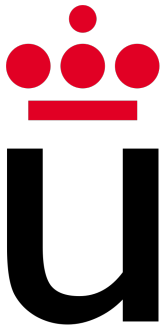


Universidad
Rey Juan Carlos

TRABAJO FIN DE GRADO

BT Studio: un IDE web para la programación de aplicaciones robóticas con Behavior Trees

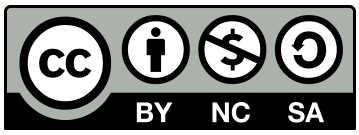
Grado en Ingeniería Robótica de Software

Escuela de Ingeniería de Fuenlabrada

Realizado por
Óscar Martínez Martínez

Dirigido por
José María Cañas

Curso académico 2023/2024



Agradecimientos

A todos mis profesores, especialmente José María, Paco, Roberto y José Centeno, por vuestra dedicación y enseñanzas. Gracias a vosotros, he llegado a ser un ingeniero.

A mi tutor de TFG, Jose María, por todo tu apoyo durante el desarrollo del trabajo y la ilusión que me has transmitido.

A mis amigos, que me han acompañado durante todo el camino.

A mi familia, sin vosotros no habría sido posible estar aquí. A mi madre, que me animó a perseguir mis sueños y estudiar robótica. Gracias por vuestro amor y apoyo constante en los momentos difíciles. A mi abuela, que le habría hecho mucha ilusión leer esta memoria, aún sin saber nada de robótica.

A Natalia, gracias por tantas alegrías y momentos juntos. Gracias por la motivación que siempre me das para seguir adelante.

Ad Astra per Aspera
Starfleet Command Motto

Resumen

La robótica es un sector que ha evolucionado enormemente en los últimos tiempos, extendiendo su aplicación más allá del ámbito industrial hacia tareas en entornos complejos y no estructurados. Este progreso se debe a mejoras en sensores, actuadores y procesadores, sumado al desarrollo de algoritmos y de técnicas de aprendizaje automático que permiten a los robots adaptarse para coexistir con los humanos y realizar tareas cada vez más útiles. Algunos ejemplos muy conocidos son la conducción autónoma o los robots de reparto.

Esto ha conllevado un aumento de la complejidad de las aplicaciones robóticas, que están formadas por un gran número de componentes con características muy dispares. Para facilitar el proceso de desarrollo de software, han surgido *middlewares* robóticos que buscan aportar una capa de abstracción, estandarizar los componentes y ofrecer herramientas de desarrollo. Avanzando en esta dirección, recientemente han surgido soluciones que buscan encapsular parte de la complejidad de las aplicaciones robóticas y proporcionar una interfaz web con varias utilidades que necesitan los programadores. Estas herramientas facilitan la integración y gestión de componentes robóticos, promoviendo un desarrollo más ágil y eficiente.

En este trabajo se estudia el diseño e implementación de BT Studio, un IDE web para la programación de aplicaciones robóticas basadas en *árboles de comportamiento*, un paradigma de programación de aplicaciones cada vez más usado en la industria robótica. En BT Studio, los usuarios pueden programar desde el navegador aplicaciones robóticas mediante acciones Python (programadas en el editor de texto incorporado) y árboles de comportamiento (definidos mediante un editor visual basado en bloques). Posteriormente, los usuarios pueden ejecutar estas aplicaciones en su máquina local o en el propio visualizador integrado en la plataforma web.

Palabras clave: robótica, árboles de comportamiento, inteligencia artificial, frontend, backend, ROS 2, Docker.

Acrónimos

TFG - Trabajo de Fin de Grado
ROS 2 - Robot Operating System 2
HTML - HyperText Markup Language
CSS - Cascading Style Sheets
AJAX - Asynchronous JavaScript and XML
IDE - Integrated Development Environment
RADI - Robotics Application Development Interface
NoSQL - Not Only SQL
XML - eXtensible Markup Language
LIDAR - Light Detection and Ranging
GPLv3 - GNU General Public License version 3
CPU - Central Processing Unit
GPU - Graphics Processing Unit
VNC - Virtual Network Computing
JSON - JavaScript Object Notation
BT - Behavior Tree

Índice general

Índice de figuras

Índice de tablas

Índice de extractos de código

1. Introducción

En este capítulo se introducirá el contexto en el que este TFG ha sido desarrollado. Para ello es necesario proporcionar definiciones y un resumen del estado del arte de las dos disciplinas en cuya intersección se encuentra el trabajo realizado: la robótica y las tecnologías web.

1.1. Robótica

1.1.1. Estado del arte

La robótica es una disciplina que integra campos como la ingeniería mecánica, la informática, la electrónica y la inteligencia artificial, para el diseño construcción y operación de sistemas robóticos. Los robots son sistemas autónomos capaces de ejecutar actividades en diversos entornos. Desde sus inicios a mediados del XX, la robótica se ha expandido desde aplicaciones preeminentemente industriales con brazos robóticos hacia la conocida como robótica de servicios. Los robots de servicio son aquellos son capaces de realizar funciones en un rango mucho mayor de entornos, en muchos casos, cambiantes y no estructurados.

Todos los sistemas robóticos tienen tres componentes fundamentales: sensores, actuadores y hardware para ejecutar algoritmos de control y toma de decisiones (principalmente, CPUs y GPUs). Sobre estos componentes se ejecutan algoritmos que a partir de la información proporcionada por los sensores, son capaces de comandar a los actuadores para ejecutar una tarea determinada. En los últimos tiempos, la autonomía y la capacidad de decisión de los robots se ha incrementado enormemente por la adopción de algoritmos deliberativos basados en el aprendizaje automático automático, como redes neuronales o *Reinforcement Learning*. Algunas de las aplicaciones robóticas que más han avanzado en los últimos tiempos son:

- **Logística en almacenes:** los robots optimizan las operaciones al transportar de manera autónoma mercancías entre ubicaciones, como el ejemplo de la figura 1.1. Equipados con sensores y sistemas de navegación avanzados, estos robots mejoran la eficiencia y reducen los errores humanos al realizar tareas

de picking, reposición y transporte de carga.

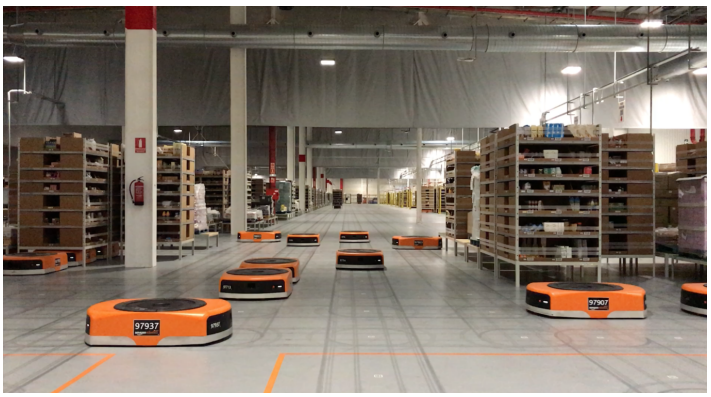


Figura 1.1: Robots empleados en los almacenes de Amazon

- **Reparto Autónomo:** esta aplicación involucra el uso de vehículos autónomos para la entrega de paquetes sin intervención humana directa. A través de tecnologías de navegación y planificación de rutas, el reparto autónomo permiten reducir los tiempos de entrega y los costes, a la vez que se aumenta la seguridad. 1.2

Figura 1.2: Robot de reparto de la empresa Starship

- **Conducción Autónoma:** los vehículos autónomos, equipados con sistemas avanzados de detección, procesamiento de datos y actuadores, son capaces de navegar en el tráfico sin intervención humana. Su desarrollo tiene como objetivo mejorar la seguridad vial, aumentar la eficiencia del transporte y facilitar la movilidad de personas con diversidad funcional. Diversas empresas como Tesla, Cruise o Waymo (1.3) ya ofrecen servicios comerciales con coches autonomos.

Figura 1.3: Taxi autónomo de la empresa Waymo

- **Robots de Limpieza:** estos robots autónomos se encargan de la limpieza de diversos entornos, desde hogares hasta espacios públicos y oficinas. Incorporan sensores para detectar suciedad y obstáculos, junto con algoritmos para planificar rutas de limpieza eficientes. 1.4

Figura 1.4: Robot de limpieza Roomba de la empresa IRobot

1.1.2. Desarrollo de aplicaciones robóticas

Las aplicaciones robóticas han ido ganando complejidad para cubrir casos de uso más avanzados. El procesamiento suele estar distrubuido en numerosos nodos, que se ejecutan a distinto ritmo y requieren distinta información para su funcionamiento. Las aplicaciones robóticas deben combinar una alta reactividad para la interacción con el entorno cambiante junto con algoritmos de navegación y toma de decisiones. Para la organización de esta creciente complejidad, han

aparecido herramientas que facilitan el desarrollo de software, como los *middlewares* robóticos o los simulares.

Los *middlewares* robóticos ofrecen un gran variedad de herramientas como abstracción de hardware, bibliotecas de comunicaciones o integración con diversos simuladores. El middleware más extendido para el desarrollo de aplicaciones robóticas es ROS 2, explicado en profundidad en la sección 3.3.1. ROS 2 permite la programación de aplicaciones como una orquesta de nodos que realizan distintas tareas y se comunican entre ellos.

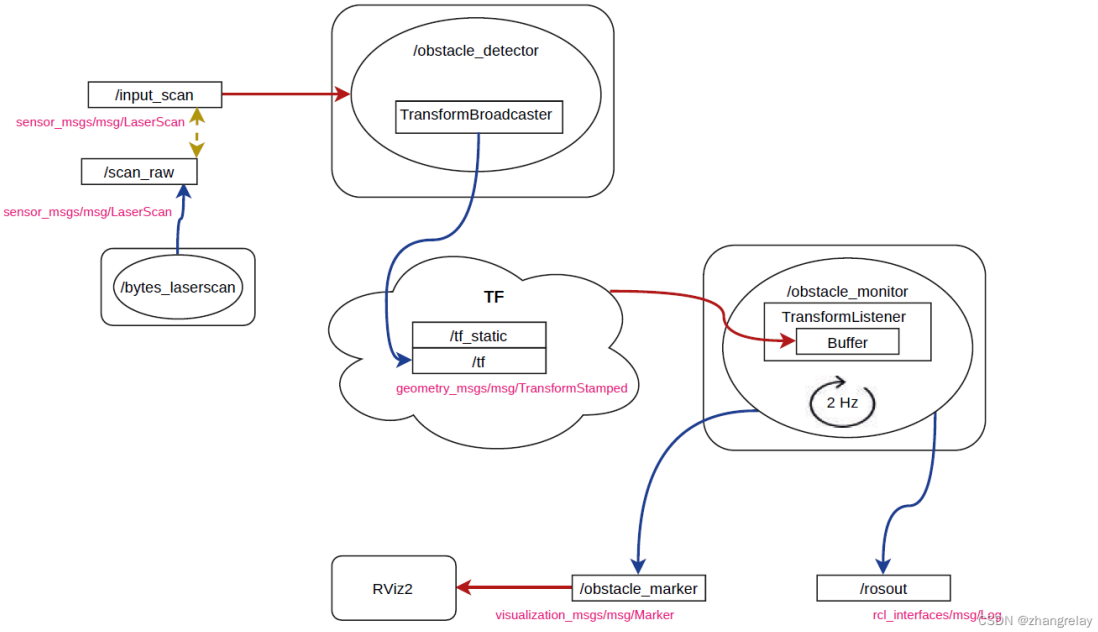


Figura 1.5: Grafo de computación de una aplicación en ROS 2

Los simuladores permiten depurar y verificar las aplicaciones robóticas en entornos físicamente realistas, proporcionando herramientas para el modelado de robots, sus sensores y actuadores. Para ello, deben replicar condiciones físicas variadas, como la fricción, la gravedad, la colisión de objetos o la iluminación. Además, los simuladores ofrecen la capacidad de ajustar parámetros y configuraciones de manera eficiente, lo que permite desarrollar nuevos algoritmos sin el riesgo de dañar equipos reales, acelerando el proceso de desarrollo y la seguridad. El simulador más utilizado es Gazebo (sección 3.3.2), que cuenta con integración total con ROS 2.

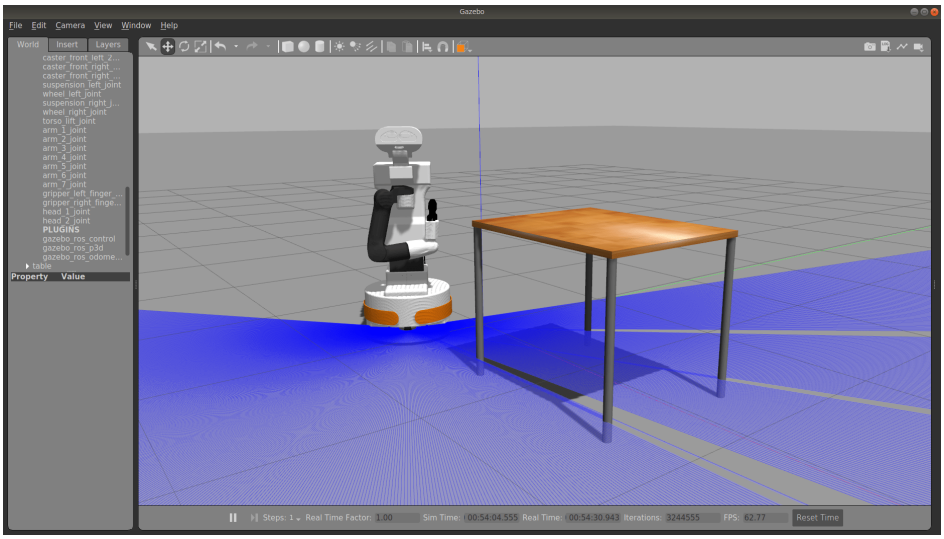


Figura 1.6: Entorno simulado mediante Gazebo

1.1.3. Paradigmas de organización de las aplicaciones robóticas

Los distintos componentes que tipicamente forman una aplicación robótica tienen funciones diversas, que deben ser ejecutadas de distintas maneras:

- **Componentes reactivos:** proporcionan una respuesta inmediata a los cambios del entorno. Operan sobre el principio de estímulo-respuesta, permitiendo al robot actuar de manera rápida y eficiente ante situaciones imprevistas, sin necesidad de planificación compleja. Estos componentes se encargan de tareas que requieren tiempos de respuesta breves, como la evasión de obstáculos o el seguimiento de una ruta preestablecida. Es por ello que se ejecutan con una frecuencia muy elevada.
- **Componentes deliberativos:** están diseñados para realizar tareas complejas que requieren una consideración anticipada de varios factores y consecuencias potenciales. Permiten al robot tomar decisiones basadas en modelos internos del mundo, planificar acciones a largo plazo y resolver problemas de manera autónoma. Se encargan de acciones que necesitan un análisis detallado y la toma de decisiones estratégicas, como la navegación en entornos desconocidos o la manipulación de objetos en tareas de montaje. Estos componentes generalmente se ejecutan a una frecuencia reducida, generando planes de actuación que los componentes reactivos llevarán a cabo.
- **Componentes de gestión de la ejecución:** ofrecen un modelo para organizar

y coordinar las acciones del robot, proporcionando un estado *interno* al robot. Hay dos tipos muy extendidos: las máquinas de estado y los árboles de comportamiento. Las máquinas de estado permiten modelar la lógica de control en situaciones donde el comportamiento del robot puede clasificarse en un conjunto finito de estados, cuya transición está determinada por un conjunto de reglas claras y deterministas. Los árboles de comportamiento, por su parte, proporcionan una estructura para manejar decisiones y comportamientos más complejos, basada en la selección dinámica de tareas y acciones reutilizables. Este último paradigma está ganando mucha fuerza en la industria, debido a su mayor flexibilidad, expresividad y facilidad de uso, existiendo numerosas herramientas para facilitar su uso.

1.2. Tecnologías web

1.2.1. Estado del arte

Las tecnologías web han evolucionado enormemente en los últimos años para proporcionar mejor experiencia del usuario, eficiencia, versatilidad y accesibilidad. Inicialmente, las páginas web eran fundamentalmente estáticas, creadas con HTML básico. Con el tiempo, la introducción de CSS permitió una mayor personalización del diseño, mientras que JavaScript trajo interactividad a las páginas web. La adopción de AJAX (Asyncronous Javascript and XML) marcó un punto de inflexión, permitiendo la carga de contenido dinámico sin necesidad de recargar la página completa.

Esta progresión hacia aplicaciones web más dinámicas y responsivas se ha acelerado con la introducción de frameworks y bibliotecas modernas como React, Vue y Angular. Estas herramientas han simplificado el desarrollo de interfaces de usuario complejas y mejorado la experiencia del usuario al ofrecer aplicaciones web con un comportamiento muy cercano a las aplicaciones nativas.

Figura 1.7: Interfaz creada con React

Las tecnologías de bases de datos también han evolucionado para satisfacer las demandas de las aplicaciones web modernas. Las bases de datos NoSQL, como MongoDB, ofrecen flexibilidad para manejar grandes volúmenes de datos no estructurados, mientras que las bases de datos relacionales como PostgreSQL han mejorado su rendimiento y escalabilidad.

Algunos ejemplos de aplicaciones web modernas son:

- **Plataformas de comercio electrónico**: estas aplicaciones web permiten a los usuarios buscar, comparar y comprar productos con facilidad desde sus hogares. Utilizan tecnologías web avanzadas para ofrecer interfaces de usuario intuitivas, procesos de pago seguros y personalizados, y una experiencia de navegación fluida incluso en dispositivos móviles.

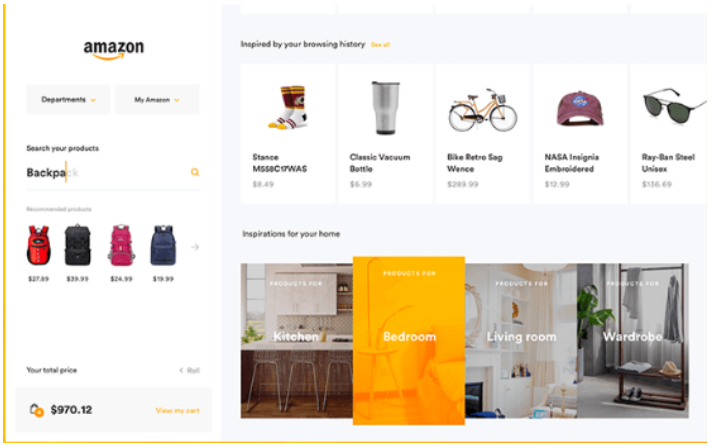


Figura 1.8: Página de comercio electrónico de Amazon

- **Aplicaciones de visualización de vídeos**: YouTube es un ejemplo prominente

de una aplicación web con elementos altamente reactivos, diseñada para transmitir contenido de video a millones de usuarios en todo el mundo. Estas aplicaciones utilizan tecnologías como HTML5 y WebRTC para ofrecer reproducción de video fluida y comunicación en tiempo real, respectivamente.

