RESUMEN TEORÍA ARQUITECTURA SOFTWARE PARA ROBOTS

ÍNDICE DE CONTENIDOS:

[] TEMA 1: INTRODUCCIÓN

CÓDIGOS UTILIZADOS EN ESTE TEMA: my_package.

} 1.1 CONCEPTOS BÁSICOS

Middleware: Capa de software entre el sistema operativo y el usuario. Son aplicaciones para realizar la programación de las mismas en algunos dominios. Generalmente contiene más que bibliotecas y drivers, incluido el desarrollo y su metodología, además de herramientas de seguimiento.
Capas del software de un robot: Middleware para robots (aplicaciones robóticas y herramientas), sistema operativo y hardware del robot/computador (otras aplicaciones).
ROS (Robot Operating System): No es un sistema operativo que reemplaza a Windows o Linux, sino un middleware que aumenta las capacidades del sistema para desarrollar aplicaciones robóticas. El número 2 en ROS2 hace referencia a que es la segunda generación de este middleware.
Distribución: Colección de bibliotecas, herramientas y aplicaciones cuyas versiones se verifican para funcionar juntas correctamente. Cada distribución tiene un nombre y está vinculada a una versión de Ubuntu.

} 1.2 LA COMUNIDAD

Comunidad: Gran conjunto de desarrolladores que contribuyen con sus propias aplicaciones y utilidades a través de repositorios públicos, a los que otros desarrolladores pueden contribuir.
Open Source: Software publicado bajo una licencia en la que el usuario tiene derechos de uso, cambio, estudio y redistribución (Apache 2 y BSD en ROS2).

} 1.3 EL GRAFO COMPUTACIONAL

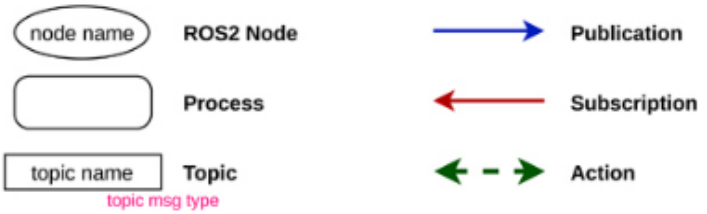
Grafo computacional: Es una aplicación de ROS2 en ejecución formada por nodos y arcos (carácter dinámico). Además, estos nodos (elementos primarios de ejecución en ROS2) se comunican entre sí para que el robot pueda realizar algunas tareas.
Topic: Canal de comunicación que acepta mensajes de un tipo único (tipo de comunicación más común en ROS2).

} 1.4 PARADIGMAS DEL GRAFO COMPUTACIONAL

Publicación/suscripción: Comunicación asíncrona donde N nodos publican mensajes a un topic que llegan a M suscriptores.
Servicios: Comunicación síncrona en la que un nodo envía una solicitud a otro y espera una respuesta, que debe ser rápida para no afectar al ciclo de control del nodo que llama al servicio (mapas).
Acciones: Comunicaciones asíncronas en las que un nodo realiza una solicitud a otro nodo, la cual suele llevar tiempo para completarse ya que el nodo solicitante puede recibir periódicamente retroalimentación o una notificación de que ha finalizado con éxito o con algún código de error (navegación).

} 1.5 MODELOS DE EJECUCIÓN

Ejecución iterativa: Ejecución del ciclo de control de un nodo a una frecuencia específica, permite controlar cuántos recursos computacionales requiere un nodo manteniendo el flujo de salida constante.
Ejecución orientada a eventos: Está determinada por la frecuencia con la que ocurren ciertos eventos, en este caso, la llegada de los mensajes al nodo.



/mobile_base/event/bumper: El Kobuki publica un mensaje del tipo kobuki_msgs/msg/BumperEvent cada vez que el bumper cambie de estado (nodos de colisión).
/mobile_base/commands/velocity: El driver del Kobuki se suscribe a este topic para ajustar la velocidad, si no recibe nada en un segundo éste se detiene, y pertenece al tipo geometry_msgs/msg/Twist.

} 1.6 NOMBRES EN ROS2

name	my_node = none	my_node = my_ns
my_topic	/my_topic	/my_ns/my_topic
/my_topic	/my_topic	/my_topic
~my_topic	/my_node/my_topic	/my_ns/my_node/my_topic

} 1.7 EL WORKSPACE

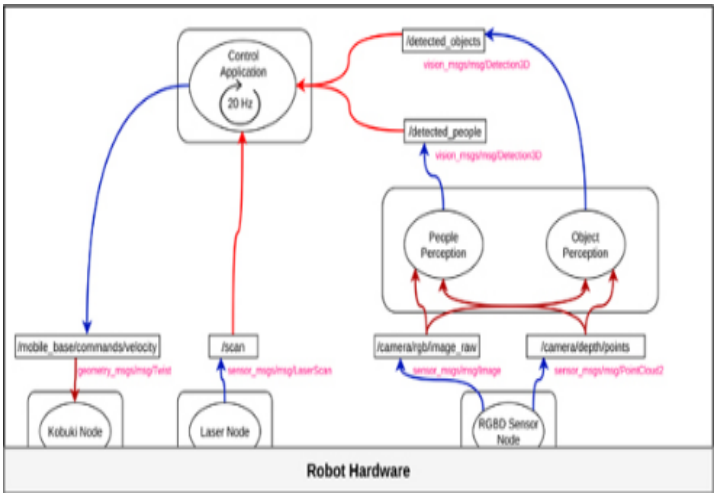
Workspace: Conjunto de software instalado en el robot u ordenador, además de los programas que desarrolla el usuario (carácter estático). Es un directorio que contiene paquetes y debe estar activo para que todo su contenido esté disponible para su uso.
Tipos de workspace: Underlay (workspace inicial), overlay (siguiente workspace creado). Las dependencias del overlay deben satisfacerse en el underlay, y si un paquete del overlay ya existe en el underlay lo esconde.
Paquete: Contiene bibliotecas, ejecutables o definiciones de mensajes con un objetivo común, y depende de otros paquetes o ejecutables para poder construirse.
Diseño de ROS2: Código de usuario (nodos de usuario), capa de cliente (rcl, rclcpp, rclpy, rclc, etc), capa de middleware (rmw, fast dds, cyclone dds, rti dds), capa de sistema operativo (Windows, Linux, iOS, etc).

