Robótica Móvil

ROS: Robot Operative System

Taihú Pire

Laboratorio de Robótica

CONICET

UNIVERSIDAD NACIONAL DE ROSARIO

CONFINGERE HOMINEM COGITANTEM

CIFASIS

Motivación

Don't reinvent the wheel. Create something new and do it faster and better by building on ROS!

¿Qué es ROS?

ROS (Robot Operating System) es un kit de desarrollo de software de código abierto para aplicaciones de robótica. ROS ofrece una plataforma de software estándar para desarrolladores de todas las industrias que los llevará desde la investigación y la creación de prototipos hasta la implementación y la producción.

- Comunidad global
- Utilizado en cursos de robótica, investigación e industria
- Acorta los tiempos de producción
- Multi-dominio: indoor / outdoor, hogareño / industrial, bajo el agua / espacio
- Multi-plataforma: Linux, Windows y macOS.
- Open-source
- Licencia permisiva (Apache 2.0)
- Soporte desde la industria

¿Qué es ROS?

Trabajaremos con la versión Ubuntu LTS más actual, esta siempre viene con una versión de ROS estable.

ROS1

ROS2

Instalar ROS

- Seguiremos la guía: https://docs.ros.org/en/<distro>/Tutorials.html
- Instalar ROS

```
sudo apt install ros-<distro>-<package>
```

- Sourcear los archivos de setup

```
source /opt/ros/<distro>/setup.bash
```

- Agregar el source al .bashrc

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

- Chequear las variables de entorno

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

- Output:

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

- Chequear el setup de ROS y problemas potenciales (lista de paquetes instalados, versiones, etc)

```
ros2 doctor --report
```

Configuración de entorno de ROS: ROS_DOMAIN_ID

- El middleware predeterminado que utiliza ROS para la comunicación es DDS (*Data Distribution Service*). En DDS, los ID de Dominio permiten que diferentes redes lógicas (cada red lógica tendrá asociado un ID de dominio) compartan una misma red física.
- Los nodos en el mismo dominio pueden descubrir y enviarse mensajes libremente, mientras que los nodos en diferentes dominios no pueden.
- Todos los nodos utilizan el ID de dominio 0 de forma predeterminada.

```
echo "export ROS_DOMAIN_ID=<your_domain_id>" >> ~/.bashrc
```

Ejemplo: talker y listener

- Primero verifiquemos que esten instalados los ejecutables `talker` y `listener` de los paquetes `demo_nodes_cpp` y `demo_nodes_py`

```
ros2 pkg executables
```

- En una terminal ejecutar (un talker en c++):

```
source /opt/ros/<distro>/setup.bash
ros2 run demo_nodes_cpp talker
```

- En otra terminal ejecutar (un listener en python):

```
source /opt/ros/<distro>/setup.bash
ros2 run demo_nodes_py listener
```

Ejemplo: Turtlesim

- Instalamos el paquete `ros-humble-turtlesim`

```
sudo apt install ros-humble-turtlesim
```

- Verificamos los ejecutables dentro del paquete

```
ros2 pkg executables turtlesim
```

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

- Corremos los ejecutables:

```
ros2 run turtlesim turtlesim_node
```

```
ros2 run turtlesim turtle_teleop_key
```

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

Ejemplo: Turtlesim (Continuación)

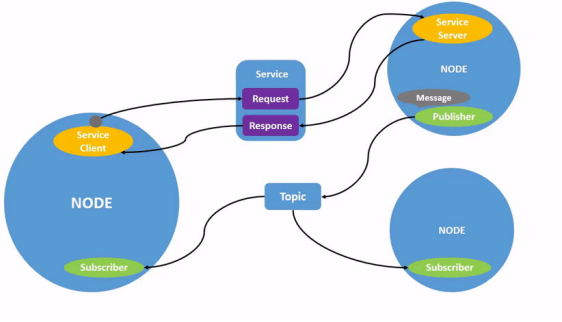
- Abrir `rqt`

```
rqt
```

- Plugins → Services → Service Caller y spawn una nueva tortuga.

