 Topics
4.2 Services

2. Graphe d'application avec ROS-2

Dans ce chapitre nous allons introduire les changements de concepts liés à ROS-2. La différence essentielle du point de vue de l'utilisateur est la disparition du concept de master. Il n'y donc plus d'annuaire d'objets. Les noeuds se découvrent les uns les autres en suivant un protocole assez similaire à celui d'Internet.

On retrouve de nombreux concepts similaires à ROS-1 :

— les nodes
— les topics
— les services

Il y a toutefois des modifications dans la gestion des paramètres qui n'existe plus maintenant que comme des paramètres rattachés à un node.

D'une manière générale tous les concepts similaires à ROS-1 sont atteignables via l'utilisation de la comande *ros2*.

2.1 Turtlesim et rqt

Cette section correspond au didacticiel Introduction à Turtlesim. Turtlesim est un petit simulateur pour apprendre ROS-2. Il illustre les concepts de ROS-2 à un niveau très simple pour ensuite les utiliser sur un robot ou un simulateur plus complet.

rqt est une interface graphique pour ROS-2. Tout ce que fait rqt peut-être fait par des commandes en ligne. Mais il fournit un moyen d'utiliser les éléments ROS-2 plus facilement et plus amicale pour les utilisateurs.

The didacticiel suppose que vous avez déjà configuré votre environnement.

2.1.1 Utilisation de turtlesim

2.1.1.1 Vérification de l'installation de turtlesim

On peut vérifier que le paquer **turtlesim** est installé en utilisant la commande suivante :

```
ros2 pkg executables turtlesim
```

Ce qui donne :

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

L'installation du package turtlesim avec linux s'effectue avec :

```
sudo apt update

sudo apt install ros-humble-turtlesim
```

2.1.1.2 Lancer turtlesim

Pour lancer le node **turtlesim** il faut lancer la commande suivante :

```
ros2 run turtlesim turtlesim_node
```

Vous devriez voir la sortie suivante dans le terminal :

```
[INFO] [1608722845.948451886] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [1608722845.985347636] [turtlesim]: Spawning turtle [turtle1] at x=[5,544445], y=[5,544445], theta
    =[0,000000]
```

Ces informations indiquent le nom par défaut de la tortue, i.e. **turtle1**, et les coordonnées où elle apparaît.

Une fenêtre bleue avec une tortue apparaît comme celle affichée dans la figure 2.1. La tortue peut-être différente car elles sont tirées aléatoirement dans un ensemble de tortues.

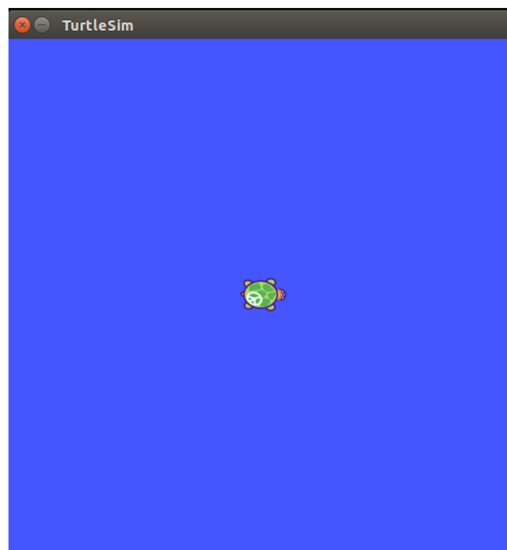


FIGURE 2.1 – Fenêtre résultat de ros2 run turtlesim turtlesim_node

2.1.1.3 Utiliser turtlesim

Pour démarrrer un node permettant de contrôler la tortue, il faut ouvrir un nouveau terminal avec l'environnement proprement configuré :

```
ros2 run turtlesim turtle_teleop_key
```

Vous devriez avoir maintenant 3 fenêtres ouvertes : un terminal où fonctionne **turtlesim_node**, un autre terminal avec **turtle_teleop_key** et la fenêtre turtlesim. Il est préférable d'arranger les fenêtres avec de façon à voir la fenêtre graphique turtlesim, mais il faut que le focus de votre souris soit sur le terminal de **turtle_teleop_key** de façon à pouvoir contrôler la tortue de **turtlesim**.

Les flèches du clavier permettent de contrôler la tortue. Elle se déplace sur l'écran utilisant son crayon ("pen") pour tracer le chemin parcouru.

> **Note 2.1** Les flèches ne font bouger la tortue que sur une distance courte. Ce comportement est motivé par le fait que réalistement on ne souhaite pas continuer à émettre une commande si, par exemple, l'opérateur perd la connection avec le robot. ■

Vous pouvez voirs les nodes, les services associés, topics et actions grâce à la commande **list** :

```
ros2 node list
ros2 topic list
ros2 service list
ros2 action list
```

Ces concepts sont développés dans d'autres didacticiels. A ce stade on ne cherche à avoir qu'une vue très générale de turtlesim, la suite utilise rqt (une interface utilisateur graphique pour ROS 2) pour regarder les services.

2.1.2 Installer rqt

Il faut ouvrir un terminal pour installer **rqt** et ses plugins. Sous linux il suffit de faire :

```
sudo apt update
sudo apt install ros-humble-rqt*
```

Pour démarrer rqt :

```
rqt
```

2.1.3 Utiliser rqt

La première fois où cette commande est démarrer la fenêtre peut-être vide. Il suffit de sélectionner dans Plugins > Services > Service Caller de la barre de menu tout en haut.

Note 2.2 rqt peut prendre du temps à localiser tous les plugins. Si en cliquant sur **Plugins** vous ne voyez pas **Services**, il faut fermer rqt, et entrer la commande **rqt –force-discover** dans le terminal

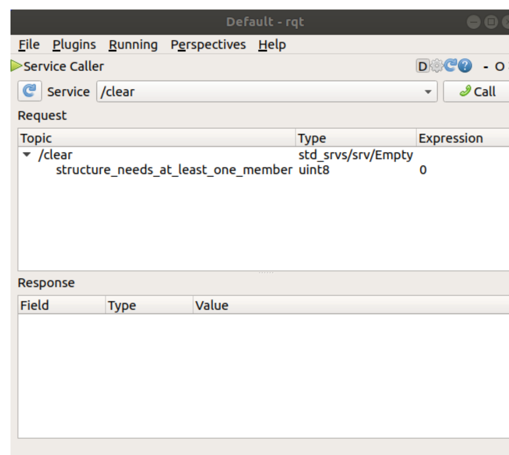


FIGURE 2.2 – Fenêtres correspondant à **rqt** avec Services

On peut utiliser le bouton **Refresh** pour être sûr d'avoir la liste de tous les services du node turtlesim.

Il faut cliquer sur la liste déroulante **Service** pour voir tous les services de turtlesim, et sélectionner le service **/spawn**.

2.1.4 Essayer le service spawn

Nous allons utiliser le service **/spawn**. Il est possible de deviner qu'à partir du nom, le service **/spawn** va créer une autre tortue dans la fenêtre graphique.

Il faut donner à la nouvelle tortue un nom unique, comme **turtle2** en double cliquant entre les quotes simples dans la colonne **Expression**. Cette expression correspond à la valeur de la variable **name** du service qui est de type **string**.

Il faut également entrer des coordonnées pour la tortue à créer, comme **x = 1.0** et **y = 1.0**.

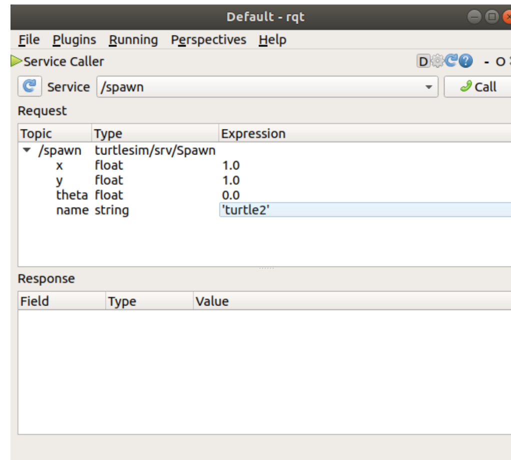


FIGURE 2.3 – Valeurs à spécifier pour l'appel au service **spawn** dans l'interface **rqt**

Note 2.3 Si vous essayez de créer une tortue avec le même nom qu'une tortue existante, par exemple le nom par défaut **turtle1**, vous obtiendrez un message d'erruer dans le terminal dans lequel est lancé **turtlesim_node** :

```
[ERROR] [turtlesim]: A turtle named [turtle1] already exists
```

Pour lancer la tortue turtle2, il faut appeller le service en cliquant sur le bouton **Call** en haut à droite de la fenêtre graphique rqt.

Vous verrez une nouvelle tortue (avec un choix aléatoire) aux coordonnées fixées en x et y.

Si vous rafraichissez la liste de services dans rqt, vous verrez qu'il y a maintenant des services liés aux nouvelles tortues **/turtle2/...** en plus de celles de **/turtle1...**.

2.1.5 Essayer le service set_pen

Essayons de changer la trace de la tortue en utilisant le service **set_pen**.

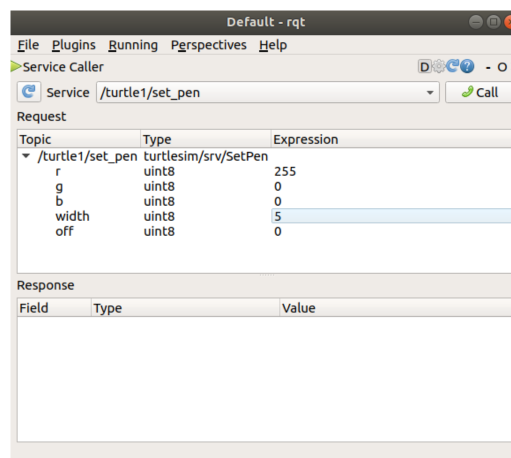


FIGURE 2.4 – Fenêtre correspondant au service **set_pen**

Les valeurs pour **r,g** et **b** sont comprises entre 0 et 255, définissent la couleur de la trace de la tortue turtle1 et **width** sa largeur.

Pour avoir la tortue turtle1 tracer une ligne rouge très visible, il faut changer la valeur **r** à 255 et la valeur **width** à 5. Il ne faut pas oublier d'appeller le service après avoir mise à jour les valeurs.

Il faut maintenant retourner au terminal où **turtle_teleop_node** est exécuté et appuyer sur les flèches, vous devriez voir que la trace de turtle1 a changé (cf. fig. 2.5)

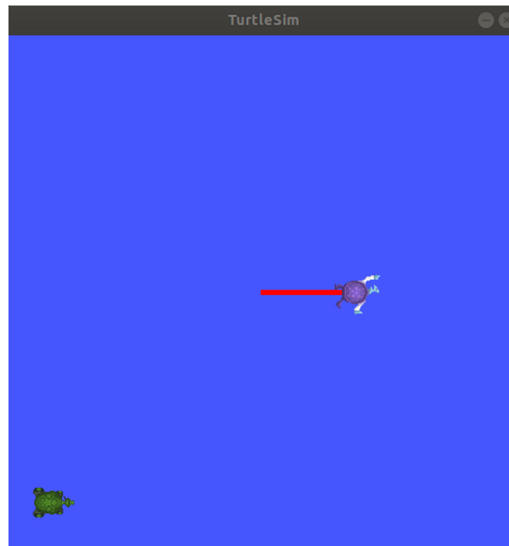


FIGURE 2.5 – Fenêtre montrant l'effet de l'appel au service **set_pen**

Vous avez probablement noté qu'il n'est pas possible de bouger turtle2. Ceci peut-être accompli en redirigeant le topic **cmd_vel** d'un node de type turtle_teleop_key vers la tortue turtle2.

2.1.6 Redirection

Dans un nouveau terminal, il faut lancer la commande suivante :

```
ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel
```

Il est maintenant possible de bouger la tortue turtle2 quand ce terminal a le focus de la souris, et la tortue turtle1 quand c'est l'autre terminal qui est actif.

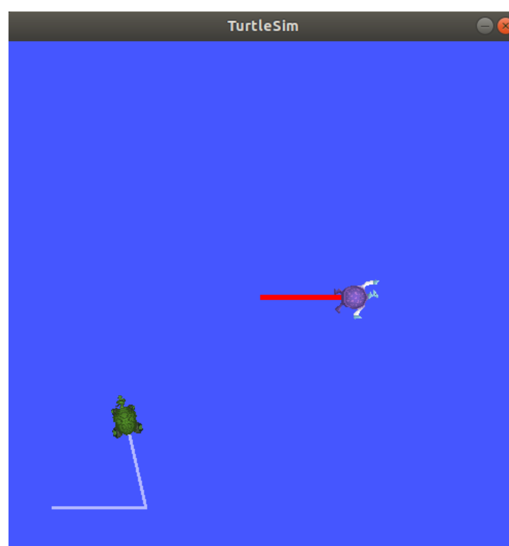


FIGURE 2.6 – Faire bouger la deuxième tortue grâche à la redirection

2.1.7 Arrêt de turtlesim

Pour arrêter la simulation, il suffit de faire la combinaison de touches **Ctrl + C** dans le terminal de **turtlesim_node** et utiliser la touche **q** dans le terminal teleop.

2.2 Node

2.2.1 Définition d'un node

Comme pour ROS-1 on peut définir un node comme étant un processsus associé à l'exécution d'un fichier exécutable dans un paquet ROS.

2.2.2 ros2 run

Pour lancer ce processus on peut utiliser la commande **ros2 run** :

```
ros2 run <package\_name> <executable\_name>
```

Par exemple pour démarrer turtlesim, il faut ouvrir un terminal et entrer la commande suivante :

```
ros2 run turtlesim turtlesim_node
```

La fenêtre graphique va s'ouvrir comme vu dans le didacticiel précédent.
Ici le nom du package est **turtlesim** et le nom de l'éxecutable **turlesim_node**.
On peut donc voir les exécutables dans un paquet ROS, ici `turtlesim`, avec :

```
ros2 pkg executables turtlesim
```

Ce qui donne :

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

Pour obtenir la liste des nodes actifs on peut utiliser la commande **ros2 node list**.

2.2.3 ros2 node list

La commande **ros2 node list** va vous montrer les noms de tous les nodes actifs. C'est utile lorsque vous souhaitez interagir avec un node, ou si vous souhaitez savoir quels sont les nodes dans un système qui en exécutent beaucoup.

En ouvrant un autre terminal et avec turtlesim toujours actif dans une autre fenêtre, il faut entrer la commande :

```
ros2 node list
```

Ce qui donne :

```
/turtlesim
```

Dans un autre terminal, taper :

```
ros2 run turtlesim turtle_teleop_key
```

Ici, le système cherche le paquet **turtlesim** et l'exécutable **turtle_teleop_key**.

En retournant dans le terminal où vous avez tapé **ros2 node list**, relancez la même commande. On voit maintenant le nom des deux nodes actifs :

```
/turtlesim
/teleop_turtle
```

2.2.4 Redirection

La redirection permet de changer des propriétés du node comme le nom d'un topic, le nom d'un service à des valeurs particuliéres. Dans le didacticiel précédent, la redirection a été utilisé pour que **turtle_teleop_key** puisse contrôler la tortue souhaitée.

Essayons maintenant de changer le nom de notre node **/turtlesim**. Dans un nouveau terminal, il faut lancer la commande suivante :

```
ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Comme on exécute une deuxième fois le node turtlesim_node, une autre fenêtre graphique s'ouvre. Toutefois, en retournant sur le terminal où on a exécutée la commande **ros2 node list**, on peut voir les trois noms suivants :

```
  /my\_turtle
  /turtlesim
  /teleop\_turtle
```

2.2.5 ros2 node info

Maintenant que vous connaissez les noms de vos nodes, vous pouvez avoir accés à plus d'information à leur propos avec :

```
ros2 node info <node_name>
```

Pour obtenir des informations sur votre dernier node **my_turtle** :

```
ros2 node info /my_turtle
```

ce qui done :

```
There are 2 nodes in the graph with the exact name "/turtlesim". You are seeing information about only one of them.
/my_turtle
  Subscribers:
    /my_turtle/cmd_vel: geometry_msgs/msg/Twist
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /my_turtle/color_sensor: turtlesim/msg/Color
    /my_turtle/pose: turtlesim/msg/Pose
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /my_turtle/set_pen: turtlesim/srv/SetPen
    /my_turtle/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /my_turtle/teleport_relative: turtlesim/srv/TeleportRelative
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:
    /my_turtle/rotate_absolute: turtlesim/action/RotateAbsolute
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

Tutoriel associé : Tutorial Understanding ROS Nodes : https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Nodes.html

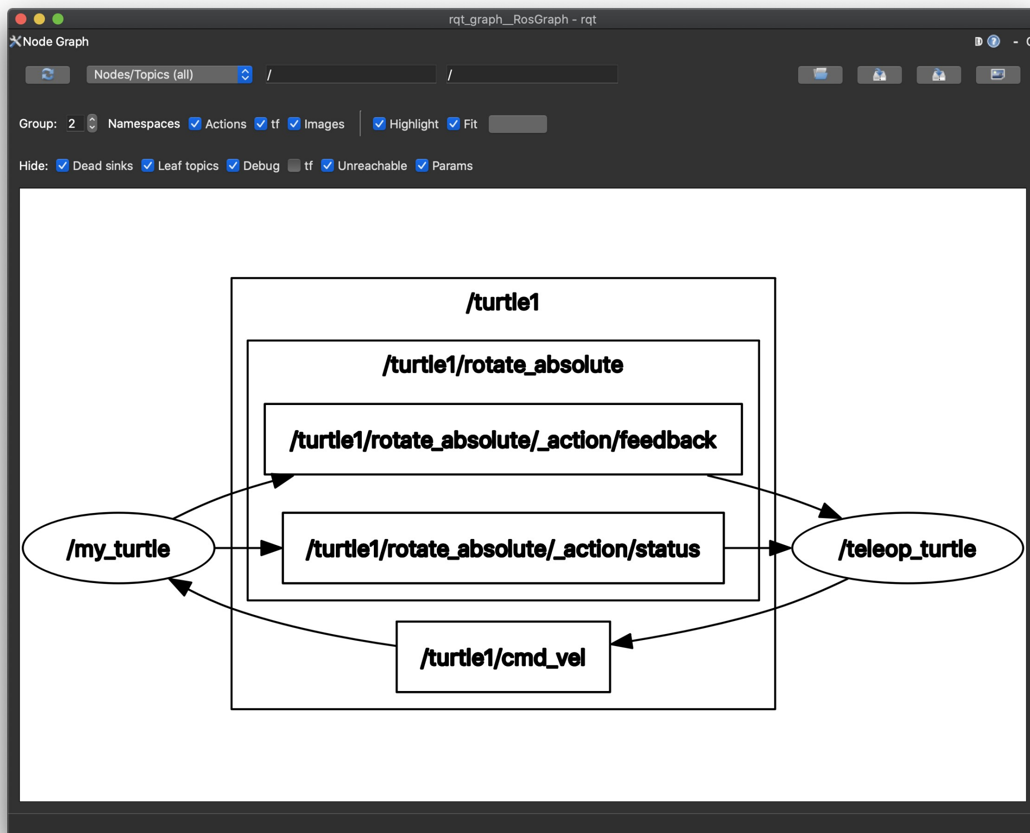


FIGURE 2.7 – Graphe de l'application **turtlesim** et **turtle_teleop_key**

2.2.6 rqt_graph

Pour visualiser le graphe de l'application on peut lancer la commande suivante :

```
rqt_graph
```

Il est également possible d'ouvrir rqt_graph en ouvrant **rqt** et en sélectionnant **Plugins** > **Introspection** > **Node Graph**.

Il est possible de voir les nodes et les topics comme dans la figure fig. 2.7. On y voit deux actions à la périphérie du graphe. Lorsque vous passez votre souris sur les topics au centre, des couleurs apparaissent.

Le graphe montre comment le node **/turtlesim** et le node **teleop_turtle** communiquent ensemble à travers un topic. Le node **teleop_turtle** publie des données (les pressions sur le clavier que vous effectuez pour bouger la tortue) sur le topic **/turtle1/cmd_vel**, et le node **/turtlesim** a souscris à ce topic pour recevoir des données.

Les mécanismes de mise en évidence de rqt_graph sont très utiles pour examiner des systèmes plus complexes avec beaucoup de nodes et des topics connectés de façons très différentes.

Pour démarrer le graphe de l'affichage des topics :

```
ros2 run rqt_plot rqt_plot
```

On obtient le graphe affiché dans la figure Fig.2.8.

2.2.7 Topic

Les topics sont des données publiées par des noeuds et auquelles les noeuds souscrivent. L'exécutable permettant d'avoir des informations sur les topics est **ros2 topic**.

La liste des commandes en ROS-2 est similaire à celle de ROS-1 :

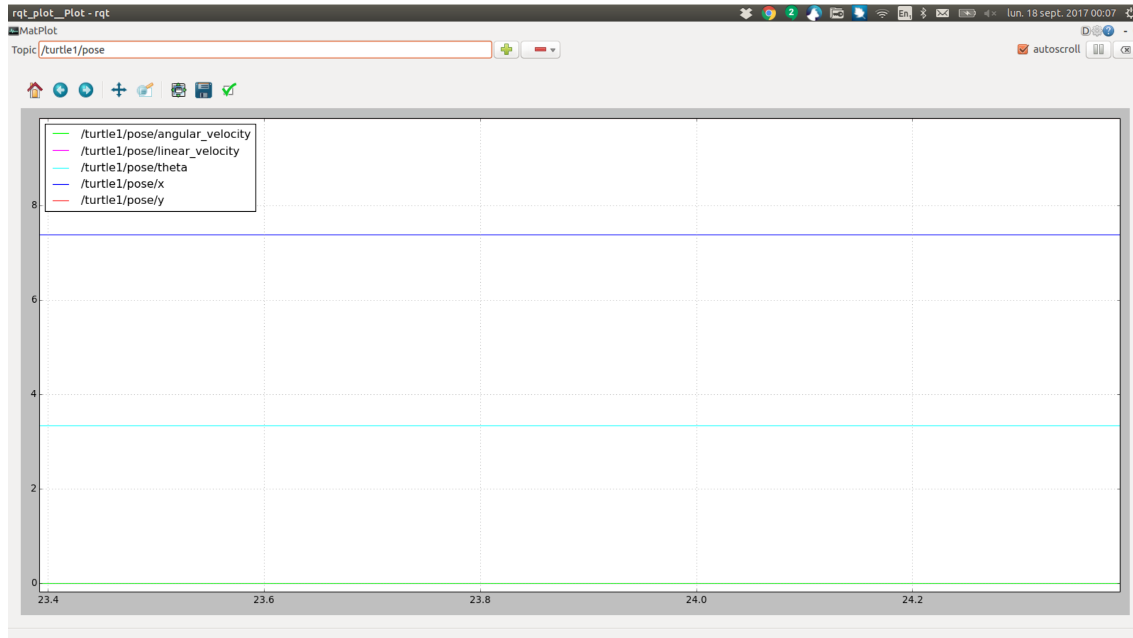


FIGURE 2.8 – Affichage de la position de la tortue dans la fenêtre

bw : Affiche la bande passante prise par le topic
echo : Affiche le contenu du topic en texte
hz : Affiche la fréquence de publication du topic
info : Fournit des informations sur un topic
list : Affiche la liste des topics actifs
pub : Publie des données sur le topic
type : Affiche le type du topic

2.2.7.1 ros2 topic list

Cette commande affiche la liste des topics.

```
ros2 topic list
```

Le résultat est le suivant :

```
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

La commande suivante

```
ros2 topic list -t
```

va retourner la même liste de topics, mais cette fois avec le type de topic ajouter et entourer de crochets.

```
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

Ces attributs, et plus particulièrement les types, montrent comment les nodes savent qu'ils échanges la même information alors que celle-ci est transmise à travers les topics.

Si vous vous demandez où se trouvent tous ces topics dans rqt_graph, vous pouvez décocher toutes les boîtes sous Hide.

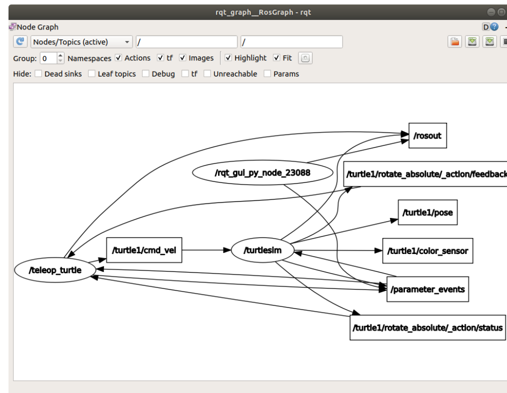


FIGURE 2.9 – Fenêtre de rqt_graph en décochant les boîtes à droite de l'option **Hide**

2.2.7.2 ros2 topic bw

On obtient la bande passante utilisée par un topic en tapant :

```
ros2 topic bw /turtle1/pose
```

On obtient alors le résultat suivant :

```
39 B/s from 52 messages
  Message size mean: 24 B min: 24 B max: 24 B
38 B/s from 52 messages
  Message size mean: 24 B min: 24 B max: 24 B
```

2.2.7.3 ros2 topic echo

Pour afficher le contenu d'un topic en fonction du message transmis on peut utiliser la commande :

```
ros2 topic echo <topic_name>
```

Par exemple, puisque nous savons que le topic **/teleop_turtle** publie des données pour **/turtlesim** sur le topic **/turtle1/cmd_vel**, utilisons la commande **echo** pour faire de l'introspection sur ce topic :

```
ros2 topic echo /turtle1/cmd\_vel
```

Cette commande ne publiera rien au départ. C'est parce que le topic attend que **/teleop_turtle** publie quelque chose.

Il faut donc mettre le focus sur le terminal où le node **turtle_teleop_key** s'exécute et utiliser les flèches pour bouger la tortue autour. En regardant le terminal où la commande **echo** s'exécute, on peut voir les données être publiées pour chaque mouvement effectué :

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

On peut alors retourner sur la fenêtre rqt_graph et déselectionner la boîte **Debug**.

Le node **/_ros2cli_26646** est le node créée par la commande **echo** (le nombre peut-être différent). On peut voir maintenant que le publisher publie des données sur le topic **/turtle1/cmd_vel**, et qu'il y a deux souscripteurs à ce topic.

2.2.7.4 ros2 topic hz

Cette commande permet d'afficher la fréquence à laquelle les données sont publiées :

```
ros2 topic hz /turtle1/pose
```

avec

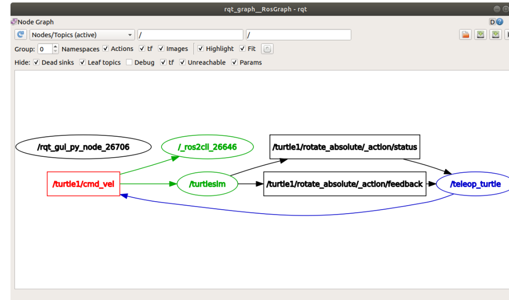


FIGURE 2.10 – Fenêtres correspondant à **rqt_graph** avec le mode Debug

```
average rate: 47.815
  min: 0.000s max: 2.621s std dev: 0.10933s window: 576
average rate: 49.036
  min: 0.000s max: 2.621s std dev: 0.10374s window: 640
```

2.2.7.5 ros2 topic info

Affiche des informations sur un topic c'est à dire le type, le nombre de publishers et le nombre de subscribers. Par exemple :

```
ros2 topic info /turtle1/pose
```

donne :

```
Type: turtlesim/msg/Pose
Publisher count: 2
Subscription count: 0
```

2.2.7.6 ros2 interface show

Les nodes envoient des données par les topics en utilisant les messages. Les publishers et les subscribers doivent envoyer et recevoir les même types de messages pour communiquer.

