UNIVERSIDAD POLITÉCNICA DE MADRID

Escuela Técnica Superior de Ingenieros Industriales

Máster Universitario en Automática y Robótica

Trabajo Fin de Máster

Arquitectura de Movimiento Resiliente para un Robot Minero

Autor: **Rodrigo Silverio Orbiso**

Tutor 1: **Ricardo Sanz Bravo**

Tutor 2: **Esther Aguado González**

2022

"La ciencia consiste en saber; la ingeniería, en hacer."

- Henry Petroski

AGRADECIMIENTOS

Agradezco a mis tutores Ricardo y Esther por ayudarme y guiarme en este trabajo.

Gracias a mis padres por haberme apoyado todo este tiempo, y el amor que me han demostrado.
A mis amigos por haber estado ahí cando fuera necesario, y sacarme unas risas.
Y a Lía por confiar en mí y apoyarme como nadie lo había hecho en tiempo.

RESUMEN EJECUTIVO

El documento describe y fundamenta el desarrollo de una arquitectura adaptativa basada en ontologías y en ROS2, para una reimplementación de la navegación de un robot minero. Se explica el diseño de una ontología basada en sensores junto con un metacontrolador para la plataforma en cuestión, que le dotan de sus capacidades de adaptación. Para trabajar con la ontología se ha utilizado Owlready2, para poder manipular la ontología en Python; y Protégé, para realizar una edición más minuciosa y menos abstracta. Se ha utilizado como razonar ontológico Pellet y se aplican las normas SWRL para poder obtener las inferencias de la ontología.

Toda la arquitectura está basada en ROS2, y la navegación se implementa utilizando el paquete Navigation2, siendo este la pieza central de toda la arquitectura. El comportamiento de la navegación se verá afectado por las inferencias y resultados de la ontología.

Se utilizará Gazebo Ignition para simular todo el entorno y la propia plataforma robótica.

ÍNDICE

AGRADECIMIENTOS III

RESUMEN EJECUTIVO V

ÍNDICE DE TABLAS XI

ÍNDICE DE FIGURAS XII

1. INTRODUCCIÓN 1

1.1. Descripción del proyecto 1

1.2. Objetivos 3

2. ANTECEDENTES E INVESTIGACIÓN PREVIA 4

2.1. Proyecto MROS de RobMoSys 5

2.2. Meta-controlador 5

2.3. Ontologías para robótica móvil 6

2.3.1. Definiciones básicas de ontologías 7

2.3.2. Resource Description Framework (RDF) 8

2.3.3. Web Ontology Language (OWL) 9

2.3.4. Semantic Web Rule Language (SWRL) 9

2.4. ROS2: Robot Operating System 10

2.4.1. Funcionamiento general de ROS2 10

2.4.1.1. Sistema de archivos 11

2.4.1.2. Nodos (Nodes) 12

2.4.1.3. Temas (Topics) 13

2.4.1.4. Mensajes (Messages) 13

2.4.1.5. Servicios (Services) 14

2.4.2. Principales diferencias entre ROS1 y ROS2 14

2.4.2.1. Cambios en la API de ROS 14

2.4.2.2. Cambios en las versiones de Python y C++ 15
2.4.2.3. Nodos: Cambios en las convenciones 15
2.4.2.4. Nodos: Múltiples nodos en el mismo ejecutable 15
2.4.2.5. Nodos: Nodos con ciclo de vida 15
2.4.2.6. Archivos de lanzamiento 16
2.4.2.7. Comunicación 16
2.4.2.8. Servicios y acciones 16
2.4.2.9. Sistema de construcción 17
2.4.3. Robot Stack 17
2.4.4. Navigation Stack 18
2.5. Gazebo: simulador de robótica 21

3. ARQUITECTURA PROPUESTA 22

3.1. Arquitectura de ROS 22
3.1.1. Utilizando Gazebo Ignition y ROS 22
3.1.2. Core Stack 23
3.1.3. Navigation Stack 24
3.1.4. Metacontrolador Rosont 26
3.2. Ontología 28
3.2.1. Las ontologías de SSN y SOSA 28
3.2.2. La ontología de Rosont 30
3.2.3. Reglas SWRL de Rosont 32
3.3. Simulación con Gazebo Ignition 34

4. RESULTADOS EXPERIMENTALES 35

4.1. Metodología de experimentación 35
4.2. Experimentación con la plataforma 36
4.2.1. Testeo de la simulación en Gazebo Ignition 36
4.2.2. Testeo de la navegación en ROS2 con Nav2 y Rviz 37
4.3. Experimentación con Rosont 40

4.3.1. Framework de testeo para Rosont 40

4.3.2. Testeo de las reglas SWRL con Protégé 41

5. CONCLUSIONES **42**

6. LÍNEAS FUTURAS DE TRABAJO **43**

BIBLIOGRAFÍA **44**

A. Código del proyecto **47**

B. Planificación, presupuesto y análisis social-profesional **48**

B.1. Planificación temporal 48

B.2. Presupuestos del proyecto 50

B.2.1. Costes directos 50

B.2.2. Costes indirectos 51

B.2.3. Presupuesto final 52

B.3. Análisis social-profesional 53

ÍNDICE DE TABLAS

3.1. Tabla con los individuos creados para Rosont 31

3.2. Configuración de los Procedimientos según las Observaciones 32

B.1. Tabla de las tareas y su planificación individual 48

B.2. Tabla con los trabajadores y sus costes asociados 50

B.3. Tabla con los equipos y su coste individual 51

B.4. Tabla con la estimación de los costes indirectos 51

B.5. Tabla con los costes totales de desarrollo 52

ÍNDICE DE FIGURAS

1.1. Arquitectura base de RobMoSys MROS. 2

2.1. Jerarquía del sistema de archivos 11

2.2. Estructura del paquete nav_sim, igt_sim y el metapaquete rosont. 12

2.3. Grafico de conexiones entre nodos mediante diferentes topics. 13

2.4. Arquitectura standard del stack de navegación en ROS. 19

2.5. Antigua arquitectura de move_base con los cambios en ROS2. 20

2.6. Simulación corriendo en Gazebo Classic. 21

3.1. Ventana de Rviz con el stack de navegación funcionando funcionando 24

3.2. Diagrama del funcionamiento de amcl en comparación con robot_localization. . . 26

3.3. Arquitectura base de la ontología SSN. 29

3.4. Arquitectura base de la ontología SOSA. 29

3.5. Ventana de Gazebo Ignition con la simulación funcionando 34

4.1. Simulación general de la cueva en Gazebo Ignition. 36

4.2. Plano detalle de las cámaras de color y de profundidad. 37

4.3. Ventana de Rviz durante la navegación dentro de la cueva. 38

4.4. Representación de la posición XY en PlotJuggler en tiempo real. 38

4.5. Árbol de topics y nodos una vez lanzado el stack de navegación. 39

4.6. Árbol de transformadas basado en los mensajes de TF. 39

4.7. Terminal de debug durante la ejecución de Rosont. 40

4.8. Pestaña de SWRLTab para el testeo en Protégé. 41

B.1. Diagrama de Gantt con la planificación del proyecto. 49

1. INTRODUCCIÓN

En la campo de la robótica autónoma, los robots necesitan realizar multitud de acciones diferentes y adaptarse a entornos constantemente cambiantes. Para lograr que esto ocurra se han integrado multitud de tecnologías y sistemas, para dotar a los robots de versatilidad y adaptabilidad, siempre en función de sus capacidades. En los últimos años se han visto grandes avances en visión por computador, inteligencia artificial y la capacidad de computo de los propios robots, utilizándose para resolver este tipo de problemas.

El proyecto que se va a presentar a continuación parte de un proyecto previo de navegación de robots en una mina. En dicho proyecto se contaba un robot móvil de accionamiento diferencial idéntico, tanto en forma como en especificaciones, al que se utiliza en este proyecto. Además se contaba con el mismo modelado de la de cueva y otras muchas similitudes que se caracterizarán más adelante.

Por tanto, en este proyecto se ha realizado la reimplementación del algoritmo de navegación para una nueva reimplementación de ROS (Robotic Operative System), llamada ROS2. Y de la reimplementación del motor de simulación Gazebo, con su nueva versión Gazebo Ignition.

Además se ha creado un metacontrolador, que implementa un razonador junto con una ontología propia del proyecto para dotar a la plataforma móvil de la habilidad de adaptarse al entorno, siempre y cuando se cumplan ciertas condiciones y realizando inferencias para establecer que debe hacer.

1.1. Descripción del proyecto

Como se ha mencionado anteriormente, este proyecto se va a basar en parte en un proyecto preexistente que realiza una implementación similar a la que busca los objetivos del proyecto. Este proyecto implementa la los modelos de RobMoSys y ROS para proveer a la plataforma de una autoadaptación que responda a un razonamiento ontológico realizado por el propio controlador sobre los modelos de la arquitectura.

Este proyecto además incluye una arquitectura muy completa y más compleja de lo que es necesario para este proyecto. Incluye modos de comportamiento según condiciones del entorno, e incluyen arboles de comportamientos adaptados para sus tareas.

También incluye un sistema de diálogo que se puede utilizar tanto el audio del robot como el de la tablet de control para comunicarse con un humano. La selección del modo de comunicación se realiza mediante el parámetro hri_mode. Si el parámetro hri_mode es .ªudio"se utiliza el audio. Si el parámetro hri_mode es "tablet", se utiliza la tableta del robot. Se utiliza Dialogflow para procesar estas entradas.

Además de crear su propios paquetes personalizados, han modificado paquetes de ROS básicos dentro de la navegación para añadirles funcionalidades para su arquitectura. Por ejemplo, algunos de estos paquetes son el propio stack de navegación (nav2) o utilidades propias de ROS como el paquete pointcloud_to_laserscan.

Su implementación incluye un paquete piloto con los comportamientos de las plataformas que se han incluido para el metacontrolador. También incluye un nodo que gestiona los modos de operación, llamado node-manager, y que es el punto de unión entre el metacontrolador, la navegación de ROS y el controlador HRI.

Su arquitectura se puede ver en la imagen a continuación:

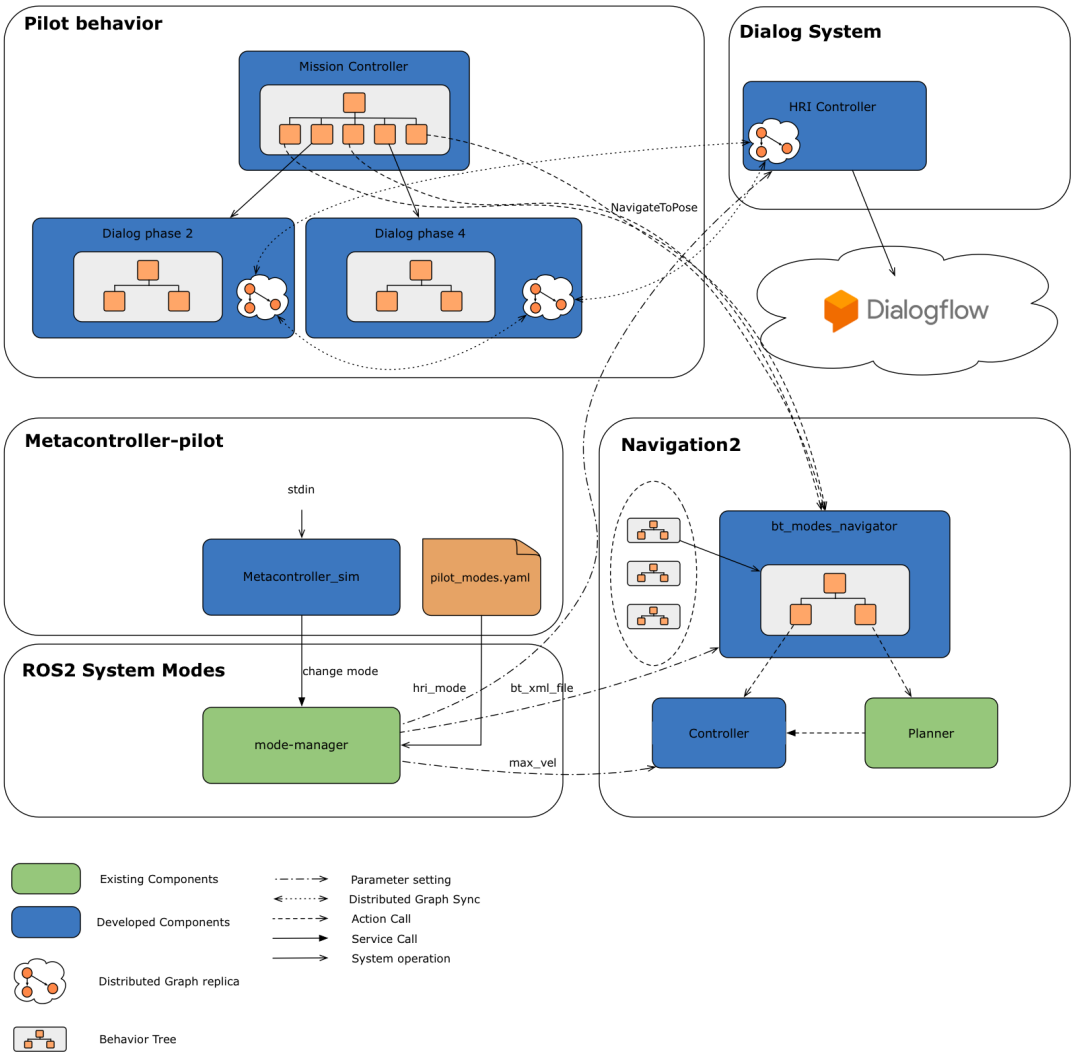

Figura 1.1: Arquitectura base de RobMoSys MROS.

Teniendo todo esto en cuenta, se van a estudiar otras posibilidades y proyectos relacionados, que cumplan con los objetivos del proyecto, y que permitan quizás innovar en este sentido. Se revisará la bilbiografia referente a estos temás, para poder dar una solución que no solo sea correcta, si no que también se aportable a otros porblemas que tengas unos objetivos similares.

1.2. Objetivos

Los objetivos del proyecto se pueden dividir en dos partes: la reimplementación del proyecto en las nuevas versiones de ROS y de Gazebo, y la creación del metacontrolador y su ontología.

Teniendo esto en cuenta se han definido los siguientes objetivos:

1. Reimplementar una arquitectura de navegación para un robot móvil subterráneo utilizando la plataforma ROS2 y Gazebo Ignition.
2. Utilizar las capacidades de reflexión de DDS/ROS2 para proporcionar un mecanismo de introspección del controlador y de autoconciencia y adaptación del robot.
3. Crear un método para cambiar entre las siguientes configuraciones mediante el uso de una ontología y un razonador:
   a) Uso del sensor LIDAR con un funcionamiento normal.
   b) Uso de una cámara RGB-D con un funcionamiento normal.
   c) Uso del sensor LIDAR a velocidad baja.
   d) Uso de una cámara RGB-D a velocidad rápida.

Se asume que los modelos del entorno, la plataforma y los mapas está correctamente diseñados y no es necesario modificar nada a priori.

2. ANTECEDENTES E INVESTIGACIÓN PREVIA

En este capítulo se describen los antecedentes y los trabajos de investigación previos relacionados, que tienen que ver con el proyecto que se va a realizar. Además, se va a profundizar en conceptos previos al desarrollo del trabajo que es necesario explicar para tener una imagen clara del proyecto, además de para dar contexto a las soluciones propuestas y al por qué de esas soluciones.

La sección 2.1 tratará de dar un poco de detalles sobre el proyecto en el que se basa este trabajo.

La sección 2.2 abarcará los proyectos sobre metacontroladores que se han revisado como parte del proceso de investigación de este trabajo. Se verán los distintos intentos que se han realizado sobre como abordar este problema, y cuales son las ventajas y desventajas de la arquitectura.

La sección 2.3 trata sobre la integración de la arquitectura de robots móviles mediante ontologías, que se centrará principalmente en las iniciativas que ofrece la Web Semántica del World Wide Web Consortium (W3C). Aunque también se revisarán otros proyectos que realizaron sus propios desarrollos de ontologías personalizadas a sus proyectos a fin de poder entender ambas líneas de desarrollo. Principalmente el estándar que ofrece W3C es bastante utilizado para resolver problemas generalistas, mientras que las ontologías creadas exclusivamente para un proyecto concreto son demasiado específicas para poder aplicarse en otros problemas. En este sentido, la solución propuesta en el presente proyecto intenta cumplir los objetivos que se han propuesto en el capítulo 1, pero también poder dar una solución que pueda utilizarse en futuros proyectos sin tener que realizar cambios muy significativos en la arquitectura. Además, se explicarán conceptos básicos sobre ontologías y todo los conceptos e ideas clave que rodean a estas.

En la sección 2.4 se tratará de hacer una introducción a Robot Operating System (ROS), que es el framework para trabajar con el robot que se ha elegido para el proyecto. También se describirán algunos proyectos de navegación que han utilizado el stack ROS2, y que servirán como base para dar solución a los objetivos de navegación de este proyecto. En este caso, la solución que se busca es una reimplantación del stack de navegación de una versión de ROS a otra. Por lo que, en lugar de buscar una innovación muy grande en esta parte del proyecto, la investigación se centrará en determinar si se han producido cambios notorios en la navegación respecto de ROS1. De esta forma, se espera poder ofrecer una solución que cumpla los objetivos, pero que además se adapte a los nuevos estándares y arquitecturas de ROS2.

En la sección 2.5 se realizará una breve introducción sobre el simulador robótico de Gazebo Ignition. Esta sección se va a centrar en explicar los cambios más recientes sobre el simuladro, yq eu al gual que el paquete de ROS ha sufrido un reforma completa en su última distribución. Se explicará brevemente en que proyectos se está utilizando y cómo, para poder tener una visión más general de las características y funcionalidades que nos ofrece esta versión del simulador respecto de otras anteriores.

2.1. Proyecto MROS de RobMoSys

El proyecto que se ha utilizado como base para este documento es un proyecto RobMoSys llamado MROS.

RobMoSys[1] es un consorcio de entidades con un enfoque de desarrollo basado en modelos para la robótica, que tiene como objetivo coordinar los mejores esfuerzos de toda la comunidad robótica para dar un paso adelante hacia un ecosistema europeo de desarrollo de software de calidad industrial. Para llevar a cabo este enfoque tienen un principio general fundamental que aplican en sus proyectos y estándares: la componibilidad. La componibilidad es la propiedad que hace que los componentes se conviertan en bloques de construcción, que puedan ser reutilizados teniendo en cuenta solo sus interfaces, y no su contenido. Esto aumenta las opciones de que los sistemas robóticos puedas ser integrados en multitud de plataformas y para completar objetivos similares.

Uno de los proyectos, y en el que se basa parcialmente este proyecto es MROS[2]. El objetivo de MROS es aprovechar el enfoque basado en modelos de RobMoSys para proyectos que quieran implementarlos en runtime. Esto lo consigue implementado un metacontrolador y utilizando la ontología de TomaSys[3].

2.2. Meta-controlador

Dentro de los metacontroladores, los que están enfocados a robótica móvil y que además utilicen ontologías son la minoria del estado del arte. Son muy pocos los proyectos que los hayan usado como tal y los pocos que lo ha usado están estrechamente relacionados con el proyecto anterior.

Enfoques que hayan puesto un modelo ontológico o conceptos similares, junto con un razonador para regular las tareas de control de otros procesos, se pueden encontrar diferentes ejemplos. Se pueden encontrar proyectos de IA [4], donde añaden un bucle de realimentación utilizando lo que llaman un lazo metacognitivo (MCL). En otros trabajos [5] se propone el diseño de componentes de software autoadaptativos basados en enfoques lógicos de control discreto, en los que los modelos de comportamiento autoadaptativos enriquecen a los mismos controladores de los componentes con un conocimiento. Este enfoque consigue mejorar la robustez del sistema.