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

ROS Graph



- El grafo de ROS es una red de elementos que procesan datos al mismo tiempo. Abarca todos los ejecutables y las conexiones entre ellos.
- Cada nodo en ROS debe ser responsable de un solo propósito de módulo (por ejemplo, un nodo para controlar los motores de las ruedas, un nodo para controlar un telémetro láser, etc.). Cada nodo puede enviar y recibir datos a otros nodos a través de tópicos, servicios, acciones o parámetros.
- Un sistema robótico completo se compone de muchos nodos que trabajan en conjunto. En ROS, un único ejecutable (programa C++, programa Python, etc.) puede contener uno o más nodos.

Volvamos al Turtlesim para ver otros conceptos

El comando `ros2 run` corre un ejecutable desde un paquete.

```
ros2 run <package_name> <executable_name>
```

La `ros2 node list` muestra los nombres de todos los nodos en ejecución.

```
ros2 node list
```

Es posible hacer *remapping*, es decir mapear propiedades predeterminadas de un nodo, como nombre de nodo, nombres de tópicos, nombres de servicio, etc., a valores personalizados.

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

Obtener información de un nodo

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

Volvamos al Turtlesim para ver otros conceptos (cont.)

Visualizar los nodos y la comunicación entre los mismos:

```
rqt_graph
```

Devuelve la misma lista de tópicos, esta vez con el tipo de tema entre paréntesis:

```
ros2 topic list -t
```

Ver los datos publicados en un tópico:

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

Ejemplo:

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

Obtener información acerca de los publicadores y subscriptores de un tópico:

```
ros2 topic info
```

Obtener detalles sobre un tipo de mensaje:

```
ros2 interface show <msg type>
```

Ejemplo:

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

Volvamos al Turtlesim para ver otros conceptos (cont.)

Publicar mensajes utilizando la línea de comandos:

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

Ejemplo:

tiny

```
ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z
    : 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

Saber la prefuencia en que se publica un mensaje

```
ros2 topic hz <topic_name>
```

Archivos launch

- A medida que se crean sistemas más complejos con más y más nodos ejecutándose simultáneamente, abrir terminales y volver a ingresar los detalles de configuración se vuelve tedioso.
- Los archivos *launch* permiten iniciar y configurar varios ejecutables que contienen nodos ROS 2 de manera simultáneamente.
- Ejecutar un solo archivo launch con el comando `ros2 launch` iniciará todo su sistema, todos los nodos y sus configuraciones, a la vez.
  Correr un archivo launch

```
ros2 launch <package_name> <launch_file_name> <launch_arguments>
```

Ejemplo:

```
ros2 launch turtlesim multisim.launch.py
```

Crear workspace

- Crear workspace

```
mkdir -p ~/dev_ws/src
cd ~/dev\_ws/src
```

- Un *workspace* es un directorio que contiene paquetes de ROS 2.
- Antes de usar ROS 2, es necesario obtener su espacio de trabajo de instalación de ROS 2 en la terminal en la que planea trabajar. Esto hace que los paquetes de ROS 2 estén disponibles para que los use en esa terminal. Para esto hacemos:

```
source /opt/ros/<distro>/setup.bash
```

- Agregar un paquete de ejemplo al workspace (dentro de `~/dev\_ws/src`)

```
git clone https://github.com/ros2/examples examples -b <distro>
```

```
git clone https://github.com/ros/ros\_tutorials.git -b <distro>-devel
```

Compilar el workspace

- Instalar colcon

```
sudo apt install python3-colcon-common-extensions
```

- En la raíz del worklspace, ejecutar

```
colcon build --symlink-install
```

  El flag `--symlink-install` permite usar enlaces simbólicos, en lugar de copiar archivos a las carpetas de ROS2 durante la instalación, siempre que sea posible. Cada paquete en ROS2 debe instalarse y todos los archivos utilizados por los nodos deben copiarse en las carpetas de instalación.

- El build genera los directorios: `build`, `install` y `log`.

Compilar el workspace

- Cuando `colcon` haya completado el build, la salida estará en el directorio `install`.
- `colcon` genera archivos `bash/bat` en el directorio `install` para ayudar a configurar el entorno. Estos archivos agregarán todos los elementos requeridos a su ruta y rutas de biblioteca, así como también proporcionarán cualquier comando bash o shell exportado por los paquetes.
- Ejemplo de uso: (no olvidar `source install/setup.bash`)

