UNIVERSIDAD POLITÉCNICA SALESIANA

SEDE CUENCA

CARRERA DE ELECTRÓNICA Y AUTOMATIZACIÓN

MIGRACIÓN DEL SOFTWARE ROS1 A ROS2 EN EL SISTEMA OPERATIVO DEL ROBOT ROVER PRAX-IA DEL GRUPO DE INVESTIGACIÓN GIIRA

Trabajo de titulación previo a la obtención del título de Ingeniero en Electrónica

AUTOR: JUAN DANIEL GUAMÁN PINTADO

TUTOR: ING. CHRISTIAN RAÚL SALAMEA PALACIOS, PhD.

Cuenca – Ecuador

2025

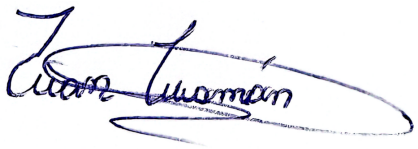
CERTIFICADO DE RESPONSABILIDAD Y AUTORÍA DEL TRABAJO DE TITULACIÓN

Yo, Juan Daniel Guamán Pintado con documento de identificación N° 0105206585; manifiesto que:

Soy el autor y responsable del presente trabajo; y, autorizo a que sin fines de lucro la Universidad Politécnica Salesiana pueda usar, difundir, reproducir o publicar de manera total o parcial el presente trabajo de titulación.

Cuenca, 23 de julio de 2025

Atentamente,



Juan Daniel Guamán Pintado
0105206585

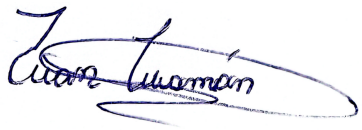
CERTIFICADO DE CESIÓN DE DERECHOS DE AUTOR DEL TRABAJO DE TITULACIÓN A LA UNIVERSIDAD POLITÉCNICA SALESIANA

Yo, Juan Daniel Guamán Pintado con documento de identificación N° 0105206585, expreso mi voluntad y por medio del presente documento cedo a la Universidad Politécnica Salesiana la titularidad sobre los derechos patrimoniales en virtud de que soy autor del Proyecto técnico: "Migración del software ROS1 a ROS2 en el sistema operativo del robot ROVER PRAX-IA del grupo de investigación GIIRA", el cual ha sido desarrollado para optar por el título de: Ingeniero en Electrónica, en la Universidad Politécnica Salesiana, quedando la Universidad facultada para ejercer plenamente los derechos cedidos anteriormente.

En concordancia con lo manifestado, suscribo este documento en el momento que hago la entrega del trabajo final en formato digital a la Biblioteca de la Universidad Politécnica Salesiana.

Cuenca, 23 de julio de 2025

Atentamente,



Juan Daniel Guamán Pintado
0105206585

CERTIFICADO DE DIRECCIÓN DEL TRABAJO DE TITULACIÓN

Yo, Christian Raúl Salamea Palacios con documento de identificación N° 0102537180, docente de la Universidad Politécnica Salesiana, declaro que bajo mi tutoría fue desarrollado el trabajo de titulación: MIGRACIÓN DEL SOFTWARE ROS1 A ROS2 EN EL SISTEMA OPERATIVO DEL ROBOT ROVER PRAX-IA DEL GRUPO DE INVESTIGACIÓN GIIRA, realizado por Juan Daniel Guamán Pintado con documento de identificación N° 0105206585, obteniendo como resultado final el trabajo de titulación bajo la opción Proyecto técnico que cumple con todos los requisitos determinados por la Universidad Politécnica Salesiana.

Cuenca, 23 de julio de 2025

Atentamente,

Ing. Christian Raúl Salamea Palacios, PhD.
0102537180

AGRADECIMIENTOS

Quiero expresar mi más sincero y respetuoso agradecimiento a todas las personas que hicieron posible la realización de este proyecto de grado. En primer lugar agradezco a Dios por ser mi guía no solo en este proyecto, sino a lo largo de toda mi carrera universitaria.

Deseo también agradecer a mi familia, en especial a mi madre quien es la persona más importante en mi vida, y quien me apoyó en esta etapa de mi vida. A mis hermanos quienes siempre tuvieron palabras de aliento cuando, por distintas situaciones, pensé en dejar mi carrera de lado.

Agradezco de manera muy sincera a mi tutor, el Dr. Christian Salamea quien con su ayuda, motivación y paciencia, fué pieza clave para lograr la realización de este trabajo, asímismo a los ingenieros Michael y Javier, ya que, sin ellos yo no hubiera podido realizar el presente proyecto.

Finalmente, quisiera reconocer el esfuerzo del personal docente y administrativo de la carrera de Ingeniería Electrónica, quienes han contribuido a mi formación durante todos estos años.

Juan Daniel Guamán Pintado

DEDICATORIA

Dedicatoria de Juan Daniel Guamán Pintado

A mi papá, allá en el cielo...

Índice general

Agradecimientos I

Dedicatoria II

Índice General III

Índice de figuras VI

Índice de tablas VII

Resumen VIII

Abstract IX

Antecedentes 1

Justificación 4

Objetivos 5

Introducción 6

1. Fundamentos teóricos y arquitectura del robot PRAX-IA 8

1.1. Sistema Operativo para Robots (ROS) 8

1.1.1. Conceptos clave del sistema ROS 9

1.1.2. Comparativa entre ROS 1 y ROS 2 10

1.1.3. Distribuciones de ROS 2 11

1.2. Robot Rover PRAX-IA 13

Robot Rover PRAX-IA 13
1.3. Arquitectura de software del rover PRAX-IA 14
1.3.1. Unidades principales del sistema 15
1.3.2. Arquitectura de nodos en ROS del rover PRAX-IA 15

2. Instalación del sistema operativo ROS 2 19
2.1. Criterios de selección de la distribución ROS 2 Humble 19
2.2. Consideraciones técnicas de compatibilidad entre 21
2.3. Contenedores Docker como solución de compatibilidad. 22
2.4. Infraestructura Isaac ROS Build Farm CDN de NVIDIA 22
2.5. Preparación del entorno de instalación 23
2.5.1. Configuración regional (locale) 23
2.5.2. Instalación de herramientas y dependencias básicas 24
2.5.3. Registro del repositorio de Isaac ROS Build Farm 24
2.5.4. Inclusión del repositorio oficial de ROS 2 24
2.6. Instalación de ROS 2 Humble y verificación del entorno 25

3. Evaluación del desempeño del sistema migrado 26
3.1. Uso de rosserial sobre micro-ROS 26
3.2. Configuración y desarrollo de nodos en Python 28
3.2.1. Uso de Visual Studio Code como entorno de desarrollo 28
3.2.2. Creación del paquete ROS 2 en Python 29
3.2.3. Descripción y estructura de los nodos en ROS 2 29
3.2.4. Construcción y ejecución del paquete ROS 2 32
3.2.5. Ejecución del nodo sensor_reader_node 33
3.3. Configuración básica del Arduino para envío de datos 34
3.3.1. Código en Arduino para adquisición de datos de sensores 34
3.4. Visualización de Datos en RViz 37
3.4.1. Procedimiento para abrir RViz y visualizar los datos 37
3.4.2. Visualización de la Temperatura 38
3.4.3. Visualización de Luz Ambiental 39

4. Conclusiones y Recomendaciones **41**

4.1. Conclusiones 41

4.2. Recomendaciones 42

Referencias **46**

Índice de figuras

1.1. Logos de algunas versiones de ROS 2. 11
1.2. Robot Rover PRAX-IA perteneciente al grupo GIIRA. 13
1.3. Sensores integrados en el Rover PRAX-IA. 14
1.4. Resumen de comunicación entre nodos 15

2.1. Logo Ros 2 "Humble Hawksbill". 20

3.1. Diagrama de flujo del código en VSCode. 32
3.2. Visualización de datos enviados y recibidos en la consola. 33
3.3. Diagrama de flujo del código en Arduino para los sensores. 36
3.4. Conexión entre sensores, Arduino y Jetson Xavier con ROS 2 37
3.5. Visualización en RViz de la lectura de temperatura (20 °C) para Cuenca, Ecuador. 39
3.6. Visualización en RViz de la lectura del sensor de luz ambiental en diferentes condiciones. 40

Índice de tablas

1.1. Resumen de nodos, tópicos y funciones en ROS1 del PRAX-IA 18

Resumen

El presente trabajo detalla la implementación del Sistema Operativo para Robots 2(ROS 2, del inglés Robot Operation System 2), en su versión llamada Humble para el robot de exploración PRAX-IA, desarrollado por el Grupo de Investigación en Interacción Robótica y Automática (GIIRA), de la Universidad Politécnica Salesiana. El propósito principal fue migrar de ROS 1 a ROS 2, asegurando que, la nueva plataforma mantuviera la funcionalidad, estabilidad y desempeño del sistema original instalado anteriormente. Dado que el sistema base de la Jetson Xavier AGX opera sobre Ubuntu 20.04, y ROS 2 Humble requiere oficialmente como base para operar el sistema operativo Ubuntu 22.04, se procedió a adoptar una solución técnica, la cual se basa en el uso del Isaac ROS Buildfarm CDN, el mismo que, permite instalar paquetes compatibles sin alterar el sistema operativo de fábrica, en nuestro caso el sistema operativo Ubuntu 20.04.

Inicialmente, se analizó la arquitectura de software en ROS 1 del robot PRAX-IA, documentando los nodos, dependencias y configuraciones para obtener información. Posteriormente, se ejecutó el proceso de migración mediante contenedores Docker personalizados, para lograr instalar ROS 2 en un entorno aislado sin comprometer los controladores de hardware ni el entorno JetPack preinstalado de la Jetson Xavier. Finalmente, se evaluó el funcionamiento del sistema, verificando la correcta ejecución de nodos, tópicos y estructuras de comunicación en tiempo real.

Como resultado, se logró una migración funcional y documentada del sistema operativo, sin afectar el rendimiento del robot ni sus capacidades actuales.

Palabras clave: Contenedor Docker; GIIRA; Isaac ROS Buildfarm; Jetson Xavier AGX; Migración ROS 1 a ROS 2; PRAX-IA; ROS 2 Humble.

Abstract

This paper details the implementation of the Robot Operation System 2 (ROS 2), in its Humble version, for the PRAX-IA exploration robot, developed by the Artificial Intelligence and Autonomous Robotics Research Group (GIIRA) at the Salesian Polytechnic University. The main objective was to migrate from ROS 1 to ROS 2, ensuring that the new platform maintained the functionality, stability, and performance of the original system installed previously. Since the Jetson Xavier AGX base system runs on Ubuntu 20.04, and ROS 2 Humble officially requires the Ubuntu 22.04 operating system, a technical solution was adopted based on the use of the Isaac ROS Buildfarm CDN, which allows the installation of compatible packages without altering the factory operating system, in our case the operating system. Initially, the software architecture of the PRAX-IA robot was analyzed in ROS 1, documenting nodes, dependencies, and configurations to obtain information. Subsequently, the migration process was executed using custom Docker containers to install ROS 2 in an isolated environment without compromising the hardware drivers or the JetPack environment preinstalled on the Jetson Xavier. Finally, the system's operation was evaluated, verifying the correct execution of nodes, topics, and real-time communication structures. As a result, a functional and documented migration of the operating system was achieved, without affecting the robot's performance or its current capabilities.