Ya enfoques más centrados en trabajos relacionados tanto con el proyecto de este documento, como con el proyecto en el que nos vamos a basar, encontramos conjuntos de publicaciones con los mismos objetivos. Estos trabajos manejan el concepto de dos ontologías: la ontología del vehículo robot que se utiliza para describir el sistema; y la ontología del metacontrolador, que es la ontología con las reglas y la estructuras para poder realizar el control de la primera ontología. Este tipo de modelos se han implementado en vehículos autónomas con la finalidad de controlar procesos y modos de ejecución del propio vehículo [6][7].

2.3. Ontologías para robótica móvil

Uno de los enfoques para hacer que la arquitectura de robots móviles dote a estos vehículos de una mayor autonomía y adaptabilidad, es el uso de instrucciones de alto nivel para simplificar la programación de robots para diferentes tareas, por ejemplo, utilizando lenguaje natural [8] y de acciones primitivas parametrizadas y definidas semánticamente [9], o habilidades. Estas descripciones semánticas son deseables porque permiten la generación automática de descripciones PDDL (Planning Domain Definition Language) para la planificación y programación de tareas o la planificación y generación de trayectorias a partir de modelos virtuales[10]. Sin embargo, en la mayoría de los casos estas abstracciones asumen que se posee una biblioteca de habilidades o acciones primitivas que pueden llegar a ser muy complejas y altamente dependientes del controlador del robot en el que se ejecutan, lo que las vuelve muy dependientes de su arquitectura. Y por tanto, limita su utilización a tareas muy específicas y perfectamente coordinadas, que pueden no ser útiles para otras configuraciones en otros robots.

La robótica basada en el conocimiento exige un razonamiento explícito y detallado sobre las capacidades del robot. Las ontologías facilitan esta tarea al hacer que este conocimiento sea explícito y transferible entre agentes. Existen varios proyectos que desarrollan este tipo de bases de conocimiento robótico, como KnowRob[11], RoboEarth[12] u Open Robotics Ontology (ORO)[13], pero están más centrados en la robótica para servicios y minimizan mucho las opciones de movilidad de los vehículos, hasta el nivel de implementar acciones de movimiento muy básicas, como girar y avanzar. Un buen resumen del estado del arte en cuanto a robots de servicio que utilizan ontologías para solventar problemas similares puede encontrarse en Haideger et al.[14].

En el caso de la robótica industrial, hay una menor cantidad de trabajos que hayan tratado este tema con profundidad, pero existen iniciativas abiertas para caracterizar semánticamente conceptos básicos que se puedan utilizar en entornos industriales. Una de ellas es la Core Ontology for Robotics and Automation (CORA)[15], basada a su vez en la Suggested Upper Merged Ontology (SUMO)[16]. El objetivo de estos proyectos es desarrollar un estándar para proporcionar una ontología global y una metodología para la representación del conocimiento y el razonamiento en robótica y la automatización, junto con la representación de conceptos en un conjunto inicial de dominios de aplicación. Intenta proporcionar una forma unificada de representar el conocimiento y un conjunto común de términos y definiciones, que permita una transferencia de conocimientos sin ambigüedades entre cualquier grupo de humanos, robots y otros sistemas artificiales. Sin embargo, los conceptos allí introducidos son más bien básicos, muy centrados en el desarrollo industrial y no alcanzan la complejidad necesaria para caracterizar el mapa de conocimiento que se necesitan para los objetivos del proyecto.

Los proyectos que utilizan las bases de conocimientos que ofrece W3C no son muchos, y se centran mayoritariamente en la organización del conocimiento dentro de la ontología, o son de áreas distintas a la robótica, por lo que sus soluciones son especificas de su área de aplicación. Un ejemplo general sería Compton et al.[17], que revisa un total de 12 ontologías publicadas entre 2001 y 2009. En este articulo se profundiza en el poder expresivo de las ontologías, y en además se especifica que conceptos no pueden describir con suficiente precisión. La mayoría de sus artículos sobre ontologías de sensores se centran en la Semantic Web y en los esfuerzos de crear proyectos en los que agentes autónomos puedan encontrar y utilizar sensores conectados a internet. Se trata de un entorno mucho más amplio y abstracto que el dominio de este proyecto, pero es un análisis muy completo y profundo.

Otros proyectos más recientes serían: OntoSensor[18], una ontología que pretende ser una base de conocimiento general de sensores para consulta e inferencia; y SensorML[19], que es una espe-

cificación de un modelo de datos genérico en Unified Modeling Language (UML) para capturar clases y asociaciones que son comunes a todos los sensores. Ambas ontologías utilizan definiciones de conocimientos muy generales y, aunque serían perfectamente adaptables a nuestro proyecto, se han quedado anticuadas y hay mejores opciones actualmente, tal y como se expondrá más adelante.

Para finalizar esta sección se va a mencionar algunos intentos interesantes y valiosos que intentan solucionar objetivos similares a los planteados en el proyecto, pero que fueron muy limitados en su alcance o hacen referencias a versiones en desuso, como Topp et al.[20] y Balakirsky et al.[21]. Un enfoque muy interesantes orientado a la introducción de una ontología de habilidades y tareas para ROS sería SkiROS[22], sin embargo la ontología del sistema SkiROS introducida allí demasiado específica y busca resolver otros tipos de problemas diferentes a los objetivos del proyecto.

No se conoce ningún proyecto relevante que haya utilizado las ontologías de sensórica propuestas por W3C para utilizarlas en la arquitectura de un robot móvil un robot móvil.

2.3.1. Definiciones básicas de ontologías

Aunque se ha abordado la definición de ontología en apartados anteriores, y se han ofrecido varias formas de explicar el concepto, se va a dar su definición formal para tenerla de referencia. Esta definición es la que es mayormente es aceptada dentro de la comunidad científica del conocimiento en la actualidad. Como dato adicional, destacar que es un termino que indudablemente viene de la filosofía y ha pasado por un largo proceso de modificaciones e interpretaciones a lo largo de la historia hasta dar con una definición satisfactoria. La siguiente definición la acuñó Borst en 1997[23]:

"Las ontologías son una especificación formal de una conceptualización compartida"

En esta definición, el término conceptualización se refiere al modelo abstracto en el que se identifican las características y conceptos más relevantes, ocurriendo este fenómeno en el mundo. La especificación debe ser formal ya que debe ser completamente interpretable por una máquina, no solo por personas humanas. La conceptualización debe ser compartida porque la ontología necesita tener una especificación que sea aceptada por el mayor número de personas posible[24].

El concepto tal y como está definido permite aunar, por ejemplo, tipos o relaciones entre entidades que existen para un uso particular en un campo concreto. Es, por tanto, un elemento fundamental para intercambiar información entre distintas herramientas de trabajo de la información, haciendo uso de una sintaxis común.

Además de definir lo que se considera una ontología propiamente, también se van a ofrecer las definiciones de sus principales componentes. Los principales elementos y definiciones se muestran a continuación, todas ellas basadas en Gruber, Thomas R. [25].

- **Clases:** Son básicamente conceptos, ideas básicas que se intentan formalizar. Éstas pueden ser un objeto, un método, un plan, una estrategia, un proceso de razonamiento... En definitiva, son las abstracciones de las entidades del sistema.
- **Atributos:** es la forma de organizar la estructura interna de las clases. Pueden ser específicos (propios de la clase) o heredados (se establecen por las relaciones que las clases

mantienen con otras clases). Se pueden declarar una serie de atributos clave dentro de cada clase de forma que no se pueda instanciar más de una particularización de dicha clase con los mismos valores de los atributos. Así, pueden buscarse elementos de una clase a través de sus atributos. Su dominio es la clase, y su rango el tipo de elemento que sea instanciado.

- **Relaciones:** representan las interacciones entre las clases. Conforman la taxonomía del dominio, por ejemplo, "parte de", "subparte de", "posee a" ... Existen relaciones de tipo binarias como las taxonómicas ("es un") o las partonómicas ("parte de"). Su dominio es la clase de relación, y su rango es la clase de individuos que se va a relacionar.
- **Funciones:** Son un tipo concreto de relación donde se identifica un único elemento mediante el cálculo de una función que considera varios elementos de la ontología. Sirva de ejemplo la función "padre de".
- **Instancias:** Sirven para representar objetos determinados de una clase. Es la forma de particularizar la clase. Se utilizan para representar elementos específicos.
- **Axiomas:** Son teoremas que se establecen sobre relaciones que deben cumplir los elementos de la ontología. Por definición, axioma quiere decir algo que siempre es verdad, y así debe aplicarse también en las ontologías. De esta manera, se convierten en una herramienta para definir relaciones, restricciones entre atributos, entre otros.

Todos estos elementos permiten crear los denominados Individuos (Individuals) que no son más que las particularizaciones de una clase que ha sido instanciada para un determinado caso. Dicho de otro modo, es la creación de entes de cualquier tipo, al que se le han asociado valores concretos a sus atributos para definirlos con concreción. Las ontologías se manejan a través de un lenguaje web semántico específico, el denominado Lenguaje de Ontologías Web (OWL). Este lenguaje fue creado para procesar ontologías, de manera que la información que estaba definidas en estas se pudiera presentar sin ambigüedades a personas o a maquinas[26]. OWL posee una mayor capacidad para expresar con claridad y precisión los significados y las semánticas que otros lenguajes conocidos, como pueden ser el RDF o el RDF Schema (RDF-S). Por tanto, OWL es capaz de llegar más lejos en cuanto a resultados de interpretación de ontologías se refiere, haciendo explícito el conocimiento implícito.

Esto ultimo es capaz de conseguirlo a través de un vocabulario más amplio para describir clases y atributos, o añadiendo más y distintas relaciones entre clases, como igualdad, cardinalidad, clases enumeradas... Es decir, existe una particularización en lenguaje OWL de cada uno de los elementos que definen las ontologías y que se presentaron anteriormente.

2.3.2. Resource Description Framework (RDF)

Resource Description Framework (RDF) es un formato para codificar datos de Resources(Recursos) de la web desarrollado por W3C [27]. Aunque en términos generales, RDF permite la definición de declaraciones lógicas con la siguiente estructura:

(sujeto predicado objeto)

Donde el sujeto es un Recurso, el predicado es una Propiedad del sujeto que se relaciona con el objeto, y el objeto como tal puede ser otro Recurso o datos (data literal). Cada Recurso tiene un Internationalized Resource Identifier (IRI), que lo identifica de forma unívoca. Un ejemplo válido de RDF menos abstracto que el anterior sería:

(stuart hasName "Stuart")

Esta representación indica que el literal "John.^{es} el valor de la Propiedad hasName del Recurso john. Un conjunto de datos RDF se modela con un grafo, donde Los recursos y los literales son los nodos y los predicados son las aristas. De esta forma se pueden relaciones de datos de forma sencilla y gráfica.

El RDF no se utiliza de forma explicita en el proyecto, pero se podrá ver su influencia a lo largo de este, ya que muchas de las ideas e innovaciones sobre ontologías parten de esta idea como inicio.

2.3.3. Web Ontology Language (OWL)

OWL aumenta la potencia expresiva que tiene RDF con constructores adicionales, que incluyen las siguientes capacidades de representación del conocimiento:

- La definición de Propiedades de Objeto, Propiedades de Datos, Propiedades transitivas, simétricas y funcionales.
- La definición de una Propiedad como la inversa de otra.
- La definición de nuevas clases mediante la especificación de restricciones sobre las propiedades.
- La definición de clases complejas como uniones, intersecciones y complementos de otras.
- La def de clases mediante la enumeración de sus instancias.
- La definición de mapeos entre clases e individuos.

Como se puede entrever, estas adiciones facilitan que la descripción del conocimiento sea muchos más rica y compleja, pudiendo especificar relaciones semánticas muy estructuradas que otros lenguajes no permiten. Se profundizará más en este lenguaje más adelante, durante el desarrollo del proyecto.

2.3.4. Semantic Web Rule Language (SWRL)

El Semantic Web Rule Language (SWRL) es un lenguaje estándar basado en OWL-DL, una extensión de OWL, y en el Rule Markup Language (RuleML) que proporciona tanto la expresividad de OWL-DL como las reglas de RuleML. Es uno de los primero enfoques que reúne ontologías y reglas para el desarrollo de arboles de conocimientos con normas marcadas. Las reglas en SWRL son reglas de implicación. Por lo tanto, la sintaxis de SWRL es de la siguiente forma[28]:

(antecedencia $\longrightarrow$ consecuencia)

Esta estructura implica que la consecuencia debe ser factible y verdadera cuando se satisface las condiciones expuestas en la antecedencia. Las expresiones OWL pueden aparecer tanto en el antecedente como en el consecuente[29]. Según la literatura, se pueden expresar diferentes tipos de reglas en SWRL. Se profundizará en las reglas y en como se utilizan en este proyecto en posteriores apartados.

2.4. ROS2: Robot Operating System

ROS2 es una versión bastante reciente del conocido framework de trabajo para robots llamado ROS. Aunque existen a día de hoy varias versiones de ROS2, al ser una renovación completa del framework las versiones tienen muchas diferencias entre si. Además muchos de los paquetes que se han utilizado ampliamente en desarrollos con ROS1, aun no han sido portados por completo a ROS2, por lo que muchos proyectos han seguido trabajando hasta el día de hoy con las versiones de ROS1 en vez de con las de ROS2.

Aun así, se ha investigado sobre la usabilidad de ROS2 para sistemas parecidos al del proyecto, y se revisado varios de los marcos de trabajo propuestos. Esta revisión bibliográfica también contará con la investigación sobre arquitecturas de comunicación en forma de publicador-suscriptor en sistemas que utilicen ROS2, incluyendo la ejecución de los nodos, así como la comunicación en el DDS subyacente.

ROS no es un sistema operativo propiamente dicho como tal, sino un meta-sistema operativo con muchas funcionalidades orientadas a la robótica y un sistema de comunicación propio. En otras palabras, ROS funciona por encima de otros sistemas operativos y permite que diferentes procesos se comuniquen entre sí durante el tiempo de ejecución. Como meta-sistema operativo, ROS ofrece una capa de comunicaciones, de forma estructurada, que se ejecuta sobre los sistemas operativos de los ordenadores anfitriones. El uso de ROS no se limita únicamente a la robótica, ya que la mayoría de las funcionalidades y herramientas de ROS son compatibles con otros tipos de hardware, y pueden utilizarse para fines diferentes a la robótica. El núcleo de ROS consta de más de dos mil paquetes, donde cada paquete tiene su funcionalidad específica. Estos a su vez de dividen en agrupaciones de paquetes que están estrechamente relacionados, y que permiten integrar funcionalidades más amplias. Estas agrupaciones reciben el nombre de stack, y algunos ejemplos pueden ser el stack de navegación o el stack de localización que se utilizan en parte en este proyecto.

Por último, ROS es un conjunto de herramientas que proporciona la funcionalidad y los servicios de un sistema operativo en un solo ordenador o en varios. Estos servicios incluyen la abstracción del hardware, el intercambio de mensajes entre procesos, la gestión de paquetes, etc. sostiene que la mayor fortaleza de ROS es el número de herramientas y paquetes de ROS que hay disponibles. La naturaleza abierta y comunitaria que tiene ROS y su entorno es lo que hace esto posible. Hay muchas áreas diferentes de programación de robótica, y para una persona o un equipo saber cómo resolver todos los problemas de programación robótica es casi imposible. Debido a ello, los equipos y laboratorios de programación robótica necesitan cooperar entre sí para crear soluciones robóticas complejas[30].

2.4.1. Funcionamiento general de ROS2

Una vez instalado ROS se va a explicar de forma resumida como está estructurado y cuáles son los elementos principales por los que está compuesto. ROS está formado por varios niveles separados que interactúan entre ellos para realizar las acciones que se requieran, y esto forma parte fundamental de su diseño modular.

2.4.1.1 Sistema de archivos

ROS utiliza una estructura de archivos concreta para tener toda la información estructurada y saber qué recursos tiene disponibles en todo momento. Un diagrama de la jerarquía del sistema de archivos se puede ver en la figura a continuación.

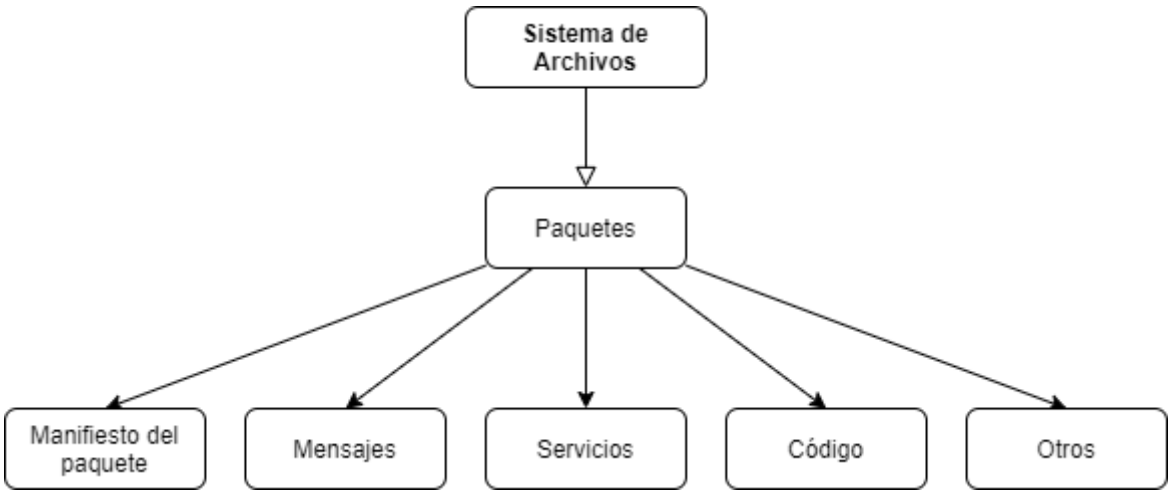

Figura 2.1: Jerarquía del sistema de archivos

Este sistema de archivos está dividido en carpetas y subcarpetas en las que están guardadas los archivos con las funcionalidades correspondientes. Estas carpetas son las siguientes:

- **Paquetes (Packages):** Los paquetes son el elemento básico del sistema de archivos y tienen el contenido mínimo para crear un programa dentro de ROS. También pueden estar programados como nodos, archivos de configuración, etc.
- **Manifiestos de los paquetes (Package manifests):** Los manifiestos proporcionan información sobre el paquete al que están ligados, las dependencias que va a utilizar, los argumentos de compilación, etc. Estos archivos son fácilmente reconocibles ya que siempre se llaman package.xml.
- **Metapaquetes (Metapackages):** son agrupaciones de paquetes que dependen unos de otros de alguna manera (referencias, dependencias, etc.) y que necesitan construirse u operarse juntos.
- **Mensajes (Messages):** Un mensaje es toda la información que un proceso manda o recibe de otros procesos usando el sistema de mensajería de ROS. ROS posee una gran librería de mensajes predeterminados que se usan para una mayor compatibilidad entre paquetes, aunque existe la posibilidad de crear estructuras de mensajes personalizados.

En la siguiente figura se puede ver la estructura del metapaquete del proyecto, donde se puede observar la estructura del sistema de archivos y muchos de los tipos de archivos anteriormente mencionados. Este ejemplo se puede encontrar directamente en la instalación de ROS cuando nos encontremos en nuestro espacio de trabajo (workspace) y, desde el terminal, ejecutemos el comando tree.