```
ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function
```

```
ros2 run examples_rclcpp_minimal_publisher publisher_member_function
```

- El comando `colcon_cd` le permite cambiar rápidamente el directorio de trabajo actual al de un paquete.

Creando un paquete en ROS2

- Un paquete puede considerarse un contenedor para código ROS 2.
- La creación de paquetes en ROS 2 utiliza `ament` como sistema de compilación y `colcon` como herramienta de compilación.
- Hay paquetes que utilizan CMake, y otros Python:
  - CMake:
    - `package.xml` contiene meta información del paquete
    - `CMakeLists.txt` describe cómo compilar el código dentro del paquete
  - Python:
    - `package.xml` contiene meta información del paquete
    - `setup.py` contiene instrucciones para de cómo instalar el paquete
    - `setup.cfg` es requerido cuando un paquete tiene ejecutables, así ROS 2 puede encontrarlos
    - `/<package_name>` un directorio con el mismo nombre del paquete. Este es usado por ROS 2 para encontrar el paquete, contiene el `__init__.py`

Creando un paquete en ROS2

- Crear un paquete:

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

- El argumento opcional `--node-name` permite crear un ejecutable con un nombre dado en el paquete.

```
ros2 pkg create test_pkg --build-type ament_cmake --node-name <node_name>
    <package_name>
```

Ejemplo: creemos un paquete que contenga Publicador y Subscriptor

```
#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;
}
```

```
ros2 run cpp_pubsub talker
```

```
ros2 run cpp_pubsub listener
```

Ejemplo: creemos un paquete que contenga Publicador y Subscriptor

```
#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 & 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;
}
```

Ejemplo: creemos un paquete que contenga Publicador y Subscriptor

```
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)

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

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

ament_package()
```

```
ros2 run cpp_pubsub talker
```

```
ros2 run cpp_pubsub listener
```

Rosbags (`ros2 bag`)

- `ros2 bag` es una herramienta de línea de comandos para grabar datos publicados sobre tópicos en su sistema.
- Acumula los datos transmitidos sobre cualquier número de tópicos y los guarda en una base de datos.
- Luego, puede reproducir los datos para reproducir los resultados de sus pruebas y experimentos.
- Grabar tópicos también es una excelente manera de compartir el trabajo y permitir que otros lo reproduzcan.

```
ros2 bag record <topic_name>
ros2 bag info <bag_file_name>
ros2 bag play <bag_file_name>
ros2 bag compress <bag_file_name>
ros2 bag decompress <bag_file_name>
...
```

Ejemplo completo: Navegación con Turtlebot3

- Ver la Guía: `https://navigation.ros.org`
- Instalar los paquetes de Nav2

```
sudo apt install ros-<ros2-distro>-navigation2 ros-<ros2-distro>-nav2-bringup
```

- Instalar los paquetes de Turtlebot 3