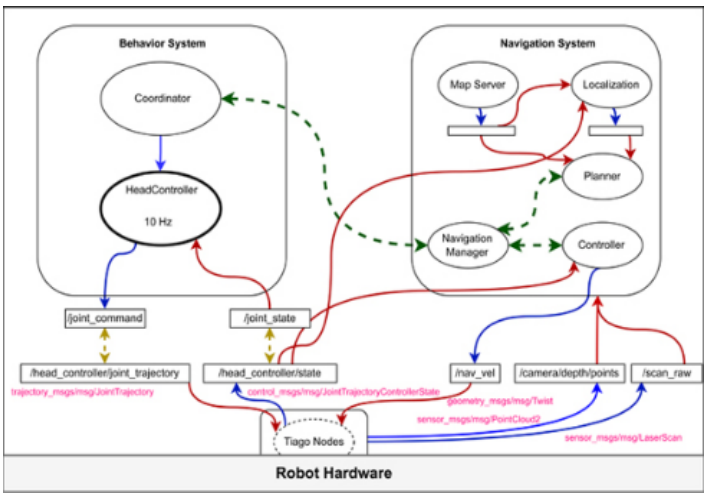
} 1.8 CREACIÓN DE UN WORKSPACE

Directorio build: Contiene los archivos intermedios de compilación (pruebas y archivos temporales).
Directorio install: Contiene los resultados de la compilación y todos los archivos necesarios para ejecutarlos (ficheros de configuración, nodos, mapas, etc …), con –symlink creamos un enlace hacia su ubicación original (carpeta build o src) en vez de copiarla.
Directorio log: Contiene un registro del proceso de compilación o prueba.

```
$ cd
$ mkdir -p <my_workspace>/src
$ cd <my_workspace>/src
$ git clone https://github.com/<usuario>/<my_workspace>.git
$ cd ~/<my_workspace>/src
$ vcs import . < <my_workspace>/third_parties.repos
$ cd ~/<my_workspace>
$ rosdep install –from-paths src –ignore-src -r -y
$ cd ~/<my_workspace>
$ colcon build –symlink-install
$ source ~/<my_workspace>/install/setup.bash
$ echo “source ~/<my_workspace>/install/setup.bash” >> /.bashrc
```

} 1.9 GRAFOS COMPUTACIONALES DE UNA CÁMARA Y DE UN ROBOT





[] TEMA 2: PRIMEROS PASOS CON ROS2

CÓDIGOS UTILIZADOS EN ESTE TEMA: br2_basics(logger, logger_class, publisher_class, pub_sub_v1_launch.py, pub_sub_v2_launch.py, param_reader, param_node_v1_launch.py, params.yaml), br2_tiago.

} 2.1 COMANDOS BÁSICOS DE ROS2

ros2: Comprueba que ya están activos tanto el underlay (/opt/ros/humble) como el overlay (~/<my_workspace>).
ros2 (estructura): ros2 <command> <verb> [<params> | <option>] *.
ros2 pkg list: Muestra la lista de todos los paquetes disponibles.
ros2 pkg executables <my_package>: Muestra los ejecutables de un paquete específico.
ros2 run <my_package> <executable>: Ejecuta un programa de un paquete específico.
ros2 launch <my_package> <executable>: Ejecuta un programa de un paquete específico mediante launcher.
ros2 node list: Lista todos los nodos que se están ejecutando en ese momento.
ros2 topic list: Lista todos los topics que se están utilizando en ese momento.
ros2 node info <node>: Lista más información acerca de un nodo específico.
ros2 topic info <topic>: Lista más información acerca de un topic específico (tipo de mensaje, número de publicadores, número de suscriptores).
ros2 interface list: Lista todos los tipos de mensajes válidos en el sistema.
ros2 interface show <message>: Lista el formato de mensaje y los campos que puede recibir.
ros2 topic echo <topic>: Lista los mensajes publicados de un topic específico, con la opción –no-arr no se muestra el contenido de los arrays de datos.
ros2 run rqt_graph rqt_graph: Muestra el grafo computacional de aquello que esté ejecutándose.
ros2 run rqt_console rqt_console: Herramientas para ver los mensajes publicados en un topic.
ros2 run <my_package> <executable> –ros-args –log-level <level>: Configura el log para establecer otro nivel mínimo de severidad al mostrarse en la salida estándar.
Ejemplo de publicación de un topic: ros2 topic pub -r "10" /int_topic std_msgs/msg/Int32 "{data: 10}".

} 2.2 EJEMPLO BÁSICO DE EJECUCIÓN

```
$ ros2 run demo_nodes_cpp talker          // Terminal 1
$ ros2 run demo_nodes_cpp listener        // Terminal 2
$ ros2 topic echo /chatter                // Terminal 3
$ ros2 run rqt_graph rqt_graph            // Terminal 4
```

} 2.3 SECUENCIA DE EJECUCIÓN DE UN PAQUETE

```
$ cd ~/<my_workspace>/src
$ ros2 pkg create –build-type <type> <my_package> –dependencies rclcpp <message_type>_msgs
$ cd ~/<my_workspace>
$ colcon build –symlink-install // colcon build –packages-select <my_package>
$ source install/setup.sh
$ ros2 run <my_package> <executable>
```

} 2.4 MÉTODOS Y OBJETOS (MAIN.CPP)

include "rclcpp/rclcpp.hpp": Permite el acceso a la mayoría de tipos y funciones de C++.
rclcpp::init(argc,argv): Extrae los argumentos con los que el proceso se ejecutó.
Creación de un nodo: auto node = rclcpp::Node::make_shared("simple_node");
Otras alternativas para la siguiente línea de código:
1. std::shared_ptr<rclcpp::Node> node = std::shared_ptr<rclcpp::Node> (new rclcpp::Node(“simple_node”));
2. std::shared_ptr<rclcpp::Node> node = std::shared_ptr<rclcpp::Node> (“simple_node”);
3. rclcpp::Node::SharedPtr node = std::make_shared<rclcpp::Node> (“simple_node”);
4. auto node = std::make_shared("simple_node");
rclcpp::spin(node): Bloquea la ejecución del programa para que el programa no termine.
rclcpp::shutdown(): Gestiona el cierre de un nodo antes de acabar el programa.

} 2.5 MÉTODOS Y OBJETOS (CMAKELISTS.TXT)

find_package (CMakeLists.txt): Se especifican los paquetes necesarios.
add_executable (CMakeLists.txt): Compila indicando el nombre del ejecutable y sus fuentes.
ament_target_dependencies (CMakeLists.txt): Hace que los encabezados y las bibliotecas de otros paquetes sean accesibles para el objetivo actual.
install (CMakeLists.txt): Instala todos los programas y librerías del paquete (tiene todos los ejecutables).

} 2.6 OTROS MÉTODOS Y OBJETOS

rclcpp::Rate: Hace que el bucle de control se detenga el tiempo suficiente para adaptarse a la tarifa seleccionada.
spin_some() y spin(): Ambos gestionan los mensajes que llegan al nodo llamando a las funciones. Con spin() espera nuevos mensajes y con spin_some() retorna cuando no queden más mensajes.
RCLCPP_INFO: Macro que imprime información.
create_wall_timer: Establece las frecuencias mediante la creación de un contador.

} 2.7 QoS EN ROS2

Default	Reliable	Volatile	Keep Last
Services	Reliable	Volatile	Normal Queue
Sensor	Best Effort	Volatile	Small Queue
DParameters	Reliable	Volatile	Large Queue

Compatibility of QoS durability profiles		Subscriber	
		Volatile	Transient Local
Publisher	Volatile	Volatile	No Connection
	Transient Local	Volatile	Transient Local

Compatibility of QoS reliability profiles		Subscriber	
		Best Effort	Reliable
Publisher	Best Effort	Best Effort	No Connection
	Reliable	Best Effort	Reliable

publisher =
node->create_publisher<std_msgs::msg::String>(“chatter”,rclcpp::QoS(100).transient_local().best_effort());
publisher_ = create_publisher<sensor_msgs::msg::LaserScan>(“scan”,rclcpp::SensorDataQoS().reliable());

} 2.8 LAUNCHERS

Launcher: Fichero de Python que especifica qué programas van a ejecutarse.
generate_launch_description(): Devuelve un objeto LaunchDescription, que contiene las acciones Node (ejecución del programa), IncludeLaunchDescription (incluye otro launcher), DeclareLaunchArgument (declara parámetros del launcher) y SetEnvironmentVariable (configura una variable de entorno).