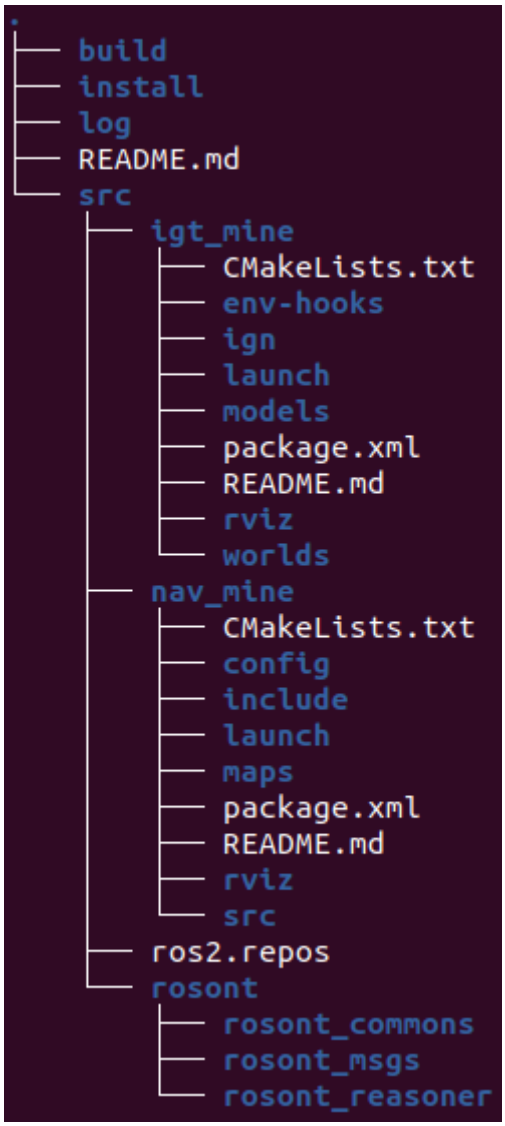

Figura 2.2: Estructura del paquete nav_sim, igt_sim y el metapaquete rosont.

2.4.1.2 Nodos (Nodes)

Como ya se ha mencionado en el anterior apartado, los paquetes forman el elemento más básico de la estructura de archivos de ROS. Normalmente un paquete es una combinación de carpetas y archivos que posee la siguiente estructura:

- **include/:** en esta carpeta se incluyen las cabeceras de las librerías que se van a usar.
- **msg/:** los mensajes personalizados deben ir en este directorio.
- **launch/:** los programas para el lanzamiento de los nodos del paquete se encuentra en este directorio.
- **scripts/:** los scripts ejecutables que se vayan a usar en el proyecto deben ir aquí.
- **srv/:** los servicios de paquete van en esta carpeta.
- **Package.xml:** Este es el manifiesto del paquete con la configuración de este.

Estos paquetes pueden contener un nodo de ROS, una librería externa que no se encuentre en ROS, archivos de datos externos, archivos de configuración, una aplicación de terceros o cualquier otro elemento que constituya un módulo útil para ROS.

Un **nodo** es un proceso que participa en la red de comunicación de ROS, ya sea mandando o recibiendo mensajes, mediante el servicio RPC o con el Servidor de Parámetros. Un ejemplo Este nodo recibe un identificador propio y único dentro del entorno de ROS, lo que hace que sea visible por el resto de nodos de la red. Esto se puede ver en la figura siguiente, en la que los nodos están representados por óvalos y los tópicos son las flechas que los unen.

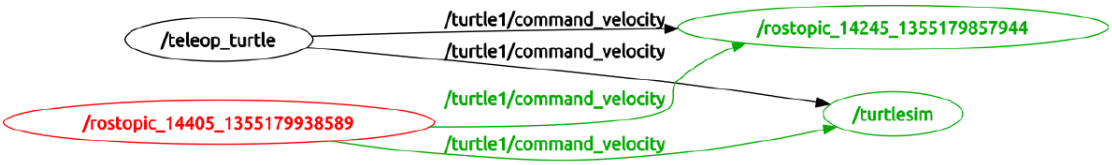

Figura 2.3: Grafico de conexiones entre nodos mediante diferentes topics.

Los nodos se construyen usando las librerías bases de ROS, que son la librería rclcpp, para nodos programados en C++, y la librería rclpy, para nodos programados en Python. Dentro de un mismo paquete puede haber varios nodos con los que se pueden usar todo tipo de librerías y herramientas, siempre y cuando se configuren adecuadamente.

2.4.1.3 Temas (Topics)

Los temas o topics son como buses de datos en los que los nodos pueden publicar y leer información. Estos buses pueden ser creados por cualquier nodo siempre que el nodo esté conectado a la red de ROS y que use un identificador único para cada topic. Los topics se pueden publicar sin necesidad de que haya otro nodo suscrito, pero un nodo no se podrá conectar a un topic si este no está creado. Además, cualquier número de nodos pueden publicar o se pueden suscribir a un topic concreto sin ninguna limitación, pero durante la publicación hay que tener cuidado para no crear conflictos entre publicaciones simultaneas.

Los topics en ROS se pueden transmitir usando los protocolos TCP/IP y UDP, por lo que los nodos deben negociar que protocolo usar según sus posibilidades. Esto hace que exista la posibilidad de poder conectar varias máquinas y que, mientras solo una haga de maestro, se puedan usar nodos y topics entre las dos máquinas de tal forma que todos formen parte de la misma red de ROS.

2.4.1.4 Mensajes (Messages)

La información que se intercambia en los topics se hace usando mensajes, que son estructuras de datos concretas que están diseñadas de antemano por el programador. Los mensajes usan diferentes tipos de datos y estructuras que deben ser conocidas siempre por todos los publicadores o suscriptores de un topic, de forma que sean capaces de construir o interpretar el mensaje respectivamente.

2.4.1.5 Servicios (Services)

El sistema de comunicación de ROS a base de mensajes está principalmente pensado para situaciones en las que tenemos un número indeterminado de publicadores y suscriptores. Debido a que algunos tipos de comunicaciones necesitan un modelo basado en solicitud y respuesta, es decir, se necesita que un nodo (activo) solicite a otro nodo (pasivo) un tipo de información y este último debe responder con lo que se le pide, pero solo cuando se le ordena. Los servicios cubren este tipo de comunicación, por lo que un servicio define un conjunto de dos mensajes de forma que en el primero se especifica la estructura y el tipo de información que define la orden, y en el segundo la estructura y el tipo de datos de la respuesta. De esta forma, un nodo ofrece el servicio y el resto de los nodos pueden hacer uso de este.

2.4.2. Principales diferencias entre ROS1 y ROS2

El equipo principal de desarrollo que está detrás de ROS1 ha aprendido, con todos esos años de experiencia, qué limitaciones presenta, qué características importantes faltan dadas las últimas innovaciones, y qué se podría mejorar de la arquitectura de la primera versión. Lamentablemente, este tipo de cambios profundos harían que la (mas o menos) existente retrocompatibilidad que se había podido aprovechar hasta ese entonces, desapareciera por completo. Así que ROS2 se desarrolló desde cero, como un proyecto completamente diferencia dos de ROS1 y por tanto es un framework de trabajo completamente nuevo.

Por ahora ROS no es muy popular en la industria, y carece de algunos de los requisitos más importantes, como el tiempo real, la seguridad, la certificación, la seguridad. Uno de los objetivos de ROS2 es hacerlo compatible con las aplicaciones industriales.

A continuación se va a profundizar en los cambios más significativos de ROS2 respecto de ROS1.

2.4.2.1 Cambios en la API de ROS

En ROS1, para C++ se utiliza la librería roscpp, y para Python, la librería rospy. Cada biblioteca se programó y creó de forma independiente y aisladamente la una de la otra. Comparten funcionales comunes en los estándares de ROS, pero hay diferencias sustanciales en cuanto métodos, funciones y utilidades, e incluso algunas características fueron desarrolladas e implementadas en una de ellas, pero no en la otra.

ROS2 tiene más capas que ROS1. Sólo hay una biblioteca base, llamada rcl, e implementada en C. Esta es la biblioteca base que contiene todas las características del núcleo de ROS2, y que se utilizarán en el desarrollo de los paquetes y nodos. No se utiliza la biblioteca rcl directamente al programar, se utilizan wrappers para cada uno de los lenguajes aceptados en ROS, por ejemplo: rclcpp para C++, rclpy para Python.

La ventaja respecto de ROS1 parece obvia, cualquier nueva funcionalidad que se quiera añadir solo necesita ser implementada en rcl. Después, solo será necesario actualizar los wrappers sobre rcl con los enlaces a la nueva funcionalidad.

2.4.2.2 Cambios en las versiones de Python y C++

En las últimas versiones de ROS1, como en Noetic, se integró el uso de Python3 para poder utilizarlo durante el desarrollo, y se quitó la compatibilidad con Python2. En este sentido no supone mucho cambio respecto a ROS2 ya que se sigue utilizando Python3 y no se puede desarrollar utilizando Python2.

En el caso de C++ si cambian más cosas. Mientras que las versiones de ROS1 utilizaban la especificación C++98, en ROS2 se puede utilizar las especificaciones C++11 y C++14 por defecto. Permitiendo hacer uso de los últimos cambios e innovaciones en el uso de este lenguaje, y permitiendo a los paquetes poder utilizar sin temor a no ser compatibles después. Hay planes para añadir la especificación de C++17 en un futuro.

2.4.2.3 Nodos: Cambios en las convenciones

Para el desarrollo de nodos en ROS1 no había una forma clara de estructurar el código para crear un nodo y codificar sus funcionalidades. Se podían añadir callbacks y funciones en cualquier parte del código, lo que volvía la implementación de una persona completamente única respecto de las demás. Esto provocaba problemas y errores a la hora de compartir módulos o stacks con la comunidad.

En ROS2 se ha cambiado esto y se ha creado una convención clara sobre cómo escribir tus nodos. Lo principal es que tienes que crear una clase que herede del objeto Node (por ejemplo: rclcpp::Node en C++, rclpy.node.Node en Python), y en esta clase se tienen todas las funcionalidades de ROS2.

2.4.2.4 Nodos: Múltiples nodos en el mismo ejecutable

En ROS1 un nodo está unido a un ejecutable. Una nueva funcionalidad, llamada Nodelets, fue añadida en ROS1 para poder escribir múltiples nodos en el mismo ejecutable, con comunicación intra-proceso, que es una funcionalidad muy útil cuando se tienen recursos de hardware limitados y/o se necesita enviar muchos mensajes entre nodos.

En ROS2, los Nodelets ya no se llaman Nodelets. La funcionalidad se ha incluido directamente como una de las características base de ROS2, y ahora se llama çomponentes". Así que, con ROS2, se puede manejar muchos nodos desde el mismo ejecutable, usando distintos componentes. Un componente es simplemente una clase de nodo ligeramente modificada para permitir la ejecución simultánea. Se puede iniciar los componentes desde un archivo de lanzamiento, la terminal, o desde un ejecutable. Y se puede activar la comunicación intra-proceso para eliminar cualquier sobrecarga de comunicación de ROS2.

2.4.2.5 Nodos: Nodos con ciclo de vida

ROS2 introduce el concepto de nodos con ciclo de vida. Un nodo con ciclo de vida tiene diferentes estados: no configurado, inactivo, activo y finalizado. Esto es muy útil cuando se necesita una fase de configuración antes de ejecutar las principales funcionalidades del nodo. Cuando se inicia un nodo de este tipo, inicialmente no está configurado, y mediante los servicios ROS2 se va cambiando de una transición a otro estado, utilizando los callbacks predefinidos para ello.

Por ejemplo, con el ciclo de vida se puede separar claramente las principales etapas de la ejecución de un nodo para un sensor: primero se asigna la memoria para los publicadores, suscriptores y otros objetos instanciados. Luego, se inicia la comunicación con el sensor. Y finalmente se ejecuta el bucle de lectura para publicar los datos.

2.4.2.6 Archivos de lanzamiento

En ROS1, los archivos de lanzamiento estaban escritos en XML y tenían una estructura muy marcada en cuanto las funcionalidad y lo que se podían realizar en ellos.

En ROS2 ahora se puede utilizar Python para escribir los archivos de lanzamiento, lo que permite realizar lanzamientos con una mayor personalización ajustada a cada caso según sea necesario. Para ello se ha añadido una API que permite iniciar nodos, recuperar archivos de configuración, añadir parámetros, etc.

2.4.2.7 Comunicación

En ROS1 existía la figura de master en ROS, y era un módulo base que debía ejecutarse antes de ejecutar cualquier otro elemento de ROS. El master en ROS1 actuaba como un servidor DNS para los nodos, otorgándoles un identificador único, permitiendo descubrir otros nodos de la red y que se dieran a conocer en la red.

En ROS2 se elimina este módulo y se le otorga a cada nodo la capacidad de descubrir otros nodos. Este cambio permite crear un sistema totalmente distribuido en el que cada nodo es completamente independiente y no está ligado a un master global que le permita realizar sus funciones.

Otra diferencia notable es que al crear una aplicación en ROS2 que afecte a varias máquinas, no se tiene que definir una máquina como master. Cada máquina es independiente y podrá arrancar por sí misma, conectarse y desconectarse entre sí, con menos configuración que en ROS1.

También desaparece el concepto de parámetros globales, ya que el servidor de parámetros que estaba ligado al master node desparece también. En ROS2 los parámetros están ligados a un nodo, y cada nodo declara y gestiona sus propios parámetros, y esos parámetros se destruyen cuando el nodo es eliminado.

Para la reconfiguración dinámica de parámetros, en lugar de depender de un servicio externo, ahora se pueden modificar creando una función de callback en el propio nodo.

2.4.2.8 Servicios y acciones

En ROS1, los servicios son síncronos es decir, cuando el cliente del servicio hace una petición al servidor se queda esperando hasta que el servidor responde (o falla). Además, en ROS1 las acciones nunca estuvieron dentro de la funcionalidades base de ROS, ya que fueron una adición hecha después de algunos años, para resolver el problema de que los servicios no eran asíncronos, y no tenían un mecanismo de retroalimentación o cancelación. Por lo tanto, las acciones en ROS1 están totalmente construidas sobre los topics de ROS1 y lo que estos pueden ofrecer.

En ROS2, los servicios son asíncronos. Cuando llamas a un servicio, puedes añadir una función

de callback que se activará cuando el servidor responda. Mientras tanto, tu hilo principal puede seguir funcionando con normalidad. Las acciones siguen utilizando topics para la retroalimentación y el estado de los objetivos, pero también servicios (asíncronos) para establecer y cancelar un objetivo o solicitar un resultado.

2.4.2.9 Sistema de construcción

En ROS1 para la construcción de todos los elementos de una aplicación o paquete se utilizaba catkin, que combinabas macros de Cmake con scripts de Python para proporcionar un sistema de construcción propio de ROS.

En ROS2 se ha cambiado este sistema a una versión nueva llamada ament, que también utiliza CMake y scripts para adaptar ciertas funcionalidades a las necesidades de ROS.

En su uso normal no implican mayores cambios que cambiar los comandos que se deben utilizar para construir utilizando una herramienta u otra, y para lanzar los paquetes construidos. Pero se han cambiado como se construyen los paquetes, como se enlazan con el resto del sistema, y una lista de pequeñas mejoras que no merece detallar salvo para funcionalidades muy especificas. Para el desarrollo de este proyecto esto ha tenido un impacto mínimo.

2.4.3. Robot Stack

Debido a que ROS es un sistema de código abierto y muy modular, hay una gran comunidad que desarrolla usando ROS y que crean paquetes y herramientas que comparten en los repositorios públicos de ROS. Si la comunidad encuentra estos paquetes útiles y los empiezan usar en sus proyectos de forma continua, al final se acaban incorporando en la dinámica habitual de desarrollo y acaban formando parte de los paquetes básicos para determinadas aplicaciones.

Es común que cuando varios paquetes que estén estrechamente relacionados porque se usan mucho para una determinada aplicación, estos paquetes se agrupen formando lo que se llama metapaquete o stack.

Uno de estos grupos de paquetes es el que se conoce como Robot Stack que, como su propio nombre indica, es un conjunto de paquetes básicos que contienen los nodos y herramientas más utilizadas en el desarrollo y control de robots. Los paquetes principales son los siguientes:

- **control_msgs:** este paquete contiene algunos de los mensajes, las acciones y los servicios básicas para el control de robots y sus elementos. Se incluyen los mensajes de diferentes datos de las articulaciones (estados, tolerancias, etc.), mensajes para la configuración de los PID,etc.
- **nav_msgs:** contiene tipos de mensajes utilizados para interactuar con la pila de navegación y los paquetes relacionados. Define tipos de mensajes como OccupancyGrid que representa un mapa y Odometry que contiene datos de odometría.
- **sensor_msgs:** este paquete contiene los principales mensajes que se utilizan con los sensores para la comunicación de los datos obtenidos por estos. Se consideran un estándar en ROS, y están integrados en la gran mayoría de paquetes principales de ROS.

- **geometry:** este metapaquete incluye paquetes para diferentes conversiones geométricas, transformación de coordenadas, paquetes para la cinemática y dinámica del robot e incluye un motor básico de físicas para colisiones.
- **urdf:** este paquete está formado por un parser en C++ que permite leer y entender los archivos escritos en Formato Unificado de Descripción de Robots (URDF), el cual se usa para construir y diseñar el modelo del robot que se va a controlar. En este modelo se suelen incluir las posiciones de los elementos que forman el robot, sus articulaciones, especificaciones inerciales, etc.
- **xacro:** este paquete permite usar los Xacros (XML Macros) en los archivos URDF, de forma que se puedan usar macros para reducir la complejidad de los archivos cuando tenemos muchos elementos con las mismas características.
- **xmacro:** este paquete es una renovación del paquete de XACRO, pero más centrado en los modelos SDF en lugar de los modelos URDF. Permite la conversión de modelos para poder utilizarlos en ROS2.
- **joint_state_publisher:** este paquete publica el estado (nombre, posición, velocidad y esfuerzo) de todas las articulaciones móviles programadas en un modelo del robot que se ha diseñado usando el formato URDF y que se publica como parámetro dentro de ROS.
- **robot_state_publisher:** este paquete publica el estado de las articulaciones del robot mediante otro paquete llamado tf2, el cual permite que se puedan realizar transformaciones entre cualquiera de las articulaciones. Este paquete toma los ángulos de unión del robot como entrada que están especificados en el modelo URDF y publica las poses 3D de los enlaces del robot, usando un modelo de árbol cinemático del robot.

Este paquetes conforman lo que se considera el stack base para lo que será el control del robot, y se complementará con otros paquetes que ofrece ROS y con los paquetes que se desarrollen para este proyecto.

2.4.4. Navigation Stack

Este es otro de los stacks importantes que se va a usar en el proyecto, pero en menor nivel ya que muchos de los paquetes que forman este stack no forman parte de los objetivos de este proyecto. Este stack proporciona los paquetes básicos para una navegación en 2D, lo cual incluye paquetes que toman los datos de odometría y los streams de datos de los sensores, y devuelven los comandos de velocidad que los motores deben de desarrollar para llegar a una pose objetivo. Los nodos y los tópicos básicos para un uso estándar del stack de navegación se pueden ver en la figura presentada a continuación:

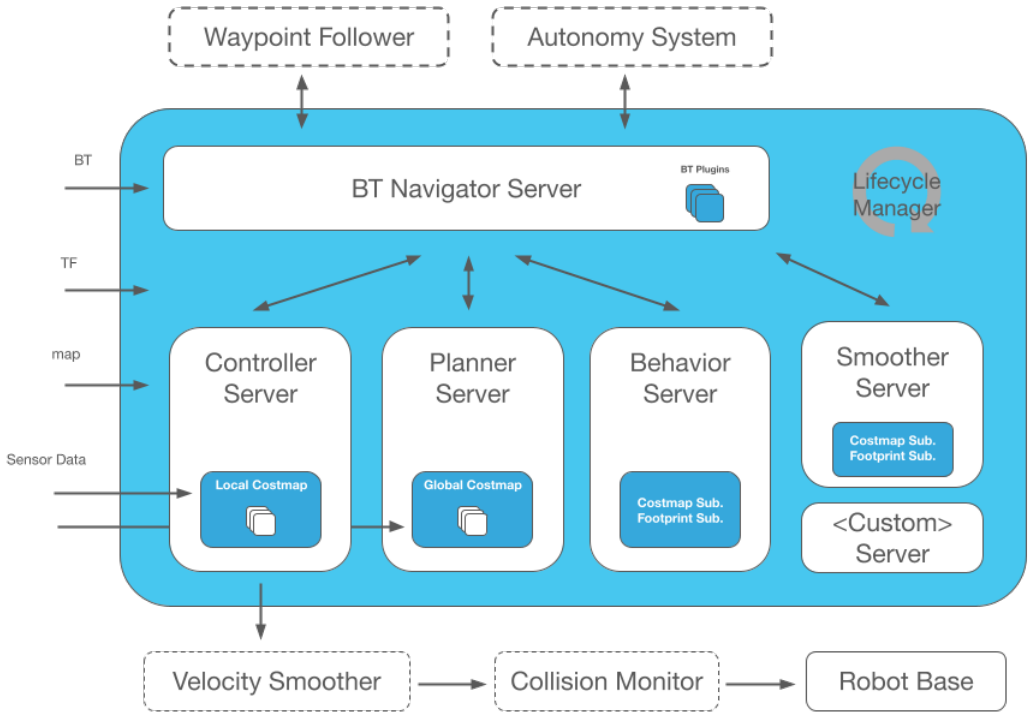

Figura 2.4: Arquitectura standard del stack de navegación en ROS.

