UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Sistemas Robóticos Cooperativos Implementación de un Motor de ALLIANCE

Informe de Proyecto de Grado presentado por

Maximiliano Videla y
José Lombardi

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisor

Dr. Gonzalo Tejera López

Montevideo, 24 de abril de 2026

Sistemas Robóticos Cooperativos Implementación de un Motor de ALLIANCE por Maximiliano Videla y José Lombardi tiene licencia CC Atribución - No Comercial - Sin Derivadas 4.0.

Agradecimientos

Agradecemos a nuestras familias y amigos por estar siempre apoyando, confiando en nuestro pequeño, pero no así sencillo proyecto. Agradecemos también a nuestro tutor, Dr. Gonzalo Tejera López, por confiar siempre en nosotros, exigiéndonos lo necesario para llegar en los tiempos acordados, por estar siempre pendiente y a la orden de nuestras dudas y consultas.

Resumen

La coordinación de Sistemas Multi-Robot en entornos no estructurados presenta desafíos significativos. Los sistemas deben ser capaces de responder eficientemente a eventos no planificados y a cambios en la composición del equipo.

Este proyecto de grado presenta el diseño, implementación y validación de un motor de control distribuido para la coordinación de equipos de robots, basado en los formalismos de la arquitectura Alliance. El desarrollo se centra en una solución reutilizable y modular construida sobre ROS 2 (Robot Operating System), utilizando el lenguaje Python.

La solución propuesta implementa un mecanismo de asignación de tareas donde cada robot, al detectar una nueva tarea, instancia dinámicamente un conjunto de nodos de comportamiento dedicados. Este conjunto incluye un nodo "Motivacional" que, basado en los conceptos de impaciencia y comunicación entre agentes, regula la selección de tareas, y un nodo "Ejecutor" que gestiona la activación de comportamientos de bajo nivel mediante un Árbitro. Esta arquitectura permite al sistema manejar misiones donde las tareas no se conocen de antemano.

La implementación se validó empíricamente mediante tres conjuntos de experimentos que evaluaron métricas clave de rendimiento: Tasa de Finalización de Misión (F_{ratio}), Tiempo Total de Misión (T_{mision}) y Tiempo Promedio de Resolución de Tarea ($\mu(T_{tarea})$). Las pruebas incluyeron: (1) escalabilidad horizontal, variando el número de agentes de 1 a 10; (2) escalabilidad vertical, incrementando la carga de 10 a 100 tareas; y (3) adaptabilidad dinámica, simulando fallos y adiciones de agentes durante la misión.

Los resultados muestran una robustez total ($F_{ratio} = 100\,\%$) incluso en el escenario de fallo. La escalabilidad horizontal mostró el beneficio de la paralelización hasta un punto de saturación (4 robots), donde el overhead de la creación de nodos y la coordinación de red superó las ganancias. La escalabilidad vertical exhibió un crecimiento lineal y predecible en T_{mision}, manteniendo un $\mu(T_{tarea})$ estable, validando la eficiencia de la arquitectura bajo carga.

Finalmente, las pruebas de adaptabilidad confirmaron la capacidad del sistema para degradarse controladamente ante fallos y absorber nuevos agentes dinámicamente, redistribuyendo la carga de trabajo con éxito. El trabajo valida la implementación como una solución eficaz y resiliente para la coordinación dinámica de robots en ROS 2.

Palabras clave: ALLIANCE, Alliance, Robótica Cooperativa, ROS2

Índice general

Capítulo 1

Introducción

La robótica ha experimentado una evolución notable desde sus inicios, transformándose en una disciplina que no solamente se ocupa de la automatización de tareas, sino que también aborda complejos problemas de cooperación y coordinación entre múltiples agentes. En la actualidad, el punto de vista de la robótica colectiva y cooperativa es, en muchos casos, indispensable, permitiendo a los robots trabajar en conjunto para lograr objetivos que serían inalcanzables para uno solo. Este enfoque ha dado lugar a arquitecturas de control sofisticadas, que permiten a los robots comunicarse y coordinarse de manera eficiente.

Una de las arquitecturas más destacadas en este contexto es Alliance (Parker, 1995), que se centra en la cooperación y la resiliencia en entornos dinámicos. Esta arquitectura está diseñada para garantizar que los sistemas multi-robot se mantengan operativos incluso en caso de fallos, aumentando así la robustez del sistema y la fiabilidad con la que completan las misiones. A medida que la robótica avanza, la integración de tecnologías como la inteligencia artificial y el aprendizaje automático promete ampliar aún más las capacidades de los sistemas colectivos, transformándolos en más adaptables y eficientes.

Este análisis se centrará en la arquitectura de control Alliance propuesta por Lynne E. Parker en 1995, partiendo de la evolución de la robótica, su estado actual y las perspectivas futuras. Se hará especial énfasis en la robótica colectiva y cooperativa, así como en las arquitecturas de control que faciliten la colaboración entre múltiples agentes. Se detallará el particular enfoque de la arquitectura Alliance, sus ventajas y características.

1.1. Contexto General de la Investigación

La importancia de los sistemas robóticos cooperativos radica en su capacidad para abordar tareas complejas que requieren la colaboración de múltiples agentes. Estos sistemas son esenciales en diversas aplicaciones, desde la exploración espacial, la búsqueda y rescate, hasta la logística y la agricultura de precisión. Al trabajar en conjunto, los robots pueden compartir información, distribuir cargas

de trabajo y adaptarse a cambios en el entorno, lo que aumenta su efectividad, o en otras palabras, mejora la eficiencia sin perder eficacia.

En el campo de la robótica, las tendencias actuales incluyen la integración de inteligencia artificial (AI por su sigla en inglés), el desarrollo de algoritmos de aprendizaje profundo, y la mejora de las capacidades de percepción y navegación en entornos no estructurados. Además, el avance en la comunicación entre robots y el uso de tecnologías como Internet de las Cosas (IoT por su sigla en inglés) están revolucionando la forma en que los sistemas robóticos cooperan y se coordinan.

Estas tendencias, si bien están fuera del alcance del presente proyecto, son parte del escenario actual que motiva y contextualiza el mismo. La integración de inteligencia artificial y algoritmos de aprendizaje profundo permitiría a los robots no solo realizar tareas individuales de manera más eficiente, sino también tomar decisiones colectivas más sofisticadas. Las mejoras en la percepción y navegación son cruciales para que los robots cooperativos puedan operar de forma segura y eficiente en entornos dinámicos y no estructurados. Una mejor percepción les permitiría detectar y evitar colisiones con mayor eficacia mientras ejecutan tareas conjuntas, o incluso compartir información sensorial con otros miembros del equipo para construir un mapa colaborativo del entorno. El avance en la comunicación entre robots y el uso de tecnologías como IoT, junto con middleware avanzado como DDS (Data Distribution Service), un estándar de comunicación orientado a datos que facilita la interoperabilidad y la comunicación distribuida sin necesidad de un servidor centralizado, el cual es utilizado por ROS 2, son esenciales para la coordinación distribuida robusta en sistemas multiagente. Una comunicación eficiente y fiable, incluso en condiciones no ideales, permite a los robots compartir información sobre su estado, las tareas completadas o los obstáculos encontrados, facilitando mecanismos de coordinación como los de Alliance, donde los robots ajustan sus motivaciones basándose en las acciones de otros. Esto es vital para la robustez del sistema frente a fallos individuales. De esta forma, este proyecto, al implementar un motor para Alliance construido sobre tecnologías como ROS 2 y su middleware DDS, sienta las bases para integrar este tipo de capacidades y mejoras en comportamientos futuros que se desarrollen sobre esta plataforma.

1.2. Problema que se aborda en el proyecto

A pesar de los avances en robótica cooperativa, existen limitaciones significativas en los sistemas de control actuales. Muchas de estas limitaciones incluyen la dificultad para gestionar la complejidad en las comunicaciones y la coordinación entre robots, especialmente en entornos dinámicos. Además, la dependencia de un único punto de control puede generar vulnerabilidades que afecten la resiliencia del sistema.

En cuanto a la implementación de la arquitectura Alliance, los desafíos son diversos. Uno de los principales es saber qué está haciendo otro robot del equipo. Como Alliance utiliza comunicación explícita, lograr que la comunicación entre

los robots sea suficientemente robusta y eficiente para soportar los mecanismos de coordinación es una tarea difícil. Esto se vuelve particularmente notorio en situaciones donde los recursos de red son limitados o inestables, ya que la posible pérdida o el retraso en la transmisión de mensajes cruciales (como los que informan sobre la actividad de otros robots) puede llevar a que los agentes operen con información desactualizada o incompleta, afectando su capacidad para tomar decisiones óptimas y coordinar sus acciones de manera efectiva. Para estos casos, la resiliencia de esta arquitectura permite a los robots redistribuir las tareas de forma que se garantice el éxito de la misión.