Keywords: Docker Container; GIIRA; Isaac ROS Buildfarm; Jetson Xavier AGX; ROS 1 to ROS 2 Migration; PRAX-IA; ROS 2 Humble.

Antecedentes

En las últimas décadas se han producido importantes avances en la robótica móvil, principalmente gracias al desarrollo de robots de exploración capaces de realizar y desarrollar tareas de navegación y reconocimiento mediante sensores y sistemas capaces de recoger información en forma de mapas que luego son utilizados para la exploración humana de lugares remotos.

En este contexto, la Administración Nacional de Aeronáutica y del Espacio (NASA, del inglés National Aeronautics and Space Administration) es el principal referente en el desarrollo de robots tipo rover. Esta agencia estadounidense ha desarrollado varios rovers para la exploración de Marte como parte de sus proyectos espaciales, el más importante de los cuales hasta la fecha es "Perseverance", que se lanzó en 2020 y aterrizó en 2021. Al igual que los demás, este róver está equipado con tecnología de vanguardia, incluyendo un avanzado sistema de cámara de alta resolución que contribuye al análisis de la geología mineral de la superficie marciana [1]. Otro sofisticado instrumento electrónico a bordo es el Analizador de Dinámica Ambiental de Marte (MEDA, del inglés Mars Environmental Dynamics Analyzer), que recoge diversos datos meteorológicos y mide parámetros como la velocidad y dirección del viento, la temperatura ambiente, la humedad y la concentración, y el tamaño de las partículas de polvo en la atmósfera marciana [2].

Más allá de las instituciones gubernamentales o industriales dedicadas a la exploración espacial, varias instituciones han invertido parte de sus recursos en el desarrollo de rovers para el estudio y desarrollo de sistemas con una amplia gama de aplicaciones, desde la investigación científica hasta la gestión urbana y la agricultura. Una de estas instituciones es la Universidad Politécnica Salesiana (UPS), donde uno de sus grupos de investigación, concretamente el Grupo de Investigación en Interacción,

Robótica y Automática (GIIRA), cuenta con un robot rover que funciona como una "Plataforma Robótica de Experimentación con Inteligencia Artificial" y que por tal motivo ha sido llamado "PRAX-IA". El robot ha venido en una constante evolución desde el año 2023, cuando inició su diseño, construcción y programación, con el trabajo de Farfán y Gonzales [3] y que continúa actualmente con otros actores. El rover está equipado con varios sensores para recopilar continuamente datos de telemetría sobre diversos parámetros atmosféricos, como temperatura, luz ambiental, humedad, luz ultravioleta, vapor, gas o CO_2. Estos datos se muestran en una computadora conectada remotamente mediante el Sistema Operativo del Robot (ROS, del inglés Robot Operation System) [4]. Este rover forma parte actualmente del Grupo de Investigación en Interacción Robótica y Automática (GIIRA) de la misma universidad, donde se utiliza para el desarrollo y la experimentación en robótica y exploración autónoma.

Entre proyectos similares, destaca el desarrollo del robot desbrozadora todoterreno con amortiguación, desarrollado en 2024 por Campos y Tomalá [5], proyecto que partió de la premisa de que el desarrollo de robots agrícolas todoterreno con suspensión de bogie basculante ofrece una oportunidad crucial para mejorar la sostenibilidad y la eficiencia en la agricultura. Estos robots aprovechan su capacidad de analizar el terreno a partir de imágenes o vídeos para realizar diversas tareas, como la aplicación de fertilizantes y pesticidas, la recolección de muestras y el riego, entre otras funciones importantes. Este enfoque ha dado como resultado un robot autónomo que opera sin instrucciones directas. Esto se ha logrado mediante un sistema de comunicación por radiofrecuencia que facilita la integración en el sector agrícola y ofrece nuevas soluciones para los agricultores. Dado que este proyecto se basa en la misma arquitectura de hardware que el rover PRAX-IA, sirve de referencia para su desarrollo.

En 2023, Bravo [6] desarrolló un sistema de navegación autónomo para un robot móvil con accionamiento diferencial con el fin de mejorar sus capacidades de navegación en entornos complejos. Este sistema, basado en ROS (Robot Operating System), permite al robot moverse de forma autónoma mientras detecta y evita obstáculos en su entorno. Una de las características más destacadas de esta

implementación es su capacidad para mapear el área de navegación en tiempo real y recopilar información importante para planificar rutas óptimas. Esto no solo aumenta la eficiencia del movimiento, sino que también mejora la seguridad del robot al reducir el riesgo de colisión. Además, la integración de sensores avanzados y algoritmos de planificación de rutas permite al robot adaptarse dinámicamente a los cambios en su entorno. Esto lo hace ideal para aplicaciones en zonas de difícil acceso, como invernaderos, terrenos irregulares o instalaciones industriales con obstáculos móviles.

En el campo de la robótica, ROS desempeña un papel similar al de Android o iOS en el mundo de los smartphones. Este sistema operativo proporciona una plataforma unificada para el desarrollo robótico, lo que facilita la investigación, la prueba de algoritmos robóticos y el desarrollo de aplicaciones a más usuarios, impulsando significativamente el progreso en este campo. Los sistemas operativos para robots son de gran importancia estratégica para el desarrollo de la industria de la robótica inteligente, y varios países compiten actualmente en la investigación de estos sistemas [7].

Introducido por Willow Garage en 2010, ROS se convirtió rápidamente en uno de los sistemas operativos más utilizados en robótica gracias a su facilidad de uso. Es uno de los sistemas más populares en la comunidad científica y tecnológica. Es un entorno de código abierto que facilita la programación, simulación y control de todo tipo de robots. Desde su introducción, ROS se ha convertido en el estándar para el desarrollo de sistemas robóticos, proporcionando herramientas esenciales para la comunicación entre nodos, la gestión de sensores y actuadores, la navegación autónoma y el control de movimiento. Sin embargo, su arquitectura centralizada presenta algunas limitaciones, especialmente en cuanto a escalabilidad y compatibilidad con arquitecturas distribuidas [8].

Para abordar estos desafíos, en 2017 se introdujo ROS2, una versión mejorada basada en el Servicio de Distribución de Datos (DDS), que permite una comunicación en tiempo real más eficiente, una mayor modularidad y un enfoque más orientado a la industria [9].

Justificación

Los constantes avances tecnológicos en robótica han impulsado la evolución de los sistemas operativos utilizados en el desarrollo de robots. En este contexto, la transición de ROS1 a ROS2 es esencial, no solo por la inminente obsolescencia de ROS1, sino también por la necesidad de optimizar la velocidad, la seguridad y la eficiencia de la comunicación entre los componentes del sistema. Sin embargo, la migración de software en robots ya en funcionamiento presenta un importante desafío técnico debido a la falta de documentación estructurada y guiada que facilite este proceso.

El rover PRAX-IA del Grupo de Investigación en Interacción Robótica y Automática (GIIRA) es un sistema desarrollado en ROS1, lo que pone en riesgo su continuidad operativa y su futura capacidad de actualización. La falta de compatibilidad entre las dos versiones y la inminente obsolescencia de ROS1, concretamente en mayo de 2025, requieren una transición que minimice la pérdida funcional y optimice el rendimiento del robot.

Este proyecto es crucial porque desarrolla una metodología estructurada para actualizar el software del rover PRAX-IA, garantizando su funcionalidad y optimizando su rendimiento. La transición de ROS1 a ROS2 no solo facilita la compatibilidad con nuevas tecnologías, sino que también garantiza mejoras en la estabilidad, la seguridad y la comunicación eficiente entre componentes. Dado que ROS1 ya no recibe soporte ni actualizaciones, mantener el sistema en esta versión podría comprometer su funcionalidad y limitar futuras mejoras. Además, la documentación generada servirá de referencia para otros proyectos que requieran una transición similar, promoviendo así el desarrollo y la continuidad de la investigación en robótica en los ámbitos académico y científico.

Objetivos

Objetivo General

- Implementar el sistema operativo ROS2 en el robot de exploración PRAX-IA del grupo de investigación GIIRA, garantizando su funcionalidad a largo plazo mediante la adaptación de sus características actuales en ROS1.

Objetivos específicos:

- Analizar la arquitectura del software del robot PRAX-IA en ROS1, basándose en la información existente del propio robot y de otros sistemas similares, definiendo la configuración necesaria para su actualización.
- Realizar la migración del sistema operativo ROS1 al entorno ROS2 en el rover PRAX-IA, renovando el código y estructura manteniendo la funcionalidad del sistema.
- Evaluar el desempeño del rover PRAX-IA tras la migración a ROS2, realizando pruebas de procesamiento, comunicación y funcionamiento cuantificando las mejoras alcanzadas respecto al sistema operativo anterior.

Introducción

Este documento describe el proceso de migración de software del rover PRAX-IA, un vehículo robótico de exploración desarrollado por el Grupo de Investigación en Interacción Robótica y Automática (GIIRA) de la Universidad Politécnica Salesiana. Esta migración implicó la implementación del Sistema Operativo del Robot (ROS) en su segunda versión, denominada ROS 2 Humble Hawksbill, con el objetivo de reemplazar la infraestructura heredada basada en ROS 1 sin comprometer el rendimiento, la estabilidad ni la funcionalidad del sistema original que operaba el rover.

El rover PRAX-IA se ejecuta actualmente en un sistema Jetson Xavier AGX con Ubuntu 20.04. Sin embargo, la versión Humble de ROS 2 está diseñada oficialmente para Ubuntu 22.04, lo cual supone una limitación técnica para sistemas como el mencionado. Actualizar el sistema operativo directamente no solo afectaría al entorno operativo, sino que también aumentaría el riesgo de fallos del sistema. " *JetPack* " optimizado, pero también puede causar problemas de compatibilidad con controladores y bibliotecas importantes.

Ante este desafío, se optó por una solución propuesta por NVIDIA, aprovechar el entorno CDN de Isaac ROS Buildfarm, que permite ejecutar ROS 2 Humble mediante contenedores Docker personalizados en Ubuntu 20.04. Un Docker es una herramienta que permite ejecutar aplicaciones dentro de contenedores, que son entornos aislados con todo lo necesario para funcionar. Esto asegura que una aplicación se ejecute igual en cualquier equipo, sin importar el sistema operativo o las configuraciones. Además, es más ligero y rápido que usar máquinas virtuales. Este enfoque permitió mantener intacto el sistema base de Jetson y ejecutar el nuevo entorno de desarrollo en un contenedor aislado y controlado.

Para garantizar una transición exitosa, se inició un análisis detallado de la arquitectura del software en ROS 1. Esto implicó identificar nodos clave, dependencias y configuraciones. Posteriormente, se implementó el nuevo entorno y se realizaron pruebas de validación para garantizar el correcto funcionamiento del sistema en ROS 2. Esto incluyó la comunicación adecuada entre nodos, la correcta implementación de tópicos y la ejecución estable de los módulos del robot.