Este stack está centrado en el path planning y la consecución de objetivos locales y globales, y es una pieza básica a la hora de desarrollar un robot con ROS que pueda realizar la técnica del SLAM. Para lo cual necesitaríamos una mayor gama de sensores de la que no disponemos, como un escáner láser para poder construir un mapa preciso y detallado del entorno. Aun así, en este stack se encuentran los mensajes básicos para odometría, se detalla el uso de la herramienta Rviz, que permite visualizar la información con la que trabaja el robot (odometría, sensores, etc.), y se profundiza en el uso del paquete tf2, para realizar transformaciones entre marcos de coordenadas usando el modelo URDF del robot.

En la transición ROS a ROS2, se han cambiado bastantes cosas y funcionalidades del propio stack de navegación. Los cambios se han centrado más en solventar las limitaciones y en mejorar la arquitectura que ya se poseía en ROS1.

El paquete *move_base* era el paquete principal del paquete de navegación, y era el que integraba toda la arquitectura del stack. Este paquete proporcionaba una implementación de una acción que, dado un objetivo en el mundo, el robot intentaría alcanzarlo actuando como una base móvil. El nodo *move_base* enlaza un planificador global y uno local para realizar la tarea de navegación global. En la siguiente figura se puede ver la arquitectura antigua que tenía stack de navegación en ROS1, y en que paquetes se han divido el stack de navegación en ROS2:

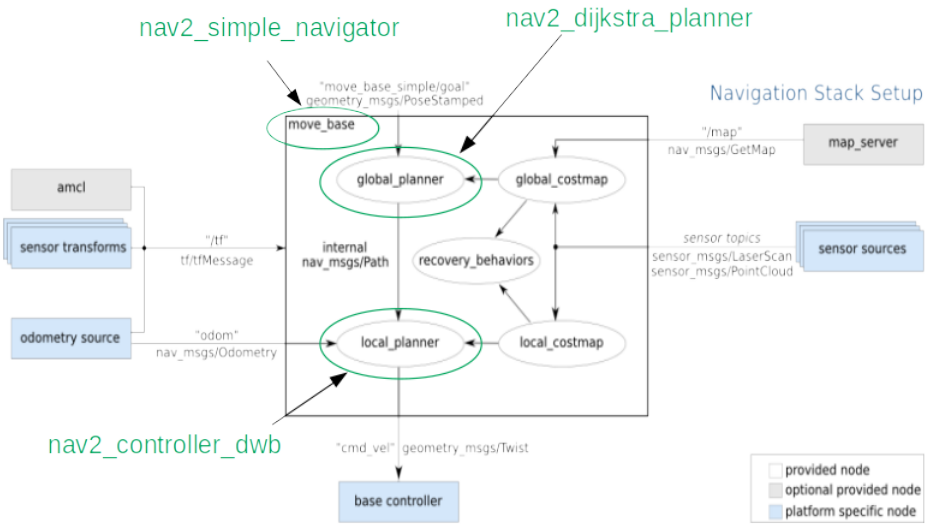

Figura 2.5: Antigua arquitectura de move_base con los cambios en ROS2.

El *nav2_bt_navigator* sustituye a *move_base* en el nivel superior, con una interfaz de acción para completar una tarea de navegación con un modelo de acción basado en un árbol. *move_base* se ha dividido en múltiples componentes y, en lugar de una única máquina de estado monolítica, *nav2* hace uso de servidores de acción y de la comunicación de baja latencia de ROS 2 para separar las tareas. Se utilizan árboles de comportamiento para hacer posible tener máquinas de estado más complejas y añadir comportamientos como servidores de acción adicionales. Estos árboles de comportamiento tienen formato XML y son altamente configurables. El propio paquete incluye ejemplos de partida que se pueden utilizar.

Los servidores de planificación, comportamientos y controladores son también servidores de acción que nav2_bt_navigator puede llamar para calcular cualquier datos que este necesite. Los 3 servidores pueden alojar muchos plugins de muchos algoritmos cada uno y ser llamados individualmente desde el árbol de comportamiento del navegador para realizar comportamientos específicos. Los plugins proporcionados por defecto han sido portados de ROS 1, a saber: DWB, NavFn, y comportamientos primitivos como girar y despejar mapas de costes. También se ha añadido un nuevo comportamiento para esperar una duración fija siempre que se necesite. Estos servidores son llamados desde *nav2_bt_navigator* a través de sus propios servidores de acción para calcular un resultado o completar una tarea. El estado es mantenido por el árbol de comportamiento del *nav2_bt_navigator* en todo momento.

2.5. Gazebo: simulador de robótica

Gazebo es uno de los simuladores de robótica de código abierto más antiguos y cuenta con una gran base de usuarios y proyectos activos, ya que es el principal simulador de la comunidad de ROS. En lugar de desarrollar todo desde de cero, Gazebo se construye sobre motores de física y renderización ya existentes. Por defecto, utiliza utiliza el motor de física ODE, pero otros como DART[31] e incluso Bullet son también son compatibles. Para el renderizado, hace uso de OGRE1 que, lamentablemente, tiene capacidades de renderizado limitadas capacidades de renderizado.

De este simulador también se ha realizado una versión completa, dejando la versión clásica del simulador en pos de una nueva versión que implementa más novedades y soluciona algunas limitaciones. Debido a las limitaciones y a la arquitectura obsoleta que ya venía arrastrando Gazebo Classic, en favor de Ignition Gazebo es decir, la próxima generación de Gazebo. Aunque está en su fase inicial de desarrollo, Ignition Gazebo es compatible con el motor de física DART y soporte para Bullet. Además de OGRE 1, el renderizado PBR está habilitado utilizando OGRE 2, y también hay un soporte parcial para el trazado de rayos con OptiX7. Tanto los motores de física como los de renderizado pueden cargarse durante el tiempo de ejecución gracias a la arquitectura basada en plugins utilizada en Gazebo Classic, que se heredado en Gazebo Ignition.

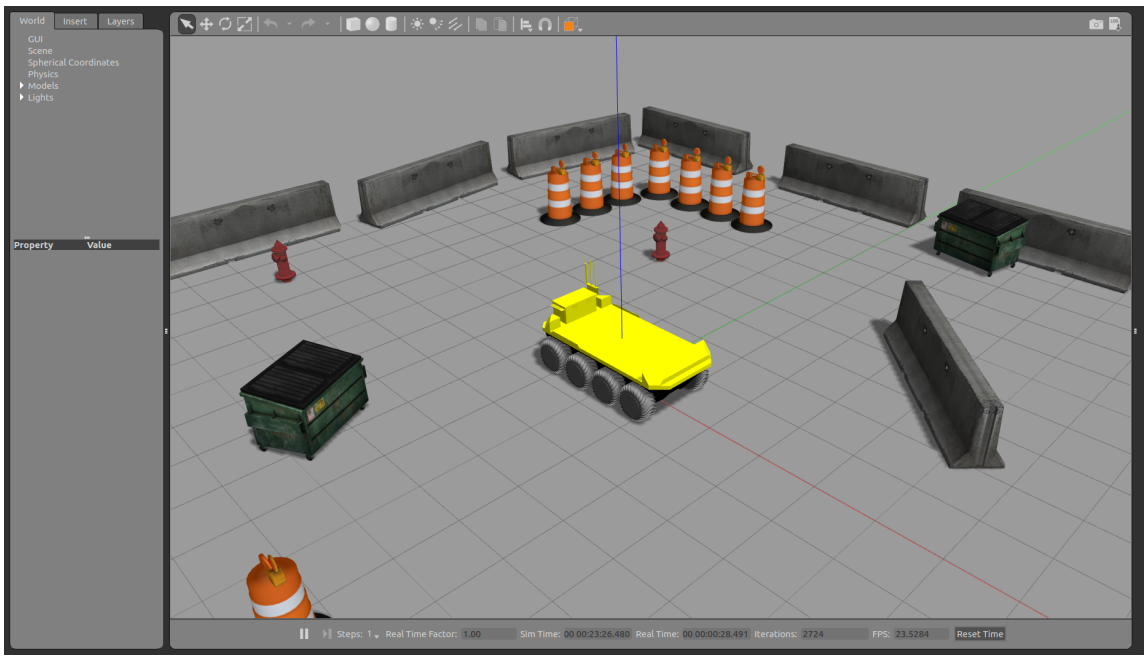

Figura 2.6: Simulación corriendo en Gazebo Classic.

3. ARQUITECTURA PROPUESTA

Este capítulo de la memoria se va a dividir en apartados, los cuales cubren los elementos que se han diseñado y desarrollado para solventar los objetivos propuestas en este proyecto.

3.1. Arquitectura de ROS

En esta sección se va a desarrollar como se han portado las funcionalidades del anterior proyecto, que utilizaba ROS1, a la nueva versión de ROS2. Se va centrar exclusivamente en las adaptaciones de ROS, dejando de lado la simulación Gazebo Ignition para las secciones posteriores.

El stack que ser quería utilizar se divide en tres partes claramente diferencias que se explicarán a continuación. A modo de pequeño inciso, se quiere matizar que la mayoría de cambios notables se ha producido en los scripts de lanzamiento, ya que se han tenido que rehacer por completo. A excepción de algunos paquetes, la mayoría de interfaces de ROS se ha mantenido intacta de ROS1 a ROS2, por lo que solo se han tenido que hacer cambios mínimos en las configuraciones que se tenían del anterior proyecto.

3.1.1. Utilizando Gazebo Ignition y ROS

Anteriormente, se podía integrar Gazebo dentro de los archivos de lanzamiento de ROS, y este se conectaba al master de ROS para poder interactuar con el resto de elementos de la arquitectura. Eso actualmente ya no es posible ya que , como se mencionó anteriormente, en ROS2 se eliminó la figura del master.

Una distinción que hay que hacer al respecto a diferencia de ROS1, es que ambas plataformas integran su propia capa de comunicación y mensajes, que funcionan de forma muy similar a lo ya conocido en ROS1. Pero son incompatibles directamente. En ROS2 es necesario utilizar un paquete se llama ros_ign_bridge. Este paquete, como su propio nombre indica, hace de puente entres las capas de comunicación de ROS y Gazebo Ignition. permite no solo la comunicación como tal, si que provee a ROS de una herramienta para convertir el formato de los mensaje de Ignitión, al formato de mensajes estándar de ROS.

Lo que técnicamente hace este paquete es recrear la estructura de topics y mensajes que se tiene en Ignition, pero en el sistema de comunicaciones de ROS. De esta forma se pueden utilizar los topics y temas e Ignition de forma nativa, igual que si fueran topics y mensajes de ROS. Pero para que esto ocurra es necesario indicar que topics se quieren utilizar, que tipo de mensajes utilizan y realizar otra configuraciones para que ROS no note las diferencias y funcione sin problemas.

Con eston en mente se han importado los siguiente topics:

- **/clock:** Necesario si queremos sincronizar el tiempo de simulación de Ignition con ROS, de forma que actúen de forma sincronizada y poder tener unos tiempos de respuesta más reales.
- **/cmd_vel:** El topic para los comandos de velocidad que se ha integrado con los controladores de Ignitión, se pasa directamente para que ROS pueda publicar con el stack de navegación las ordenes de velocidad que necesite.

- **/odometry:** La odometría proporcionada por el simulador, que será la fuente de información que se utilice en el paquete de robot_localization.
- **/laserscan:** El topic con las medidas del LiDAR, para que se puedan utilizar con el nodo de amcl para la localización del vehículo, y para la construcciones del mapa global y los mapas de coste.
- **/xxxx_camera:** Las imágenes captadas por las cámaras en tiempo real, que no son utilizadas en este proyecto, salvo para mostrarlas en Rviz. También se ofrece el topic CameraInfo, que ofrece toda la información sobre la configuración y detalles técnicos de las cámaras. En este sentido se pasan tanto las imágenes de la cámara de color, como la de la cámara de profundidad.
- **/joint_state:** Ya que el modelo URDF y SDF del robot se carga únicamente en Ignition, este ofrece también el estado de las articulaciones en tiempo real según la simulación.

Además, también se realizan algunas transformaciones estáticas entre elementos que solo se encuentran en Ignition, ya que al pasarle a ROS topics con frames que no ha visto, no puede añadirlos a los modelos de elementos que ya tiene el creados. Por lo que se añaden de forma que se puedan asignar los elementos cargados en Ignition a posiciones equivalentes y presentes en la estructura de ROS.

En los scripts de lanzamiento se han configurado tanto el LiDAR como la cámara RGBD, con sus frecuencias de funcionamiento normal (sin los modos Slow o Fast que veremos a continuación). Debido a los cambios en ROS2, no es necesario configurar ningún tipo de reconfiguración dinámica para cambiar parámetros como se hacía en ROS1. Por lo que para poder pasar de una configuración de los sensores a otra, basta con utilizar funciones de callback que reinicialicen esos parámetros, y el resto del sistema se adaptará al cambio.

Uno de los cambios notables respecto de Gazebo Classic es la renovación completa de los plugins de Gazebo para ROS que se tenían anteriormente. Aunque en este caso no nos influye demasiado para el control de robots, si es cierto que para las cámaras basadas en RealSense no hay ningún plugin todavía que pueda simular sus características, tal y como se podía hacer en Gazebo Classic. Aun así, Gazebo nos da la posibilidad de poder dotar a las cámaras de la habilidad de detectar la profundidad de forma nativa, y publicar los datos en un topic de Ignition. Se pierden algunas de las funcionalidad que brindaban los sensores en versiones anteriores, pero para los objetivos de este proyecto sirven perfectamente.

3.1.2. Core Stack

Los cambios en la arquitectura más básica y fundamental de ROS no han sido muy grandes, ya que en la renovación de ROS2 se ha buscado actualizar las funcionalidades pero manteniendo las interfaces se llevan utilizando años. Es por eso que no ha habido que realizar cambios muy notables en el stack base del robot.

Los paquetes básicos como el joint_state_publisher o el robot_state_publisher, para publicar las articulaciones del vehículo basándose en el modelo URDF, y para publicar las transformaciones entre articulaciones utilizando tf2, no han sufrido mayores cambios de interfaz. Se han podido implementar prácticamente sin cambios notables respecto de sus versiones anteriores.

La configuración de Rviz se ha mantenido prácticamente intacta, ya que la mayoría de cambios no e pueden apreciar en la GUI, y se pueden configurar fácilmente con el ratón. Los cambios

en su configuración en el script de lanzamiento han sido mínimos, y fácilmente superables. Un ejemplo de la GUI de Rviz funcioanndo se puede ver a continuación:

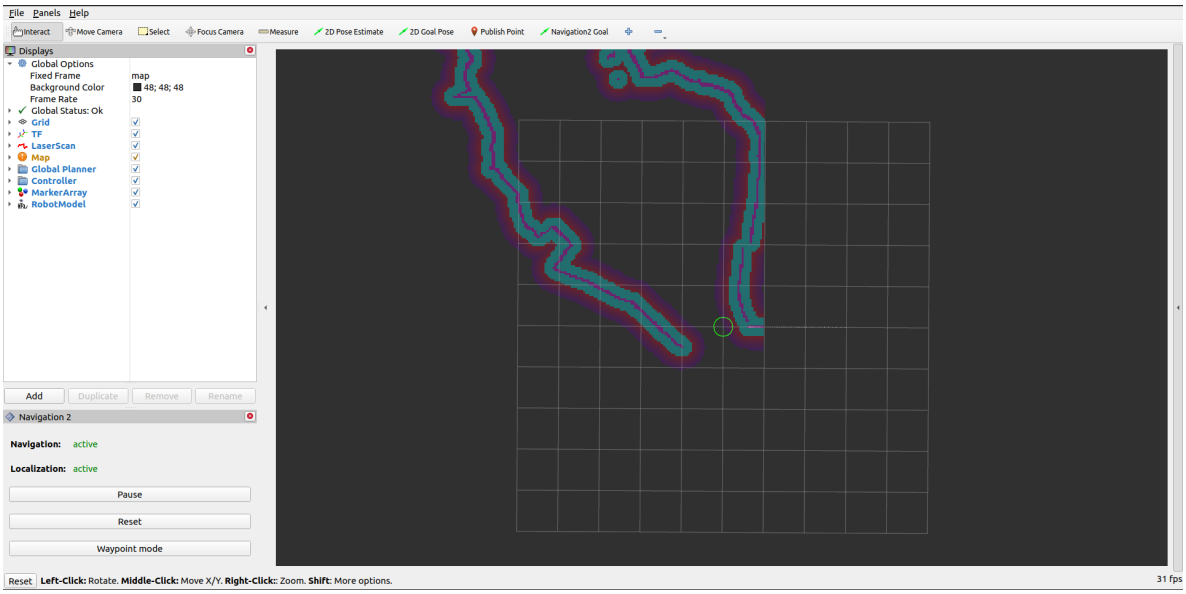

Figura 3.1: Ventana de Rviz con el stack de navegación funcionando funcionando .

Se ha añadido el paquete robot_localization para añadir un filtro EKF y fusionar los datos de odometría junto con los datos que nos ofrece la IMU. Aunque en la simulación no de dispone de IMU como tal, se ha configurado y el paquete fusionará los datos que le lleguen cuando le lleguen, por lo que puede funcionar aunque no obtenga datos de la IMU. En este caso lo que se realiza es una etapa previa a la hora de calcular la localización del robot, que permitirá suavizar su movimiento basado puramente en la odometría y que el vehículo tenga una localización más suave, sin saltos abruptos ni cambios inesperados. Este paquete nos ofrecerá un posicionamiento en el mapa local del robot, permitiendo ajustes en los mensajes de salida que podemos personalizar a nuestra conveniencia. Se pueden obtener estimaciones bastante precisas de las velocidades y aceleraciones que ocurren en cada eje Además, este paquete ofrece la posibilidad de calcular la transformación de coordenadas entre el frame base_link y frame odom, de forma que no tengamos que proporcionárselo de otro modo, como forzando una publicación asíncrona de los mismos.