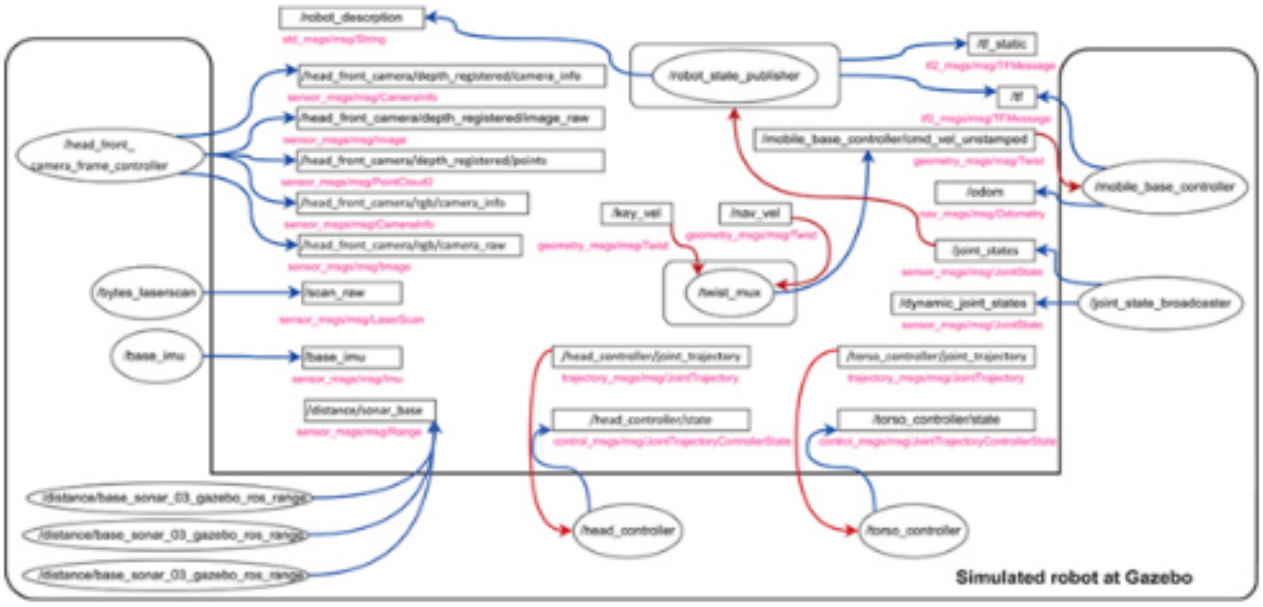
} 2.9 PARÁMETROS Y FICHEROS DE EJECUCIÓN

Parámetros: Son utilizados por un nodo para configurar operaciones (bool, string, array).
declare_parameter(<name>,<value>): Declara los parámetros de un nodo.
get_parameter(<name>,<store_variable>): Obtiene el valor de un parámetro.
ros2 run <my_package> <executable> –ros-args -p <variable>:=<value>: Ejecuta el programa estableciendo un valor específico a cada parámetro.
ros2 run <my_package> <executable> –ros-args –params-file install/<my_package>/share/<my_package>/config/params.yaml: Ejecuta el programa pasándole como argumento el fichero que contiene todos los parámetros.
Ejecutor: Objeto que se añade a los nodos para que puedan ejecutarse conjuntamente.

} 2.10 ENTORNO DE SIMULACIÓN (GAZEBO Y RVIZ2)

ros2 launch br2_tiago sim.launch.py world:=<world>: Lanza el simulador en un mundo específico.
/twist_mux (nodo externo): Crea varios suscriptores a topics que reciben velocidades del robot de diferentes fuentes.
/robot_state_publisher (nodo externo): Lee la descripción de un archivo URDF y se suscribe a cada una de las articulaciones del robot, además de crear y actualizar los frames del robot en el sistema de TFs (relaciona diferentes formas geométricas según los ejes de referencia del robot).
/camera_info (nodo izquierdo): Contiene los valores intrínsecos de la cámara.
/joint_state_broadcaster (nodo derecho): Publica el estado de cada una de las articulaciones del robot.
/mobile_base_controller (nodo derecho): Hace que el robot se mueva según los comandos que va recibiendo.
Remap: Permite cambiar el nombre de uno de los topics durante la ejecución.
/scan_raw y /head_front_camera/rgb/image_raw: Topics con información del láser y de la cámara.
ros2 run teleop_twist_keyboard teleop_twist_keyboard –ros-args -r cmd_vel:=key_vel: Lanzar teleoperador.
ros2 run rviz2 rviz2: Lanzar rviz2 (muestra información geométrica y sensorial).
base_footprint: Eje de coordenadas 3D de un frame.
Odom: Frame representado por la posición inicial del robot.

} 2.11 GRAFO DE COMPUTACIÓN DE UN ROBOT EN SIMULADOR

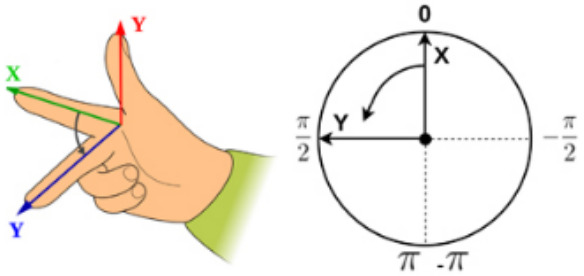


[] TEMA 3: ESQUIVANDO OBSTÁCULOS CON FSMs

CÓDIGOS UTILIZADOS EN ESTE TEMA: br2_fsm_bumpgo_cpp, br2_fsm_bumpgo_py, br2_tiago.

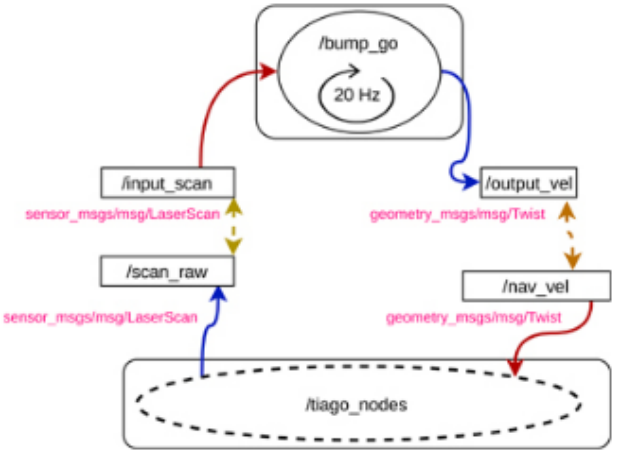
} 3.1 CONCEPTOS BÁSICOS

Comportamiento Bump-And-Go: Utiliza los sensores del robot para detectar obstáculos cerca del mismo. Cuando éste detecta un obstáculo, retrocede y gira durante un tiempo fijo para avanzar de nuevo.
Máquina de estados finita: Modelo computacional matemático que se usa para determinar el comportamiento de un robot (el robot se guía por el estado de salida hasta que se cumpla lo propuesto en una transición).