El trabajo se centró exclusivamente en modificar el sistema operativo sin modificar el hardware ni implementar nuevas funciones. No obstante, los resultados obtenidos sientan las bases para futuras mejoras y prolongan la vida útil del sistema existente mediante el uso de una plataforma moderna, segura y escalable.

Capítulo 1

Fundamentos teóricos y arquitectura del robot PRAX-IA

El objetivo de este capítulo es comprender los conceptos fundamentales del desarrollo e implementación de la segunda versión del Sistema Operativo del Robot (ROS) en el contexto del robot PRAX-IA. Una visión general sistemática presenta los elementos esenciales del entorno de desarrollo para sistemas robóticos modernos y justifica las decisiones tomadas en este trabajo.

Primero, se analiza la evolución de ROS 1 a ROS 2, destacando las mejoras en áreas como la comunicación entre nodos, la seguridad, la compatibilidad con sistemas en tiempo real y la escalabilidad.

Además, se ofrece una visión general del software del robot PRAX-IA, incluyendo sus componentes principales, como sensores, controladores y módulos de procesamiento. Esto proporciona una visión técnica suficiente para comprender la necesidad de adaptarse a la nueva arquitectura de ROS 2.

1.1. Sistema Operativo para Robots (ROS)

El Sistema Operativo de Robots (ROS) es una solución de código abierto que aborda la necesidad crítica de intercambio de información para la detección, el control, la planificación, la simulación y la implementación de robots. No debe confundirse con una biblioteca, sino con un ecosistema de software (el término

"sistema operativo"podría resultar demasiado fuerte) que facilita la integración, el mantenimiento y la implementación de nuevas funciones y hardware, desde la simulación hasta la implementación física [10]. Su estructura permite una gestión eficiente de la comunicación entre los diferentes módulos del sistema, la estandarización de los formatos de mensaje y la coordinación de las transformaciones entre sistemas de referencia. Además, proporciona una base sólida para la integración de sensores, actuadores y algoritmos de control, lo que promueve la escalabilidad y la reutilización de código en diferentes proyectos de robótica.

1.1.1. Conceptos clave del sistema ROS

Nodos. Los nodos son las unidades de procesamiento básicas en ROS. Cada nodo es un proceso independiente que realiza tareas específicas, como leer sensores, procesar datos o controlar actuadores. Estos nodos trabajan juntos para formar sistemas distribuidos complejos.

Master. En ROS 1, el master o roscore es un nodo central que administra la comunicación entre nodos, facilitando el registro y descubrimiento de publicadores y suscriptores. En ROS 2, esta característica se eliminó para dejar espacio para una arquitectura distribuida sin nodos maestros, mejorando la escalabilidad y la robustez.
Tópicos (Topics). Los tópicos son canales de comunicación basados en el patrón publicador-suscriptor. Los nodos pueden publicar mensajes en un tema o suscribirse para recibirlos, lo que permite el intercambio asincrónico de información.

Servicios (Services). Los servicios permiten la comunicación sincrónica entre nodos mediante un esquema de llamada y respuesta, lo cual es útil para operaciones que requieren interacción directa y espera de resultados.

Acciones (Actions). Las acciones amplían los servicios para tareas que pueden consumir mucho tiempo o ser anticipadas, brindando retroalimentación continua a medida que se ejecutan.

Mensajes. Los mensajes son estructuras de datos que se transmiten entre temas, servicios o acciones. Se definen en archivos específicos y pueden contener datos simples o complejos.

Parámetros. Los parámetros son variables configurables que permiten a los

nodos cambiar su comportamiento en tiempo de ejecución sin tener que volver a compilar el código.

Paquetes (Packages). Los paquetes son unidades de distribución de software en ROS que agrupan nodos, mensajes, servicios, configuraciones y documentación relacionada para que puedan administrarse y reutilizarse fácilmente.

Workspace. El workspace es donde se desarrollan, compilan y organizan los paquetes ROS, lo que facilita la gestión de dependencias y versiones.

Archivos de lanzamiento (Launch files). Los archivos de lanzamiento permiten iniciar múltiples nodos y configurar parámetros de forma automática y organizada, lo que simplifica la implementación del sistema.

1.1.2. Comparativa entre ROS 1 y ROS 2

ROS1, la primera generación de este marco, obtuvo una amplia aceptación en la investigación y el desarrollo robótico gracias a su comunidad activa y su estructura flexible. Sin embargo, con el tiempo, se hicieron evidentes las limitaciones en escalabilidad, seguridad y compatibilidad con sistemas distribuidos en tiempo real, lo que condujo al desarrollo de ROS2. Esta nueva versión se basa en tecnologías más robustas, como el Servicio de Distribución de Datos (DDS, del inglés Data Distribution Service), que permite una mayor eficiencia en la comunicación, seguridad integrada, control de calidad del servicio y una mejor compatibilidad con sistemas embebidos y de producción [11].

ROS 1 se lanzó en 2010 y se consolidó como el estándar de facto para aplicaciones robóticas en entornos académicos y de investigación. Su arquitectura, basada en la comunicación centralizada a través del nodo maestro (ROScore), facilitó el desarrollo inicial, pero con el tiempo presentó limitaciones, especialmente en entornos industriales o distribuidos. Algunos de sus inconvenientes incluyen:

- Falta de soporte nativo para sistemas en tiempo real.
- Dificultad para escalar a múltiples dispositivos o redes distribuidas.
- Limitaciones en la seguridad de la comunicación.

- Soporte hasta mayo de 2025.

ROS 2, cuyo desarrollo comenzó formalmente en 2014, fue diseñado para superar estas limitaciones. Entre las principales mejoras se encuentran:

- Eliminación del nodo maestro, permitiendo una arquitectura completamente distribuida.
- Integración de DDS (Data Distribution Service) como middleware de comunicación, que permite definir políticas de Calidad de Servicio (QoS).
- Soporte nativo para sistemas en tiempo real y plataformas embebidas.
- Mayor seguridad mediante autenticación y cifrado de mensajes.
- Compatibilidad multiplataforma, permitiendo la ejecución en sistemas Linux, Windows y macOS.

Estas mejoras posicionan a ROS 2 como una herramienta robusta y escalable para proyectos de robótica modernos que requieren interoperabilidad, modularidad y ejecución confiable en condiciones exigentes, así como soporte a largo plazo para proyectos futuros.

1.1.3. Distribuciones de ROS 2

En la figura 1.1 se pueden observar varios de los logos de algunas distribuciones de ROS 2.

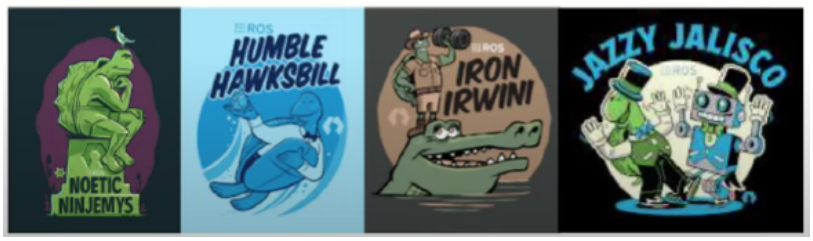


Figura 1.1: Logos de algunas versiones de ROS 2.

Desde su lanzamiento, se han publicado varias distribuciones oficiales de ROS 2, que han mejorado su funcionalidad y estabilidad, además de ser compatibles con

nuevas plataformas. Las distribuciones más importantes se describen brevemente a continuación:

- **Ardent Apalone (2017):** Primera distribución estable de ROS 2, centrada en sentar las bases para el middleware DDS y proporcionar soporte básico de nodos y comunicaciones. Su soporte finalizó en diciembre de 2018.
- **Bouncy Bolson (2018):** Introdujo mejoras en la calidad de servicio, mejor integración con sistemas embebidos y ampliación del soporte multiplataforma. Tuvo soporte hasta julio de 2019.
- **Crystal Clemmys (2019):** Se enfocó en la estabilidad y en mejorar herramientas de desarrollo, incluyendo el soporte para Windows y mejoras en la seguridad. Su ciclo de soporte terminó en diciembre de 2019.
- **Dashing Diademata (2019):** Primera distribución con soporte a largo plazo (LTS), ofreciendo mayor estabilidad para aplicaciones industriales y autónomas. Su soporte se extendió hasta mayo de 2021.
- **Eloquent Elusor (2020):** Mejoras en rendimiento, usabilidad y nuevos paquetes para facilitar el desarrollo de aplicaciones complejas. Fue mantenida hasta noviembre de 2020.
- **Foxy Fitzroy (2020):** Segunda versión LTS, ampliamente adoptada por la comunidad, consolidó muchas características clave y mejoró la compatibilidad con micro-ROS. Su soporte se mantendrá hasta mayo de 2023.
- **Humble Hawksbill (2022):** Última versión LTS hasta la fecha, optimizada para aplicaciones robóticas modernas, con mejor integración con plataformas NVIDIA y soporte extendido hasta mayo de 2027.
- **Iron Irwini (2023):** Lanzamiento más reciente que continúa con mejoras en rendimiento, estabilidad y soporte multiplataforma, preparando la transición hacia futuros estándares. Su soporte terminó noviembre de 2024.

1.2. Robot Rover PRAX-IA

El rover PRAX-IA es un robot móvil diseñado para la investigación y la educación académica. Fue construido para permitir la navegación autónoma en terrenos difíciles, la visión artificial y la inteligencia artificial en robótica. Como se puede observar en la figura 1.2 el rover está diseñado de forma que pueda navegar en dichos terrenos.

Figura 1.2: Robot Rover PRAX-IA perteneciente al grupo GIIRA.

Segun el autor Ismail Hakki :"La navegación es una operación fundamental compuesta por muchas funciones y rutinas de un robot móvil y varias implementaciones en forma de bibliotecas y paquetes están disponibles para ROS. Por lo tanto, la pila de navegación podría asumirse como la colección de las herramientas integradas que proporcionan las funciones necesarias para navegar por un robot móvil"[12]. Por esta razón, el rover PRAX-IA está diseñado como una plataforma terrestre con capacidades de teleoperación, con navegación controlada remotamente mediante una interfaz externa. Su arquitectura se basa en ROS como sistema de control y comunicación, lo que permite la integración modular de sensores y nodos de procesamiento. Esta tecnología proporciona al rover un entorno flexible y escalable para la monitorización, la exploración y la adquisición de datos en tiempo real.

El robot está equipado con una gama de sensores ambientales, incluyendo detectores de gas, sensores de luz, sensores de vapor, sensores de radiación UV y sensores de temperatura y humedad, como se aprecia en la figura 1.3. Estos elementos permiten al sistema recopilar información importante sobre las condiciones

atmosféricas de su entorno. Esto lo convierte en una herramienta versátil para aplicaciones como la exploración científica, el análisis ambiental o el patrullaje de zonas remotas.

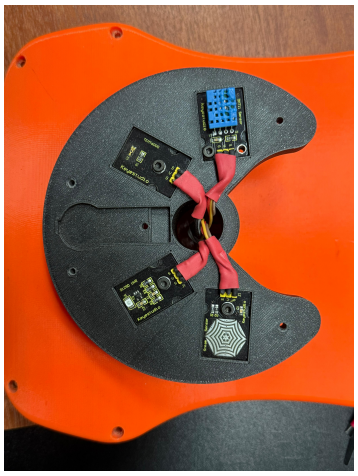


Figura 1.3: Sensores integrados en el Rover PRAX-IA.