3.1.3. Navigation Stack

Como se ha comentado anteriormente, el stack de navegación sufrió algunos cambios en la transición de ROS1 a ROS2, por lo que ha sido necesario realizar una readaptación profunda de ciertos aspectos del paquete. El cambio principal ha sido que el nodo bt_navigator incluye ahora la posibilidad de carga plugins de navegación para cargar comportamientos y algoritmos de navegación. Se pueden definir comportamientos que el vehículo puede realizar, y hacer que el nodo cambie y elija entre ellos para conseguir un comportamiento de navegación más complejo que antes. Por defecto se han cargado la mayoría de plugins, ya que los comportamientos de recuperación y los movimientos primitivos ahora están mucho mas subdivididos que antes. La lista utilizada sería la siguiente:

`nav2_compute_path_to_pose_action_bt_node`

```
nav2_follow_path_action_bt_node
nav2_back_up_action_bt_node
nav2_spin_action_bt_node
nav2_wait_action_bt_node
nav2_clear_costmap_service_bt_node
nav2_is_stuck_condition_bt_node
nav2_goal_reached_condition_bt_node
nav2_goal_updated_condition_bt_node
nav2_initial_pose_received_condition_bt_node
nav2_reinitialize_global_localization_service_bt_node
nav2_rate_controller_bt_node
nav2_distance_controller_bt_node
nav2_speed_controller_bt_node
nav2_truncate_path_action_bt_node
nav2_goal_updater_node_bt_node
nav2_recovery_node_bt_node
nav2_pipeline_sequence_bt_node
nav2_round_robin_node_bt_node
nav2_transform_available_condition_bt_node
nav2_time_expired_condition_bt_node
nav2_distance_traveled_condition_bt_node
```

Todos estos plugins conforman la base de comportamientos que el nodo de navegación utilizará en conjunto con el árbol de comportamiento definido por el archivo:

navigate_w_replanning_and_recovery.xml

Este archivo integra una navegación dinámica que permite replaneamientos sobre la marcha, según configuración, y habilita al controlador la posibilidad de utilizar comportamientos de recuperación si en algún momento se queda bloqueado.

Para el controller_server se ha mantenido su configuración, utilizando los mismo plugins para sus diferentes apartados: el plugin SimpleProgressChecker para el comprobar el progreso del vehículo, el plugin SimpleGoalChecker para comprobar si se ha llegado a la meta o no, y el planificador DWBLocalPlanner para realizar la planificación de rutas entre el origen y el destino.

La configuración de los mapas de coste local y global es idéntica, ya que no se han producido cambios en ninguno de las configuraciones entre ROS1 y ROS2. Los nodos que se ocupan de administrar el mapa global, como por ejemplo map_server, siguen teniendo también las mismas configuraciones.

El nodo amcl se utiliza para obtener una estimación probabilística de la localización del vehículo en el mapa utilizando las medidas del LiDAR y/o de la cámara RGBD, y un algoritmo para el posicionamiento mediante Monte Carlo. Esto nos permite localizar al robot en el mapa global, con una mayor fiabilidad que utilizar otros métodos como extrapolar la posición directamente de la odometría. No han sido necesarios cambios significativos en la configuración de este paquete ya que se han mantenido las interfaces de ROS1.

A continuación se expone un pequeño diagrama que explica el cometido del paquete de amcl para las transformaciones de coordenadas entre los distintos frames del proyecto.

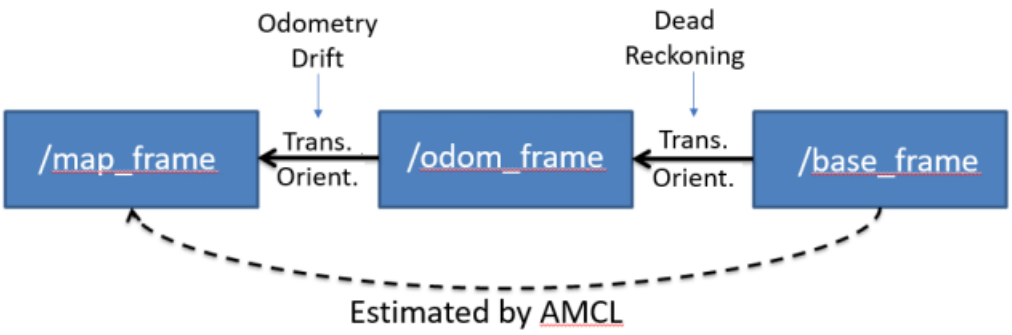

Figura 3.2: Diagrama del funcionamiento de amcl en comparación con robot_localization.

Aunque el nodo de amcl ofrezca una transformación directa entre el base_frame del robot y el map_frame del mundo, en este caso proveen ambas alternativas haciendo uso del paquete robot_localization que se mencionó anteriormente. Y la odometria fusionada por robot_localization es utilizada por amcl como otra fuente de información más. En este caso, la transformación más importante que realiza amcl, es la que se da entre el map_frame y el odom_frame, ya que es el único nodo de la arquitectura presentada que puede realizarlo.

3.1.4. Metacontrolador Rosont

El metacontrolador Rosont se ha creado en consonancia con los objetivos del proyecto y buscando ofrecer una solución que pueda ser reutilizable para solucionar diferentes objetivos y no solamente los que nos ocupan en este proyecto. En este sentido se ha intentado crear algo que resulte novedoso, pero a la vez basado en los estándares y soluciones que han probado funcionar en la industria.

El meta-paquete de Rosont se subdivide a su vez en tres paquetes diferentes que implementan diferentes elementos. Se ha seguido las convecciones heredadas de ROS1 en cuanto a separar los elementos que puedan utilizarse de forma independiente, por lo tanto, los paquetes son:

- **/rosont_commons:** En este paquete se guardan las ontologías que se utilizarán como base (ssn, sosa y rosont), además de otros archivos de configuración y ayuda del metacontrolador.
- **/rosont_msgs:** Este paquete contiene los mensajes que se vana utilizar con el metacontrolador y para comunicarse con nodos externos a este. Incluye un mensaje sobre el estado actual de la plataforma según el razonador y una lista de valores opcional que se pueden parsear. También incluye una acción para cambiar el modo de trabajo de los sensores según sea necesario y las condiciones así lo permitan.
- **/rosont_reasoner:** Este es el paquete principal de Rosont y en el se encuentran el metacontrolador, el razonar y los scripts de lanzamiento para cada uno de ellos.

Para poder trabajar con ontologias OWL de forma comoda en Python, se ha utilizado la librería llamada Owlready2[32]. Owlready2 permite realizar una programación orientada a la ontologías, de forma que se pueden cargar ontologías OWL 2.0 como objetos de Python, modificarlas,

guardarlas y realizar razonamientos mediante múltiples razonadores como HermiT (incluido) o Pellet. Debido a la naturaleza de las ontologías SSN y SOSA es necesario utilizar el razonador de Pellet, ya que HermiT no funciona correctamente en el editor Protégé al ser ontologías tan grandes.

Siguiendo las convenciones de la industria, para el metacontrolador se han creado dos clases: una clase principal llamada RosontMetacontrol que aglutina todas la funciones y variables del metacontrolador como tal; y una clase secundaria para el razonador con todas las funcionalidades que este necesita, llamada Reasoner.

Estas dos clases serán las encargadas de interactuar con la ontologías de Rosont, realizar las inferencias que sean necesarias y aplicar los resultados obtenidos de esas inferencias. Para ello se utiliza un bucle que realiza ese proceso una y otra vez. Tras una primera inicialización de las variables y de la ontología, se sigue el siguiente procedimiento:

1. Se ejecuta el razonamiento de las ontologías para obtener las inferencias.
2. Se evalúan los resultados de las inferencias, comprobando cuales son las salidas activas.
3. Se adaptan los sensores a los modos que se hayan obtenido como resultado, reconfigurando los parámetros

Los cambios en los Resultados de las Observaciones se introducen en la ontología mediante callbacks con los datos de los sensores. Se introducen los valores como propiedades hasSimpleResult de los Resultados de la ontología, y ésta ya se encargará mediante inferencias de elegir las salidas correctas. Se profundizará en siguientes apartados como se produce este razonamiento.

Para los sensores de luz y voltaje de la batería se obtienen los valores de intensidad y voltaje, después se transforma su rango de forma proporcional para obtener un intervalo de [0.00, 1.00]. Para obtener los datos de distancia es algo diferente. Como el LiDAR y la cámara RGBD ofrecen multitud de medidas con una frecuencia altísima, se ha decidido que los más interesante es trabajar con las distancias mínimas que se obtengan de ambos sensores. Es decir, si se detecta un objeto muy cercano a menos de 0.5 metros de la plataforma móvil, lo más relevante del entorno inmediato de la plataforma es ese objeto. Por tanto, se decidió que la mejor forma de darle datos a los ontología sin sobrecargarla, era utilizando la menor medida que se reciba de la cámara RGBD y del LiDAR, siempre y cuando se recibiese un numero determinado de veces seguidas para evitar falsos positivos. Y dada que la lógica de los modos está pensada para que la plataforma responda a los cambios en su entorno más inmediato, tiene sentido que sea así.

Si ocurre cualquier error inesperado, como que no haya salidas disponibles, el nodo lanzará alertas para avisar al usuario de que ha ocurrido cualquier problema con las ontologias o con el proceso de funcionamiento.

Para el cambio en la configuación del robot se ha decidido utilizar la reconfiguración dinámica de parámetros, específicamente en este caso la opción de reconfigurar los topics y las configuraciones de los sensores. De esta forma, utilizando propias funcionalidades de *rclpy* y los cambios de ROS2 respecto de ROS1 en cuanto a gestión de parámetros, podemos cambiar entre topics a voluntad y cambiar las velocidades de procesado según se necesite.

3.2. Ontología

En este capitulo se va a presentar la ontología que se ha diseñado para utilizarse finalmente en el proyecto. Como se ha visto en anteriores apartados, se han realizado varios intentos en cuanto a ontologías orientadas a sensores, cada una con unas finalidades y características diferentes.

Tras revisar detenidamente todas las alternativas, se llegó a la conclusión de que diseñar una ontología completa enfocada a sensores desde cero, dada la variedad de intentos que se habían realizado, no supondría ningún tipo de ventaja sobre los anteriores diseños. Sin embargo, se podría utilizar la ontología más idónea como base y realizar las pequeñas modificaciones que fueran necesarias para adaptarla a las necesidades del proyecto. De esta forma se podría partir con una ontología lo suficientemente robusta para que sirviera de base, conocer de antemano sus ventajas y desventajas, y centrarse en la adaptación de la ontología al nuevo proyecto.

Por estos motivos, se ha elegido un conjunto de dos ontologías como base que están estrechamente relacionadas: la ontología llamada Semantic Sensor Network (SSN)[33] y la ontología llamada Sensor, Observation, Sample, and Actuator (SOSA)[33]. Ambas ontologías desarrolladas por el World Wide Web Consortium (W3C), siendo SOSA además producto de una colaboración con el Open Geospatial Consortium (OGC).

Para la creación y la edición de las ontologías en OWL, se ha utilizado Protegé, que es un software de manipulación de ontologías muy completo y repleto de funcionalidades.

3.2.1. Las ontologías de SSN y SOSA

La ontología Semantic Sensor Network (SSN) es una ontología para describir los sensores y sus observaciones, los procedimientos implicados, las características estudiadas de interés, las muestras utilizadas para ello y las propiedades observadas, así como los actuadores. La ontología cubre gran parte del estándar SensorML, omitiendo algunas características como las calibraciones, la descripción de procesos y los tipos de datos, cada uno de ellos no específico de cada tipo de sensor como tal.

SSN sigue una arquitectura de modularización horizontal y vertical al incluir una ontología central ligera (light) pero autocontenida llamada SOSA (Sensor, Observación, Muestra y Actuador) para sus clases y propiedades elementales. Con su diferente alcance y diferentes grados de axiomatización, SSN y SOSA son capaces de soportar una amplia gama de aplicaciones y casos de uso diferentes sin necesidad de grandes cambios.

La arquitectura de estas dos ontologías se pueden ver en las siguiente figuras:

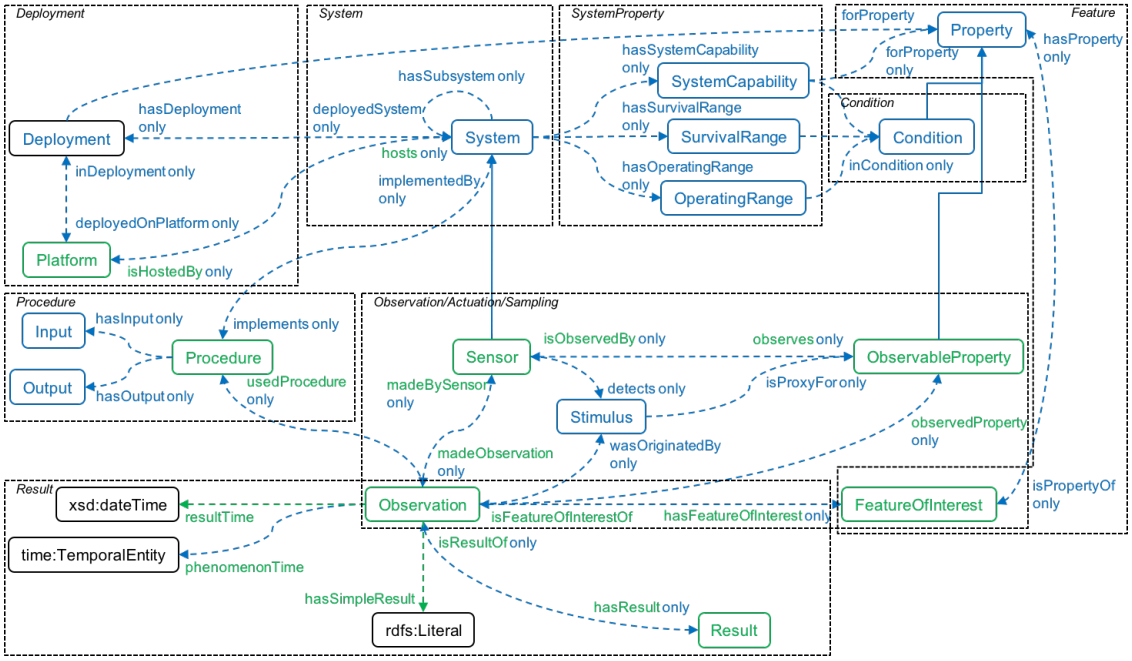

Figura 3.3: Arquitectura base de la ontología SSN.

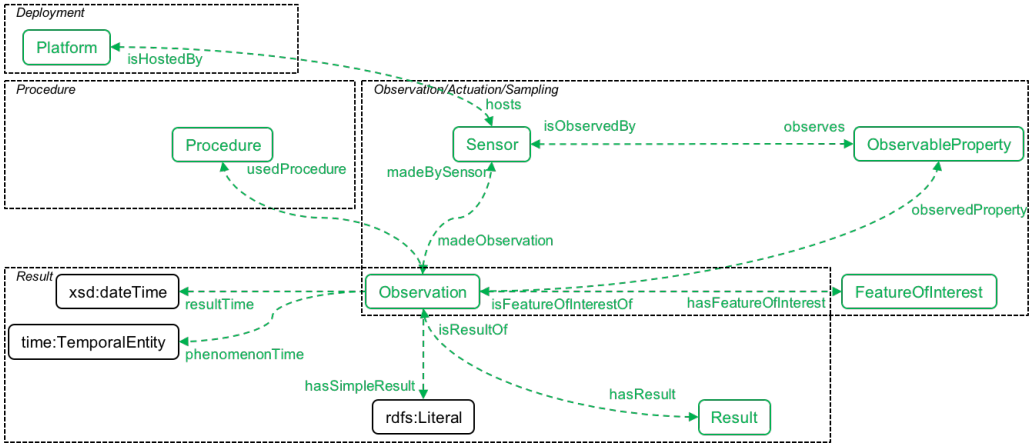

Figura 3.4: Arquitectura base de la ontología SOSA.

Como se puede apreciar, se puede ver claramente como SOSA es una pequeña parte de la arquitectura de SSN, por lo que aunque es una arquitectura más simple, le faltan algunas de la funcionalidades de SSN. Además, en este tipo de casos en los que se van a realizar trabajos futuros sobre este trabajo, es preferible utilizar una ontología más amplia que permita implementar más elementos en el proyecto.

Ambas ontologías están compuestas por una gran variedad de clases y propiedades de objeto ya predefinidas. No se van a explicar la totalidad de estos elementos, ya que serían demasiados para el alcance del proyecto y tampoco se utilizan todo ellos. Por ello, se presentan a continuación las principales clases que se van a utilizar durante el proyecto, obtenidas directamente de la documentación:

- **Observation:** Acto de llevar a cabo un Procedimiento (de Observación) para estimar o calcular un valor de una propiedad de un FeatureOfInterest. Se vincula a un sensor para describir qué hizo la observación y cómo; se vincula a un ObservableProperty para describir de qué se estima el resultado, y a un FeatureOfInterest para detallar a qué se asoció esa propiedad.
- **Output:** Cualquier información que se comunique desde un Procedimiento.
- **Property:** Cualidad de una entidad. Aspecto de una entidad que es intrínseco y no puede existir sin la entidad.
- **Procedure:** Un flujo de trabajo, protocolo, plan, algoritmo o método computacional que especifica cómo realizar una observación, crear una muestra o realizar un cambio en el estado del mundo (a través de un actuador). Un procedimiento es reutilizable y puede participar en muchas observaciones, muestreos o actuaciones. Explica los pasos que hay que seguir para obtener resultados reproducibles.
- **Result:** Dispositivo, agente (incluidos los seres humanos) o software (simulación) que participa en un procedimiento o lo ejecuta. Los sensores responden a un estímulo, por ejemplo, un cambio en el entorno, o a datos de entrada compuestos por los resultados de observaciones anteriores, y generan un resultado. Los sensores pueden ser alojados por Plataformas.

En el proyecto se utilizan todas estas clases. Las propiedades no se van a desarrollar dada la cantidad que son, y además son bastante explicativas con el nombre que ya tienen actualmente.

3.2.2. La ontología de Rosont

Basándose en las dos ontologías anteriores como base, se ha creado una ontología adicional que es la que se utiliza en el metacontrolador con los paquetes de ROS. Está ontología ha sido creada con la idea de aprovechar la estructura que ofrecen SSN y SOSA. No se ha realizado ninguna modificación sobre las entidades que componen ambas ontologías base, pero sí que se han creado elementos adicionales, además de los individuos.

Los individuos creados responden a las necesidades del proyecto. Se han añadido los elementos, las entradas y salidas necesarias con la información de la que se dispone a la hora de ejecutar el metacontrolador y el resto de la arquitectura de ROS. Se contemplan los siguientes individuos:

Tabla 3.1: Tabla con los individuos creados para Rosont

Individuals	***Class***	***Data_properties***
mea_distance	Result	hasSimpleResult
mea_light	Result	hasSimpleResult
mea_voltage	Result	hasSimpleResult
out_lidar_normal	Output	hasStatus
out_lidar_slow	Output	hasStatus
out_rgbd_fast	Output	hasStatus
out_rgbd_normal	Output	hasStatus
sen_battery	Sensor	
sen_lidar	Sensor	
sen_light	Sensor	
sen_rgbd	Sensor	
st_fast_mode	Procedure	hasStatus
st_normal_mode	Procedure	hasStatus
st_slow_mode	Procedure	hasStatus
val_distance_high	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_distance_normal	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_distance_low	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_light_high	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_light_normal	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_light_low	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_voltage_high	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_voltage_normal	Observation	hasStatus; hasUpperLimit; hasLowerLimit
val_voltage_low	Observation	hasStatus; hasUpperLimit; hasLowerLimit

Se han creado individuos que cubren la mayoría de las funcionalidades que ofrece la actual implementación. Se podría concretar y definir más el modelo de conocimiento, definiendo la plataforma, actuadores y demás elementos, pero como actualmente no uso para ese conocimiento se ha decidido dejarlos sin declarar.

A los Resultados se les ha asignado la propiedad de tener un resultado numérico concreto en el intervalo [0.00, 1.00], de forma que las diferentes Observaciones puedan utilizarlos en los procesos de la ontología.

Las Observaciones a su vez tienen un límite superior (hasUpperLimit) e inferior (hasLowerLimit) que permite clasificar su Resultado asignado dentro o fuera de esos límites. Si está dentro, se cambiará su estado (hasStatus) a True, pero si está fuera de los límites, tendrá en cambio el valor de False.

Los Procedimientos utilizarán las Observaciones para determinar si deben activar las Salidas correspondientes o no, cambiado su estado (hasStatus) según se cumplan las condiciones o no.