Otro desafío radica en el diseño de algoritmos que permitan la detección y gestión de fallos de manera efectiva, garantizando que el sistema mantenga su operatividad a pesar de la pérdida de uno o varios agentes. Además, la integración de esta arquitectura en sistemas existentes puede requerir adaptaciones importantes en la infraestructura y los protocolos de comunicación.

1.3. Objetivos

El objetivo principal del proyecto es implementar la arquitectura Alliance de manera que se genere un paquete accesible y reutilizable para su integración en diversos trabajos y proyectos futuros. Este paquete facilitará a los desarrolladores y a la comunidad de investigación la adopción de Alliance en aplicaciones de robótica cooperativa, promoviendo su uso en entornos prácticos.

Para validar la efectividad de la implementación de Alliance, se ha elegido como caso de estudio la recolección de manzanas en una granja. Este escenario presenta un conjunto de interesantes desafíos, que ilustran la necesidad de un sistema robótico cooperativo eficiente. Los principales desafíos planteados son:

1. Entorno Dinámico: Las granjas suelen ser entornos cambiantes, con variaciones en la disposición de los árboles, condiciones meteorológicas, la disponibilidad de frutas, y la circulación de personas y otros robots. Esto exige que los robots sean capaces de adaptarse rápidamente a las condiciones del entorno.

2. Colaboración entre Robots: La recolección de manzanas puede ser optimizada mediante la cooperación entre múltiples robots. Cada robot puede encargarse de recolectar en puntos diferentes simultáneamente, comunicándose entre sí para maximizar la eficiencia y evitar redundancias.

3. Manejo de Fallos: En un entorno agrícola, los robots pueden enfrentar obstáculos inesperados, como daños mecánicos o condiciones adversas. La implementación de Alliance permitirá a los robots detectar fallos y ajustar sus comportamientos en consecuencia, asegurando que la operación de recolección no se interrumpa o que se vea afectada lo menos posible.

4. Deficiencias en infraestructura de comunicación. En los entornos agrícolas, las redes de comunicación pueden ser débiles o inestables. Esta deficiencia

en infraestructura de comunicación presenta un desafío para la coordinación robótica. La arquitectura Alliance, al ser distribuída y no depender de un servidor central, está diseñada para operar con la comunicación inter-robot. Es decir que cada robot debe ser capaz de tomar decisiones de manera autónoma, basándose en información local y en la comunicación esporádica con sus pares. Esta capacidad de resiliencia frente a la pérdida de comunicación asegura que, incluso si un robot pierde la conexión con los demás o si la red se interrumpe, pueda seguir operando de manera efectiva. Esto contrasta con los sistemas centralizados, que colapsarían ante una falla en la red.

1.4. Desarrollo del proyecto

En esta sección se describe la información esencial sobre el camino que siguió el proyecto en términos de tecnología y enfoque, y se presenta desde una perspectiva de decisiones de ingeniería y justificaciones técnicas.

Inicialmente, la concepción del proyecto se planteó utilizando un stack tecnológico basado en Python 2.7 (Python Software Foundation, 2025), la plataforma de desarrollo robótico ROS Kinetic Kame (Open Source Robotics Foundation, Inc., 2025) y el simulador CoppeliaSim Edu (conocido anteriormente como V-REP) (Coppelia Robotics AG, 2025), ejecutándose sobre Ubuntu 16.04 (esta era la versión recomendada para esa distribución de ROS en ese momento). Esta configuración fue seleccionada considerando las versiones estables y las herramientas disponibles en ese momento para el desarrollo de sistemas robóticos. Se implementó una estrategia inicial de control del movimiento para los robots, organizada en una capa de comportamientos mediante una máquina de estados con tareas concurrentes.

Durante esta fase inicial, la rápida evolución tanto de ROS como del simulador generó desafíos significativos. La compatibilidad entre versiones se convirtió en un obstáculo, dificultando la búsqueda de soporte y soluciones en la documentación oficial y foros, que consistentemente recomendaban la actualización a versiones más recientes. Si bien se alcanzó un estado de avance considerable, la identificación y corrección de fallas complejas (por ejemplo, problemas en el control del movimiento que llevaban a colisiones o desorientación) se volvieron arduas debido a la interdependencia entre la estrategia de control, la implementación de la arquitectura y el sistema de comunicación.

Ante este panorama y la aparición de ROS 2 (Stanford Artificial Intelligence Laboratory et al., 2022), que prometía mejoras sustanciales en arquitecturas distribuidas (eliminando la dependencia de un nodo maestro, un punto de fallo crítico en ROS 1, que afectaba la robustez del entorno distribuido), se tomó una decisión estratégica en acuerdo con el tutor: migrar el proyecto a ROS 2 para aprovechar sus ventajas en robustez y escalabilidad en entornos distribuidos. La nueva pila tecnológica adoptada incluyó Python 3.12, ROS 2 Humble Hawksbill (Open Source Robotics Foundation, 2025b) y CoppeliaSim Edu V4.8.0 rev0, compatible con Ubuntu 22.04.

Esta migración representó una oportunidad para reestructurar el enfoque del proyecto. Se priorizó el desarrollo de un "motor" de Alliance funcional y reusable como paquete independiente en ROS 2, con funcionalidades básicas de la arquitectura y comportamientos genéricos. Este enfoque modular permitiría asegurar el correcto funcionamiento de la implementación de la arquitectura Alliance de forma desacoplada de las particularidades de los comportamientos específicos. Posteriormente, se implementaría el caso de estudio de la granja de manzanas como una extensión de este motor, desarrollando los comportamientos concretos requeridos para la misión. Este cambio no alteró el alcance final del proyecto, sino que redefinió la estrategia de desarrollo para mejorar la calidad y reutilización del resultado principal: el motor de Alliance.

1.5. Organización del documento

Este documento se encuentra organizado en diferentes capítulos, los cuales están estructurados de la siguiente forma:

- Capítulo 2: Marco teórico, se presenta un recorrido estructurado a través de los conceptos clave en robótica, desde la definición de "robot" hasta la exploración de arquitecturas de control avanzadas.
- Capítulo 3: Antecedentes en temáticas relacionadas al proyecto.
- Capítulo 4: ALLIANCE. Se presenta la arquitectura detallando los diferentes conceptos que plantea y sus fórmulas asociadas.
- Capítulo 5: Implementación del motor de Alliance. Se detalla la estructura de la solución propuesta. Se describe en profundidad el trabajo realizado, las decisiones de diseño tomadas y detalles de implementación relevantes de la solución final.
- Capítulo 6: Implementación del caso de estudio, Granja de manzanas. Se detalla la implementación de la solución al caso de estudio planteado, reutilizando y extendiendo la funcionalidad del paquete descrito en el capítulo anterior. Se describen en profundidad los comportamientos utilizados y su esquema de control dentro de la arquitectura.
- Capítulo 7: Experimentación, aquí se proponen diferentes métricas e indicadores para validar la eficacia y eficiencia de la solución propuesta.
- Capítulo 8: Conclusiones y trabajos a futuro.
- Apéndice A: Se describe ROS 2 y se contrasta con su antecesor ROS.
- Apéndice B: Se describe CoppeliaSim Edu como entorno virtual de trabajo para simular la operación de los robots en su entorno.
- Apéndice C: Manual de usuario, se da un paso a paso para que cualquier usuario pueda descargar y ejecutar el código del proyecto.

Capítulo 2

Marco teórico

Este capítulo expone una base conceptual del conocimiento necesario para comprender el problema planteado, la motivación de la implementación del motor de Alliance, y el desarrollo final del proyecto, proporcionando un contexto sólido para el resto del documento.

Este marco teórico no pretende dar una explicación exhaustiva de cada tema, sino introducir al lector en la problemática, tras una breve reseña. Para tener un conocimiento más profundo de cada tema, el lector podrá dirigirse a las referencias.

Se dará una breve explicación de lo que son y qué tipos de arquitecturas de control son las más utilizadas, un concepto general de Alliance y, finalmente, un repunte de qué beneficios tienen ROS y ROS 2 como entornos de desarrollo para robótica.