Les types de topic que nous avons vu précédemment après avoir exécuté **ros2 topic list -t** nous permettent de connaître le type de message a utilisé sur chaque topic. Ainsi le topic **cmd_vel** a le type :

```
geometry_msgs/msg/Twist
```

Ceci spécifie que dans le paquet **geometry_msgs** il y a un **msg** appelé **Twist**.

Il est possible d'exécuter **ros2 interface show <msg type>** sur ce type pour le connaître en détails, spécifiquement la structure des données que le message attend.

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

Le résultat est le suivant :

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

Cette information nous indique le node **/turtlesim** attend un message avec deux vecteurs, **linear** et **angular** de trois éléments chacun. Cela correspond aux données vues passant du node **/teleop_turtle** au node **/turtlesim** avec la commande **echo**.

2.2.7.7 ros2 topic type

Cette commande affiche le message (structure de données au sens de ROS) d'un topic. Par exemple :

```
ros2 topic type /turtle1/pose
```

affiche le type du topic (appelé message) :

```
turtlesim/msg/Pose
```

2.2.7.8 Afficher la structure d’un topic (message)

Pour afficher la structure d’un message il faut utiliser *ros2 interface show* :

```
ros2 interface show turtlesim/msg/Pose
```

ce qui affiche :

```
float32 x
float32 y
float32 theta

float32 linear_velocity
float32 angular_velocity
```

2.2.7.9 ros2 topic pub

Cette commande permet de publier des données sur un topic. Elle suit la structure suivante :

```
ros2 topic pub <topic_name> <msg_type> '<args>'
```

Par exemple si on revient à notre exemple on peut essayer :

```
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: 1.8}}"
```

Cette commande demande à la tortue de tourner sur elle-même à la vitesse angulaire de 1.8 rad/s. Grâce à la complétion automatique il est possible de taper uniquement le nom du topic qui étend ensuite le type du topic et la structure de la donnée à transmettre. La durée de la prise en compte de l’information est limitée, on peut utiliser l’option **-r 1** pour répéter l’émission à une fréquence d’1 Hz.

2.3 Service

Les services sont une autre façon pour les noeuds de communiquer entre eux. Les services permettent aux noeuds d’envoyer des *requêtes* et de recevoir des *réponses*. Tandis que les topics permettent aux nodes de souscrire aux flux de données

Dans la suite on supposera que le noeud **turtlesim** continue à fonctionner. Il va servir à illustrer les commandes vues dans ce paragraphe.

La commande **ros2 service** permet d’interagir facilement avec le système client/serveur de ROS appelés *services*. **ros2 service** a plusieurs commandes qui peuvent être utilisées sur les services comme le montre l’aide en ligne :

call : Appelle le service
find : Trouve une liste de services diponible pour un type spécifique
list : Liste des services actifs
type : Affiche le type d’un service

2.3.1 ros2 service list

La commande

```
ros2 service list
```

affiche la liste des services fournis par les nodes de l’application.

```
/clear
/kill
/reset
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
```

On y trouve 9 services fournis par le node **turtlesim** : clear, kill, reset, spawn, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/describe_parameters, turtlesim/get_parameters,turtlesim/list_parameters, turtlesim/set_parameters, turtlesim/set_parameters_atomically. Les 6 services relatifs aux paramètres se retrouvent avec tous les nodes.

Les deux services relatifs au node **rosout** : /rosout/get_loggers and /rosout/set_logger_level n'existent plus en ROS-2.

2.3.2 ros2 service type (service)

Il est possible d'avoir le type de service fourni en utilisant la commande :

```
ros2 service type <service>
```

Ce qui donne pour le service clear :

```
ros2 service type /clear
```

Le résultat est le suivant :

```
std_srvs/srv/Empty
```

Ce service est donc vide car il n'y a pas d'arguments d'entrée et de sortie (i.e. aucune donnée n'est envoyée lorqu'une requête est effectuée et aucune donnée n'est transmise lors de la réponse).

2.3.3 ros2 service list -t

Pour voir tous les types des services actifs en même temps, on peut ajouter l'option **–show-types**, dont l'abbréviation est **-t** à la commande **list** :

```
ros2 service list -t
```

Le résultat est le suivant :

```
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
...
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
```

2.3.4 ros2 interface show

Pour obtenir la structure de la requête et de la réponse on peut utiliser la commande **interface**. Si on souhaite afficher cette structure pour le service **turtlesim/srv/Spawn** :

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

le rérsultat est :

```
float32 x
float32 y
float32 theta
string name
---
string name
```

La requête est donc constituée de trois réels qui sont respectivement la position et l'orientation de la tortue. Le quatrième argument est le nom du node et est optionnel. Le type de la réponse est spécifié après le séparateur définit par "—".

Voyons maintenant comment faire appel à un service.

2.3.5 ros2 service call

Il est possible de faire une requête en utilisant la commande :

```
ros2 service call [service] [args]
```

Pour effacer la fenêtre graphique on va utiliser le service **/clear**. On a vu précédemment que le type est **Empty**. Pour voir sa structure on peut faire :

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

Le résultat est le suivant :

```
---
```

Comme nous aurions pu le deviner grâce au nom, ceci signifie que le service n'a ni argument d'entrée ni argument de sortie.

On peut donc l'appeller de la manière suivante :

```
ros2 service call /clear
```

Comme on peut s'y attendre les traces de la fenêtre du noeud **turtlesim** est effacé. Considérons maintenant un cas où le service a des arguments en examinant le service **spawn** :

```
ros2 service type /spawn
```

Le résultat est :

```
turtlesim/srv/Spawn
```

Ce service nous permet de lancer une autre tortue, par exemple :

```
ros2 service call /spawn turtlesim/srv/Spawn "{x:0.0, y:0.0, theta: 0.0, name: "my_turtle"}"
```

On trouve maintenant une deuxième tortue dans la fenêtre comme représenter dans la figure Fig.2.11.

FIGURE 2.11 – Fenêtre résultat de ros2 run call /spawn turtlesim/srv/Spawn "x :0.0, y :0.0, theta : 0.0, name : 'my_turtle'"

2.4 Paramètres

ROS-2 gère les paramètres qui permet de stocker des données. Contrairement à ROS-1 les paramètres ne sont pas centralisés mais répartis sur chacun des nodes.

Les mécanismes de synchronisation des paramètres est complètement différent des topics qui implémente un système de flux de données. Ils servent notamment à stocker : le modèle du robot (**robot_description**), des trajectoires de mouvements, des gains, et toutes informations pertinentes. Les paramètres peuvent se charger et se sauvegarder grâce au format YAML (Yet Another Language). Ils peuvent également être définis en ligne de commande. Le serveur de paramètre peut stocker des entiers, des réels, des booléens,

des dictionnaires et des listes. Les paramètres utilisent le langage YAML pour la syntaxe de ces paramètres. Dans des cas simples, YAML paraît trés naturel.

Les commandes suivantes permettent de gérer les paramètres :

delete : Efface un paramètre
describe : Montre les informations décrivant le paramètre
dump : Sauve les paramètres dans un fichier
get : Affiche la valeur d'un paramètre
list : Donne la liste des paramètres
set : Spécifie la valeur d'un paramètre

La suite des commandes suppose que les deux nodes de l'exemple turtlesim sont lancés :

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

2.4.1 ros2 param list

Il est possible d'avoir la liste des paramètres en tapant la commande :

```
ros2 param list
```

On obtient alors la liste suivante :

```
/teleop_turtle:
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
```

Chaque paramètre est dans le namespace d'un node. On trouve donc deux espaces de noms : `teleop_turtle` et `/turtlesim`.

Les 3 premiers paramètres de `turtlesim` permettent de contrôler la couleur du fond de la fenêtre du node `turtlesim`.

La commande suivante change la valeur du canal rouge de la couleur de fond :

```
ros2 param set /turtlesim background_r 150
```

Le résultat est représenté dans la figure 2.12.

On peut voir la valeur des autres paramètres. Par exemple, pour avoir la valeur du canal vert de la couleur de fond on peut utiliser :

```
ros2 param set /background_g
```

2.4.2 ros2 param dump

On peut aussi utiliser la commande suivante pour voir toutes les valeurs de paramètres du node `turtlesim` :

```
ros2 param dump /turtlesim
```

Le résultat est le fichier turtlesim.yaml :

```
turtlesim:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
    use_sim_time: false
```

Il est ensuite possible de recharger ces paramètres.

FIGURE 2.12 – Résultat de l'appel **ros2 param set /turtlesim background_r 150**

2.4.3 ros2 param load

La structure générale de la command est :

```
ros2 run <package_name> <executable_name>  --ros-args --params-file <file_name>
```

Pour charger le fichier :

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

2.5 Actions

Cette section correspond au didacticiel Introduction aux actions

2.5.1 Introduction

Les actions sont l'un des types de communications de ROS-2 et sont utilisées pour des tâches relativement longues. Elles sont constituées de trois parties : un but, un retour d'état, et un résultat.

Les actions sont construites sur les topics et les services. Leurs fonctionnalités sont similaires aux services, excepté que les actions peuvent être annulées. Elles fournissent également un retout constant sur l'état à l'opposé des services qui ne retournent qu'une seule réponse.

Les actions utilisent un modèle client-serveur, similaire au modèle publisher-subscriber. Un node "action client" envoit un but à un node serveur d'action "action server" qui accuse la réception du but et retourne un flux de retour d'état et un résultat.

2.5.2 Préparation

Il faut démarrer les deux nodes de turtlesim : **/turtlesim** et **/teleop_turtle**. Pour cela il faut démarrer un nouveau terminal et démarrer :

```
ros2 run turtlesim turtle_sim_node
```

```
ros2 run turtlesim turtle_teleop_key
```

2.5.3 Utilisation des actions

Quand le node **/teleop_turtle** est lancé, il est possible de voir le message suivant :

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

La deuxième ligne correspond aux actions. (La première ligne correspond au topic "cmd_vel" traité dans les didacticiels précédents).

On peut noter qur les lettres **G|B|V|C|D|E|R|T** forment une boîte autour de la lettre **F** sur un clavier US QWERTY. Chaque touche autour de **F** correspond à un orientation pour la tortue de la fenêtre graphique. Par exemple, la lettre **E** fait tourner la tortue vers le coin en haut à gauche.

Il est intéressant de regarder le terminal où le node **/turtlesim** est exécuté. A chaque fois que l'on presse une des touches, un goal est envoyé au serveur d'action (action server) qui fait partir du node **/turtlesim**. Le but est de faire tourner la tortue dans une direction particulière. Un message affichant le résultat du goal s'affiche une fois que la tortue est effectuée :

```
[INFO] [turtlesim]: Rotation goal completed successfully
```

La touche **F** annule le goal en cours d'exécution.

On peut par exemple appuyer sur **C** puis presser la touche **F** avant que la tortue puisse terminer sa rotation. Dans le terminal où le node **/turtlesim** s'exécute, on peut voir le message :

```
[INFO] [turtlesim]: Rotation goal canceled
```

Non seulement il est possible pour le client d'annuler un goal (par une entrée dans teleop) mais le serveur (le node **/turtlesim**) peut également. Quand le service choisit de stopper le goal, on dit qu'il "avorte" le goal.

On peut par exemple appuyer sur **D**, puis la touche **G** avant que la première rotation ne soit terminée. Dans le terminal où le node **/turtlesim** s'exécute on peut voir le message :

```
[WARN] [turtlesim]: Rotation goal received before a previous goal finished. Aborting previous goal
```

Le serveur d'action (action server) choisit d'avorter le premier but car il en reçu un nouveau. Il aurait pu choisir une autre stratégie comme rejecter le nouveau but ou exécuter le second but après que le premier soit finit. On ne peut pas supposer que chaque serveur d'action va choisir d'annuler le but courant quand il en reçoit un nouveau.

2.5.4 ros2 node info

Pour voir les actions du node **/turtlesim**, il faut ouvrir un nouveau terminal et lancer la commande :

```
ros2 node info /turtlesim
```

Cette commande retourne la liste des subscribers, publishers, services, serveurs d'action et clients d'action pour le node **/turtlesim** :

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

On peut noter que l'action **/turtle1/rotate_absolute** pour **/turtlesim** est dans la liste **Action Servers**. Ce qui signifit que **/turtlesim** répond et fournit le retour d'état pour l'action **/turtle1/rotate_absolute**.

Le node **/teleop_turtle** a le nom **/turtle1/rotate_absolute** dans la liste **Action Clients** ce qui signifie qu'il envoit des buts pour cette action.

```
ros2 node info /teleop_turtle
```

ce qui retourne :

```
/teleop_turtle
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Service Servers:
    /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
    /teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
    /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
    /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

2.5.5 ros2 action list

Pour afficher toutes les actions dans un graphe ROS, il faut exécuter la commande :

```
ros2 action list
```

Ce qui retourne :

```
/turtle1/rotate_absolute
```

C'est la seule action dans le graphe ROS disponible pour le moment. Elle contrôle la rotation de la tortue comme nous l'avons vue précédemment. Il est également possible de savoir qu'il y a un seul action client (qui fait partie de **/teleop_turtle**) et un serveur d'action (qui fait partie de **/turtlesim**) pour cette action en utilisant la commande **ros2 node info <node_name>**.

2.5.5.1 ros2 action list -t

Les actions ont des types, de la même manière que les topics et les services. Pour trouver le type de **/turtle1/rotate_absolute** il faut lancer la commande :

```
ros2 action list -t
```

ce qui retourne :

```
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

Dans les crochets à la droite du nom de chaque action (dans ce cas **/turtle1/rotate_absolute**) est le type de l'action : **/turtlesim/action/RotateAbsolute**. Cette information est nécessaire quand il faut exécuter une action à partir de la ligne de commande ou à partir de code.

2.5.6 ros2 action info

Il est possible d'introspecter avec plus de détails l'action **/turtle1/rotate_absolute** avec la commande :

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

ce qui retourne :

```
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

Ceci nous donne de façon concise les informations obtenues avec la commande **ros2 node info** sur chacun des nodes : Le node **/teleop_turtle** un action client et le node **/turtlesim** est un serveur d'action de l'action **/turtle1/rotate_absolute**.

2.5.7 ros2 interface show

Afin d'exécuter un but d'action (action goal) il faut connaître la structure d'une action. Rappellons qu'il est possible d'identifier le type de l'action **/turtle1/rotate_absolute** lorsque l'on exécute la commande **ros2 action list -t**. Il faut pour cela entrer la commande suivante avec le type de l'action dans le terminal :

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

Ce qui retourne :

```
# 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
```

La première section de ce message au-dessus de — est la structure (type de données et nom) de la requête du but. La partie suivante est la structure du résultat. La dernière partie est la structure du retour d'état.

2.5.8 ros2 action send_goal

Il est possible d'envoyer un but d'action (action goal) à partir la ligne de commande avec la syntaxe suivante :

```
ros2 action send_goal <action_name> <action_type> <values>
```

avec **<values>** doit être au format YAML.

En gardant un oeil sur la fenêtre turtlesim, entrer la commande suivante dans un terminal :

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

La tortue doit se mettre à tourner, et on doit voir le message suivant dans le terminal :

```
Waiting for an action server to become available...
Sending goal:
   theta: 1.57

Goal accepted with ID: f8db8f44410849eaa93d3feb747dd444

Result:
  delta: -1.568000316619873

Goal finished with status: SUCCEEDED
```

Tous les buts ont un identifiant unique dans le message de retour. Il est possible également de voir le résultat, un champ avec le nom **delta**, qui est le déplacement à la position de départ.

Pour voir le retour de ce but, il suffit d'ajouter **–feedback** à la commande **ros2 action send_goal**.

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

Le terminal va retourner le message :

```
Sending goal:
   theta: -1.57

Goal accepted with ID: e6092c831f994afda92f0086f220da27

Feedback:
  remaining: -3.1268222332000732

Feedback:
  remaining: -3.1108222007751465...



Result:
  delta: 3.1200008392333984

Goal finished with status: SUCCEEDED
```

Le retour d'état s'effectue en donnant l'angle en radians qu'il reste à effectuer jusqu'à ce que le but soit atteint.

2.5.9 Résumé

Les actions sont comme des services qui permettent d'exécuter des tâches longues, fournissent un retour régulier, et sont annulables.

Un système robotique utilisent les actions pour de nombreux comportements : la navigation, l'exécution de contrôleurs, la planification de mouvements. Un but d'action peut dire à un robot d'aller vers une position. Tant que le robot navigue vers la position finale, il peut envoyer le long du chemin sa position (i.e. un retour) et finalement un message final lorsqu'il a atteint sa destination.

Turtlesim a un serveur d'action auquel un client d'action peut envoyer des buts pour orienter les tortues. Dans ce didacticiel l' action **/turtle1/rotate_absolute** est analysée pour avoir une meilleure idée de ce que sont les actions et comment elles fonctionnent.

2.6 rqt_console

rqt est une interface d'affichage qui ne fait pas uniquement de la 3D et qui se peuple avec des plugins. Elle permet de construire une interface de contrôle incrémentalement. L'exécutable permettant d'afficher les messages des noeuds de façon centralisé est **rqt_console**.

2.6.1 rqt

Démarrer *rqt* se fait simplement avec :

```
rqt
```

La première fois où cette commande est démarrer la fenêtre peut-être vide. Il suffit de sélectionner dans Plugins > Logging > Console pour obtenir l'équivalent de :

```
ros2 run rqt_console rqt_console
```

2.6.2 rqt_console

```
ros2 run rqt_console rqt_console
```

La commande permet de lancer le système d'affichage des messages d'alertes et d'erreur. Cela crée la fenêtre affichée dans la figure Fig. 2.13. Pour illustrer cet exemple on peut lancer le noeud **turtlesim** avec :

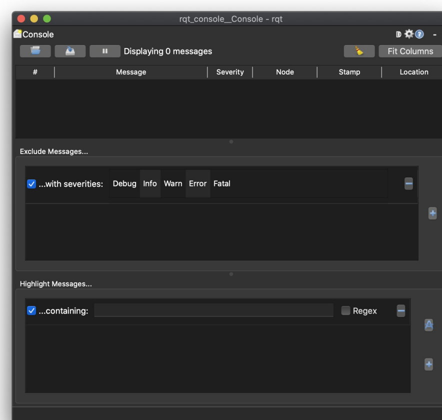


FIGURE 2.13 – Fenêtres correspondant à **rqt_console_console**

```
ros2 run turtlesim turtlesim_node
```

La position de la tortue va alors s'afficher dans la console car il s'agit d'un message de niveau INFO. Il est possible de changer le niveau des messages affichés. En le mettant par exemple à WARN, et en envoyant la commande suivant à **turtlesim** :

```
ros2 topic pub -r 1 /turtle1/cmd_vel geometry_msgs/Twist "{linear: {x: 2000.0, y: 0.0, z: 0.0}, angular: {x: 0.0,
    y: 0.0, z: 2000.0}}"
```

Elle pousse la tortue dans les coins de l'environnement et on peut voir un message affichant un warning dans la fenêtre du node **rqt_console_console** affichée dans la figure.2.14.

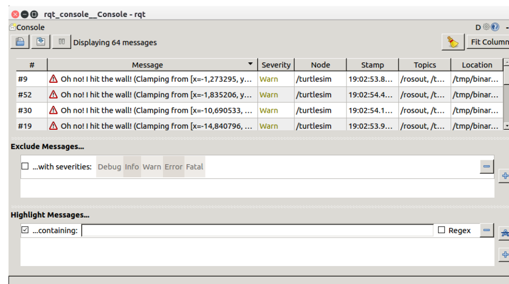


FIGURE 2.14 – Fenêtres correspondant à **rqt_console_console** lorsque la tortue s'écrase dans les murs

2.6.3 Niveaux d'enregistrement

Les niveaux d'enregistrements (logger levels) sont ordonnés par sévérité : Fatal, Error, Warn, Info, Debug.

Fatal : le message indique que le système va s'arrêter pour se protéger de tout préjudice.

Error : le message indique que le système ne va pas être endommager mais qu'il ne va pas fonctionner normalement.

Warn : le message indique une activité inhabituelle ou un résultat non idéal qui peut réprésenter un problème plus profond mais n'impact la fonctionnalité immédiatement.

Info : le message indique des mises à jour d'évènements et de statut qui permettent de visualiser que le système fonctionne correctement.

Debug : le message fournit des informations qui détaille le fonctionnement pas à pas du système.

Le niveau par défaut est **Info**. Seuls les messages d'un niveau de sévérité égal ou supérieur s'afficheront.

Normalement seuls les messages **Debug** seront cachés car ils sont les seuls à avoir un niveau de sévérité inférieurà **Info**. Par exemple, en fixant le niveau par défaut à **Warn**, on ne voit que les messages de sévérité **Warn**, **Error** et **Fatal**.

2.6.3.1 Fixé le niveau d'enregistrement par défaut

Il est possible de spécifier le niveau par défaut d'enregistrement au démarrage du node en utilisant les arguments d'entrée. Les messages de warning de sévérité **Info** ne s'affichent plus dans le terminal. C'est parce que ces messages de sévérité ont un niveau inférieur au niveau par défaut spécifié i.e. **Info**.

```
ros2 run turtlesim turtlesim_node --ros-args --log-level WARN
```

Cependant le mécanisme permettant de changer à la volée les niveaux de logs ne sont pas encore complétement supportés.

Tutoriel associé : Tutorial Using rqt console et roslaunch `https://docs.ros.org/en/humble/Tutorials/Rqt-Console/Using-Rqt-Console.html` ;

2.7 Bag : fichier de données

La commande ros2 *bag* fournit les sous commandes suivantes :

info : Affiche des informations sur le fichier de données à la console

play : Rejoue les données enregistrées dans le fichier de données .bag

record : Enregistre des données ROS dans un fichier de données au format .bag

Il est possible d'enregistrer et de rejouer des flux de données transmis par les topics. Cela s'effectue en utilisant la commande **ros2 bag**. Par exemple la ligne de commande suivante enregistre toutes les données qui passe par les topics à partir du moment où la commande est active (et si des évènements ou données sont génèrés sur les topics).

```
ros2 bag record -a
```

On peut par exemple lancer le node `turtlesim` et le noeud `teleop_key` pour envoyer des commandes.

Par défaut le nom du fichier `rosbag` est constitué de l'année, du mois et du jour et post fixé par `.bag`. Il est possible ensuite de n'enregistrer que certains topics, et de spécifier un nom de fichier. Par exemple la ligne de commande suivante n'enregistre que les topics `/turtle1/cmd_vel` et `/turtle1/pose` dans le fichier `subset.bag`.

```
ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose
```

Pour rejouer un fichier de données il suffit de faire :

```
ros2 bag play subset.bag
```

2.8 Launch : démarrer une application

2.8.1 Introduction

ros2 launch est une commande qui permet de lancer plusieurs noeuds. **ros2 launch** exécute un fichier python pour lancer plusieurs noeuds. Le fichier python doit se trouver dans un répertoire *launch*.

Dans ROS-2 les fichiers XMLs de fichier de launch ne sont plus proposés. Effectivement utiliser un fichier python permet d'avoir un meilleur contrôle sur la synchronisation du lancement des nodes de calcul.

2.8.2 Un exemple

Cet exemple se base sur l'utilisation de turtlesim. On créée tout d'abord un répertoire launch avec :

```
mkdir launch
```

Il faut créer un fichier turtlesim_mimic_launch.py en entrant la commande suivant :

```
touch launch/turtlesim_mimic_launch.py
```

Il suffit ensuite de copier coller le code python suivant :

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

Les deux premières lignes importent les modules python relatifs à la fonctionnalité de launch.

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

La description de la structure de launch est définie dans :

```python
def generate_launch_description():
   return LaunchDescription([

   ])
```

Dans la structure *LaunchDepscription* on trouve 3 nodes tous faisant parti du paquet *turtlesim*. Le but étant de lancer deux fenêtres turtlesim et un node qui mime les mouvements d'une tortue vers une autre.

Les deux premières parties lance les deux fenêtres turtlesim.

