Introduction à ROS 2

Sommaire :

- Pourquoi ROS 2
- Distributions actuelles / Architecture ROS 2
- Le middleware DDS : introduction et *Quality of Service*
- Fichiers Launch
- Gestion du cycle de vie
- Publisher / Subscriber : Exemple
- Diagnostique
- Temps réel
- Tester ROS 2
- ...

Le développement de ROS 1

- Développé en 2007 pour faire fonctionner le robot humanoïde Willow Garage PR2
- Choix de faire beaucoup d'abstractions afin de pouvoir réutiliser les développements

Caractéristiques principales :

- Intelligence embarquée
- Temps réel non-nécessaire
- Nécessité d'une connectivité performante
- Plutôt pour des applications académiques
- Le plus flexible possible

Pourquoi ROS 2 ?

- De nouvelles technologies pertinentes sont apparues
- Projets industriels importants
- Coopération entre plusieurs robots
- Multi-OS
- Systèmes temps-réel
- Systèmes plus robustes pour des applications plus critiques

Développement de ROS 2 en parallèle de ROS : ROS fonctionne très bien dans sa plage d'utilisation.

Distributions actuelles

Distro	Release date	Logo	EOL date
Eloquent Elusor	Nov 22nd, 2019		Nov 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		Dec 2019