2.1. El origen del término robot

La palabra "robot" proviene del checo "robota", que significa "trabajo forzado" o "servidumbre". Fue popularizada por el escritor checo Karel Čapek en su obra de teatro R.U.R. (Robots Universales Rossum), estrenada en 1920. En esta obra, Čapek imaginó un mundo donde se fabricaban seres artificiales para realizar trabajos pesados, a los que llamó "robots". R.U.R. inspiró a numerosos escritores de ciencia ficción posteriores, como Isaac Asimov, quien acuñó las famosas Tres Leyes de la Robótica:

- Primera Ley. Un robot no hará daño a un ser humano o, por inacción, permitir que un ser humano sufra daño. Esta es la ley más fundamental y establece que la seguridad humana siempre debe ser la prioridad máxima para un robot.

- Segunda Ley. Un robot debe obedecer las órdenes dadas por los seres humanos, excepto cuando estas órdenes entren en conflicto con la Primera

Ley. Esta ley establece una jerarquía de órdenes, donde la seguridad humana siempre prevalece.

- Tercera Ley. Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la Primera o la Segunda Ley. Esta ley garantiza la autoconservación del robot, pero solo en la medida en que no ponga en peligro a los humanos.

Las Tres Leyes de la Robótica de Isaac Asimov, aunque concebidas en el ámbito de la ciencia ficción, siguen siendo relevantes en el debate actual sobre el desarrollo de la Inteligencia Artificial y la Robótica. Estas leyes nos invitan a reflexionar sobre los principios éticos que deben guiar la creación de sistemas autónomos capaces de tomar decisiones complejas. Al imaginar un futuro donde los robots coexisten con los humanos, Asimov anticipó preocupaciones actuales como la seguridad, la privacidad y la responsabilidad. La creciente sofisticación de la IA y la Robótica plantea desafíos sin precedentes, y es fundamental establecer marcos regulatorios que garanticen que la tecnología se utilice para el beneficio de la humanidad.

Dada la amplia difusión del término "robot" en la ciencia ficción, en el imaginario colectivo se ha creado una idea muy ambiciosa de tal máquina en referencia a sus capacidades; sin embargo, el camino académico y de ingeniería hacia la construcción de artefactos con funcionalidades que se acerquen a esas concepciones y que sean capaces de operar en el mundo real ha sido un proceso largo y muy arduo, que continúa perfeccionándose día a día. A lo largo de la historia, la humanidad ha concebido y construido diversos autómatas, evolucionando desde los mecanismos más simples hasta los complejos sistemas robóticos actuales. Lograr que estos sistemas no solo ejecuten movimientos preprogramados, sino que perciban su entorno, tomen decisiones autónomas y realicen tareas complejas, especialmente en interacción con otros agentes, requiere de una organización interna sofisticada.

La definición de esta organización interna, es decir, cómo se estructuran los componentes de percepción, procesamiento de información, toma de decisiones y acción de un robot, es lo que se conoce como **Arquitectura de control robótico**. Estas arquitecturas son fundamentales para determinar las capacidades de un robot, su reactividad, su capacidad de planificación y, crucialmente para este proyecto, su habilidad para coordinarse y cooperar con otros robots en sistemas multiagente. El diseño adecuado de la arquitectura de control es, por lo tanto, un pilar esencial en la construcción de sistemas robóticos capaces de abordar los desafíos del mundo real.

En las siguientes secciones de este capítulo, se explorarán los diferentes enfoques y paradigmas existentes en las arquitecturas de control robótico, desde las más reactivas hasta las deliberativas e híbridas. Nos centraremos particularmente en aquellas relevantes para sistemas robóticos multiagente, lo que nos permitirá introducir y contextualizar adecuadamente la arquitectura Alliance como arquitectura de control para coordinar agentes robóticos racionales (aquellos que maximizan su medida de rendimiento basándose en las evidencias aporta-

das y el conocimiento almacenado), que es el foco principal de la implementación desarrollada en el presente proyecto.

2.2. Arquitecturas de Control

Una Arquitectura de Control Robótico define como se organizan y comunican todos los componentes (de hardware y software) del robot, desde los sensores hasta los actuadores, pasando por la unidad de control (Matarić, 2007, Cap. 11).

Al diseñar un sistema robótico, es necesario prestar especial atención a la arquitectura de control que se va a utilizar, ya sea en un sistema robótico individual o en un sistema robótico cooperativo. En ambos casos, las arquitecturas de control son las encargadas de ordenar la ejecución de los comportamientos que el sistema robótico llevará a cabo (Arkin, 1998).

2.2.1. Características

Algunas de las características para tener en cuenta al momento de evaluar una arquitectura son: modularidad, lugar de aplicación, portabilidad, robustez.

El paradigma SPA(Siegwart, Nourbakhsh, y Scaramuzza, 2011) define 3 primitivas: Sensar (SENSE), función que tiene como entrada los datos de los sensores y como salida la información sensada; Planificar (PLAN), función que tiene como entrada la información (sensorial o cognitiva), y como salida las directivas para el robot; Actuar (ACT), función que tiene como entradas la información sensada y/o las directivas, y como salidas los comandos para los actuadores.

Ya sea por la relación entre las primitivas Sensar, Planificar, Actuar, o por la manera en que los datos sensoriales son procesados y distribuidos en el sistema, se pueden clasificar las arquitecturas de control en: arquitecturas jerárquicas/deliberativas (planificación centralizada, lentas pero adecuadas para tareas complejas), arquitecturas reactivas (respuesta rápida a estímulos, pero limitadas en tareas complejas), arquitecturas híbridas (combinan elementos de las anteriores), arquitecturas basadas en comportamientos (descomposición en comportamientos independientes).

2.2.2. Sistemas robóticos mono agente

En el contexto de sistemas de un solo agente, adquieren especial relevancia las arquitecturas basadas en los paradigmas deliberativo, reactivo e híbrido.

Enfoque deliberativo, basado en planes

La estrategia de este enfoque sigue el paradigma jerárquico Sense-Plan-Act (SPA), sensar su entorno, pensar y luego actuar. Utiliza un modelo del mundo. Se analiza la información obtenida por los sensores, el conocimiento adquirido y se crea un plan que luego se ejecuta. Arquitecturas de este tipo son, por ejemplo:

- Procedural Reasoning System (PRS) (Georgeff, Lansky, y Schoppers, 1987). Utiliza un modelo interno del mundo y un conjunto de planes para tomar decisiones. Es adecuado para tareas complejas que requieren planificación y razonamiento.

- Planificación clásica (simbólica). En este enfoque, el comportamiento se genera mediante un planificador que, a partir de un modelo lógico del mundo y un objetivo formalizado, realiza búsqueda en el espacio de estados para encontrar secuencias de operadores (planes) que transformen el estado inicial en el objetivo. Las acciones se representan mediante operadores con precondiciones y efectos, y la selección de acciones surge del proceso de búsqueda. Shakey the Robot (Nilsson, 1984) es el ejemplo paradigmático, utilizando el planificador STRIPS (Stanford Research Institute Problem Solver), que mediante búsqueda heurística en espacio de estados genera planes de ejecución motriz a partir de representaciones lógicas del entorno y objetivos definidos.

Enfoque reactivo

Este enfoque se caracteriza por no utilizar ninguna representación interna del mundo que lo rodea, siendo su accionar no pensar y actuar, es decir, sin planificación previa. Permite que el robot reaccione rápidamente a estímulos del entorno, potencialmente peligrosos. Algunas arquitecturas reactivas relevantes son:

- Subsumption (Brooks, 1986). La arquitectura Subsumption es un enfoque de diseño para sistemas de control de robots autónomos, concebido por Rodney Brooks en 1986. Esta arquitectura se basa en un modelo reactivo que prioriza la interacción directa del robot con su entorno en lugar de depender de representaciones internas complejas o de una planificación centralizada. A diferencia de las arquitecturas tradicionales deliberativas, subsumption enfatiza la capacidad del robot de responder rápidamente a estímulos del entorno mediante comportamientos simples y modulares. El núcleo de la arquitectura consiste en una colección de módulos de comportamiento independientes, cada uno diseñado para llevar a cabo una tarea específica, como evitar obstáculos, seguir una pared o buscar un objetivo. Estos módulos están organizados en una jerarquía de capas, donde las capas inferiores gestionan comportamientos básicos y las superiores implementan acciones más complejas. Cada capa puede subsumir o inhibir las salidas de las capas inferiores si considera que sus propias acciones son más relevantes para la situación actual, de ahí el nombre de la arquitectura.

- Esquemas de control motor (motor-control schemas) (Arkin, 1989). Se centra en el control directo de los actuadores del robot, utilizando esquemas o patrones de movimiento predefinidos. Es eficaz para tareas que requieren respuestas rápidas y precisas.

- Campos de potencial (Arkin, 1998) (R. Arkin). Este método crea campos de fuerza virtuales alrededor de los objetos en el entorno. El robot se mueve siguiendo el gradiente de estos campos, evitando obstáculos y dirigiéndose hacia objetivos. Se utiliza frecuentemente para la navegación evitando obstáculos.