} 3.2 GRAFO DE COMPUTACIÓN DE UNA MÁQUINA DE ESTADOS FINITA

/nav_vel: Topic del tipo geometry_msgs/msg/Twist donde se envían las velocidades de traslación y rotación del robot.



} 3.3 IMPLEMENTACIÓN DE LA MÁQUINA DE ESTADOS EN C++ Y PYTHON

control_cycle(): Método de ciclo de control que funciona a una frecuencia de 20 Hz.
Variables temporales: int state_ almacena el estado actual, rclcpp::Time state_ts_ indica el tiempo transcurrido en el estado actual permitiendo cambiar a otros estados usando timeouts, y rclcpp::TimerBase::SharedPtr timer_ se encarga de llamar al ciclo de control cada 50 ms.
Otras alternativas para la línea de código void scan_callback(sensor_msgs::msg::LaserScan::UniquePtr msg):
1. void scan_callback(const sensor_msgs::msg::LaserScan & msg);
2. void scan_callback(sensor_msgs::msg::UniquePtr msg);
3. void scan_callback(sensor_msgs::msg::SharedConstPtr msg);
4. void scan_callback(const sensor_msgs::msg::SharedConstPtr & msg);
5. void scan_callback(sensor_msgs::msg::LaserScan::SharedPtr msg);

get_clock(): Método que pregunta por el tiempo, con now() se obtiene el tiempo actual.
Time.from_msg(): Crea un objeto Time a partir del Timestamp de un mensaje.
ros2 run br2_fsm_bumpgo_cpp bumper –ros-args -r output_vel:=/nav_vel -r input_scan:=/scan_raw -p use_sim_time:=true: Ejecución del programa de la máquina de estados, con use_sim_time:=true toma el tiempo del topic /clock, correspondiente al tiempo del simulador en vez del de nuestra máquina.

[] TEMA 4: EL SUBSISTEMA TF

CÓDIGOS UTILIZADOS EN ESTE TEMA: br2_tf2_detector, br2_tiago.

} 4.1 DEFINICIÓN DE TFs Y MATRIZ DE TRANSFORMACIÓN

Subsistema de TFs: Permite definir diferentes ejes de referencia (frames) y la relación geométrica entre ellos incluido en estado cambiante (las coordenadas de un frame se pueden recalcular para otro frame).
Ejemplo de uso para las coordenadas: Sensor de distancia (x,y,z), robot (x,y,z,roll,pitch,yaw), navegación (x,y,yaw).

$$P_B = RT_{A\to B} * P_A$$

$$\begin{pmatrix} x_B \\ y_B \\ z_B \\ 1 \end{pmatrix} = \begin{pmatrix} R^{xx}_{A\to B} & R^{xy}_{A\to B} & R^{xz}_{A\to B} & T^{x}_{A\to B} \\ R^{yx}_{A\to B} & R^{yy}_{A\to B} & R^{yz}_{A\to B} & T^{y}_{A\to B} \\ R^{zx}_{A\to B} & R^{zy}_{A\to B} & R^{zz}_{A\to B} & T^{z}_{A\to B} \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_A \\ y_A \\ z_A \\ 1 \end{pmatrix}$$

} 4.2 TOPICS DE TFs

/tf: Topic del tipo tf2_msgs/msg/TFMessage para transformaciones que varían dinámicamente como las articulaciones de un robot (válidos para un tiempo corto).
/tf_static: Topic del tipo tf2_msgs/msg/TFMessage para transformaciones que no varían con el tiempo, además posee un QoS del tipo transient_local donde todos los nodos suscritos reciben todas las transformadas publicadas hasta el momento aunque éstas no cambian con el tiempo (geometría del robot).

} 4.3 EJES DE COORDENADAS PARA TFs

/base_footprint: Es la raíz de las TFs de un robot y corresponde al centro del mismo en el suelo y es útil para transformar la información de los sensores del robot a este eje para relacionarlos entre sí.
/base_link: Es el elemento secundario de /base_footprint, por lo que corresponde nuevamente al centro del robot pero esta vez por encima del nivel del suelo.
/odom: Es el marco principal de /base_footprint y la transformación que los relaciona indica el desplazamiento del robot desde que comenzó a desplazarse.

} 4.4 REPRESENTACIÓN GRÁFICA DE TFs

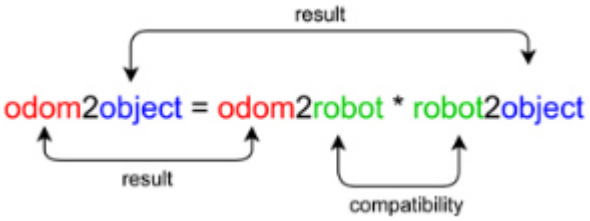
ros2 run rqt_tf_tree rqt_tf_tree: Muestra el árbol de transformadas del robot simulado. También se utiliza para saber si hay una TF entre dos frames, la rotación de un frame A a un frame B, o la transformación de una coordenada de un frame A a un frame B en un momento específico.

} 4.5 PUBLICACIÓN DE UNA TRANSFORMADA

```
geometry_msgs::msg::TransformStamped detection_tf;
detection_tf.header.frame_id = “base_footprint”;
detection_tf.header.stamp = now();
detection_tf.child_frame_id = “detected_obstacle”;
detection_tf.transform.trnslation.x= 1.0;
tf_broadcaster_->sendTransform(detection_tf);
tf2_ros::Buffer tfBuffer;
tf2_ros::TransformListener tfListener(tfBuffer);
geometry_msgs::msg::TransformStamped odom2obstacle;
odom2obstacle = tfBuffer_.lookupTransform(“odom”,”detected_obstacle”,tf2::TimePointZero);
```

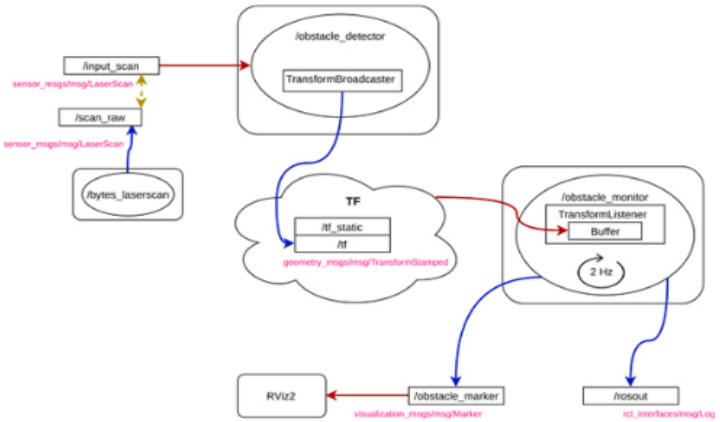
tf2::TimePointZero: Indica el último instante de tiempo en el que se obtuvo la transformada.