```python
Node(
    package='turtlesim',
    namespace='turtlesim1',
    executable='turtlesim_node',
    name='sim'
),
Node(
    package='turtlesim',
    namespace='turtlesim2',
    executable='turtlesim_node',
    name='sim'
    ),
```

On peut noter que la seule différence entre les deux nodes correspond à leur valeurs d'espace de noms (namespace). Un espace de nom unique permet au système de démarrer les deux fenêtres sans conflit de noms pour les topics ou les nodes.

Les deux tortues recoivent des commandes à travers le même topic et publient leurs posent sur le même topic. Sans l'utilisation d'un namespace unique, il n'y aurait moyen de distinguer les messages pour une tortue ou l'autre.

Le node final vient également du paquet `turtlesim` mais avec un exécutable différent `mimic`.

On peut le vérifier avec **rqt_graph** qui donne la figure Fig.2.15. On trouve également deux topics nommés **/turtlesim1/turtle1/pose** et **/turtlesim2/turtle1/cmd_vel**. Si on exécute :

```
ros2 topic list
```

On constate que l'on trouve deux ensembles de topics correspondants aux deux nodes de **turtlesim** et aux deux namespaces **turtlesim1** et **turtlesim2**.

Tutoriel associé : Tutorial Creating a launch file `https://index.ros.org/doc/ros2/Tutorials/Launch-Files/Creating-Launch-Files`

2.8.3 Préparer son package

La suite correspond au tutoriel Launching/monitoring multiple nodes with Launch `https://index.ros.org/doc/ros2/Tutorials/Launch-system/`

2.8.3.1 Paquet python

Pour les paquets python, la structure du paquet doit ressembler à :

```
  src/
    my_package/
      launch/
      setup.py
      setup.cfg
      package.xml
```

Pour que colcon puisse trouver les fichiers launch, il faut informer les outils de paramètrage de python qu'il y a des fichiers de launch dans le champ data_files de setup.

Donc pour le fichier setup.py il faut mettre :

```python
import os
from glob import glob
from setuptools import setup

package_name = 'my_package'

setup(
```

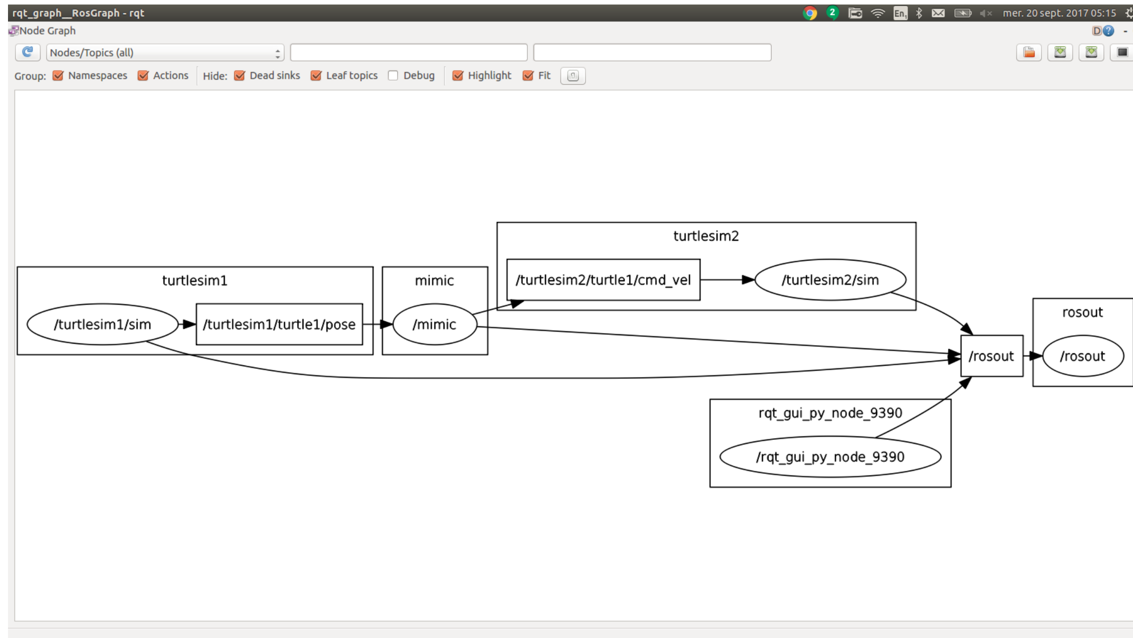


FIGURE 2.15 – Graphe correspondant au fichier turtlemimic.launch

```
8      # Other parameters ...
9      data_files=[
10         # ... Other data files
11         # Mettre tous les fichiers de launch. La ligne la plus importante est ici !
12         (os.path.join('share', package_name), glob('launch/*.launch.py'))
13     ]
14 )
```

2.8.3.2 Paquet C++

Si votre paquet est exclusivement un paquet C++, il suffit d'ajuster le fichier CMakeLists.txt en ajoutant :

```
1 # Install launch files.
2 install(DIRECTORY
3   launch
4   DESTINATION share/${PROJECT_NAME}/
5 )
```

à la fin du fichier mais avant ament_package().

2.8.3.3 Ecrire le fichier de launch

Dans le répertoire launch, il faut créer le fichier launch avec le suffixe `.launch.py`. Par exemple le nom de fichier peut-être `my_script.launch.py`.

Le suffixe `.launch.py` n'est pas spécifiquement obligatoire. Une autre option populaire est d'utiliser `_launch.py` utilisé dans l'exemple précédent. Si vous souhaitez changer le suffixe, il faut s'assurer de changer l'argument `glob()` dans le fichier `setup.py`.

2.9 Création de messages et de services

2.9.1 Introduction

C'est une bonne pratique en général de réutiliser des interfaces qui existent déjà. Mais il peut s'avérer nécessaire de créer vos propres messages ou services.

Les messages sont définis par des structures appellées **msg**, tandis que les services le sont par des les structures **srv**.

- **msg** : msg files sont de simples fichiers textes qui donnent les champs d'un message ROS. Ils sont utilisés pour génèrer le code source des messages dans des languages différents.
- **srv** : un fichier srv décrit un service. Il est composé de deux parties : une requête et une réponse.

Les fichiers msg sont stockés dans le répertoire msg d'un paquet, et les fichiers srv sont stockés dans le répertoire srv .

Les fichiers msg sont de simples fichiers textes avec un type de champ et un nom de champ par ligne. Les types de champs possibles sont :

- int8, int16, int32, int64 (plus uint *)
- float32, float64
- string
- time, duration
- other msg files
- des tableaux à taille variables ([])
- des tableaux à taille fixes ([C], avec C la taille du tableau)

Il existe un type spécial dans ROS : **Header**, l'entête contient une trace temporelle (horodatage) et des informations sur les repères de réfèrence qui sont utilisés le plus couramment en ROS. Il est fréquent que la première ligne d'un message soit **Header header**.

Voici un exemple qui utilise un entête, une chaîne de caractères, et deux autres messages :

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

srv files sont comme des fichiers msg, à part qu'ils contiennet deux parties : une requête et une réponse. Les deux parties sont séparées par une ligne '—' Voici un exemple de fichier srv :

```
int64 A
int64 B
---
int64 Sum
```

Pour la suite nous allons créer un package ROS-2 dans le workspace dev_ws avec la commande suivante :

```
cd ~/dev_ws/src
ros2 pkg create --build-type ament_cmake tutorial_interfaces
```

tutorial_interfaces est le nom du nouveau paquet. Il s'agit d'un paquet CMake, car il n'y a pour l'instant pas moyen de générer un fichier .msg ou .srv à partir d'un paquet Python pur. Il est cependant possible de créer une interface personalisée dans un paquet CMake puis de l'utiliser dans un paquet Python.

2.9.2 Création d'un msg

2.9.2.1 Format du fichier

Créons maintenant un msg dans le paquet tutorial_interfaces

```
mkdir msg
echo "int64 num" > msg/Num.msg
```

Le message précèdent ne contient qu'une ligne. Il est possible de faire des fichiers plus compliqués en ajoutant plusieurs élèments (un par ligne) comme ceci :

```
string first_name
string last_name
uint8 age
uint32 score
```

2.9.3 Création d'un srv

2.9.3.1 Format du fichier

Les fichiers srv se trouvent par convention dans le répertoire srv d'un paquet :

```
mkdir srv
```

Dans ce répertoire turtorial_interfaces/srv nous allons maintenant créer un fichier AddThreeInts.srv avec la requête et la réponse suivantes :

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

Notons que la requête et la réponse sont séparées par les trois tirets.

2.9.4 CMakeList.txt

Afin de transformer les fichiers Num.msg et AddThreeInts.srv en fichier C++, Python et autres languages, il faut modifier le fichier CMakeLists.txt du paquet.

Il suffit de rajouter les lignes suivantes :

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "srv/AddThreeInts.srv"
 )
```

2.9.5 package.xml

Il faut maintenant modifier le fichier *package.xml* pour que les trois lignes suivant aient été décommentées :

```
<build_depend>rosidl_default_generators</build_depend>

<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

2.9.6 Génération des codes

Afin de générer les codes dans les différents languages que l'on souhaite il faut compiler le paquet en utilisant la commande suivante dans le répertoire /dev_ws :

```
colcon build --packages-select tutorial_interfaces
```

Les interfaces sont maintenant découverables pour les autres paquets ROS-2.

2.9.7 Vérifier la création du fichier msg et srv

Dans un nouveau terminal il faut sourcer le fichier de configuration setup.bash de l'espace /dev_ws .

```
 . install/setup.bash
```

Il est maintenant possible d'afficher le message du paquet tutorial_interfaces en utilisant la commande suivante :

```
ros2 interface show tutorial_interfaces/msg/Num
```

retourne :

```
int64 num
```

et la commande suivante :

```
ros2 interface show tutorial_interfaces/srv/AddThreeInts
```

retourne :

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

3. Les paquets ROS

Le système de gestion des paquets ROS est probablement un élèment clé de la réussite de ROS. Il est bien plus simple que le système de gestion des paquets par Debian. La contre-partie est que ce système ne fonctionne pas au-delà de l'ecosystème ROS.

3.1 Configuration de l'environnement

3.1.1 Initialiser les variables d'environnement

Afin de pouvoir utiliser ROS il est nécessaire de configurer son environnement shell. Sous linux, bash est le shell le plus utilisé. Il suffit donc d'ajouter la ligne suivante au fichier *.bashrc*

```
source /opt/ros/humble/setup.bash
```

Pour créer son espace de travail il faut d'abord créer le répertoire associé

```
mkdir -p ~/dev_ws/src
cd ~/dev_ws/src
```

Le tutoriel associé est : Tutorial Installing and Configuring Your ROS Environment : `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html`

Une fois le fichier **setup.bash** sourcé on peut trouver deux variables d'environnement trés importantes :

```
  env | grep ROS
```

Le résultat est :

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

La variable d'environnment ROS_PACKAGE_PATH n'existe plus, elle est remplacée par AMENT_PREFIX_PATH.

3.1.2 Navigation dans le système de paquets

La commande **rospack** disponible sous ROS1 n'existe plus sous ROS-2.

Comme il arrive souvent de devoir naviguer entre des paquets systèmes et des paquets de l'environnement de développement, il existe un équivalent de **cd** qui permet de naviguer directement entre les paquets :

```
colcon_cd xacro
```

Il faut cependant avoir spécifié l'espace de travail auparavant avec :

```
cd /opt/ros/humble
colcon_cd --set
```

3.2 Structure générale des paquets ROS

Pour l'organisation des paquets ROS une bonne pratique consiste à les regrouper par cohérence fonctionnelle. Par exemple si un groupe de roboticiens travaille sur des algorithmes de contrôle, il vaut mieux les regrouper ensemble. Il est possible de faire cela sous forme d'un espace de travail (appelé également *workspace*). La figure Fig.3.1 représente une telle architecture. Dans un workspace, il faut impérativement créer un repertoire **src** dans lequel on peut créer un repertoire par paquet qui contient les fichiers d'un paquet.

Par exemple le paquet **package_1** contient au minimum deux fichiers :

- CMakeLists.txt : Le fichier indiquant comment compiler et installer le paquet
- package.xml : Le fichier ROS décrivant l'identité du paquet et ses dépendences.

Il suffit de mettre des paquets dans le même répertoire sous le répertoire **src** et sans aucun autre fichier pour faire des regroupements. Dans la figure Fig.3.1 on trouve par exemple le répertoire **meta_package_i** qui regroupe les paquets de **meta_package_i_sub_0** à **meta_package_i_sub_j**. On trouve par exemple des projets sous forme de dépôts qui regroupe de nombreux paquets légers ou qui contiennent essentiellement des fichiers de configuration. C'est par exemple le cas pour les dépots qui contiennent les paquets décrivant un robot. C'est le cas du robot TIAGO de la société PAL-Robotics https://github.com/pal-robotics/tiago_robot.

Au même niveau que **src**, la plomberie de ROS va créer trois répertoires : **build**, **log**, **install**.

Le répertoire **build** est utilisé pour la construction des paquets et à ce titre contient tous les fichiers objets. Le répertoire **install** ne contient que les fichiers qui sont explicitement installés à travers les directives d'installation spécifiées dans le fichier **CMakeLists.txt** des paquets. Le répertoire **log** contient les fichiers des sorties des opérations de compilation et d'installation.

3.3 Création d'un paquet

Pour la création d'un paquet il faut donc se placer dans le répertoire **src** de l'espace de travail choisi. Dans la suite on supposera que l'espace de travail est le répertoire **dev_ws**.

```
cd ~/dev_ws/src
```

Pour créer un nouveau package et ses dépendences il faut utiliser la commande **dev_create_pkg** en spécifiant le nom du paquet et chacune des dépendences.

```
ros2 pkg create beginner_tutorials --dependencies std_msgs rospy roscpp
ros2 pkg create [package name] [depend1] [depend2] [depend3]
```

On peut alors afficher la liste des paquets disponibles en utilisant la commande :

```
ros2 pack list
```

Les dépendences sont stockées dans le fichier **package.xml**.

```
colcon_cd beginner_tutorials
cat package.xml
<package>
...
<buildtool_depend>ament_cmake_ros</buildtool_depend>
<build_depend>roscpp</build_depend>
...
</package>
```

3.4 Description des paquets : le fichier package.xml

Jusqu'à ROS groovy les paquets devait inclure un fichier de description appelé **manifest.xml**. Depuis l'introduction de **catkin** le fichier qui doit être inclus est **package.xml**.

Il existe trois versions du format spécifié dans le champ <package>. Pour les nouveaux paquets, le format recommandé est le format 3.

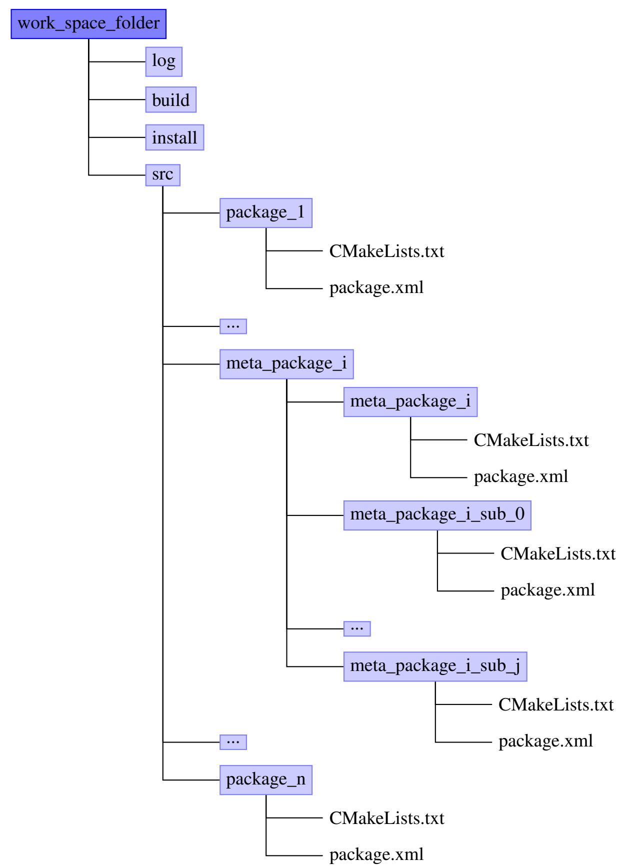


FIGURE 3.1 – Organisation de l'espace de travail : les paquets sources n'ont pas de structure hiérarchique (ou à un seul niveau lorsque les dépôts contiennent plusieurs paquets ROS).

Note 3.1 Le format 3 du fichier package.xm est défini plus en détail dans la https ://ros.org/reps/rep-0149.html. Le format 2 est définit dans la https ://ros.org/reps/rep-0140.html Le format 1 est définit dans la https ://ros.org/reps/rep-0127.html

```
<package format="3">
</package>
```

Il y 5 champs nécessaires pour que le manifeste du paquet soit complet :

— <name> : Le nom du paquet.
— <version> : Le numéro du version du paquet.
— <description> : Une description du contenu du paquet.
— <maintainer> : Le nom de la personne qui s'occupe de la maintenance du paquet ROS.
— <license> : La license sous laquelle le code est publié.

Les licenses les plus utilisées sous ROS sont Apache-2.0 (ROS2) ou BSD (ROS1) car elles permettent d'utiliser ces codes également pour des applications commerciales.

Il est important de faire attention au contexte dans lequel le code peut-être licensié notamment dans un cadre professionnel.

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

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <maintainer email="jane.doe@example.com">Jane Doe</maintainer>

  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>
```

Le site web du paquet peut-être indiqué dans le champ <url>. C'est l'endroit où on trouve généralement la documentation.

```
  <!-- Url tags are optional, but mutiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/beginner_tutorials</url> -->
```

Certains paquets servent à utiliser des logiciels tierces dans le cadre de ROS. Dans ce cas le paquet ROS encapsule ce logiciel tierce. C'est dans le champ <author> que l'on peut spécifier le nom de l'auteur de ce logiciel tierce. Il est possible de mettre plusieurs champs <author>.

```
  <!-- Author tags are optional, mutiple are allowed, one per tag -->
  <!-- Authors do not have to be maintianers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->
```

Les dépendences sont de 6 ordres :

— build dependencies (format 1 & 2) : spécifie les paquets nécessaires pour construire ce paquet. Il peut s'agir des paquets qui contiennet des en-têtes nécessaires à la compilation.
— build export dependencies (format 2) : spécifie les paquets nécessaires pour construire des librairies avec ce paquet. Notamment lorsque les en-têtes publiques de ce paquet font référence à ces paquets.
— execution dependencies (format 1 & 2) : spécifie les paquets nécessaires pour exécuter des programmes fournis par ce paquet. C'est le cas par exemple lorsque ce paquet dépend de librairies partagées.
— test dependencies (format 1 & 2) : spécifie uniquement les dépendences additionnelles pour les tests unitaires.
— build tool dependencies (format 1 & 2) : spécifie les outils de construction système nécessaires pour le construire. Typiquement le seul outil nécessaire est catkin. Dans le cas de la compilation croisée les outils sont ceux de l'architecture sur laquelle la compilation est effectuée.
— doc dependencies (format 2) : spécifie les outils pour générer la documentation.

Pour notre exemple voici les dépendences au format 1 avec ROS melodic exprimés dans le fichier **package.xml**.

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>message_generation</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use exec_depend for packages you need at runtime: -->
<!--   <exec_depend>message_runtime</exec_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Lorsque l'on souhaite groupé ensemble des paquets nous avons vu qu'il était possible de créer un **metapackage**. Ceci peut-être accompli en créant un paquet qui contient dans le tag suivant dans la section export :

```
<export>
  <metapackage />
</export>
```

Les metapackages ne peuvent avoir des dépendences d'exécution que sur les paquets qu'il regroupe.

```
<!-- The export tag contains other, unspecified, tags -->
<export>
  <!-- Other tools can request additional information be placed here -->

</export>
</package>
```

3.5 Compilation des paquets humble

La compilation de tous les paquets s'effectue avec une seule commande. Cette commande doit être exécutée dans le répertoire du workspace. C'est cette commande qui déclenche la création des répertoires **build** et **install**.

```
cd ~/dev_ws/
colcon build
```

Note 3.2 Contrairement à ROS1 **colcon build** effectue une installation automatiquement. ■

Les répertoires **build** et **install** étant créés automatiquement ils peuvent être effacés sans problème tant que le fichier **CMakeLists.txt** est cohérent avec l'environnement de programmation. Il est indispensable de le faire si le workspace a été déplacé sur le disque.

3.6 Chaînage de workspace

Lorsque la commande **colcon build** est utilisée celle-ci crée un fichier nommé **setup.bash** dans le répertoire **install**. Ce fichier une fois sourcé permet ensuite de travailler dans le workspace en initialisant correctement toutes les variables d'environnements. Un point important à noter est que ces variable d'environnements peuvent déjà avoir une valeur lors de l'appel à **colcon build** . Cette valeur est utilisée lors de la création du fichier **setup.bash**.

Ceci permet de faire du chaînage de workspace.

Par exemple on souhaite créer un workspace nommé **ping_ws** qui dépend du workspkace **pong_ws**. Avant de créer le répertoire **install** il faut s'assurer que la variable d'environnement **AMENT_PREFIX_PATH** ne contient que la valeur

```
env | grep AMENT
AMENT_PREFIX_PATH=/opt/ros/humble
```

Une fois que la commande :

```
colcon build
```

a été utilisée, on peut sourcer le fichier **pong_ws/install/setup.bash**

On a alors :

```
env | grep ROS
AMENT_PREFIX_PATH=~/pong_ws/install/cpp_pubsub:/opt/ros/humble
```

Tous les répertoires des paquets de **pong_ws** se trouvent dans la variable **AMENT_PREFIX_PATH**.

On peut donc faire maintenant un workspace **ping_ws** qui utilise **pong_ws** on peut alors après avoir sourcer le fichier **setup.bash** du workspace **pong_ws** faire soit **colcon build**. On a alors :

```
env | grep ROS
AMENT_PREFIX_PATH=~/ping_ws/install/ping_package:~/pong_ws/install/cpp_pubsub:/opt/ros/melodic/share
```

4. Ecrire des nodes ROS-2

Dans ce chapitre nous allons examiner l'écriture en C++ et en python des *nodes* implémentant les concepts vus dans le chapitre précédent. Les *nodes* est le mot dans la communauté ROS pour désigner un executable connecté au middleware ROS. Nous allons commencer par les *topics* puis examiner l'utilisation des *services*.

4.1 Topics

Dans ce paragraphe nous allons examiner l'écriture d'un node qui émet constamment un message et d'un autre node qui reçoit le message.

4.1.1 Création d'un paquet

Il faut d'abord ouvrir un terminal en s'assurant que l'environnement ROS-2 est bien sourcé de telle sorte que les commandes ROS-2 fonctionnent.

Aller ensuite dans le répertoire `dev_ws` créer précédemment. Il faut également se souvenir que les paquets doivent être créés dans le répertoire `src` et pas à la racine du répertoire.

4.1.1.1 Création d'un paquet C++

Il faut donc aller dans `dev_ws/src` et lancer la commande de création de paquet :

```
ros2 pkg create --build-type ament_cmake cpp_pubsub
```

Il faut naviguer jusqu'à `dev_ws/src/cpp_pubsub/src`. C'est dans ce répertoire que les fichiers sources des exécutables de tous les paquets CMake se trouvent.

4.1.1.2 Création d'un paquet Python

En allant dans le répertoire dans `dev_ws/src` et lancer la commande de création de paquet :

```
ros2 pkg create --build-type ament_python py_pubsub
```

Le terminal doit retourner un message confirmant la création du paquet `py_pubsub` et de tous les fichiers et répertoires nécessaires.

4.1.2 Emetteur

Dans l'émetteur nous devons faire les choses suivantes :

1. Initialiser le système ROS.
2. Avertir que le node va publier des messages *std_msgs/String* sur le topic "chatter".
3. Boucler sur la publication de messages sur "chatter" 10 fois par seconde.

4.1.2.1 C++

4.1.2.1.1 Obtenir le code

Il faut tout d'abord créer un répertoire src dans le paquet `cpp_pubsub` :

```
mkdir src
```

Ce répertoire va contenir tous les fichiers sources C++ du paquet.
Il faut ensuite créer le fichier `publisher_member_function.cpp` en faisant :

```
wget -O publisher_member_function.cpp https://raw.githubusercontent.com/ros2/examples/master/rclcpp/topics/
    minimal_publisher/member_function.cpp
```

4.1.2.1.2 Le code lui-même

Le code est le suivant :

```
#include <chrono>
#include <functional>
#include <memory>
#include <string>

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

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
* member function as a callback from the timer. */

class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer(
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }

  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + 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<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

4.1.2.1.3 Explications

L'entête ros.h

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

permet d'inclurer toutes les en-têtes ROS nécessaires d'une manière efficace et compact.

L'entête "std_msgs/msg/string.hpp" :

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

permet d'accéder à la déclaration C++ des messages std_msgs/String. Cette entête est générée automatiquement à partir du fichier **std_msgs/msg/String.msg** qui se trouve dans le paquet std_msgs.

```
  rclcpp::init(argc, argv);
```

initialise ROS. Contrairement à ROS-1 le nom n'est plus spécifié à ce niveau. On démarre ensuite un mécanisme de synchronisation en attente de la fin de la réalisation de la classe MinimalPublisher.

```
  rclcpp::spin(std::make_shared<MinimalPublisher>());
```

Une fois que le processus de la classe est terminé, on indique au système que le Node est arrêté.

```
  rclcpp::shutdown();
```

Le membre publisher_ déclare l'émetteur à la ligne 34 :

```
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
```

il est ensuite instancié à la ligne 20 :

```
  publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
```

Le type du publisher est défini par le paramètre de template std_msgs : :msg : :String, le nom du topic est topic, et la taille de la file d'attente est définie à 10.

Le timer timer_ déclaré à la ligne 33 :

```
  rclcpp::TimerBase::SharedPtr timer_;
```

est instancié avec les ligne 21 et 22 :

```
  timer_ = this->create_wall_timer(
  500ms, std::bind(&MinimalPublisher::timer_callback, this));
```

Le timer créé appelle la méthode MinimalPublisher : :timer_callback toutes les 500 *ms*.

La classe MinimalPublisher hérite de la classe rclcpp : :Node. Chaque référence à this dans le code fait référence à la classe Node. La classe est instanciée avec le constructeur de la façon suivante :

```
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
```

Cela permet de nommer le Node minimal_publisher et d'initialisé le compteur count_ à zéro.

La méthode timer_callback est celle où les données des messages sont spécifiées et où les messages sont publiés. La macro RCLCPP_INFO permet que chaque message publié soit affiché à la console.

4.1.2.1.4 Ajouter les dépendences dans le package.xml

Il faut aller dans le répertoire au-dessus du répertoire src, c'est à dire dans dev_ws/src/cpp_pubsub où les fichiers CMakeLists.txt et package.xml se trouvent.

Il faut ouvrir le fichier package.xml dans votre éditeur de texte préféré.

Il faut remplir les champs <description>, <maintener>, et <license> :

```
<description>Examples of minimal publisher/subscriber using rclcpp</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Il faut ensuite ajouter deux nouvelle lignes après la dépendence buildtool d' ament_cmake :

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

Ceci déclare la dépendence aux paquets rclrcpp et std_msgs.

Il faut ensuite s'assurer d'avoir bien sauvegarder le fichier.

4.1.2.1.5 CMakeLists.txt

Il faut ensuite ouvrir le fichier CMakeLists.txt . En-dessous de la ligne qui déclare la dépendence à ament_cmake find_package(ament_cmake REQUIRED) , il faut ajouter :

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

Ensuite, pour obtenir les exécutables et le nommer talker pour pouvoir le démarrer en utilisant ros2 run :

```
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)
```

Il faut ensuite ajouter la section d'installation pour que la commande ros2 run le trouve :

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

Il est ensuite possible de nettoyer le fichier CMakeLists.txt en enlevant les sections et commentaires inutiles de façon à ce qu'il ressemble à cela :

```
cmake_minimum_required(VERSION 3.5)
project(cpp_pubsub)

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

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)
find_package(std_msgs REQUIRED)

add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

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

ament_package()
```

Il est maintenant possible de construire le paquet, de sourcer l'environnement et de le lancer, mais il faut le node subscriber pour voir l'ensemble du système fonctionner.

4.1.2.2 Python

4.1.2.2.1 Récupérer le fichier python

Pour le code python, contrairement à ROS-1 le code python se trouve dans un répertoire qui porte le même nom que le paquet. Il faut aller dans le répertoire py_pubsub dans le paquet py_pubsub qui se trouve dans le répertoire dev_ws/src/py_pubsub/py_pubsub .

Il est possible de télécharger le script python du publisher publisher_member_function.py dans le répertoire py_pubsub et de le rendre executable :

```
wget https://raw.githubusercontent.com/ros2/examples/master/rclpy/topics/minimal_publisher/
   examples_rclpy_minimal_publisher/publisher_member_function.py
```

Le fichier se trouve maintenant à côté du fichier __init__.py .

4.1.2.2.2 Le fichier python

```python
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

4.1.2.2.3 Explications

La première ligne importe le module python pour interagir avec la partie système de ROS-2. La deuxième ligne importe la classe Node dont va dériver la classe utilisée dans ce script python.

```python
import rclpy
from rclpy.node import Node
```

La ligne suivante importe le module python qui permet d'accéder au type `String` dont le node a besoin pour construire le message à envoyer au topic.

Ces lignes représentent les dépendences du Node. Il ne faut pas oublier que ces dépendences doivent être ajoutées au fichier `package.xml`.

```python
class MinimalPublisher(Node):
```

La classe `MinimalPublisher` hérite de la classe `Node` avec cette ligne.

```python
def __init__(self):
    super().__init__('minimal_publisher')
    self.publisher_ = self.create_publisher(String, 'topic', 10)
    timer_period = 0.5  # seconds
    self.timer = self.create_timer(timer_period, self.timer_callback)
    self.i = 0
```

Dans la suite, le constructeur spécifie le nom du Node à `minimal_publisher`. Puis l'appel à `create_publisher` déclare que le Node va publier des messages de type `String` (importé par le module `std_msgs.msg`) sur le topic `topic` avec une taille de file d'attente de 10. La taille de la file d'attente est un champ de QoS (Qualité de Service) qui spécifie le nombre de messages qui sont stockés si un subscriber ne les reçoit pas assez vite.

Ensuite un timer est créé avec un fonction de rappel toutes les 500 *ms*. `self.i` est un compteur utilisé dans la fonction de rappel.

La méthode `timer_callback` créé un message avec la valeur du compteur ajoutée, et le publie sur la console avec l'appel à `get_logger().info`.