Enfoque híbrido (Reactivo/Deliberativo)

Pensar y actuar independientemente, potencialmente en paralelo. La intención principal es tomar lo mejor de los dos mundos, siendo necesaria una capa extra para poder resolver la comunicación y conflictos entre las capas reactiva y deliberativa; por este motivo, estas arquitecturas son conocidas como arquitecturas de tres capas (ver Figura 2.1).

Figura 2.1: Arquitectura de 3 capas

Las siguientes son algunas arquitecturas híbridas:

- AuRA (Arkin, Riseman, y Hanson, 1987). Propuesta por Arkin, Riseman y Hanson, la cual combina una base reactiva de comportamientos simples con un planificador deliberativo que coordina la misión del robot; la capa deliberativa influye en la selección de comportamientos, pero la ejecución inmediata recae en la parte reactiva.

- Atlantis (Gat, 1991). Organiza explícitamente estas funciones en una arquitectura de tres capas: un *controlador* estrictamente reactivo para acciones en tiempo real, un *secuenciador* intermedio que traduce metas en rutinas, y un *deliberador* que genera planes de alto nivel; esta separación clara entre niveles reduce la carga computacional del deliberador sin perder reactividad.

- Planner-Reactor (Lyons y Hendriks, 1992). Propuesta por Lyons y Hendriks, se centra en la interacción dinámica entre un planificador de alto

nivel y un sistema reactivo en paralelo: el planificador ajusta continuamente sus planes en función de la retroalimentación del reactor, lo que permite enfrentar entornos altamente dinámicos donde los planes deben adaptarse sobre la marcha.

2.2.3. Sistemas robóticos multiagente

Los sistemas robóticos multiagente se caracterizan por llevar a cabo tareas conjuntas o lograr objetivos comunes mediante la coordinación de múltiples robots. La arquitectura de control define cómo se organiza esta coordinación y cómo se distribuyen las decisiones dentro del sistema (Dudek, Jenkin, Milios, y Wilkes, 1996; Cao, Fukunaga, y Kahng, 1997).

La coordinación en sistemas robóticos puede lograrse mediante comunicación explícita, donde los agentes intercambian mensajes, o mediante coordinación implícita, en la que los agentes infieren el comportamiento de otros a partir de observaciones del entorno. Asimismo, puede emplearse comunicación indirecta a través del entorno (stigmergy) o mecanismos de negociación para la asignación de tareas (Parker, 1998; Gerkey y Matarić, 2002).

Dependiendo de qué agentes realicen la toma de decisiones, los modelos de coordinación se ubican en un espectro que va desde arquitecturas centralizadas hasta completamente distribuidas (Dudek y cols., 1996).

En una coordinación centralizada, una unidad central toma las decisiones y distribuye las tareas, supervisando el progreso del sistema. Este enfoque facilita la planificación global y la optimización de tareas, dado que toda la información está disponible en un único punto. Sin embargo, presenta limitaciones en términos de escalabilidad y robustez, debido a la existencia de cuellos de botella y un único punto de fallo (Cao y cols., 1997; Shoham y Leyton-Brown, 2009).

En contraste, en un modelo distribuido, cada robot toma decisiones de manera autónoma basándose en información local y en la interacción con otros agentes. Este enfoque ofrece alta tolerancia a fallos y buena escalabilidad, ya que los robots pueden incorporarse o retirarse sin afectar significativamente al sistema. No obstante, puede resultar difícil alcanzar una coordinación global óptima y pueden surgir conflictos o redundancias en la asignación de tareas (Gerkey y Matarić, 2002; Olfati-Saber, Fax, y Murray, 2007).

Finalmente, las arquitecturas jerárquicas combinan elementos de los enfoques centralizados y distribuidos. En estos sistemas, algunos agentes asumen roles de liderazgo —potencialmente dinámicos— encargados de la supervisión y asignación de tareas, mientras que otros ejecutan dichas tareas. Este enfoque busca equilibrar coordinación global y autonomía local, aunque introduce dependencias en los nodos de nivel superior, que pueden convertirse en puntos críticos ante fallos y requerir mecanismos de reconfiguración (Parker, 1998; Cao y cols., 1997).

En la Tabla 2.1 se elaboró una comparación entre los tres modelos de coordinación de arquitecturas.

Característica	Centralizada	Distribuida	Jerárquica
Tolerancia a fallos	Baja	Alta	Moderada
Coordinación	Alta	Baja/Moderada	Moderada
Escalabilidad	Baja/Moderada	Alta	Moderada
Robustez	Baja	Alta	Moderada
Eficiencia	Alta (óptimo global)	Moderada/Alta	Equilibrada

Tabla 2.1: Comparación conceptual de arquitecturas de control en sistemas multi-robot [adaptado de (Dudek y cols., 1996; Gerkey y Matarić, 2002; Olfati-Saber y cols., 2007)].

2.2.4. Arquitectura basada en comportamientos

Este tipo de arquitecturas descomponen el control del robot en módulos independientes llamados comportamientos, cada uno responsable de una tarea específica (por ejemplo, evitar obstáculos, navegar a un sitio determinado). La coordinación entre estos comportamientos determina la acción final del robot. Este enfoque se inspira en la conducta y el comportamiento animal. Se define comportamiento animal como la asignación de una entrada sensorial a un patrón de acciones motoras que son usadas para alcanzar una tarea. Una de las características clave que el comportamiento animal aporta a la robótica es que la construcción de comportamientos complejos se desarrolle a partir de comportamientos simples e independientes, que acoplan fuertemente sensado con actuación. Un sistema basado en comportamientos no mantiene un modelo del mundo, pero puede contener máquinas de estados finitos, que se encargan de conectar directamente sensores y actuadores. Este enfoque representa una extensión del paradigma reactivo, permitiendo la agrupación de comportamientos simples y orientados a tareas específicas para formar comportamientos emergentes más complejos. La descomposición orientada a tareas es una característica fundamental de esta aproximación. Esta modularidad facilita la creación de sistemas robóticos escalables y adaptables a entornos dinámicos. Los mecanismos más utilizados para lograr la coordinación de comportamientos son: subsunción (se define una jerarquía estricta entre capas de comportamientos donde los de nivel superior subsumen o inhiben a los de capas inferiores), arbitraje (selecciona el comportamiento ganador), fusión (combina las salidas de varios comportamientos).

Una arquitectura representativa del tipo multiagente y basada en comportamientos es “Alliance”.

Características principales de la arquitectura seleccionada para implementar en el presente proyecto

Alliance es una arquitectura de control multiagente basada en comportamientos que se centra en la coordinación distribuida a través de la motivación de los comportamientos. Cada conjunto de comportamientos tiene una moti-

vación que representa su interés en activarse, y un mecanismo de supresión selecciona el conjunto de comportamientos con mayor motivación. Alliance puede incluir aprendizaje para adaptar las motivaciones. Su enfoque la hace robusta en entornos dinámicos y con comunicación limitada. Se presentará en detalle la arquitectura Alliance en el Capítulo 4.

Capítulo 3

Antecedentes

Se presentan como antecedentes de temas relacionados algunos trabajos realizados. El primero es la tesis de maestría del Dr. Gonzalo Tejera López, titulada "Contribución al diseño de sistemas multi robots utilizando ALLIANCE" (Tejera, 2004).

Luego la tesis de doctorado: "Multi-robot exploration under non-ideal communication conditions" del Dr. Facundo Benavides (Benavides, 2019), donde se evalúan dos enfoques distintos y complementarios de exploración de escenarios no conocidos bajo condiciones de comunicación no ideales.

Otro antecedente estudiado es la tesis de grado realizada por el Ing. Mauro Mottini, "ALLIANCE sobre Torocó" (Mottini, 2021).

A continuación se presenta un pequeño resumen de las tesis incluyendo las motivaciones, los objetivos y las conclusiones a las cuales se llegaron.

3.1. Contribución al diseño de sistemas multi-robots utilizando ALLIANCE

La tesis de maestría de Gonzalo Tejera López, titulada "Contribución al diseño de sistemas multi robots utilizando ALLIANCE", presentada en 2004 en la Facultad de Ingeniería de la Universidad de la República (Uruguay), se centra en la mejora de arquitecturas de control para sistemas multi-robot, con énfasis en la robustez y la cooperación en entornos dinámicos.