Para este proyecto se ha definido la siguiente configuración como base para determinar las salidas de la ontología:

Tabla 3.2: Configuración de los Procedimientos según las Observaciones

	LIDAR			CÁMARA RGBD		
Modes	Slow	Normal	Fast	Slow	Normal	Fast
Low Distance		X				X
Normal Distance		X			X	
High Distance	X					
Low Voltage	X					
Normal Voltage		X			X	
High Voltage		X				X
Low Light	X					
Normal Light		X			X	
High Light		X				X

Se ha seguido la siguiente lógica:

- Si el vehículo tiene unas condiciones degradadas, o se encuentra en un entorno suficientemente seguro se utilizará el modo Slow, cuyo único sensor es el LiDAR.
- Si el vehículo tiene unas condiciones normales, se utilizará los modos Normales de ambos sensores, salvo que en algún caso se requiera el modo Fast de la cámara RGBD.
- Si se tienen una condiciones más que óptimas o el robot se encuentra en una situación peligrosa, se utilizará el modo Fast de la cámara RGBD para aprovechar sus ventajas.

Se puede apreciar que habrá momentos en los que el modo Normal y el modo Fast estén trabajando a la vez en diferentes sensores. Mientras que siempre que se trabaje en modo Slow, solo se trabajará con el LiDAR, apagando la cámara RGBD. Este tipo de restricciones se habrán de programar en la ontología mediante reglas SWRL, como veremos a continuación.

3.2.3. Reglas SWRL de Rosont

Una vez que se tiene la estructura base de la ontología, con las propiedades y los individuos necesarios, se hace necesario establecer unas reglas de funcionamiento que permitan poder realizar un razonamiento correcto y obtener las salidas deseadas de nuestra ontología. Para ello se ha utilizado un conjunto de reglas SWRL, que nos permiten actualizar las propiedades de nuestros individuos siempre y cuando se cumplan ciertas condiciones, tal y como vimos anteriormente.

Las reglas se pueden dividir en 3 grupos, en función de a quien afecten: Observaciones, Procedimientos y Salidas. Las reglas que afectan a las Observaciones son las siguientes:

Regla 1

```
sosa:Observation(?obs) ^
rosont:hasLowerLimit(?obs, ?lolim) ^
sosa:hasResult(?obs, ?res) ^
sosa:hasSimpleResult(?res, ?val) ^
swrlb:lessThan(?val, ?lolim) -> rosont:hasStatus(?obs, false)
```

Regla 2
```
sosa:Observation(?obs) ^
rosont:hasUpperLimit(?obs, ?uplim) ^
sosa:hasResult(?obs, ?res) ^
sosa:hasSimpleResult(?res, ?val) ^
swrlb:greaterThanOrEqual(?val, ?uplim) -> rosont:hasStatus(?obs, false)
```

Regla 3
```
sosa:Observation(?obs) ^
rosont:hasUpperLimit(?obs, ?uplim) ^
rosont:hasLowerLimit(?obs, ?lolim) ^
sosa:hasResult(?obs, ?res) ^
sosa:hasSimpleResult(?res, ?val) ^
swrlb:greaterThanOrEqual(?val, ?lolim) ^
swrlb:lessThan(?val, ?uplim) -> rosont:hasStatus(?obs, true)
```

Las reglas que afectan a los Procedimientos son solo dos:

Regla 4
```
sosa:Procedure(?prc) ^
rosont:hasStatus(?prc, ?st) ^
swrlb:equal(?st, true) -> rosont:hasStatus(?prc, true)
```

Regla 5
```
sosa:Procedure(?prc) ^
rosont:hasStatus(?prc, ?st) ^
swrlb:equal(?st, false) -> rosont:hasStatus(?prc, false)
```

Las reglas que afectan a las Salidas son especificas de cada una hasta un total de 8, pero tienen una estructura similar a esta:

Regla 6
```
rosont:hasStatus(rosont:st_fast_mode, ?st) ^
swrlb:equal(?st, false) -> rosont:hasStatus(rosont:out_rgbd_fast, false)
```

Regla 7
```
rosont:hasStatus(rosont:st_fast_mode, ?st) ^
swrlb:equal(?st, true) -> rosont:hasStatus(rosont:out_rgbd_fast, true)
```

Regla 8
```
rosont:hasStatus(rosont:st_normal_mode, ?stm) ^
rosont:hasStatus(rosont:st_slow_mode, ?sst) ^
swrlb:equal(?stm, true) ^ swrlb:equal(?sts, false) ->
rosont:hasStatus(rosont:out_lidar_normal, true)
```

En el caso de las salidas debido que se tendrían que activar con el modo Fast o el Normal, se revisa también que no este activado el Slow mode, ya que cuando se activa este último es el único que puede estar activo. Con este sencillo set de reglas actualizamos las propiedades más importantes de nuestro árbol de conocimiento y generamos las salidas según el estado de los sensores.

3.3. Simulación con Gazebo Ignition

La simulación con Gazebo Ignition ha sido sorprendentemente fácil de realizar y ha resultado ser una transición muy fluida viniendo de utilizar Gazebo Classic. Los modelos tanto del robot como de la cueva venían en lso formatos adecuados para poder utilizarlos en Ignition sin modificación alguna. Solo se han añadido al mundo por defecto que se carga con Ignition y se han colocado en sus respectivas posiciones.

Ignition nos permite movernos por el entorno de la cueva y seguir al robot de forma automática. A continuación se puede ver una imagen de la simulación funcionando con todos los modelos cargados.

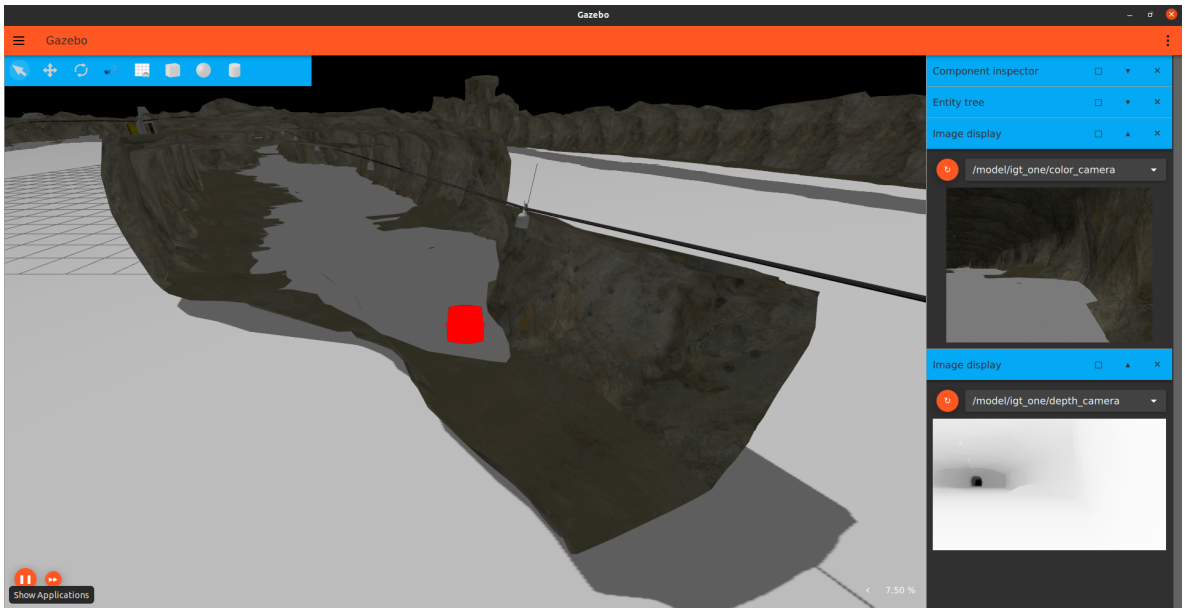

Figura 3.5: Ventana de Gazebo Ignition con la simulación funcionando .

En el panel de la izquierda se pueden ver las imágenes en tiempo real que captan las cámaras de color y profundidad. La mayoría de datos es mejor visualizarlos desde RVIZ, ya que permite mostrar todos los datos de interés, sin la necesidad de ver el espacio 3D de la cueva.

4. RESULTADOS EXPERIMENTALES

En esta sección vamos a proceder explicar la metodología que se ha seguido a la hora de testear las implementaciones de la plataforma y de la ontología de Rosont, y cómo se ha comprobado que todo funciona según los esperado. Posteriormente se presentaran algunos de los resultados de las pruebas y del funcionamiento de la plataforma y también de la ontología.

4.1. Metodología de experimentación

Dado lo complejo y único de este proyecto, es difícil realizar un testeo estandarizado que cubra una sería de puntos comunes con otros proyectos. Debido a la complejidad del proyecto se ha decidido dividir el testeo en dos partes separadas: testear la plataforma robótica en ROS, junto con la navegación y la simulación; y por otra parte testear el funcionamiento de la ontología de Rosont con una serie de test personalizados mediante ROS.

Aunque esta separación facilita las cosas, ya que no es necesario crear un framework de testeo personalizado que permita analizar toda la simulación en su conjunto, se siguen teniendo una serie de problemas:

1. **Falta de benchmarks para testeo en ROS**: Debido a la naturaleza abierta de ROS, salvo en implementaciones muy básicas y estandarizadas de ciertos tipos de robots (como el TurtleBot) no se disponen de ningún framework de testeo estandarizado. Además, debido a que la implementación del proyecto utiliza herramientas muy nuevas, como Gazebo Ignition o ROS2, no se dispone de prácticamente recursos para la realización de test de ningún tipo. Esto hace que no se pueda proporcionar medidas cuantitativas de la mejora en la navegación del robot, y haya que recurrir a aproximaciones más someras.

2. **Limitaciones en la simulación y funcionamiento de Gazebo**: aunque actualmente ha mejorado bastante, es cierto que trabajar con Gazebo y ROS fuera de sistemas Linux sigue suponiendo una gran quebradero de cabeza. Además, se tiene la dificultad de que los drivers de Nvidia o de AMD no suelen funcionar correctamente en sistemas Linux o hay que realizar configuraciones de kernel bastante inestables. Por todo esto y más, realizar simulaciones con Gazebo seguidas (y sobre todo con Ignition que es más inestable que Classic) es prácticamente imposible o muy costoso a nivel computacional.

3. **Falta de herramientas para ontologías**: Aunque en este proyecto se ha conseguido implementar la ontología de Rosont con éxito, bien es cierto que no se disponen de muchas herramientas para trabajar con ontologías. Y además, las pocas que hay como Protégé, son bastante incompatibles con otras implementaciones en frameworks diferentes.

4. **Limitaciones para el testeo de ontologías en ROS**: debido a que principalmente este itipo de implementaciones que involucran ontologías con ROS es bastante novedoso, no hay ningún tipo de paquete, variables o entorno preparados para poder trabajar con ellas. Por lo que la forma principal de testear el comportamiento de la ontología es a través de los mensajes que se envían los nodos de ROS y los mensajes de debug que se programen manualmente.

Por estas razones principalmente se han creado las pruebas y condiciones que se verán en las siguientes secciones a continuación.

4.2. Experimentación con la plataforma

La mayoría de la experimentación con ROS se ha realizado utilizando PlotJuggler[34], una herramienta creada por Davide Faconti que permite graficar los mensajes de ROS para ver su evolución en el tiempo, y que está muy estandarizada dentro de ROS.

A continuación se detallan las pruebas experimentales que se han realizado para comprobar que la plataforma funcionaba correctamente en ROS2, que las transformaciones de coordenadas se realizaban correctamente y que había ningún problema con la simulación de Ignition.

4.2.1. Testeo de la simulación en Gazebo Ignition

La simulación en Ignition es uno de los puntos cruciales de la experimentación con la plataforma, ya que representa uno de los cambios principales en este implementación respecto de las anteriores. Lo principal era conseguir que Ignition pudiera integrar los modelos de la plataforma robótica y que se pudieran simular los principales sensores de forma correcta.

Para ello se ha realizado un testeo progresivo de la simulación y se han ido arreglando pequeños desperfectos y bugs que se han ido encontrando durante el desarrollo del proyecto. A continuación se pueden ver algunas capturas del motor de Ignition funcionando con el modelo de la cueva y la plataforma robótica.

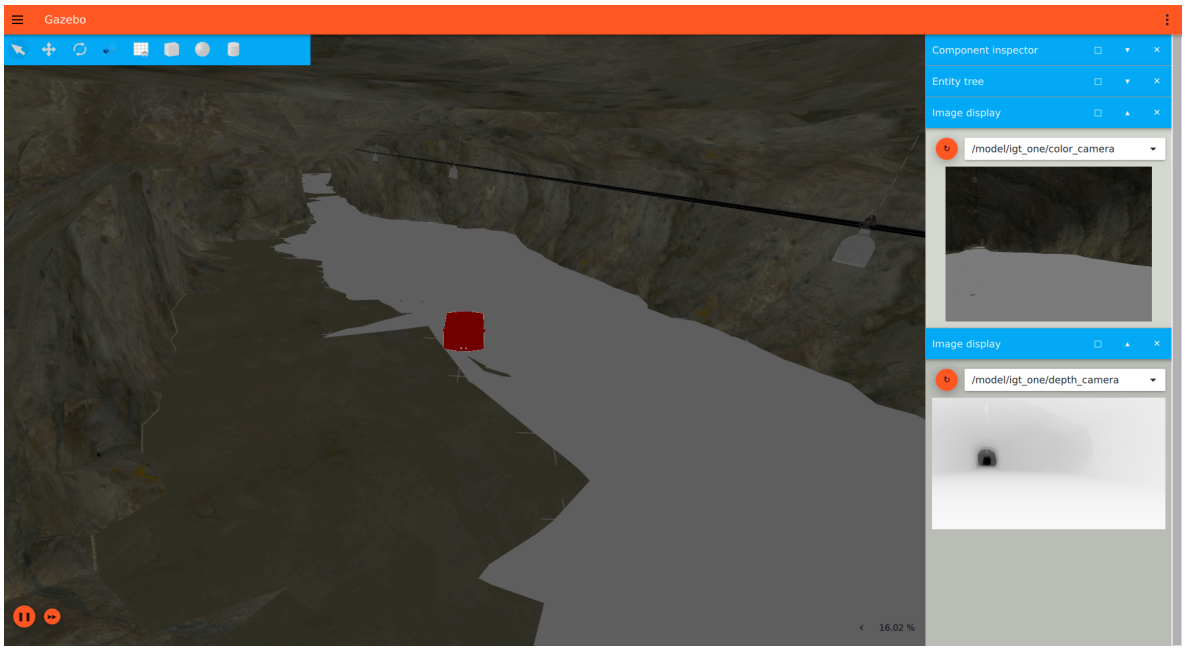

Figura 4.1: Simulación general de la cueva en Gazebo Ignition.

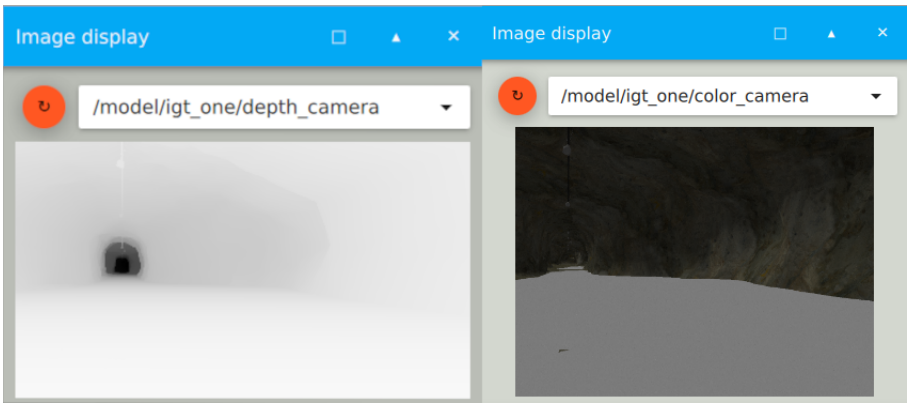

Figura 4.2: Plano detalle de las cámaras de color y de profundidad.

4.2.2. Testeo de la navegación en ROS2 con Nav2 y Rviz

Para el testeo de la navegación se va a comprobar mediante el uso de PlotJuggler qué tal se mueve la plataforma una vez que se le ordene llegar a un punto y si es capaz de mantener una navegación precisa y estable durante una mayoría de tiempo.

Para esto hay que tener en cuenta una cosa muy importante: el tiempo y la precisión que tengan la convergencia del nodo de AMCL es crucial para este punto. En este caso para el proyecto se ha decidido realizar una convergencia más lenta pero mucho más precisa. De esta manera el AMCL tardará más tiempo en converger inicialmente, pero se posicionará dentro del mapa mucho mejor según vaya pasando el tiempo. Esto es importante porque durante el movimiento, se podrá notar como la plataforma tarda un tiempo en localizarse correctamente dentro del mapa. Pasado un tiempo ira corrigiendo su posición hasta que las detecciones del LiDAR y/o de la cámara coincidan con el mapa se tiene de la cueva. Con esto se consigue que el robot tenga un posicionamiento inicial más pobre, pero conseguimos que una vez localizado pueda seguir el mapa que tiene guardado.

A continuación pueden verse algunas de las capturas de Rviz y de PlotJuggler cuando se navega por la cueva y se le manda un comando de *Navigation2 Goal*, para que vaya a un punto concreto de la cueva.

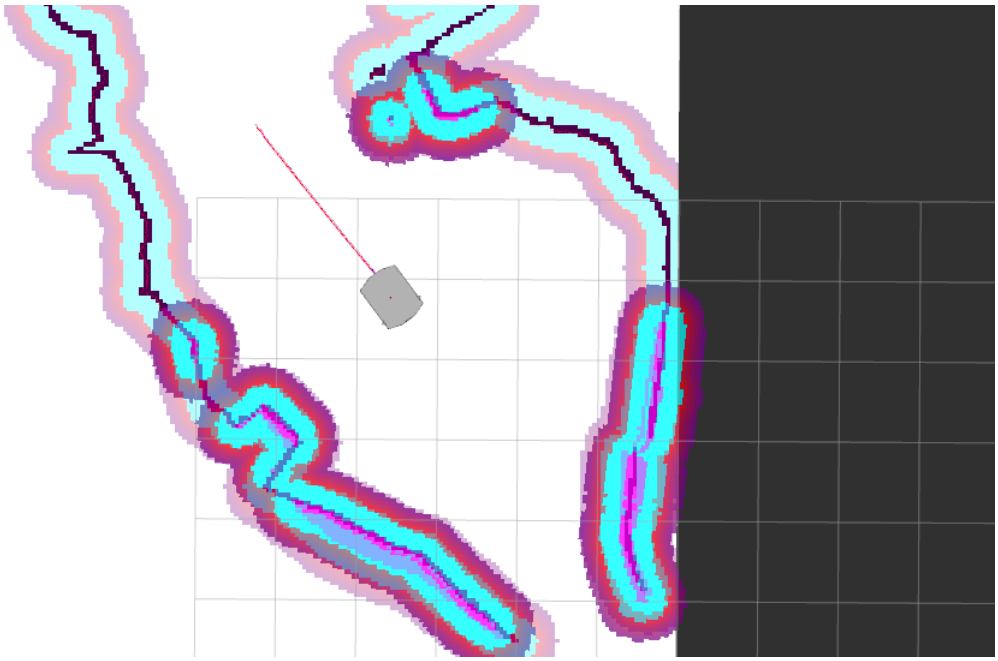

Figura 4.3: Ventana de Rviz durante la navegación dentro de la cueva.