Una característica distintiva del rover es su capacidad para navegar tanto en terrenos llanos como irregulares. Esto se logra gracias a una robusta configuración mecánica y un sistema de propulsión adecuado. Esta adaptabilidad lo hace ideal para entornos desestructurados y subraya su utilidad en tareas donde la movilidad y la resistencia física son cruciales [13].

1.3. Arquitectura de software del rover PRAX-IA

El robot PRAX-IA cuenta con una arquitectura de software basada en el Sistema Operativo de Robot (ROS), versión ROS1 Melodic. Este sistema permite la ejecución modular de nodos que se comunican mediante temas y servicios. La ejecución principal se realiza sobre una unidad NVIDIA Xavier AGX con Ubuntu 20.04.

La arquitectura consta de módulos de percepción, control, procesamiento y monitorización. La integración de sensores, actuadores y monitorización en tiempo real se logra mediante herramientas como Rosserial, RViz y ROS Mobile.

1.3.1. Unidades principales del sistema

- **NVIDIA Jetson Xavier AGX:** ejecuta los nodos ROS y es el núcleo computacional.
- **Arduino Mega 2560:** encargado del control de motores y lectura de sensores.
- **RealSense D455:** cámara RGB-D integrada mediante el nodo `realsense2_camera`.
- **Tablet Android:** usada como interfaz de usuario mediante la app ROS Mobile.

La NVIDIA Xavier AGX se comunica con el Arduino Mega a través de conexión serial, utilizando el paquete `rosserial_python`. El sistema está distribuido en nodos especializados que permiten separar tareas de percepción, navegación y control, según se desee.

1.3.2. Arquitectura de nodos en ROS del rover PRAX-IA

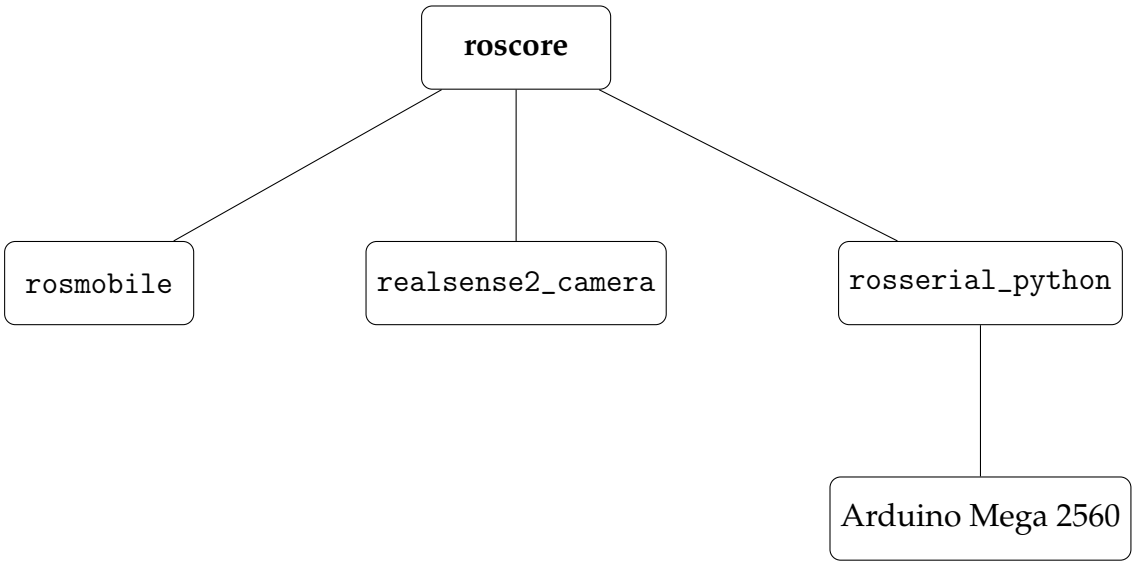


Figura 1.4: Resumen de comunicación entre nodos

El sistema operativo del rover PRAX-IA se basa en la arquitectura de nodos característica del Sistema Operativo del Robot (ROS) en su versión Melodic. Cada nodo realiza una función específica dentro del sistema y se comunica con otros nodos mediante temas o servicios, lo que permite una operación distribuida y modular.

Los nodos principales utilizados en el sistema PRAX-IA se dividen en tres grupos según su función: percepción, control y soporte.

Nodos de percepción

- `/realsense2_camera`: nodo responsable de adquirir y publicar los datos de la cámara Intel RealSense D455. Este nodo emite información de imagen RGB, profundidad y alineación entre ambas mediante tópicos como: `/camera/color/image_raw` y `/camera/depth/image_raw`.
- `/sensor_ambiental_node`: nodo desarrollado para leer datos de sensores conectados al Arduino Mega (temperatura, gas, humedad, luz). Publica datos en tópicos específicos como `/sensor/temperatura`, `/sensor/gas`, etc.

Nodos de control

- `/cmd_vel`: tópico principal para enviar comandos de velocidad lineal y angular al rover. Es suscrito por el nodo de bajo nivel ubicado en el Arduino Mega, que ejecuta los movimientos.
- `/arduino_bridge_node`: nodo que comunica ROS con el microcontrolador a través de `rosserial_python`. Se encarga de recibir mensajes de velocidad desde `/cmd_vel` y de enviar información de odometría o sensores hacia ROS.

Nodos de soporte y visualización

- `/rosout`: nodo estándar de ROS que publica mensajes de log y depuración.
- `/rviz`: herramienta de visualización 3D que suscribe los tópicos publicados por la cámara, sensores y odometría, permitiendo monitorear el estado del sistema.
- `/rosserial_python`: nodo puente para establecer la conexión serial entre ROS y el microcontrolador Arduino.
- `/rosmobile`: nodo creado mediante la aplicación ROS Mobile en Android, utilizado para el control remoto del robot mediante interfaz táctil. Se suscribe a tópicos como `/cmd_vel` y visualiza estados del sistema.

Comunicación entre nodos

El nodo maestro (`roscore`) gestiona el registro y la comunicación entre todos los nodos. La NVIDIA Xavier AGX ejecuta el nodo maestro, el nodo de la cámara, el nodo de interfaz serial y los nodos de sensores. El Arduino Mega, por su parte, ejecuta código embebido que responde a los mensajes de ROS mediante `rosserial`. El nodo `/cmd_vel` es publicado por la aplicación móvil ROS Mobile y suscrito por el nodo del Arduino para ejecutar comandos de movimiento. La cámara Intel RealSense D455 publica imágenes RGB y datos de profundidad que pueden ser visualizados en tiempo real a través de `RViz`. Los sensores ambientales también transmiten continuamente sus valores medidos (temperatura, gas, humedad, luz) a dispositivos específicos para su monitorización remota o posterior procesamiento. Esta integración distribuida permite el funcionamiento autónomo y coordinado del sistema y facilita la percepción, el control y la monitorización del entorno en tiempo real.

Esta arquitectura distribuida permitió realizar pruebas de campo, controlar el robot desde una tableta Android, visualizar datos en tiempo real y una estructura de software fácilmente escalable.

A continuación en la tabla 1.1 se muestra una vista general con detalles de los diferentes nodos del sistema:

Tabla 1.1: Resumen de nodos, tópicos y funciones en ROS1 del PRAX-IA

Nodo ROS	**Tópico principal**	**Función**
`/roscore`	`/roscore`	Nodo maestro. Coordina todos los demás nodos del sistema.
`/rosserial_python`	`/cmd_vel`, `/odom`, `/sensor`	Nodo puente entre ROS y Arduino mediante comunicación serial.
`/realsense2_camera`	`/camera/color/image`, `/camera/depth/image`	Nodo de la cámara RealSense D455. Publica imágenes RGB y de profundidad.
`/sensor_ambiental_node`	`/sensor/temperatura`, `/sensor/humedad`, `/sensor/gas`	Nodo de adquisición de datos ambientales.
`/rosmobile`	`/cmd_vel`, `/camera/image_raw`	Nodo cliente en Android para control remoto del robot.
`/rviz`	Visualiza múltiples tópicos	Herramienta de visualización 3D para monitorear el sistema.

Capítulo 2

Instalación del sistema operativo ROS 2

Este capítulo aborda la compleja tarea de migrar el entorno de desarrollo del robot PRAX-IA, implementado originalmente en el Sistema Operativo Robot (ROS) versión 1, a su sucesor, ROS 2, concretamente a su distribución Humble Hawksbill. Este proceso de migración requiere no solo una reestructuración del software y los paquetes utilizados, sino también una cuidadosa adaptación al entorno de hardware en el que se ejecuta el sistema; en este caso, la plataforma NVIDIA Jetson Xavier AGX. Esta plataforma ejecuta Ubuntu 20.04 LTS por defecto, lo cual supone una limitación, ya que ROS 2 Humble se desarrolló para ejecutarse en Ubuntu 22.04. Este capítulo detalla las soluciones técnicas implementadas para esta migración sin modificar el sistema operativo base. Esto garantiza la estabilidad y la compatibilidad del sistema y permite una implementación moderna sin comprometer el rendimiento ni las funcionalidades del robot.

2.1. Criterios de selección de la distribución ROS 2 Humble

Durante el proceso de migración de ROS 1 a ROS versión 2, fue necesario seleccionar cuidadosamente la distribución más adecuada.Aunque existen versiones más recientes de ROS 2 como Iron Irwini o Jazzy Jalisco (también referida como ROS 2 Kaiyu), se optó por la distribución Humble Hawksbill debido a razones técnicas, prácticas y de soporte a largo plazo. En la figura 2.1 se observa el logo de dicha

distribución.

Figura 2.1: Logo Ros 2 "Humble Hawksbill".

En primer lugar, Humble es una distribución LTS (Long Term Support), lo que significa que cuenta con soporte extendido oficial hasta mayo de 2027 por parte de Open Robotics [14]. Este soporte extendido garantiza actualizaciones de seguridad, correcciones de errores críticos y estabilidad del sistema durante varios años, lo cual es fundamental en aplicaciones robóticas donde la confiabilidad del entorno operativo es una prioridad máxima.

Por el contrario, versiones como Kaiyu ofrecen nuevas funciones interesantes, pero también conllevan riesgos: menor madurez, cambios frecuentes en la API y posible falta de compatibilidad con bibliotecas optimizadas para plataformas Jetson. Además, al ser una distribución más reciente, la documentación disponible y el soporte de la comunidad aún están en desarrollo, lo que puede alargar los tiempos de desarrollo y aumentar el riesgo de errores en producción.

Otro criterio importante fue la amplia adopción industrial de Humble. Muchas bibliotecas, herramientas de terceros y paquetes de controladores compatibles con hardware especializado, como cámaras, LiDARs y motores, están optimizados y certificados para funcionar en ROS 2 Humble [15]. Esto garantiza una integración de componentes más fluida sin necesidad de personalización adicional o desarrollo de bajo nivel.

Finalmente, la compatibilidad con la infraestructura Isaac ROS Build Farm de NVIDIA fue un factor decisivo. Esta infraestructura proporciona imágenes de Docker optimizadas específicamente para ROS 2 Humble en Ubuntu 20.04, lo que permite una instalación segura sin afectar el sistema base de Jetson Xavier AGX [16].