} 4.6 REGLA DE NOMBRADO DE TFs



} 4.7 GRAFO DE COMPUTACIÓN Y VISUAL MARKERS

Visual Markers: Permiten publicar elementos visuales 3D que pueden visualizarse en RViz2 desde un nodo sin necesidad de usar los mensajes generados por las macros RCLCPP_* (flechas, líneas, cilindros, etc).



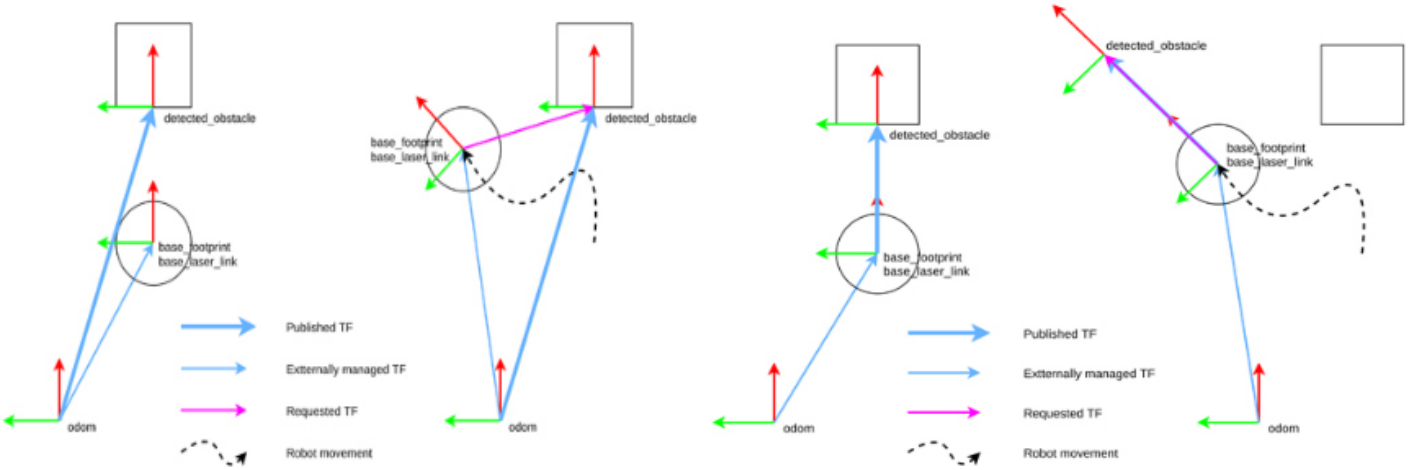
target_link_libraries (CMakeLists.txt): Instala las librerías en el mismo sitio que los ejecutables.

} 4.8 OBJETOS Y MÉTODOS DEL PAQUETE BR2_TF2_DETECTOR

std::shared_ptr<tf2_ros::StaticTransformBroadcaster> tf_broadcaster_: Publica una transformada en /tf_static, si se quiere /tf se debe usar TransformBroadCaster. Además, maneja la publicación de TFs estáticas con mensajes del tipo geometry_msgs/msg/TransformStamped.
detection_tf.header = msg->header: Frame fuente o frame padre de la transformada.
detection_tf.child_frame_id = "detected_obstacle": ID del nuevo frame que va a crearse (obstáculo detectado).
detection_tf.transform.translation.x: Traslación y rotación del eje X.
tf_broadcaster_->sendTransform(detection_tf): Envía la transformada al subsistema de TFs.
tf2_ros::TransformListener tf_listener_: Accede al sistema de TFs y actualiza el tf_buffer_ pasado como parámetro para consultar el valor de una TF existente.
rclcpp::Publisher<visualization_msgs::msg::Marker>::SharedPtr marker_pub_: Publicador de Visual Markers.
robot2obstacle = tf_buffer_.lookupTransform("base_footprint", "detected_obstacle", tf2::TimePointZero): Calcula la transformación geométrica de un frame a otro.
geometry_msgs::msg::TransformStamped: Tipo de mensajes utilizados para publicar TFs y es el resultado devuelto por lookupTransform().
tf2::Transform: Tipo de datos de la biblioteca tf2 que permite realizar operaciones.
tf2::Stamped<tf2::Transform>: Similar al anterior con un encabezado que indica marca de tiempo.
tf2::fromMsg o tf2::toMsg: Funciones de transformación que transforman un mensaje a tf2 y viceversa.

} 4.9 EJEMPLOS DE EJECUCIÓN Y DIAGRAMAS DE TFs

```
$ ros2 launch br2_tiago sim.launch.py world:=empty                                          // Terminal 1
$ ros2 launch br2_tf2_detector detector_basic.launch.py                                     // Terminal 2
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard –ros-args -r cmd_vel:=/key_vel       // Terminal 3
$ ros2 run rviz2 rviz2 –ros-args -p use_sim_time:=true                                      // Terminal 4
```




```
$ ros2 launch br2_tiago sim.launch.py world:=empty                                          // Terminal 1
$ ros2 launch br2_tf2_detector detector_improved.launch.py                                  // Terminal 2
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard –ros-args -r cmd_vel:=/key_vel       // Terminal 3
$ ros2 run rviz2 rviz2 –ros-args -p use_sim_time:=true                                      // Terminal 4
```

[] TEMA 5: COMPORTAMIENTOS REACTIVOS

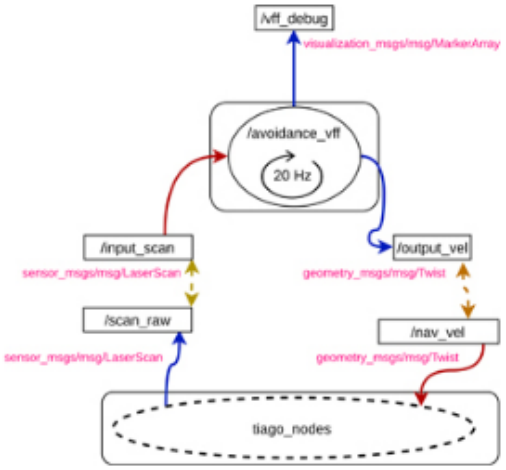
CÓDIGOS UTILIZADOS EN ESTE TEMA: br2_vff_avoidance, br2_tracking_msgs, br2_tracking, br2_tiago.

} 5.1 VECTORES DEL ALGORITMO VFF

Vector de atracción: Apunta siempre en línea recta para que el robot se mueve en línea recta en ausencia de obstáculos.
Vector de repulsión: Se calcula a partir de las lecturas del láser, donde el obstáculo produce un vector de repulsión inversamente proporcional a su distancia.
Vector resultante: Es la suma de los dos anteriores y calcula la velocidad de control, donde la velocidad lineal depende del módulo vectorial resultante y el ángulo de giro depende del ángulo vectorial resultante.

} 5.2 GRAFO COMPUTACIONAL PARA ESQUIVAR OBSTÁCULOS

Topic /vff_debug: Publica Visual Markers que visualizan los diferentes vectores del algoritmo VFF.