El trabajo parte del estudio detallado de la arquitectura Alliance, diseñada para asignar tareas de manera robusta a robots heterogéneos frente a cambios inesperados en el entorno y modificaciones en el equipo de robots. Posteriormente, analiza su sucesora, L-Alliance, que mejora la selección cooperativa de acciones y reduce la necesidad de ajuste de parámetros por parte del diseñador. La tesis propone modificaciones al modelo formal de estas arquitecturas y nuevas heurísticas para mejorar la asignación de tareas, buscando un mejor rendimiento del equipo.

Para validar las propuestas, se implementaron simulaciones utilizando el simulador de robots Khepera YAKS, enfocándose en una tarea de recolección de objetos mediante máquinas de estado y comportamientos reactivos. Además, se definió un núcleo denominado N-Alliance, sobre el cual se implementaron Alliance, L-Alliance y las nuevas propuestas, brindando mayor flexibilidad y conservando las virtudes de la arquitectura original.

Los resultados obtenidos muestran que las modificaciones y heurísticas propuestas mejoran la asignación de tareas y el rendimiento del equipo de robots en comparación con las versiones anteriores de la arquitectura. Este trabajo contribuye al diseño de sistemas multi-robot más eficientes y adaptables, destacando la importancia de arquitecturas de control robustas y cooperativas en entornos dinámicos.

3.2. Multi-robot exploration under non-ideal communication conditions

En el contexto de la robótica autónoma, la exploración multi-robot en entornos desconocidos y con restricciones de comunicación es un problema fundamental, especialmente relevante en aplicaciones como la exploración planetaria, la vigilancia, el rescate, la agricultura y la limpieza de áreas peligrosas. Este problema se complica aún más cuando múltiples robots deben coordinarse sin depender de una conexión constante y estable, debido a las limitaciones de comunicación inalámbrica en entornos reales. Estas restricciones requieren que las estrategias de exploración tomen en cuenta no solo la cobertura completa del área, sino también la conexión intermitente y la necesidad de evitar el aislamiento de robots, lo cual puede causar pérdida de información y afectar la coordinación del equipo.

La tesis "Multi-robot exploration under non-ideal communication conditions" aborda el desafío de la exploración cooperativa en sistemas multi-robot cuando las condiciones de comunicación son subóptimas. A diferencia de las otras, ésta no se basa en comportamientos. Este trabajo se centra en desarrollar estrategias que permitan a un equipo de robots móviles explorar entornos desconocidos de manera eficiente, incluso cuando la conectividad entre ellos es limitada o intermitente.

Una de las principales contribuciones de la tesis es la propuesta de una estrategia multiobjetivo auto-adaptativa que equilibra el rendimiento de la exploración y el nivel de conectividad entre los robots. Esta estrategia permite que los robots tomen decisiones de manera autónoma y asincrónica, ajustando dinámicamente sus objetivos en función de las condiciones de comunicación y las características del entorno. Además, se introducen dos roles específicos para los robots: explorador y retransmisor. Los exploradores se encargan de investigar nuevas áreas, mientras que los retransmisores se posicionan estratégicamente para mantener la conectividad dentro del equipo. En la Figura 3.1 se puede ver la arquitectura de un nodo explorador. En la Figura 3.2 se muestra la arquitectura

de un nodo en formato retransmisor.

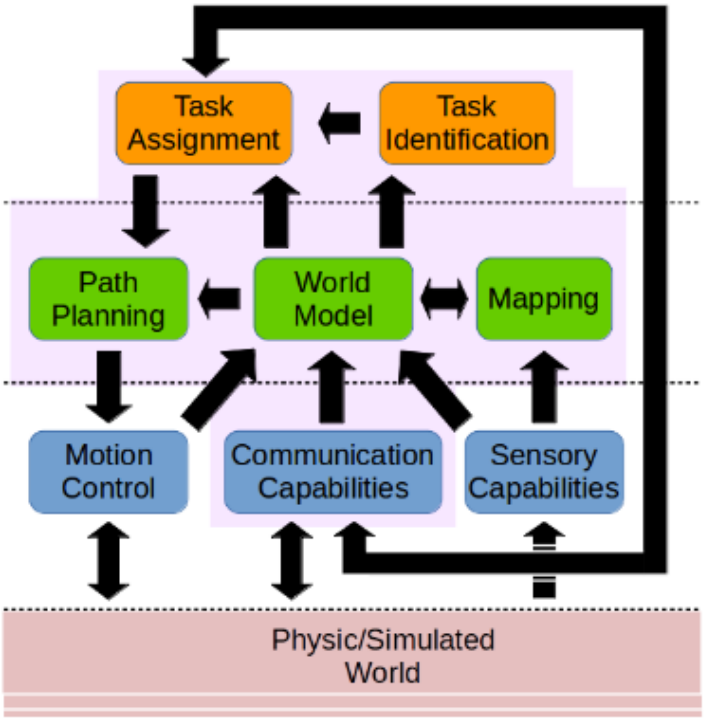

Figura 3.1: Arquitectura agente explorador, Fuente:(Benavides, 2019)

Para abordar el problema de posicionamiento de los retransmisores, la tesis presenta un enfoque novedoso que, basándose en un modelo de comunicación, permite una colocación eficiente de los retransmisores en tiempo polinómico. Este método evita la necesidad de resolver problemas complejos, por ejemplo, el problema de Steiner, Steiner Minimum Spanning Tree, lo que simplifica la implementación y mejora la eficiencia del sistema.

Los resultados obtenidos demuestran que las estrategias propuestas son capaces de reducir significativamente los períodos de desconexión entre los robots sin comprometer el tiempo total de exploración. Esto se traduce en sistemas más robustos y eficientes, capaces de operar en entornos donde las condiciones de comunicación son adversas o impredecibles. En resumen, la tesis ofrece soluciones innovadoras para la exploración multi-robot en escenarios con restricciones de comunicación, contribuyendo al avance de la robótica cooperativa y autónoma.

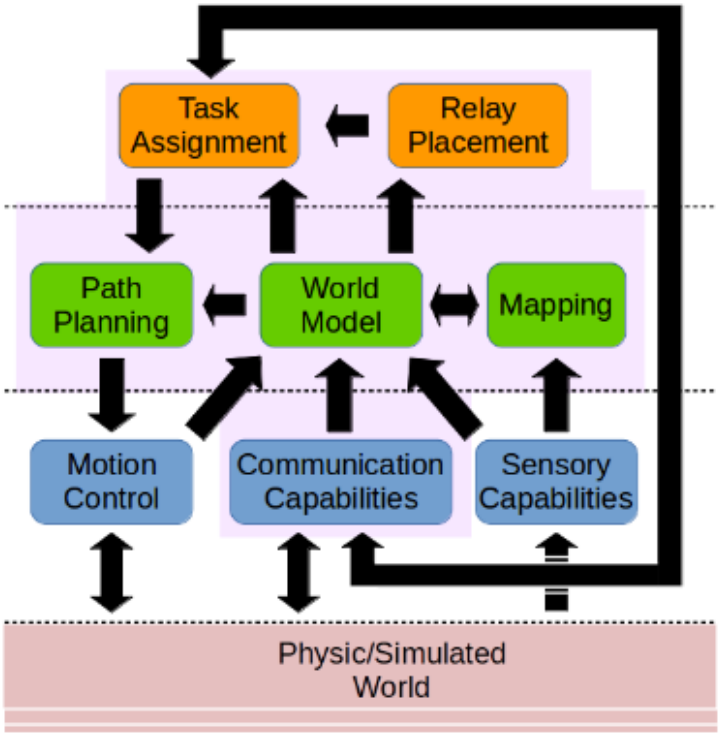

Figura 3.2: Arquitectura agente retrasmisor, Fuente:(Benavides, 2019)

3.3. ALLIANCE sobre Torocó

La motivación de la tesis de grado, ALLIANCE sobre Torocó : arquitecturas cooperativas basadas en comportamientos, surge del interés en mejorar la cooperación entre robots móviles en entornos dinámicos y cambiantes. En el ámbito de la robótica, la coordinación de múltiples robots representa un desafío clave, especialmente cuando se busca que operen de manera autónoma y sin dependencia de un control centralizado. Este trabajo explora enfoques basados en comportamientos para diseñar arquitecturas cooperativas, con el objetivo de lograr interacciones eficientes entre robots en escenarios donde la comunicación puede ser limitada o fallida.

Uno de los principales incentivos de su trabajo es la posibilidad de desarrollar sistemas multi-robot escalables y robustos, que puedan adaptarse a diversas tareas sin requerir una programación explícita para cada situación. La motivación también radica en la necesidad de soluciones flexibles y descentralizadas, que permitan a los robots responder de manera emergente a las condiciones del entorno. En este sentido se proponen métodos que puedan ser aplicados en diversas áreas, como la exploración autónoma, la logística y el rescate en situaciones

de desastre.