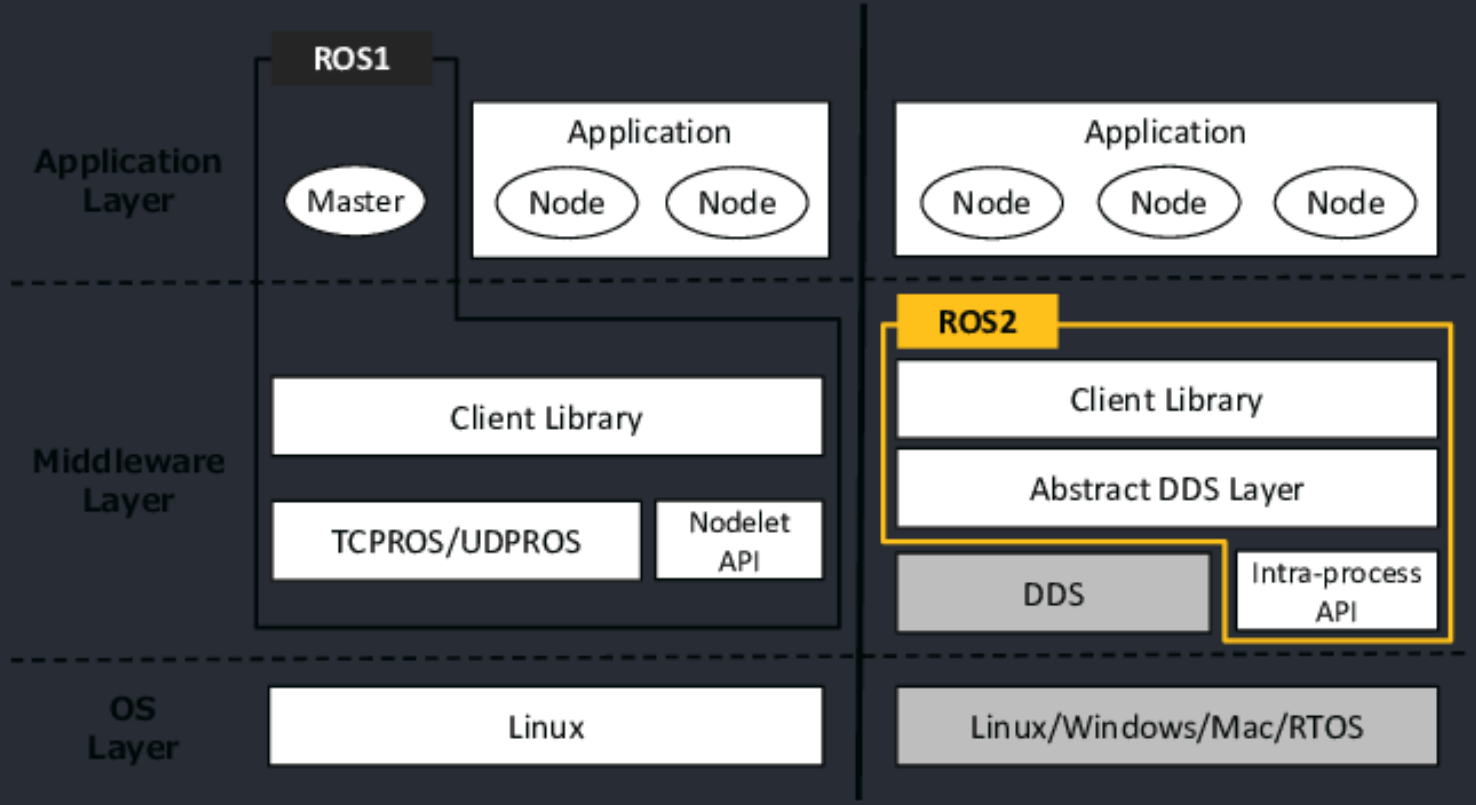
- Foxy Fitzroy en mai 2020

Les différences importantes

Applications	ROS	ROS2
Plates-formes	Testé sur Ubuntu Maintenu sur d'autres versions de Linux ainsi que sur OS X	Actuellement testé et supporté sur Ubuntu Xenial, OS X El Capitan ainsi que Windows 10
C++	C++03 // n'utilise pas pas les capacités de C++11 dans ses API	Utilise principalement C+++11Commencez et prévoyez d'utiliser C++14 & C+++17
Python	Cible Python 2	>= Python 3.5
Middleware	Format de sérialisation personnalisé (protocole de transport + mécanisme de découverte central)	Actuellement, toutes les implémentations de cette interface sont basées sur le standard DDS
Synchroniser la durée et les mesures de temps	La durée et les types de temps sont définis dans les bibliothèques clientes, ils sont codés en C++ et Python	Ces types sont définis comme des messages et sont donc cohérents d'un langage de programmation à l'autre
Composants avec cycle de vie	Chaque nœud a généralement sa propre fonction principale.	Le cycle de vie peut être utilisé par des outils comme roslaunch pour démarrer un système composé de nombreux composants de manière déterministe
Modèle multi-threads	Le développeur ne peut choisir qu'entre une exécution mono-thread ou multi-thread.	Des modèles d'exécution plus fins sont disponibles et des exécuteurs personnalisés peuvent être facilement implémentés.
Noeuds multiples	Il n'est pas possible de créer plus d'un nœud dans un processus.	Il est possible de créer plusieurs nœuds dans un processus.
roslaunch	Les fichiers de roslaunch sont définis en XML avec des capacités très limitées.	Les fichiers de lancement sont écrits en Python ce qui permet d'utiliser des logiques plus complexes comme les conditionnels, etc.