```
sudo apt install ros-<ros2-distro>-turtlebot3*
```

- Setear las variables de entorno y sourcear Gazebo (recuerde agregar estas líneas a `.bashrc`):

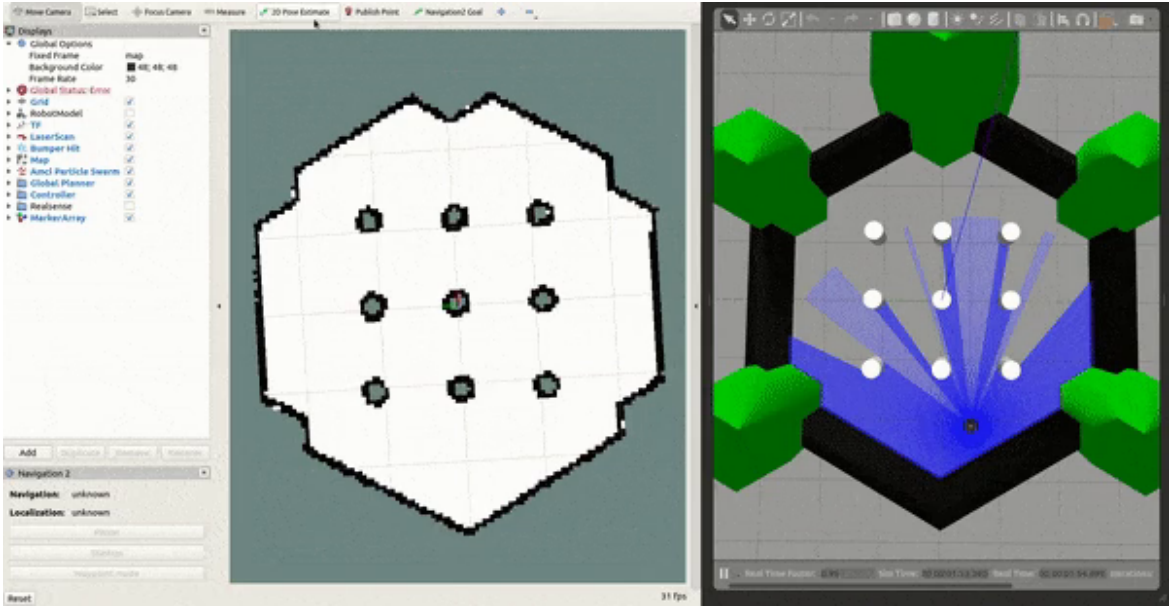
```
export TURTLEBOT3_MODEL=waffle
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:/opt/ros/<ros2-distro>/share/
    turtlebot3_gazebo/models
source /usr/share/gazebo/setup.bash
```

- Lanzar la simulación. El parámetro `headless:=False` permite visualizar el simulador Gazebo.

```
ros2 launch nav2_bringup tb3_simulation_launch.py headless:=False
```

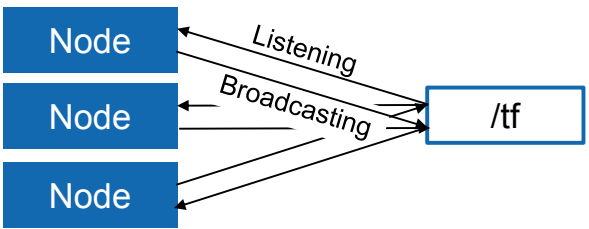
https://ros2-industrial-workshop.readthedocs.io/en/latest/_source/navigation/ROS2-Navigation.html
https://docs.ros.org/en/humble/Tutorials/Advanced/Simulators/Ignition.html
https://navigation.ros.org/index.html

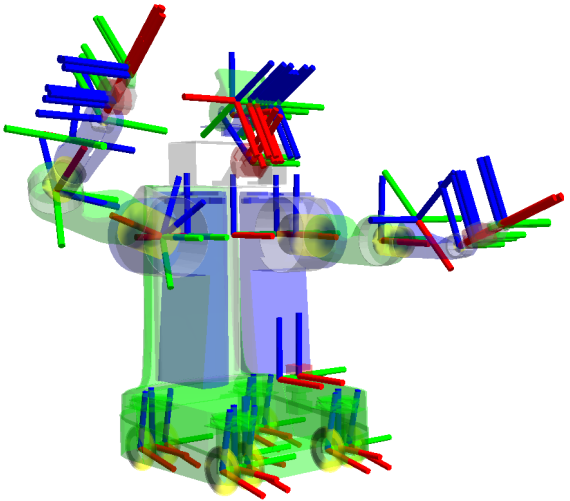
Ejemplo completo: Navegación con Turtlebot3



TF: transformaciones en ROS

- Herramienta para seguir los marcos de coordenada a través del tiempo
- Mantiene las relaciones entre los marcos de coordenadas en una estructura de árbol buffereado en el tiempo
- Permite transformar puntos, vectores, etc entre marcos de coordenadas para un determinado tiempo
- Implementado como un modelo publicador/subscriptor en los tópicos /tf y /tf_static





TF: transformaciones en ROS

- TF listeners usan un buffer para escuchar las transformaciones broadcasteadas
- Se puede consultar una transformación específica al árbol de transformaciones
- El tipo de mensaje de tf2 (`tf2_msgs/TFMessage.msg`)

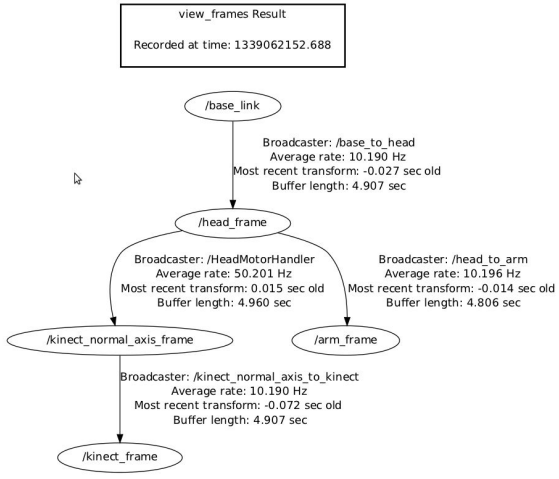
```
geometry_msgs/TransformStamped[] transforms
```

- El tipo de mensaje `geometry_msgs/TransformStamped`