```
def timer_callback(self):
    msg = String()
    msg.data = 'Hello World: %d' % self.i
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: "%s"' % msg.data)
    self.i += 1
```

Finalement la fonction main est définie :

```
def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
```

La librarie rclpy est initialisée, le Node est créé, puis une boucle d'attente d'événements est appelée avec spin. Ceci permet au Node d'attendre les appels du timer.

4.1.2.2.4 Ajouter les dépendences

Il faut aller dans le répertoire dev_ws/src/py_pubsub où les fichiers setup.py, setup.cfg, et package.xml ont été créés. Il faut ensuite ouvrir le fichier package.xml avec un éditeur de texte, et remplir les balises : <description>, <maintainer> et <license>.

```
<description>Examples of minimal publisher/subscriber using rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Après la dépendence à l'outil de construction ament_python il faut ajouter les dépendences correspondant aux imports de python.

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Ceci déclare le besoin des paquets rclpy et std_msgs quand le code est exécuté.

4.1.2.2.5 Ajouter un point d'entrée

Il faut ouvrir le fichier setup.py. Il faut ensuite que les champs maintainer, maintainer_email, description et license correspondent aux champs du fichier package.xml :

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache License 2.0',
```

Il faut ajouter la ligne suivante dans les crochets de console_scripts du champ entry_points :

```
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
        ],
},
```

4.1.2.2.6 Vérification de setup.cfg

Le contenu de setup.cfg doit être correctement peuplé automatiquement de la façon suivante :

```
[develop]
script-dir=$base/lib/py_pubsub
[install]
install-scripts=$base/lib/py_pubsub
```

Ceci dit simplement que setuptools mets l'exécutable dans `lib`, parce que `ros2 run` les cherchera ici.

4.1.3 Souscripteur

Les étapes du souscripteur peuvent se résumer de la façon suivante :
— Initialiser le système ROS
— Souscrire au topic `topic`
— Spin, attendre que les messages arrivent
— Quand le message arrive la fonction `topic_callback()` est appellée.

4.1.3.1 C++

4.1.3.1.1 Obtenir le code

Il faut retourner dans le répertoire `dev_ws/src/cpp_pubsub/src` pour créer le souscripteur. Pour obtenir le code il suffit d'utiliser la commande suivante :

```
wget -O subscriber_member_function.cpp https://raw.githubusercontent.com/ros2/examples/master/rclcpp/topics/
     minimal_subscriber/member_function.cpp
\end{lstlistinng}


\paragraph{Le code lui-êmme}
\begin{lstlisting}[language=C++]
#include <memory>

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

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

  private:
    void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", 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<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

4.1.3.1.2 Explications

Le code du souscripteur est presque identique à celui de l'émetteur. Le node est maintenant appellé `minimal_subscriber` et le constructeur du node utilise la méthode `create_subscription` pour exécuter la méthode de rappel.

Il n'y a pas de minuteur car la méthode de rappel est appelée dés qu'un message arrive.

```
public:
  MinimalSubscriber()
  : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<std_msgs::msg::String>(
    "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
```

```
13   }
```

Rappellons qu'il faut que le nom du topic et que le type de message correspondent à la fois dans le code de l'émetteur et du souscripteur pour qu'ils puissent communiquer.

La méthode `topic_callback` reçoit la chaîne de caractères par le message de données via le topic, et l'affiche simplement dans la console grâce à la macro `RCLCPP_INFO`.

La seule déclaration de champ est celle de la souscription.

```
48 private:
49   void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
50   {
51     RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
52   }
53   rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
```

La fonction `main` est exactement la même, à part qu'il démarre le Node `MinimalSubscriber`. Pour le node émetteur l'appel à la méthode `spin` démarre le minuteur, mais le souscripteur cela signifie simplement se préparer à recevoir des messages lorsqu'ils arrivent.

Puisque ce node a les mêmes dépendences il n'y a rien à ajouter à `package.xml`.

4.1.3.1.3 CMakeLists.txt

Il faut réouvrir le fichier `CMakeLists.txt`, et ajouter l'exécutable et la cible après l'entrée de l'émetteur.

```
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

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

4.1.3.2 Python

Il faut retourner dans le répertoire `dev_ws/src/py_pubsub/py_pubsub` pour créer le node émetteur.

4.1.3.2.1 Obtenir le code

```
1 wget https://raw.githubusercontent.com/ros2/examples/master/rclpy/topics/minimal_subscriber/
      examples_rclpy_minimal_subscriber/subscriber_member_function.py
```

On a donc maintenant les fichiers suivants :

```
1 __init__.py  publisher_member_function.py  subscriber_member_function.py
```

4.1.3.2.2 Le code lui-même

Ouvrir le fichier `subscriber_member_function.py` donne le code suivant :

```
1  import rclpy
2  from rclpy.node import Node
3
4  from std_msgs.msg import String
5
6
7  class MinimalSubscriber(Node):
8
9      def __init__(self):
10         super().__init__('minimal_subscriber')
11         self.subscription = self.create_subscription(
12             String,
13             'topic',
14             self.listener_callback,
15             10)
16         self.subscription  # prevent unused variable warning
17
18     def listener_callback(self, msg):
19         self.get_logger().info('I heard: "%s"' % msg.data)
```

```
20
21
22 def main(args=None):
23     rclpy.init(args=args)
24
25     minimal_subscriber = MinimalSubscriber()
26
27     rclpy.spin(minimal_subscriber)
28
29     # Destroy the node explicitly
30     # (optional - otherwise it will be done automatically
31     # when the garbage collector destroys the node object)
32     minimal_subscriber.destroy_node()
33     rclpy.shutdown()
34
35 if __name__ == '__main__':
36     main()
```

4.1.3.2.3 Explications

Le code du souscripteur est quasiment identique à celui de l'émetteur. Le constructeur créée un souscripteur avec les mêmes arguments que l'émetteur. Rappellons qu'il faut que le nom du topic et que le type de message correspondent à la fois dans le code de l'émetteur et du souscripteur pour qu'ils puissent communiquer.

```
15 self.subscription = self.create_subscription(
16     String,
17     'topic',
18     self.listener_callback,
19     10)
```

Le constructeur du souscripteur et la fonction de rappel ne contiennent pas de timer parce que cela n'est pas utile. La fonction de rappel est appellée dés qu'un message est reçu.

La fonction de rappel simplement affiche un message sur la console, avec la donnée qu'elle a reçue. Rappellons que l'émetteur définit `msg.data = 'Hello World : %d' % self.i`

```
1 def listener_callback(self, msg):
2     self.get_logger().info('I heard: "%s"' % msg.data)
```

La définition de la fonction `main` est presque exactement la même, replaçant la création et l'appel à l'émetteur par le souscripteur.

```
1 minimal_subscriber = MinimalSubscriber()
2
3 rclpy.spin(minimal_subscriber)
```

Comme le node a les mêmes dépendences que l'émetteur, il n'y a rien à rajouter au fichier `package.xml`. Le fichier `setup.cfg` peut rester également non touché.

4.1.3.2.4 Rajouter un point d'entrée

Il faut réouvrir le fichier `setup.py` et ajouter le point d'entrée pour le souscripteur après le point d'entrée de l'émetteur. Le champ `entry_points` qui doit maintenant ressembler à

```
1 entry_points={
2         'console_scripts': [
3                 'talker = py_pubsub.publisher_member_function:main',
4                 'listener = py_pubsub.subscriber_member_function:main',
5         ],
6 },
```

Il faut s'assurer que le fichier est sauvé, et que le système pub/sub doit être prêt à l'utilisation.

4.1.4 Lancer les nodes

4.1.4.1 Dépendences

Cette partie n'est valable que sous Linux.

Les paquets rclpy , std_msgs sont normalement déjà installés sur le système ROS-2. C'est une bonne pratique de lancer rosdep à la racine de l'espace de travail dev_ws pour vérifier les dépendences avant de construire le paquet :

```
rosdep install -i --from-path src --rosdistro <distro> -y
```

4.1.4.2 Compiler

4.1.4.2.1 C++

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
colcon build --packages-select cpp_pubsub
```

4.1.4.2.2 Python

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
colcon build --packages-select py_pubsub
```

4.1.4.3 Sourcer l'environnement

Afin de pouvoir lancer les nodes il suffit de sourcer le fichier suivant :

```
source ./dev_ws/install/setup.bash
```

4.1.4.4 Lancer les nodes

4.1.4.4.1 C++

Dans un terminal il faut lancer la commande suivante :

```
ros2 run cpp_pubsub talker
```

Le terminal doit afficher les messages suivant toutes les 0.5 s :

```
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [minimal_publisher]: Publishing: "Hello World: 4"
...
```

Dans un autre terminal il faut lancer la commande :

```
ros2 run cpp_pubsub listener
```

Le souscripteur va afficher les messages suivants :

```
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber]: I heard: "Hello World: 12"
[INFO] [minimal_subscriber]: I heard: "Hello World: 13"
[INFO] [minimal_subscriber]: I heard: "Hello World: 14"
```

Pour arrêter les nodes il suffit de taper Ctrl+C .

4.1.4.4.2 Python

Dans un terminal il faut lancer la commande suivante :

```
ros2 run py_pubsub talker
```

Le terminal doit afficher les messages suivant toutes les 0.5 s :

```
[INFO] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [minimal_publisher]: Publishing: "Hello World: 4"
...
```

Dans un autre terminal il faut lancer la commande :

```
ros2 run py_pubsub listener
```

Le souscripteur va afficher les messages suivants :

```
[INFO] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [minimal_subscriber]: I heard: "Hello World: 12"
[INFO] [minimal_subscriber]: I heard: "Hello World: 13"
[INFO] [minimal_subscriber]: I heard: "Hello World: 14"
```

Pour arrêter les nodes il suffit de taper `Ctrl+C`.

4.2 Services

Dans cette partie nous allons écrire un serveur et un client pour le service **AddTwoInts.srv**. Le serveur va recevoir deux entiers et retourner la somme. Avant de commencer il faut s'assurer que le fichier **AddTwoInts.srv** existe bien dans le répertoire srv.

Il faut d'abord ouvrir un terminal en s'assurant que l'environnement ROS-2 est bien sourcé de telle sorte que les commandes ROS-2 fonctionnent.

Aller ensuite dans le répertoire `dev_ws` créer précédemment. Il faut également se souvenir que les paquets doivent être créés dans le répertoire `src` et pas à la racine du répertoire.

4.2.1 Création d'un paquet

Il faut d'abord ouvrir un terminal en s'assurant que l'environnement ROS-2 est bien sourcé de telle sorte que les commandes ROS-2 fonctionnent.

Aller ensuite dans le répertoire `dev_ws` créer précédemment. Il faut également se souvenir que les paquets doivent être créés dans le répertoire `src` et pas à la racine du répertoire.

4.2.1.1 Création d'un paquet C++

Il faut donc aller dans `dev_ws/src` et lancer la commande de création de paquet :

```
ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp example_interfaces
```

4.2.1.2 Création d'un paquet python

Il faut donc aller dans `dev_ws/src` et lancer la commande de création de paquet :

```
ros2 pkg create --build-type ament_python py_srvcli --dependencies rclpy example_interfaces
```

4.2.1.3 Explications

L'argument `–dependencies` ajoute automatiquement les dépendences correspondantes dans le fichier `package.xml`. `example_interfaces` est le paquet qui inclut le fichier .srv nécessaire pour construire les requêtes et les réponses.

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

Les deux premières lignes sont les paramètres de la requête, et en dessous la réponse.

4.2.1.4 Mise à jour de `package.xml`

Parce que la commande précédent indique les dépendences il n'y a pas besoin de les ajouter dans le fichier `package.xml`. Il faut cependant s'assurer d'ajouter la description, l'email du mainteneur et son nom, et la license.

4.2.1.4.1 C++

```
<description>C++ client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

4.2.1.4.2 Python

```
<description>Python client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

4.2.2 Serveur

Dans cette partie, nous allons écrire un serveur et un client pour le service `AddTwoInts.srv`. Le serveur va recevoir deux entiers et retourner la somme.

— La première étape est l'initialisation de la librairie client ROS-2.
— La deuxième étape est de créer le Node `add_two_ints_server`.
— La troisième étape est de créer le service.
— La quatrième étape est d'attendre les requêtes des clients.

4.2.2.1 C++

4.2.2.1.1 Le code lui même

Il faut aller dans le répertoire `dev_ws/src/cpp_srvcli/src` et créer le fichier `add_two_ints_server.cpp`. Il suffit ensuite de copier coller dans le fichier le code suivant :

```
1  #include "rclcpp/rclcpp.hpp"
2  #include "example_interfaces/srv/add_two_ints.hpp"
3
4  #include <memory>
5
6  void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
7            std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>      response)
8  {
9    response->sum = request->a + request->b;
10   RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
11                 request->a, request->b);
12   RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)response->sum);
13 }
14
15 int main(int argc, char **argv)
16 {
17   rclcpp::init(argc, argv);
18
19   std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");
20
21   rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
22     node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);
23
24   RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
25
26   rclcpp::spin(node);
27   rclcpp::shutdown();
28 }
```

Les deux premières lignes d' `include` correspondent aux dépendences du paquet.

La méthode `add` ajoute deux entiers à partir de la requête et fournie la somme en retour, et notifie la console de son statut en utilisant les logs.

```
1 void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
2          std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>      response)
3 {
4     response->sum = request->a + request->b;
5     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
6         request->a, request->b);
7     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)response->sum);
8 }
```

La fonction `main` accomplit les lignes suivantes :

— Initialisation de la librairie client de ROS-2 :

```
17     rclcpp::init(argc, argv);
```

— Créer un node nommé `add_two_ints_server` :

```
19     std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");
```

— Créer un service nommé add_two_ints pour ce node et le publie à travers le réseau. La méthode de rappel est add :

```
rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);
```

— Affiche un message de log lorsqu'il est prêt :

```
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
```

— Lance le système d'attente de requête :

```
rclcpp::spin(node);
```

4.2.2.1.2 Ajout de l'exécutable

La macro add_executable génère un exécutable que l'on peut exécuter avec ros2 run . Il faut ajouter le bloc suivant à CMakeLists.txt pour créer un exécutable nommé server :

```
add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
rclcpp example_interfaces)
```

Pour que ros2 run trouve l'exécutable, il faut ajouter les lignes suivantes à la fin du fichier, juste avant ament_package() :

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

4.2.2.2 Python

4.2.2.2.1 Obtenir le code

Dans le répertoire dev_ws/src/py_srvcli/py_srvcli , il faut créer un nouveau fichier nommé service_member_function.py en copiant le code suivant :

```
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node


class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_callback)

    def add_two_ints_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))

        return response


def main(args=None):
    rclpy.init(args=args)

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)

    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

4.2.2.2.2 Explications

Le premier `import` permet d'importer les types du module `AddTwoInts` du paquet `example_interfaces`. Le deuxième `import` importe la librairie client Python ROS-2 et spécifiquement la classe Node.

```python
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node
```

Le constructeur de la classe `MinimalService` initialise le node avec le nom `minimal_service`. Ensuite il crée un service et définit le type, nom et méthode de rappel.

```python
def __init__(self):
    super().__init__('minimal_service')
    self.srv = self.create_service(AddTwoInts, 'add_two_ints', self.add_two_ints_callback)
```

La définition de la méthode de rappel reçoit la requête et les données, effectue la somme et la retourne dans une structure de réponse.

```python
def add_two_ints_callback(self, request, response):
    response.sum = request.a + request.b
    self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))

    return response
```

Finalement la classe principale initialise la librairie client ROS-2, instancie la classe `MinimalService` pour créer le service et lance le node pour gérer les événements.

4.2.2.2.3 Ajouter un point d'entrée

Pour permettre la commande `ros2 run` de lancer le node, il faut ajouter le point d'entrée au fichier `setup.py` (localisé dans le répertoire `dev_ws/src/py_srvcli`).

Il faut ajouter la ligne suivante entre les crochets `'console_scripts' :`

```python
'service = py_srvcli.service_member_function:main',
```

4.2.3 Client

4.2.3.1 C++

4.2.3.1.1 Le code

Dans le répertoire `dev_ws/src/cpp_srvcli/src` il faut créer un fichier appellé `add_two_ints_client.cpp` et copier la suite dedans :

```cpp
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

#include <chrono>
#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  if (argc != 3) {
      RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
      return 1;
  }

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
  rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
    node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");

  auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
  request->a = atoll(argv[1]);
  request->b = atoll(argv[2]);

```

```
  while (!client->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service. Exiting.");
      return 0;
    }
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
  }

  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::FutureReturnCode::SUCCESS)
  {
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
  } else {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
  }

  rclcpp::shutdown();
  return 0;
}
```

4.2.3.1.2 Explications

De façon similaire au node de service, les lignes de code suivantes crée le node et crée le client pour ce node :

```
  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
  rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
  node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");
```

Ensuite, la requête est crée. Sa structure est définie par le fichier `.srv` mentionné précédemment :

```
  auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
  request->a = atoll(argv[1]);
  request->b = atoll(argv[2]);
```

La boucle `while` donne au client 1 seconde pour chercher les nodes de service dans le réseau. S'il n'en trouve pas alors il va contineur à attendre.

```
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
```

Alors le client envoit sa requête, et le node est en attente jusqu'à ce qu'il reçoive sa réponse ou échoue.

4.2.3.1.3 Ajouter un exécutable

Il faut retourner dans `CMakeLists.txt` pour ajouter l'exécutable et la cible pour le nouveau node. Après avoir enlever des sections et des commenaires inutilement générés automatiquement, le fichier `CMakeLists.txt` devrait ressembler à ceci :

```
cmake_minimum_required(VERSION 3.5)
project(cpp_srvcli)

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(example_interfaces REQUIRED)

add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
  rclcpp example_interfaces)

add_executable(client src/add_two_ints_client.cpp)
ament_target_dependencies(client
  rclcpp example_interfaces)

install(TARGETS
  server
  client
  DESTINATION lib/${PROJECT_NAME})

ament_package()
```

4.2.3.2 Python

4.2.3.2.1 Le code lui-même

Dans le répertoire dev_ws/src/py_srvcli/py_srvcli, il faut créer un nouveau fichier nommé client_member_function.py et y copier le fichier suivant :

```
import sys

from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node


class MinimalClientAsync(Node):

    def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddTwoInts, 'add_two_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddTwoInts.Request()

    def send_request(self):
        self.req.a = int(sys.argv[1])
        self.req.b = int(sys.argv[2])
        self.future = self.cli.call_async(self.req)


def main(args=None):
    rclpy.init(args=args)

    minimal_client = MinimalClientAsync()
    minimal_client.send_request()

    while rclpy.ok():
        rclpy.spin_once(minimal_client)
        if minimal_client.future.done():
            try:
                response = minimal_client.future.result()
            except Exception as e:
                minimal_client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                minimal_client.get_logger().info(
                    'Result of add_two_ints: for %d + %d = %d' %
                    (minimal_client.req.a, minimal_client.req.b, response.sum))
            break

    minimal_client.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
main()
```

4.2.3.2.2 Explications du code

Le seul import différent pour le client est import sys. Le code du node client utilise sys.argv pour avoir accés à la ligne de commande afin d'extraire les arguments pour la requête.

La définition du constructeur crée un client avec le même type et nom que le node de service. Le type et nom doivent être les mêmes pour le client et le service pour être capable de communiquer.

La boucle while dans le constructeur vérifie si un service avec le même nom et le même type sont disponibles chaque seconde.

En dessous du constructeur se trouve la définition de la requête, suivi de main.

4.2.3.2.3 Ajouter un point d'entrée

Comme le node de service, vous devez aussi ajouter un point d'entrée pour être capable d'exécuter le node client.

Le champ entry_points du fichier setup.py qui doit ressembler à ceci :

```
entry_points={
    'console_scripts': [
```

```
        'service = py_srvcli.service_member_function:main',
        'client = py_srvcli.client_member_function:main',
    ],
},
```

4.2.4 Lancer les nodes

4.2.4.1 Dépendences

Cette partie n'est valable que sous Linux.

Les paquets rclpy, std_msgs sont normalement déjà installés sur le système ROS-2. C'est une bonne pratique de lancer rosdep à la racine de l'espace de travail dev_ws pour vérifier les dépendences avant de construire le paquet :

```
rosdep install -i --from-path src --rosdistro <distro> -y
```

4.2.4.2 Compiler

4.2.4.2.1 C++

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
colcon build --packages-select cpp_srvcli
```

4.2.4.2.2 Python

Il suffit ensuite de lancer la commande suivante dans un terminal :

```
colcon build --packages-select py_srvcli
```

4.2.4.3 Sourcer l'environnement

Afin de pouvoir lancer les nodes il suffit de sourcer le fichier suivant :

```
source ./dev_ws/install/setup.bash
```

4.2.4.4 Lancer les nodes

4.2.4.4.1 C++

Dans un terminal il faut lancer la commande suivante :

```
ros2 run cpp_srvcli server
```

Le terminal doit retourner le message suivant et ensuite attendre :

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

Il faut ouvrir un autre terminal, sourcer les fichiers de configuration dans le répertoire dev_ws. Il faut ensuite démarrer le node client, suivi par deux entiers séparés par un espace :

```
ros2 run cpp_srvcli client 2 3
```

Si vous choisissez 2 ou 3 par exemple, le client doit recevoir une réponse comme :

```
[INFO] [rclcpp]: Sum: 5
```

En retournant dans le terminal où le node de service fonctionne, les messages de logs indiquent les requêtes et les données reçues, et la réponse retournée.

```
[INFO] [rclcpp]: Incoming request
a: 2 b: 3
[INFO] [rclcpp]: sending back response: [5]
```

Les nodes sont arrêtés avec Ctrl+C.

4.2.4.4.2 Python

Dans un terminal il faut lancer la commande suivante :

```
ros2 run py_srvcli service
```

Le node attend la requête d'un client.

Il faut alors ouvrir un autre terminal sourcer les fichiers de configuration dans le répertoire de travail dev_ws . Il faut démarrer le node client suivi de deux entiers séparés par un espace :

```
ros2 run py_srvcli client 2 3
```

Si vous choisissez 2 ou 3 par exemple, le client va recevoir une réponse qui ressemble à :

```
[INFO] [minimal_client_async]: Result of add_two_ints: for 2 + 3 = 5
```

Il faut retourner au terminal où le node de service a été lancé. Celui affiche un message correspondant à la requête reçue :

```
[INFO] [minimal_service]: Incoming request
a: 2 b: 3
```

Il faut utiliser Ctrl+C pour arrêter les nodes.

II Travaux Pratiques

5 TurtleBot3 73
5.1 Démarrage
5.2 Gazebo
5.3 Construction de cartes et navigation

5. TurtleBot3

Ce chapitre est une version résumée des tutoriaux indiqués à chaque paragraphe.

5.1 Démarrage

5.1.1 Introduction Modèle du TurtleBot3

Dans ce contexte particulier, le TurtleBot3 sera simulé dans le container du docker.

Il existe plusieurs modèles de TurtleBot3 : burger, waffle, waffle_pi. Dans ce TP nous utiliserons le modèle waffle.

Il faut donc vérifier que votre variable d'environnement TURTLEBOT3_MODEL possède la bonne valeur. Cette variable d'environnement doit être initialisée de la façon suivante :

```
export TURTLEBOT3_MODEL=waffle
```

Pour avoir cette valeur à chaque fois, il faut inclure cette ligne dans votre fichier **.bashrc**.

5.1.2 Installation

Si vous ne parvenez pas à faire fonctionner gazebo, vous pouvez installer les paquets nécessaires avec :

```
sudo apt update
sudo apt install ros-humble-turtlebot3-description ros-humble-turtlebot3-simulations ros-humble-turtlebot3-bringup
      ros-humble-turtlebot3-navigation2 ros-humble-turtlebot3 ros-humble-xacro ros-humble-turtlebot3-teleop ros-
     humble-turtlebot3-cartographer
```

Si ROS ne trouve le package turtlebot3_bringup vous pouvez l'installer avec :

```
apt install ros-humble-turtlebot3-bringup
```

5.2 Gazebo

Pour simuler le robot sous Gazebo comme représenté dans la figure Fig.5.1, il faut lancer :

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Pour obtenir une maison (ATTENTION : il ne faut pas lancer deux fois gazebo) :

```
ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```

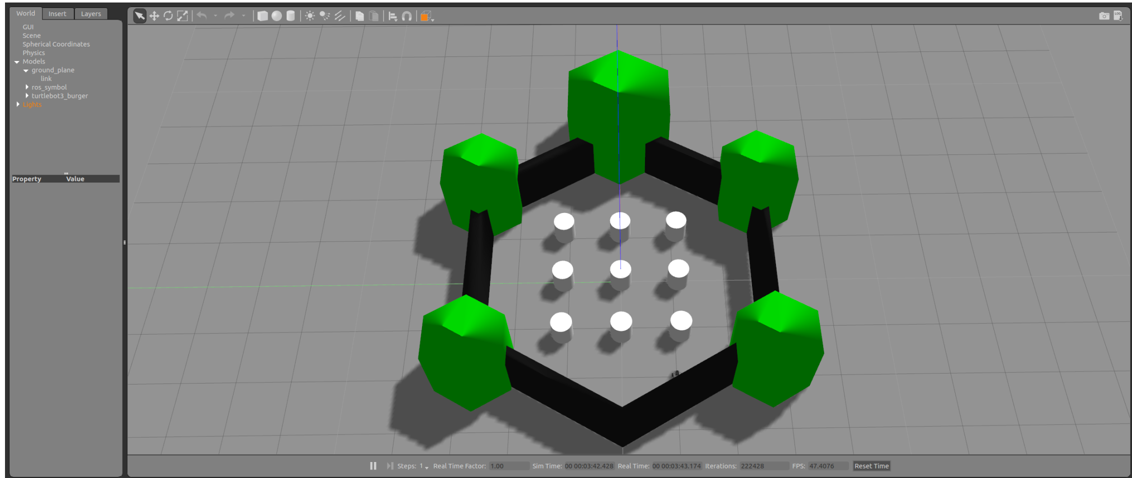


FIGURE 5.1 – Le turtlebot 3 dans une arêne hexagonale avec des piliers verts

Notes :
— Si Gazebo est lancé en premier, il est possible que sa base de données des modèles se mette à jour. Cette opération peut prendre quelques minutes.
— Assurez vous que vos variables d'environnement soient bien à jour et notamment que le fichier **setup.bash** soit bien appellé dans le fichier **.bashrc** :

```
source /opt/ros/release_name/setup.bash
```

5.2.1 Téléopérer le turtlebot

Dans un deuxième terminal vous pouvez téléopérer le robot en tapant sur le PC (et non sur le turtlebot) :

```
ros2 run turtlebot3_teleop teleop_keyboard
```

Dans le terminal vous devez voir apparaitre les lignes suivantes :

```
Control Your Turtlebot!
---------------------------
Moving around:
       w
   a   s    d
       x

w/x : increase/decrease only linear speed by 10%
a/d : increase/decrease only angular speed by 10%
space key, s : force stop

CTRL-C to quit

currently:  speed 0.2 turn 1
```

Les touches *w* et *x* permettent d'aller en avant et en arrière. Les touches *a* et *d* permettent de tourner à droite et à gauche.

Le tutorial associé est disponible ici.