- **Raffinement des fonctionnalités de ROS**

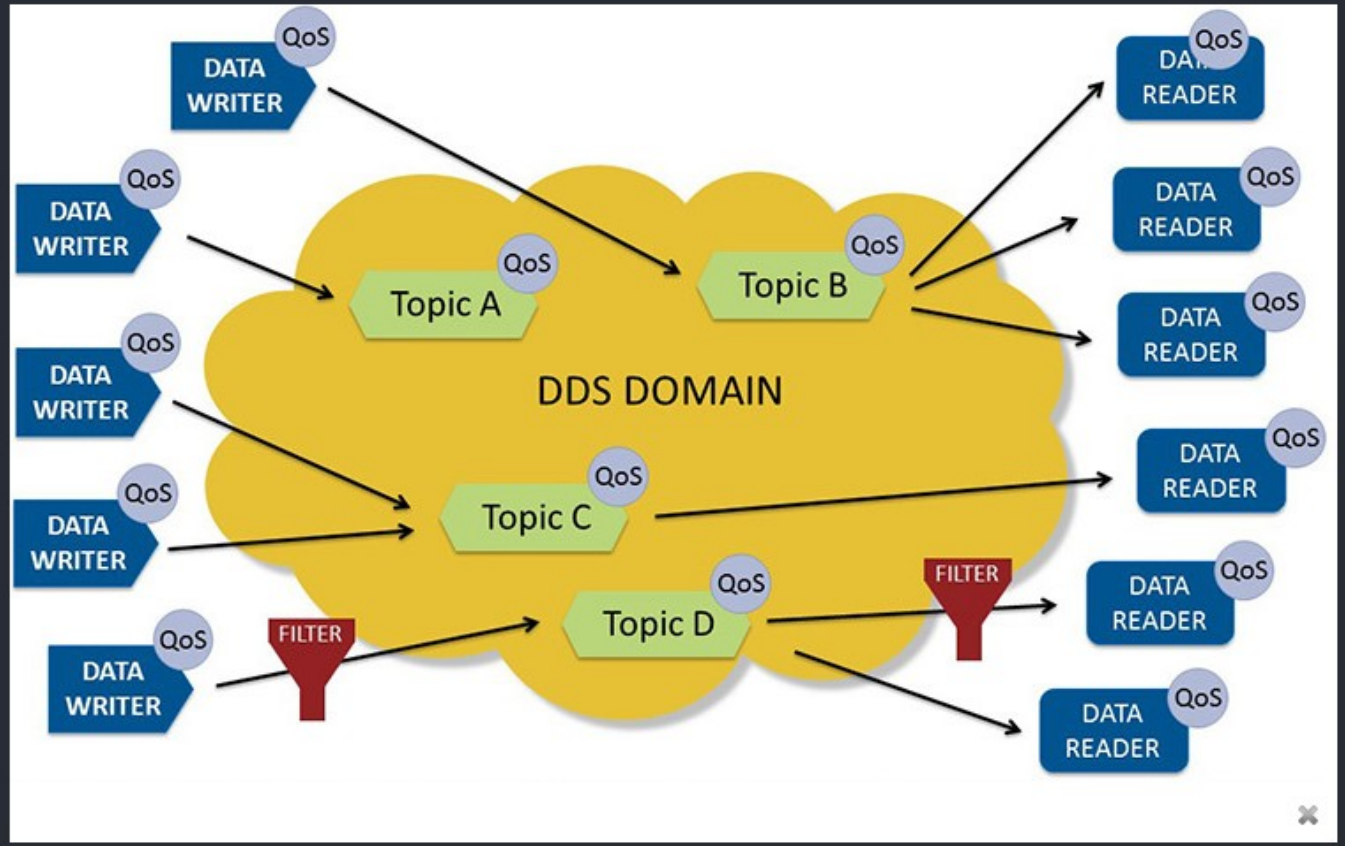
Architecture ROS 2



Construction de ROS 2 sur le middleware DDS (Data Distribution Service)

DDS : normes spécifiées par l'OMG proposant une technologie évoluée d'échanges de données en réseau.

- Utilisation par ROS pour la publication et la souscription de messages



- ROS 2 a été développé pour que l'implémentation de DDS soit cachée pour les développeurs de ROS 2
- Différents vendeurs de DDS : choix à ne pas négliger pour des applications spécifiques

DDS (2)

Intérêts :

- DDS remplace complètement le ROS master : pas de **point central**
- Moins de code à maintenir
- **API** déjà écrite
- **Communication flexible** : de nombreuses possibilités selon l'application (QOS : Quality of service)
- Optimisation du trafic (utilisation de la mémoire partagée...)
- Développé pour des application **temps-réel critiques**
- **Communauté active** afin de faire évoluer les caractéristiques

Installation de DDS : https://index.ros.org/doc/ros2/Installation/DDS-Implementations/