```
std_msgs/Header header
uint32 seqtime stamp
string frame_id
string child_frame_id
geometry_msgs/Transform transform
```

- El tipo de mensaje `geometry_msgs/Transform`

```
geometry_msgs/Vector3 translation
geometry_msgs/Quaternion rotation
```



TF: transformaciones en ROS

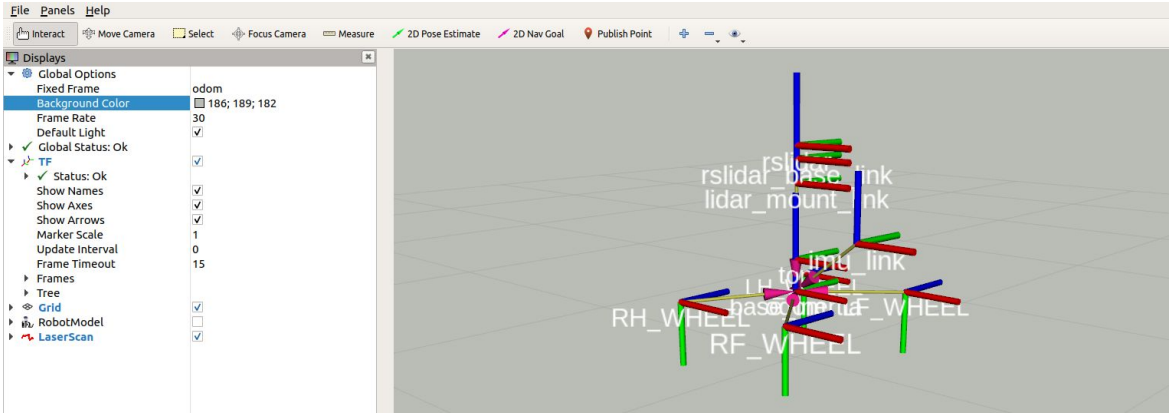
- Visualizar el árbol de transformaciones:

```
ros2 run tf2_tools view_frames
```

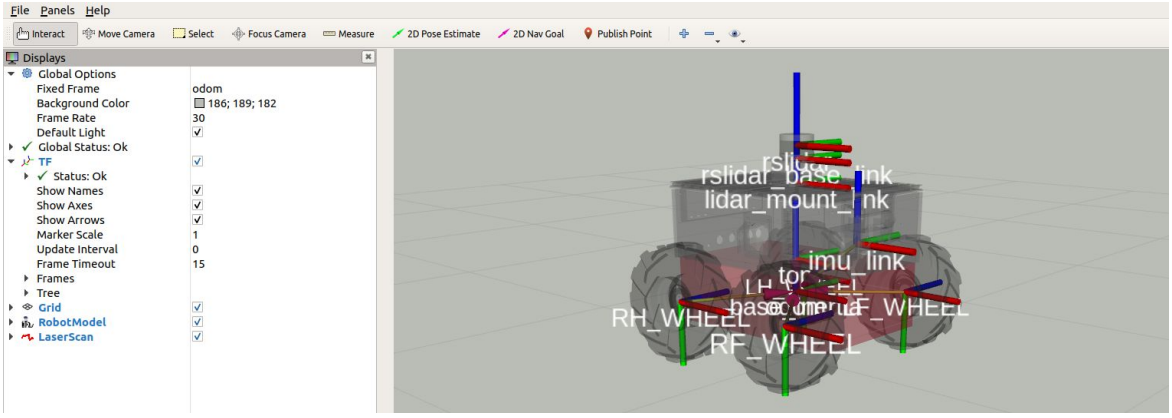
- Imprimir información a cerca de la transformación entre dos marcos de coordenadas:

```
ros2 run tf2_ros tf2_echo [reference_frame] [target_frame]
```

Visualizar transformaciones en RViz (`rviz2`)



Visualizar transformaciones en RViz (`rviz2`)



Ejemplo: TF2 con turtles

- Tutorial: `https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Introduction-To-Tf2.html`
- Instalación

```
sudo apt-get install ros-<distro>-turtle-tf2-py ros-<distro>-tf2-tools ros-humble-tf-
    transformations
```

- En una terminal, lanzar el launch `turtle_tf2_demo.launch.py`

```
ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
```

- En otra terminal, ejecutar `turtle_teleop_key`

```
ros2 run turtlesim turtle_teleop_key
```

- Visualizar el árbol de transformaciones:

```
ros2 run tf2_tools view_frames
```

- `tf2_echo` informa la transformación entre dos cuadros cualesquiera transmitidos a través de ROS.

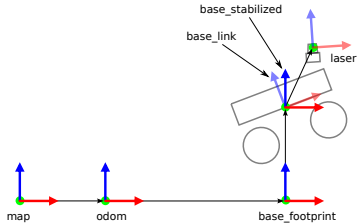
```
ros2 run tf2_ros tf2_echo [source_frame] [target_frame]
```

```
ros2 run tf2_ros tf2_echo turtle2 turtle1
```

- Visualizar en rviz2

```
ros2 run rviz2 rviz2 -d $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz
```

Sistemas de coordenadas para plataformas Móviles



- El marco de coordenadas llamado `/base_link` está rígidamente unido a la base del robot móvil. El `/base_link` es el marco de referencia del robot.
- El marco `/odom` contiene la pose del robot en el mundo según lo informado por algún sensor de odometría (Ej: odometría de ruedas, Visual-Inertial Odometry, etc). La pose del robot en el marco de coordenadas `/odom` es continua; sin embargo, debido a errores en la odometría, el error se acumula con el tiempo.
- Para corregir la desviación de los sensores odométricos, algunos métodos de localización introducen una transformación entre `/map` → `/odom`. Esta transformación es la corrección que se ha calculado en función de la entrada de sensores, como LiDAR o GNSS. La pose del robot en `/map` no es continua (puede saltar en varios intervalos de tiempo) porque los métodos de localización que dan este tipo de pose normalmente realizan correcciones a una frecuencia más baja.
- Si tienes un sensor de odometría perfecto y un GNSS perfecto, la transformación `/map` mapa → `/odom` tf será la identidad.

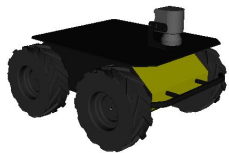
Sistemas de coordenadas para plataformas Móviles

mejorar slide con REP-103 y REP-105

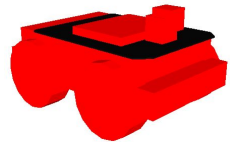
- Ver documento REP-105: `https://ros.org/reps/rep-0105.html#coordinate-frames`
- Ver documento REP-103: `https://www.ros.org/reps/rep-0103.html`

Unified Robot description Format (URDF)

- Se define en un archivo XML la representación de un modelo del robot
  - Descripción de la Cinemática y Dinámica del robot
  - Representación Visual
  - Modelo de colisión
- La generación del archivo URDF puede ser a través de un script en XACRO



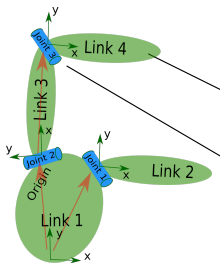
Modelo Visual



Modelo de colisión

Unified Robot description Format (URDF)

- La descripción consiste en un conjunto de elementos links y un conjunto de elementos joints
- Los joints conectan los links entre sí



robot.urdf