} 5.3 OBJETOS Y MÉTODOS DEL PAQUETE BR2_VFF_AVOIDANCE

get_vff(): Calcula los tres vectores del algoritmo VFF mientras lee los datos del láser y se almacenan en la variable VFFVectors.
Vector de atracción: Siempre (1,0) ya que siempre va hacia delante.
Vector de repulsión: Inversamente proporcional a la distancia al obstáculo (lecturas mínimas, ángulo + pi).
Vector resultante: Suma de los dos vectores anteriores.
std::clamp(): Controla los rangos de velocidad a partir de los módulos y ángulos obtenidos.
visualization_msgs::msg::Marker: Crea un Visual Marker con un color especificado como parámetro de entrada (tipo de mensaje visualization_msgs/msg/Marker).
visualization_msgs::msg::MarkerArray marker: Vector que contiene un Visual Marker para cada vector VFF (tipo de mensaje visualization_msgs/msg/MarkerArray)

} 5.4 VENTAJAS DE TESTEAR EN DESARROLLO

Ventajas: Una vez probadas unas partes del software, asegurarse de que éstas no cambian otras partes previamente desarrolladas, las pruebas son incrementales produciendo un desarrollo más rápido, la tarea de revisión se simplifica si recibe aportaciones de otros desarrolladores (el desarrollador solo se tiene que ocupar del funcionamiento).
Tag <test_depend>: Contiene las dependencias necesarias para testear el paquete.
colcon build –symlink-install –packages-select <my_package> –cmake-args -DBUILD_TESTING=OFF: Paquetes no tomados en cuenta en las dependencias una vez se haya compilado el workspace o el mismo paquete.
Macro ASSERT_*: Comprueba los valores esperados en la función de entrada (ASSERT_EQ, ASSERT_NEAR, ASSERT_LT, ASSERT_GT).

} 5.5 TESTS DE INTEGRACIÓN

1. Crea un nodo AvoidanceNodeTest o AvoidanceNode para probarlo.
2. Crea un nodo genérico llamado test_node para crear un editor de escaneo láser y un suscriptor de velocidad.
3. Al crear el suscriptor de velocidad, se ha especificado una función lambda como devolución de llamada. Esta función lambda accede a la variable last_vel para actualizarlo con el último mensaje recibido por el topic output_vel.
4. Crea un ejecutor y agrega ambos nodos para ejecutarlos.
5. Durante una segunda publicación a 30 Hz en input_scan hace una lectura sintética del sensor.
6. Verifica que las velocidades publicadas sean correctas.

$ cd <my_workspace>/
$ build/<my_package>/tests/<my_tests>
log/latest_test/<my_package>/stdout_stderr.log: Fichero que contiene dónde han fallado los tests.

} 5.6 ALGUNOS CONCEPTOS NUEVOS

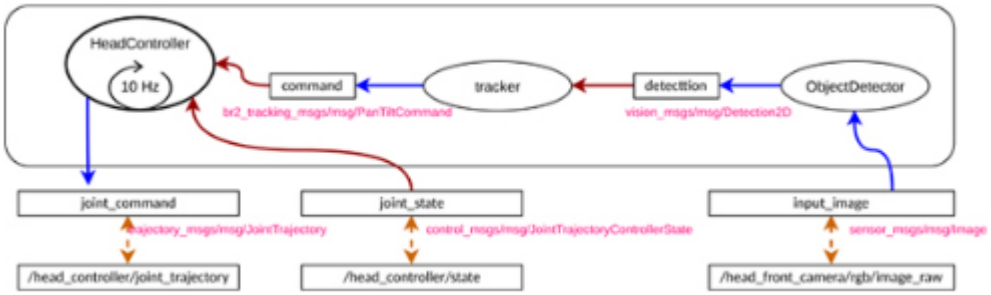
Análisis de imagen: Proporcionan información de percepción más compleja de la que normalmente se puede extraer (Visión Artificial y la biblioteca OpenCV).
Control a nivel conjunto: En vez de solo enviar velocidades al robot, se envían posiciones directamente a las articulaciones del cuello del robot.
Nodos de ciclo de vida (lifecycle nodes): Sirven para controlar el ciclo de vida de un nodo, así como su puesta en marcha, activación y desactivación.

} 5.7 MODELOS DE PERCEPCIÓN Y ACTUACIÓN

Drivers de la cámara: Publican solo una vez en QoS transient_local información sobre los parámetros intrínsecos, de distorsión, proyección y matriz (sensor_msgs/msg/CameraInfo o sensor_msgs/msg/Image).
ros2 run image_transport list_transports: Muestra todos los plugins de transporte.
HSV vs RGB: HSV toma tono, saturación y valor, permitiendo establecer gamas de colores más robustas para cambios de iluminación (componente V).
Transformación de un cv::mat a HSV:

```
cv::Mat img_hsv;
cv::cvtColor(cv_ptr->image, img_hsv, cv::COLOR_BGR2HSV);
cv::Matlb filtered;
cv::inRange(img_hsv, cv::scalar(15,50,20),cv::Scalar(20,200,200),filtered);
```

Modelo de acción: Control de la posición de la cabeza del robot mediantes dos articulaciones, head_1_joint para el control horizontal y head_2_joint para control vertical, controladas por el framework ros2_control. Con joint_trajectory_controller se pueden controlar las dos articulaciones del robot.



} 5.8 NODOS DE CICLO DE VIDA (LIFECYCLE NODES)

Nodos de ciclo de vida (LifeCycle Nodes): Tiempo de vida definido por estados y transiciones.
Características: Estado inicial desconfigurado por lo que debe configurarse para entrar al estado inactivo; funciona cuando está en estado activo; puede pasar de activo a inactivo y viceversa mediante transiciones de activación y desactivación; realiza tareas y comprobaciones en cada transición; pasa a estado finalizado en caso de error o cuando el nodo haya completado su tarea.
ros2 lifecycle nodes: Muestra qué nodos de control de vida (LifeCycle) se están ejecutando en ese momento.
ros2 lifecycle get <node>: Comprueba en qué estado se encuentra un nodo específico.
ros2 lifecycle list <node>: Muestra las transiciones disponibles en el estado actual.
ros2 lifecycle set <node> activate: Activa un nodo específico.
ros2 lifecycle set <node> deactivate: Desactiva un nodo específico.

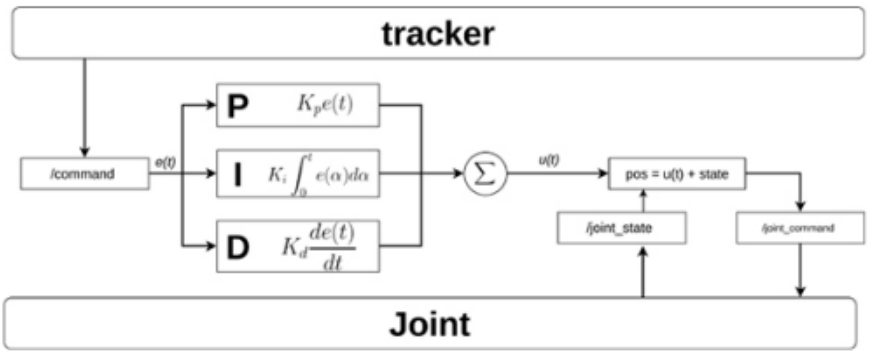
Modelo de ejecución: Deben ser predecibles, pueden coordinar su puesta en marcha al haber múltiples nodos, pueden configurar algunos pedidos al iniciarse, pueden iniciarse con más opciones más allá de las de un constructor.