En conjunto, estos factores posicionan a Humble como la opción más sólida, mejor documentada y sostenible para este proyecto, garantizando una base tecnológica sólida compatible con el hardware en uso sin comprometer la escalabilidad ni la estabilidad futura.

2.2. Consideraciones técnicas de compatibilidad entre

Una de las principales limitaciones identificadas en este trabajo fue la incompatibilidad entre el sistema operativo preinstalado en Jetson Xavier AGX (Ubuntu 20.04) y la distribución ROS 2 Humble, que fue desarrollada exclusivamente para correr en Ubuntu 22.04 [14]. Cambiar el sistema operativo no era una opción viable porque la plataforma Jetson tiene controladores y bibliotecas proporcionadas por NVIDIA que están optimizadas para la versión actual [17]. Una actualización forzada a Ubuntu 22.04 podría resultar en la pérdida de compatibilidad con bibliotecas críticas para el funcionamiento de la cámara, la aceleración de hardware, la comunicación con sensores y otras funciones importantes, incluyendo la pérdida de aceleración de la Unidad de Procesamiento Gráfico(GPU, por sus siglas en inglés Graphics Processing Unit).

Por lo tanto, era necesario encontrar una solución alternativa que permitiera instalar y ejecutar ROS 2 Humble en Ubuntu 20.04 sin modificar el entorno base. Esta solución llegó en forma de contenedores Docker personalizados mantenidos y distribuidos por NVIDIA a través de su infraestructura Isaac ROS Build Farm CDN [16].

2.3. Contenedores Docker como solución de compatibilidad.

Docker es una tecnología para crear y ejecutar contenedores. Estos son entornos virtualizados, pero ligeros, que contienen todo lo necesario para ejecutar una aplicación. A diferencia de una máquina virtual tradicional, Docker comparte el kernel con el sistema operativo host, lo que lo hace significativamente más eficiente en términos de rendimiento y consumo de recursos [18].

En este proyecto, se utilizó Docker para ejecutar una imagen de ROS 2 Humble basada en Ubuntu 22.04, encapsulada en un contenedor que se ejecuta en Ubuntu 20.04. Esto eliminó la necesidad de actualizar el sistema operativo Jetson, conservó los valores predeterminados de fábrica y evitó el riesgo de incompatibilidad con los controladores y bibliotecas de NVIDIA [19].

Este enfoque también facilita la portabilidad del entorno de desarrollo, ya que el contenedor puede replicarse y ejecutarse en otras máquinas sin tener que reconfigurar todo el entorno desde cero. Además, proporciona un alto grado de aislamiento, lo cual resulta útil durante la fase de prueba y validación.

2.4. Infraestructura Isaac ROS Build Farm CDN de NVIDIA

Tras analizar varias alternativas y revisar documentación oficial, informes técnicos y foros de desarrolladores específicos de las plataformas NVIDIA Jetson, concluimos que el sistema CDN Isaac ROS Build Farm es la solución más robusta y eficiente. Esta infraestructura fue desarrollada por NVIDIA para solucionar la incompatibilidad entre ROS 2 Humble y Ubuntu 20.04 sin necesidad de reemplazar ni modificar el sistema operativo base en la plataforma Jetson [16].

Isaac ROS Build Farm es una bifurcación personalizada del Build Farm oficial de ROS 2. A diferencia del repositorio original, este sistema ha sido adaptado para compilar y distribuir paquetes compatibles con Ubuntu 20.04 Focal, tomando en cuenta las dependencias específicas y las bibliotecas optimizadas que requiere el

ecosistema de desarrollo de NVIDIA, como CUDA, TensorRT y DeepStream [20].

Cabe destacar que esta infraestructura proporciona no solo archivos fuente o instrucciones de compilación, sino también imágenes Docker preconfiguradas que se pueden implementar de inmediato. Estas imágenes contienen el entorno completo de ROS 2 Humble en Ubuntu 22.04 y pueden ejecutarse en un contenedor en cualquier sistema basado en Ubuntu 20.04.

La red de distribución (CDN, del inglés Content Delivery Network) empleada por Isaac ROS Build Farm garantiza que los recursos estén disponibles con baja latencia desde cualquier región del mundo, lo cual mejora considerablemente el tiempo de descarga e instalación de paquetes. Además, el uso de firmas digitales y claves GPG garantiza la autenticidad e integridad del software descargado, añadiendo una capa adicional de seguridad al sistema [16].

En resumen, Isaac ROS Build Farm representa una estrategia robusta, eficiente y fácil de mantener para superar los obstáculos que plantea la incompatibilidad de versiones. Esta solución permite que el desarrollo robótico siga aprovechando las últimas tecnologías ROS sin interferir con el sistema operativo nativo del hardware Jetson, garantizando así la estabilidad, el rendimiento y la compatibilidad del proyecto a largo plazo.

2.5. Preparación del entorno de instalación

Al inicio del proceso, fue necesario asegurar que el sistema host, en este caso Ubuntu 20.04, estuviera correctamente configurado regionalmente y contara con herramientas básicas de gestión de paquetes.

2.5.1. Configuración regional (locale)

La configuración regional garantiza que el sistema interprete correctamente los caracteres y formatos de texto. Esto es importante para evitar errores al instalar paquetes que manejan cadenas de texto, archivos de configuración o mensajes en varios idiomas, siendo importante seleccionar la región US, ya que además existe la región CH que pertenece a China.

```
sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
```

2.5.2. Instalación de herramientas y dependencias básicas

Se instalan herramientas como `gnupg` para verificación de firmas, `wget` para descargas, y `software-properties-common` para la gestión avanzada de repositorios. También se habilita el repositorio `universe` de Ubuntu, que contiene muchos paquetes utilizados por ROS.

```
sudo apt update && sudo apt install gnupg wget
sudo apt install software-properties-common
sudo add-apt-repository universe
```

2.5.3. Registro del repositorio de Isaac ROS Build Farm

Para que el sistema pueda instalar paquetes desde los servidores de NVIDIA, es necesario registrar su clave GPG y añadir el repositorio a la lista de fuentes de software del sistema, esto con el objetivo de poder descargar todo lo necesario para poner en marcha la instalación

```
wget -q0 - https://isaac.download.nvidia.com/isaac-ros/repos.key | sudo apt-key add -
echo 'deb https://isaac.download.nvidia.com/isaac-ros/ubuntu/main focal main' | sudo tee -a
↪ /etc/apt/sources.list
```

2.5.4. Inclusión del repositorio oficial de ROS 2

A pesar de usar el Build Farm de NVIDIA, ciertos paquetes siguen dependiendo del repositorio oficial de ROS 2. Por eso, se añade también su llave y fuente, para evitar confusiones en el sistema:

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
↪ /usr/share/keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
↪ http://packages.ros.org/ros2/ubuntu focal main" | sudo tee /etc/apt/sources.list.d/ros2.list >
↪ /dev/null
```

2.6. Instalación de ROS 2 Humble y verificación del entorno

Una vez configurados todos los repositorios y claves, actualizamos la lista de paquetes e instalamos la distribución completa de ROS 2 Humble:

```
sudo apt update
sudo apt install ros-humble-desktop
```

Esta instalación incluye herramientas como RViz, rqt y paquetes de ejemplo que le permiten comenzar a desarrollar de inmediato. Para completar la configuración del entorno, cargue el entorno ROS 2 en la terminal:

```
source /opt/ros/humble/setup.bash
```

Este comando debe incluirse también en el archivo `.bashrc` para que se cargue automáticamente cada vez que se abre una nueva terminal. Con esto, el sistema está completamente preparado para ejecutar proyectos en ROS 2 Humble sobre la Jetson Xavier AGX sin necesidad de alterar el sistema operativo original.

Capítulo 3

Evaluación del desempeño del sistema migrado

tikz

El objetivo de este capítulo es evaluar el rendimiento y la funcionalidad del robot PRAX-IA tras la migración del Sistema Operativo para Robot (ROS) en su versión Melodic, a ROS 2 Humble Hawksbill. El análisis se centra en determinar si se han conservado las funcionalidades originales del sistema, así como las posibles mejoras o limitaciones derivadas de la transición. Para ello, se desarrollaron pruebas específicas que abarcan aspectos clave del funcionamiento del sistema, como la comunicación entre nodos, la ejecución de tareas y la interacción con sensores.

3.1. Uso de rosserial sobre micro-ROS

Durante el desarrollo de este proyecto, se evaluaron diversas alternativas para facilitar la comunicación entre el microcontrolador Arduino MEGA 2560 y el entorno del Sistema Operativo para Robot (ROS, del inglés Robot Operating System) en su versión 2 en la plataforma Jetson Xavier AGX. Las dos opciones principales en el ecosistema ROS eran el protocolo Rosserial y la arquitectura Micro-ROS, más moderna.

Micro-ROS es una solución desarrollada por eProsima en colaboración con la comunidad de ROS 2, diseñada para extender las capacidades del middleware

Servicio de Distribución de Datos (DDS, del inglés Data Distribution Service) a microcontroladores y sistemas embebidos con recursos limitados. A diferencia de rosserial, que actúa como un puente serial simplificado, Micro-ROS permite que el propio microcontrolador actúe como un nodo ROS 2 nativo dentro de la red de comunicación, lo que representa una ventaja significativa en términos de sincronización, robustez y escalabilidad [21].

Sin embargo, la introducción de Micro-ROS está sujeta a ciertas limitaciones de hardware. En particular, su implementación requiere que el microcontrolador sea compatible con un sistema operativo en tiempo real (RTOS, del inglés Real-Time Operating System), como FreeRTOS, NuttX o Zephyr. Además, demanda recursos considerables en cuanto a memoria RAM (al menos 96 KB recomendados) y capacidad de procesamiento. El Arduino MEGA 2560, basado en un microcontrolador ATmega2560 de arquitectura AVR de 8 bits, posee solamente 8 KB de SRAM y no cuenta con soporte nativo para ejecutar un RTOS, lo cual imposibilita técnicamente la implementación de micro-ROS sobre el rover.

Por estas razones, se optó por utilizar rosserial, que, aunque fue originalmente concebida para entornos anteriores a ROS 2, la comunidad ha desarrollado adaptaciones del paquete `rosserial_python` que permiten su funcionamiento parcial con ROS 2, ofreciendo una solución viable en contextos donde el hardware es limitado o no se justifica una migración completa a micro-ROS [22].

Aunque no ofrece todas las ventajas del modelo distribuido de ROS 2, rosserial permitió en este proyecto cumplir con los objetivos funcionales: establecer un canal de comunicación eficiente entre Arduino y ROS 2, recibir datos de sensores en tiempo real, y visualizar los resultados utilizando herramientas como `rqt_plot`. Todo esto se logró sin cambiar de plataforma ni invertir en microcontroladores más avanzados, aprovechando el hardware existente. Para ello, también se empleó un gestor de nodos como VS Code, que utiliza Python como lenguaje de desarrollo.

3.2. Configuración y desarrollo de nodos en Python

En el contexto de la migración del robot PRAX-IA a ROS 2 Humble Hawksbill, se desarrolló un nodo en Python encargado de la adquisición y publicación de datos provenientes de los sensores conectados al Arduino MEGA. Para lograr esto, elegimos utilizar el editor de código Visual Studio Code (VS Code) como entorno de desarrollo integrado debido a su compatibilidad con sistemas Ubuntu, integración con terminales y soporte extendido para Python, lo que facilita la escritura, depuración y ejecución de nodos ROS 2.

Para el desarrollo de la integración entre el Arduino y ROS 2 se optó por utilizar nodos personalizados en Python en lugar de *rosserial*, debido a que ROS 2 no ofrece soporte oficial para esta librería, como ya vimos anteriormente, esta decisión también permite una mayor flexibilidad y control sobre el manejo de la comunicación serial.

3.2.1. Uso de Visual Studio Code como entorno de desarrollo

Visual Studio Code (VS Code) es un editor de código fuente multiplataforma ampliamente utilizado en el desarrollo de software, incluyendo proyectos con ROS 2. Sus ventajas principales para este trabajo incluyen:

- Soporte nativo para Python y ROS 2 mediante extensiones.
- Integración con terminal y control de versiones.
- Facilita la navegación y edición de archivos del workspace ROS 2.

Para comenzar con la creación de el entorno de desarrollo se debe abrir VS Code en la carpeta raíz del workspace ROS 2. Esto se hace desde la terminal de Ubuntu con:

```
cd ~/ros2_ws
code .
```

Este comando abre VS Code en la carpeta actual, permitiendo visualizar y editar los archivos del workspace.

3.2.2. Creación del paquete ROS 2 en Python

Dentro de VS Code, en la terminal integrada o en una terminal externa, se crea el paquete llamado `sensor_reader` para contener el nodo que leerá datos seriales del Arduino y publicará en ROS 2. El comando usado es:

```
ros2 pkg create --build-type ament_python sensor_reader --dependencies rclpy
↪ std_msgs
```

Este comando genera la estructura básica del paquete Python con las dependencias necesarias para ROS 2 (el cliente `rclpy` y el mensaje estándar `std_msgs`).

3.2.3. Descripción y estructura de los nodos en ROS 2

En ROS 2, un nodo es una unidad de ejecución que puede publicar o suscribirse a temas, realizar servicios o acciones y manejar la lógica de control y comunicación. En este proyecto, el nodo `sensor_reader_node.py` tiene la responsabilidad de leer los datos enviados por el Arduino a través del puerto serial, procesarlos y publicarlos en un tópico para que otros nodos o interfaces puedan consumirlos.

Dentro del paquete `sensor_reader`, se crea una carpeta llamada `sensor_reader` donde se ubica el archivo Python con el código del nodo. En VS Code, se crea el archivo:

```
sensor_reader/sensor_reader/sensor_reader_node.py
```

El contenido del archivo es el siguiente:

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32MultiArray
import serial

class SensorReaderNode(Node):
    def __init__(self):
        super().__init__('sensor_reader_node')

        self.publisher_ = self.create_publisher(
            Float32MultiArray,
            'sensores_data',
            10
        )

        self.serial_port = serial.Serial('/dev/ttyUSB0', 115200)

        self.timer = self.create_timer(0.1, self.read_serial_data)

    def read_serial_data(self):
        if self.serial_port.in_waiting:
            line = self.serial_port.readline().decode('utf-8').strip()
            try:

                values = list(map(float, line.split(',')))

                msg = Float32MultiArray(data=values)

                self.publisher_.publish(msg)

            except ValueError:
                self.get_logger().warn('Error al convertir los datos recibidos
                ↪  del serial')

def main(args=None):
    rclpy.init(args=args)
    node = SensorReaderNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()
```

- Se importa la librería `rclpy` que es el cliente ROS 2 para Python, y la clase `Node` para crear nodos.
- Se usa `Float32MultiArray` de `std_msgs` para enviar múltiples datos flotantes (en este caso, los valores de los sensores).
- La librería `serial` permite la comunicación con el Arduino vía puerto serial.
- En el constructor `__init__`, se inicializa el nodo con un nombre único `sensor_reader_node`.
- Se configura el publicador para publicar en el tópico `sensores_data` con un tamaño de buffer 10.
- Se abre el puerto serial con la velocidad 115200 baudios (que debe coincidir con la configuración en Arduino).
- Se crea un temporizador que cada 0.1 segundos ejecuta la función `read_serial_data`.
- La función `read_serial_data` lee las líneas disponibles en el puerto serial, las decodifica, y convierte los valores separados por coma a números flotantes. Luego crea un mensaje ROS con estos datos y lo publica.
- En caso de error en la conversión de datos, el nodo imprime una advertencia para facilitar la depuración.
- La función `main` inicializa el entorno ROS 2, crea el nodo, y ejecuta un ciclo continuo para mantener el nodo activo hasta que se detenga.

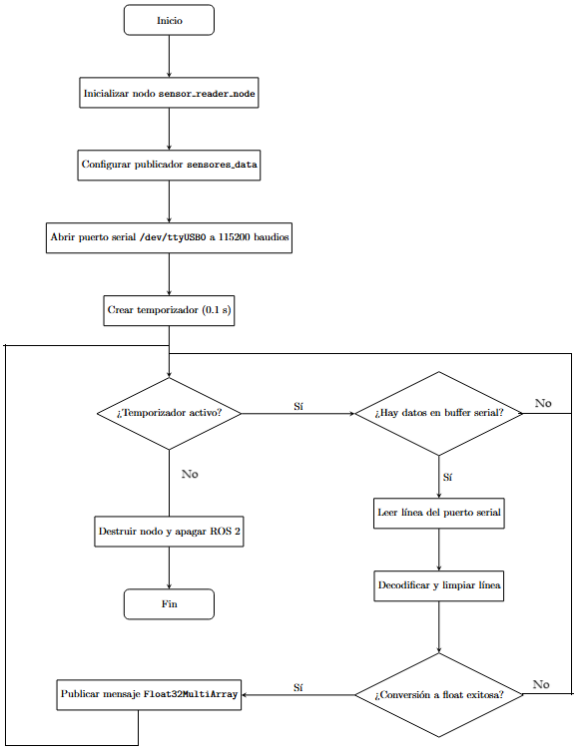


Figura 3.1: Diagrama de flujo del código en VSCode.

3.2.4. Construcción y ejecución del paquete ROS 2

Una vez creado el código del nodo en Python dentro del paquete `sensor_reader`, es necesario compilar el workspace ROS 2 para que el sistema reconozca el nuevo paquete y pueda ejecutarlo.

Para ello, se siguen los siguientes pasos desde la terminal de Ubuntu, preferiblemente dentro de VS Code o en una terminal independiente:

```
# Moverse al directorio raíz del workspace ROS 2
cd ~/ros2_ws

# Construir todos los paquetes del workspace, incluyendo sensor_reader
colcon build

# Fuente del setup para usar los paquetes construidos
source install/setup.bash
```

El comando `colcon build` compila todos los paquetes del workspace, generando los ejecutables, bibliotecas y scripts necesarios para que ROS 2 los pueda cargar y ejecutar.

Posteriormente, el comando `source install/setup.bash` actualiza el entorno de la terminal para reconocer los nuevos paquetes y nodos disponibles.

3.2.5. Ejecución del nodo sensor_reader_node

Con el workspace construido y configurado, se procede a ejecutar el nodo en ROS 2 usando el siguiente comando:

```
ros2 run sensor_reader sensor_reader_node
```

Este comando inicia la ejecución del nodo `sensor_reader_node` contenido en el paquete `sensor_reader`.

Mientras el nodo está en ejecución, se muestran en la consola mensajes informativos con los valores recibidos y publicados, lo que permite verificar que la lectura serial y la publicación de datos se están realizando correctamente, como se observa en la figura 3.2.

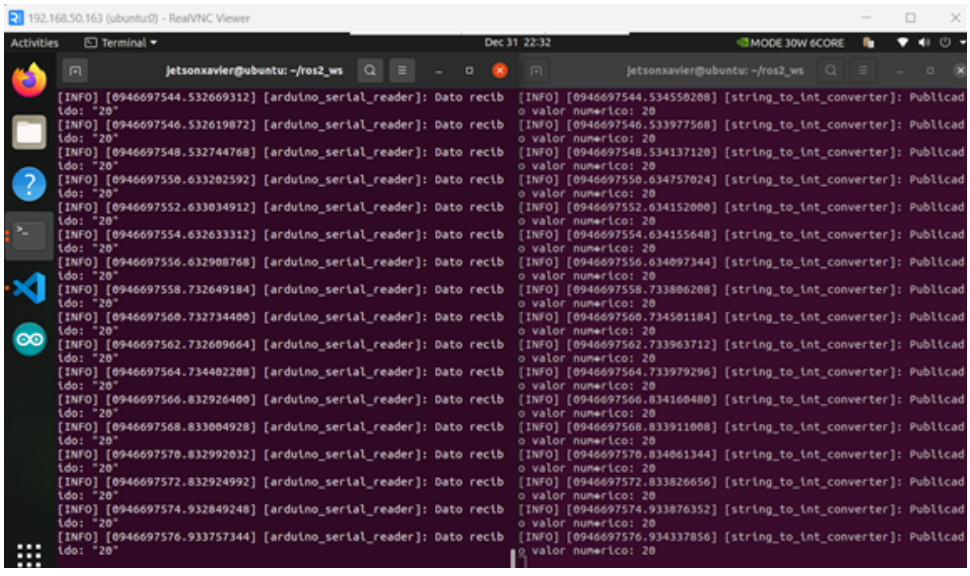


Figura 3.2: Visualización de datos enviados y recibidos en la consola.

3.3. Configuración básica del Arduino para envío de datos

Para que el nodo Python en ROS 2 reciba y procese los datos de los sensores, Arduino debe enviar la información de forma clara y ordenada a través de la interfaz serie. La configuración básica de Arduino incluye:

- Inicializar la comunicación serial a una velocidad estándar, usualmente `115200` baudios, mediante `Serial.begin(115200)` en la función `setup()`.
- Leer los sensores conectados a los pines analógicos, realizando los promedios o filtrados necesarios para obtener valores estables.
- Enviar los datos de los sensores concatenados en una línea separada por comas, por ejemplo: `valor_sensor1,valor_sensor2,valor_sensor3,valor_sensor4`.

Este formato simple permite que el nodo Python interprete fácilmente cada línea recibida como un conjunto de valores separados, facilitando la conversión y publicación en tópicos ROS 2.