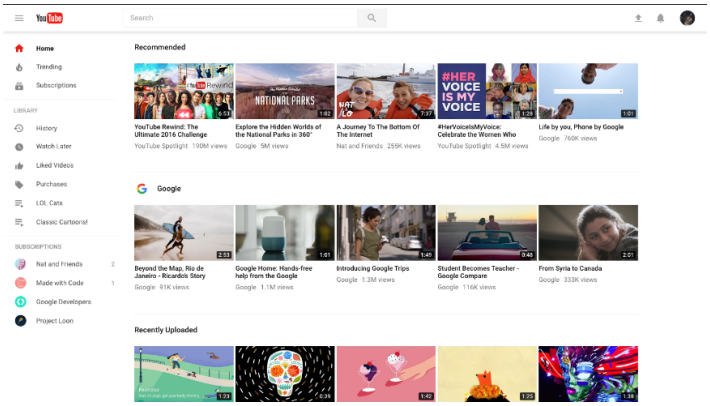


Figura 1.9: Interfaz de Youtube

1.2.2. Plataformas web para programar

Existen diversas plataformas web diseñadas para facilitar el aprendizaje, el desarrollo y la ejecución de proyectos de software con varios niveles de complejidad y para distintos públicos objetivos. Estas plataformas hacen uso de tecnologías web modernas para permitir a los usuarios programar, ejecutar y depurar aplicaciones sin salir del navegador. Algunos ejemplos son:

- **Overleaf**[1] [1]: es una herramienta para la de edición de textos en línea basada en LaTeX, diseñada para la elaboración colaborativa de documentos científicos y técnicos. Permite a usuarios trabajar juntos en tiempo real desde cualquier navegador, ofreciendo acceso a plantillas para distintos tipos de documentos. Simplifica la creación de documentos con composición tipográfica de alta calidad, sin requerir instalación de software LaTeX, haciendo más accesible la escritura académica y científica.
- **Scratch**[2]: es una plataforma educativa que permite a niños y principiantes introducirse en el mundo de la programación mediante una interfaz visual basada en bloques. Los usuarios pueden crear juegos, historias interactivas y

[1] https://www.overleaf.com/
[2] https://scratch.mit.edu/

animaciones de forma intuitiva, arrastrando y soltando bloques de código para formar secuencias lógicas.

- **CodePen**[3]: es una comunidad de desarrollo social para diseñadores y desarrolladores web. Permite a los usuarios crear, compartir y explorar fragmentos de HTML, CSS y JavaScript, conocidos como "Pens". Permite la visualización inmediata de los resultados, lo que lo convierte en una herramienta educativa y de prototipado muy versátil.
- **Arduino Web IDE**[4]: permite desarrollar y cargar programas (sketches) a placas Arduino directamente desde el navegador. Esta plataforma proporciona un IDE completo, con soporte para la edición de código, la gestión de bibliotecas y el acceso a una amplia gama de ejemplos.
- **Replit**[5]: ofrece un IDE donde los usuarios pueden escribir, ejecutar y compartir código en docenas de lenguajes de programación sin necesidad de configurar un entorno de desarrollo local.

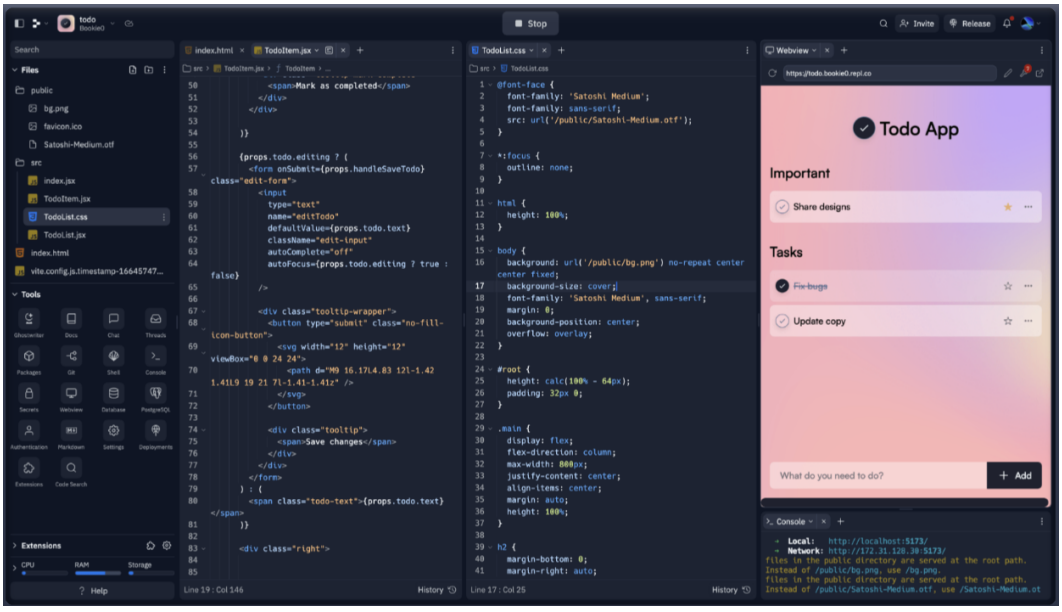


Figura 1.10: Interfaz de Replit

[3] https://codepen.io/
[4] https://app.arduino.cc/
[5] https://replit.com/

1.3. Plataformas web para la programación de aplicaciones robóticas

En la intersección de las plataformas mencionadas en el apartado anterior con la robótica, se hallan aplicaciones destinadas a la programación de aplicaciones robóticas desde el navegador. El concepto es el mismo: proporcionar un IDE web donde los usuarios dispongan de herramientas suficientes para realizar todas las tareas propias del desarrollo de una aplicación. En el caso de la programación de aplicaciones robóticas, estas plataformas deben proporcionar entornos de simulación especializados y una forma de organizar la ejecución de las aplicaciones.

1.3.1. Plataformas educativas

Dentro de las plataformas web para la programación de aplicaciones robóticas, hay algunas enfocadas en el aprendizaje. Su objetivo no es la creación de aplicaciones para entornos de producción, sino educar sobre diversos conceptos importantes del ámbito de la robótica proporcionado herramientas y entornos de prueba. Las dos plataformas de estas características más conocidas son:

- **TheConstruct**[6]**:** ofrece una amplia gama de cursos y simulaciones para aprender a programar robots utilizando ROS. La plataforma brinda acceso a un IDE basado en web y simuladores que corren en la nube, permitiendo a los usuarios desarrollar y probar sus aplicaciones y soluciones sobre robots simulados. Los cursos abarcan desde niveles básicos hasta avanzados, cubriendo temas como la navegación, manipulación y percepción, haciendo hincapié en la programación de robots en entornos realistas.
- **Riders.ai**[7]**:** esta plataforma permite a los usuarios participar en competiciones de programación de robots, donde pueden medir sus habilidades contra las de otros programadores. A través de su IDE web, los participantes tienen la oportunidad de escribir, probar y optimizar su código en simulaciones que replican desafíos de robótica del mundo real.

[6] https://app.theconstructsim.com/login/
[7] https://riders.ai/en

1.3.2. Plataformas profesionales

En los últimos dos años, se encuentran en auge las plataformas web dedicadas a la programación y despliegue de soluciones robóticas en entornos de producción. Estas ofrecen la posibilidad de crear aplicaciones robóticas avanzadas mediante el uso de algoritmos y herramientas de vanguardia, brindando funcionalidades más sofisticadas y sólidas en comparación con sus contrapartes educativas. Su propósito principal es democratizar el acceso a la programación de robots para una amplia gama de profesionales en el campo de la informática, facilitando el diseño de nuevos algoritmos sin la necesidad de enfocarse en la infraestructura robótica subyacente requerida para su ejecución. Las dos plataformas más conocidas son:

- **MoveitPro**[8]**:** desarrollada por PickNik se centra en la programación de brazos robóticos en entornos no estructurados. Utiliza árboles de comportamientos para gestionar la ejecución de la aplicación, proporcionando bloques predefinidos para tareas de planificación y percepción, entre otros. Facilita mucho la transición al brazo real mediante un simulador altamente realista que actúa como gemelo digital. Está construido sobre la librería *MoveIt*, estándar en la comunidad ROS 2 para la programación de manipuladores robóticos.
- **Flowstate**[9]**:** desarrollado por Intrinsic, una empresa propiedad de Google. Tiene como objetivo permitir la programación de aplicaciones industriales completas mediante un lenguaje de programación visual basado en bloques. Proporciona bibliotecas de bloques para tareas complejas, como la estimación de movimiento o la detección de objetos. Además, los usuarios pueden expandir estas bibliotecas o crear las suyas propias. Ofrece un entorno de simulación especializado, basado en ROS 2 y Gazebo.
- **Asimovo**[10]**:** es una plataforma de desarrollo de aplicaciones robóticas completas, especialmente enfocada en la robótica de servicio. Proporciona una extensa colección de herramientas necesarias para el desarrollo de soluciones robóticas para producción. Además, proporciona una *biblioteca* de robots y entornos de simulación.

El objetivo de este TFG es la creación de una plataforma *open source* similar, que permita programar aplicaciones robóticas basadas en árboles de comportamiento

[8] https://picknik.ai/pro/
[9] https://www.intrinsic.ai/flowstate/
[10] https://asimovo.com/

desde un navegador web.

2. Objetivos y metodología

2.1. Objetivos

Una vez explorada la motivación tras la creación de una herramienta como BT Studio, es necesario establecer objetivos concretos que guíen su desarrollo.

El objetivo principal de este TFG es la creación de un IDE web *open source* completamente funcional que potenciales usuarios puedan usar desde sus equipos, es decir, de manera offline tras un proceso de instalación. El desarrollo de este objetivo puede ser dividido en cuatro partes diferenciadas:

1. Programación de un editor visual usando tecnologías web modernas como REACT y Django.
2. Desarrollo de un traductor capaz de generar código Python a partir de la codificación de árboles de comportamiento como diagramas web. Esto permite transformar una representación web en aplicaciones robóticas ejecutables.
3. Integración con el RADI de RoboticsAcademy [1], añadiendo la capacidad de ejecución de las aplicaciones desde el browser, usando un entorno dockerizado.
4. Generación de aplicaciones de ejemplo para demostrar las capacidades de la solución. Se hará uso de un simulador robótico para su implementación.
    - *Laser Bump and Go*: comportamiento reactivo para evitar obstáculos usando medidas procedentes de un LIDAR.
    - *Follow Person*: usando reconocimiento visual, conseguir un comportamiento reactivo capaz de seguir a una persona.

Además, se define como objetivo adicional la integración de BT Studio en la plataforma Unibotics[2], con el objetivo de facilitar su uso y hacer la plataforma accesible a un mayor número de usuarios.

[1] https://github.com/JdeRobot/RoboticsAcademy
[2] https://unibotics.org/

El cumplimiento de cada uno de estos objetivos será validado adecuadamente en el capítulo 5.

2.2. Metodología

El modelo de trabajo de este TFG se basa en tres puntos importantes:

- **Reuniones semanales con mi tutor**: gracias a esta frecuencia, se consigue un desarrollo ágil y bien enfocado, con un feedback que puede ser muy detallado.
- **Mentalidad open source**: tras una fase inicial de experimentación, el trabajo se realizó en un repositorio público de GitHub [3]. Durante el proceso de desarrollo se trabajó con issues, parches y versiones, facilitando el uso y el estudio por parte de desarrolladores interesados. Se recibieron incluso algunas sugerencias y preguntas.

  La mentalidad *open source*[4] es crucial en el sector de la robótica como ha quedado demostrado con el gran éxito de ROS y ROS 2. Este TFG se adhiere estrictamente a este mentalidad, estando este texto bajo la licencia Creative Commons Attribution-ShareAlike 4.0 International y todo el código asociado bajo GPLv3.
- **Roadmap preciso**: el proyecto se desarrolló siguiendo un roadmap claro y estructurado, organizado en diferentes fases con objetivos específicos. Se adoptó una metodología ágil para guiar el desarrollo, lo que permitió una planificación flexible y adaptativa. El trabajo se dividió en sprints, cada uno enfocado en la preparación e implementación de distintas funcionalidades. Este enfoque promovió una comunicación constante y efectiva con mi tutor, permitiendo ajustes rápidos del plan basados en el feedback y los resultados obtenidos en cada sprint.

2.3. Plan de trabajo

El desarrollo de este Trabajo de Fin de Grado se ha producido entre junio de 2023 y marzo de 2024.

1. **Estudio de soluciones similares y estado del arte**

[3] https://github.com/JdeRobot/bt-studio
[4] https://opensource.org/osd/

2. **Familiarización con tecnologías de desarrollo web**: especialmente Django, Javascript y REACT. Estas tecnologías se usan en conjunto con aquellas más propias del ámbito de la robótica, como Behavior Trees, ROS 2 o Docker. Las características de cada tecnología y su uso en el TFG se detallan en el capítulo 3

3. **Versión 0.1**: ejecución offline básica nativa, con un traductor a Python de árboles definidos en ficheros de texto.
    - Tiene como objetivo la generación de paquetes ROS 2 para su uso en instalaciones locales del usuario.
    - La generación de árboles de comportamiento está basada en la librería py-trees.
    - Los árboles se definen en un fichero XML.
    - Frontend básico basado en Javascript, HTML y CSS.

4. **Versión 0.2**: ejecución offline completa, con aplicaciones definidas en el editor visual web.
    - Los árboles se definen de manera gráfica con un editor de bloques.
    - Frontend basado en REACT y Typescript.
    - Posibilidad de crear y guardar proyectos multifichero.
    - Soporte completo para los distintos componentes presentes en py-trees.
    - Mecanismo de gestión de dependencias.
    - Integración con herramientas de soporte, como un visualizador de ejecución.

5. **Versión 0.3**: ejecución dockerizada
    - Integrada con el RADI de RoboticsAcademy, lo que permite la ejecución de las aplicaciones en el browser, sin necesidad de instalación local.
    - Posibilidad de usar todos los robots y escenarios presentes en Robotic Infrastructure.
    - Consola y otras herramientas de depuración.

6. **Aplicaciones robóticas de validación**: desarrollo de las dos aplicaciones propuestas, que permiten la validación de la versión 0.2 y 0.3. Para la implementación de las soluciones, se hará uso de distintas librerías y

paquetes, lo que permitirá medir la compatibilidad y facilidad de uso de la plataforma, así como la extracción de métricas de rendimiento. Estas aplicaciones serán incluidas en el repositorio del proyecto para su consulta.

7. **Integración en Unibotics**: el código será integrado en la plataforma Unibotics, donde cualquier usuario podrá usar BT Studio tras registrarse, lo que potencialmente podría aumentar la adopción de la plataforma.

3. Fundamentos técnicos

En esta sección se detallan los fundamentos técnicos detrás de BT Studio, es decir, todas las herramientas, tecnologías y librerías utilizadas para su desarrollo. Dada la naturaleza del trabajo realizado, todas estas herramientas son de software y están divididas generalmente en dos tipos: herramientas usadas para el desarrollo web y herramientas para el desarrollo robótico. La única excepción es el lenguaje de programación Python, que se usa en ambos desarrollos.

3.1. Lenguaje Python

Python[1] es un lenguaje de programación de alto nivel interpretado, orientado a objetos (aunque también permite programación funcional) y con semántica dinámica. Es ampliamente utilizado en un gran número de aplicaciones, desde servidores web a inteligencia artificial. Para el desarrollo de BT Studio se ha utilizado la versión 3.10.12.

Entre sus características más destacadas podemos encontrar:

- Sintaxis sencilla y legible
- *Batteries included*[2]: contiene mucha funcionalidad adicional out-of-the-box incluida en su librería standard. Además, existen infinidad de librerías de terceros.
- Tiene una licencia de código abierto, lo que ha propiciado adopción masiva. Esto es especialmente relevante para su adopción en el campo de la robótica.
- *Tipado dinámico*: no es necesario declarar el tipo de las variables al inicializarlas y este puede cambiar en función del valor de la variable en un determinado momento.
- Es un lenguaje interpretado, por lo que no necesita compilar el código. Las sentencias del programa se ejecutan conforme se leen.

La versatilidad que ofrece Python conlleva una reducción del rendimiento y el

[1] https://www.python.org/
[2] https://peps.python.org/pep-0206/

consumo de memoria, lo que impone limitaciones en el tipo de sistemas donde se pueden desplegar programas python.

Sin embargo, para el presente TFG el uso de Python se ve justificado por necesidades de diseño, concretamente tres:

- El motor de ejecutor de árboles utilizado sólo permite describir las acciones en Python. Las aplicaciones generadas con BT Studio deben estar escritas en este lenguaje.
- Es el soportado por el entorno de ejecución de Unibotics, donde se integrará BT Studio.
- El backend web de BT Studio utiliza el framework Django, escrito en Python. Este framework es el estándar de la industria.

3.2. Herramientas de desarrollo web

Estas herramientas fueron usadas para la programación del backend y el frontend web. La función del primero es servir la página web del editor y proporcionar distintos servicios a través de de una API REST, mientras que la del segundo es mostrar una interfaz gráfica e interactiva en el navegador que permita usar todas las funcionalidades soportadas por el backend. Es especialmente destacable dentro del frontend la funcionalidad para la creación de árboles de comportamiento mediante bloques visuales.

3.2.1. HTML

HTML[3] es un lenguaje de marcado usado para definir la estructura de un documento web mediante etiquetas, que encapsulan las diferentes partes del contenido para otorgarles determinada apariencia o funcionalidad. Estos archivos no se ejecutan, sino que son leídos y representados por los navegadores de forma transparente al usuario. Para el desarrollo de este trabajo, se utilizó la última versión del *living standard* de HTML[4].

La funcionalidad de HTML se ve complementada por dos lenguajes adicionales: CSS, que aporta estilos a las diferentes partes de la web y Javascript, que provee dinamismo y reactividad.

[3] https://html.spec.whatwg.org/

[4] https://developer.mozilla.org/en-US/docs/Glossary/HTML5

HTML se ha utilizado para el desarrollo del frontend de BT Studio, aportando la estructura básica de su interfaz.

3.2.2. Javascript

Javascript[5] es un lenguaje de de programación de alto nivel interpretado, utilizado para agregar interactividad y dinamismo a las páginas webs. Permite realizar un gran número de acciones (como el control multimedia o la llamada a APIs) sin recargar toda la página web. Para el desarrollo de BT Studio, se ha empleado la versión EcmaScript 2023.

Concretamente, para el desarrollo del trabajo, Javascript se ha utilizado dentro de la librería REACT.

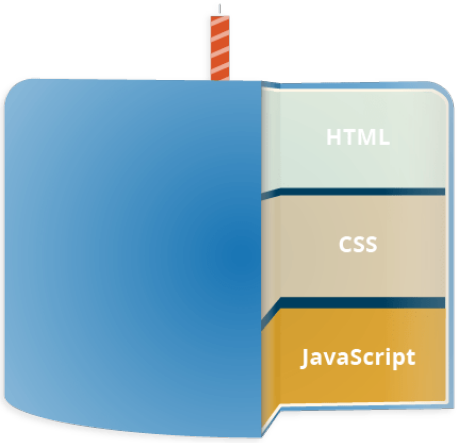


Figura 3.1: Stack típico de desarrollo web

3.2.3. React

React[6] es una biblioteca de JavaScript de código abierto diseñada para crear interfaces de usuario, con el objetivo concreto de facilitar el desarrollo de aplicaciones en una sola página. Es mantenida por Facebook y una comunidad de desarrolladores individuales y empresas. La versión empleada en este TFG fue la 18.2.0.

Sus características más importantes son[2]:

[5] https://ecma-international.org/publications-and-standards/standards/ecma-262/
[6] https://react.dev/reference/react

- **Componentes reutilizables**: React se basa en la composición de componentes que gestionan su propio estado, lo que permite desarrollar interfaces complejas a partir de pequeñas piezas reutilizables y aisladas.
- **Virtual DOM**: utiliza un DOM virtual que es una representación en memoria del DOM real. Esto permite a React optimizar la actualización del DOM, actualizando solo las partes que han cambiado en cada momento, mejorando significativamente el rendimiento.
- **JSX**: introduce una sintaxis que permite escribir la estructura del componente de UI en un código que se asemeja a HTML dentro de archivos JavaScript. Esto mejora la legibilidad del código y facilita el desarrollo.
- **One-way data flow**: React enfatiza un flujo de datos unidireccional, lo que facilita el rastreo de cambios a lo largo de la aplicación y mejora la predictibilidad y la facilidad de depuración.
- **Hooks**: son funciones que permiten a los componentes funcionales tener estado y acceder a características del ciclo de vida de React sin necesidad de convertirlos en clases. Los Hooks más comunes son useState y useEffect, que permiten una gestión del estado y efectos secundarios más sencilla.
- **Ecosistema extenso**: existe un amplio ecosistema de herramientas, bibliotecas y frameworks construidos alrededor de React, como Redux para la gestión del estado, React Router para la navegación y Next.js para la generación de sitios estáticos y aplicaciones backend.