5.3 Construction de cartes et navigation

5.3.1 Construction de cartes

Dans cette partie, le robot utilise une ligne de l'image de profondeur fournie par le laser pour construire une carte de l'environnement. Le tutorial associé est disponible ici.

1. En simulation il faut lancer le fichier de launch **gmapping_demo** sur un autre terminal. Remarque importante, cette commande lance un node **rviz** :

```
# From turtlebot laptop
ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
```

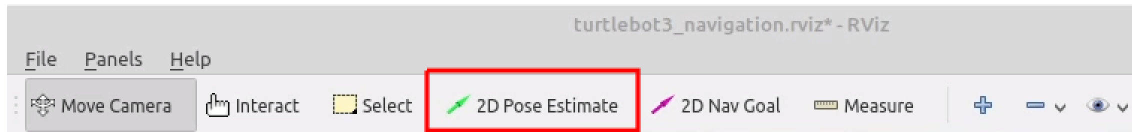


FIGURE 5.2 – Button pour l'estimation de la pose 2D

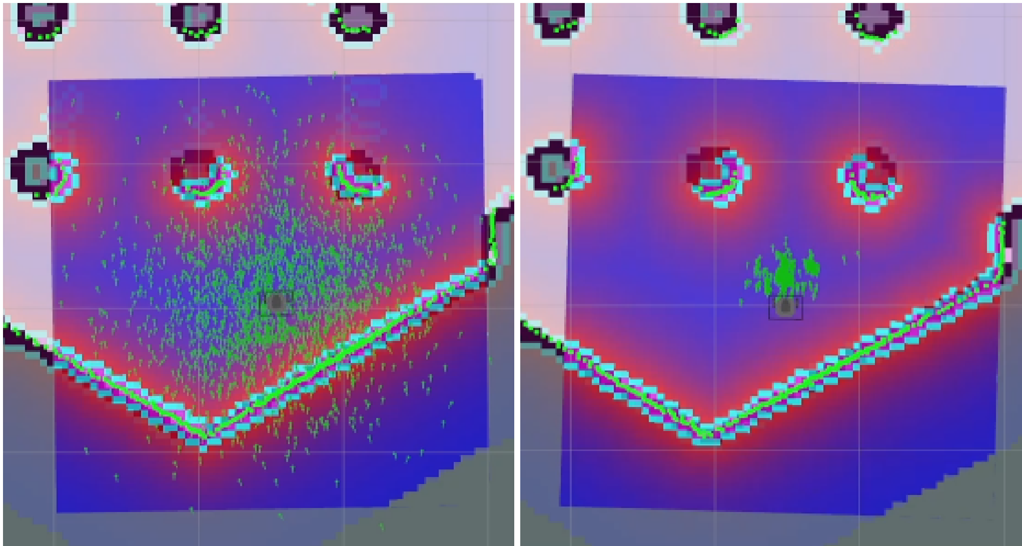


FIGURE 5.3 – AMCL partical

2. Il faut alors explorer l'environnement pour construire la carte avec les nodes de téléopération (voir paragraphe 5.2.1).
3. Une fois que la carte est suffisante, il faut la sauver en utilisant la commande suivante dans un nouveau terminal :

```
ros2 run nav2_map_server map_saver_cli -f ~/map
```

Note : Ne coupez aucun serveur pour ne pas perdre la carte !

5.3.2 Navigation

Dans cette partie, le robot utilise une carte de l'environnement. Le tutorial associé est disponible ici.

1. Sur le robot il faut lancer le fichier de launch en spécifiant la carte générée :

```
export TURTLEBOT_MAP_FILE=/tmp/my_map.yaml
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True map:=$HOME/map.yaml
```

5.3.3 RVIZ

5.3.3.1 Localisation du turtlebot

Lorsqu'il démarre le turtlebot ne sait pas où il est. Pour lui fournir sa localisation approximative sur la carte :

— Cliquer sur le bouton "2D Pose Estimate"
— Cliquer sur la carte où le TurtleBot se trouve approximativement et pointer la direction du TurtleBot.

Vous devriez voir une collection de flèches qui sont des hypothèses de positions du TurtleBot. Le scan du laser doit s'aligner approximativement avec les murs de la carte. Si l'algorithme ne converge pas, il est possible de répéter la procédure.

5.3.3.2 Teleoperation

Les opérations de téléopérations peuvent fonctionner en parallèle de la navigation. Elles sont toutefois prioritaires sur les comportements de navigation autonomes si une commande est envoyée. Souvent, il est

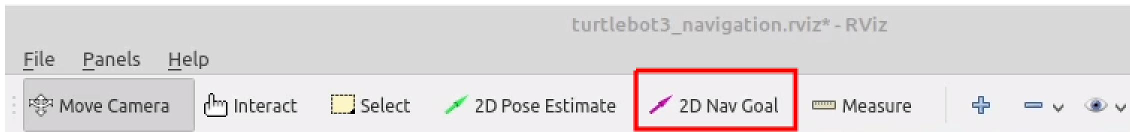


FIGURE 5.4 – Définir un but de navigation avec le bouton de RVIZ

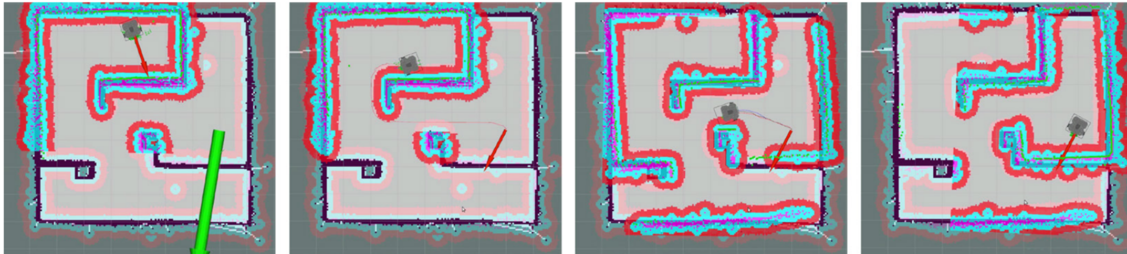


FIGURE 5.5 – Le système trouve un chemin à partir du but de navigation défini

utile de téléopérer le robot lorsque la localisation a été fournie au robot de façon à ce que l'algoritme converge vers une bonne estimation. En effet de nouvelles caractéristiques de l'environnement permettent de mieux discriminer les hypothèses.

5.3.3.3 Lancer un but de navigation

Une fois le robot localisé, il lui est possible de planifier un chemin dans l'environnement. Pour lui envoyer un but :

— Cliquer sur le boutton "2D Nav Goal" (voir Fig.5.4).
— Cliquer sur la carte où l'on souhaite voir aller le Turtlebot et pointer la direction que celui-ci doit avoir à la fin. Le système trouve par lui même le chemin à suivre. Il s'arrête si des obstacles se trouve sur son chemin. (voir Fig.5.5)

La planification peut ne pas fonctionner s'il y a des obstacles où si le but est bloqué.

III Modèles et Simulation

6 Universal Robot Description Format (URDF) 79
6.1 Capteurs - Sensors
6.2 Corps - Link
6.3 Transmission
6.4 Joint
6.5 Gazebo
6.6 model_state
6.7 model

7 Simulation 87
7.1 Introduction
7.2 Algorithme général d'un simulateur système
7.3 Algorithme général d'un moteur dynamique

8 Gazebo - Ignition 89
8.1 Introduction
8.2 SDF format
8.3 Plugin
8.4 Modèle de robot en SDF

6. Universal Robot Description Format (URDF)

Le format URDF décrit au format XML le modèle d'un robot. Le paquet urdfm implémente le parser qui analyse ce modèle. Ce chapitre se base largement sur la page ros suivante http://wiki.ros.org/urdf/XML.

6.1 Capteurs - Sensors

6.1.1 Norme courante

L'élément **<sensor>** décrit les propriétés basiques d'un capteur visuel (caméra/capteur de rayons) Voici un exemple d'un élément décrivant une caméra :

```
  <sensor name="my_camera_sensor" update_rate="20">
  <parent link="optical_frame_link_name"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <camera>
    <image width="640" height="480" hfov="1.5708" format="RGB8" near="0.01" far="50.0"/>
  </camera>
 </sensor>
```

Et voici un exemple d'un élément représentant un laser :

```
<sensor name="my_ray_sensor" update_rate="20">
   <parent link="optical_frame_link_name"/>
   <origin xyz="0 0 0" rpy="0 0 0"/>
   <ray>
     <horizontal samples="100" resolution="1" min_angle="-1.5708" max_angle="1.5708"/>
     <vertical samples="1" resolution="1" min_angle="0" max_angle="0"/>
   </ray>
</sensor>
```

Cet élément a les attributs suivants :

- **name** : *(requis) (string)*
  Le nom du capteur
- **update_rate** : *(optional) (float) (Hz)*
  La fréquence a laquelle le capteur produit les données. Si cette valeur n'est pas spécifiée les données sont générées à chaque cycle.
- **type** : *(nouveau) (required) (string)*
  Le type du capteur peut être **camera**,**ray**,**imu**,**magnetometer**,**gps**, **force_torque**,**contact**,**sonar**,**rfidtag**,**rfid**.

Les éléments cet élément sont les suivants :

- **<parent>** : *(required)*
  link*(required) (string)*
  Le nom du corps auquel le capteur est attaché.
- **<origin>** : *(optional : à defaut l'identité si ce n'est pas spécifié*
  Il s'agit de la position de l'origine du capteur optique relative au repère de référence du corps parent. Le repère du capteur optique adopte la convention suivante : z-vers le devant, x à droite, y en bas.
  - **xyz** : *(optionel : par défaut à 0*
    Représente la position suivant les axes x, y, z.
  - **rpy** : *(optionel : par défaut à l'identité*
    Représente les axes fixes roll, pitch et yaw en radians.
- **<camera>** : *(optionel)*

  - **<image>** : *(requis)*

    - **width** : *(requis) (unsigned int) (pixels)*
      Largeur de l'image en pixels
    - **height** : *(requis) (unsigned int) (pixels)*
      Hauteur de l'image en pixels
    - **format** : *(requis)(string)*
      Format de l'image. Peut-être chacune des chaînes définies dans le fichier **image_encodings.h** dans le paquet **sensors_msgs**.
    - **hfov** : *((requis)(float)(radians)*
      Largeur du champ de vue de la caméra (*rad*).
    - **near** : *(requis)(float)(m)*
      Distance la plus proche de la perception de la caméra. (*m*)

    - **far** : *(requis)(float)(m)*
      Distance la plus lointaine de la perception de la caméra. (*m*)

- **<ray>** : *(optionel)*

  - **<horizontal>** : *(optionel)*

    - **samples** : *(optionel : default 1)(unsigned int)*
      Le nombre de rayons simulés pour générer un cycle complet de perception laser.
    - **resolution** : *(optionel : default 1)(float)*
      Ce nombre est multiplié par **samples** pour déterminer le nombre de données produite. Si la résolution est inférieure à 1 les données de profondeur sont interpolées, si la résolution est supérieure à 1 les données sont moyennées.
    - **min_angle** : *(optionel : default 0)(float)(radians)* :

    - **max_angle** : *(optionel : default 0)(float)(radians)* :
      Doit être supérieur ou égale à **min_angle**.
  - **<vertical>** : *(optionel)*

    - **samples** : *(optionel : default 1)(unsigned int)*
      Le nombre de rayons simulés pour générer un cycle complet de perception laser.
    - **resolution** : *(optionel : default 1)(float)*
      Ce nombre est multiplié par **samples** pour déterminer le nombre de données produite. Si la résolution est inférieure à 1 les données de profondeur sont interpolées, si la résolution est supérieure à 1 les données sont moyennées.
    - **min_angle** : *(optionel : default 0)(float)(radians)* :

    - **max_angle** : *(optionel : default 0)(float)(radians)* :
      Doit être supérieur ou égale à **min_angle**.

6.1.2 Limitations

Peu de capteur peuvent exprimés dans le format URDF. C'est une faiblesse particuliérement problématique pour des programmes qui en ont besoin. C'est le cas pour la simulation et le contrôle. Cela démontre également que dans la pratique la plupart des développements utilisent très peu les simulations des capteurs ou le font à travers d'autres formats. C'est notamment le cas pour gazebo qui utilise le format SDF.

6.2 Corps - Link

L'élément **link** décrit un corps rigide avec une inertie et des caractéristiques visuelles.

```
16 <link name="my_link">
17   <inertial>
18     <origin xyz="0 0 0.5" rpy="0 0 0"/>
19     <mass value="1"/>
20     <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0" izz="100" />
21   </inertial>
22
23   <visual>
24     <origin xyz="0 0 0" rpy="0 0 0" />
25     <geometry>
26       <box size="1 1 1" />
27     </geometry>
28     <material name="Cyan">
29       <color rgba="0 1.0 1.0 1.0"/>
30     </material>
31   </visual>
32
33   <collision>
34     <origin xyz="0 0 0" rpy="0 0 0"/>
35     <geometry>
36       <cylinder radius="1" length="0.5"/>
37     </geometry>
38   </collision>
39 </link>
```

6.2.1 Attributs

— **name** : *(requis) (string)*
Le nom du corps lui même.

6.2.2 Elements

— **<inertial>** *(optionel)*
Les propriétés inertielles du corps.
  — **<origin>** *(optionel : l'identité si rien n'est spécifié)*
  Il s'agit de la pose du repère de référence inertiel, relative au repère du corps de référence. L'origine du repère de référence inertiel doit être au centre de gravité. Les axes du repère de référence inertiel n'ont pas forcément besoin d'être alignés avec les axes principaux de l'inertie.
    — **xyz** *(optionel : par défaut à zéro*
    Représente le décalage selon les axes x, y, z
    — **rpy** *(optionel : par défaut égale à l'identité si non spécifié*
    Représente la rotation du corps selon les axes $(roll, pitchyaw)$ en radians.
  — **<mass>** La masse du corps est fixée par l'attribut **value** de cet élément
  — **<inertia>** La matrix d'inertie 3×3 représentée dans le repère inertiel. Parce que cette matrice d'inertie est symétrique, seuls les 6 éléments diagonaux supérieurs sont donnés ici en utilisant les attributs **ixx**,**ixy**,**ixz**,**iyy**, **yz**,**izz**.

— **<visual>** *(optionel)* Les propriétés visuelles du corps. Cet élément spécifie la forme de l'objet (boite, cylindre, etc.) pour la visualisation. **Note** : il est possible d'avoir plusieurs blocks <visual> pour le même corps. Leur union définit la représentation visuelle du corps.
  — **name** *(optionel)* Spécifie un nom pour une partie de la géométrie d'un corps. Cela est particulièremen utile pour faire référence à une partie de la géométrie d'un corps.
  — **<origin>** Le repère de référence de l'élément visuel par rapport au repère de référence du corps.
    — **xyz** Représente la position de l'élément visuel dans le repère de référence du corps (en m).
    — **rpy** Représente l'orientation de l'élément visuel dans le repère de référence du corps (en *radians*).

- **<geometry>** *(requis)* La forme de l'objet visuel. Cela peut-être fait l'un des élèments suivants :
  - **box** L'attribut **size** contients la taille (longueur, larger, hauteur) de la boîte. L'origine de la boîte est le centre.
  - **cylinder** Spécifie le rayon (**<radius>**) et la hauteur (**length**) du cylindre. L'origine du cylindre est son centre.
  - **sphere** Spécifie le rayon (**<radius>**). L'origine de la sphère est son centre.
  - **mesh** Il s'agit d'un fichier décrivant une surface par des triangles. Il est spécifié par un nom de fichier (**<filename>**) et un facteur d'échelle (**<scale>**) aligné suivant l'axe de la boîte englobante de la surface. Le format recommandé est Collada (**.dae**) mais les fichiers STL (**.stl**) sont également supportés. Les fichiers de surfaces ne sont pas transférés entre les machines faisant référence au même modèle (/robot_description). Cela doit être un fichier local.
- **<material>** *(optionel)* Spécifie le matériel de l'élément visuel. Il est permis de spécifier un élément matériel à l'extérieur de l'élément **<link>** dans l'élément **<robot>**. Dans un élèment link il est possible de référencer le matériel par son nom.
  - **name** *(optionel)* Spécifie le nom du matériel.
  - **<color>** *(optionel)*
    rgba La couleur d'un matériel spécifié par un ensemble de quatre nombres représentant rouge/vert/blue/alpha, chacun dans l'intervalle $[0,1]$.
  - **<texture>** *(optionel)*
    La texture d'un matériel est spécifié par un fichier nommé **filename**.

- **<collision>** (optionel) Les propriétés de collision d'un corps. Elles peuvent être différentes des informations visuelles. Par exemple, on utilise souvent des modèles simplifés pour la collision afin de réduire les temps de calcul. Il peut y avoir des instances multiples de **collision** pour un même corps. L'union des géométries que ces instances définissent forment la représentation pour la collision de ce corps.
  - **name** (optionel)
    Spécifie un nom pour une partie géométrique du corps. Ceci est utile pour faire référene à des parties spécifiques de la géométrie d'un corps.
  - **<origin>** (optionel : défaut à l'identité si n'est pas spécifié) Le référentiel de référence de l'élément collision exprimé dans référentiel de référence du corps.
    - **xyz** (optionel : par défaut à zéro) Représente la position de l'élément de collision.
    - **rpy** (optionel : par défaut à l'identité) Représente les axes fixes roulis, tangage et lacet exprimés en radians.
  - **<geometry>** Cette partie suit la même description que pour l'élément **visuel** décrit plus haut.

6.2.3 Résolution recommandée pour les mailles

Pour la détection de collision utilisant les paquets ROS de planification de mouvements le moins de face possible est recommandé pour les mailles de collision qui sont spécifiées dans l'URDF (idéalement moins que 1000). Si possible, l'approximation des éléments pour la collision avec des primitives géométriques est encouragée.

6.2.4 Elements multiples des corps pour la collision

Il a été décidé que les fichiers URDFs ne devait pas accepter des multiples groupes d'éléments de collision pour les corps, et cela même s'il y a des applications pour cela. En effet l'URDF a été conçu pour ne représenter que les propriétés actuelles du robot, et non pas pour des algorithmes extérieurs de détection de collision. Dans l'URDF les éléments **visual** doivent être le plus précis possible, les éléments de **collision** doivent être toujours une approximation la plus proche, mais avec bien moins de triangles dans les mailles.

Si des modèles de collision moins précis sont nécessaires pour le control ou la détection de collision il est possible de placer ces mailles/géométries dans des éléments XMLs spécialisés. Par exemple, si vos contrôleurs ont besoin de représentations particulièrement simplifiées, il est possible d'ajouter la balise **<collision_checking>** après l'élément **<collision>**.

Voici un exemple :

```
40 <link name="torso">
41   <visual>
42     <origin rpy="0 0 0" xyz="0 0 0"/>
```

```
43      <geometry>
44        <mesh filename="package://robot_description/meshes/base_link.DAE"/>
45      </geometry>
46    </visual>
47    <collision>
48      <origin rpy="0 0 0" xyz="-0.065 0 0.0"/>
49      <geometry>
50        <mesh filename="package://robot_description/meshes/base_link_simple.DAE"/>
51      </geometry>
52    </collision>
53    <collision_checking>
54      <origin rpy="0 0 0" xyz="-0.065 0 0.0"/>
55      <geometry>
56        <cylinder length="0.7" radius="0.27"/>
57      </geometry>
58    </collision_checking>
59    <inertial>
60      ...
61    </inertial>
62  </link>
```

Le parseur URDF ignorera ces éléments spécialisés et un programme particulier et spécialisé peut parser le XML pour obtenir cette information.

6.3 Transmission

Un élément de transmission est une extension du modèle URDF de description des robots qui est utilisé pour décrire la relation entre un actionneur et un joint. Cela permet de modéliser des engrenages ou des mécanismes parallèles. Une transmission transforme des variables d'efforts et de flots de telle sorte que leur produit (la puissance) reste constante. Plusieurs actionneurs peuvent être liés à de multiples joints à travers une transmission complexe.

Voici un exemple d'un élément de transmission :

```
63 <transmission name="simple_trans">
64   <type>transmission_interface/SimpleTransmission</type>
65   <joint name="foo_joint">
66     <hardwareInterface>EffortJointInterface</hardwareInterface>
67   </joint>
68   <actuator name="foo_motor">
69     <mechanicalReduction>50</mechanicalReduction>
70     <hardwareInterface>EffortJointInterface</hardwareInterface>
71   </actuator>
72   </transmission>
```

6.3.1 Attributs de transmission

L'élément de transmission a un attribut :

— **name** (requis)
  Spécifie le nom unique d'une transmission.

6.3.2 Eléments de transmission

La transmission a les éléments suivants :

— **<type>** (une occurence)
  Spécifie le type de transmission.
— **<joint>** (une ou plusieurs occurences)
  Un joint auquel la transmission est connectée. Le joint est spécifié par l'attribut **name**, et les sous-éléments suivants :
  — **<hardwareInterface>** (une ou plusieurs occurences)
    Spécifie une interface matérielle supportée (par exemple **EffortJointInterface** ou **PositionJointInterface**). Notons que la valeur de cet élément doit être **EffortJointInterface** quand cette transmission est chargée dans Gazebo et **hardware_interface/EffortJointInterface** quand cette transmission est chargée dans RobotHW.
— **<actuator>** (une ou plusieurs occurences)
  Il s'agit de l'actionneur auquel la transmission est connectée. L'actionneur est spécifié son attribut **name** et les sous éléments suivants :

- **<mechanicalReduction>** (optionel)
  Spécifie la réduction mécanique de la transmission joint/actionneur. Cette balise n'est pas nécessaire pour toutes les transmissions.
- **<hardwareInterface>** (optionel) (une ou plusieurs occurences) Spécifie une interface matérielle supportée. Notons que la balise **<hardwareInterface>** ne doit être spécifiée que pour des releases précédent Indigo. L'endroit pour utiliser cette balise est la balise **<joint>**.

6.3.3 Notes de développement

Pour le moment seul le projet **ros_control** utilise ces éléments de transmissions. Développer un format de transmission plus complet qui est extensible à tous les cas est quelque chose de difficile. Une discussion est disponible ici.

6.4 Joint

6.4.1 Elément <joint>

L'élément joint décrit la cinématique et la dynamique d'un joint et spécifie les limites de sécurité. Voici un exemple d'un élément joint :

```
73  <joint name="my_joint" type="floating">
74    <origin xyz="0 0 1" rpy="0 0 3.1416"/>
75    <parent link="link1"/>
76    <child link="link2"/>
77
78    <calibration rising="0.0"/>
79    <dynamics damping="0.0" friction="0.0"/>
80    <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
81    <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_limit="0.5" />
82 </joint>
```

6.4.2 Attributs

L'élément joint a deux attributs :

- **name** (requis)
  Spécifie un nom unique pour le joint.
- **type** (requis)
  Spécifie le type de joint qui peut prendre une des valeurs suivantes :
  - *revolute* - une charnière qui tourne autour d'un axe spécifié par les attributs décrits ci-dessous et les limites inférieures et supérieures.
  - *continuous* - une charnière qui tourne autour d'un axe spécifié sans aucune limite.
  - *prismatic* - un joint coulissant le long d'un axe spécifié avec un intervalle d'utilisation limité par une valeur inférieure et une valeur supérieure.
  - *fixed* - Ce n'est pas réellement un joint car il ne bouge pas. Tous les degrées de liberté sont fixés. Ce type de joint ne requiert pas d'axe et n'a pas d'intervalle d'utilisation spécifié par une valeur supérieure et inférieure.
  - *floating* - Ce joint permet des mouvements suivant les 6 degrés de liberté.
  - *planar* - Ce joint permet des mouvements dans un plan perpendiculaire à l'axe.

6.4.3 Elements

L'élément **joint** a les sous éléments suivants :

- **<origin>** (optionel : défaut à l'identité si non spécifié)
  Il s'agit de la transformée du corps parent vers le corps enfant. Le joint est localisé à l'origine du référentiel enfant.
  - **xyz** (optionel : par défaut à zéro)
    Représente la position du joint.
  - **rpy** (optionel : par défaut au vector zéro)
    Représente la rotation autour un axe de rotation fixé : d'abord autour de l'axe de roulis (roll), puis autour de l'axe de tangage (pitch), puis celui de lacet (yaw). Tous les angles sont spécifiés en radians.

- **<parent>** *(requis)*
  Le corps parent avec l'attribut nécessaire :
  - **link** Le nom du corps qui est le parent du joint dans cet arbre.
- **<child>** *(requis)*
  Le corps enfant avec l'attribut nécessaire :
  - **link** Le nom du corps qui est l'enfant du joint dans cet arbre.

6.5 Gazebo

6.5.1 Eléments pour les corps/links

Nom	Type	Description
material	value	Le matériel de l'élément visuel
gravity	bool	Utilisation de la gravité
dampingFactor	double	La constante de décroissance de la descente exponentielle pour la vitesse d'un corps - Utilise la valeur et multiplie la vitesse précédente du corps pary $(1 - dampingFactor)$.
maxVel	double	Vitesse maximum contact d'un corps (corrigée par troncature).
minDepth	double	Profondeur minimum autorisée avant d'appliquer une impulsion de correction.
mu1,mu2	double	Coefficients de friction μ pour les directions principales de contact pour les surfaces de contacts comme définies par le moteur dynamique Open Dynamics Engine (ODE) (voir la description des paramètres dans le guide d'utilisateur d'ODE)
fdir1	string	3-tuple spécifiant la direction de mu1 dans le repère de référence local des collisions.
kp,kd	double	Coefficient de rigidité k_p et d'amortissement k_d pour les contacts des corps rigides comme défini par ODE (ODE utilisent erp et cfm mais il existe un mapping entre la erp/cfm et la rigidité/amorti)
selfCollide	bool	Si ce paramètre est à vrai, le corps peut entrer en collision avec d'autres corps du modèle.
maxContacts	int	Le nombre maximum de contacts possibles entre deux entités. Cette valeur écrase la valeur spécifiée la valeur de l'élément $max_contacts$ définit dans la section **physics**.
laserRetro	double	Valeur d'intensité retournée par le capteur laser.

6.5.2 Eléments pour les joints

Nom	Type	Description
stopCfm,stopErp	double	Arrêt de la constraint force mixing (cfm) et error reduction parameter (erp) utilisée par ODE
provideFeedback	bool	Permet aux joints de publier leur information de tenseur (force-torque) à travers un plugin Gazebo
implicitSpringDamper, cfmDamping	bool	Si ce drapeau est à vrai, ODE va utiliser ERP and CFM pour simuler l'amorti. C'est une méthode numérique plus stable pour l'amortissement que la méthode par défaut. L'élément **cfmDamping** est obsolète et doit être changé en **implicitSpringDamper**.
fudgeFactor	double	Met à l'échelle l'excés du moteur du joint dans les limites du joint. Doit-être entre zéro et un.

6.6 model_state

Note : Cette partie est un travail en progrès qui n'est pas utilisé à l'heure actuelle. La représentation des configurations pour des groupes de joints peut-être également utilisée en utilisant le format srdf.