En la anterior imagen puede verse como el nodo de AMCL ha convergido y el mapa de coste global coincide con las detecciones realizadas por la plataforma, por lo que consigue localizarse de forma muy precisa. Además, aunque el LiDAR tiene un rango mayor del mostrado en los mapas de coste, se reduce un poco la distancia máxima de las detecciones para mejorar la localización en el mapa. Esto se traduce en una navegación muy suave y fluida, como puede verse en la representación de la posición del robot durante esa misma ruta, que muestra la siguiente figura:

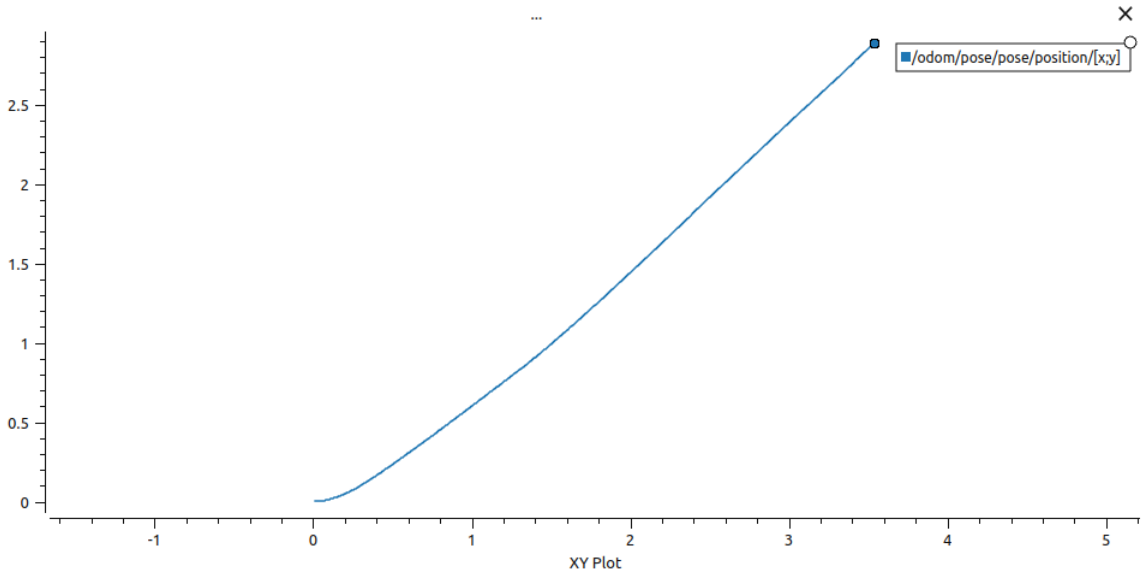

Figura 4.4: Representación de la posición XY en PlotJuggler en tiempo real.

Se puede ver como realiza una pequeña curva inicial para colocarse en la trayectoria recta hacía punto al que se le ha pedido que vaya, y se aprecia claramente como la posición se mantiene sin saltos, ni curvas y de forma muy suave.

Para finalizar esta parte también se ha comprobado que los topics correspondientes estaban correctamente programados, y que se podían ver las conexiones entre nodos que se esperan del stack de navegación de ROS2. Se puede ver el árbol de conexiones en la imagen a continuación.

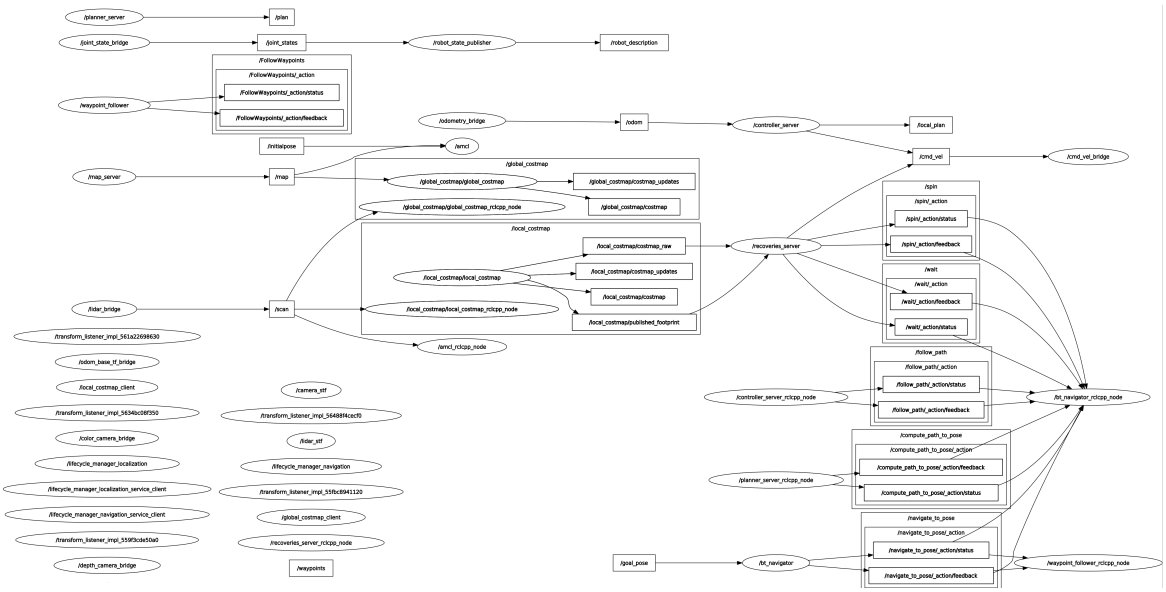

Figura 4.5: Árbol de topics y nodos una vez lanzado el stack de navegación.

Y por último, se ha comprobado que las transformaciones de coordenadas entre frames se realizan correctamente y que se obtienen los frames especificados en los modelos 3D de la plataforma.

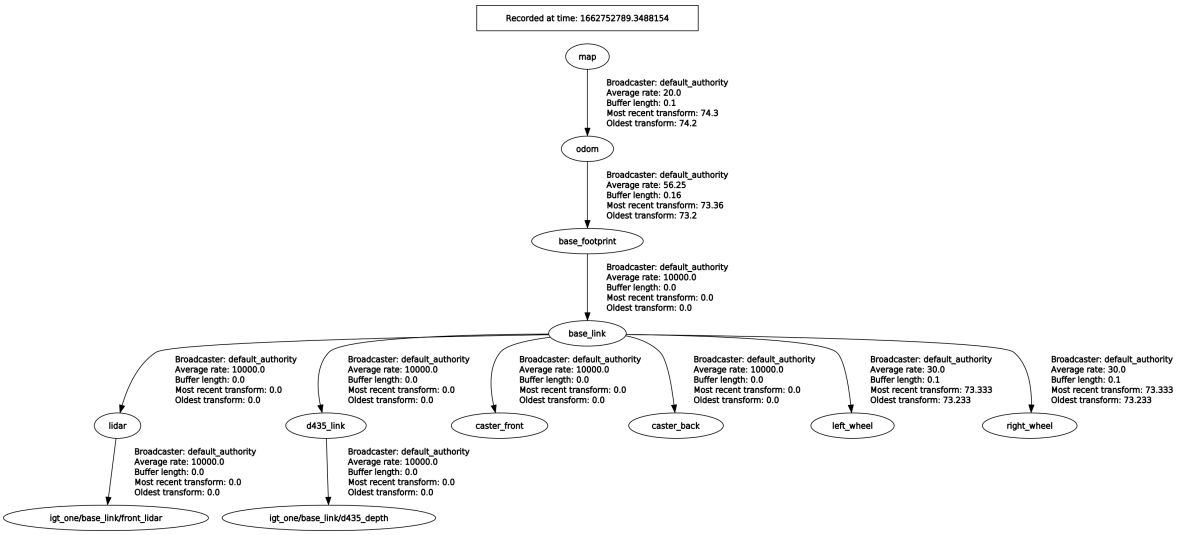

Figura 4.6: Árbol de transformadas basado en los mensajes de TF.

Se puede ver que se dispone de todos los frames generales, como *map*, *odom*, *base_footprint* y *base_link*, donde están el resto de transformadas con o elementos de la plataforma especificados en el modelo SDF.

4.3. Experimentación con Rosont

Dentro de las opciones que se barajaban para la experimentación y el testeo de ontologías en ROS , entraba la posibilidad de poder registrar los datos de los mensajes para poder presentarlos Plot Juggler. Pero se ha encontrado el problema de que los tipos de mensaje sutilizados para el Diagnostico no están del todo soportados y no se pueden representar bien el paso de un estado a otro.

4.3.1. Framework de testeo para Rosont

Debido a estas complicaciones y a otra serie de factores, para la experimentación con Rosont se ha decidido crear un pequeño framework. Este framework ha consistido en recrear los mensajes de logging para que se puedan revisar desde la pantalla de terminal, pero además poder mostrar todo el proceso del razonador y cuales son las decisiones que toma según le van llegando valores nuevos.

Este framework utiliza un publicador diseñado solo para el testeo de la ontología. De esta forma se puede testear el nodo del Rosont si necesidad de involucrar a Gazebo ni los nodos del stack de navegación.

Un ejemplo de su funcionamiento se puede ver a continuación, dónde se puede ver una salida de terminal con los mensajes de logging durante una recepción de una nueva medida.

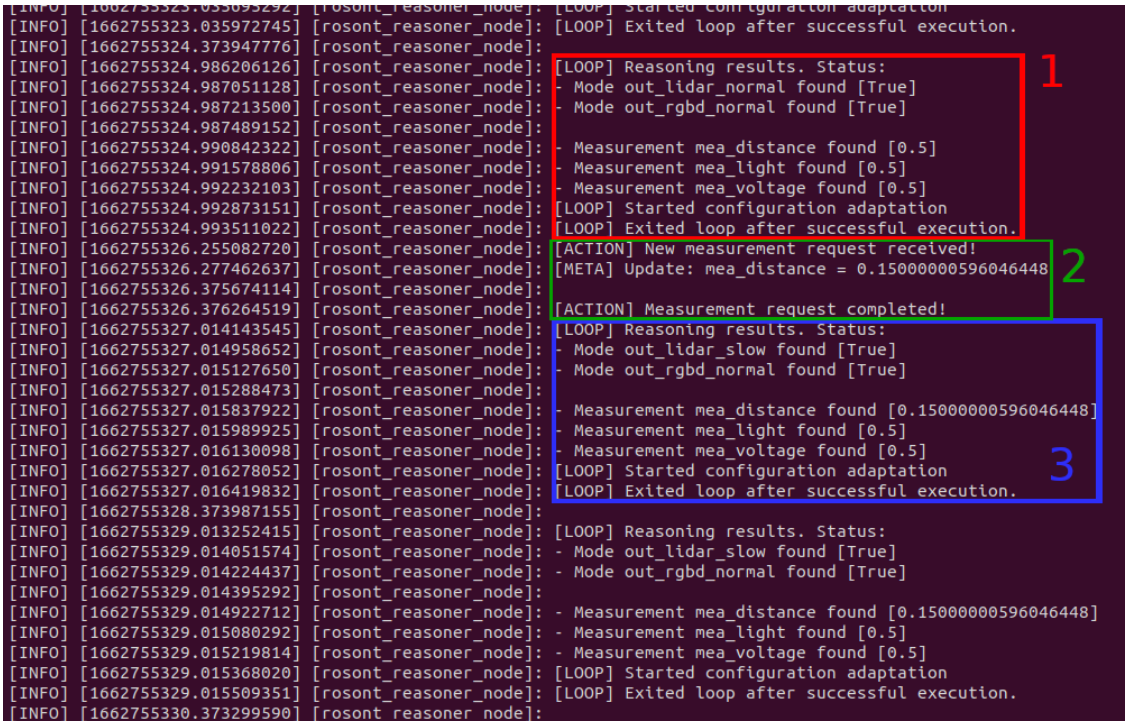

Figura 4.7: Terminal de debug durante la ejecución de Rosont.

Se puede apreciar todo el proceso de cambio en la ontología cuando llega una nueva medida de distancia. Esto son los pasos:

1. **Estado 1 (en rojo)**: Se muestra el estado actual junto con los dos modos de funcionamiento activos de la ontología, que en este caso son out_lidar_normal y out_rgbd_normal. Esto es debido a la plataforma se encuentra en los puntos centrales del estado de las medidas, ya que todas están en 0.5 sobre 1.
2. **Estado 2 (en verde)**: Se recibe una actualización de la medida de distancia, en este caso con un valor 0,15 sobre 1. Vemos que además se completa todo la Acción para actualizar la medida.
3. **Estado 3 (en azul)**: Con la anterior actualización se debe realizar un nuevo razonamiento para establecer los nuevos modos de funcionamiento, si es que es necesario. En este caso Se puede ver como el estado del LiDAR ha cambiado de velocidad de Normal a Slow, para adecuarse a las nuevas restricciones del entorno.

Y este proceso entero se repite cada vez que se recibe un nuevo dato actualizado desde el stack de navegación.

4.3.2. Testeo de las reglas SWRL con Protégé

También cabe mencionar, que aunque no se puede testear de una forma tan automática y rápida como en el caso explicado anteriormente, también se ha usado Protégé para hacer tests. En este caso se han utilizado las pestañas de SWRL, ROWL y SQWRL para el testeo del funcionamiento de la reglas y de la ontología de Rosont en general. Esto por supuesto ha consistido en un trabajo manual para comprobar que las reglas se ejecutaban según las condiciones que se han explicado en este proyecto.

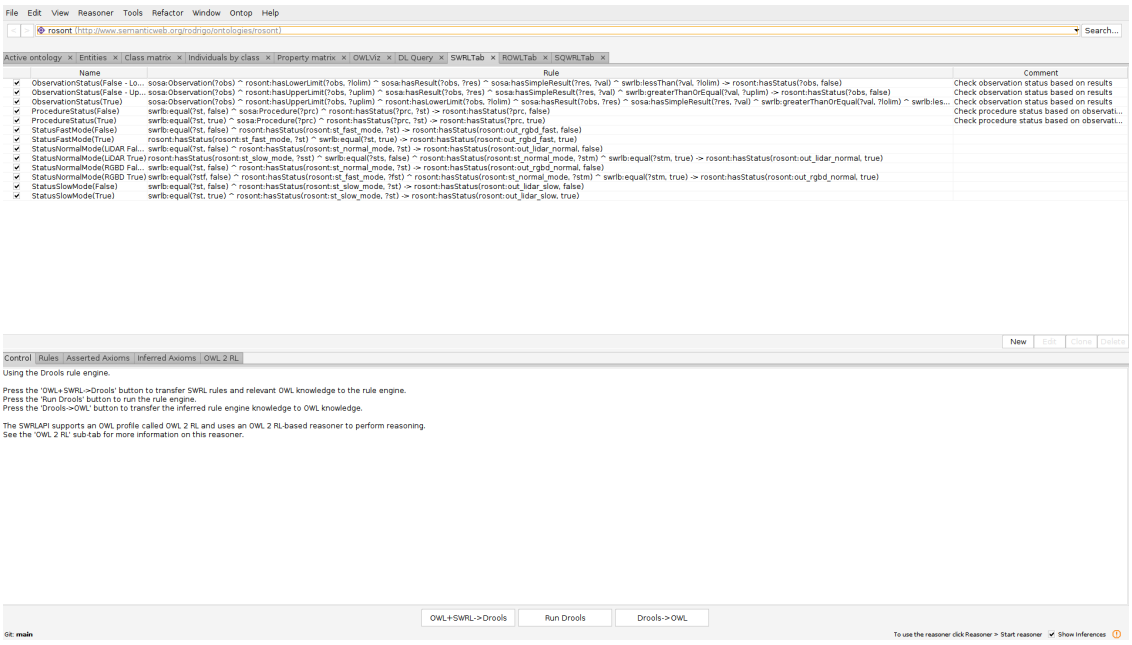

Figura 4.8: Pestaña de SWRLTab para el testeo en Protégé.

5. CONCLUSIONES

En este proyecto se ha desarrollado un metacrontrolador que opera un razonador de ontologías enfocadas a sensores. También se ha reimplementado la navegación de la plataforma utilizando las versiones nuevas de ROS2 y Gazebo Ignition.

El metacontrolador ha demostrado ser un concepto interesante cuanto menos. Es una forma novedosa de abordar el rediseño de los parámetros de controladores robóticos, sobre la marcha y utilizando arboles de conocimientos del entorno del robot. Además, al utilizar ontologías se puede delegar gran parte del control y del coste de computo necesario para la reparametrización del sistema, en un sistema agnóstico e independiente como es una ontología. Esto permite crear un sistema con una gran resiliencia ya que, aunque alguno de los Individuos que forma la ontología no funcionase como debiera, se podría identificar cual de todos ellos es y seguir el profundizando por el árbol de conocimientos hasta dar con la causa, o el ente que empieza con el error.

Las ontologías han demostrado su valor en el proyecto. Aunque dada su capacidad de abstracción para explicar conocimientos es necesario tener un cierto nivel de manejo de ontologías, arboles de conocimiento y similares, al final han acabado siendo una herramienta muy útil. Aunque es difícil crear ontologías que se adapten con gran precisión a arquitecturas específicas, a la vez que permiten detallar sus elementos con gran precisión, y que sirven como estándares para otros trabajos, es una gran herramienta para poder representar conocimiento y que las maquinas lo entiendan. La investigación de trabajo relacionados dejó claro que los intentos de crear ontologías para proyecto acaban teniendo resultados muy variados, y que por desgracia no es un campo que haya recibido mucha atención por el momento, menos aún en robótica.

En el caso de los desarrollos en ROS2. Ha sido una clara oportunidad de profundizar en las diferencias que hay entre las versiones de ROS1 y ROS2, ya que toda la reimplementación ha ido en ese sentido. Se puede ver que en las nuevas distribuciones de ROS2 se están incluyendo muchísimas mejoras, pero aunque el proyecto ha sido desarrollado principalmente en ROS Foxy Fitzroy, las nuevas versiones como Galactic Geochelone están introduciendo machas mejoras en los paquetes del stack de navegación. A la vez, en ROS2 se están afianzando de nuevo numerosas convenciones de diseño, programación y ejecución que faltaban en la comunidad de ROS1. Siendo una comunidad de código abierto, y tan participativa, muchas veces se hecha de menos la cohesión y coherencia que tienen frameworks más cerrados. Por ello me alegra ver que la transición a ROS2 se esta realizando progresivamente y de una forma tan fluida, teniendo en cuenta que los cambios entre ambas versiones de ROS suponen una gran barrera de entrada para que muchos proyectos decidan cambiar de distribución.

También en el caso de Gazebo Ignition se ha podido observar que las funcionalidad que hacían tan famoso al simulador de robótica, siguen estando ahí y con muchas innovaciones y utilidades nuevas. Las nuevas actualizaciones en la renderización de objetos hacen que simulaciones tan grandes (como la cueva con la que trabajamos en este proyecto) puedan correr sin mayores problemas en ordenadores de nueva generación y con un rendimiento increíble. Una vez que los principales paquetes de ROS que se utilizaban con Gazebo Classic hagan su transición a ROS2, y aprovechen para implementar los cambios de Gazebo Ignition, estoy bastante seguro de que veremos stacks de simulación muy completos y variados en los próximos años.

La plataforma al finalizar este proyecto tiene implementada el stack de navegación bajo ROS2, un metacontrolador que utiliza ontologías de sensores y que está programado bajo Python, y todo ello preparado para ser simulado utilizando Gazebo Ignition.

6. LÍNEAS FUTURAS DE TRABAJO

Para finalizar este trabajo, se van a presentar algunas ideas o conceptos que, junto con los resultados obtenidos de este trabajo, podrían derivar en nuevos proyectos o líneas de posible investigación.

Uno de los principales desarrollos que se pueden realizar con este proyecto sería termina de integrar todos los sensores y actuadores dentro de la ontología que ya se ha creado con Rosont. Por ejemplo, se podrían añadir los motores de la plataforma como Actuadores, y definir parte de su comportamiento en función como una inferencia de la ontología (límites de velocidad por ejemplo).

Ya que la plataforma posee una cámara de visión se podría añadir procesamiento de imágenes a la arquitectura de ROS para poder desarrollar otros proyectos con ello. Se podría tratar de ampliar la ontología de Rosont, que es una ontología enfocada a sensórica, con elementos más propios de otros ámbitos, como lenguaje escrito, reconocimiento de patrones, etc.

Tanto la ontología como la metodología empleada, son plenamente portables a otros robots y arquitecturas similares, por lo que se pueden utilizar en infinidad de proyectos de robótica móvil.