3.3.1. Código en Arduino para adquisición de datos de sensores

El código implementado en Arduino se centra en la lectura de cuatro sensores analógicos: un sensor UV, un sensor de luz ambiental, un sensor de vapor y un sensor de gas. Para mejorar la precisión de la medición, se calcula un promedio de múltiples valores de muestra para cada sensor. Las funciones clave para adquirir y procesar señales analógicas se describen a continuación.

```
// Promedia 4 muestras del pin analógico especificado
int averageAnalog(int pin) {
  int v = 0;
  for (int i = 0; i < 4; i++) {
    v += analogRead(pin);
  }
  return v / 4;
```

```
}

// Lectura y mapeo del sensor UV
int averageAnalog_sensor_UV(int pin) {
  int v = 0;
  for (int i = 0; i < 4; i++) {
    v += analogRead(pin);
  }
  int j = map(v / 4, 0, 38, 1, 11);
  return j;
}

// Lectura y cálculo del sensor de luz ambiental (%)
int averageAnalog_sensor_LzAmb(int pin) {
  int v = 0;
  for (int i = 0; i < 4; i++) {
    v += analogRead(pin);
  }
  float light = v / 4 * 0.0976; // Conversión a porcentaje
  return (int)light;
}

// Lectura y mapeo del sensor de vapor (% humedad)
int averageAnalog_sensor_vapor(int pin) {
  int v = 0;
  for (int i = 0; i < 4; i++) {
    v += analogRead(pin);
  }
  int y = map(v / 4, 0, 1023, 40, 100);
  return y;
}

// Lectura y mapeo del sensor de gas (ppm)
int averageAnalog_sensor_gas(int pin) {
  int v = 0;
  for (int i = 0; i < 4; i++) {
```

```
    v += analogRead(pin);
  }
  int y = map(v / 4, 25, 1000, 250, 1000);
  return y;
}
```

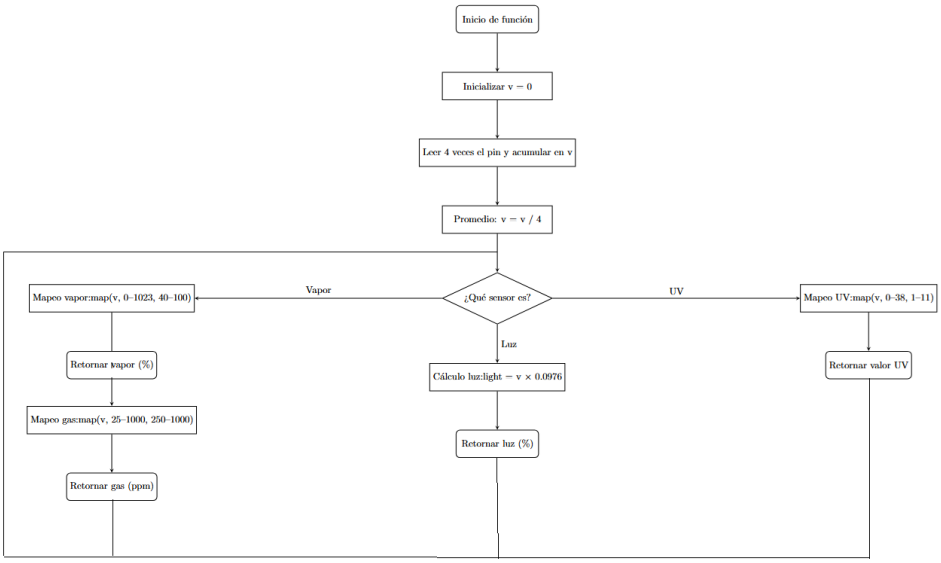


Figura 3.3: Diagrama de flujo del código en Arduino para los sensores.

Estas funciones estabilizan las lecturas del sensor y compensan las fluctuaciones breves del ruido eléctrico. Cada función realiza la medición analógica cuatro veces y promedia los resultados para una medición más fiable.

Posteriormente, mediante la función `map()`, se ajustan los rangos de valores crudos del ADC hacia las unidades físicas o porcentajes correspondientes, adecuando la interpretación de la señal para su uso en los nodos de ROS 2.

El microcontrolador envía estos datos procesados por el puerto serial para ser escuchados y publicados en ROS 2 mediante el nodo personalizado..

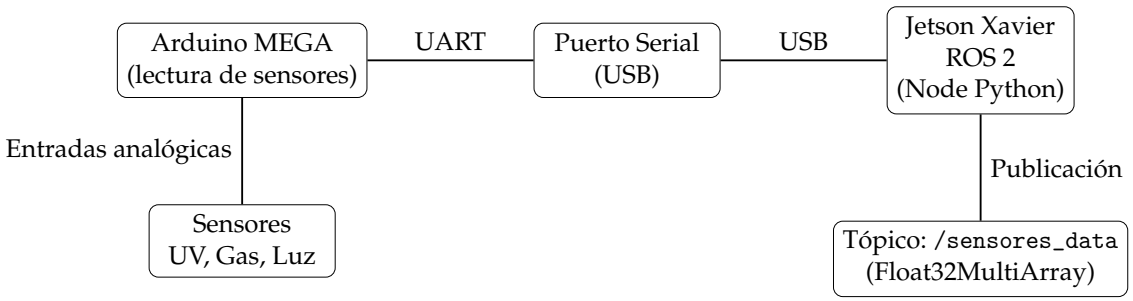


Figura 3.4: Conexión entre sensores, Arduino y Jetson Xavier con ROS 2

Con esta configuración básica, el Arduino funciona como un simple transmisor de datos serial compatible con el nodo ROS 2 desarrollado, asegurando una comunicación efectiva sin necesidad de librerías especiales de ROS 2 representados gráficamente en forma de onda en RViz.

3.4. Visualización de Datos en RViz

Una parte fundamental en el desarrollo e implementación del sistema es la visualización efectiva de los datos que provienen de los sensores conectados al Arduino y publicados en ROS 2. Para esto se utilizó **RViz**, la herramienta oficial de visualización 3D para ROS que permite observar en tiempo real los valores y estados del robot y sus sensores.

En este proyecto, RViz permitió la monitorización de los valores de temperatura y luz ambiental enviados desde Arduino mediante un nodo personalizado desarrollado en Python. Esto facilitó la validación de la correcta recopilación y transmisión de datos, permitiendo compararlos con los valores ambientales reales.

3.4.1. Procedimiento para abrir RViz y visualizar los datos

Para iniciar la visualización en RViz, se siguieron los siguientes pasos en el sistema Ubuntu donde corre ROS 2 Humble:

```
# 1. Abrir un terminal y activar el entorno ROS 2
source /opt/ros/humble/setup.bash

# 2. Ejecutar el nodo que publica los datos de sensores (ejemplo)
ros2 run sensor_reader sensor_reader_node

# 3. Abrir otro terminal para iniciar RViz
rviz2
```

Al abrir RViz, se debe configurar la visualización de los tópicos correspondientes publicados por el nodo. Para ello:

1. En la ventana de RViz, en la sección "Displays", hacer clic en el botón `Add`.
2. Seleccionar el tipo de visualización `Plot` o `Message` dependiendo del tipo de datos que se desean mostrar.
3. Seleccionar el tópico deseado (por ejemplo, `/temperatura` para temperatura o `/sensores_data` para el arreglo de sensores).
4. Ajustar las propiedades de la visualización como colores, rangos, escalas, etc.

3.4.2. Visualización de la Temperatura

El sensor de temperatura conectado al Arduino reporta valores en grados Celsius. Para validar su precisión, se comparó la lectura en RViz con la temperatura real de la ciudad de Cuenca, Ecuador, que en el momento de la prueba era aproximadamente 20 °C, según fuentes meteorológicas en línea como Google Weather. Todo esto está reflejado claramente en la figura 3.5.

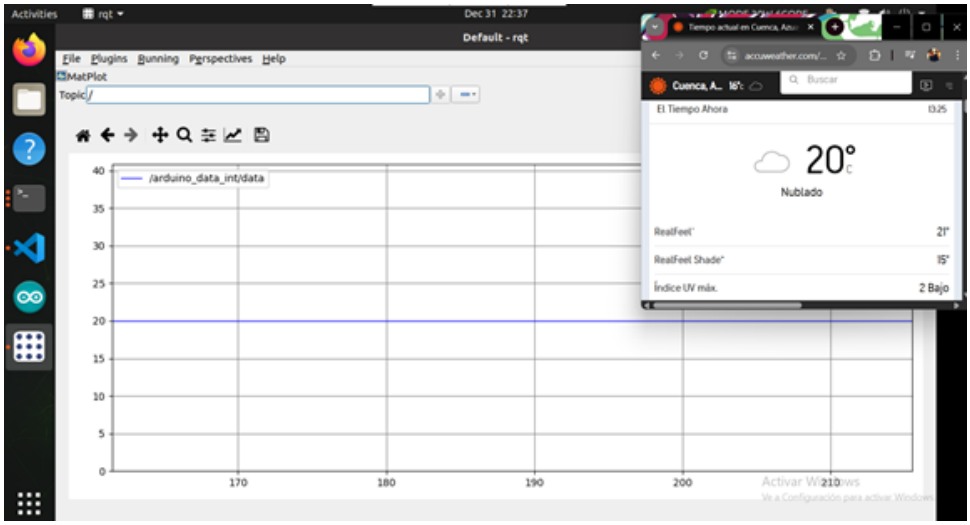


Figura 3.5: Visualización en RViz de la lectura de temperatura (20 °C) para Cuenca, Ecuador.

La lectura mostrada en RViz corresponde al valor enviado desde el Arduino y publicado en el tópico `/temperatura`, confirmando la correcta operación del sistema y la comunicación entre nodos.

3.4.3. Visualización de Luz Ambiental

El sensor de luz ambiental proporciona un valor analógico que varía en función de la iluminación recibida. Se realizaron pruebas con diferentes condiciones de luz para validar la respuesta del sensor:

- **Sensor tapado**: El valor se aproxima a 0, indicando ausencia de luz.
- **Iluminación con linterna de celular**: El valor se acerca a 1000, indicando máxima luminosidad.
- **Luz ambiente normal**: El valor se sitúa aproximadamente a 500.

Estas variaciones se observaron en RViz en tiempo real, evidenciando la capacidad del sistema para captar y mostrar cambios en las condiciones lumínicas del entorno, como se observa en la figura 3.6.

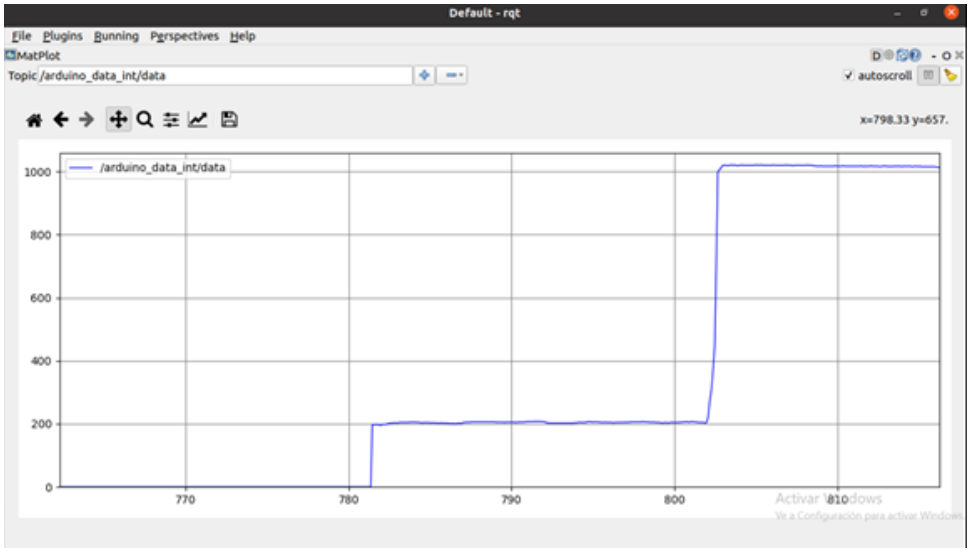


Figura 3.6: Visualización en RViz de la lectura del sensor de luz ambiental en diferentes condiciones.