React se utilizó para el desarrollo del frontend de BT Studio, facilitando la construcción de una interfaz compleja mediante la división eficiente en componentes funcionales interconectados. Los componentes de React incluyen elementos de renderizado gráfico, de control y de comunicación.

Projectstorm React Diagrams

Como se detallará en la capítulo 4, el elemento más importante de la interfaz de BT Studio es el editor de árboles de comportamiento. Es por ello que fue necesario elegir una libería de creación de diagramas lo suficientemente versátil y extensible.

La librería elegida fue **react-diagrams** de ProjectStorm[7], enfocada en la creación y manipulación de diagramas complejos y altamente interactivos dentro de aplicaciones web. Concretamente se utilizó la versión 7.0.3.

[7] https://github.com/projectstorm/react-diagrams

Las características que la hacen especialmente adecuada para este caso de uso son:

- **Personalización avanzada**: ofrece una API rica y flexible que permite a los desarrolladores personalizar componentes, estilos, comportamientos y eventos de los diagramas para ajustarse a necesidades específicas.
- **Modelo de objetos extensible**: la base de la librería es su modelo de objetos, que facilita la definición de nodos, enlaces y puertos personalizados, así como la lógica para sus interacciones e interconexiones.
- **Interactividad**: Soporta interacciones del usuario como arrastrar y soltar, selección, zoom y más, lo que produce diagramas visualmente atractivos y de gran usabilidad.
- **Serialización y deserialización**: los diagramas pueden ser serializados a un formato de datos (como JSON), permitiendo su almacenamiento, transferencia y reconstrucción posterior. Esta función es clave para la generación de código ejecutable a partir de los diagramas.

3.2.4. Django

Django[8] es un entorno de desarrollo web de código abierto, gratuito y escrito en Python. Proporciona una estructura de organización específica junto a diferentes herramientas que ayudan a resolver tareas típicas de desarrollo web, facilitando la creación aplicaciones web complejas y robustas. Para el desarrollo de BT Studio se ha utilizado la última versión LTS, Django 4.2, lo cual garantiza actualizaciones hasta 2025.

Las características más importantes de django para el desarrollo de BT Studio son:

- **Rapidez**: fue diseñado para crear aplicaciones web lo más rápido posible. Esto implica que tiene un diseño claro, bien estructurado y con una gran variedad de funcionalidades y herramientas adicionales.
- **Seguridad**: incluye herramientas para evitar los problemas más comunes de seguridad, como son los ataques XSS, el CSRF o la inyección SQL. Además, tiene un soporte completo para el protocolo HTTPS. Estas funcionalidades son importantes para la última fase del trabajo, la integración en Unibotics.

[8] https://www.djangoproject.com/

- **Escalable**: una vez implementada una funcionalidad en Django, esta puede ser usada desde el nivel de prototipo hasta el despliegue en aplicación web con miles de usuarios con cambios mínimos. Esto lo consigue mediante herramientas como cachés o la división de bases de datos en particiones.
- **Plantillas**: son archivos HTML enriquecidos con marcadores y etiquetas especiales de Django, que permiten la generación dinámica del contenido final a mostrar.
- **Patrón modelo-vista-controlador**: es un patrón de diseño de software muy utilizado para la implementación de interfaces y su correspondiente lógica de control. Promueve una separación entre la lógica del un software y las visualizaciones asociadas a este.
  - *Modelo*: define los datos necesarios para una determinada aplicación.
  - *Vista*: define como se muestran los datos de la aplicación, así como la interfaz para realizar operaciones sobre los mismos.
  - *Controlador*: contiene la lógica para actualizar los modelos y las vistas en respuesta a las acciones de los usuarios.
- **Fácil mantenimiento**: al utilizar el patrón MVC y plantillas, Django promueve el desarollo de código modular y fácilmente reutilizable. Esto además encaja muy bien con un método de desarrollo de software libre.

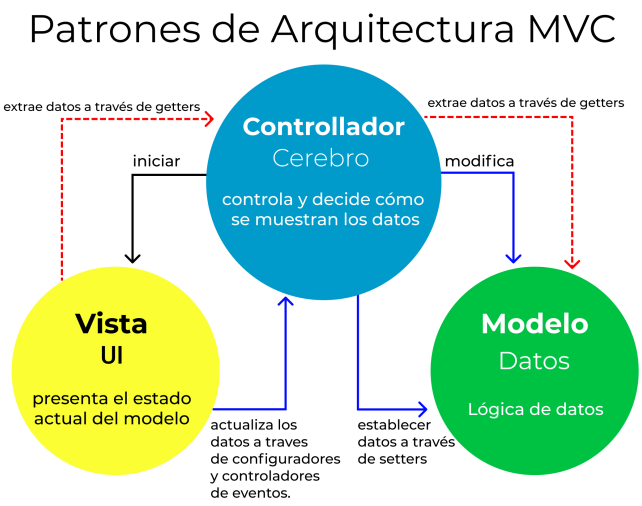


Figura 3.2: Esquema Modelo-Vista-Controlador

3.2.5. WebSocket

WebSocket[9] es un protocolo de comunicación full-duplex sobre una única conexión TCP. Este protocolo facilita la interacción en tiempo real entre el cliente y el servidor, siendo fundamental para aplicaciones que requieren una comunicación bidireccional persistente y con baja latencia. En BT Studio, son utilizados para la comunicación entre la interfaz del IDE y el backend de ejecución de aplicaciones robóticas.

Las características más importantes de este protocolo son:

- **Comunicación Full-Duplex:** WebSockets permite que tanto el cliente como el servidor envíen datos simultáneamente y en cualquier momento. Mejoran la interactividad y el rendimiento de las aplicaciones web al eliminar la necesidad de realizar múltiples conexiones HTTP para la comunicación bidireccional.
- **Sesión persistente:** a diferencia del modelo de solicitud-respuesta utilizado en HTTP, WebSockets establece una conexión persistente que permanece abierta, permitiendo el intercambio de mensajes hasta que la conexión es cerrada por el cliente o el servidor.
- **Menor sobrecarga:** tras el establecimiento inicial de la conexión, la sobrecarga de datos es significativamente menor en comparación con HTTP, ya que los encabezados no necesitan ser enviados con cada mensaje. Esto reduce significativamente el uso de ancho de banda y el tiempo de respuesta.
- **Compatibilidad con navegadores:** todos los navegadores modernos soportan el protocolo, lo que permite a los desarrolladores implementar aplicaciones en tiempo real sin la necesidad de recurrir a soluciones alternativas menos eficientes como polling o long-polling.
- **Facilidad de uso:** la API de WebSockets es relativamente simple y fácil de usar, permitiendo a los desarrolladores establecer una comunicación bidireccional cliente-servidor con muy pocas líneas de código.

[9] https://www.rfc-editor.org/rfc/rfc6455

3.2.6. VNC en la web

Virtual Network Computing[10] es una tecnología que permite la visualización y control remoto de una máquina que actúa como servidor a través de otra máquina que actúa como cliente.

En el contexto web, VNC se utiliza para transmitir la interfaz gráfica de un sistema a través de un navegador web, lo que facilita el acceso remoto sin la necesidad de software adicional más allá de un navegador estándar. En BT Studio, el frontend del IDE incluye un cliente VNC, que se conecta a un servidor VNC creado por el ejecutor dockerizado para visualizar la simulación de la aplicación robótica.

Las características fundamentales de VNC en una implementación web incluyen:

- **Navegador como cliente VNC:** utilizando tecnologías web como HTML5 y WebSockets, los navegadores pueden implementar el protocolo VNC.Esto permito a los usuario controlar sistemas remotos directamente desde el navegador sin instalar clientes de VNC tradicionales. Al depender de tecnologías web estándar, las soluciones VNC basadas en web son compatibles con cualquier sistema operativo que pueda ejecutar un navegador moderno
- **Seguridad:** las implementaciones web de VNC pueden utilizar protocolos de seguridad como TLS/SSL para cifrar la conexión entre el navegador y el servidor VNC.
- **Interactividad:** aprovechando WebSockets para una comunicación bidireccional eficiente, las implementaciones web de VNC pueden ofrecen una experiencia interactiva con apenas latencia. Esto es crucial en BT Studio, donde se usan para visualizar una simulación robótica, por naturaleza muy reactiva.

[10] https://quentinsf.com/publications/virtual-network-computing/vnc-ieee.pdf

3.3. Herramientas de desarrollo robótico

3.3.1. ROS 2

ROS 2[11] (Robot Operating System 2) es un conjunto de librerías y herramientas para el desarrollo de aplicaciones robóticas, incluyendo una extensa colección de drivers y algoritmos estado del arte. Se puede consultar su diseño en profundidad en [3]. La versión utilizada en este TFG es Humble Hawksbill.

Las aplicaciones generadas por BT Studio son aplicaciones ROS 2, con un nodo capaz de interaccionar mediante topics con los drivers de los sensores y actuadores del robot.

Características de diseño

- **Arquitectura distribuida:** utiliza un modelo de comunicación basado en DDS (Data Distribution Service) para el manejo de la comunicación entre los componentes del sistema. Esto permite además el uso de distintas políticas de calidades de servicio, que garantizan unos requisitos concretos a las comunicaciones.
- **Flexibilidad en la comunicación:** soporta diferentes patrones de comunicación, incluyendo publicación/suscripción, servicios y acciones, lo que permite una amplia gama de interacciones entre los componentes del sistema.
- **API homogénea en Python y C++**: la funcionalidad básica de ROS 2 se encuentra en la librería rcl, escrita en C. ROS 2 proporciona APIs en lenguajes de alto nivel a través de librerías clientes, que permiten a los usuarios acceder a los componentes del sistema. Estas librerías clientes son `rclcpp` para C++ y `rclpy` para Python, y pueden se usadas de manera simultánea para distintos componentes dentro de una aplicación robótica.
- **Herramientas de desarrollo y depuración:** proporciona un conjunto robusto de herramientas para la depuración y visualización de sistemas (como Rviz2), lo que facilita el desarrollo y la prueba de aplicaciones robóticas complejas.
- **Soporte multiplataforma:** ROS 2 es compatible con múltiples sistemas operativos, incluyendo Linux, MacOS, Windows y otros sistemas operativos

[11] https://docs.ros.org/en/humble/index.html

en tiempo real, ampliando su aplicabilidad en diversos entornos robóticos.

Características funcionales básicas

ROS 2 es un middleware basado en un mecanismo de publicación/subscripción anónimo que permite el paso de mensajes (con tipado fuerte) entre diversos procesos. El núcleo funcional del sistema es el grafo de computación de ROS, que define la estructura de los diversos nodos del sistema y la comunicación entre ellos.

Los componentes funcionales básicos de ROS 2 son los siguientes:

- **Nodos**: son la unidad de computación básica del grafo de ROS. Usan una librería cliente para comunicarse con otros nodos que pueden estar en el mismo proceso, en otros o incluso en una máquina diferente.
- **Interfaces:** en ROS 2, los nodos pueden interaccionar con el grafo computación mediante topics, servicios o acciones. Para la descripción de estas interfaces se usa un lenguaje de definición de interfaces simplificado (IDL), que facilita la generación de su código fuente en diferentes lenguajes.
- **Descubrimiento de nodos**: es llevado acabo de manera automática por el *rmw*, el middleware encargado de las comunicaciones en ROS. Cuando un nodo arranca, anuncia su presencia a todos los nodos dentro de su mismo dominio de ROS. Este anuncio se repite de manera frecuente para permitir el descubrimiento por parte de nodos recién incorporados al grafo. De igual manera, los nodos anuncian cuando se desconectan.
- **Topics**: permiten conectar productores de datos con consumidores a través de un canal de nombre común. El rmw de ROS 2 permite distintas calidades de servicio para la recepción de estos mensajes.
- **Servicios:** son llamadas a procedimientos remotos, que un nodo cliente pueden utilizar para invocar la ejecución de tareas y obtener un resultado por parte de un nodo servidor. Generalmente se utilizan para computaciones livianas y son bloqueantes para el nodo cliente. ROS 2 también soporta servicios no bloqueantes mediante los futuros de C++.
- **Acciones:** son llamadas a procedimientos remotos de larga duración. Las acciones proporcionan feedback mientras se están ejecutando y pueden ser canceladas en cualquier momento.

- **Parámetros:** permiten la configuración de los nodos cuando se inician y en tiempo de ejecución sin cambios en el código. En ROS 2, los parámetros están asociados a cada nodo y estos deben declarar todos los parámetros que aceptan junto con su tipo.
- **Launch system:** tiene como objetivo automatizar el lanzamiento de muchos nodos (como es típico en una aplicación robótica) desde un solo comando. Permite definir cómo y cuándo ejecutar cada programa con una sintaxis común de ROS que facilita su reutilización.

3.3.2. Simulador Gazebo

Gazebo[12] es un simulador 3D de código abierto utilizado ampliamente en la investigación y desarrollo de sistemas robóticos y de inteligencia artificial[4]. Ofrece la capacidad de simular con precisión el funcionamiento de robots en entornos complejos y dinámicos[5]. En BT Studio, es el entorno de simulación incluido en el ejecutor dockerizado. La versión utilizada para este TFG es Gazebo 11.

Sus características más importantes son:

- **Simulación física detallada:** Gazebo utiliza motores de física avanzados como ODE (Open Dynamics Engine), Bullet, Simbody o DART para proporcionar una simulación realista de fuerzas, colisiones y materiales. Esto permite a los usuarios robots y sistemas de control en entornos seguros y controlados antes de la implementación real. Esta funcionalidad se implementa el el servidor de Gazebo, conocido como *gzserver*.
- **Entornos ricos y personalizables:** los usuarios pueden crear entornos detallados con una amplia variedad de objetos, texturas y condiciones de iluminación. Estos entornos generalmente se codifican en el formato SDF[13], basado en XML.
- **Modelado de sensores y actuadores:** Gazebo proporciona modelos de sensores y actuadores que imitan de manera realista el comportamiento y las limitaciones de los modelos reales. Esto incluye cámaras, sensores LIDAR, GPS, y actuadores como motores y servos.
- **Interfaz gráfica y herramientas de usuario:** a través de su interfaz gráfica,

[12] https://gazebosim.org/home
[13] http://sdformat.org/

Gazebo permite la visualización y manipulación de simulaciones, incluyendo la capacidad de controlar el tiempo de simulación, visualizar datos de sensores y editar en tiempo real las propiedades de los objetos y entornos. Todas herramientas se incluyen en el cliente gráfico de Gazebo, conocido como *gzclient*.

- **Integración con ROS 2:** Gazebo se integra totalmente con ROS 2, permitiendo a los usuarios aprovechar las herramientas para la comunicación entre nodos, control y depuración. El software de ROS 2 puede interaccionar con la simulación de Gazebo como si estuviera operando en hardware real.

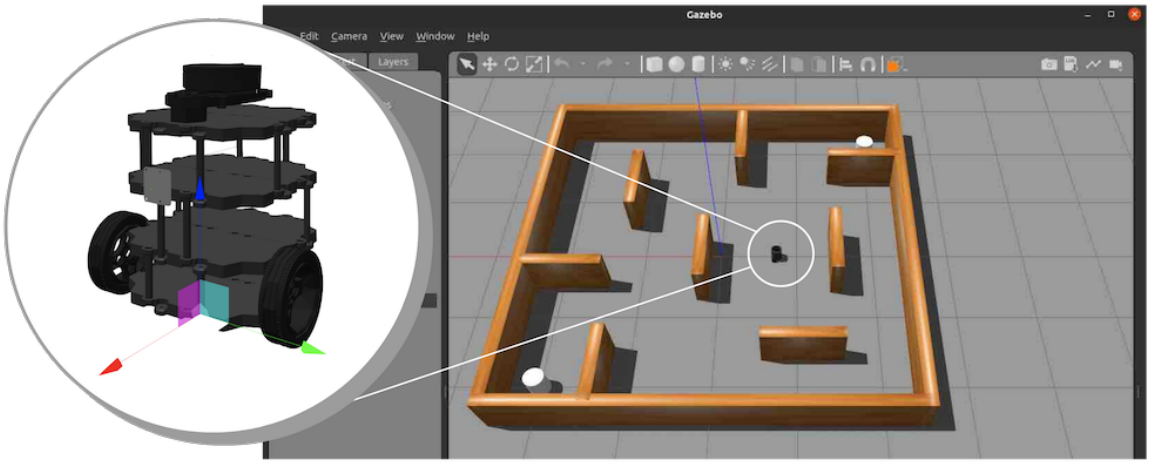


Figura 3.3: Cliente gráfico de Gazebo

3.3.3. Árboles de comportamiento

Definición

Los Árboles de Comportamiento[6] constituyen un marco para organizar la forma en la que un agente autónomo, ya sea un robot o una entidad virtual en un videojuego, alterna entre diferentes tareas. El requisito fundamental para modelar una determinada actividad con un BT es que esta pueda ser dividida en distintas sub-actividades reutilizables, que son denotadas como acciones o modos de control.

Esta metodología ofrece una manera altamente eficaz de construir sistemas complejos que son al mismo tiempo modulares y reactivos, facilitando la creación de sistemas autónomos capaces de adaptarse dinámicamente a cambios en el

entorno o en los objetivos de la tarea.

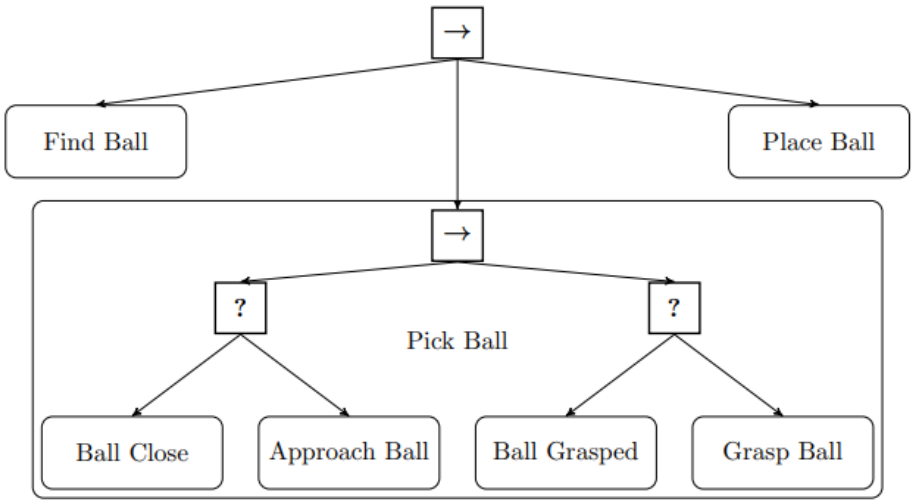


Figura 3.4: Ejemplo básico de árbol de comportamiento

Características

Concretamente, los árboles de comportamiento se pueden entender como un árbol dirigido que parte de un nodo raíz, donde los nodos internos reciben el nombre de nodos de control de flujo y los externos (nodos hoja) reciben el nombre de nodos de ejecución. El nodo raíz es el único nodo sin padres, el resto de nodos tiene exactamente uno. Los nodos de control de flujo tienen por lo menos un hijo, mientras que los nodos de ejecución no tienen ninguno.