} 5.9 OBJETOS Y MÉTODOS DEL PAQUETE BR2_TRACKING_MSGS

Formato de mensaje: <type> <message>
rosidl_generate_interfaces (CMakeLists.txt): Especifica dónde están definidas las interfaces.

} 5.10 OBJETOS Y MÉTODOS DEL PAQUETE BR2_TRACKING

cv::boundingRect: Calcula un bounding box a partir de la máscara resultante del filtro de color.
cv::moments: Calcula el centro de masa de estos píxeles.
control_msgs::msg::JointTrajectoryControllerState: Mensaje que informa del nombre de las articulaciones controladas, así como sus trayectorias deseadas, actuales y de error.

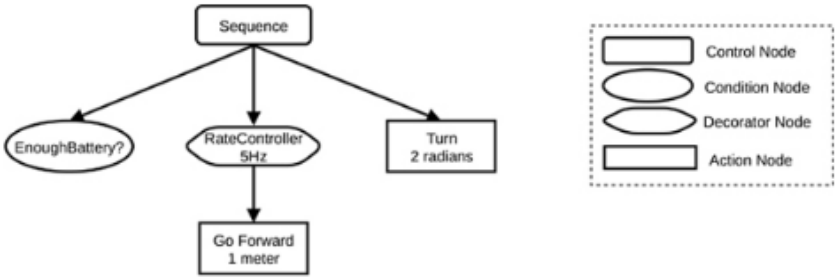


[] TEMA 6: COMPORTAMIENTOS ROBÓTICOS CON BEHAVIOR TREES

CÓDIGOS UTILIZADOS EN ESTE TEMA: br2_bt_bumpgo, br2_bt_patrolling, br2_navigation, br2_tiago.

} 6.1 BEHAVIOR TREES (DEFINICIÓN Y CONCEPTOS BÁSICOS)

Behavior Tree (BT): Modelo matemático para codificar el control de un sistema. Es una forma de estructurar el cambio entre diferentes tareas en un agente autónomo. Es una estructura de datos jerárquica definida recursivamente desde un nodo raíz con varios nodos secundarios.
Tick SUCCESS: El nodo ha completado su misión con éxito.
Tick FAILURE: El nodo ha fallado en su misión.
Tick RUNNING: El nodo aún no ha completado su misión.
Nodo de control: Tienen hijos 1-N (contagiar la garrapata a sus hijos).
Nodo decorador: Nodos de control con un solo hijo.
Nodo de acción: Son las hojas del árbol (generan el control requerido por el solicitante).
Nodo de condición: Son nodos que no devuelven RUNNING (solo devuelve SUCCESS o FAILURE).



Desarrollo de nodos: Los nodos están diseñados, desarrollados y compilados en esta fase, donde pasan a formar parte de la biblioteca de nodos disponibles en la misma categoría que los nodos centrales.
Despliegue: El Behavior Tree se compone utilizando los nodos disponibles y teniendo en cuenta el comportamiento de cada uno de los nodos (puede haber diferentes comportamientos para nodos iguales).
Blackboard: Almacenamiento de parejas clave-valor a la que todos los nodos del árbol pueden acceder.

} 6.2 TIPOS DE NODOS DE CONTROL DE UN BEHAVIOR TREE

	Value returned by child		
Control Node Type	**FAILURE**	**SUCCESS**	**RUNNING**
`Sequence`	Return FAILURE and restart sequence	Tick next child. Return SUCCESS if no more child	Return RUNNING and tick again
`ReactiveSequence`	Return FAILURE and restart sequence	Tick next child. Return SUCCESS if no more child	Return RUNNING and restart sequence
`SequenceStar`	Return FAILURE and tick again	Tick next child. Return SUCCESS if no more child	Return RUNNING and tick again
`Fallback`	Tick next child. Return FAILURE if no more child	Return SUCCESS	Return RUNNING and tick again
`ReactiveFallback`	Tick next child. Return FAILURE if no more child	Return SUCCESS	Return RUNNING and restart sequence
`InverterNode`	Return SUCCESS	Return FAILURE	Return RUNNING
`ForceSuccessNode`	Return SUCCESS	Return SUCCESS	Return RUNNING
`ForceFailureNode`	Return FAILURE	Return FAILURE	Return RUNNING
`RepeatNode` (N)	Return FAILURE	Return RUNNING N timesbefore returning SUCCESS	Return RUNNING
`RetryNode` (N)	Return RUNNING N times before returning FAILURE	Return SUCCESS	Return RUNNING

Nodos de secuencia: Sequence, ReactiveSequence y SequenceStar.
Nodos de retroalimentación: Fallback y ReactiveFallback.
Nodos decoradores: RepeatNode y RetryNode.

} 6.3 OBJETOS Y MÉTODOS DEL PAQUETE BR2_BT_BUMPGO

ros2 run groot Groot: Herramienta para crear y desarrollar Behavior Trees.

Nuevos directorios en el paquete: Directorio tests (testeo del BT con gtest), directorio cmake (fichero FindZMQ.cmake para depurar el BT en tiempo de ejecución), directorio behavior_tree_xml (fichero xml con la estructura del BT).

BehaviorTree: Es la especificación de la estructura del árbol (tipos de nodos y nodos secundarios), en el CMakeLists.txt cada nodo debe compilarse como librerías separadas.

TreeNodesModel: Define los nodos personalizados que se han creado, indicando puertos de entrada y salida.

BT::ActionNodeBase: Implementa los métodos del constructor, donde recibe el nombre del archivo xml y una BT::NodeConfiguration que contiene un puntero a la Blackboard compartida por todos los nudos del árbol.

halt(): Se le llama cuando el árbol termina su ejecución y se utiliza para llevar a cabo cualquier limpieza que requiera el nodo.

tick(): Implementa la operación tick descrita anteriormente.

static(): Devuelve los puertos del nodo.