Quality of services

Différentes politiques (paramètres) de transmission de messages :

- **History :** *Keep last* or *Keep all*
  - Sauvegarde de l'ensemble ou des dernières données arrivées
- **Depth :**
  - Taille des données sauvegardées.
- **Reliability :** *Best effort* ou *reliable*
  - Robustesse de la transmission : perte autorisée si le réseau est mauvais ou garantie de la livraison des données (mais nécessite plus de temps)
- **Durability :** *Transient local* ou *Volatile*
  - Responsabilité ou non de la transmission accordée au publisher

QOS profiles

ROS 2 fournit ainsi des profiles regroupant des politiques selon l'usage :

- *Default QoS settings* : profiles ROS 1 : *volatile* and *keep last*
- *Services* : *reliable*...
- *Sensor data* : dernière données les plus importantes (*best effort* et *small queue*) ...
- *Parameters* : pas de perte et *large queue* ...

D'autres politiques et profiles spécifiques :

- https://github.com/ros2/rmw/blob/release-latest/rmw/include/rmw/qos_profiles.h
- https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/

Environnement ROS 2

`Colcon` est le nouvel outil de build qui regroupe `catkin_make`, `catkin_tools` et `ament_tools`

Lors de la création d'un workspace (`colcon built` à la place de `catkin_create_package`), à partir du répertoire src va créer :

- le répertoire `Build`
- le répertoire `install`
- le répertoire `log`

Un package utilisant `ament_cmake` utilise le même CMakeLists.txt A universal build tool : http://design.ros2.org/articles/build_tool.html

Les outils de build

Build system : compile un unique package.

- `Make` , `CMake` et même `Autotools`
- `catkin` basé sur `CMake` , permet d'automatiser certaines tâches
- `ament_cmake` : évolution de `catkin` pour ROS 2, permet plus de souplesse sur le paramétrage du package après la compilation, compilation sur tous les OS
- `Python setuptools`

Build tool : opère sur plusieurs packages et lance un build system pour chaque package.

- `catkin_make` : invoque les `Cmake` de l'ensemble des packages (`CMake` configurées par `catkin` ...) dans un unique contexte.
- `catkin_make_isolated` : permet d'éviter la compilation des packages dans un unique contexte (évite la parallélisation de la compilation, même namespace...)
- `ament_tools` : développé pour compiler les packages ROS 2
- `colcon` remplace maintenant `ament_tools`

Pour en savoir plus : http://design.ros2.org/articles/build_tool.html Différences entre `ament_cmake` et `catkin` : http://design.ros2.org/articles/build_tool.html

(*Attention à la configuration de l'environnement (`find_package` , CmakeLists.txt, `LD_LIBRARY_PATH` ...)*)

Fichiers Launch

ROS 2 fournit un framework `launch_ros` proposant un nombre important de fonctionnalités permettant de lancer, stopper les noeuds, déclencher différents événements...

Les fichiers Launch sont développés en Python.

Exemple du package `turtlesim` :

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

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

```
import launch
import launch.actions
import launch.substitutions
import launch_ros.actions


def generate_launch_description():
    return launch.LaunchDescription([
        launch.actions.DeclareLaunchArgument(
            'node_prefix',
            default_value=[launch.substitutions.EnvironmentVariable('USER'), '_'],
            description='Prefix for node names'),
        launch_ros.actions.Node(
            package='demo_nodes_cpp', node_executable='talker', output='screen',
            node_name=[launch.substitutions.LaunchConfiguration('node_prefix'), 'talker']),
    ])
```

Fichiers launch (2)

Fichier launch : exécuter un processus, générer des événements, renseigner des informations...