Un árbol de comportamiento empieza su ejecución cuando el nodo raíz genera señales a una determinada frecuencia, llamadas *ticks*, que son propagadas hacia los hijos. Los ticks sirven como testigo: los nodos se ejecutan sí y sólo sí reciben ticks. Una vez reciben un tick, los hijos pueden devolver *Running* si están ejecutando una tarea, *Success* si ha conseguido su objetivo o *Failure* en caso contrario.

En la formulación clásica de los árboles de comportamientos, que es la utilizada en su mayor parte en este trabajo, existen cuatro categorías de nodos de control de flujo (*Sequence*, *Fallback*, *Parallel* y *Decorator*) y dos de nodos de ejecución (*Action* y *Condition*).

Funcionamiento de los nodos de control de flujo:

- **Sequence:** ejecuta el algoritmo de la figura 3.5, que corresponde a dirigir los ticks de manera secuencial a sus hijos de izquierda a derecha hasta que alguno devuelve *Running* o *Failure*. Cuando esto ocurre, devuelve

acordemente lo mismo a su padre. Sólo devuelve *Success* si y sólo si todos los hijos lo devuelven.

Algorithm 1: Pseudocode of a Sequence node with N children

```
for i ← 1 to N do
    childStatus ← Tick (child(i))
    if childStatus = Running then
        return Running
    else if childStatus = Failure then
        return Failure
return Success
```

Figura 3.5: Algoritmo de un nodo tipo *Sequence*

- **Fallback:** ejecuta el algoritmo de la figura 3.6, que corresponde a dirigir los ticks de manera secuencial a sus hijos de izquierda a derecha hasta que alguno devuelve *Running* o *Success*. Cuando esto ocurre, devuelve acordemente lo mismo a su padre. Sólo devuelve *Failure* si y sólo sí todos los hijos lo devuelven.

Algorithm 2: Pseudocode of a Fallback node with N children

```
for i ← 1 to N do
    childStatus ← Tick (child(i))
    if childStatus = Running then
        return Running
    else if childStatus = Success then
        return Success
return Failure
```

Figura 3.6: Algoritmo de un nodo tipo *Fallback*

- **Parallel:** ejecuta el algoritmo de la figura 3.7, que corresponde a dirigir los ticks a todos sus hijos a la vez y devolver *Success* si M hijos devuelven *Success*, *Failure* si N - M + 1 hijos devuelven *Failure* y *Running* en cualquier otro caso. N es el número de hijos y M un umbral definido por el usuario.

Algorithm 3: Pseudocode of a Parallel node with N children and success threshold M

```
for i ← 1 to N do
    childStatus(i) ← Tick (child(i))
if Σ_{i:childStatus(i)=Success} 1 ≥ M then
    return Success
else if Σ_{i:childStatus(i)=Failure} 1 > N − M then
    return Failure
return Running
```

Figura 3.7: Algoritmo de un nodo tipo *Parallel*

- **Decorator:** son nodos de un solo hijo que propagan el tick y manipulan el estado de salida del hijo en función de reglas predefinidas. Por ejemplo, un decorador de tipo *invert* invierte la salida del hijo.

Funcionamiento de los nodos de ejecución:

- **Action:** al recibir un tick, ejecutan un comando que produce un efecto. Su valor de retorno depende del estado del comando: *success* si ha tenido éxito, *failure* en caso contrario y *running* mientras está en ejecución.
- **Condition:** al recibir un tick, comprueba una proposición y devuelve *success* o *failure* en función de su cumplimiento. Estos nodos nunca devuelven *running*.

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

Tabla 3.1: Funcionamiento de los nodos de un árbol de comportamiento tradicional

Ventajas

Los árboles de comportamiento facilitan el modelado de aplicaciones complejas y heterogéneas mediante componentes funcionales interconectados bajo reglas definidas[7]. Esto proporciona ventajas significativas:

- *Jerarquía intrínseca:* permiten la creación de comportamientos altamente complejos mediante la inclusión de árboles completos como subestructuras dentro de un árbol mayor.
- *Semántica en su representación gráfica:* los componentes pueden ser visualizados y enlazados en un diagrama, cuyo significado resulta intuitivo con solo comprender las reglas de los distintos tipos de componentes.
- *Alta expresividad:* los nodos de control de flujo ofrecen un lenguaje más amplio que otras arquitecturas para la descripción de comportamientos complejos. Además, la biblioteca de nodos puede ser extendida fácilmente.
- *Centralización de la lógica de aplicación:* Concentran la lógica de la aplicación en una única ubicación, la estructura del árbol, permitiendo conceptualizar

las aplicaciones robóticas como una orquesta de nodos guiada por el árbol de comportamiento.

Los árboles de comportamiento son herramientas óptimas para el desarrollo de aplicaciones robóticas, por lo que su uso está cada vez más extendido dentro del sector [8], reemplazando a las tradicionales máquinas de estados [9].

3.3.4. BT.cpp

BT.cpp[14] es una librería en C++ que facilita el desarrollo de árboles de comportamiento, destacando por su flexibilidad, facilidad de uso y rendimiento elevado. Estas cualidades la han establecido como el estándar para la programación de BTs en la comunidad robótica. La versión utilizada en BT Studio es la 4.5.

Características

- **Construcción de árboles en tiempo de ejecución:** los árboles se codifican con un lenguaje de marcado basado en XML, lo cual posibilita una clara distinción entre la definición del árbol y la implementación de las acciones. La biblioteca provee funciones para leer este XML y convertirlo en un objeto ejecutable.
- **Colección de nodos extendida:** implementa un número mayor de nodos de control de flujo, como por ejemplo, las *ReactiveSequence*. Esto permite definir comportamientos más complejos.
- **Blackboard y puertos:** la *blackboard* es una memoria compartida que actúa como un diccionario de datos accesible por todos los nodos del árbol. Permite la comunicación y el intercambio de información entre nodos mediante puertos, que se conectan utilizando la misma clave del *Blackboard*, permitiendo así una gestión de datos eficiente y dinámica dentro del árbol.
- **Soporte para acciones asíncronas:** permite la ejecución de acciones en paralelo. Esto es útil por ejemplo para la implementación de rutinas no bloqueantes.
- **Amplia adopción:** es el estándar de facto para la generación de árboles de comportamiento en ROS y ROS 2, lo cual garantiza la facilidad de uso, amplia documentación y ejemplos disponibles.

[14] https://github.com/BehaviorTree/BehaviorTree.CPP

```xml
<root BTCPP_format="4">
    <BehaviorTree ID="MainTree">
       <Sequence name="root_sequence">
           <SaySomething   name="action_hello" message="Hello"/>
           <OpenGripper    name="open_gripper"/>
           <ApproachObject name="approach_object"/>
           <CloseGripper   name="close_gripper"/>
       </Sequence>
    </BehaviorTree>
</root>
```

Extracto de código 3.1: Ejemplo de definición de árbol mediante XML

Type of ControlNode	Child returns FAILURE	Child returns RUNNING
Sequence	Restart	Tick again
ReactiveSequence	Restart	Restart
SequenceWithMemory	Tick again	Tick again

Figura 3.8: Comportamiento de las secuencias soportadas por BT.cpp

Type of ControlNode	Child returns RUNNING
Fallback	Tick again
ReactiveFallback	Restart

Figura 3.9: Comportamiento de los *fallbacks* soportados por BT.cpp

Uso en BT Studio

Debido a requisitos funcionales, como el uso de Django y entornos de ejecución dockerizados, BT Studio está implementado en Python, lo cual imposibilita el uso directo de BT.cpp. Sin embargo, esta librería proporciona a BT Studio dos características fundamentales:

- **Lenguaje de definición de árboles:** los árboles se definen con la sintaxis y estructura propias del lenguaje XML de BT.cpp. Esto implica que los usuarios podrán crear árboles en BT Studio usando sus conocimientos de BT.cpp e incluso, importar árboles ya creados con esta herramienta.
- **Creación de árboles en tiempo de ejecución:** BT Studio implementa en Python numerosa funcionalidad de BT.cpp, como el generador de árboles desde XML o el uso de los puertos. El objetivo es proveer al usuario de la misma API que BT.cpp en Python.

3.3.5. PyTrees

PyTrees[15] es una librería para el desarrollo de árboles de comportamiento en Python, con integración completa con ROS 2. Durante el desarrollo de este trabajo, se empleó la versión 2.2.

Características

- **Enfocado en el desarrollo rápido:** implementada en Python para permitir una curva de aprendizaje más corta y un ciclo de desarrollo más rápido. Su principal objetivo es posibilitar el desarrollo de BTs por parte de un conjunto de desarrolladores más amplio sin conocimientos explícitos en ingeniería de control.
- **Aplicaciones de tamaño medio:** la librería está diseñada para manejar aplicaciones de tamaño en el orden de cientos de comportamientos. La librería prima la facilidad de uso y el pragmatismo de implementación sobre la eficiencia y la escalabilidad. Algunas consecuencias de estas decisiones son la falta de soporte para la ejecución paralelizada de nodos del árbol o un soporte muy reducido para compartir datos entre nodos.
- **Sin soporte para tiempo real:** la librería no contiene funcionalidad para garantizar restricciones temporales en la ejecución de los nodos. Esta funcionalidad apenas tiene soporte en Python.
- **No hay lenguaje de codificación de árboles**: en PyTrees, las acciones se implementan como clases de Python que heredan de una clase nodo base. Para definir la estructura del árbol, es necesario escribir código específico que

[15] https://github.com/splintered-reality/py_trees

una estas clases de la manera adecuada. Esto empeora significativamente la modularidad y explicabilidad de los árboles, siendo necesario cambiar el código fuente para cambiar el comportamiento de la aplicación.

- **Blackboard como singleton**: un singleton es un patrón de diseño muy habitual, cuya función es garantizar que sólo existe una instancia de una clase determinada. En la librería, la *blackboard* no es más que una instancia única compartida entre los hijos donde estos pueden leer y escribir. Dado que la librería no soporta paralelismo, no existen problemas de concurrencia.

Uso en BT Studio

Estas características hacen a PyTrees una opción funcionalmente inferior a BT.cpp, con menor escalabilidad y rendimiento. Sin embargo, fue escogida como motor de ejecución de árboles de comportamiento de BT Studio por tres motivos principales:

- **Gran flexibilidad de desarrollo:** la librería tan solo aporta una estructura para la definición de acciones, una blackboard básica y herramientas de depuración. Esto permite extender su funcionalidad fácilmente para soportar características de BT.cpp, principalmente el lenguaje XML de definición de árboles.
- **No necesita compilación:** si bien para la ejecución en un entorno local los lenguajes compilados como C++ proporcionan mejor rendimiento, el proceso de compilación en entornos dockerizados puede conllevar grandes latencias para el usuario. Concretamente, las aplicaciones de BT Studio están pensadas para ser ejecutadas con el RADI de RoboticsAcademy, que sólo soporta actualmente aplicaciones ROS escritas en Python.
- **Capacidades suficientes:** no se prevé el uso de la plataforma en aplicaciones que necesiten más de decenas de árboles de comportamiento. El rendimiento de PyTrees en estos casos es adecuado.

Biblioteca de nodos

PyTrees sólo implementa los nodos de control de flujo de un árbol de comportamiento básico, divididos en su terminología propia en dos grupos, *composites* y *decorators*.

Los *composites* son los encargados de gestionar el movimiento del *tick* y la librería define tres: *Selector* (Fallback), *Sequence* y *Parallel*. Su funcionamiento es idéntico al estudiado en la sección teórica sobre árboles de comportamiento y al de BT.cpp.

Por otro lado, la librería define un gran número de *decorators*, algunos comunes con BT.cpp y otros propios. En BT Studio, sólo se soportan aquellos presentes en BT.cpp, que se consideran suficientes para expresar una gran variedad de comportamientos. De igual manera, la funcionalidad de PyTrees debe ser extendida para soportar todos las *Sequences* y *Fallbacks* de BT.cpp.

Definición de nodos de ejecución

En PyTrees, los nodos de ejecución reciben el nombre de *Behaviour* y no existe una separación explícita entre acciones y condiciones. Todos los comportamientos se implementan como una subclase de una clase base, donde se definen los siguientes métodos:

- **__init__(name)**: inicialización mínima de la clase con la información necesaria para la introspección de los árboles mediante herramientas de depuración offline.
- **setup()**: método para inicialización diferida que se debe llamar manualmente o a través de métodos de la librería que configuran el comportamiento junto con sus descendientes. Adecuado para inicialización de hardware, middleware o cualquier configuración necesaria que no se desee en el constructor por interferir con la representación del árbol en gráficos o validaciones.
- **initialise()**: invocado siempre que el estado anterior del comportamiento no sea *RUNNING*. Aquí se realiza cualquier preparación requerida antes de comenzar la ejecución de una iteración del comportamiento.
- **update()**: este método se llama en cada tick del comportamiento. Dentro de este, se debe implementar la lógica principal del comportamiento, incluyendo decisiones, monitoreo o cualquier acción no bloqueante. Debe retornar un estado de *py_trees.common.Status*, que puede ser *RUNNING*, *SUCCESS*, o *FAILURE*, basado en el resultado de la lógica implementada.
- **terminate()**: se llama cuando el comportamiento cambia a un estado no activo (*SUCCESS*, *FAILURE* o *INVALID*). Es útil para realizar limpieza o acciones finales al terminar o interrumpir el comportamiento.

Cada uno de estos métodos proporciona un punto de intervención en el ciclo de vida del comportamiento, permitiendo una implementación detallada y controlada de las acciones y condiciones dentro del árbol de comportamiento. Todas los nodos de ejecución de BT Studio se implementarán de esta manera.

3.3.6. Contenedores Docker

Docker[16] es una plataforma de contenedorización que permite empaquetar una aplicación y sus dependencias en un contenedor virtual que puede ejecutarse en cualquier sistema operativo que soporte docker. Los contenedores pueden ser entendidos como máquinas virtuales ligeras, ofreciendo una capa de abstracción y visualización a nivel de sistema operativo. Las características principales de Docker incluyen:

- **Portabilidad**: gracias a los contenedores, las aplicaciones pueden ejecutarse de manera consistente en diversos entornos, desde un desarrollo local hasta la producción en la nube.
- **Aislamiento**: cada contenedor opera de manera aislada, lo que mejora la seguridad y reduce las interferencias entre aplicaciones.
- **Eficiencia**: Docker utiliza los recursos del sistema operativo subyacente de manera más eficiente que las máquinas virtuales tradicionales, lo que resulta en un mejor rendimiento y menor consumo de recursos.
- **Gestión de imágenes**: Docker utiliza imágenes para crear contenedores. Estas imágenes pueden tener control de versiones, ser almacenadas en repositorios y compartidas, facilitando la colaboración y el despliegue.
- **Dockerfile**: un archivo de texto que contiene todas las órdenes necesarias para construir una imagen Docker. Esto permite automatizar la creación de imágenes de manera reproducible.

La adopción de Docker en el desarrollo y despliegue de aplicaciones facilita la integración continua y la entrega continua (CI/CD), mejorando así los ciclos de desarrollo y permitiendo una mayor agilidad. En este trabajo, el uso de docker ha sido mayoritariamente para la creación y despliegue de contenedores de entornos de ejecución de aplicaciones robóticas. El uso de Docker para este propósito se encuentra en auge dentro de la industria [10].

[16] https://www.docker.com/

3.3.7. Unibotics

Unibotics[17] es una plataforma web para la programación de aplicaciones robóticas[11]. Proporciona un frontend web que permite la edición y ejecución de aplicaciones robóticas escritas en código Python desde el navegador.

Para conseguir esto, proporciona un un backend robótico basado en ROS, llamado RADI[18]. Está definido como una imagen docker, que permite lanzar el backend robótico como un contenedor docker con todas las dependencias y herramientas previamente instaladas. Dentro de este contenedor, se ejecuta un programa *manager*, encargado de comunicarse con el frontend mediante websockets y recibir distintos comandos, como lanzar un escenario robótico simulado o ejecutar el código del usuario.

Actualmente, la aplicación predominante en Unibotics es RoboticsAcademy[12], cuyo objetivo es el aprendizaje de conceptos y algoritmos robóticos a través de la resolución de diversos ejercicios. Ofrece una capa de abstracción para que los usuarios sólo tengan que preocuparse del desarrollo algorítmico, sin la complejidad inherente de un middleware como ROS.

RoboticsAcademy puede ser usada de manera offline, a través del servidor incluido en el propio contenedor docker que el usuario descarga en su máquina. Al ser integrada en Unibotics, se añade una base de datos de usuarios, lo que posibilita a los usuarios guardar sus proyectos y monitorizar su progreso. La única diferencia de arquitectura en este caso es que la página web se sirve desde un backend preparado para producción alojado en la nube de Amazon. La comunicación con los componentes del backend robótico se realiza de la misma forma.

[17] https://unibotics.org/

[18] https://hub.docker.com/repository/docker/jderobot/robotics-academy/general

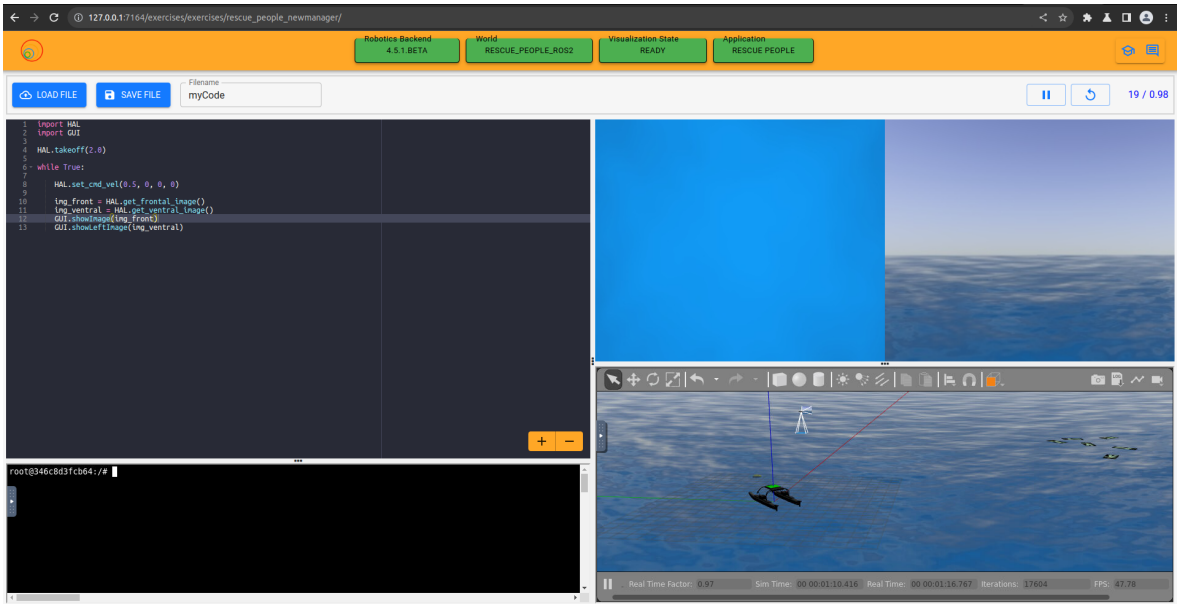


Figura 3.10: Aplicación robótica básica ejecutando en RoboticsAcademy

JdeRobot[19], la asociación de software libre que mantiene ambos proyectos, está actualmente desarrollando nuevas aplicaciones para soportar en la plataforma la creación y ejecución de aplicaciones robóticas genéricas. Estas aplicaciones serán multifichero y estarán organizadas en distintos paradigmas: aplicación distribuida, bloques visuales o árboles de comportamiento. Al integrarse con Unibotics, BT Studio pretende implementar este último paradigma.