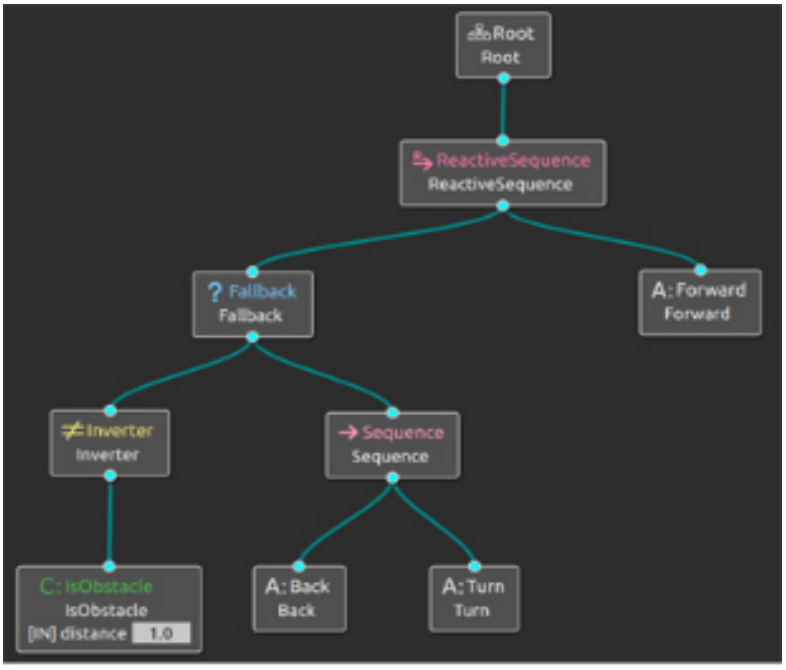
Objeto loader: Busca la librería del sistema que lea el BT como un plugin.

Macro BT_REGISTER_NODES: Permite a un nodo del BT estar conectado con su implementación sin necesidad de librería.

get_package_share_directory(): Obtiene la ruta completa del paquete.

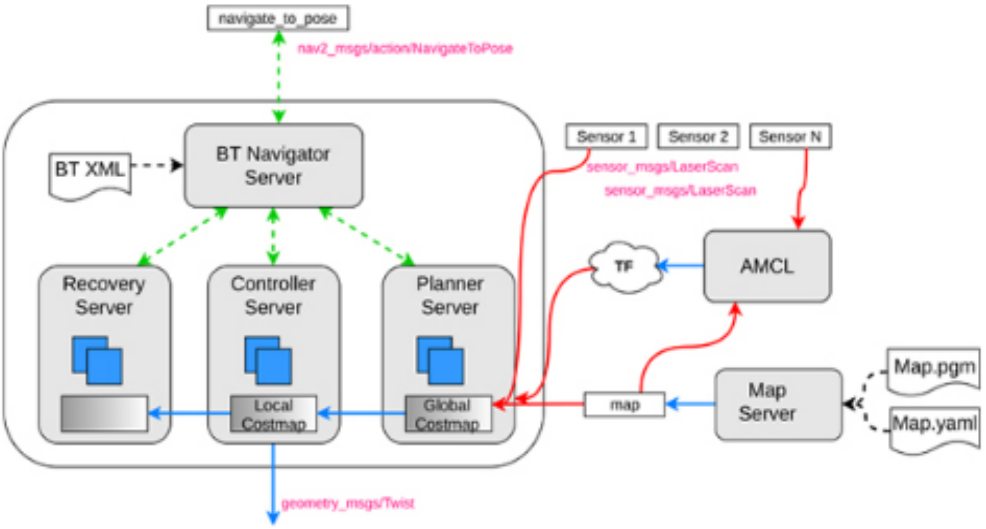
Objeto PublisherZMQ: Publica toda la información necesaria para depurar el BT en tiempo de ejecución.

build/<my_package>/tests/<node> –ros-args -r input_scan:=/scan_raw -r output_vel:=/key_vel -p use_sim_time:=true: Testeo de un nodo de un BT.



} 6.4 DESCRIPCIÓN DE NAV2

Nav2: Sistema de navegación de ROS2 diseñado para ser modular, configurable y escalable.
Map Server: Lee un mapa de dos archivos y lo publica como nav_msgs/msg/OccupancyGrid cuyos nodos se manejan internamente como un costmap 2D.
AMCL: Utiliza información sensorial (lectura de distancia del láser y el mapa) para calcular la posición del robot, obteniendo como salida una transformación geométrica que indica la posición del robot (map -> odom).
Planner Server: Calcula una ruta desde el origen hasta el destino tomando como entrada el destino, la posición actual del robot y el mapa de entorno, además de crear un mapa de costos a partir del mapa original cuyas paredes se engordan con el radio del robot y un cierto márgen de seguridad.
Controller Server: Recibe la ruta calculada por el Planner Server y publica las velocidades enviadas a las base del robot usando un mapa donde los obstáculos más cercanos son codificados y utilizados por algoritmos como plugins para calcular velocidades.
Recuperation Server: Posee varias herramientas de recuperación como giros contínuos, limpieza de mapas de costo y ralentización del movimiento.
BT Navigator Server: Objeto que da funcionamiento a todos los demás, recibiendo solicitudes de navegación en forma de acciones (navigate_to_pose en el topic nav2_msgs/action/NavigateToPose).

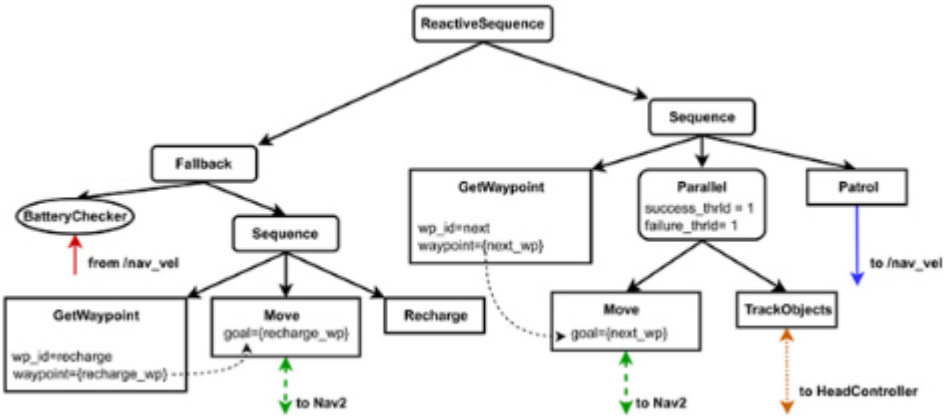


ros2 launch br2_navigation tiago_navigation.launch.py: Lanza la navegación.
ros2 launch nav2_bringup tb3_simulation.launch.py: Lanza la navegación, donde el paquete nav2_bringup se encuentra en /opt/ros/humble/share/nav2_bringup que incluye launchers, mapas y parámetros para la simulación con el TurtleBot3 (por defecto).

} 6.5 CREACIÓN Y GUARDADO DE UN MAPA USANDO NAV2

$ ros2 launch br2_tiago sim.launch.py
$ rviz2 –ros-args -p use_sim_time:=true
$ ros2 launch slam_toolbox online_async_launch.py
params_file:=<path_completo_<mapper_params_online_async.yaml>> use_sim_time:=true
$ ros2 launch nav2_map_server map_saver_server.launch.py
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard –ros-args -r cmd_vel:=key_vel -p use_sim_time:=true
$ ros2 run nav2_map_server map_saver_cli –ros-args -p use_sim_time:=true

} 6.6 OBJETOS Y MÉTODOS DEL PAQUETE BR2_BT_PATROLLING



Move::on_success(): Indica que la navegación ha finalizado.
on_tick(): Toma el objetivo de la entrada y lo asigna a goal_ (variable que se envía a Nav2).
Servicios de los nodos de control de vida: <node>/get_state (devuelve el estado del LifeCycleNode), <node>/set_state (establece el estado del LifeCycleNode).