```
<robot name="robot">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint> .... </joint>
  <joint> .... </joint>
  <joint> .... </joint>
</robot>
```

```
<link name="link_name">
  <visual>
    <geometry>
      <mesh filename="mesh.dae"/>
    </geometry>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="0.4" ixy="0.0" .../>
  </inertial>
</link>
```

```
<joint name="joint_name" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" upper="0.548" ... />
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="parent_link_name"/>
  <child link="child_link_name"/>
</joint>
```

Unified Robot description Format (URDF)

- La descripción del robot (URDF) está guardada en un servidor de parámetros como `/robot_description`
- Se puede visualizar el robot en RViz con el plugin `RobotModel`

`control.launch`

```
...
<include file="$(find smb_description)/launch/load.launch">
<arg name="simulation"
value="$(arg simulation)"/>
<arg name="description_name" value="$(arg robot_description)"/>
<arg name="description_file" value="$(arg description_file)"/>
<arg name="wheel_joint_type" value="continuous"/>
<arg name="robot_namespace"
value="$(arg robot_namespace)"/>
</include>
...
```

`load.launch`

```
...
<param name="$(arg description_name)" command="$(find xacro)/xacro
$(arg description_file)
wheel_joint_type:=$(arg wheel_joint_type)
simulation:=$(arg simulation)
robot_namespace:=$(arg robot_namespace)
lidar:=$(arg lidar)
description_name_xacro:=$(arg description_name)
publish_tf:=$(arg publish_tf)"/>
</launch>
...
```

Simulation Description Format (SDF)

- Se define en un archivo XML para describir
  - Entorno (luces, gravedad, etc)
  - Sensores
  - Robots
- SDF es el formato estándar de Gazebo
- Gazebo convierte automáticamente URDF a SDF

Estrategias de Debugging

- Compilar y correr código frecuentemente para atrapar errores tempranamente
- Entender los mensajes de error de compilación y ejecución
- Usar herramientas de análisis para verificar el flujo de los datos: `ros2 node info`, `ros2 topic echo`, `ros2 wtf`, `rqt_graph`, etc.
- Visualizar y plotear los datos: `RViz`, `RQT Multiplot`, `rqt_bag`, etc.
- Dividir el programa en pequeños pasos y verificar resultados intermedios (`ROS_INFO`, `ROS_DEBUG`, etc)
- Hacer el código robusto verificando argumentos de entrada y utilizar try-catch para atrapar excepciones
- Extender y optimizar el código solo cuando la versión básica este funcionando
- Si las cosas no tienen sentido, limpiar el workspace
- Debugguear con una IDE (utilizar breakpoints)
- Escribir Unit-tests y test de integración para encontrar regresiones

Tarea

- Ver cómo crear servicios
- Ver cómo crear mensajes
- Uso de parámetros en un nodo
- `rqt_bag`
- `rqt_logger_level`
- `rqt_console`
- `rqt_graph`
- `rqt_multiplot`
- `rqt_image_view`
- `foxglove` (Alternativa a `RViz`)

Bibliografía

[1] Carol Fairchild y Thomas L Harman. *ROS Robotics by Example: Bring Life to Your Robot Using ROS Robotic Applications*. Packt Publishing, 2016. ISBN: 1785286706. DOI: http://hdl.handle.net/10657.1/888.