Para permitir la ejecución de aplicaciones genéricas, el programa *manager* debe ser actualizado y generalizado a los distintos casos de uso. Para la integración de BT Studio con Unibotics, se ha estudiado en profundidad el funcionamiento de este programa y se ha contribuido a su desarrollo a través de commits en sus correspondientes repositorios.

El RADI de RoboticsAcademy contiene una colección de universos robóticos (mundos simulados, ficheros *launcher* de ROS, configuraciones específicas para cierto robot, etc), almacenada en el repositorio RoboticsInfraestructure[20] y preinstalados en el RADI. Al ser compatible con el RADI de RoboticsAcademy, BT Studio hará uso de esta colección.

Como se ha detallado en su correspondiente sección, uno de los objetivos finales de este TFG es la ejecución de aplicaciones de BT Studio usando este entorno dockerizado, tanto de manera offline como integrada en Unibotics.

[19] https://jderobot.github.io/
[20] https://github.com/JdeRobot/RoboticsInfrastructure

4. BT Studio

En este capítulo, se detallarán los principios de diseño que guían el desarrollo de BT Studio y sus diferentes componentes funcionales. Se hablará sobre el diseño y las consideraciones de implementación de cada uno de ellos.

4.1. Diseño

4.1.1. Fundamentos

Como se ha comentado en capítulos anteriores, el desarrollo de BT Studio sigue una mentalidad open source y está disponible para todo usuario interesado. Es por ello, que su diseño debe facilitar el entendimiento de la herramienta y la adaptabilidad a posibles casos de uso futuros.

Para garantizar estas características, el diseño de BT Studio está guiado por tres principios fundamentales, derivados de la definición de buena arquitectura de software proporcionada en la introducción:

- **Claridad:** cada componente del sistema tiene que tener una función concreta.
- **Modularidad:** los componentes deben ser lo más independientes posibles entre ellos. De manera complementaria, deben existir interfaces de comunicación estándar entre los mismos. La combinación de ambos requisitos dota de utilidad inherente a cada componente del sistema.
- **Extensibilidad:** facilidad para añadir nuevos componentes al sistema con el mínimo número de cambios en componentes ya existentes.

4.1.2. Componentes funcionales de la plataforma

El objetivo de BT Studio es proporcionar una interfaz web versátil y de fácil uso que permita a los usuarios la creación de aplicaciones robóticas complejas usando árboles de comportamiento. Para el cumplimiento de este objetivo, es necesario establecer las distintas funcionalidades que debe soportar la plataforma.

En primer lugar, BT Studio debe proporcionar un IDE web que sirva como interfaz única para el usuario. Desde este IDE, el usuario debe ser capaz de crear diferentes aplicaciones robóticas, guardarlas, editarlas y ejecutarlas.

En segundo lugar, la plataforma debe realizar un paso intermedio de traducción desde la defnición concreta de aplicaciones de BT Studio a código ejecutable. Este proceso debe ser totalmente transparente para el usuario, es decir, tras la creación de una aplicación, el usuario debe ser capaz de ejecutarla directamente desde el IDE sin la necesidad de realizar ninguna acción intermedia de configuración.

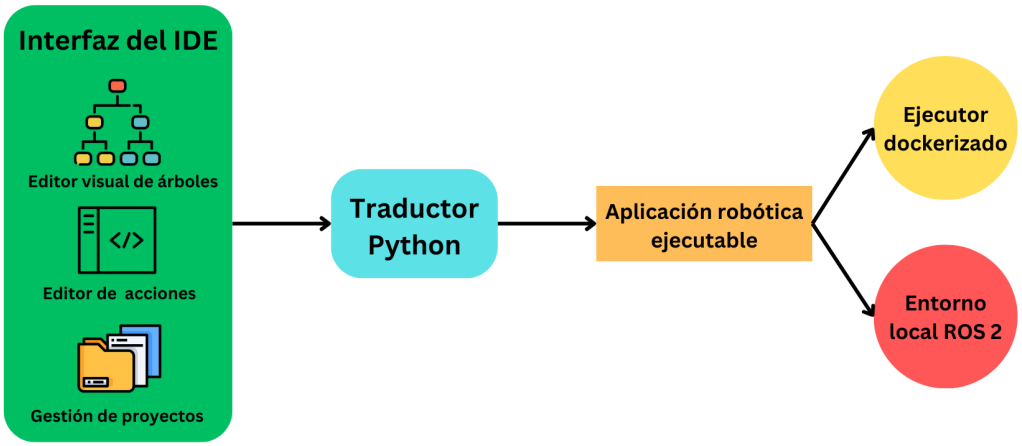


Figura 4.1: Proceso de generación de aplicaciones en BT Studio

Finalmente, BT Studio debe contener herramientas para visualizar y depurar la ejecución de la aplicación robótica desde el propio IDE. Esto requiere proporcionar al usuario una interfaz para iniciar, pausar y aplicar acciones sobre el entorno de ejecución. Este entorno de ejecución es el RADI de RoboticsAcademy, que contiene un entorno de simulación en ROS 2, con escenarios y configuraciones incluidas, tal y como se detalló el capítulo 3. Para la comunicación con el RAM (Robotics Application Manager) , el RADI define un protocolo de comunicación a través de WebSockets. BT Studio implementa este protocolo para realizar acciones en el entorno de ejecución desde la interfaz del IDE.

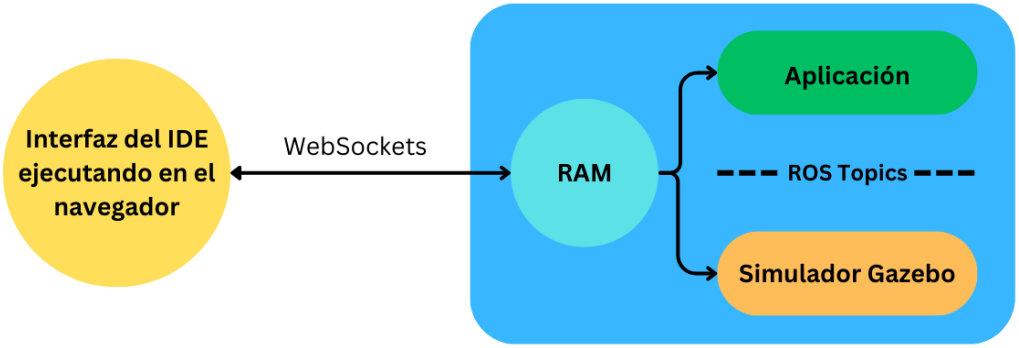


Figura 4.2: Ejecución de una aplicación robótica en un entorno dockerizado

4.1.3. Definición de aplicaciones robóticas en BT Studio

En BT Studio, una aplicación robótica basada en árboles de comportamiento está definida por dos ingredientes:

- **Acciones:** programada cada una en un fichero Python, siguiendo la estructura estándar para la definición de acciones que proporciona la librería PyTrees. Se escribirán mediante el editor de texto incluido en BT Studio. Cada acción es una clase que implementa distintas funciones propias de un nodo de ejecución de un árbol de comportamiento, como *update* (equivalente a *tick* en PyTrees) o *terminate*.

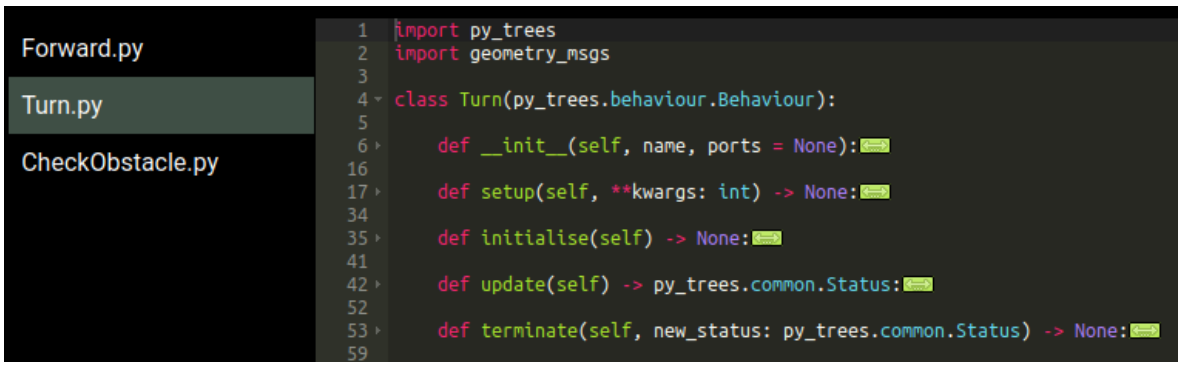


Figura 4.3: Estructura de una acción en BT Studio

- **Árbol de comportamiento:** definidos de manera gráfica mediante el editor visual de BT Studio, que permite definir gráficos personalizados con toda la semántica necesaria para definir un árbol de comportamiento. Estos gráficos pueden ser codificados en formato JSON. Los nodos de control de flujo soportados son aquellos definidos por la librería BT.cpp.

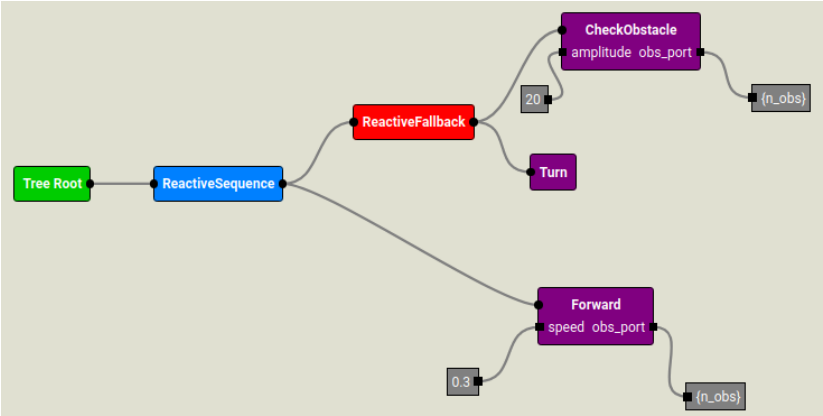


Figura 4.4: Árbol de comportamiento de ejemplo creado con BT Studio

Esta es la unidad básica de funcionamiento en BT Studio. Todas las aplicaciones que los usuarios crean, editan y ejecutan están definidas de esta manera. Todos los archivos asociados a una aplicación constituyen un proyecto.

4.2. Interfaz del IDE web

La funcionalidad del IDE web está dividida en dos componentes estándar de toda aplicación web: la funcionalidad del lado del servidor (backend) y la funcionalidad del lado del cliente (frontend). La primera implementa las capacidades funcionales del sistema mientras que la segunda proporciona una interfaz para usarlas. En BT Studio, ambas funcionalidades están implementando usando Django como *framework*.

El backend está implementado como una webapp, llamada *tree_api*, que proporciona un endpoint REST para solicitar la ejecución de diversas acciones en el servidor. Estas acciones están escritas en Python y constituyen la API de BT Studio: es la funcionalidad de la que disponen las implementaciones del frontend.

El frontend se implementa como otra webapp de Django, llamada *react_frontend* cuya función es servir el código de todos los componentes React de la interfaz, que son empaquetados previamente usando WebPack.

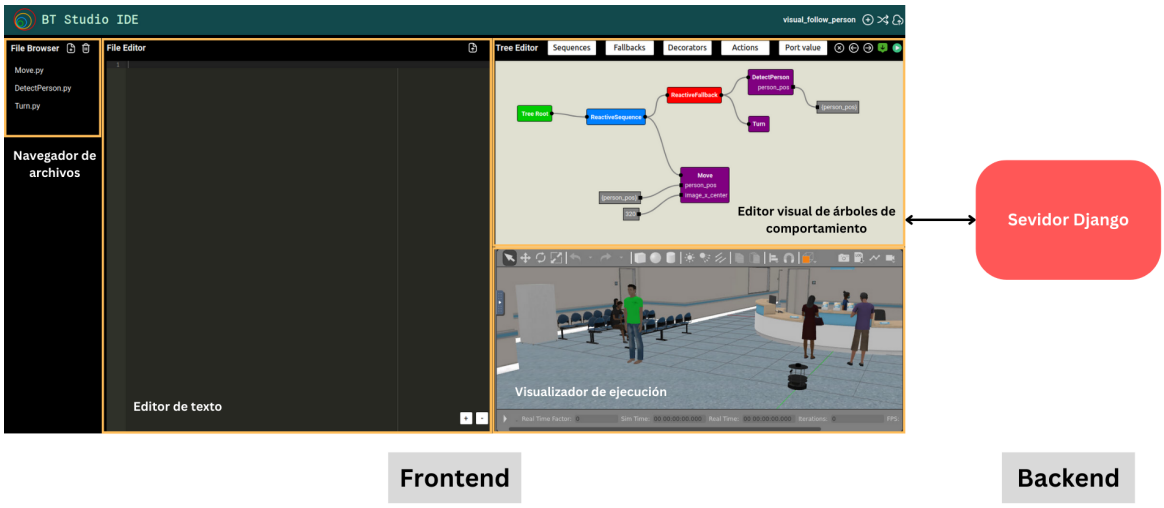


Figura 4.5: Componentes de la interfaz del IDE web

Para comunicarse con el backend, el frontend genera solicitudes GET y POST en función de las interacciones del usuario, utilizando la librería Axios. El backend recibe estas peticiones, ejecuta las funciones necesarias para cumplirlas y devuelve un estado de salida que el frontend puede usar para proporcionar feedback al usuario.

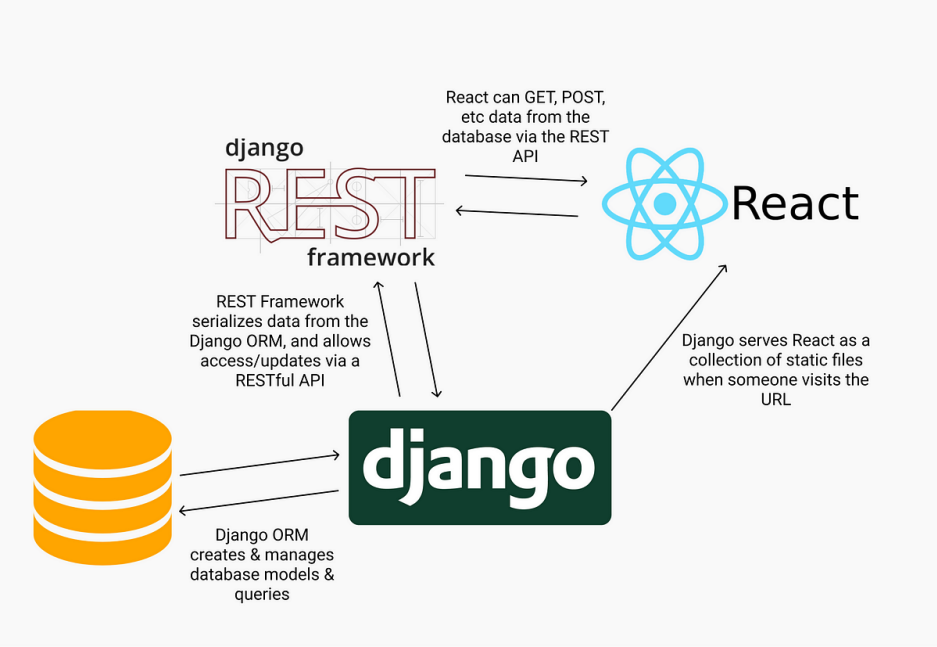


Figura 4.6: Configuración de Django en BT Studio

4.2.1. Backend

El backend proporciona una API para la gestión de proyectos, archivos y generación de aplicaciones robóticas ejecutables. A través de esta API, BT Studio es capaz de manejar una colección de proyectos del usuario.

Concretamente, la funcionalidad encargada de la generación de aplicaciones constituye un componente separado de BT Studio, el *traductor de aplicaciones*. Este componente puede ser usado de manera independiente, componiendo una aplicación robótica en Python a partir de una definición de árbol en XML y de la definición de las acciones en Python.

Sin embargo, para proporcionar esta funcionalidad al usuario, es necesario la interacción con el traductor a través del backend. La explicación detallada del funcionamiento y capacidades del mismo se abordarán en la siguiente sección, enfocándose aquí en describir cómo se accede y se utiliza desde el backend.

Las funciones que soporta el backend son las siguientes:

- **create_project:** permite crear un nuevo proyecto si este no existe, generando la estructura de directorios necesaria en el servidor.
- **get_project_list:** devuelve una lista de todos los proyectos existentes en el sistema, permitiendo a los usuarios ver los proyectos disponibles.
- **save_project_graph:** guarda la representación gráfica de diagrama del árbol en formato JSON, facilitando la persistencia del estado del proyecto.
- **get_project_graph:** recupera la representación gráfica del proyecto en formato JSON, permitiendo su edición o revisión.
- **get_file_list:** enumera todos los archivos dentro de un proyecto específico.
- **get_file:** permite la recuperación del contenido de un archivo específico, permitiendo la edición o revisión de los mismos.
- **create_file:** crea un nuevo archivo vacío dentro de un proyecto, permitiendo la adición de nuevos recursos al proyecto.
- **delete_file:** elimina un archivo específico de un proyecto.
- **save_file:** actualiza el contenido de un archivo específico, asegurando que los cambios realizados por los usuarios sean guardados correctamente.
- **generate_app:** devuelve un archivo zip con una aplicación ejecutable en ROS 2. En primer lugar, traduce la representación del árbol guardada en JSON a

XML. Posteriormente, ejecuta las funciones del traductor, que recibe el XML y los archivos de las acciones y genera el zip de la aplicación. Este puede ser descargado por el usuario y ejecutado offline o enviado al ejecutor dockerizado.

- **get_dockerized_app:** devuelve un archivo zip con una aplicación ejecutable por el RADI. En primer lugar, traduce la representación del árbol guardada en JSON a XML. Posteriormente, ejecuta las funciones del traductor, que recibe el XML y los archivos de las acciones y general un fichero XML autocontenido. A este XML se le añaden un script que sirve como *entrypoint* para su ejecución dockerizada y las dependencias necesarias para su ejecución.

4.2.2. Frontend

El frontend del IDE web es el encargado de proporcionar la interfaz de usuario necesaria para interaccionar con toda la funcionalidad proporcionada por el backend.

Cada sección del IDE web está implementada como un componente independiente de React, lo que permite separar las funciones de la interfaz de manera muy sencilla y escalable. En React, los componentes son piezas de código JSX que gestionan su propio estado y pueden recibir información de componentes padre a través de variables especiales conocidas como *props*, lo que permite mantener una información de estado homogénea en todos los componentes.

Los componentes pueden estar a su vez compuestos por otros componentes que implementan parte de su funcionalidad, formando una jerarquía. Los *props* se pasan exclusivamente de padre a hijos, por lo que esta jerarquía es un concepto fundamental del funcionamiento del frontend.

Además, en React es posible transmitir el *setter* de un estado (función de actualización proporcionada por el hook `useState`) como un *prop* a componentes hijos. Esta técnica posibilita que un componente hijo tenga la capacidad de modificar directamente el estado de su componente padre. Este método ayuda a mantener la modularidad y reusabilidad de los componentes, ya que facilita que los cambios de estado efectuados por un componente hijo no solo actualicen al componente padre, sino que también se reflejen en otros componentes hermanos que dependan de dicho estado. De esta manera, se mantiene una estructura de componentes clara y eficiente, evitando la necesidad de implementar lógicas más complejas para la comunicación entre componentes.