A lo largo del trabajo, se implementa y prueba la arquitectura cooperativa Alliance, sobre Torocó, evaluando su desempeño en entornos simulados y reales. A través de distintos experimentos, se observa cómo los robots pueden realizar tareas colectivas sin una planificación central, simplemente siguiendo principios de cooperación basados en comportamientos predefinidos. Esto demuestra que es posible lograr una coordinación eficiente sin necesidad de una comunicación constante entre los agentes.

En cuanto a las conclusiones, se destaca que las arquitecturas cooperativas basadas en comportamientos ofrecen ventajas significativas en términos de adaptabilidad y flexibilidad. El trabajo muestra que este enfoque es viable para la coordinación de múltiples robots en entornos dinámicos y que, en muchos casos, resulta más eficiente que métodos tradicionales basados en planificación centralizada. Además, se resalta que la simplicidad de los comportamientos individuales puede dar lugar a soluciones emergentes complejas y eficientes.

Otro punto importante es que, si bien la arquitectura propuesta demuestra ser robusta en diversas situaciones, existen desafíos pendientes en cuanto a la optimización del comportamiento colectivo. Se sugiere que futuras investigaciones podrían explorar la incorporación de técnicas de aprendizaje automático para mejorar la toma de decisiones y la adaptación de los robots a entornos aún más impredecibles.

Capítulo 4

ALLIANCE

Alliance se define como una arquitectura de control tolerante a fallos, para equipos de robots heterogéneos. Estos equipos llevan a cabo misiones compuestas por tareas independientes y poco acopladas. Cada uno de los robots posee su propio sistema de toma de decisiones, lo cual les permite en todo momento seleccionar de forma adecuada las acciones en función de los requisitos de la misión en curso, de acuerdo con las condiciones del entorno y de acuerdo al estado interno de cada robot.

Alliance es una arquitectura completamente distribuida basada en comportamientos que incorpora el uso de motivaciones para la selección de tareas, utilizando funciones como la impaciencia (impatience) y la aquiescencia (acquiescence), valores que cada robot va calculando para poder lograr seleccionar la acción más adecuada en cada momento.

Dado que los equipos de robots cooperativos trabajan en entornos dinámicos e impredecibles, Alliance permite a los miembros del grupo responder de forma robusta y confiable a cambios en el entorno y a cambios en la conformación del equipo, tanto por fallas mecánicas como también a nuevas habilidades aprendidas, nuevos miembros del equipo y también a bajas de los mismos.

La arquitectura planteada tiene una serie de supuestos que deben cumplirse para su correcto funcionamiento; estos son:

1. cada robot puede detectar el efecto de sus acciones con una probabilidad mayor a 0
2. un robot puede detectar la acción de los demás robots por algún medio que esté disponible, con una probabilidad mayor a 0, por ejemplo mediante broadcast
3. los robots utilizan un mismo lenguaje
4. los robots no mienten y tampoco tienen intenciones contrarias a la misión, al menos intencionalmente
5. el medio de comunicación no está garantizado durante toda la misión

6. los robots no poseen sensores y actuadores perfectos y pueden fallar en cualquier momento

7. si un robot falla es posible que no logre comunicar su falla al resto del equipo

8. no se dispone de un sistema centralizado que posea información del entorno.

Cada robot posee un conjunto de comportamientos de bajo y alto nivel, que van desde acciones primitivas (ej. evitar obstáculos) hasta tareas más complejas (exploración). Estas tareas están organizadas en "behavior sets" o conjuntos de comportamientos, donde solamente uno está activo a la vez, resolviendo conflictos entre tareas simultáneas y permitiendo a los robots cumplir distintos objetivos de manera eficiente.

4.1. Comportamientos motivacionales

Si un robot r_i tiene la capacidad de resolver una tarea t_j, entonces existe un conjunto de comportamientos a_{ij} implementado en el robot r_i que resuelve dicha tarea.

Por lo tanto, cuando un robot r_i activa un conjunto de comportamientos a_{ij} es porque está trabajando en la resolución de la tarea t_j y no en otra. Consecuentemente, si el robot r_i desea trabajar en resolver la tarea t_j, deberá activar su conjunto de comportamientos a_{ij} correspondiente.

Teniendo en cuenta que un robot puede tener varios conjuntos de comportamientos y estos pueden generar conflictos entre sí, se debe tener un mecanismo que permita seleccionar cuál de estos conjuntos estará activo en cada momento y que desactive a los demás conjuntos para evitar conflictos. Es aquí donde surgen los comportamientos motivacionales, encargados de activar el conjunto de comportamientos correspondiente.

Los comportamientos motivacionales reciben en todo momento un conjunto de entradas, donde se distingue entre información de sensores, datos de la comunicación entre robots, retroalimentación inhibitoria de otros comportamientos activos y también motivaciones internas llamadas impaciencia (impatience) y aquiescencia (acquiescence). El resultado de un comportamiento motivacional en un momento dado es el nivel de activación de su conjunto de comportamientos correspondiente, representado como un número con un valor mayor o igual a 0. A dicho valor lo llamaremos la motivación del conjunto de comportamientos. Lo representaremos como $m_{ij}(t)$ que se leerá, la motivación del robot i para la tarea j en el tiempo t. Cuando este valor supera cierto umbral θ definido para cada robot, activa el conjunto de comportamientos correspondiente e inhibe a los demás comportamientos motivacionales mediante la inhibición cruzada. La Figura 4.1 muestra la arquitectura descrita anteriormente, con los conjuntos de comportamientos alimentados por la información de los sensores, y los correspondientes comportamientos motivacionales con la posibilidad de suprimir la

salida del conjunto de comportamientos. También se muestra que los comportamientos motivacionales reciben información sensorial, datos de otros robots, e inhibición cruzada de otros comportamientos motivacionales activos. Se muestra también la jerarquía de las capas que contienen comportamientos de bajo nivel que son suprimidos por los conjuntos de comportamientos de más alto nivel.

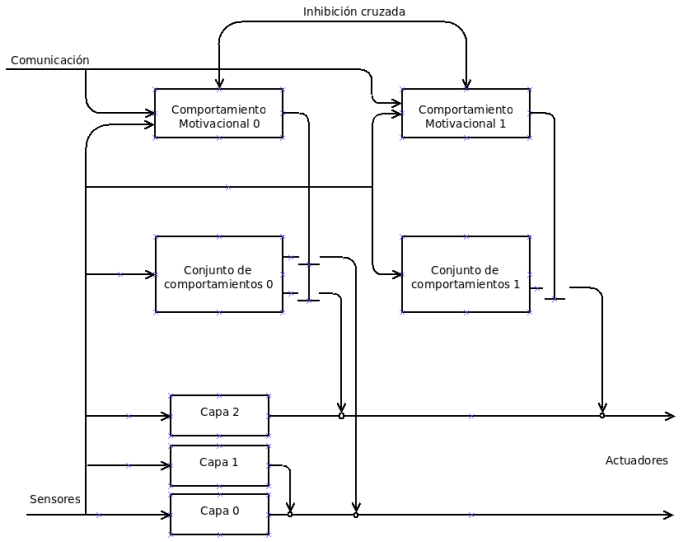

Figura 4.1: Diagrama Arquitectura ALLIANCE

4.2. Definición formal del problema

Lo primero que se define es R=$\{r_1, r_2, ..., r_n\}$, que representa un conjunto de n robots heterogéneos, los cuales conforman el equipo.

Un conjunto T=$\{t_1, t_2, ..., t_m\}$ de m diferentes tareas que conforman la misión a completar.

Cada robot r_i dispone de k conjuntos de comportamientos que se llamarán A$_i$=$\{a_{i1}, a_{i2}, ..., a_{ik}\}$

Dado que diferentes robots pueden tener diferentes formas de realizar la misma tarea, se necesita una forma de hacer referencia a la tarea en la que está trabajando un robot cuando activa un conjunto de comportamientos. Por lo tanto, se define el conjunto de n funciones $\{h_1(a_{1k}), h_2(a_{2k}), ..., h_n(a_{nk})\}$ donde $h_i(a_{ik})$ devuelve la tarea $t_j = h_i(a_{ik}), t_j \in T$, que es la tarea en la que trabajará el robot r_i al activar el conjunto de comportamientos a_{ik}. Se observa que como los n robots son potencialmente heterogéneos, se definen entonces n funciones h.

4.3. Nivel de activación

El parámetro θ define un umbral para la motivación calculada (nivel de activación), por encima del cual el conjunto de comportamientos correspondiente debe activarse y suprimir a los demás conjuntos de comportamientos del mismo robot. De esta forma, habrá un solo conjunto de comportamientos activo a la vez en cada robot.