class: `launch.LaunchDescription` : lance le Launch

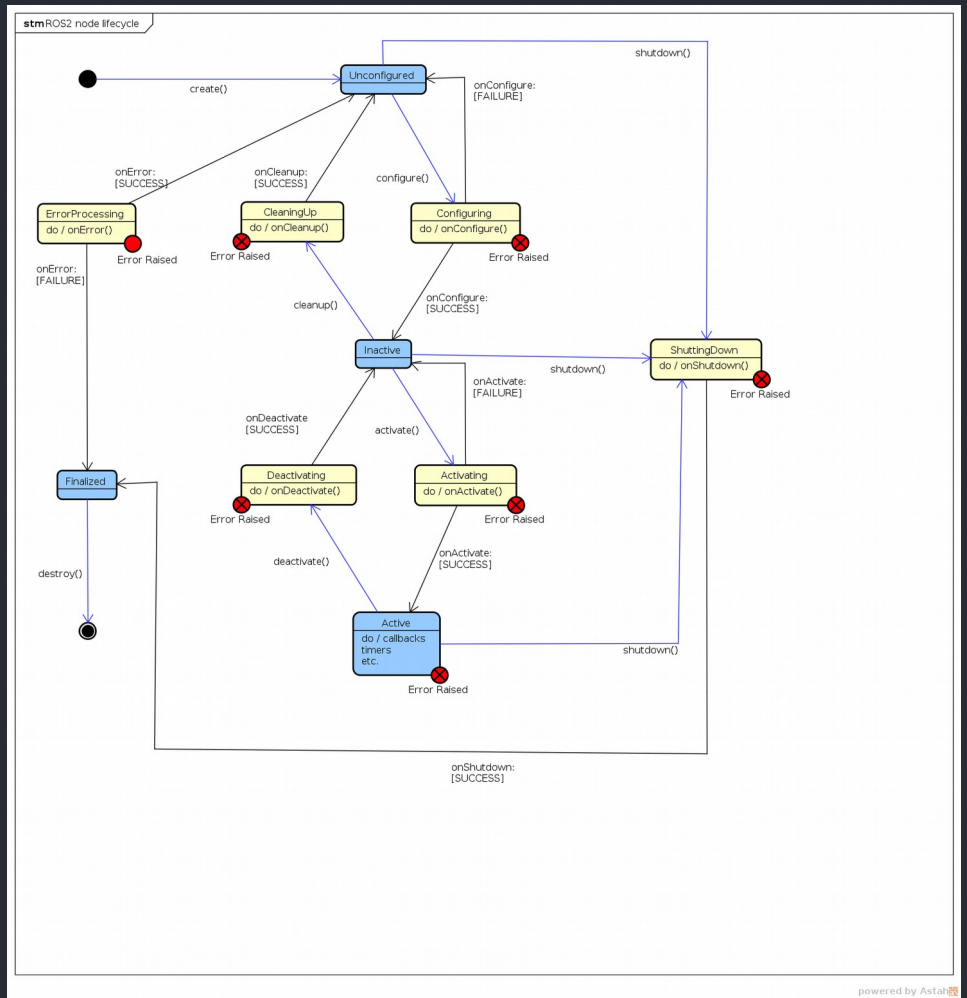
Différentes options proposées :

- Description :
- Actions (*class:* `launch.Action`) : ensemble de méthodes permettant à l'utilisateur d'interagir avec le système.
  - *IncludeLaunchDescription`*
  - *SetLaunchConfiguration`*
  - *SetEnvironmentVariable*
  - *TimerAction*
  - *EmitEvent*
- Substitutions (*class:* `launch.Substitutions`) : Arguments des méthodes Actions
  - *LaunchDescriptionArgument*
  - *EnvironmentVariable*
  - *FindExecutable*
  - *LaunchConfiguration*
- Launch Service et Event Handlers (*class:`launch.EventHandler*) : Réactions au événements extérieurs.