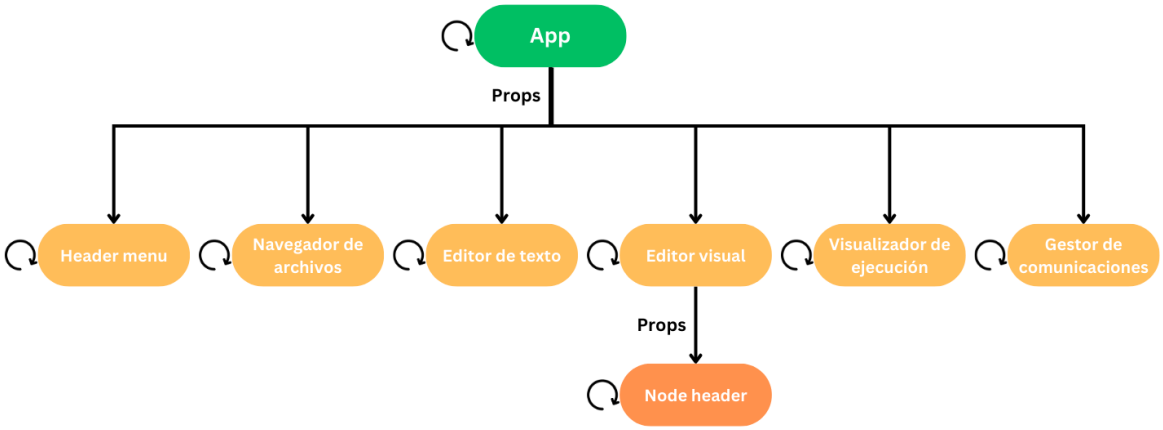


Figura 4.7: Jerarquía de componentes en BT Studio

Los componentes del frontend manejan la lógica de interacción, obteniendo la información requerida mediante *props* o realizando peticiones al backend. Cada componente se encarga de reunir de forma autónoma los datos necesarios para su operación y actualiza su estado interno acorde a estos, reflejando visualmente las modificaciones pertinentes. Tal información puede ser distribuida a lo largo de la jerarquía de componentes hacia sus descendientes. Por ejemplo, el componente encargado de la edición de archivos determinará qué archivos mostrar basándose en el proyecto actualmente seleccionado, información que recibe directamente desde el componente raíz de la aplicación.

Gestor de comunicaciones

Este componente se encarga de la comunicación entre el frontend y el RADI. Implementa distintas funciones para la comunicación mediante mensajes websocket con un formato determinado. El protocolo exacto para controlar la ejecución de la aplicación se detalla en la sección 4.4.

El gestor de comunicaciones es distinto al resto de componentes debido a que no contiene componentes de renderizado visual. Es usado como un *singleton*: se crea una única instancia de él nada más arrancar la interfaz. Esta instancia se pasa mediante *props* a los componentes que necesiten interaccionar con el backend robótico. De esta manera, se mantiene un estado único y compartido de comonicación.

Navegador de Proyectos

Este componente es esencial para la gestión de proyectos dentro de la aplicación, permitiendo la creación, visualización y navegación entre proyectos. Ofrece una interfaz clara para el seguimiento de los cambios y el estado actual del proyecto seleccionado. Asimismo, habilita el almacenamiento del estado del diagrama del árbol.

Es importante destacar que el guardado de las acciones asociadas a un determinado proyecto no se realiza en este componente, sino en el navegador de archivos. El navegador de archivos recibirá un *prop* con el nombre del proyecto actual (establecido por este componente) para poder realizar las operaciones de manera adecuada.

Figura 4.8: Apariencia del componente con cambios sin guardar en el proyecto

- **Endpoints utilizados**
  - `/tree_api/create_project`: para crear nuevos proyectos.
  - `/tree_api/save_project_graph`: guarda el estado actual del diagrama del árbol del proyecto.
  - `/tree_api/get_project_list`: obtiene la lista de todos los proyectos disponibles.
- **Props recibidos**
  - `setCurrentProjectName`: función para actualizar el nombre del proyecto actual.
  - `currentProjectName`: almacena el nombre del proyecto seleccionado.
  - `modelJson`: representación del estado del diagrama del árbol en formato JSON.
  - `projectChanges`: indica el estado de los cambios en el proyecto actual.
  - `setProjectChanges`: función para actualizar el estado de los cambios en el proyecto.

Navegador de Archivos

Este componente permite al usuario la creación, eliminación, visualización y selección de archivos dentro de un proyecto específico.

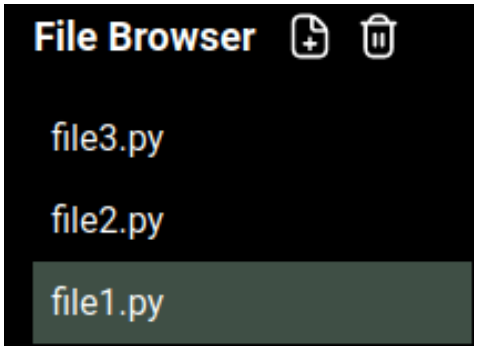


Figura 4.9: Apariencia del componente con tres archivos en el proyecto

- **Endpoints utilizados**
  - `/tree_api/get_file_list`: obtiene la lista de archivos del proyecto actual.
  - `/tree_api/create_file`: crea un nuevo archivo dentro del proyecto actual.
  - `/tree_api/delete_file`: elimina un archivo específico del proyecto actual.
- **Props recibidos**
  - `setCurrentFilename`: función para actualizar el nombre del archivo actualmente seleccionado. Esto permitirá al editor de código saber qué archivo debe mostrar al usuario.
  - `currentFilename`: almacena el nombre del archivo seleccionado.
  - `currentProjectname`: almacena el nombre del proyecto actual. Permite cargar la lista de archivos adecuada.
  - `setProjectChanges`: función para indicar si hay cambios en el proyecto.

Editor de texto

Permite editar el contenido del archivo seleccionado dentro de un proyecto, soportando la edición de texto con resaltado de sintaxis para Python. Integra funcionalidades como guardar cambios, ajustar el tamaño de fuente y visualizar indicadores de cambios no guardados.

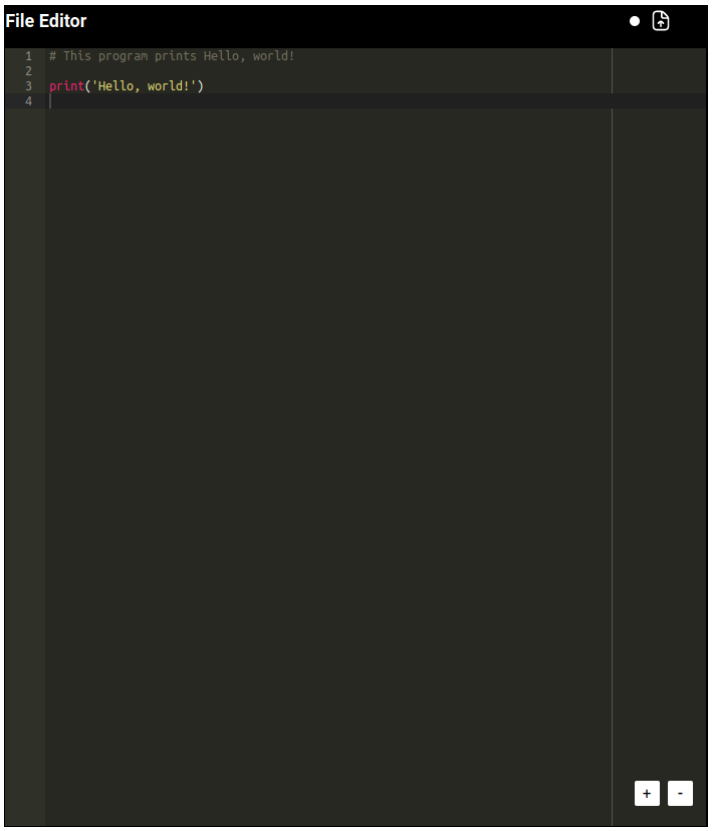


Figura 4.10: Interfaz del editor mostrando un archivo Python

- **Endpoints utilizados**
  - `/tree_api/get_file`: obtiene el contenido de un archivo específico basado en `currentProjectname` y `currentFilename`.
  - `/tree_api/save_file/`: guarda el contenido editado de un archivo específico en el servidor.
- **Props recibidos**
  - `currentFilename`: nombre del archivo actualmente seleccionado para edición.
  - `currentProjectname`: nombre del proyecto actual, usado para cargar y guardar archivos específicos.
  - `setProjectChanges`): función para indicar que ha habido cambios en el proyecto, para actualizar todas las interfaces que dependen de ello.

Editor visual de árboles

Es una paleta de trabajo donde se pueden añadir, arrastrar y borrar distintos bloques. Sus funcionalidades clave son:

- Creación dinámica de nodos y enlaces entre ellos. Se pueden añadir todos los nodos de control de flujo soportados y los nodos de ejecución definidos por las acciones. Soporta añadir de manera dinámica puertos a los distintos nodos y asignarles un valor. Como se estudió en la sección de BT.cpp, el valor de estos puertos puede ser una constante o una referencia a un valor de memoria compartida.
- Serialización del modelo del diagrama para su almacenamiento y posterior recuperación.
- Integración con *endpoints* de API para la recuperación y almacenamiento de diagramas.
- Personalización de nodos y puertos con modelos y fábricas específicas.
- Integración con *endpoints* de la API para la descarga y ejecución de aplicaciones.
- Comunicación con el backend robótico dockerizado a través del gestor de comunicaciones para el control de ejecución de las aplicaciones.

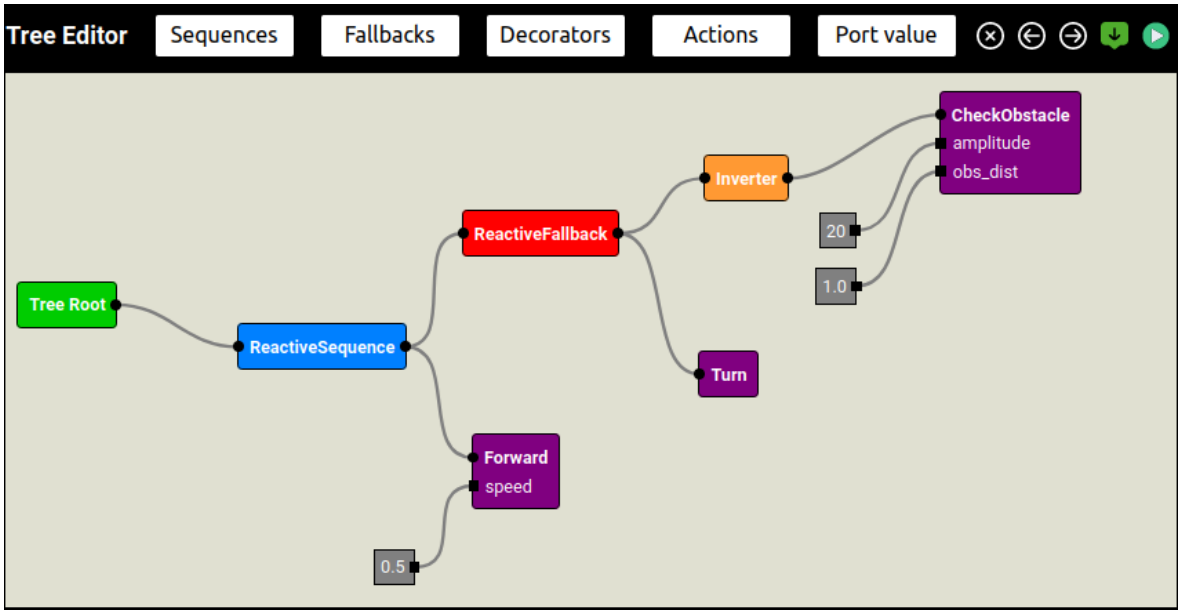


Figura 4.11: Editor visual de árboles con un árbol de ejemplo

- **Endpoints utilizados**

- `/tree_api/get_project_graph/`: obtiene el gráfico del proyecto actual basado en `currentProjectname`.
- `/tree_api/generate_app/`: genera y devuelve un archivo comprimido de la aplicación basada en el modelo del diagrama actual.
- `/get_dockerized_app/`: genera un archivo zip de la aplicación preparada para ser ejecutada en un entorno dockerizado basado en el modelo de diagrama actual.

- **Props recibidos**
  - `currentProjectname`: nombre del proyecto actual, usado para cargar y guardar el gráfico del proyecto.
  - `setModelJson`: función para actualizar el estado con el JSON del modelo del diagrama.
  - `setProjectChanges`: función para indicar que ha habido cambios en el proyecto. Esto cambiará indicadores en otros componentes.
  - `gazeboEnabled`: indica si la visualización del entorno dockerizado está activada. Esto es importante para activar el botón de ejecutar aplicaciones.
  - `manager`: instancia del gestor de comunicaciones entre el frontend y el backend robótico dockerizado.

Visualizador de ejecución

Permite visualizar el funcionamiento de la aplicación robótica. Es un cliente VNC extremadamente minimalista, que se conecta al servidor VNC proporcionado por el RADI cuando se establece la conexión.

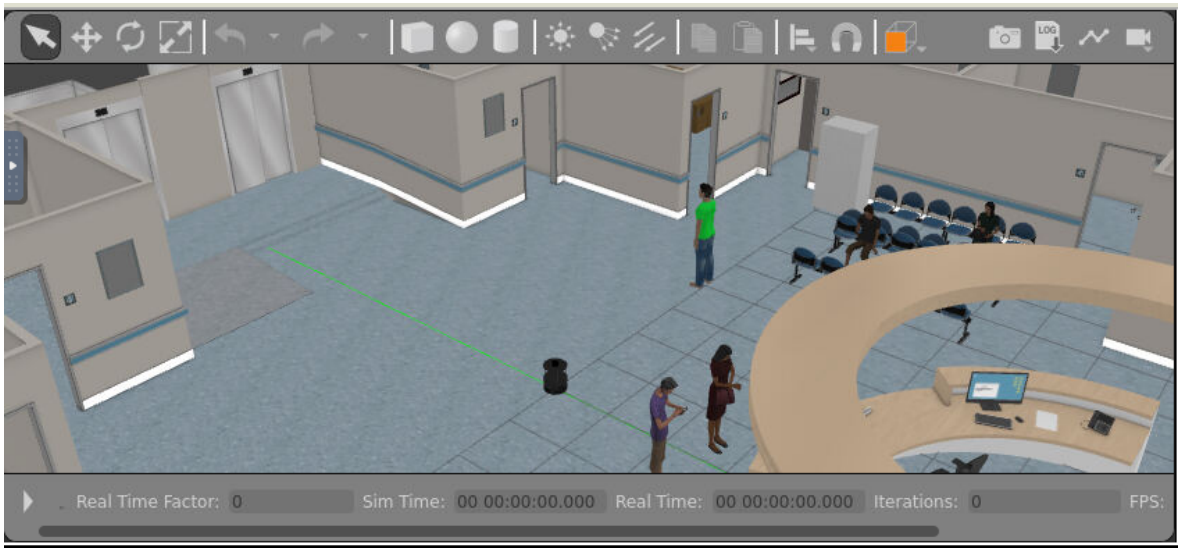


Figura 4.12: Visor de ejecución de aplicaciones robóticas

- **Endpoints utilizados:** ninguno
- **Props recibidos**
  - `gazeboEnabled`: indica si la comunicación con el entorno dockerizado se ha establecido correctamente y este ha lanzado el visor de Gazebo. Sólo cuando es *True* el visualizador intenta conectarse al servidor VNC. En caso contrario, muestra la pantalla de carga.

4.3. Traductor

El traductor es el componente de BT Studio encargado de transformar un listado de acciones programadas en Python y un diagrama de árbol de comportamiento codificado en JSON en una aplicación robótica auto contenida ejecutable en un entorno ROS 2.

Una aplicación ROS 2 autocontenida es aquella con todos los archivos y configuraciones necesarias para ser directamente instalada y ejecutada en un entorno ROS 2. Este entorno ROS 2 puede ser proporcionado por el usuario en su propia máquina o por un contenedor docker. Durante el desarrollo de este TFG se probaron ambas opciones.

Este componente puede actuar de manera totalmente independiente al IDE web y fue el primero en ser desarrollado, tal y como se indica en la planificación. El backend del IDE web proporciona acceso a la funcionalidad del mismo a través del endpoint `/tree_api/generate_app`.

El traductor está dividido en tres componentes diferenciados que se ejecutan de manera secuencial: el traductor JSON-XML, el generador de árboles XML autocontenidos y el generador de aplicaciones, como se puede observar en la figura 4.13. Este último componente parte de un paquete plantilla, que incluye configuraciones y código para la ejecución de los árboles de comportamiento.

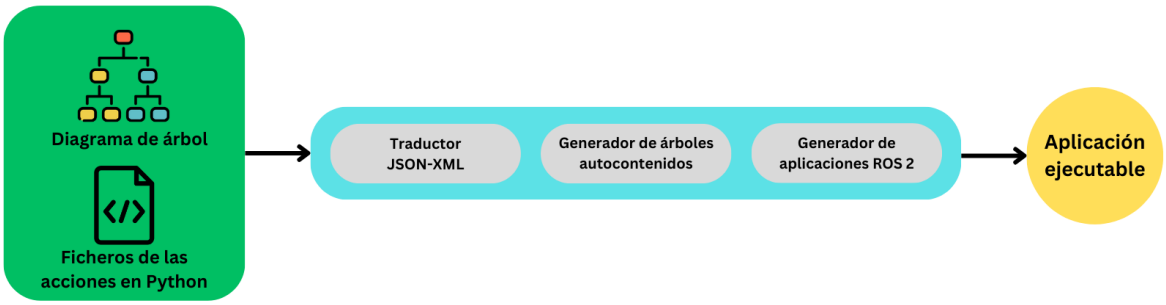


Figura 4.13: Secuencia de generación de aplicaciones mediante el traductor

```
# 1. Generate a basic xml tree from the JSON definition
if json_tree_path != "":
    json_translator.translate(json_tree_path, xml_tree_path)

# 2. Generate a self-contained xml tree
tree_generator.generate(xml_tree_path, action_path,
    self_contained_tree_path)

# 3. Using the self-contained tree, package the ROS 2 app
zip_file_path = app_generator.generate(self_contained_tree_path, app_name
    , template_path, action_path, tree_gardener_src)
```

Extracto de código 4.1: Pipeline de generación de aplicaciones

4.3.1. Traductor JSON-XML

Características

En BT Studio, el almacenamiento de los árboles se realiza en formato JSON para facilitar el proceso de visualización y edición con la librería ReactDiagrams. Sin embargo, este formato incluye muchos elementos propios de la visualización del gráfico que reducen la expresividad y portabilidad de los árboles.

El objetivo de este componente es transformar la representación JSON específica de ReactDiagrams a otra XML con el formato de la librería BT.cpp, mucho más compatible con aplicaciones robóticas. Este paso es opcional y sólo se ejecuta cuando el traductor se usa junto al IDE web. Un usuario offline puede proporcionar a la herramienta el árbol directamente en XML.