6.6.1 Elément <model_state>

Cet élément décrit un état basic du modèle URDF correspondant. Voici un exemple de l'élément state :

```
83   <model_state model="pr2" time_stamp="0.1">
84    <joint_state joint="r_shoulder_pan_joint" position="0" velocity="0" effort="0"/>
85    <joint_state joint="r_shoulder_lift_joint" position="0" velocity="0" effort="0"/>
86  </model_state>
```

6.6.2 Model State

— **<model_state>**

  — **model** (requis : string)
  Le nom du modèle dans l'URDF correspondant.
  — **timestamp** (optionnel : float, en secondes)
  Estampillage temporel de cet état en secondes.
  — **<joint_state>** (optionnel : string).

    — **joint** (requis : string)
    Le nom du joint auquel cet état se référe.
    — **position** (optionnel : float ou tableau de floats)
    Position pour chaque degré de liberté de ce joint.
    — **velocity** (optionnel : float ou tableau de floats)
    Vitesse pour chaque degré de liberté de ce joint.
    — **effort** (optionnel : float ou tableau de floats)
    Effort pour chaque degré de liberté de ce joint.

6.7 model

Le format de description unifié des robots (URDF pour Unified Robot Description Format) est une spécificiation XML pour décrire un robot. La spécification ne peut pas décrire tous les formats bien que celui-ci ait été conçu pour être le plus général possible. La limitation principale est que seule des structures en arbre peuvent être représentée ce qui ne permet pas de représenter des robots parallèles. De plus les specifications supposent que le robot est constitué de corps rigides connectés par des joints, les éléments flexibles ne sont pas supportés. La spécification couvre :
— La description cinématique et dynamique du robot
— La représentation visuelle du robot
— Modèle de collision du robot

La description d'un robot est constitué d'un ensemble de corps, et de joints connectant tous les corps ensembles. La description typique d'un robot ressemble donc à :

```
87  <robot name="pr2">
88    <link> ... </link>
89    <link> ... </link>
90    <link> ... </link>
91
92    <joint>  ....  </joint>
93    <joint>  ....  </joint>
94    <joint>  ....  </joint>
95  </robot>
```

Vous pouvez constater que la racine du format URDF est l'élément **<robot>**.

7. Simulation

7.1 Introduction

La simulation dans le contexte de la robotique permet en général de vérifier qu'un logiciel ou plus généralement une application distribuée, fait ce qui a été spécifié [5]. Le simulateur dans ce cas va remplacer le monde extérieur, et plus précisément il va calculer les actions des actionneurs et les signaux que devraient percevoir les capteurs. Il faut donc pour cela un simulateur de la dynamique du monde extérieur et un simulateur des phénomènes physique observés par les capteurs. La complexité des simulateurs vient du fait qu'il y a toujours une différence entre le monde réel et le monde simulé. Suivant le degré de réalité souhaité, il peut être nécessaire d'utiliser des algorithmes différents. Par exemple on ne simule pas de la même façon un robot qui évolue dans l'eau, un robot à roues, ou encore un drone.

Il existe cependant une confusion entre des architectures logicielles permettant de simuler

- une application logicielle complète. Elle permet d'utiliser le même logiciel en simulation et dans la réalité.
- la physique d'interaction entre le robot et son environnement

On appelle en général le premier un simulateur système tandis que le deuxième est appelé le moteur dynamique du simulateur. Gazebo et VREP rentre dans la première catégorie. ODE, Bullet et Newton sont dans la deuxième catégorie. Il arrive également que pour contrôler un robot il soit nécessaire de simuler la dynamique entre le robot et son environnement. Il s'agit encore d'une autre catégorie où le rôle essentiel du simulateur est de pouvoir fournir à un instant donné une commande à envoyer aux moteurs. Les formulations mathématiques peuvent être différentes car l'application est différente. Elle peuvent néanmoins partager un grand nombre de points communs.

On peut synthètiser la différence entre les simulateurs systèmes dont les buts sont :

- de simuler les capteurs, les actionneurs et l'environnement,
- pouvoir aller de la simulation au robot en minimisant les modifications,
- représenter de façon précise la réalité (grâce à un simulateur dynamique),
- maintenir la stabilité du calcul

tandis que les simulateurs de contrôle sont généralement prévus pour :

- le temps réel,
- capturer la dynamique principale du robot,
- être intégré dans la boucle de contrôle

Exemples de simulateurs systèmes : Gazebo, Stage, Morse, OpenHRP. Exemples de simulateurs de contrôle : MuJoCo, RBDL, Pinocchio, Robotran.

7.2 Algorithme général d'un simulateur système

Data: Modèle du robot, modèle du monde, état initial du robot et du monde
Result: Etat du monde, sortie des capteurs
Initialisation;
while *La simulation n'est pas finie* **do**
  for *chaque contrôleur c* **do**
    Lire les entrées capteurs pour le contrôleur c. Calculer la commande du contrôleur c.
  end
  Calculer un pas de la dynamique. Récupérer l'état du monde. Afficher l'état du monde. **for** *chaque contrôleur c* **do**
    Appliqué la commande du contrôleur c.
  end
end

7.3 Algorithme général d'un moteur dynamique

Il existe de nombreuses façons de poser le problème [2] mais on trouve l'algorithme général suivant :

Data: Modèle du robot et du monde, état courant du robot et du monde, paire de collisions, Force externes
Result: Position du robot et du monde
En utilisant la commande du contrôleur;
Calcul de la position des corps par intégration (par exemple en utilisant Runge-Kutta 4-5) de la dynamique des corps;
Détection des collisions;
Calcul des forces de contacts;

(voir le rigid body pipeline 1 dans [3]).

Par exemple le simulateur système Gazebo et Morse utilisent tous les deux ODE pour la simulation. Mais Gazebo peut également utiliser 3 autres moteurs dynamiques : Bullet [3], DART [6] et Simbody [7].

7.3.1 Problème 1 : Realité

La simulation de l'impact est très souvent difficile à effectuer de façon précise. Par exemple dans [1] la simulation du franchissement d'obstacles d'un robot humanoïde avec OpenHRP 2.x a fourni un impact de 800 N. Cependant sur le robot réel l'impact était de 1200 N qui aurait pu mener à la perte des capteurs de force localisés dans les chevilles du robot. La version d'OpenHRP 3.x a permis une meilleur simulation dynamique permettant de mieux discriminer les mouvements menant à des impacts trop violents des mouvements admissibles. La précision de la simulation nécessite cependant plus de calcul et donc prend plus de temps.

D'une manière générale il est difficile d'avoir un haut niveau de fiabilité en simulation. La compliance des actionneurs et des structures mécaniques est souvent difficile à réaliser car elle s'effectue à des fréquences nécessitant une très grande précision de simulation. La simulation de la vision est également souvent nécessaire afin de pouvoir simuler des algorithmes de traitement d'images.

7.3.2 Problème 2 : Flexibilité logicielle

Du point de vue logiciel on souhaiterait pouvoir passer de la simulation système du robot sans aucune modification ou simplement à travers de fichiers de configuration. Il est également souhaitable de pouvoir changer des parties du simulateur par exemple le moteur dynamique, afin de pouvoir faire des simulations dans des modes différents. Pour cela il est nécessaire d'avoir un middelware suffisamment puissant et pouvoir simuler les capteurs.

8. Gazebo - Ignition

8.1 Introduction

Gazebo est le simulateur historique sur ROS. La dernière version est Gazebo 11 est d'après `http://gazebosim.org/#features` la fin de vie de Gazebo est prévue pour le 29 Janvier 2025. Le successeur de Gazebo est Ignition. Gazebo est développé sous licence Apache 2.0. Gazebo est un simulateur système. Il permet d'interfacer les logiciels développés pour le robot avec le simulateur. Il est possible d'étendre les capacités du simulateur grâce à un système de plugins.

Pour la suite on peut cloner le package suivant dans un workspace **gazebo_ws** :

```
mkdir -p $HOME/gazebo_ws/src
cd $HOME/gazebo_ws/src
git clone https://github.com/olivier-stasse/gazebo_ros_demos.git
```

Pour le construire :

```
cd $HOME/gazebo_ws/
colcon build --packages-select
```

8.2 SDF format

Le format SDF est formalisé ici : `http://sdformat.org/`, il permet de décrire les différents éléments nécessaires à la simulation : le modèle des éléments du monde, le robot et les modèles de contact à utiliser.

Un exemple extrêmement simple est donné ici :

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://sun</uri>
    </include>

    <!-- reference to your plugin -->
    <plugin name="gazebo_tutorials" filename="libgazebo_tutorials.so"/>
  </world>
</sdf>
```

Ce fichier XML est un exemple très simple. On peut spécifier le format grâce à la ligne :

```
<sdf version="1.4">
```

On décrit le monde entre les balises **world**.
On peut ensuite mettre des modèles :

```
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://sun</uri>
    </include>
```

Ici le sol avec **ground_plane** et le soleil comme lumière (**sun**).
Enfin on peut spécifier un plugin nommé **gazebo_tutorials** inclus dans la bibliothèque **libgazebo_tutorials.so** grâce à :

```
      <plugin name="gazebo_tutorials" filename="libgazebo_tutorials.so"/>
```

8.2.1 Quand utiliser le format SDF ou le format URDF ?

Lorsque l'on utilise ROS dans une application robotique une bonne pratique est de ne pas faire de fichier SDF spécifique pour son robot. En effet, gazebo est capable de lire un fichier URDF et d'en faire un fichier SDF. Les plugins de gazebo à utiliser peuvent être indiqués par des balises gazebo. Celles-ci seront ignorées par les autres logiciels, MoveIt par exemple, qui ne les utilise pas.

Afin de pouvoir contrôler le robot à la fois en simulation et en contrôle il est généralement conseillé d'utiliser ros-control. Ce projet offre une abstraction du robot qui permet de tester le même logiciel en simulation et sur le vrai robot.

Cependant il nécessite un démarrage asynchrone qui rend l'utilisation plus compliquée et faire perdre du temps.

Enfin si le format URDF permet de modéliser un robot il ne permet pas de modéliser un monde et de fournir les informations nécessaires à sa simulation. Donc pour modéliser un environnement il n'est pas possible d'utiliser le format SDF.

Si votre robot n'utilise pas ros-control, il peut-être également plus avantageux d'utiliser directement le format SDF.

8.3 Plugin

8.3.1 Plugin Gazebo

Il existe 3 sortes de plugin sous Gazebo :

— Les plugin Sensor qui peuvent être mis dans des balises XML **<sensor>**
— Les plugin Model qui peuvent être mis dans des balises XML **<model>** mais pas **<sensor>**
— Les plugin World qui peuvent être mis dans des balises XML **<world>** mais pas **<model>** ni **<sensor>**

Il existe un certains nombre de plugins déjà disponibles à cette adresse : `https://github.com/ros-simulation/gazebo_ros_pkgs`

8.3.2 Plugin : exemple de base

Ce plugin permet de faire le lien entre Gazebo et ROS. Il s'agit d'un plugin **physics : :World**. Il est égalemnt possible de faire un plugin de type **sensors : :Sensor** ou de type **rendering : :Visual**.

Le plugin suivant est très simple.

```
#include <gazebo/common/Plugin.hh>
#include <rclcpp/rclcpp.hpp>
#include <gazebo_ros/node.hpp>

namespace gazebo
{
class WorldPluginTutorial : public WorldPlugin
{
public:
  WorldPluginTutorial() : WorldPlugin()
  {

  }
```

```
14
15   void Load(physics::WorldPtr _world, sdf::ElementPtr _sdf)
16   {
17     gazebo_ros::Node::SharedPtr ros_node = gazebo_ros::Node::Get(_sdf);
18
19     // Make sure the ROS node for Gazebo has already been initialized
20     if (!rclcpp::ok())
21     {
22       RCLCPP_FATAL(ros_node->get_logger(),
23        "A ROS node for Gazebo has not been initialized, unable to load plugin. ");
24       return;
25     }
26
27     RCLCPP_INFO(ros_node->get_logger(), "Hello World!");
28
29   }
30
31 };
32   GZ_REGISTER_WORLD_PLUGIN(WorldPluginTutorial)
33 }
```

Il suffit d'hériter de la classe de base WorldPlugin. Puis à part le constructeur qui permet de récupérer un pointeur vers le node créer dans ce contexte, de vérifier que ros a bien été initialisé et qui affiche "Hello World !". Il suffit ensuite d'implémenter la méthode **Load**. La dernière opération à faire est d'utiliser une macro pour créer la factory du plugin.

Le fichier **CMakeLists.txt** est le suivant :

```
1  cmake_minimum_required(VERSION 3.5)
2  project(gazebo_tutorials)
3
4  # Default to C++14
5  if(NOT CMAKE_CXX_STANDARD)
6    set(CMAKE_CXX_STANDARD 14)
7  endif()
8
9  if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
10   # we dont use add_compile_options with pedantic in message packages
11   # because the Python C extensions dont comply with it
12   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -Wpedantic -Wno-inconsistent-missing-override")
13 endif()
14
15 # find dependencies
16 find_package(rclcpp REQUIRED)
17 find_package(ament_cmake REQUIRED)
18
19
20 # Depend on system install of Gazebo
21 find_package(gazebo REQUIRED)
22 find_package(gazebo_ros REQUIRED)
23
24 #warning: variable CMAKE_CXX_FLAGS is modified
25 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${GAZEBO_CXX_FLAGS}")
26
27 #warning: use of link_directories() is strongly discouraged
28 MESSAGE(STATUS "GAZEBO_INSTALL_LIB_DIR: ${GAZEBO_INSTALL_LIB_DIR}")
29 link_directories(${gazebo_dev_LIBRARY_DIRS})
30 link_directories(${GAZEBO_INSTALL_LIB_DIR})
31 include_directories(${Boost_INCLUDE_DIR} ${GAZEBO_INCLUDE_DIRS})
32
33
34 add_library(${PROJECT_NAME} SHARED src/simple_world_plugin.cpp)
35 target_include_directories(${PROJECT_NAME} PUBLIC
36   $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
37   $<INSTALL_INTERFACE:include>)
38
39 ament_target_dependencies(
40   ${PROJECT_NAME}
41   "rclcpp"
42   "gazebo_ros"
43 )
44 ament_export_libraries(${PROJECT_NAME})
45
46 link_directories(${GAZEBO_INSTALL_LIB_DIR})
47
48 install(TARGETS ${PROJECT_NAME}
49   EXPORT export_${PROJECT_NAME}
50   ARCHIVE DESTINATION lib/${PROJECT_NAME}
51   LIBRARY DESTINATION lib/${PROJECT_NAME}
52   RUNTIME DESTINATION lib/${PROJECT_NAME}
53 )
54 install(DIRECTORY launch
```

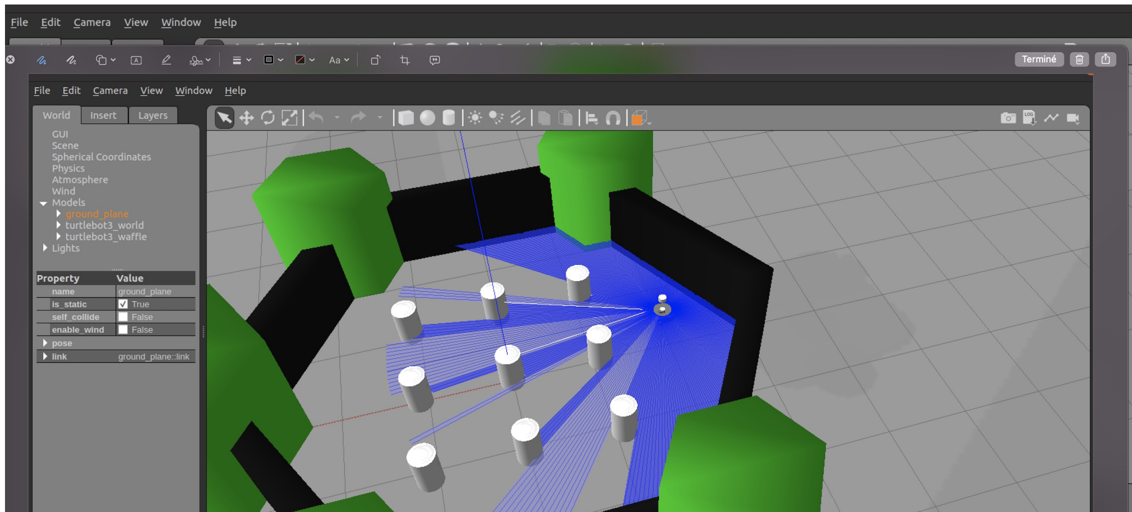


FIGURE 8.1 – Robot turtlebot3 modifié avec un capteur solaire

```
  DESTINATION share/${PROJECT_NAME})

install(DIRECTORY worlds
  DESTINATION share/${PROJECT_NAME})

ament_package()
```

8.4 Modèle de robot en SDF

8.4.1 Introduction

Dans cet exemple le but est de transformer le modèle du turtlebot3 waffle et lui rajouter un capteur. Il s'agit d'un capteur qui détecte la direction du soleil. Pour cela on utilise un plugin qui permet de calculer la direction du soleil en supposant qu'une lumière dans l'environnement s'appelle soleil.

8.4.1.1 Préparation

Le code est disponible sur github. Vous pouvez l'installer dans votre espace de travail de la façon suivante :

```
cd ~/dev_ws/src
git clone https://github.com/olivier-stasse/gazebo_ros_demos
```

Si vous êtes sur Foxy :

```
git checkout -b foxy -t origin/foxy
```

Cette commande permet de créer une branche locale nommée foxy qui suit la branche distance foxy qui se trouve sur le serveur origin.

Pour compiler le package :

```
cd ~/dev_ws/
colcon build --packages-select gazebo_tutorials
```

Il ensuite modifier la variable d'environnement **LD_LIBRARY_PATH** de la façon suivante :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${HOME}/dev_ws/install/gazebo_tutorials/lib/gazebo_tutorials
```

Pour lancer la simulation :

```
ros2 launch gazebo_tutorials turtlebot3.launch.py
```

Le résultat est représenté dans la figure Fig.8.1. On peut y voir le capteur solaire représenté par un cylindre au dessus du robot.

8.4.1.2 Organization

Le paquet **gazebo_tutorials** possède deux répertoires nommés **models** et **worlds**. Le répertoire **models** contient deux répertoires : l'un se nomme **sunloin** et l'autre **turtlebot_waffle**. Le modèle **sunloin** est une version modifiée du modèle **sun** fournit par défaut par Gazebo. Le modèle **sunloin** place une lumière à $256e^9$ km au lieu de $10m$ à la verticale du sol. Le répertoire **turtlebot_waffle** contient le modèle du robot Turtlebot3-Waffle qui a été modifié pour inclure un capteur supplémentaire par rapport au modèle initial.

8.4.2 Modèle du soleil

Ce modèle est définit dans le répertoire **gazebo_tutorials/models/sunloin**. Le soleil est donc définit de la façon suivante :

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <light type="directional" name="sunloin">
    <cast_shadows>true</cast_shadows>
    <pose>0 0 256e9 0 0 0</pose>
    <diffuse>0.8 0.8 0.8 1</diffuse>
    <specular>0.2 0.2 0.2 1</specular>
    <attenuation>
      <range>1000</range>
      <constant>0.9</constant>
      <linear>0.01</linear>
      <quadratic>0.001</quadratic>
    </attenuation>
    <direction>-0.5 0.1 -0.9</direction>
  </light>
</sdf>
```

Après la spécification du xml et de la version de la balise sdf, la lumière est définie grâce à la balise **light**. Elle est de type directionnel et son nom est **sunloin**.

La position de la lumière est spécifiée par la balise **pose** sous le format $(x, y, z, \theta_x, \theta_y, \theta_z)$. Ici le champ a été modifié par rapport au modèle par défaut pour être plus proche de la distance réelle. Les deux champs **diffuse** et **specular** correspondent à :

— **diffuse** : La couleur que renvoit un objet sous une lumière blanche. Cette couleur est perçue comme la couleur réelle de l'objet plutôt que la réflection de la lumière.
— **specular** : La couleur que renvoit la lumière suite à sa réflection sur un objet. Elle dépend de la direction (balise **direction**) de la lumière, de la pose de la caméra, et de la normale de la surface de l'objet.

L'attenuation de la lumière est définie par la balise **attenuation** par les champs suivants :

— range : La distance sur laquelle l'attenuation s'effectue
— constant : La constante du facteur d'attenuation. 1.0 signifie qu'elle ne s'attenue jamais, 0.0 qu'elle complétement atténuée.
— linear : Le facteur d'attenuation linéaire de la lumiére. 1.0 signifie que l'attenuation s'effectue de façon homogène sur la distance.
— quadratic : Le facteur d'attenuation quadratique : ajoute une courbure à l'atténuation.

8.4.3 Modèle du monde pour notre robot

Le fichier **worlds/waffle/model.sdf** décrit le monde dans lequel le robot **turtlebot3_waffle** évolue.

```
<?xml version="1.0"?>
<sdf version="1.6">
  <world name="default">

    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://sunloin</uri>
    </include>

    <scene>
      <shadows>false</shadows>
    </scene>

    <gui fullscreen='0'>
      <camera name='user_camera'>
        <pose frame=''>0.319654 -0.235002 9.29441 0 1.5138 0.009599</pose>
```

```
        <view_controller>orbit</view_controller>
        <projection_type>perspective</projection_type>
      </camera>
    </gui>

    <physics type="ode">
      <real_time_update_rate>1000.0</real_time_update_rate>
      <max_step_size>0.001</max_step_size>
      <real_time_factor>1</real_time_factor>
      <ode>
        <solver>
          <type>quick</type>
          <iters>150</iters>
          <precon_iters>0</precon_iters>
          <sor>1.400000</sor>
          <use_dynamic_moi_rescaling>1</use_dynamic_moi_rescaling>
        </solver>
        <constraints>
          <cfm>0.00001</cfm>
          <erp>0.2</erp>
          <contact_max_correcting_vel>2000.000000</contact_max_correcting_vel>
          <contact_surface_layer>0.01000</contact_surface_layer>
        </constraints>
      </ode>
    </physics>

    <model name="turtlebot3_world">
      <static>1</static>
      <include>
        <uri>model://turtlebot3_world</uri>
      </include>
    </model>

    <include>
      <pose>-2.0 -0.5 0.01 0.0 0.0 0.0</pose>
      <uri>model://turtlebot3_waffle</uri>
    </include>