BIBLIOGRAFÍA

[1] *RobMoSys*. RobMoSys. URL: https://robmosys.eu/.

[2] C. Hernández Corbato et al. «MROS: Runtime Adaptation For Robot Control Architectures». En: (oct. de 2020).

[3] C. Hernández, J. Bermejo-Alonso y R. Sanz. «A self-adaptation framework based on functional knowledge for augmented autonomy in robots». En: *Integrated Computer-Aided Engineering* 25.2 (14 de mar. de 2018). Ed. por C. Schlenoff, S. Balakirsky y H. Christensen, págs. 157-172. ISSN: 10692509, 18758835. DOI: 10.3233/ICA-180565. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/ICA-180565.

[4] M. D. Schmill et al. «Ontologies for reasoning about failures in AI systems». En: *in Proceedings from the Workshop on Metareasoning in Agent Based Systems at the Sixth International Joint Conference on Autonomous Agents and Multiagent Sytems*. 2007.

[5] F. Alvares, E. Rutten y L. Seinturier. «A domain-specific language for the control of self-adaptive component-based architecture». En: *Journal of Systems and Software* 130 (ago. de 2017), págs. 94-112. ISSN: 01641212. DOI: 10.1016/j.jss.2017.01.030. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121217300201.

[6] J. Bermejo-Alonso y R. Sanz. «An Ontological Framework for Autonomous Systems Modelling». En: 2010.

[7] J. Bermejo-Alonso, C. Hernandez y R. Sanz. «Model-based engineering of autonomous systems using ontologies and metamodels». En: *2016 IEEE International Symposium on Systems Engineering (ISSE)*. 2016 IEEE International Symposium on Systems Engineering (ISSE). Edinburgh, United Kingdom: IEEE, oct. de 2016, págs. 1-8. ISBN: 978-1-5090-0793-6. DOI: 10.1109/SysEng.2016.7753185. URL: http://ieeexplore.ieee.org/document/7753185/.

[8] M. Stenmark y P. Nugues. «Natural language programming of industrial robots». En: *IEEE ISR 2013*. IEEE ISR 2013. Oct. de 2013, págs. 1-5. DOI: 10.1109/ISR.2013.6695630.

[9] J. Felip, J. Laaksonen, A. Morales y V. Kyrki. «Manipulation primitives: A paradigm for abstraction and execution of grasping and manipulation tasks». En: *Robotics and Autonomous Systems* 61.3 (mar. de 2013), págs. 283-296. ISSN: 09218890. DOI: 10.1016/j.robot.2012.11.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0921889012002217.

[10] A. Perzylo et al. «Intuitive instruction of industrial robots: Semantic process descriptions for small lot production». En: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Daejeon, South Korea: IEEE, oct. de 2016, págs. 2293-2300. ISBN: 978-1-5090-3762-9. DOI: 10.1109/IROS.2016.7759358. URL: http://ieeexplore.ieee.org/document/7759358/.

[11] M. Tenorth y M. Beetz. «KnowRob: A knowledge processing infrastructure for cognition-enabled robots». En: *The International Journal of Robotics Research* 32.5 (abr. de 2013),

págs. 566-590. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364913481635. URL: http://journals.sagepub.com/doi/10.1177/0278364913481635.

[12] M. Waibel et al. «RoboEarth». En: *IEEE Robotics & Automation Magazine* 18.2 (jun. de 2011), págs. 69-82. ISSN: 1070-9932. DOI: 10.1109/MRA.2011.941632. URL: http://ieeexplore.ieee.org/document/5876227/.

[13] S. Lemaignan, R. Ros, L. Mösenlechner, R. Alami y M. Beetz. «ORO, a knowledge management platform for cognitive architectures in robotics». En: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010). Taipei: IEEE, oct. de 2010, págs. 3548-3553. ISBN: 978-1-4244-6674-0. DOI: 10.1109/IROS.2010.5649547. URL: http://ieeexplore.ieee.org/document/5649547/.

[14] T. Haidegger et al. «Applied ontologies and standards for service robots». En: *Robotics and Autonomous Systems* 61.11 (nov. de 2013), págs. 1215-1223. ISSN: 09218890. DOI: 10.1016/j.robot.2013.05.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S092188901300105X.

[15] E. Prestes et al. «Towards a core ontology for robotics and automation». En: *Robotics and Autonomous Systems* 61.11 (nov. de 2013), págs. 1193-1204. ISSN: 09218890. DOI: 10.1016/j.robot.2013.04.005. URL: https://linkinghub.elsevier.com/retrieve/pii/S0921889013000596.

[16] A. Pease, I. Niles y J. Li. «The Suggested Upper Merged Ontology: AL arge Ontology for the Semantic Web and its Applications». En: (ago. de 2002).

[17] M. Compton, C. Henson, H. Neuhaus, L. Lefort y A. Sheth. «A Survey of the Semantic Specification of Sensors». En: vol. 522. Ene. de 2009.

[18] D. Russomanno, C. Kothari y O. Thomas. «Sensor ontologies: from shallow to deep models». En: *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST '05.* Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory (SSST05). Tuskegee, AL, USA: IEEE, 2005, págs. 107-112. ISBN: 978-0-7803-8808-6. DOI: 10.1109/SSST.2005.1460887. URL: http://ieeexplore.ieee.org/document/1460887/.

[19] M. Botts. «OpenGIS® Sensor Model Language (SensorML) Implementation Specification. Version 1.0.0.» En: (2007). Col. de UNESCO/IOC, UNESCO/IOC, M. Botts y A. Robin. Medium: 180pp. Publisher: Open Geospatial Consortium. DOI: 10.25607/OBP-646. URL: https://www.oceanbestpractices.net/handle/11329/1121.

[20] E. A. Topp et al. «Ontology-Based Knowledge Representation for Increased Skill Reusability in Industrial Robots». En: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Madrid: IEEE, oct. de 2018, págs. 5672-5678. ISBN: 978-1-5386-8094-0. DOI: 10.1109/IROS.2018.8593566. URL: https://ieeexplore.ieee.org/document/8593566/.

[21] S. Balakirsky, Z. Kootbally, C. Schlenoff, T. Kramer y S. Gupta. «An industrial robotic knowledge representation for kit building applications». En: *2012 IEEE/RSJ International*

Conference on Intelligent Robots and Systems. 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2012). Vilamoura-Algarve, Portugal: IEEE, oct. de 2012, págs. 1365-1370. ISBN: 978-1-4673-1736-8 978-1-4673-1737-5 978-1-4673-1735-1. DOI: 10.1109/IROS.2012.6385871. URL: http://ieeexplore.ieee.org/document/6385871/.

[22] L. Jacobsson. «A Module-Based Skill Ontology for Industrial Robots». En: 2015.

[23] W. N. Borst y W. N. Borst. «Construction of Engineering Ontologies for Knowledge Sharing and Reuse». ISBN: 90-365-0988-2. Tesis doct. Netherlands: University of Twente, 5 de sep. de 1997.

[24] Ó. Corcho, M. Fernández-López y A. Gómez-Pérez. «Methodologies, tools and languages for building ontologies: Where is their meeting point?» En: *Data Knowl. Eng.* 46 (2003), págs. 41-64.

[25] T. R. Gruber. «A translation approach to portable ontology specifications». En: *Knowledge Acquisition* 5.2 (jun. de 1993), págs. 199-220. ISSN: 10428143. DOI: 10.1006/knac.1993.1008. URL: https://linkinghub.elsevier.com/retrieve/pii/S1042814383710083.

[26] *OWL Web Ontology Language Overview*. URL: https://www.w3.org/TR/owl-features/.

[27] *RDF - Semantic Web Standards*. URL: https://www.w3.org/RDF/.

[28] D. Mun y K. Ramani. «Knowledge-based part similarity measurement utilizing ontology and multi-criteria decision making technique». En: *Advanced Engineering Informatics* 25.2 (abr. de 2011), págs. 119-130. ISSN: 14740346. DOI: 10.1016/j.aei.2010.07.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S1474034610000704.

[29] V. Hirankitti y M. Trang. «A Meta-reasoning Approach for Reasoning with SWRL Ontologies». En: *Lecture Notes in Engineering and Computer Science* 2188 (mar. de 2011).

[30] *ROS: Why ROS?* URL: https://www.ros.org/blog/why-ros/.

[31] J. Lee et al. «DART: Dynamic Animation and Robotics Toolkit». En: *The Journal of Open Source Software* 3.22 (5 de feb. de 2018), pág. 500. ISSN: 2475-9066. DOI: 10.21105/joss.00500. URL: http://joss.theoj.org/papers/10.21105/joss.00500.

[32] *Welcome to Owlready2's documentation! — Owlready2 0.36 documentation*. URL: https://owlready2.readthedocs.io/en/v0.37/.

[33] *Semantic Sensor Network Ontology*. URL: https://www.w3.org/TR/vocab-ssn/.

[34] D. Faconti. *Facontidavide/Plotjuggler: The time series visualization tool that you deserve.* URL: https://github.com/facontidavide/PlotJuggler.

A. Código del proyecto

Debido a que el código del proyecto es demasiado extenso para poder ser añadido a este documento, y además se compone de varios módulos independientes, se ha decidido subirlo a un repositorio de Github.

Repositorio del proyecto: https://github.com/rsilverioo/ontology_miner_robot

B. Planificación, presupuesto y análisis social-profesional

Para este proyecto se ha desarrollado una planificación temporal a modo de seguimiento del trabajo y tareas realizadas. De esta forma se pretendía poder mostrar un resumen en el tiempo de cuanto se había tardado para realización del proyecto, a la vez que se podría comprobar que si la planificación previa que se había realizado resultaba correcta.

Además de la planificación temporal, se ha creado un presupuesto creando una estimación de los costes asociados del proyecto cómo si fuera un proyecto creado integralmente en una empresa privada y dependiente de las regulaciones que atañerían al mismo.

En conjunto con este presupuesto, se ha realizado un pequeño estudio de cual sería el efecto de las integraciones del proyecto en el plano social y profesional de automatización robótica.

B.1. Planificación temporal

En esta sección se van a presentar las tareas y desarrollos que se han llevado a cabo en la realización del TFM. Toda la planificación se ha creado teniendo como base la asignación horaria de los créditos ETCS, que se ha estimado en torno a 25 horas por crédito, haciendo un total de 300 horas de trabajo.

Tabla B.1: Tabla de las tareas y su planificación individual

Descripción	***Inicio***	***Final***	***Horas***
Investigación previa del proyecto	10/01/2022	25/01/2022	15
Configuración del entorno inicial de desarrollo	25/01/2022	03/02/2022	32
Integración del stack de navegación en ROS2	04/02/2022	28/03/2022	60
Testeo de la navegación	28/03/2022	05/04/2022	12
Implementar la aceleración gráfica en Linux	06/04/2022	19/04/2022	15
Investigación sobre ontologías y razonadores	20/04/2022	02/05/2022	22
Implementación de Rosont en Python	03/05/2022	02/06/2022	72
Implementación de las reglas SWRL	02/06/2022	13/06/2022	20
Reimplementación de Rosont en ROS2	13/06/2022	15/07/2022	36
Testeo de la navegación con ontologías	15/07/2022	27/07/2022	16
Redacción y edición del proyecto	28/07/2022	22/08/2022	30

A continuación, se presenta también un diagrama de Gantt con la misma planificación anterior, para poder apreciar la ejecución de las tareas y como se superponen algunas de ellas.

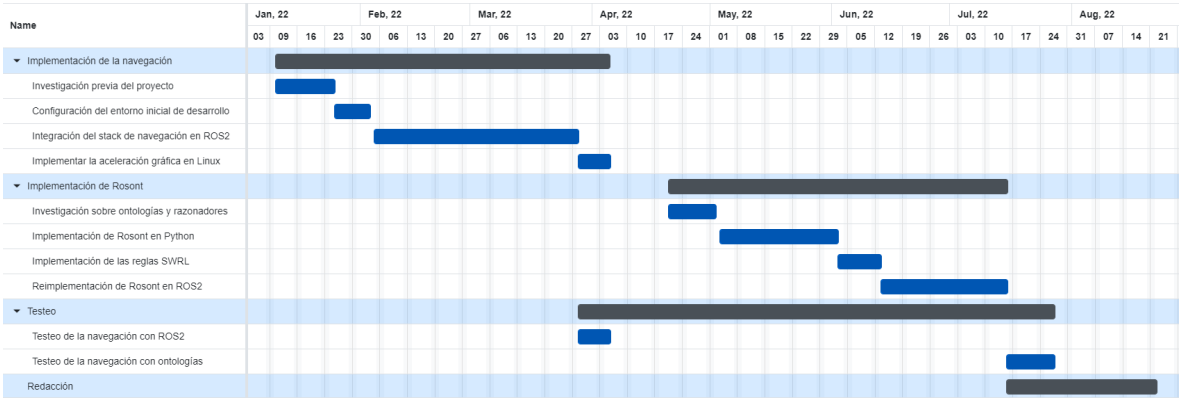

Figura B.1: Diagrama de Gantt con la planificación del proyecto.

Aunque los tiempos de testeo y redacción no parecen demasiado largos, solo se han tenido en cuenta los periodos en los que principalmente se estaba redactando o testeando, no en los que se hacían pequeñas tareas a la vez que se programaba o investigaba.

B.2. Presupuestos del proyecto

Para este proyecto se ha considerado que el alumno que ha realizado el trabajo es el principal trabajador del proyecto y que las figuras de los tutores del TFM son superiores o consultores externos que han ayudado en la realización del mismo. De esta forma se puede realizar una estimación de los sueldos y tiempos que cada uno ha invertido en el desarrollo del proyecto. También se han tenido en cuenta otros costes, como los materiales o los gastos de Internet o energía.

Como excepción se ha decidido que al ser un proyecto continuación de otro anterior, la plataforma robótica ha sido prestada o donada a coste cero, por lo que no era necesario añadirla al presupuesto.

Con estas consideraciones se puede dividir el presupuesto según la tipología de sus costes tal y como se muestra en los siguientes apartados.

B.2.1. Costes directos

Para calcular el coste de cada empleado se ha utilizado el Convenio de ingenierías y oficinas técnicas, que cifra el sueldo bruto mínimo en 14.000€ para un total de 1.792 horas anuales. Dado que este coste es el mínimo, y está bastante alejado de la realidad se ha decidido adecuarlo algo más a los costes reales para el mercado actual y proyectos similares.

También se ha considerado que el alumno es un investigador con contrato, debido al número de horas que invierte en el proyecto. El project manager (tutor) es un superior contratado pero con una asignación de horas menor en el proyecto, y la asesora externa (cotutora) se ha contratado para ofrecer soporte puntual a un proyecto de investigación.

Con estos cambios en cuenta se ha creado la siguiente tabla:

Tabla B.2: Tabla con los trabajadores y sus costes asociados

Trabajadores	***€/h***	***Horas***	***Coste €***
Trabajador (asalariado)	16	330	5.280,00
Project manager (tutor)	24	35	840,00
Asesora externa (cotutora)	20	25	500,00
Total			6.620,00

Además de los costes de personal también se ha requerido de material para la realización de este proyecto. En este sentido se han calculado únicamente los costes asociados directamente al desarrollo del proyecto, y se ha asumido que los equipos del project manager y de la asesora externa forman parte de asignaciones presupuestarias anteriores o son equipos propios.

Para el proyecto se ha utilizado un equipo de ultima generación modelo Acer Predator Orion 3000, que posee las siguientes principales características:

- Procesador Intel® CoreTM i7-12700
- Memoria RAM 16GB Up to 64 GB of Dual-channel DDR4 3200 MHz
- Disco duro 1 TB M.2 2280 PCIe SSD
- Controlador gráfico NVIDIA® GeForce® RTX™ 3060 12GB GDDR6

Además de esto se han utilizado otros periféricos con dicho equipo, que se puede ven en la siguiente tabla.

Tabla B.3: Tabla con los equipos y su coste individual

Equipo	***Unidades***	***Coste/u***
Acer Predator Orion 3000	1	1.370,00 €
Monitor Acer Nitro KG2 27"	2	180,00 €
Trust TKM-250 (ratón + teclado)	1	18,00 €
Total		1.748,00 €

No se tienen en cuenta los costes de software ya que la totalidad de los programas y frameworks utilizados en el proyecto son de licencia gratuita y con posibilidad de utilizarlos en proyectos comerciales solo con citarlos.

B.2.2. Costes indirectos

Los costes indirectos hacen referencia a costes asociados al uso de equipos, o al propio trabajo de desarrollo pero que no se pueden asignar a una tarea o proceso concreto. En este caso se consideran como costes indirectos la electricidad utilizada, los gastos de agua y los gastos de internet. Estos costes se estiman como un 15 % de los costes totales directos.

Tabla B.4: Tabla con la estimación de los costes indirectos

Costes	***Coste***
Coste de los trabajadores	6.620,00 €
Coste de los equipos	1.748,00 €
Total de costes directos	8.368,00 €
***Costes Indirectos* (15 % *CD*)**	1.255,20 €

B.2.3. Presupuesto final

Con los anteriores costes asociados se obtiene el coste total del proyecto que se puede ver a continuación.

Tabla B.5: Tabla con los costes totales de desarrollo

Costes	***Coste***
Costes directos	8.368,00 €
Costes indirectos	1.255,20 €
Costes totales	9,623,20 €

Con este presupuesto se pretenden ofrecer una orientación en cuales hubieran sido los costes reales del desarrollo del proyecto si se hubieran llevado a cabo en una institución privada, y sin mediar ningún tipo de acuerdo.

B.3. Análisis social-profesional

Como adición al proyecto también se va a añadir una pequeña sección dónde se va a realizar un pequeño análisis del impacto de la robotización de la minería y otros ámbitos.

En general, las nuevas tecnologías, y en concreto la robótica, están teniendo y van a a tener un gran impacto en el mundo laboral y profesional de muchos sectores. Al fin y al cabo la robotización de procesos supone una reducción de los costes en la producción y un aumento en la calidad de los resultados y productos obtenidos.

Partiendo de esa base, el desarrollo que se ha visto en los últimos años es que los robots ha pasado de resolver tareas sencillas y muy mecánicas, con las limitaciones que eso conlleva, a resolver tareas complejas que parecían estar ligadas a operarios humanos. En este sentido, el movimiento de los últimos años, y que ha impactado fuertemente el mercado laboral, ha sido que los robots pueden competir por trabajos que anteriormente no se podían automatizar.

Esta posibilidad de que los robots puedan suplir los empleos típicamente humanos, ha creado mucha polarización en ciertos sectores económicos, ya que estos cambios van a cambiar la estructura profesional de muchas industrias.

El resultado de este cambio será que, en el corto plazo, bastantes puestos de trabajo se pierdan debido a la automatización de estos empleos. Los operarios que se reemplacen por robots tendrán muy difícil volver a ocupar puestos similares, ya que en el medio y largo plazo toda la industria tenderá a la robotización de empleos.

Por otra parte, en el medio y largo plazo también se crearán puestos de trabajo, para el diseño la fabricación y el mantenimiento de estos operarios robóticos. Estos puestos de de trabajo serán empleos de nueva creación y se deberán definir con el paso de los años y las necesidades del mercado. Estos empleos, se han creado y irán creando con el tiempo de forma progresiva a la vez que se incrementará las oportunidades de formación para estos puestos de trabajo.

Siguiendo con esta misma línea, las empresas que no se adhieran al avance de la tecnología en términos de robotización de empleos, se encontrarán con que sus competidores aumentaran su rendimiento productivo a la vez que mejoran la calidad de sus productos y resultados. Las empresas que quieran mantenerse al margen tendrán que especializarse en trabajos más .ªrtesanalesz con actividades que dependan principalmente en las decisiones humanas.

El día en el que todas las tareas sean realizadas por robots queda aún muy lejos y de momento se antoja como algo irreal e imposible aunque cada día esa posibilidad parezca más cerca.