4.4. Información sensorial

La información sensorial provee a los comportamientos motivacionales la información necesaria para determinar si un conjunto de comportamientos puede ser activado en un momento dado de la misión. Para esto se define la función *sensory_feedback*:

$$sensory\_feedback_{ij}\,(t) = \begin{cases} 1 & \text{si la retroalimentación sensorial del robot } r_i \\ & \text{en el tiempo } t \text{ indica que el conjunto de comportamientos } a_{ij} \text{ es aplicable} \\ 0 & \text{en otro caso} \end{cases}$$

4.5. Comunicación entre robots

Para poder diagramar esto se utilizan dos parámetros, ρ_i y τ_i. El primero, ρ_i, representa la frecuencia con la cual el robot r_i emite mensajes broadcast informando su actividad. El segundo parámetro, τ_i (utilizado en el cálculo de la impaciencia y de la aquiescencia), representa un nivel adicional de tolerancia a fallos, mediante un período de tiempo en el cual el robot r_i no recibe mensajes de otro robot sin considerar que este cesó la tarea.

Luego se define la función que indica si el robot recibió el mensaje de broadcast de otro:

$$comm\_received(i,k,j,t_1,t_2) = \begin{cases} 1, & \text{si el robot } r_i \text{ recibió un mensaje} \\ & \text{del robot } r_k \text{ referenciando la tarea} \\ & h_i(a_{ij}) \text{ en el intervalo } (t_1,t_2),\ t_1 < t_2, \\ 0, & \text{en caso contrario.} \end{cases}$$

4.6. Supresión entre comportamientos motivacionales

Un robot puede y debe tener solo un comportamiento motivacional activo en cada momento para evitar conflictos. Para poder cumplir con esto, cuando un comportamiento motivacional se activa, comienza a inhibir a los demás comportamientos motivacionales del propio robot para evitar que estos se activen.

Cuando el robot termina la tarea o decide abandonarla, desactiva el conjunto de comportamientos correspondiente y deja de inhibir a los otros comportamientos motivacionales.

$$activity\_supression_{ij}(t) = \begin{cases} 0, & \text{si algún otro conjunto de comportamientos } a_{ik} \\ & \text{se encuentra activo,} \\ & k \neq j, \text{ en el robot } r_i \text{ en el tiempo } t, \\ 1, & \text{en caso contrario.} \end{cases}$$

4.7. Impaciencia del robot

Para poder calcular la impaciencia de un robot, se definen 3 parámetros, $\phi_{ij}(k,t)$, $\delta\_slow_{ij}(k,t)$, y $\delta\_fast_{ij}(t)$. El primer parámetro, $\phi_{ij}(k,t)$, indica el tiempo durante el cual el robot r_i permite que las comunicaciones del robot r_k afecten la motivación del conjunto de comportamientos a_{ij}. Los parámetros $\delta\_slow_{ij}(k,t)$ y $\delta\_fast_{ij}(t)$ indican las tasas de impaciencia del robot r_i respecto al conjunto de comportamientos a_{ij} mientras un robot r_k esté trabajando en la tarea $h_i(a_{ij})$ o mientras no haya ningún robot trabajando en la tarea $h_i(a_{ij})$ respectivamente.

Definimos primero 2 condiciones de recepción de comunicación:

- $C_1 : comm\_received(i,k,j,t-\tau_i,t) = 1$
- $C_2 : comm\_received(i,k,j,0,t-\phi_{ij}(k,t)) = 0$

Utilizando estas definiciones, la función se expresa de la siguiente forma:

$$impatience_{ij}(t) = \begin{cases} \text{mín}_k\left(\delta\_slow_{ij}(k,t)\right) & \text{si } C_1 \text{ y } C_2 \\ \delta\_fast_{ij}(t) & \text{en otro caso} \end{cases} \tag{4.1}$$

Además de poder calcular la impaciencia del robot, es necesario volver a 0 el valor de la impaciencia; para esto se define la siguiente función:

Definimos δt como el tiempo transcurrido desde el último chequeo de comunicación. Sean C_1 y C_2 las condiciones de recepción tales que:

- $C_1 : comm\_received(i,k,j,t-\delta t,t) = 1$
- $C_2 : comm\_received(i,k,j,0,t-\delta t) = 0$

El reinicio de la impaciencia se determina mediante la siguiente función:

$$impatience\_reset_{ij}(t) = \begin{cases} 0 & \text{si } \exists k \mid (C_1 \text{ y } C_2) \\ 1 & \text{en otro caso} \end{cases} \tag{4.2}$$

Esta función de reinicio hace que la motivación se restablezca a 0 si el robot r_i acaba de recibir su primer mensaje del robot r_k indicando que r_k está realizando la tarea $h_i(a_{ij})$. Esta función permite que la motivación se restablezca

no más de una vez por cada miembro del equipo del robot que intente la tarea $h_i(a_{ij})$. Permitir que la motivación se restablezca repetidamente por el mismo robot permitiría que un robot persistente, pero que falla, pusiera en peligro la finalización de la misión.

4.8. Aquiescencia del robot

Se utilizan dos parámetros para implementar la aquiescencia, $\psi_{ij}(t)$ y $\lambda_{ij}(t)$. El primero, $\psi_{ij}(t)$, es el tiempo en el que el robot r_i quiere mantener activo el conjunto de comportamientos a_{ij} antes de abandonarlo y dejarlo a otro robot. El segundo parámetro, $\lambda_{ij}(t)$, es el tiempo que el robot r_i mantiene el conjunto de comportamientos a_{ij} activo antes de darse por vencido e intentar otra tarea.

$$acquiescence_{ij}(t) = \begin{cases} 0 & \text{si [(el conjunto de comportamientos } a_{ij} \text{ del robot } r_i \\ & \text{ha estado activo por mas de } \psi_{ij}(t) \\ & \text{unidades de tiempo en el tiempo } t) \text{ y} \\ & (\exists x.comm\_received(i, x, j, t - \tau_i, t) = 1)] \\ & \text{o} \\ & \text{(el conjunto de comportamientos } a_{ij} \text{ del robot } r_i \\ & \text{ha estado activo por mas de } \lambda_{ij}(t) \\ & \text{unidades de tiempo en el tiempo } t) \\ 1 & \text{en otro caso} \end{cases}$$

Esta función dice que un robot r_i no abandonará un conjunto de comportamientos a_{ij} hasta que se cumpla una de las siguientes condiciones:

- r_i ha trabajado en la tarea $h_i(a_{ij})$ por un tiempo mayor a $\psi_{ij}(t)$ y otro robot ha tomado la tarea $h_i(a_{ij})$.
- r_i ha trabajado en la tarea $h_i(a_{ij})$ por un tiempo mayor a $\lambda_{ij}(t)$.

4.9. Cálculo de la motivación

Por último, el cálculo de la motivación se realiza de la siguiente manera:

$$\begin{aligned} m_{ij}(0) &= 0 \\ m_{ij}(t) &= [m_{ij}(t-1) + impatience_{ij}(t)] \\ &\quad \times sensory\_feedback_{ij}(t) \\ &\quad \times activity\_suppression_{ij}(t) \\ &\quad \times impatience\_reset_{ij}(t) \\ &\quad \times acquiescence_{ij}(t) \end{aligned} \tag{4.3}$$

Al principio la motivación m_{ij} correspondiente al robot r_i para el conjunto de comportamientos a_{ij} se inicializa en 0. A medida que el tiempo t avanza, este valor se incrementa a menos que suceda alguna de estas situaciones:

- $sensory\_feedback_{ij}\,(t) = 0$: la información sensorial indica que el conjunto de comportamientos no es aplicable en ese momento.
- $activity\_suppression_{ij}\,(t) = 0$: hay otro conjunto de comportamientos de r_i activo.
- $impatience\_reset_{ij}\,(t) = 0$: otro robot tomó la tarea $h_i(a_{ij})$ por primera vez.
- $acquiescence_{ij}\,(t) = 0$: el robot decide abandonar la tarea $h_i(a_{ij})$.

En cualquiera de estas situaciones, la motivación vuelve a 0. En otro caso, la motivación crece hasta superar el umbral θ y el conjunto de comportamientos correspondiente pasa a estar activo en el robot r_i. Cuando un conjunto de comportamientos a_{ij} se activa en el robot r_i, r_i transmite su actividad a los otros robots a una tasa ρ_i.

Capítulo 5

Implementación del motor de ALLIANCE

Este capítulo describe la implementación del motor de Alliance, detallando la arquitectura del sistema, los componentes principales, la interacción entre ellos y las decisiones de diseño clave. Se presentan diagramas UML para facilitar la comprensión de la estructura y el flujo del sistema.