El monitoreo en RViz fue crucial para comprobar que el nodo Python y la comunicación serial funcionaran adecuadamente, reflejando en el entorno ROS 2 los valores físicos de los sensores conectados al Arduino.

En esta etapa del proyecto se procedió a realizar la visualización detallada únicamente de dos sensores: el sensor de temperatura y el sensor de luz ambiental, debido a su relevancia para la validación inicial del sistema. Sin embargo, es importante destacar que la metodología de adquisición, publicación y visualización de datos implementada es completamente extensible a los otros dos sensores restantes (vapor y gas). La estructura del nodo en Python, la publicación en tópicos y la configuración en RViz permiten agregar sin dificultades adicionales los tópicos correspondientes a estos sensores, asegurando así una supervisión integral de todas las variables medidas por el sistema. Por lo tanto, la visualización de los sensores de vapor y gas seguiría un proceso análogo al aquí presentado para los sensores de temperatura y luz, garantizando la escalabilidad y modularidad del diseño

Capítulo 4

Conclusiones y Recomendaciones

4.1. Conclusiones

En el primer capítulo se logró establecer la comunicación entre el microcontrolador Arduino MEGA y la Jetson Xavier, utilizando la librería rosserial como herramienta principal para la integración inicial con ROS 2. Esta implementación permitió una primera conexión funcional entre ambos dispositivos. No obstante, durante el proceso se identificaron limitaciones significativas: entre ellas, la falta de soporte oficial para ROS 2, la necesidad de emplear el puente ros1_bridge y ciertos problemas de latencia y compatibilidad, especialmente en entornos distribuidos. Estos inconvenientes evidenciaron la necesidad de migrar hacia una solución más adecuada y nativa para el ecosistema ROS 2.

Más adelante en el segundo capítulo, se desarrollaron nodos personalizados en Python capaces de leer información desde el puerto serial de la Jetson Xavier, lo que permitió aprovechar al máximo las ventajas que ofrece ROS 2. La lectura de datos desde sensores como UV, luz ambiental, vapor y gas se gestionó mediante tópicos individuales, facilitando su visualización en tiempo real. Esta nueva arquitectura no solo incrementó la confiabilidad y el control sobre el flujo de datos, sino que también mejoró notablemente la escalabilidad del sistema, sentando una base sólida para su crecimiento futuro.

El tercer capítulo estuvo enfocado en la integración de herramientas complementarias como rqt_plot para la representación gráfica de los datos y RealVNC

Viewer para el acceso remoto a la Jetson Xavier. Estas incorporaciones permitieron validar el comportamiento del sistema sin necesidad de interactuar físicamente con el hardware, optimizando los procesos de monitoreo, depuración y supervisión. Gracias a estas herramientas, el entorno de trabajo se volvió mucho más flexible y eficiente, facilitando el desarrollo y la verificación del sistema en tiempo real.

La migración de ROS 1 a ROS 2 no solo fue exitosa, sino que también tuvo un impacto positivo en la arquitectura y funcionalidad general del rover. Lejos de afectar el rendimiento ni deshabilitar componentes, la transición permitió una mejor organización de los nodos, mayor estabilidad en la comunicación de datos y un uso más completo del hardware disponible. El uso de ROS 2 permitió la construcción de un sistema distribuido y escalable en el que los sensores conectados a Arduino podían leerse, publicarse y visualizarse de forma más eficiente y robusta. Esta experiencia demuestra que la adopción de ROS 2 no implica una pérdida de compatibilidad, sino que ofrece mejoras notables en rendimiento, flexibilidad y preparación para el futuro de sistemas robóticos complejos.

4.2. Recomendaciones

Con base en los resultados obtenidos y las lecciones aprendidas durante el desarrollo del proyecto, se recomienda considerar el uso de microcontroladores con soporte nativo para ROS 2 en futuras implementaciones de rovers. Las alternativas más adecuadas que se incluyen en la familia Arduino son: Arduino Nano 33 IoT, Arduino Nano 33 BLE, Arduino Portenta H7, Arduino MKR WiFi 1010. Por otro lado, se pueden usar otras alternativas diferentes a Arduino, como lo son: ESP32, la familia STM32 y las placas Teensy 4.x, ya que todas son compatibles con Micro-ROS, lo que elimina la necesidad de puentes intermedios y mejora significativamente la integración en sistemas distribuidos. Además, ofrecen mayores recursos de procesamiento, conectividad y flexibilidad.

También se recomienda ampliar el número y la variedad de sensores para mejorar la adquisición de información ambiental, optimizar los nodos desarrollados mediante la gestión de errores y la validación de datos, y automatizar el proceso

de arranque del sistema para facilitar la implementación en aplicaciones reales. Finalmente, una buena organización del código y una documentación clara y actualizada son esenciales para garantizar la escalabilidad a largo plazo y el mantenimiento eficiente del proyecto.

Referencias

[1] NASA, *Curiosity – NASA Mars Exploration*, Accessed: 2023-03-26, 2023. dirección: https://mars.nasa.gov/msl/home/.

[2] NASA, *Mission Overview - NASA Mars*, Accessed: 2025-03-26, 2020. dirección: https://mars.nasa.gov/mars2020/mission/overview/.

[3] D. Farfán y J. Gonzales, «Reingeniería y Construcción de un Robot Móvil Teleoperado del Grupo de Investigación en Interacción, Robótica y Automática para Navegación en Terrenos Nivelados y No Nivelados con Telemetría de Variables Atmosféricas y de Entorno,» Tesis de mtría., Universidad Politécnica Salesiana, Cuenca, Ecuador, 2023. dirección: http://dspace.ups.edu.ec/handle/123456789/25784.

[4] D. Farfán y J. Gonzales, «Reingeniería y construcción de un robot móvil teleoperado del Grupo de Investigación en Interacción, Robótica y Automática para navegación en terrenos nivelados y no nivelados con telemetría de variables atmosféricas y de entorno,» Universidad Politécnica Salesiana (UPS), Cuenca, Ecuador, 2023. dirección: https://www.ups.edu.ec/.

[5] A. Campos y J. Tomalá, «Robot Desbrozador Todoterreno Amortiguado,» Tesis de mtría., Universidad Politécnica Salesiana, Cuenca, Ecuador, 2024. dirección: http://dspace.ups.edu.ec/handle/123456789/27853.

[6] J. Bravo, «Robot Navegador Autónomo,» Tesis de mtría., Universidad Politécnica Salesiana, Cuenca, Ecuador, 2023. dirección: http://dspace.ups.edu.ec/handle/123456789/27853.

[7] F. Duan, W. Li e Y. Tan, «ROS Debugging,» en *Intelligent Robot*. Singapore: Springer, 2023. DOI: https://doi-org.ecups.idm.oclc.org/10.1007/978-981-19-8253-8_4.

[8] G. Stavrinos, «ROS2 For ROS1 Users,» en *Robot Operating System (ROS)* (Studies in Computational Intelligence), A. Koubaa, ed., Studies in Computational Intelligence. Cham: Springer, 2021, vol. 895. DOI: https://doi-org.ecups.idm.oclc.org/10.1007/978-3-030-45956-7_2.

[9] L. Dust y S. Mubeen, «Dynamic Priority Scheduling for Periodic Systems Using ROS 2,» en *Engineering of Computer-Based Systems. ECBS 2023*, ép. Lecture Notes in Computer Science, J. Kofroň, T. Margaria y C. Seceleanu, eds., vol. 14390, Springer, Cham, 2024. DOI: 10.1007/978-3-031-49252-5_20. dirección: https://doi.org/10.1007/978-3-031-49252-5_20.

[10] D. St-Onge y D. Herath, «The Robot Operating System (ROS1 & 2): Programming Paradigms and Deployment,» en *Foundations of Robotics*, D. Herath y D. St-Onge, eds., Singapore: Springer, 2022. DOI: 10.1007/978-981-19-1983-1_5. dirección: https://doi-org.ecups.idm.oclc.org/10.1007/978-981-19-1983-1_5.

[11] Y. Ye, Z. Nie, X. Liu et al., «ROS2 Real-time Performance Optimization and Evaluation,» *Chinese Journal of Mechanical Engineering*, vol. 36, n.º 1, pág. 144, 2023. DOI: 10.1186/s10033-023-00976-5. dirección: https://doi-org.ecups.idm.oclc.org/10.1186/s10033-023-00976-5.

[12] I. H. Savci, A. Yilmaz, S. Karaman, H. Ocakli y H. Temeltas, «Improving Navigation Stack of a ROS-Enabled Industrial Autonomous Mobile Robot (AMR) to be Incorporated in a Large-Scale Automotive Production,» *International Journal of Advanced Manufacturing Technology*, vol. 120, n.º 5-6, págs. 3647-3668, mayo de 2022, ISSN: 1433-3015. DOI: 10.1007/s00170-022-08883-0. dirección: https://bibliotecas.ups.edu.ec:3401/article/10.1007/s00170-022-08883-0.

[13] D. Farfán y J. Gonzales, «Reingeniería y construcción de un robot móvil teleoperado del Grupo de Investigación en Interacción, Robótica y Automática para navegación en terrenos nivelados y no nivelados con telemetría de variables atmosféricas y de entorno,» Universidad Politécnica Salesiana (UPS), Cuenca, Ecuador, 2023. dirección: https://www.ups.edu.ec/.

[14] O. Robotics, *ROS 2 Humble Hawksbill (Long Term Support)*, https://docs.ros.org/en/humble/index.html, Accedido el 16 de julio de 2025, 2022.

[15] O. Robotics, *ROS 2 Ecosystem Overview*, https://index.ros.org/doc/ros2/, Accedido el 16 de julio de 2025, 2023.

[16] N. I. ROS, *Getting Started with Isaac ROS Build Farm CDN*, `https://nvidia-isaac-ros.github.io/v/release-2.1/getting_started/isaac_ros_buildfarm_cdn.html`, Accedido el 16 de julio de 2025, 2024.

[17] NVIDIA, *Jetson Linux Developer Guide*, `https://docs.nvidia.com/jetson/`, Accedido el 16 de julio de 2025, 2024.

[18] D. Inc., *Docker Documentation*, `https://docs.docker.com/`, Accedido el 16 de julio de 2025, 2024.

[19] N. Developer, *Running Docker Containers on Jetson Devices*, `https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html`, Accedido el 16 de julio de 2025, 2024.

[20] NVIDIA, *CUDA Toolkit Documentation for Jetson*, `https://docs.nvidia.com/cuda/`, Accedido el 16 de julio de 2025, 2024.

[21] eProsima y R. 2. Community, *micro-ROS Documentation*, `https://micro-ros.github.io/`, Accedido: 2025-07-22, 2023.

[22] R. Community, *rosserial Wiki*, `http://wiki.ros.org/rosserial`, Accedido: 2025-07-22, 2023.