Funcionamiento

El funcionamiento de este componente se estructura en los siguientes pasos:

1. **Análisis del JSON:** el proceso se inicia con la lectura y análisis del contenido JSON, extrayendo las estructuras de datos que representan los nodos y las conexiones (links) entre estos. Cada nodo en el JSON tiene asociados diversos atributos, como tipo, puertos y, opcionalmente, datos adicionales que definen su comportamiento. Se ignoran todos los atributos gráficos.
2. **Construcción de la estructura del árbol:** a partir de la información obtenida, se construye internamente la estructura del árbol, en forma de grafo. Esto implica identificar las relaciones padre-hijo entre nodos, basándose en las conexiones definidas y rellenar un diccionario que las codifique. Los nodos de tipo *tag* se ignoran en esta fase, ya que su propósito es meramente descriptivo dentro de ReactDiagrams y no tienen equivalencia directa en el árbol de comportamiento de BT.cpp. Esto se debe a que en BT.cpp los puertos y valores se definen dentro del nodo, no como una conexión valor-puerto.
3. **Extracción de los puertos:** para cada nodo, se identifican y procesan los puertos de entrada y salida, que están vinculados a valores o nodos específicos. Esta información se utiliza para asignar propiedades y valores a los nodos en el documento XML final, reflejando así la configuración y los datos de los nodos.
4. **Generación del XML básico:** se crea el elemento raíz XML y, mediante un proceso recursivo, se van añadiendo los nodos del árbol como elementos XML, respetando la jerarquía y las relaciones padre-hijo identificadas previamente.
5. **Inclusión de atributos y metadatos:** además del nombre y tipo de cada nodo, se incluyen como atributos en los elementos XML los datos obtenidos de los puertos de entrada/salida. Esto implica que añadir el nombre y el valor de sus puertos a cada etiqueta XML.
6. **Formateo y salida:** el documento XML se formatea para mejorar la

legibilidad, convirtiendo la estructura de elementos XML en una cadena de texto bien estructurada y legible. Esta representación XML se guarda entonces en el archivo especificado por el usuario. El archivo generado siempre es una representación válida de un árbol en BT.cpp.

Este proceso permite a los usuarios de BT Studio alternar entre la edición visual y el uso de los árboles en entornos robóticos sin la necesidad de reestructurar o redefinir los árboles de comportamiento.

4.3.2. Generador de árboles autocontenidos

Características

Este componente del traductor está diseñado para aumentar la portabilidad y autonomía de los árboles de comportamiento generados con BT Studio. Al integrar todas las acciones y comportamientos definidos directamente dentro de un único archivo XML, facilita la distribución y ejecución de estos árboles en diferentes entornos sin necesidad de mantener múltiples archivos. El código presente dentro de la plantilla de ejecución de árboles puede preparar árboles ejecutables tan solo leyendo los contenidos de este archivo.

Funcionamiento

El proceso de generación de un árbol autocontenido se realiza en varios pasos fundamentales, asegurando que el archivo XML final pueda ser directamente utilizado:

1. **Lectura del XML de entrada:** el generador comienza leyendo el archivo XML que define la estructura del árbol de comportamiento. Este archivo contiene la definición de nodos y su organización, pero sin incluir el código de las acciones que estos nodos ejecutan.
2. **Análisis de la estructura del árbol:** la herramienta identifica todos los nodos de ejecución únicos presentes en el árbol. Estos nodos corresponden a los comportamientos escritos por el usuario.
3. **Recopilación del código de los nodos de ejecución:** para cada nodo de ejecución identificado, el generador busca el código correspondiente en el directorio especificado. Este código define la lógica de cada nodo.

4. **Incorporación del código de las acciones al árbol:** el código de cada acción se incrusta directamente dentro del archivo XML como parte de una nueva sección dedicada. Esto se realiza creando un elemento Code con el nombre de cada acción, dentro del cual se inserta el código de la acción como texto.
5. **Formateo final:** el generador aplica un formateo específico al archivo XML para asegurar que la estructura resultante sea legible y esté correctamente indentada. Este paso es crucial para mantener la claridad del archivo, lo cual es especialmente importante con la adición del código de las acciones.

4.3.3. Generador de aplicaciones ROS 2

Características

El generador de aplicaciones se encarga de empaquetar árboles de comportamiento y sus dependencias en un paquete ROS 2 a partir de una plantilla base. Esta plantilla sirve como esqueleto para el paquete final y contiene archivos de configuración, scripts de ejecución y placeholders. Además, proporciona un *entrypoint* para la ejecución de la aplicación.

El proceso de generación del paquete implica la adaptación de la plantilla configurando archivos específicos, incluyendo árboles de comportamiento personalizados, y actualizando las dependencias para asegurar la compatibilidad y funcionalidad dentro del ecosistema ROS 2.

Funcionamiento

El funcionamiento del generador se puede desglosar en los siguientes pasos:

1. **Preparación de la plantilla:** el proceso comienza con la copia de una plantilla predefinida de ROS 2 a un directorio temporal.
2. **Personalización del paquete:** se renombran archivos y directorios dentro de la plantilla, reemplazando el nombre genérico de la plantilla por el nombre específico de la aplicación proporcionado por el usuario. Además, se modifican los contenidos de archivos clave (como `package.xml`, `setup.py`, y otros) para reflejar el nombre de la aplicación y otros metadatos importantes.
3. **Recopilación de dependencias únicas:** se analiza el código de las acciones presente en el XML autocontenido para identificar importaciones únicas de

Python y actualizar el archivo `package.xml` con estas dependencias. Esto garantiza que el todas las dependencias necesarias para las acciones están correctamente declaradas y pueden ser resueltas por rosdep.

4. **Empaquetado y compresión**: finalmente, todos los archivos y directorios modificados se empaquetan en un archivo ZIP. Este archivo comprimido contiene el paquete ROS 2 completo y personalizado, listo para ser instalado en un entorno de desarrollo de ROS 2.

TreeGardener: generación y ejecución de árboles

Dentro de la plantilla, se incluye un paquete ROS 2 encargado de proporcionar la funcionalidad de creación y ejecución de los árboles de comportamiento. En el *entrypoint* del paquete de aplicación generado, se utiliza *TreeGardener* como dependencia para generar un nodo de ROS 2 ejecutable a partir del XML autocontenido.

TreeGardener proporciona dos funcionalidades diferenciadas:

- **TreeFactory**: es el núcleo de generación del árbol de comportamiento.
    - **Extensión de funcionalidad:** implementa un conjunto de clases que extienden o modifican el comportamiento de los nodos de control de flujo proporcionados por *py-trees* para que funcionen de manera idéntica a BT.cpp.
    - **Diccionario de nodos personalizado:** incluye un diccionario de nodos personalizados, que asociada a cada nodo presente en BT.cpp una clase con el mismo comportamiento implementada con *py-trees*. Las acciones definidas por el usuario en el archivo XML autocontenido también se incluyen en el diccionario.
    - **Creación de árboles ejecutables:** utiliza el diccionario de nodos para construir dinámicamente un árbol de comportamiento ejecutable como nodo de ROS 2. Es decir, capaz de acceder a las herramientas de ROS 2 como callbacks. Este proceso se realiza recorriendo la estructura el árbol de manera recursiva e instanciando las clases que definen los bloques correspondientes. Este es el paso donde las acciones se convierten en código ejecutable, por lo que todas las dependencias deben ser accesibles.

- **TreeTools**: proporciona utilidades y herramientas auxiliares para la interacción de los nodos.
    - **Implementación de la Blackboard**: proporciona una implementación muy minimalista y sencilla de la memoria compartida como una clase singleton.
    - **Acceso a la Blackboard:** contiene métodos para la manipulación de la blackboard global y la obtención de contenidos de puertos especificados en las acciones. Permite a las acciones del árbol leer y escribir información relevante durante la ejecución.
    - **Dependencia de las acciones:** las acciones deben importar esta librería en su definición. Cuando *TreeFactory* instancia las clases obtenidas del XML autocontenido, *TreeTools* se encuentra disponible en el path.

```
# Init tree
root = construct_behaviour_tree_from_xml(xml_doc)
tree = py_trees_ros.trees.BehaviourTree(root=root unicode_tree_debug=
    False)

# Setup tree
try:
    tree.setup(timeout=timeout)
except py_trees_ros.exceptions.TimedOutError as e:
    tree.shutdown()
    return None

return tree
```

Extracto de código 4.2: Generación de un árbol ejecutable con TreeFactory

4.4. Ejecutor dockerizado

El último componente de BT Studio es el ejecutor dockerizado, encargado de proporcionar un entorno con todas las herramientas necesarias para la visualización de la ejecución de las aplicaciones. Para el desarrollo BT Studio, se realizaron modificaciones sobre el RADI (*Robotics Application Docker Image*) 4.4.32

de RoboticsAcademy para permitir el lanzamiento de aplicaciones robótica genéricas.

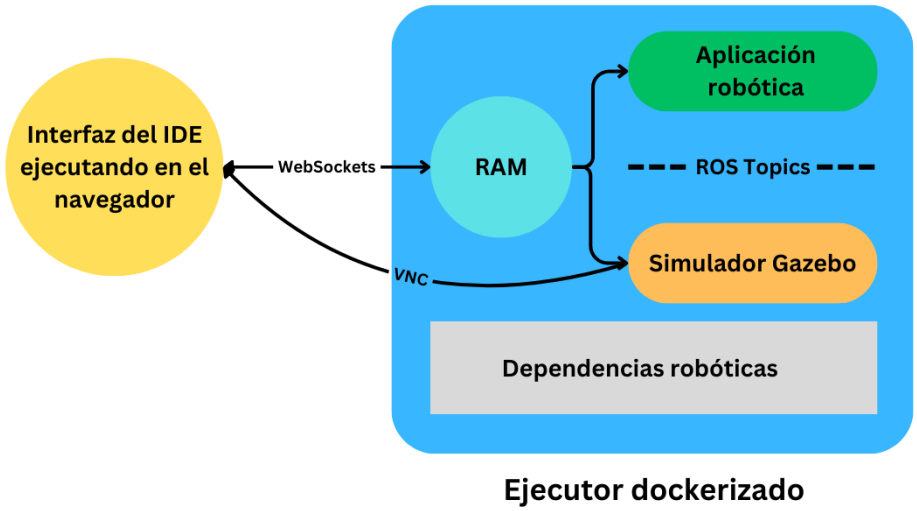


Figura 4.14: Ejecución de una aplicación robótica con el RADI de RoboticsAcademy

4.4.1. Arquitectura

El RADI es una imagen basada en Ubuntu 20.04 que permite lanzar un contenedor Docker con un entorno de desarrollo ROS 2 completo junto con diversas herramientas adicionales. Dentro de este contenedor existe una extensa colección mundos para el simulador Gazebo con sus correspondientes *launchers*, denominados universos.

Para el control de las distintas herramientas, dentro del contenedor se ejecuta un programa gestor, conocido como *RoboticsApplicationManager* (RAM). Este programa es capaz de lanzar universos, preparar distintas formas de visualización (generalmente visores VNC) y controlar la ejecución de aplicaciones robóticas. La interfaz del IDE desde el navegador web se comunica mediante websockets con el *manager* para ejecutar las acciones requeridas por el usuario en cada momento. El manager es capaz de lanzar acciones como subprocesos del sistema, lo que permite un mejor aprovechamiento de los recursos del sistema.

Herramientas incluidas en el RADI

Las herramientas incluidas dentro del entorno de desarrollo son las siguientes

1. **ROS2 Humble**

- Simulador Gazebo
- RViz2

2. **Python 3.10**
    - websocket_server
    - Django
    - websockets
    - asyncio
3. **Xvfb:** xserver virtual.
4. **Aceleración GPU:** VirtualGL
5. **Servidores VNC**
    - TurboVNC
    - noVNC
6. **Dependencias habituales de aplicaciones robóticas**
    - OpenCV
    - OMPL
    - PyTorch
    - TensorFlow
    - MoveIt
    - PX4
    - AeroStack2
7. **RoboticsAcademy**
8. **RoboticsInfrastructure**
9. **RoboticsApplicationManager**

4.4.2. Ejecución de aplicaciones

Cuando se lanza el RADI, el *manager* arranca un servidor websocket en el puerto 7163, que escucha constantemente esperando conexiones entrantes por parte del

cliente. Una vez establecida la conexión, la interfaz del IDE puede enviar distintos comandos a través del componente de comunicaciones anteriormente explicado.

El lanzamiento de aplicaciones dentro del RADI está dividido en cuatro pasos y se realiza de manera secuencial y monitorizada, de manera análoga a los peldaños de una escalera (figura 4.15). El *manager* utiliza una máquina de estado para mantener una representación de su estado actual y las posibles transiciones.

Esto implica que el cliente debe enviar los comandos en el orden adecuado, esperando a la confirmación por parte del *manager* antes de continuar. Esto permite controlar el ciclo de vida de la ejecución de las aplicaciones de manera muy precisa, con un comportamiento completamente determinista por parte del *manager.*

De manera secuencial, los cuatro mensajes que manda la interfaz del IDE son para iniciar la ejecución de una aplicación son:

1. **Connect**: abre la conexión con el servidor websocket. El estado del *manager* pasa a ser *connected*.

2. **LaunchUniverse**: envía un mensaje que contiene el universo que se desea lanzar, indicando el *launcher* correspondiente. Dado que el contenedor no posee interfaz gráfica, en este paso sólo se lanza el servidor de Gazebo (gzserver). El estado del *manager* pasa a ser*universe_ready*.

3. **PrepareVisualization**: una vez el mundo se está ejecutando, se comunica el tipo de visualización que requiere la aplicación. BT Studio sólo requiere un visor VNC conectado al cliente de Gazebo. Para simular la interfaz gráfica, el *manager* crea un escritorio virtual utilizando Xvfb y ejecuta allí el cliente de Gazebo (gzclient). Una vez se ha completado este paso, el cliente VNC del frontend puede conectarse mediante la dirección proporcionada. El estado del *manager* pasa a ser *visualization_ready*.

4. **RunApplication:** una vez el usuario ha generado una aplicación robótica con BT Studio y decide ejecutarla, se envía este mensaje incluyendo el zip de la aplicación codificado en el formato base64. El *manager* lo descomprime y prepara de la manera adecuada con todas sus dependencias y ejecuta el *entrypoint* de la aplicación. El estado del *manager* pasa a ser *application_ready*. Tanto el simulador, las herramientas de visualización como la aplicación robótica con sus dependencias se ejecutan dentro del contenedor. Una de las ventajas de esto es que BT Studio es multiplataforma.

Durante toda esta secuencia, el *manager* ejecuta los distintos componentes como subprocesos, utilizando la librería *subprocess* de Python. Al hacerlo, almacena

el PID de cada subproceso, lo que permite ejecutar acciones sobre ellos mediante llamadas estándar del sistema Linux. Las acciones soportadas por el *manager* sobre los componentes lanzados son:

- **Sobre la aplicación**
  - **Pause:** suspende la ejecución del proceso. Equivalente a Crtl+Z desde una terminal o a enviar la señal STOP con el comando kill.
  - **Resume:** continúa la ejecución del proceso. Equivalente a enviar la señal CONT con el comando kill.
  - **TerminateApplication:** cierra de manera recursiva todos los procesos asociados a la ejecucón de la aplicación. Además, reinicia y pausa la simulación en Gazebo. El estado del *manager* pasa de *application_ready* a *visualzation_ready*.
- **Sobre la visualización**
  - **TerminateVisualization:** cierra de manera recursiva todos los procesos asociados a la visualización, como el servidor VNC o gzclient. El estado del *manager* pasa de *visualzation_ready* a *universe_ready*.
- **Sobre el entorno de simulación**
  - **TerminateUniverse:** cierra de manera recursiva todos los procesos asociados a la ejecución del universo, especialmente gzserver. El estado del *manager* pasa de *universe_ready* a *connected*.

Es importante destacar que la secuencia de inicio de una aplicación puede ser revertida hasta el estado deseado ejecutando los comandos en orden inverso. Esto permite por ejemplo lanzar distintas aplicaciones sin necesidad de reiniciar la visualización o el entorno de simulación.

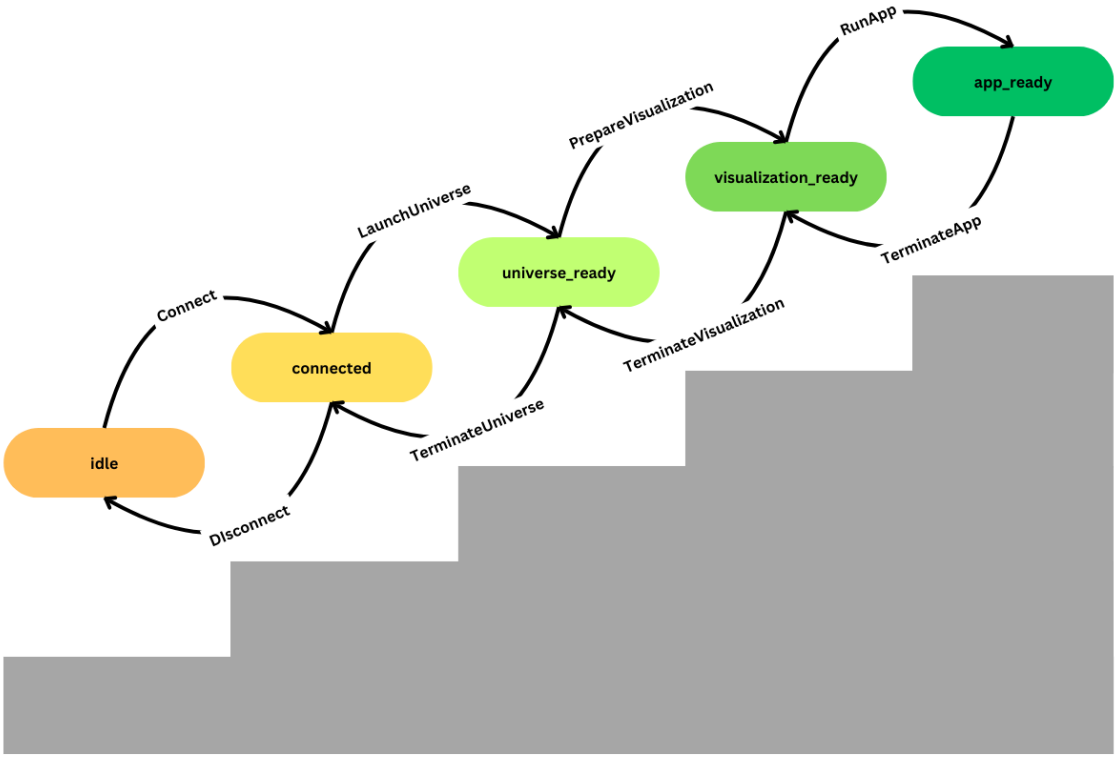


Figura 4.15: Escalera de transiciones en el RADI durante la ejecución dockerizada de una aplicación robótica

5. Validación experimental

En este capítulo se presenta la validación experimental de los objetivos propuestos en el capítulo 2. Concretamente, se proporcionará una descripción detallada de las pruebas experimentales realizadas y se valorarán sus resultados.

5.1. Procedimiento experimental

El procedimiento de validación experimental consiste en el desarrollo de dos aplicaciones robóticas de ejemplo utilizando exclusivamente BT Studio. El proceso de creación de las mismas y el resultado final obtenido será documentado a través de vídeos. Estos muestran que BT Studio cuenta con las siguientes funcionalidades:

- Acceso desde cualquier navegador tras un proceso de instalación debidamente documentado.
- Mecanismo para gestionar y navegar entre distintos proyectos. Cada proyecto debe almacenar un conjunto de acciones y un árbol de comportamiento.
- Edición de ficheros Python para programar acciones de un árbol de comportamiento. Para proporcionar esta funcionalidad, es necesario un navegador de archivos y un editor de texto. Idealmente, proporciona resaltado de sintaxis Python, facilitando al usuario la programación.
- Acceso en las acciones a diversas librerías mediante los mecanismos estándar de *binding* de Python. Por ejemplo, importando *rclpy*, se podría acceder a toda la funcionalidad de ROS 2.
- Creación de árboles de comportamiento mediante un editor visual basado en bloques. Mediante este editor, es posible la creación de árboles de comportamiento que contengan todos los tipos de bloques proporcionados por la librería BT.cpp. También es posible utilizar las acciones programadas por el usuario como bloques dentro de este editor.
- Integración con el RADI de RoboticsAcademy para la ejecución de las aplicaciones programadas por los usuarios.
- Mecanismos para el control y visualización de la ejecución de las aplicaciones desde el navegador.