Architecture : https://github.com/ros2/launch/blob/master/launch/doc/source/architecture.rst

Gestion du cycle de vie des noeuds : `lifecycleNode`

Caractéristique essentielle pour la sécurité du robot : meilleur contrôle sur l'état du système :



Gestion du cycle de vie (2)

Définition de :

- 4 **états primaires** : unconfigured, inactive, active, finalized
- 6 **états transitoires** : configuring, cleaningup, activating ...
- 7 **transitions**

Utilisation : Héritage de la classe `class LifecycleTalker : public rclcpp_lifecycle::LifecycleNode`

```
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_configure(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_activate(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_deactivate(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_cleanup(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_shutdown(const rclcpp_lifecycle::State & previous_state)
```

Gestion du cycle de vie (3)

- Monitoring des états via des topics et dans des services.
- Changement d'états manuellement :
  - par le service `change_state`
  - par la commande : `ros2 lifecycle set`
- Changement d'états automatique via le launch :
  - On définit des transitions via l'attribut `EmitEvent` (`TRANSITION_CONFIGURE` , `TRANSITION_ACTIVATE` ...)
  - On définit ensuite des actions via `RegisterEventHandler` selon les états

En cours de développement...

Pour en savoir plus :

- http://design.ros2.org/articles/node_lifecycle.html
- https://index.ros.org/p/lifecycle/github-ros2-demos/
- exemple de roslaunch avec un lifecycleNode : https://www.stereolabs.com/docs/ros2/lifecycle/

Publisher en Ros 2

```
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()
```

Subscriber en Ros2

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

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


def main(args=None):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()
    rclpy.spin(minimal_subscriber)

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


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

Diagnostique : `ros2doctor`

La commande `ros 2doctor` va vérifier différents aspects de ROS : la version, le réseau, les tâches en cours, l'environnement... Afin d'informer le développeur sur des possibles erreurs.

1. Vérifier la configuration : `ros2 doctor` dans un terminal
2. Vérifier le système après avoir lancer un noeud en lançant `ros2 doctor`

```
UserWarning: Publisher without subscriber detected on /turtle1/color_sensor
UserWarning: Publisher without subscriber detected on /turtle1/pose.
```

3. Avoir un compte-rendu complet : `ros2 doctor --report`

```
NETWORK CONFIGURATION
...

PLATFORM INFORMATION
..  RMW MIDDLEWARE
...

ROS 2 INFORMATION
...

TOPIC LIST
```

Programmation temps-réel

- Respect des **dead-lines** et systèmes totalement **déterministes**.
- Le scheduler de Linux, Windows, Mac OS ne permet et/ou n'autorise pas par défaut ces applications
- Besoin du patch noyau RT_PREEMPT
- Nécessité d'un code qui délivre une exécution déterministe :
    - Gestion de la mémoire
    - Synchronisation des threads
    - Tester les performances
- De nombreux travaux sur une implémentation de librairies real-time.
- De nombreuses questions à résoudre...en cours de développement...

Pour en savoir plus :

- https://index.ros.org/doc/ros2/Tutorials/Real-Time-Programming/
- http://design.ros2.org/articles/realtime_background.html

Migrer de ROS à ROS2

- Création d'un pont entre ROS 1 et ROS 2 : https://github.com/ros2/ros1_bridge/blob/master/README.md
- Possibilité d'avoir de faire communiquer des publishers et des subscribers des différentes versions.

Tester ROS 2

Utilisation de docker au départ conseillé

```
docker pull osrf/ros:eloquent-desktop
docker run -it osrf/ros:eloquent-desktop
  root@<container-id>:/# ros2 pkg list
```

Images ROS : https://hub.docker.com/r/osrf/ros/

Références intéressantes :

- Introduction : https://www.generationrobots.com/blog/fr/ros2-quest-ce-qui-change-par-rapport-a-ros/
- DDS : https://design.ros2.org/articles/ros_on_dds.html
- **Documentation générale** : https://index.ros.org/doc/ros2/
- **ROS 2 design** : http://design.ros2.org/
- Exemple de codes: https://github.com/ros2/examples
- Démonstration de différents concepts : https://github.com/ros2/demos/tree/eloquent