  </world>
</sdf>
```

Le monde utilisé posséde donc un sol (ligne 6) et le soleil décrit précédemment (ligne 10). La section comprise entre les balises **gui** spécifie la caméra de l'utilisateur avec sa pose, son type de vue et le type de projection 3D. Elle correspond aux lignes (17-23).

Les paramètres de la simulation de la physique sont spécifiés entre les balises **physics**. On peut spécifier la vitesse de mise à jour temps réelle, la taille maximale d'une étape et le facteur temps réel.

La simulation dynamique est réalisée par ODE, et ses paramètres sont spécifiés dans la balise **ode**. Le solveur spécifié ici essait de résoudre rapidement le problème. Il utilise 150 itérations maximum. Les contraintes peuvent être relachées en utilisant les paramètres **cfm** et **erp**.

L'environnement de **turtlebot3_world** est spécifié par le modèle **turtlebot3_world/model.sdf** (46- 51). Il s'agit des colonnes vertes, des piliers, et des murs.

Le robot que nous avons modifié est spécifié dans le modèle **turtlebot3_waffle** (lignes 53- 56).

8.4.4 Modèle du robot turtlebot3_waffle modifié

8.4.4.1 Le modèle

Ce modèle est définit dans le répertoire **gazebo_tutorials/models/turtlebot3_waffle**. Il est donné de façon complète ici par le fichier model.sdf :

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="turtlebot3_waffle">
  <pose>0.0 0.0 0.0 0.0 0.0 0.0</pose>

    <link name="base_footprint"/>

    <link name="base_link">

      <inertial>
        <pose>-0.064 0 0.048 0 0 0</pose>
        <inertia>
          <ixx>4.2111447e-02</ixx>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyy>4.2111447e-02</iyy>
          <iyz>0</iyz>
```

```
18          <izz>7.5254874e-02</izz>
19        </inertia>
20        <mass>1.3729096e+00</mass>
21      </inertial>
22
23      <collision name="base_collision">
24        <pose>-0.064 0 0.048 0 0 0</pose>
25        <geometry>
26          <box>
27            <size>0.265 0.265 0.089</size>
28          </box>
29        </geometry>
30      </collision>
31
32      <visual name="base_visual">
33        <pose>-0.064 0 0 0 0 0</pose>
34        <geometry>
35          <mesh>
36            <uri>model://turtlebot3_waffle/meshes/waffle_base.dae</uri>
37            <scale>0.001 0.001 0.001</scale>
38          </mesh>
39        </geometry>
40      </visual>
41    </link>
42
43    <link name="imu_link">
44      <sensor name="tb3_imu" type="imu">
45        <always_on>true</always_on>
46        <update_rate>200</update_rate>
47        <imu>
48          <angular_velocity>
49            <x>
50              <noise type="gaussian">
51                <mean>0.0</mean>
52                <stddev>2e-4</stddev>
53              </noise>
54            </x>
55            <y>
56              <noise type="gaussian">
57                <mean>0.0</mean>
58                <stddev>2e-4</stddev>
59              </noise>
60            </y>
61            <z>
62              <noise type="gaussian">
63                <mean>0.0</mean>
64                <stddev>2e-4</stddev>
65              </noise>
66            </z>
67          </angular_velocity>
68          <linear_acceleration>
69            <x>
70              <noise type="gaussian">
71                <mean>0.0</mean>
72                <stddev>1.7e-2</stddev>
73              </noise>
74            </x>
75            <y>
76              <noise type="gaussian">
77                <mean>0.0</mean>
78                <stddev>1.7e-2</stddev>
79              </noise>
80            </y>
81            <z>
82              <noise type="gaussian">
83                <mean>0.0</mean>
84                <stddev>1.7e-2</stddev>
85              </noise>
86            </z>
87          </linear_acceleration>
88        </imu>
89        <plugin name="turtlebot3_imu" filename="libgazebo_ros_imu_sensor.so">
90          <ros>
91            <!-- <namespace>/tb3</namespace> -->
92            <remapping>~/out:=imu</remapping>
93          </ros>
94          </plugin>
95      </sensor>
96    </link>
97
98    <link name="base_scan">
99      <inertial>
100       <pose>-0.052 0 0.111 0 0 0</pose>
101       <inertia>
```

```
102          <ixx>0.001</ixx>
103          <ixy>0.000</ixy>
104          <ixz>0.000</ixz>
105          <iyy>0.001</iyy>
106          <iyz>0.000</iyz>
107          <izz>0.001</izz>
108        </inertia>
109        <mass>0.114</mass>
110      </inertial>
111
112      <collision name="lidar_sensor_collision">
113        <pose>-0.052 0 0.111 0 0 0</pose>
114        <geometry>
115          <cylinder>
116            <radius>0.0508</radius>
117            <length>0.055</length>
118          </cylinder>
119        </geometry>
120      </collision>
121
122      <visual name="lidar_sensor_visual">
123        <pose>-0.064 0 0.121 0 0 0</pose>
124        <geometry>
125          <mesh>
126            <uri>model://turtlebot3_waffle/meshes/lds.dae</uri>
127            <scale>0.001 0.001 0.001</scale>
128          </mesh>
129        </geometry>
130      </visual>
131
132      <sensor name="hls_lfcd_lds" type="ray">
133        <always_on>true</always_on>
134        <visualize>true</visualize>
135        <pose>-0.064 0 0.121 0 0 0</pose>
136        <update_rate>5</update_rate>
137        <ray>
138          <scan>
139            <horizontal>
140              <samples>360</samples>
141              <resolution>1.000000</resolution>
142              <min_angle>0.000000</min_angle>
143              <max_angle>6.280000</max_angle>
144            </horizontal>
145          </scan>
146          <range>
147            <min>0.120000</min>
148            <max>3.5</max>
149            <resolution>0.015000</resolution>
150          </range>
151          <noise>
152            <type>gaussian</type>
153            <mean>0.0</mean>
154            <stddev>0.01</stddev>
155          </noise>
156        </ray>
157        <plugin name="turtlebot3_laserscan" filename="libgazebo_ros_ray_sensor.so">
158          <ros>
159            <!-- <namespace>/tb3</namespace> -->
160            <remapping>~/out:=scan</remapping>
161          </ros>
162          <output_type>sensor_msgs/LaserScan</output_type>
163          <frame_name>base_scan</frame_name>
164        </plugin>
165      </sensor>
166    </link>
167
168    <link name="wheel_left_link">
169
170      <inertial>
171        <pose>0.0 0.144 0.023 -1.57 0 0</pose>
172        <inertia>
173          <ixx>1.1175580e-05</ixx>
174          <ixy>-4.2369783e-11</ixy>
175          <ixz>-5.9381719e-09</ixz>
176          <iyy>1.1192413e-05</iyy>
177          <iyz>-1.4400107e-11</iyz>
178          <izz>2.0712558e-05</izz>
179        </inertia>
180        <mass>0.1</mass>
181      </inertial>
182
183      <collision name="wheel_left_collision">
184        <pose>0.0 0.144 0.023 -1.57 0 0</pose>
185        <geometry>
```

```
186           <cylinder>
187             <radius>0.033</radius>
188             <length>0.018</length>
189           </cylinder>
190         </geometry>
191         <surface>
192           <!-- This friction pamareter don't contain reliable data!! -->
193           <friction>
194             <ode>
195               <mu>100000.0</mu>
196               <mu2>100000.0</mu2>
197               <fdir1>0 0 0</fdir1>
198               <slip1>0.0</slip1>
199               <slip2>0.0</slip2>
200             </ode>
201           </friction>
202           <contact>
203             <ode>
204               <soft_cfm>0</soft_cfm>
205               <soft_erp>0.2</soft_erp>
206               <kp>1e+5</kp>
207               <kd>1</kd>
208               <max_vel>0.01</max_vel>
209               <min_depth>0.001</min_depth>
210             </ode>
211           </contact>
212         </surface>
213       </collision>
214
215       <visual name="wheel_left_visual">
216         <pose>0.0 0.144 0.023 0 0 0</pose>
217         <geometry>
218           <mesh>
219             <uri>model://turtlebot3_waffle/meshes/tire.dae</uri>
220             <scale>0.001 0.001 0.001</scale>
221           </mesh>
222         </geometry>
223       </visual>
224     </link>
225
226     <link name="wheel_right_link">
227
228       <inertial>
229         <pose>0.0 -0.144 0.023 -1.57 0 0</pose>
230         <inertia>
231           <ixx>1.1175580e-05</ixx>
232           <ixy>-4.2369783e-11</ixy>
233           <ixz>-5.9381719e-09</ixz>
234           <iyy>1.1192413e-05</iyy>
235           <iyz>-1.4400107e-11</iyz>
236           <izz>2.0712558e-05</izz>
237         </inertia>
238         <mass>0.1</mass>
239       </inertial>
240
241       <collision name="wheel_right_collision">
242         <pose>0.0 -0.144 0.023 -1.57 0 0</pose>
243         <geometry>
244           <cylinder>
245             <radius>0.033</radius>
246             <length>0.018</length>
247           </cylinder>
248         </geometry>
249         <surface>
250           <!-- This friction pamareter don't contain reliable data!! -->
251           <friction>
252             <ode>
253               <mu>100000.0</mu>
254               <mu2>100000.0</mu2>
255               <fdir1>0 0 0</fdir1>
256               <slip1>0.0</slip1>
257               <slip2>0.0</slip2>
258             </ode>
259           </friction>
260           <contact>
261             <ode>
262               <soft_cfm>0</soft_cfm>
263               <soft_erp>0.2</soft_erp>
264               <kp>1e+5</kp>
265               <kd>1</kd>
266               <max_vel>0.01</max_vel>
267               <min_depth>0.001</min_depth>
268             </ode>
269           </contact>
```

```
270         </surface>
271       </collision>
272
273       <visual name="wheel_right_visual">
274         <pose>0.0 -0.144 0.023 0 0 0</pose>
275         <geometry>
276           <mesh>
277             <uri>model://turtlebot3_waffle/meshes/tire.dae</uri>
278             <scale>0.001 0.001 0.001</scale>
279           </mesh>
280         </geometry>
281       </visual>
282     </link>
283
284     <link name='caster_back_right_link'>
285       <pose>-0.177 -0.064 -0.004 -1.57 0 0</pose>
286       <inertial>
287         <mass>0.001</mass>
288         <inertia>
289           <ixx>0.00001</ixx>
290           <ixy>0.000</ixy>
291           <ixz>0.000</ixz>
292           <iyy>0.00001</iyy>
293           <iyz>0.000</iyz>
294           <izz>0.00001</izz>
295         </inertia>
296       </inertial>
297       <collision name='collision'>
298         <geometry>
299           <sphere>
300             <radius>0.005000</radius>
301           </sphere>
302         </geometry>
303         <surface>
304           <contact>
305             <ode>
306               <soft_cfm>0</soft_cfm>
307               <soft_erp>0.2</soft_erp>
308               <kp>1e+5</kp>
309               <kd>1</kd>
310               <max_vel>0.01</max_vel>
311               <min_depth>0.001</min_depth>
312             </ode>
313           </contact>
314         </surface>
315       </collision>
316     </link>
317
318     <link name='caster_back_left_link'>
319       <pose>-0.177 0.064 -0.004 -1.57 0 0</pose>
320       <inertial>
321         <mass>0.001</mass>
322         <inertia>
323           <ixx>0.00001</ixx>
324           <ixy>0.000</ixy>
325           <ixz>0.000</ixz>
326           <iyy>0.00001</iyy>
327           <iyz>0.000</iyz>
328           <izz>0.00001</izz>
329         </inertia>
330       </inertial>
331       <collision name='collision'>
332         <geometry>
333           <sphere>
334             <radius>0.005000</radius>
335           </sphere>
336         </geometry>
337         <surface>
338           <contact>
339             <ode>
340               <soft_cfm>0</soft_cfm>
341               <soft_erp>0.2</soft_erp>
342               <kp>1e+5</kp>
343               <kd>1</kd>
344               <max_vel>0.01</max_vel>
345               <min_depth>0.001</min_depth>
346             </ode>
347           </contact>
348         </surface>
349       </collision>
350     </link>
351
352     <link name="camera_link"/>
353
```

```
    <link name="camera_rgb_frame">
      <inertial>
        <pose>0.069 -0.047 0.107 0 0 0</pose>
        <inertia>
          <ixx>0.001</ixx>
          <ixy>0.000</ixy>
          <ixz>0.000</ixz>
          <iyy>0.001</iyy>
          <iyz>0.000</iyz>
          <izz>0.001</izz>
        </inertia>
        <mass>0.035</mass>
      </inertial>

      <pose>0.069 -0.047 0.107 0 0 0</pose>
      <sensor name="camera" type="camera">
        <always_on>true</always_on>
        <visualize>true</visualize>
        <update_rate>30</update_rate>
        <camera name="intel_realsense_r200">
          <horizontal_fov>1.02974</horizontal_fov>
          <image>
            <width>1920</width>
            <height>1080</height>
            <format>R8G8B8</format>
          </image>
          <clip>
            <near>0.02</near>
            <far>300</far>
          </clip>
          <noise>
            <type>gaussian</type>
            <!-- Noise is sampled independently per pixel on each frame.
                  That pixel's noise value is added to each of its color
                  channels, which at that point lie in the range [0,1]. -->
            <mean>0.0</mean>
            <stddev>0.007</stddev>
          </noise>
        </camera>
          <plugin name="camera_driver" filename="libgazebo_ros_camera.so">
            <ros>
              <!-- <namespace>test_cam</namespace> -->
              <!-- <remapping>image_raw:=image_demo</remapping> -->
              <!-- <remapping>camera_info:=camera_info_demo</remapping> -->
            </ros>
            <!-- camera_name>omit so it defaults to sensor name</camera_name-->
            <!-- frame_name>omit so it defaults to link name</frameName-->
            <!-- <hack_baseline>0.07</hack_baseline> -->
          </plugin>
      </sensor>
    </link>

    <link name="base_solar_sensor">
      <inertial>
        <pose>-0.052 0 0.111 0 0 0</pose>
        <inertia>
          <ixx>0.001</ixx>
          <ixy>0.000</ixy>
          <ixz>0.000</ixz>
          <iyy>0.001</iyy>
          <iyz>0.000</iyz>
          <izz>0.001</izz>
        </inertia>
        <mass>0.114</mass>
      </inertial>

      <collision name="solar_sensor_collision">
        <pose>-0.052 0 0.311 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius>0.0508</radius>
            <length>0.055</length>
          </cylinder>
        </geometry>
      </collision>

      <visual name="solar_sensor_visual">
        <pose>-0.064 0 0.321 0 0 0</pose>
        <geometry>
    <cylinder>
      <radius>0.0508</radius>
            <length>0.055</length>
          </cylinder>
        </geometry>
```

```
      </visual>

    </link>

    <joint name="base_joint" type="fixed">
      <parent>base_footprint</parent>
      <child>base_link</child>
      <pose>0.0 0.0 0.010 0 0 0</pose>
    </joint>

    <joint name="wheel_left_joint" type="revolute">
      <parent>base_link</parent>
      <child>wheel_left_link</child>
      <pose>0.0 0.144 0.023 -1.57 0 0</pose>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>

    <joint name="wheel_right_joint" type="revolute">
      <parent>base_link</parent>
      <child>wheel_right_link</child>
      <pose>0.0 -0.144 0.023 -1.57 0 0</pose>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>

    <joint name='caster_back_right_joint' type='ball'>
      <parent>base_link</parent>
      <child>caster_back_right_link</child>
    </joint>

    <joint name='caster_back_left_joint' type='ball'>
      <parent>base_link</parent>
      <child>caster_back_left_link</child>
    </joint>

    <joint name="imu_joint" type="fixed">
      <parent>base_link</parent>
      <child>imu_link</child>
      <pose>-0.032 0 0.068 0 0 0</pose>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>

    <joint name="lidar_joint" type="fixed">
      <parent>base_link</parent>
      <child>base_scan</child>
      <pose>-0.064 0 0.121 0 0 0</pose>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>

    <joint name="camera_joint" type="fixed">
      <parent>base_link</parent>
      <child>camera_link</child>
      <pose>0.064 -0.065 0.094 0 0 0</pose>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>

    <joint name="camera_rgb_joint" type="fixed">
      <parent>camera_link</parent>
      <child>camera_rgb_frame</child>
      <pose>0.005 0.018 0.013 0 0 0</pose>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>

    <joint name="solar_sensor_joint" type="fixed">
      <parent>base_link</parent>
      <child>base_solar_sensor</child>
      <pose>-0.064 0 0.321 0 0 0</pose>
      <axis>
        <xyz>0 0 1</xyz>
      </axis>
    </joint>

    <plugin name="turtlebot3_diff_drive" filename="libgazebo_ros_diff_drive.so">
```

```

      <ros>
        <!-- <namespace>/tb3</namespace> -->
      </ros>

      <update_rate>30</update_rate>

      <!-- wheels -->
      <left_joint>wheel_left_joint</left_joint>
      <right_joint>wheel_right_joint</right_joint>

      <!-- kinematics -->
      <wheel_separation>0.287</wheel_separation>
      <wheel_diameter>0.066</wheel_diameter>

      <!-- limits -->
      <max_wheel_torque>20</max_wheel_torque>
      <max_wheel_acceleration>1.0</max_wheel_acceleration>

      <command_topic>cmd_vel</command_topic>

      <!-- output -->
      <publish_odom>true</publish_odom>
      <publish_odom_tf>true</publish_odom_tf>
      <publish_wheel_tf>false</publish_wheel_tf>

      <odometry_topic>odom</odometry_topic>
      <odometry_frame>odom</odometry_frame>
      <robot_base_frame>base_footprint</robot_base_frame>

    </plugin>

    <plugin name="turtlebot3_joint_state" filename="libgazebo_ros_joint_state_publisher.so">
      <ros>
        <!-- <namespace>/tb3</namespace> -->
        <remapping>~/out:=joint_states</remapping>
      </ros>
      <update_rate>30</update_rate>
      <joint_name>wheel_left_joint</joint_name>
      <joint_name>wheel_right_joint</joint_name>
    </plugin>

    <plugin name="turtlebot3_solar_sensor" filename="libsolar_sensor.so">
      <update_rate>30</update_rate>
      <light_name>sunloin</light_name>
      <model_name>turtlebot3_waffle</model_name>
      <frame_name>base_solar_sensor</frame_name>
      </plugin>

  </model>
</sdf>
```

8.4.4.2 Explications du modèle

Le modèle est spécifié entre deux balises **model**. Il est constitué de corps spécifiés par la balise **link** et d'articulations spécifiées par la balise **joint**.

La base du robot **turtlebot3_waffle** est nommé **base_link** (voir ligne 8).

Un corps, comme pour le format URDF, est défini par des paramètres inertiels exprimés dans la balise **inertial**. Pour le corps **base_link** il s'agit des ligne 10-21. Le centre de masse est défini par la balise **pose** dans la balise **inertial**. Il s' agit de la ligne 11 pour le corps **base_link**. La matrice d'inertie du corps, supposé solide, est exprimée dans la balise **inertia**. (ligne 12). La masse du corps est exprimée dans la balise **mass** dans la balise **inertial** (ligne 20).

Afin de pouvoir calculer les collisions, et limiter le coût calculatoire il est préférable de ne pas utilser un modèle compliqué de la géométrie de l'objet. On sépare pour cela la partie pour calculer les collisions et la partie pour visualiser l'objet. La partie collision est spécifiée avec la balise **collision** (ligne 23 pour **base_link**). La partie géométrie est spécifiée avec la balise **geometry** (ligne 25 pour **base_link**). Ici par exemple pour l'objet **base_link** la collision est modélisée par une boîte d'une taille de $26.5cm$. La balise **visual** est utilisée pour spécifiée par la géométrie de l'objet utilisée pour la visualisation du robot. Ici pour l'objet **base_link** la géométrie est spécifié par un fichier DAE qui fournit une soupe de triangles, ou de polygone (ligne 34).

Comme il s'agit d'un robot mobile il n'est pas fixé dans le monde comme un robot industriel 6 axes classique. On ne trouve donc aucun joint de type **fixed** entre la base (**base_footprint** et le repère **world**.

Afin de pouvoir bouger le robot **turtlebot3_waffle** le robot utilise deux roues actionnées dont les

corps sont spécifiés par **wheel_right_link** (lignes 226 - 282) et **wheel_left_link** (lignes 168 - 224). Le robot conserve son équilibre grâce à deux roues folles : **cast_back_right_link** (lignes 284 - 316) et **caster_back_left_link** (lignes 318 - 350).

Les joints qui permettent de bouger le robot pour les roues sont : **wheel_left_joint** (lignes 448 - 455) et **wheel_right_joint** (lignes 457 - 464).

Les joints des roues libres sont : **caster_back_left_joint** (lignes 471 - 474) et **caster_back_right_joint** (lignes 466 - 469).

Les capteurs sont considérés comme des corps. On a ainsi une centrale inertielle qui est définie par le corps **imu_link** (lignes 43-96). A l'intérieur des balises **link**, on trouve la définition de la simulation du capteur entre deux balises **sensor** (lignes 44-88). On y trouve la fréquence de mise à jour avec la balise **update_rate**. Des champs spécifiques permettent de définir la loi de probabilités du bruit du capteur. Dans ce cas, un bruit Gaussien est ajouté sur chacun des axes (x,y,z) de la vitesse angulaire et l'accélératon linéaire. Il est défini par une valeur moyenne et une déviation standard. Chacun des axes est considéré comme une variable indépendante. Enfin pour que les informations du capteur soient disponibles via un topic ROS, il faut utiliser un plugin gazebo. Dans le cas de la centrale inertielle on peut utiliser le plugin **libgazebo_ros_imu_sensor** (lignes 89-94).

8.4.4.3 Capteur solaire

Le capteur solaire est défini par le corps **base_solar_sensor**. Cela correspond aux lignes 406 - 440 du fichier donnant le modèle du robot. La géométrie pour la collision et la visualisation sont les mêmes. C'est à dire un cylindre de rayon 0.0508 et de longueur 0.055.

8.4.5 Plugin du capteur solaire

Le but de ce plugin est donner la direction du soleil sous la forme d'un topic de type **sensor_msgs : :msg : :Imu**. Le nom par défaut du topic sur lequel les informations seront publiées est **sunloin_imu**. La fréquence de mise à jour est donnée par le champ **update_rate**. Le nom du corps correspondant au capteur par défaut est **base_solar_sensor**. Il est possible de spécifier des offsets avec les balises **offsets** et **rpy_offsets**.

8.4.5.1 Le code

Le code implémentant cette stratégie est donné ici, la licence ayant été retirée par souci de compacité :

```
#include <gazebo_ros/node.hpp>
#include <gazebo_ros/utils.hpp>
#include <gazebo_ros/conversions/geometry_msgs.hpp>
#include <gazebo_ros/conversions/builtin_interfaces.hpp>
#include <gazebo_tutorials/solar_sensor.hpp>
#include <sensor_msgs/msg/imu.hpp>
#include <rclcpp/rclcpp.hpp>

#ifdef NO_ERROR
// NO_ERROR is a macro defined in Windows that's used as an enum in tf2
#undef NO_ERROR
#endif

#ifdef IGN_PROFILER_ENABLE
#include <ignition/common/Profiler.hh>
#endif

#include <string>
#include <memory>

namespace gazebo_plugins
{

class SolarSensorPrivate
{
public:

  /// Callback to be called at every simulation iteration
  /// \param[in] info Updated simulation info
  void OnUpdate(const gazebo::common::UpdateInfo & info);

  /// The link being traked.
  gazebo::physics::LightPtr light_{nullptr};

  /// The body of the frame to display pose, twist
  gazebo::physics::LinkPtr reference_link_{nullptr};

  /// Pointer to ros node
  gazebo_ros::Node::SharedPtr ros_node_{nullptr};
```

```
  /// Odometry publisher
  rclcpp::Publisher<sensor_msgs::msg::Imu>::SharedPtr pub_{nullptr};

  /// Odom topic name
  std::string topic_name_{"sunloin_imu"};

  /// Frame transform name, should match name of reference link, or be world.
  std::string frame_name_{"base_solar_sensor"};

  /// Constant xyz and rpy offsets
  ignition::math::Pose3d offset_;

  /// Keep track of the last update time.
  gazebo::common::Time last_time_;

  /// Publish rate in Hz.
  double update_rate_{0.0};

  /// Gaussian noise
  double gaussian_noise_;

  /// Pointer to the update event connection
  gazebo::event::ConnectionPtr update_connection_{nullptr};
};

SolarSensor::SolarSensor()
: impl_(std::make_unique<SolarSensorPrivate>())
{
}

SolarSensor::~SolarSensor()
{
}

// Load the controller
void SolarSensor::Load(gazebo::physics::ModelPtr model,
           sdf::ElementPtr sdf)
{
  // Configure the plugin from the SDF file
  impl_->ros_node_ = gazebo_ros::Node::Get(sdf);

  // Get QoS profiles
  const gazebo_ros::QoS & qos = impl_->ros_node_->get_qos();

  if (!sdf->HasElement("update_rate")) {
    RCLCPP_DEBUG(
      impl_->ros_node_->get_logger(), "solar_sensor plugin missing <update_rate>, defaults to 0.0"
      " (as fast as possible)");
  } else {
    impl_->update_rate_ = sdf->GetElement("update_rate")->Get<double>();
  }

  std::string light_name;
  if (!sdf->HasElement("light_name")) {
    RCLCPP_ERROR(impl_->ros_node_->get_logger(), "Missing <light_name>, cannot proceed");
    return;
  } else {
    light_name = sdf->GetElement("light_name")->Get<std::string>();
  }

  gazebo::physics::WorldPtr world = model->GetWorld();
  impl_->light_ = world->LightByName(light_name);
  if (!impl_->light_) {
    RCLCPP_ERROR(
      impl_->ros_node_->get_logger(), "light_name: %s does not exist\n",
      light_name.c_str());
    return;
  }

  impl_->pub_ = impl_->ros_node_->create_publisher<sensor_msgs::msg::Imu>(
    impl_->topic_name_, qos.get_publisher_qos(
      impl_->topic_name_, rclcpp::SensorDataQoS().reliable()));
  impl_->topic_name_ = impl_->pub_->get_topic_name();
  RCLCPP_DEBUG(
    impl_->ros_node_->get_logger(), "Publishing on topic [%s]", impl_->topic_name_.c_str());

  if (sdf->HasElement("xyz_offsets")) {
    RCLCPP_WARN(
      impl_->ros_node_->get_logger(), "<xyz_offsets> is deprecated, use <xyz_offset> instead.");
    impl_->offset_.Pos() = sdf->GetElement("xyz_offsets")->Get<ignition::math::Vector3d>();
  }
  if (!sdf->HasElement("xyz_offset")) {
    if (!sdf->HasElement("xyz_offsets")) {
```

```
      RCLCPP_DEBUG(impl_->ros_node_->get_logger(), "Missing <xyz_offset>, defaults to 0s");
    }
  } else {
    impl_->offset_.Pos() = sdf->GetElement("xyz_offset")->Get<ignition::math::Vector3d>();
  }

  if (sdf->HasElement("rpy_offsets")) {
    RCLCPP_WARN(
      impl_->ros_node_->get_logger(), "<rpy_offsets> is deprecated, use <rpy_offset> instead.");
    impl_->offset_.Rot() = ignition::math::Quaterniond(
      sdf->GetElement("rpy_offsets")->Get<ignition::math::Vector3d>());
  }
  if (!sdf->HasElement("rpy_offset")) {
    if (!sdf->HasElement("rpy_offsets")) {
      RCLCPP_DEBUG(impl_->ros_node_->get_logger(), "Missing <rpy_offset>, defaults to 0s");
    }
  } else {
    impl_->offset_.Rot() = ignition::math::Quaterniond(
      sdf->GetElement("rpy_offset")->Get<ignition::math::Vector3d>());
  }

  if (!sdf->HasElement("gaussian_noise")) {
    RCLCPP_DEBUG(impl_->ros_node_->get_logger(), "Missing <gassian_noise>, defaults to 0.0");
    impl_->gaussian_noise_ = 0;
  } else {
    impl_->gaussian_noise_ = sdf->GetElement("gaussian_noise")->Get<double>();
  }

  impl_->last_time_ = world->SimTime();

  if (!sdf->HasElement("frame_name")) {
    RCLCPP_DEBUG(
      impl_->ros_node_->get_logger(), "Missing <frame_name>, defaults to base_solar_sensor");
  } else {
    impl_->frame_name_ = sdf->GetElement("frame_name")->Get<std::string>();
  }

  if (model!=nullptr)
  {
    impl_->reference_link_ = model->GetLink(impl_->frame_name_);
    if (!impl_->reference_link_) {
      RCLCPP_WARN(
        impl_->ros_node_->get_logger(), "<frame_name> [%s] does not exist.",
        impl_->frame_name_.c_str());
    }
  }

  // Listen to the update event. This event is broadcast every simulation iteration
  impl_->update_connection_ = gazebo::event::Events::ConnectWorldUpdateBegin(
    std::bind(&SolarSensorPrivate::OnUpdate, impl_.get(), std::placeholders::_1));
}

// Update the controller
void SolarSensorPrivate::OnUpdate(const gazebo::common::UpdateInfo & info)
{
  if (!light_) {
    return;
  }
#ifdef IGN_PROFILER_ENABLE
  IGN_PROFILE("SolarSensorPrivate::OnUpdate");
#endif
  gazebo::common::Time current_time = info.simTime;

  if (current_time < last_time_) {
    RCLCPP_WARN(ros_node_->get_logger(), "Negative update time difference detected.");
    last_time_ = current_time;
  }

  // Rate control
  if (update_rate_ > 0 &&
    (current_time - last_time_).Double() < (1.0 / update_rate_))
  {
    return;
  }

  // If we don't have any subscribers, don't bother composing and sending the message
  if (ros_node_->count_subscribers(topic_name_) == 0) {
    return;
  }

  // Differentiate to get accelerations
  double tmp_dt = current_time.Double() - last_time_.Double();
  if (tmp_dt == 0) {
    return;
```

```
208   }
209 #ifdef IGN_PROFILER_ENABLE
210   IGN_PROFILE_BEGIN("fill ROS message");
211 #endif
212   sensor_msgs::msg::Imu imu_msg;
213 
214   // Copy data into pose message
215   imu_msg.header.frame_id = frame_name_;
216   imu_msg.header.stamp = gazebo_ros::Convert<builtin_interfaces::msg::Time>(current_time);
217 
218   // Get inertial rates
219   ignition::math::Vector3d vpos = light_->WorldLinearVel();
220   ignition::math::Vector3d veul = light_->WorldAngularVel();
221 
222   // Get pose/orientation
223   auto pose = light_->WorldPose();
224 
225   // Apply reference frame
226   if (reference_link_) {
227     // Convert to relative pose, rates
228     auto frame_pose = reference_link_->WorldPose();
229     auto frame_vpos = reference_link_->WorldLinearVel();
230     auto frame_veul = reference_link_->WorldAngularVel();
231 
232     pose.Pos() = pose.Pos() - frame_pose.Pos();
233     pose.Pos().Normalize();
234     pose.Pos() = frame_pose.Rot().RotateVectorReverse(pose.Pos());
235     pose.Rot() *= frame_pose.Rot().Inverse();
236 
237     vpos = frame_pose.Rot().RotateVector(vpos - frame_vpos);
238     veul = frame_pose.Rot().RotateVector(veul - frame_veul);
239   }
240 
241   // Apply constant offsets
242 
243   // Apply XYZ offsets and get position and rotation components
244   pose.Pos() = pose.Pos() + offset_.Pos();
245   // Apply RPY offsets
246   pose.Rot() = offset_.Rot() * pose.Rot();
247   pose.Rot().Normalize();
248 
249   double pitch = std::asin(-pose.Pos().Z());
250   double yaw= std::atan2(pose.Pos().Y(), pose.Pos().X());
251 
252   ignition::math::Quaterniond compass_to_sunloin(0.0, pitch, yaw);
253   // Fill out messages
254   imu_msg.orientation = gazebo_ros::Convert<geometry_msgs::msg::Quaternion>(compass_to_sunloin);
255 
256   // Fill in covariance matrix
257   /// @TODO: let user set separate linear and angular covariance values
258   double gn2 = gaussian_noise_ * gaussian_noise_;
259   imu_msg.orientation_covariance[0] = gn2;
260   imu_msg.orientation_covariance[4] = gn2;
261   imu_msg.orientation_covariance[8] = gn2;
262 
263 #ifdef IGN_PROFILER_ENABLE
264   IGN_PROFILE_END();
265   IGN_PROFILE_BEGIN("publish");
266 #endif
267   // Publish to ROS
268   pub_->publish(imu_msg);
269 #ifdef IGN_PROFILER_ENABLE
270   IGN_PROFILE_END();
271 #endif
272   // Save last time stamp
273   last_time_ = current_time;
274 }
275 
276 GZ_REGISTER_MODEL_PLUGIN(SolarSensor)
277 
278 }  // namespace gazebo_plugins
```

8.4.5.2 En-têtes

Le plugin utilise une architecture d'implémentation privée qui permet d'éviter d'exposer inutilement certains détails des champs pour l'API.