Se desarrollaron dos aplicaciones ilustrativas: Laser Bump and Go y Visual Follow Person. Fueron implementadas en primer lugar utilizando la versión offline de BT Studio y posteriormente el procedimiento fue replicado con BT Studio integrado con Unibotics D1. En los vídeos se muestra una ejecución típica de ambas aplicaciones, tanto offline como en Unibotics D1.

5.2. Aplicación Laser Bump and Go

Esta aplicación permite a un robot ejecutar un comportamiento clásico de chocar y girar. Este es un comportamiento reactivo simple: el robot avanza hacia delante mientras no detecta un obstáculo, en caso contrario, gira. En este caso, la detección de los obstáculos se realiza mediante un sensor laser a bordo del robot.

5.2.1. Resumen

- **Código para consulta:** https://github.com/JdeRobot/bt-studio/tree/main/backend/filesystem/laser_bump_and_go
- **Plataforma robótica utilizada:** TurtleBot2 simulado mediante Gazebo.
- **Sensores utilizados:** LIDAR 2D.

5.2.2. Descripción

Listado de acciones

- **Forward:** obtiene la velocidad lineal deseada a través del puerto de entrada *lin_speed* y la publica a través del topic */cmd_vel*. Cuando le llega un tick, siempre devuelve *Running*.
- **Turn:** obtiene la velocidad angular deseada a través del puerto de entrada *ang_speed* y la publica a través del topic */cmd_vel*. Cuando le llega un tick, siempre devuelve *Running*.
- **CheckObstacle:** se suscribe al topic */scan* donde se publican las medidas del laser. Cuando le llega un tick, comprueba las medidas del láser dentro de la amplitud deseada que obtiene a través del puerto de entrada *amplitude*. Posteriormente comprueba si alguna de estas medidas es menor a la distancia

mínima deseada que obtiene a través del puerto de entrada *obs_dist*. Si la distancia es menor, devuelve *Success* y en caso contrario, *Failure*.

Árbol de comportamiento

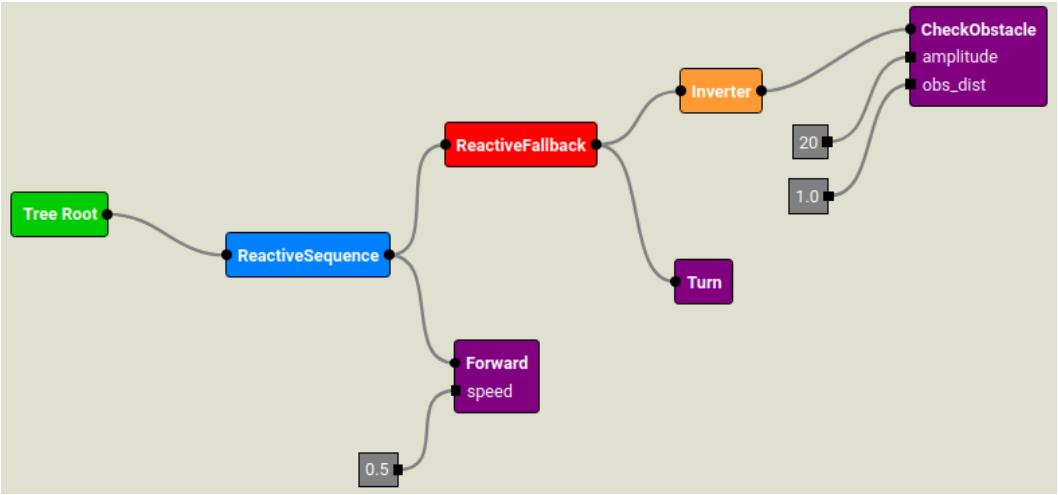


Figura 5.1: Árbol de comportamiento de Laser Bump and Go

En primer lugar, se ejecuta el *ReactiveFallback*, que hace *tick* a *CheckObstacle*. Si este detecta un obstáculo devolverá *Success*, pero al pasar por el *Inverter* se convierte en *Failure*, lo que produce que se haga *tick* a *Turn*, que reiniciaría el *callback* debido a que siempre devuelve *Running*.

En caso de que *CheckObstacle* no detecte un obstáculo devolverá *Failure*, que al pasar por el *Inverter* se convierte en *Success*. Esto producirá que se termine la ejecución del *callback* y se ejecute *Forward*. Como este siempre devuelve *Running*, se reiniciará la ejecución de la *ReactiveSequence*.

5.2.3. Ejecución típica

El vídeo demostrativo de esta aplicación se puede encontrar en el siguiente enlace: `https://youtu.be/agVe5SAAA6s`.

Los momentos más destacados de la ejecución de la aplicación son:

- `0:45`: comienza la ejecución. Como *CheckObstacle* no detecta un obstáculo, se sale del callback y se ejecuta *Forward*. Tras ello se reiniciará la secuencia. Mientras esto ocurra el robot avanza hacia delante de manera constante.

- `1:00`: *CheckObstacle* detecta un obstáculo, por lo que se ejecutará el siguiente componente del *callback*, *Turn*. Esto produce que mientras se se detecte un obstáculo, el robot gire de manera constante.
- `1:06`: *CheckObstacle* ya no detecta un obstáculo, por lo que se deja de ejecutar *Turn* y se vuelve a ejecutar *Forward*.
- `1:10`: se vuelve a detectar un obstáculo. Se ejecuta *Turn* de nuevo

Este comportamiento se repite en el resto del vídeo.

5.3. Aplicación Visual Follow Person

Esta aplicación permite a el robot seguir a una persona de manera reactiva. Utilizando un filtro de color sobre la imagen de la cámara, detecta a una persona determinada en el escenario y comanda las velocidades lineares y angulares adecuadas para su correcto seguimiento, manteniendo a la persona en el centro de la imagen. En caso de perder a la persona, el robot gira para intentar recuperarla.

5.3.1. Resumen

- **Código para consulta:** https://github.com/JdeRobot/bt-studio/tree/main/backend/filesystem/visual_follow_person
- **Plataforma robótica utilizada:** TurtleBot2 simulado mediante Gazebo.
- **Sensores utilizados:** Cámara RGB

5.3.2. Descripción

Listado de acciones

- **DetectPerson:** se suscribe al topic */depth_camera/image_raw* donde se publican las imágenes de la cámara. Cuando llega un tick, aplica un filtro de color verde e intenta calcular el centroide de la imagen filtrada. Si este centroide existe, se está detectando a la persona y se publica el valor de su componente X en el puerto de salida *person_pos*. Si se ha detectado a la persona se devuelve *Success*, en caso contrario *Failure*.

- **Turn:** gira a una velocidad angular indicada en el puerto de entrada *ang_speed*. Cuando le llega un tick, siempre devuelve *Running*.
- **Move:** recibe por el puerto de entrada *person_pos* la posición de la persona en el eje X de la imagen y por el puerto de entrada *image_x_center* el centro en X de la imagen. Con esta información calcula el error de seguimiento como la diferencia entre el centro de la imagen y la posición de la persona. Aplicando un controlador proporcional, calcula la velocidad lineal y angular adecuada. Una vez calculada, la publica a través del topic */cmd_vel*. Cuando le llega un tick, siempre devuelve *Running*.

Árbol de comportamiento

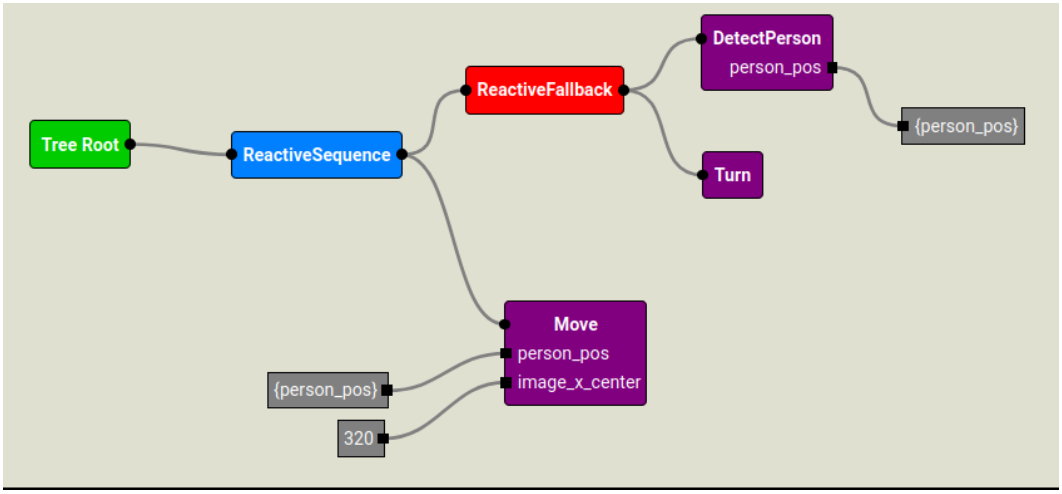


Figura 5.2: Árbol de comportamiento de Visual Follow Person

En primer lugar, se ejecuta el *ReactiveFallback*, que hace *tick* a *DetectPerson*. Si este detecta a una persona devolverá *Success* y terminará la ejecución del *callback*. En caso contrario se hará *tick* a *Turn*, que al devolver siempre *Running*, reiniciará el *callback*. Cuando termina el *ReactiveFallback* porque se ha detectado una persona, se hace *tick* a Move, que siempre devuelve *Running*. Esto provocará el reinicio de la ejecución de la *ReactiveSequence*.

5.3.3. Ejecución típica

El vídeo demostrativo de esta aplicación se puede encontrar en el siguiente enlace: https://youtu.be/a4c-nRevF_c.

Los momentos más destacados de la ejecución de la aplicación son:

- `0:45`: comienza la ejecución. Durante toda la ejecución *DetectPerson* consigue identificar una persona en la imagen. Esto produce que se salga del *callback* y se ejecute *Move*. Este calcula las velocidades necesarias para mantener a la persona en el centro de la imagen. *DetectPerson* mantiene actualizada en cada iteración la posición de la persona.

5.4. Integración en Unibotics

En Unibotics, existen tres fases de desarrollo:

- **D1:** despliegue en la máquina local del desarrollador. Permite implementar y probar cambios de manera ágil, sin consecuencia de ningún tipo en caso de romper algún componente.
- **D2:** despliegue en un servidor de pruebas, conectado a una granja reducida, donde se ejecutan las aplicaciones robóticas. Sirve para comprobar que los cambios funcionan en un entorno muy cercano al de producción.
- **D3:** despliegue en el entorno de producción. Es al que pueden acceder los usuarios a través de la url del proyecto.

Durante el desarrollo de este trabajo, se ha ha conseguido integrar BT Studio en el despliegue D1. Esto supone un paso fundamental de la integración con Unibotics, que deberá ser validado en los despliegues posteriores.

Para demostrar la integración de BT Studio con Unibotics D1 se ha grabado un vídeo ejecutando la aplicación *Laser Bump and Go*. Este vídeo se puede encontrar en el siguiente enlace: https://youtu.be/q-Ui2epQEgE

6. Conclusiones

En este capítulo se valorarán los resultados producidos durante este TFG y se propondrán futuras líneas de desarrollo relacionadas.

6.1. Cumplimiento de objetivos

En esta sección, se revisa el listado de objetivos propuestos en el capítulo 2 y se evalua de manera desglosada si han sido satisfechos exitosamente. Se detallará la forma en la que cada uno de los objetivos ha sido desarrollado y los criterios disponibles para su correcta validación.

6.1.1. Objetivo principal

El objetivo principal de este TFG era la creación de un IDE web *open source* basado en árboles de comportamiento completamente funcional que los desarrolladores de aplicaciones robóticas puedan usar desde sus equipos, es decir, de manera offline tras un proceso de instalación. Se pretendía que tras la finalización de este trabajo, los potenciales usuarios dispusieran de una herramienta completa ejecutable desde un navegador web para el diseño y ejecución de aplicaciones robóticas utilizando árboles de comportamiento.

En el capítulo 3, se detallaron todas las tecnologías que se han utilizado para el desarrollo de dicha herramienta. Estas tecnologías se dividen en aquellas necesarias para la implementación del IDE web, como React o Django; y aquellas utilizadas para la ejecución y visualización de aplicaciones robóticas, como ROS 2, Gazebo o Docker.

En el capítulo 4, en primer lugar se establecieron los criterios de diseño de la plataforma y los distintos componentes necesarios para alcanzar la funcionalidad deseada. En segundo lugar, se analizó de manera exhaustiva la implementación de cada uno de los componentes que conforman la herramienta: la interfaz del IDE (con su frontend y su backend), el traductor y el ejecutor dockerizado.

Los componentes explicados en ese capítulo fueron implementados de manera exitosa, dando forma a BT Studio, disponible bajo la licencia GPLv3 en un

repositorio de Github[1].

En el capítulo 5, se validó experimental que la solución cuenta con toda la funcionalidad propuesta en el objetivo principal desarrollando con ella dos aplicaciones robóticas completas de ejemplo y sus correspondientes vídeos demostrativos. Tras seguir las instrucciones de instalación de BT Studio, los usuarios tienen la capacidad de crear y ejecutar desde el navegador aplicaciones robóticas complejas basadas en árboles de comportamiento.

En vista de todas las cuestiones anteriores, el objetivo principal de este TFG se considera satisfecho en su totalidad.

6.1.2. Objetivos secundarios

A continuación se detallan los objetivos secundarios propuestos y los criterios proporcionados para evaluar su cumplimiento.

1. **Programación de un editor visual web:** basado en *React* y *ReactDiagrams*. Concretamente, para el desarrollo del primer componente funcional de BT Studio, la interfaz gráfica, cuya implementación de detalla en la sección 4.2. Debido a esto, este objetivo se da por satisfecho.

2. **Desarrollo de un traductor capaz de generar código Python a partir de la codificación de árboles de comportamiento como diagramas web**: con su conversor de *json* a *xml*, su generador de árboles en *Python* y su empaquetador de aplicaciones ROS 2. Constituye el segundo componente funcional de BT Studio, cuya implementación de detalla en la sección 4.3. Es por ello que este objetivo se considera adecuadamente satisfecho.

3. **Integración con el RADI de RoboticsAcademy**: fue necesario cumplir este objetivo para proporcionar la capacidad de ejecución dockerizada de aplicaciones robóticas a BT Studio. Su implementación puede ser consultada en la sección 4.4. En consecuencia, este objetivo considera cumplido satisfactoriamente.

4. **Generación de aplicaciones de ejemplo para demostrar las capacidades de la solución**: tras el desarrollo de BT Studio, ambas aplicaciones propuestas (Laser Bump and Go y Follow Person) fueron satisfactoriamente implementadas, como se demuestra en el capítulo 5.

[1] https://github.com/JdeRobot/bt-studio

5. **Integración en Unibotics:** como se explica y demuestra en capítulo 5, hasta la fecha BT Studio sólo ha podido ser integrado en el despliegue D1 por motivos técnicos ajenos al desarrollo de este trabajo. Es por esto que este objetivo se considera parcialmente satisfecho.

6.2. Futuras líneas de desarrollo

A la vista del grado de cumplimiento de los objetivos propuestos, se considera que BT Studio es una herramienta completamente funcional y potencialmente útil para un gran número de usuarios. Es por ello que las futuras líneas de desarrollo deben ir enfocadas en aumentar las capacidades del sistema y facilitar su uso a un número mayor de usuarios.

Las líneas propuestas son:

- **Integración en Unibotics D3:** con el objetivo que BT Studio pueda ser utilizado sin la necesidad de instalación.
- **Difusión internacional:** con el objetivo de generar una comunidad de usuarios y aportar valor a la comunidad *open soruce*.
- **Soporte extendido de universos en el RADI:** actualmente, BT Studio no soporta todos los universos proporcionados por el RADI de RoboticsAcademy. Se podría añadir una base de datos de universos y elementos a la interfaz web para permitir una correcta selección.
- **Biblioteca de universos y robots propias de los usuarios:** esto permitiría a los usuarios crear, editar y guardar universos y robots desde el navegador. Además, se podría añadir la capacidad de importar entornos de simulación en formato XML, el estándar en ROS 2 tanto para la definición de mundos de Gazebo como de robots. Estos entornos se guardarían asociados a los proyectos de los usuarios y podrían ser lanzados bajo demanda.
- **Aplicaciones multinodo**: actualmente, las aplicaciones se ejecutan en un único nodo de ROS 2 y sólo pueden controlar de manera efectiva la ejecución de un comportamiento. Con cambios en el lanzamiento de las aplicaciones y la introducción de *launchers* personalizados por los usuarios, se posibilitaría la ejecución en paralelo de distintos servicios de ROS 2, como el *stack* de navegación.
- **Biblioteca de árboles de comportamiento:** el objetivo de esta funcionalidad

sería proporcionar junto a BT Studio una colección de comportamientos robóticos frecuentes reutilizables, como la manipulación de objetos o la detección de personas. Además, los usuarios podrían generar y contribuir con nuevos comportamientos desarrollados en sus proyectos. Esta biblioteca deberá ir acompañada de herramientas más potentes para la composición y la reutilización de árboles, como la capacidad de expandir o colapsar subárboles.

Bibliografía

[1] John D Lees-Miller. Overleaf: Scientific writing and publishing in the age of the cloud. In *PKP Scholarly Publishing Conference 2015*, 2015.

[2] Alex Banks and Eve Porcello. *Learning react: Modern patterns for developing react apps*. O'Reilly Media, 2020.

[3] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), May 2022. ISSN 2470-9476. doi: 10.1126/scirobotics.abm6074. URL http://dx.doi.org/10.1126/scirobotics.abm6074.

[4] Denis Chikurtev. Mobile robot simulation and navigation in ros and gazebo. In *2020 International Conference Automatics and Informatics (ICAI)*, pages 1–6, 2020. doi: 10.1109/ICAI50593.2020.9311330.

[5] Kenta Takaya, Toshinori Asai, Valeri Kroumov, and Florentin Smarandache. Simulation environment for mobile robots testing using ros and gazebo. In *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 96–101, 2016. doi: 10.1109/ICSTCC.2016.7790647.

[6] Michele Colledanchise and Petter Ögren. Behavior trees in robotics and ai, July 2018. URL http://dx.doi.org/10.1201/9780429489105.

[7] Petter Ögren and Christopher I. Sprague. Behavior trees in robot control systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 5(1):81–107, 2022. doi: 10.1146/annurev-control-042920-095314. URL https://doi.org/10.1146/annurev-control-042920-095314.

[8] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154:104096, 2022. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2022.104096. URL https://www.sciencedirect.com/science/article/pii/S0921889022000513.

[9] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. Behavior trees and state machines in robotics applications. *IEEE Transactions on Software Engineering*, 49(9):4243–4267, 2023. doi: 10.1109/TSE.2023.3269081.

[10] Ruffin White and Henrik Christensen. *ROS and Docker*, pages 285–307. Springer International Publishing, Cham, 2017. ISBN 978-3-319-54927-9. doi: 10.1007/978-3-319-54927-9_9. URL https://doi.org/10.1007/978-3-319-54927-9_9.

[11] David Roldán-Álvarez, José M. Cañas, David Valladares, Pedro Arias-Perez, and Sakshay Mahna. Unibotics: open ros-based online framework for practical learning of robotics in higher education. *Multimedia Tools and Applications*, 2023. ISSN 1573-7721. doi: 10.1007/s11042-023-17514-z. URL https://doi.org/10.1007/s11042-023-17514-z.

[12] José M. Cañas, Eduardo Perdices, Lía García-Pérez, and Jesús Fernández-Conde. A ros-based open tool for intelligent robotics education. *Applied Sciences*, 10(21), 2020. ISSN 2076-3417. doi: 10.3390/app10217419. URL https://www.mdpi.com/2076-3417/10/21/7419.