Pour cette raison la définition de la classe faisant réellement le travail est inclue dans ce fichier au lieu d'être exposée dans un fichier d'en-tête.

Les entêtes commencent par celles liant Gazebo et ROS (lignes 1- 4). On y trouve l'entête permettant l'accés à l'API des Nodes de ROS, un entête pour des utilitaires génériques. Puis les entêtes de conversions

sont utilisés afin de pouvoir passer des outils mathématiques de Gazebo (librairie ignition) aux messages ROS. Il faut ensuite spécifier le fichier qui contient la déclaration de la classe **SolarSensor** (ligne 5) On trouve ensuite l'entête pour déclarer les messages ROS pour les imus (ligne 6). Il y a ensuite l'entête permettant d'utiliser le C++ avec ROS (rclcpp).

8.4.5.3 Déclaration de SolarSensorPrivate

Après des en-têtes systèmes standards (string et memory), l'implémentation privée du capteur solaire est déclarée (lignes 24-64). Nommée **SolarSensorPrivate** elle stocke :
- un pointeur partagé sur le corps où se trouve la lumière **light_** (ligne 33)
- un pointeur partagé sur le corps de référence du modèle du robot **reference_link_** (ligne 36)
- un pointeur partagé sur le node ROS associé au simulateur **ros_node_** (ligne 39)
- un publisher ROS de message ROS Imu **pub_** (ligne 42)
- le nom du topic sur lequel publié le message d'IMU **topic_name_** (ligne 45)
- le nom du corps de référence du robot **frame_name_** (ligne 48)
- l'offset de pose du capteur **offset_** (ligne 51)
- la dernière date de la mise à jour du capteur **offset_** (ligne 54)
- la vitesse désirée de mise à jour du capteur **update_rate_** (ligne 57)
- le bruit Gaussien du capteur **Gaussian_** (ligne 60)
- la fonction d'appel lorsqu'une mise à jour est nécessaire **update_connection_** (ligne 63)

L'instanciation de la classe privée se fait dans le constructor du plugin **SolarSensor** (ligne 67).

8.4.5.4 Méthode SolarSensor : :Load

La méthode **SolarSensor : :Load** s'occupe de lire le fichier SDF et d'initialiser tous les paramètres du plugin et donc de l'implémentation privée stockée dans le champ **impl_**. Les deux paramètres de la méthode sont le lien vers le modèle dans lequel est situé l'appel au plugin (**model**), et le pointeur vers les éléments SDF paramètres du plugin.

La première chose est de stocker le pointeur vers le Node ROS (ligne 80). On peut ensuite stocker la qualité de service utilisé par le Node (83). La méthode ensuite récupère la valeur de la balise **update_rate** (lignes 85- 91). La méthode ensuite récupère le nom de la lumière **light_name** (lignes 93- 99). Grâce à cette information il est alors possible grâce à la référence au monde d'obtenir la lumière correspondante (**light_**).

La méthode créée ensuite un publisher ROS pour publier un topic de type **sensor_msgs : :msg : :Imu** avec le nom **topic_name_**, et la qualité de service de type **reliable** (lignes 110-115).

La méthode lit ensuite les valeurs de décalage avec les balises **xyz_offset** (lignes 117- 128) et **rpy_offset** (lignes 130- 143) si elles existent.

Le bruit gaussien est également lu de la balise **gaussian_noise** (lignes 145- 150).

La variable **last_time_** est ensuite initialisé avec le temps simulé.

Il faut ensuite lire le nom du frame de référence **frame_name** (lignes 154-169) pour ensuite trouver le corps de référence grâce au pointeur sur le monde. La dernière opération est de connecter la mise à jour du plugin avec l'appel à la méthode de la méthode **OnUpdate** de l'implémentation privée du plugin.

8.4.5.5 Méthode SolarSensorPrivate : :OnUpdate

La méthode vérifie qu'une lumière a été trouvée et retourne immédiatement si ce n'est pas le cas.

Ensuite, la méthode vérifie que le temps écoulé correspond à la fréquence de mise à jour demandée par l'utilisateur. Si ce n'est pas le cas elle retourne immédiatement (lignes 185-197). Afin de gagner du temps, la méthode retourne immédiatement si aucun autre Node n'a souscrit au topic (ligne 201).

La construction du message à publier sur le topic s'effectue des lignes 212 à 261.

Tout d'abord la direction du soleil est exprimée dans le corps du capteur. Ensuite le temps est converti du format Gazebo au format ROS.

La vitesse, l'accéleration et enfin la pose de la lumière (si elle bouge) sont ensuite récupérées. Elles sont initialement exprimées dans le repère du monde. Puis ces quantités sont projetées de façon à être exprimées dans le repère du capteur (lignes 226-239).

Si des offsets sont fournis ils sont utilisés des lignes 241 à 247.

On calcule ensuite la direction de la lumière en calculant la direction du vecteur vers la lumière en coordonnées polaire (lignes 249 à 250). Le résultat est ensuite convertit en quaternion puis utiliser pour remplir le champ orientation du message à publier.

Il reste ensuite à remplir la matrice de covariance avec le bruit Gaussien pour compléter le message.

Le message est ensuite publier (ligne 268.

Appendices

9 Mots clefs pour les fichiers launch 109

9.1 <launch>

10 Rappels sur le bash 111

10.1 Lien symbolique
10.2 Gestion des variables d'environnement
10.3 Fichier .bashrc

11 Utiliser docker pour ROS 113

11.1 Préparer votre ordinateur
11.2 Utiliser l'image docker
11.3 Installer les images docker de ROS
11.4 Construire l'image docker
11.5 Mac

12 Mémo 117

Bibliography 121

Books
Articles
Chapitre de livres
Autres

9. Mots clefs pour les fichiers launch

9.1 <launch>

La balise tag est l'èlèment racine de tout fichier roslaunch. Son unique rôle est d'être un conteneur pour les autres èlèments.

9.1.1 Attributs

deprecated="deprecation message" ROS 1.1 : Indique aux utilisateurs que roslaunch n'est plus utilisé.

9.1.2 Elements

— <node> Lance un node.
— <param> Affecte un paramètre dans le serveur de paramètre.
— <remap> Renomme un espace de nom.
— <machine> Déclare une machine à utiliser dans la phase de démarrage.
— <rosparam> Affecte des paramètres en chargant un fichier rosparam.
— <include> Inclus d'autres fichiers roslaunch.
— <env> Spécifie une variable d'environnement pour les noeuds lancés.
— <test> Lance un node de test (voir rostest).
— <arg> Déclare un argument.
— <group> Groupe des élm̀ents partageant un espace de nom ou un renommage.

10. Rappels sur le bash

10.1 Lien symbolique

10.1.1 Voir les liens symboliques

Pour lister le contenu de votre répertoire vous pouvez utiliser la commande suivante :

```
ls
```

La sortie est la suivante :

```
  catkin_ws             Music
  catkin_ws_prenom_nom  Pictures
  Desktop               Public
  Documents             Templates
  Downloads             Videos
  examples.desktop
```

Pour savoir si le répertoire **catkin_ws** est un raccourci vers un autre répertoire vous pouvez utiliser la commande suivante :

```
ls -al catkin_ws
```

Le résultat est alors :

```
lrwxrwxrwx   1 pnom pnom        20 dec.   6 08:43 catkin_ws -> catkin_ws_prenom_nom
```

10.1.2 Créer un lien symbolique

Pour créer un lien de votre répertoire **catkin_ws_prenom_nom** vers **catkin_ws** :

```
ln -s catkin_ws_prenom_nom catkin_ws
```

10.1.3 Enlever un lien symbolique

Pour enlever un lien, il faut faire comme pour enlever un fichier :

```
rm catkin_ws
```

10.2 Gestion des variables d'environnement

La liste des variables d'environnements peut s'obtenir en faisant :

```
env
```

La liste des variables d'environnements peut s'obtenir en cherchant les lignes contenant la chaîne ROS :

```
env | grep ROS
```

donne le résultat suivant pour humble :

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

On peut voir également des variables très importantes avec :

```
env | grep ros
```

donne

```
AMENT_PREFIX_PATH=/opt/ros/humble
PYTHONPATH=/opt/ros/humble/lib/python3.10/site-packages:/opt/ros/humble/local/lib/python3.10/dist-packages
LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu/gazebo-11/plugins:/opt/ros/humble/opt/rviz_ogre_vendor/lib:/opt/ros/
    humble/lib/x86_64-linux-gnu:/opt/ros/humble/lib
PATH=/opt/ros/humble/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
    :/snap/bin:/snap/bin
```

Les variables d'environnement sont locales à un terminal. Si vous avez ouvert plusieurs terminaux la modification de la valeur ne peut affecter que celui où la modification a été faites.

10.3 Fichier .bashrc

Le fichier **.bashrc** est lu à chaque fois qu'un terminal est lancé. La version qui est lue peut-être différente si le fichier est modifié d'un lancement à l'autre. Il est possible de forcer le fichier **.bashrc** à lire un autre fichier. Par exemple pour lire le fichier **setup.bash** qui se trouve dans votre répertoire **dev_ws** :

```
source /home/etudiant/dev_ws/devel/setup.bash
```

11. Utiliser docker pour ROS

11.1 Préparer votre ordinateur

Pour préparer votre ordinateur en distanciel, voici la liste des choses à faire :

— Installer docker pour votre machine

— Récupérer l'image docker pour les TPs :

```
docker pull aipprimecaoccitanie/ros-introduction-robmob:latest
```

— Lancer l'image :

```
docker run -p 6080:6080 --name ros-intro-vnc -it aipprimecaoccitanie/ros-introduction-robmob:latest
```

— Dans votre navigateur :

```
http://localhost:6080/vnc.html?host=localhost&port=6080
```

Vous devriez voir apparaître l'image représentée dans la figure Fig.11.1.

Si le navigateur indique l'erreur `timeout:` il faut augmenter la valeur timeout en allant dans le menu de la roue crantée. Cliquer directement sur connect sans rentrer de mot de passe. Vous devriez voir ensuite l'image représentée dans la figure Fig.11.2

11.2 Utiliser l'image docker

Démarrer le container spécifique à l'AIP s'effectue avec la commande suivante :

```
docker run -p 6080:6080 --name ros-intro-vnc -it aip-primeca-occitanie/ros-intro-robmob:latest
```

Le nom de l'image est spécifié en dernier et cette commande a créé le container **ros-intro-vnc**.

11.2.1 Lancer un bash

Il est ensuite possible de démarrer un executable bash dans une autre console en tapant :

```
docker exec -it ros-intro-vnc bash
```

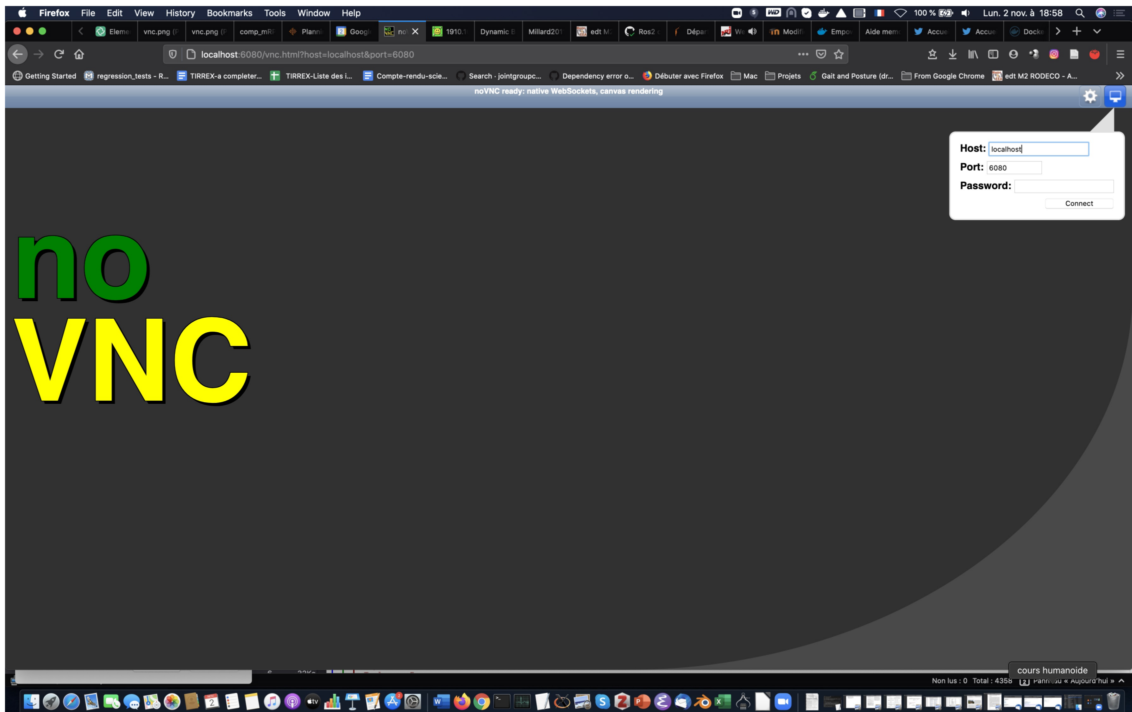


FIGURE 11.1 – Lancer le client VNC

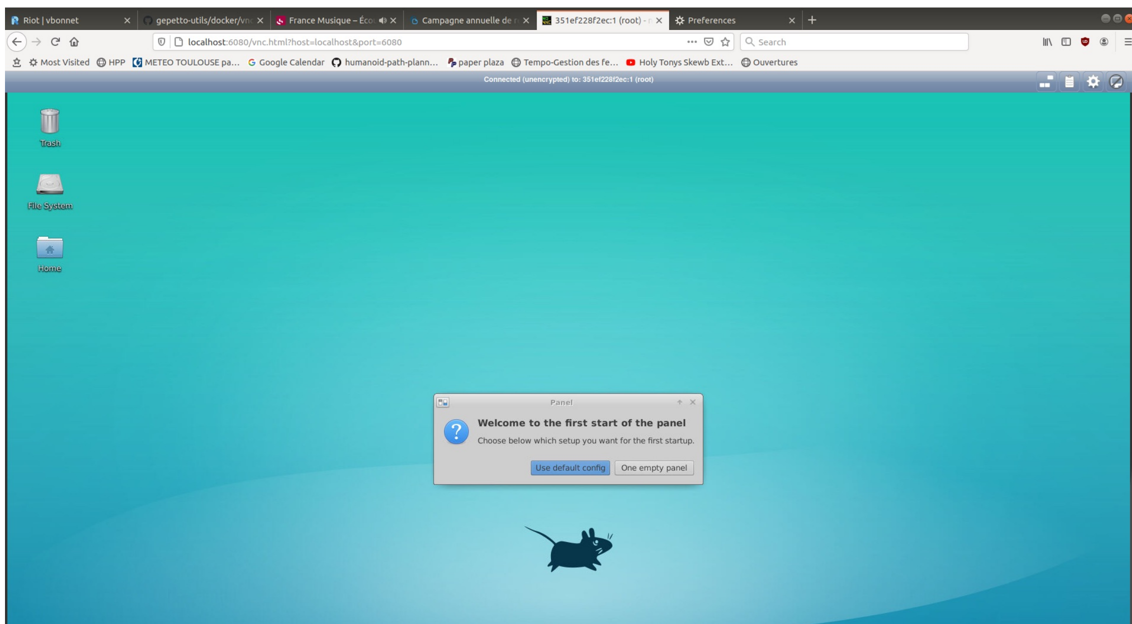


FIGURE 11.2 – Le gestionnaire de fenêtre XFCE

11.2.2 Stopper, redémarrer une image

Pour avoir la liste des containers :

```
docker container ps
```

La sortie typique est :

```
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                     NAMES
3d5bcf2a273c        cc925c2a2ebd        "/ros_entrypoint.sh ..." 10 hours ago        Up 9 hours
     0.0.0.0:6080->6080/tcp   ros-intro-vnc
```

L'identifiant du container est indiqué dans la première colonne. Ici il s'agit de 3d5bcf2a273c.
Pour stopper le container

```
docker container stop
```

Pour démarrer le container :

```
docker start -ai 3d5bcf2a273c
```

11.2.3 Copier un fichier de et vers un docker

Pour copier le fichier test.dat qui se trouve dans le repertoire /tmp vers le répertoire /root du container ros-intro-vnc

```
docker cp /tmp/test.dat ros-intro-vnc:/root/
```

Pour copier le fichier test.dat du répertoire root du container ros-intro-vnc vers le répertoire /tmp

```
docker cp ros-intro-vnc:/root/test.dat /tmp
```

11.3 Installer les images docker de ROS

Les images docker ROS sont disponibles ici à l'adresse suivante : https://hub.docker.com/_/ros
Par exemple pour installer l'image docker de melodic pour un robot il faut taper :

```
docker pull osrf/ros:melodic-desktop-full
```

En général le nom des images docker est nom_distribution_ros-type_install_ros-version_os avec :
— nom_distribution : Le nom de la distribution ROS
— type_install : le type d'installation ROS. Elle peuvent être soit de type destkop, desktop-full ou robot. Les deux premières correspondant à des installations sur des machines de développements, tandis que la dernière correspondant au robot.
— version_os : la version du système d'exploitation. Par exemple ici bionic correspondant au nom de la distribution Ubuntu 18

Pour démarrer une image docker de ROS classique :

```
docker run --name ros-melodic -it osrf/ros:melodic-desktop-full
```

11.3.1 Mise à jour de l'image

Une fois dans l'image vous pouvez faire

```
apt update
apt install
```

11.3.2 Erreurs

Il arrive parfois que vous ayez l'erreur suivante :

```
W: GPG error: http://security.ubuntu.com/ubuntu bionic-security InRelease: At least one invalid signature was
     encountered.
E: The repository 'http://security.ubuntu.com/ubuntu bionic-security InRelease' is not signed.
```

Il se peut simplement que docker prenne trop de place sur le disque typiquement jusqu'à 50 Go. Pour le vérifier :

```
docker system df
```

Pour résoudre le problème :

```
docker system prune
```

Cette commande retire les containers, réseaux et images non utilisées.

11.4 Construire l'image docker

Vous pouvez utiliser la repository https://github.com/aip-primeca-occitanie/ros-introduction-robmob

```
cd ros-vnc
docker build .
```

Pour créer un tag :

```
cd ros-vnc
docker build . -t aip-primeca-occitanie/ros-introduction-robmob
```

11.5 Mac

11.5.1 Interface X11 sous Mac

La première étape est d'installer le serveur XQuartz sur votre ordinateur. Il faut ensuite autoriser les connections des clients réseaux dans la configuration XQuartz.

Dans le terminal du Mac vous devez autoriser le terminal du docker à accéder au terminal X :

```
xhost + 127.0.0.1
```

Il faut ensuite rediriger la sortie du bash dans le docker vers le host du docker :

```
export DISPLAY=host.docker.internal:0
```

12. Mémo

ROS Mémo

Outils en ligne de commande pour le système de fichiers

rospack	Un outil pour inspecter les packages/stacks
roscd	Change de répertoire vers le paquet ou la stack.
rosls	Liste les paquets et les stacks.
catkin_create_pkg	Créer un nouveau paquet ROS.
rosdep	Installe le système des dépendences des paquets ROS.
catkin_make	Construit les paquets ROS.
rosws	Commande pour manipuler les workspaces.

Usage :
```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rosls [package[/subdir]]
$ catkin_create_pkg [package name]
$ catkin_make [package]
$ rosdep install [package]
$ roswtf or roswtf [file]
```

Outils en ligne de commande les plus utilisés

roscore

Une collection de **blue**nodes et de programmes qui prérequis pour une application ROS. Il est nécessaire d'avoir roscore pour permettre aux nodes ROS de communiquer.

roscore contient
- master
- parameter server
- rosout

Usage :
```
$ roscore
```

rosmsg/rossrv

rossmsg/rossrv affiche les champs des Message/Service (msg/srv)

Usage :

`$ rosmsg show`	Affiche les champs dans le message
`$ rosmsg md5`	Affiche les valeurs md5 des messages
`$ rosmsg package`	Liste tous les messages d'un paquet
`$ rosmsg packages`	Liste tous les packages avec un message

Exemples :

Affiche le msg Pose
```
$ rosmsg show Pose
```
Liste les messages du paquet nav_msgs
```
$ rosmsg package nav_msgs
```
Liste les packages ayant des messages
```
$ rosmsg packages
```

rosrun

rosrun permet d'exécuter un programme sans changer de répertoire :

Usage :
```
$ rosrun package executable
```
Exemple :

Lancer turtlesim
```
rosrun turtlesim turtlesim_node
```

rosnode

Affiche des informations sur les nodes ROS, c.a.d. les publications, souscriptions et connections.

Usage :

`$ rosnode ping`	Test la connectivité du node
`$ rosnode list`	Liste les nodes actifs
`$ rosnode info`	Affiche les informations d'un node
`$ rosnode machine`	Liste les nodes exécutés sur une machine particulière
`$ rosnode kill`	Termine l'exécution d'un node

Exemples :

Termine tous les nodes

```
$ rosnode kill -a
```

Liste tous les nodes sur une machine

```
$ rosnode machine sqy.local
```

Test tous les nodes

```
$ rosnode ping -all
```

roslaunch

Démarre les nodes localement et à distance via SSH, et initialise les paramètres.

Exemples :

Lancer un fichier d'un package

```
roslaunch package filename.launch
```

Lancer sur un port différent :

```
roslaunch -p 1234 package filename.launch
```

Lancer uniquement les nodes locaux :

```
roslaunch -local package filename.launch
```

rostopic

Un outil pour afficher des informations à propos des topics c.a.d des publishers, des subscribers, la fréquence de publications et les messages.

Usage :

`$ rostopic bw`	Affiche la bande passante d'un topic
`$ rostopic echo`	Affiche le contenu du topic
`$ rostopic hz`	Affiche la fréquence de mise à jour d'un topic
`$ rostopic list`	Affiche la liste des topics
`$ rostopic pub`	Publie des données sur un topic
`$ rostopic type`	Affiche le type d'un topic (message)
`$ rostopic find`	Affiche les topics d'un certain type

Exemples :

Publie hello à une fréquence de $10\,Hz$

```
$ rostopic pub -r 10 /topic_name std_msgs/String hello
```

Affiche le topic /topic_name

```
$ rostopic echo -c /topic_name
```

Test tous les nodes

```
$ rosnode ping -all
```

rosparam

Un outil permet de lire et écrire des parameters ROS sur le serveur de paramètre, en utilisant notamment des fichiers YAML

Usage :

`$ rosparam set`	Spécifie un paramètre
`$ rosparam get`	Affiche un paramètre
`$ rosparam load`	Charge des paramètres à partir d'un fichier
`$ rosparam dump`	Affiche les paramètres disponibles
`$ rosparam delete`	Enlève un paramètre du serveur
`$ rosparam list`	Liste le nom des paramètres

Exemples :

Liste tous les paramètres dans un namespace

```
$ rosparam list /namespace
```

Spécifier un paramètre *foo* avec une liste comme une chaîne, un entier, un réel.

```
$ rosparam set /foo "['1', 1, 1.0]"
```

Sauve les paramètres d'un namespace spécifique dans le fichier **dump.yaml**

```
$ rosparam dump dump.yaml /namespace
```

rosservice

Cet outil permet de lister et d'appeller les services ROS.

Usage :

`$ rosservice list`	Affiche les informations relatives aux services actifs
`$ rosservice node`	Affiche le nom d'un node fournissant un service
`$ rosservice call`	Appelle le service avec les arguments donnés
`$ rosservice args`	Liste les arguments d'un service
`$ rosservice type`	Affiche le type de service (arguments d'entrées et de sorties)
`$ rosservice uri`	Affiche l'URI du service ROS
`$ rosservice find`	Affiche les services fournissant un type

Exemples :

Appel un service à partir de la ligne de commande

```
$ rosparam call /add_two_ints 1 2
```

Enchaîne la sortie de rosservice à l'entrée de rossrv pour voir le type du service

```
$ rosservice type add_two_ints | rossrv show
```

Affiche tous les services d'un type particulier

```
$ rosservice find rospy_tutorials/AddTwoInts
```

Outils en ligne de commande pour enregistrer des données

rosbag

C'est un ensemble d'outils pour enregistrer et rejouer des données des topics ROS. Il a pour but d'être très performant et éviter la déserialisation et la reserialisation des messages.

rosbag record génère un fichier ".bag" (ainsi nommé pour des raisons historiques) dans lequel le contenu de tous les topics passés en arguments sont enregistrés.

Exemples :

Record all topics :

```
$ rosbag record -a
```

Record select topics :

```
$ rosbag record topic1 topic2
```

rosbag play prend le contenu d'un fichier ou plusieurs fichiers ".bag" et le rejoue synchronisé temporellement.

Exemples :

Rejoue tous les messages sans attendre :

```
$ rosbag play -a demo_log.bag
```

Rejoue tous les fichiers de bag à la fois :

```
$ rosbag play demo1.bag demo2.bag
```

Outils graphiques

rqt_graph

Affiche le graphe d'une application ROS.

```
$ rosrun rqt_graph rqt_graph
```

rqt_plot

Affiche les données relatives à un ou plusieurs champs des topics ROS en utilisant matplotlib.

```
$ rosrun rqt_plot rqt_plot
```

rosbag

Un outil pour visualiser, inspecter et rejouer des historiques de messages ROS (non temps réel)

Enregistre tous les topics

```
$ rosbag record -a
```

Rejoue tous les topics

```
$ rosbag play -a topic_name
```

rqt_console

Un outil pour afficher et filtrer les messages sur rosout

Affiche les consoles

```
$ rosrun rqt_console rqt_console
```

tf_echo

Un outil qui affiche les informations relatives à propos d'une transformation particulière entre un repère source et un repère cible

Utilisation

```
$ rosrun tf tf_echo <source_frame>
<target_frame>
```

Examples :

Pour afficher la transformation entre /map et /odom :

```
$ rosrun tf tf_echo /map /odom
```

view_frames

Un outil pour visualiser l'arbre complet des transformations de coordonnées

Usage :

```
$ rosrun tf view_frames
$ evince frames.pdf
```

Bibliography

Books

[Bro16] Bernard BROGLIATO. *Nonsmooth Mechanics*. Third edition. Springer International Publishing Switzerland, 2016 (cf. page 88).

Articles

[Sta+09] O. STASSE et al. « Strategies for humanoid robots to dynamically walk over large obstacles ». In : *IEEE Transactions on Robotics* 4 (2009), pages 960-967 (cf. page 88).

[Lee+18] J. LEE et al. « DART : Dynamic Animation and Robotics Toolkit ». In : *Journal of Open Source Software* 3.22 (2018) (cf. page 88).

[Set+18] A. SETH et al. « OpenSim : Simulating musculoskeletal dynamics and neuromuscular control to study human and animal movement ». In : *PLOS Computational Biology* (2018) (cf. page 88).

Chapitre de livres

Autres

[Goo16] GOOGLE. 2016. URL : `https://developers.google.com/project-tango/` (cf. page 10).