5.1. Introducción

Si bien la arquitectura Alliance proporciona un marco conceptual robusto para la cooperación multi-robot, su aplicación práctica en diversas misiones y con equipos heterogéneos requiere una implementación de software eficiente y adaptable. En lugar de implementar la lógica de Alliance desde cero para cada nueva aplicación o caso de estudio, resulta beneficioso desarrollar un núcleo o "motor" genérico que encapsule los mecanismos fundamentales de la arquitectura.

Este motor abstraerá la lógica central de Alliance, incluyendo el cálculo de motivaciones, los mecanismos de selección de comportamientos y el manejo de la comunicación según los principios de la arquitectura (como la impaciencia, la aquiescencia y la detección de la actividad de otros agentes). Al separar esta lógica del control específico de las tareas o las particularidades del hardware de los robots, se logran varias ventajas clave:

- **Reusabilidad:** El motor puede ser integrado en una amplia gama de proyectos de robótica cooperativa que deseen emplear la arquitectura Alliance, sin necesidad de reimplementar sus principios fundamentales.
- **Modularidad y Mantenibilidad:** Facilita la organización del código y simplifica el mantenimiento, ya que las actualizaciones o mejoras a la lógica central de Alliance pueden realizarse en el motor sin afectar los comportamientos específicos de cada misión.

- **Agilidad en el Desarrollo:** Permite a los desarrolladores enfocarse en la implementación de los comportamientos necesarios para una tarea específica, asumiendo que la coordinación a nivel arquitectónico será manejada por el motor.
- **Plataforma para Experimentación:** Provee una base sólida y verificada para experimentar con variaciones de los parámetros de Alliance o proponer extensiones a la arquitectura.

Por estas razones, el desarrollo de un motor genérico de Alliance se justifica como una contribución valiosa para facilitar la aplicación de esta arquitectura en la comunidad de robótica.

5.2. Plataforma y lenguaje de base

Para la creación del "*Motor de Alliance*" se implementó un paquete de ROS 2 extensible con las funcionalidades básicas de Alliance, al que se llamó "`alliance`".

5.2.1. Acerca de ROS 2

ROS (Quigley, 2009) (Robot Operating System) es una plataforma de software de código abierto que proporciona un conjunto de herramientas, bibliotecas y convenciones diseñadas para facilitar el desarrollo de aplicaciones en robótica.

Además, cuenta con una comunidad de usuarios muy extensa y activa, que hace que la herramienta esté en desarrollo permanente y brinde las funcionalidades más recientes en robótica.

ROS 2 utiliza DDS (Data Distribution Service) como middleware de comunicación, lo que proporciona una comunicación robusta, escalable y eficiente. La naturaleza distribuida de Alliance y la necesidad de comunicación (sin depender de un nodo central) entre múltiples agentes hacen que ROS 2 sea una elección adecuada.

Para profundizar en las características de ROS 2 se puede consultar el Apéndice A de este documento.

En el marco de ROS 2, los **nodos** son procesos independientes que ejecutan una o varias funciones específicas dentro del sistema robótico, como, por ejemplo, el control de sensores, la planificación de movimientos o la coordinación de comportamientos. Estos nodos se comunican entre sí a través de un sistema de **tópicos**, que funciona como canales de mensajería basados en el paradigma *publicador–suscriptor*. En este modelo, un nodo puede *publicar* mensajes en un tópico, mientras que uno o varios nodos pueden *suscribirse* a dicho tópico para recibir esa información. De esta manera, ROS 2 asegura una comunicación desacoplada, escalable y distribuida entre los distintos componentes del sistema.

Se utilizaron nodos de ROS 2 para representar los comportamientos motivacionales y tópicos para la comunicación explícita entre nodos.

La versión utilizada para la implementación fue ROS 2 Humble Hawksbill (Open Source Robotics Foundation, 2025b).

Los dos lenguajes utilizados por la comunidad y recomendados para la realización de proyectos con ROS/ROS 2 son C++ y Python. Las bibliotecas ofrecidas por ROS y ROS 2 están plenamente implementadas en ambos lenguajes; por lo tanto, la elección de uno u otro no implica ninguna restricción al alcance de este sistema.

La elección del lenguaje de programación no representa un requerimiento (no funcional) para el desarrollo del proyecto.

Debido a la experiencia previa de los integrantes del equipo, se definió utilizar Python como lenguaje de programación para implementar la lógica de los comportamientos y el mecanismo de arbitraje.

En el ecosistema de ROS 2, el desarrollo se basa en la biblioteca cliente de bajo nivel `rcl` (*ROS Client Library*), que proporciona la interfaz central de comunicación. Sobre esta base, se construyen las implementaciones específicas para cada lenguaje: para Python, la biblioteca cliente oficial es `rclpy` (*ROS Client Library for Python*), mientras que para C++ es `rclcpp`. Esta arquitectura de doble implementación garantiza la interoperabilidad total, permitiendo que distintos componentes del sistema, desarrollados en lenguajes diferentes (por ejemplo, `rclpy` para la lógica de alto nivel y `rclcpp` para tareas de rendimiento crítico), coexistan y se comuniquen fluidamente dentro del mismo paquete de ROS 2.

5.2.2. Herramientas del Ecosistema Python

La biblioteca `rclpy` provee las clases y funciones esenciales para el *middleware* de ROS, permitiendo la creación de nodos, la gestión de la comunicación (tópicos, servicios y acciones) y el manejo de la concurrencia a través de sus distintos **ejecutores** (*executors*). Adicionalmente, el desarrollo en Python se complementa con paquetes de utilidad como `rosdep` para la gestión de dependencias, los sistemas de construcción `ament_cmake` y `ament_python` para el *build* de paquetes mixtos, y herramientas de alto nivel como `ros2cli` para la administración desde la línea de comandos. Juntos, `rclpy` con estas y otras herramientas forman el ecosistema completo que facilita la implementación eficiente y robusta de sistemas robóticos, asegurando la correcta concurrencia y el despliegue de los componentes.

5.3. Comunicación entre agentes

Todo el modelo de comunicaciones de la solución se resuelve mediante las primitivas que ofrece ROS 2, un entorno completamente distribuido. Esta es la principal diferencia con su predecesor ROS, que requería de un nodo maestro para su ejecución y, por lo tanto, generaba un cuello de botella que hacía que una falla en el nodo maestro paralizara la operativa de todos los robots, con la consecuencia de que no se podría concluir la misión.

De esta forma, se abstraen los problemas de comunicación inherentes al tipo de red al que se conectan los robots, área de cobertura, descubrimiento, etc., y quedan excluidos del alcance del presente proyecto, siendo ROS 2 el encargado de lidiar con algunos de dichos problemas.

No obstante, es posible que ocurra un fallo en las comunicaciones de algún robot en particular (entre otros tantos tipos de fallos) pero este caso sí está cubierto por Alliance, y el sistema visto como un todo (es decir, el conjunto de todos los robots) garantiza la adaptación, la reorganización y redistribución de tareas, y por lo tanto, la finalización exitosa de la misión.

Las primitivas de comunicación entre nodos ofrecidas por ROS 2 son:

- **tópicos**: publicación estilo broadcast, lectores y escritores asíncronos.
- **servicios**: request nominado a un nodo específico, response al emisor.
- **acciones**: similar a los servicios pero con callbacks intermedios que indican avance de la acción. Posibilidad de cancelar la acción.

En la presente solución, se utilizaron únicamente tópicos como forma de comunicación entre los nodos. Dado que ROS 2 es un sistema totalmente distribuido, cualquier nodo puede suscribirse a un tópico y recibir sus mensajes.

Como la única forma que tiene ROS 2 para diferenciar los tópicos es por su nombre, entonces, a los tópicos que se utilizan localmente entre componentes internos de un robot se les añade como prefijo en su nombre el id del robot que los utiliza. La misma idea se aplica a los tópicos referidos a una tarea en particular, conteniendo además un prefijo con el id de la tarea.

5.3.1. Paquete ALLIANCE_INTERFACES

Todas las interfaces, y en particular los tópicos que se utilizaron para este proyecto, operan sobre un tipo de datos. El tipo de datos puede ser simple o estructurado. ROS 2 provee varios tipos de datos predefinidos en sus librerías, por ejemplo, en las librerías: `std_msgs` o `geometry_msgs`.

ROS 2 utiliza un lenguaje de descripción simplificado, el lenguaje de definición de interfaz (IDL, por sus siglas en inglés), para describir estas interfaces. Esta descripción facilita que las herramientas de ROS 2 generen automáticamente el código fuente para el tipo de interfaz en varios lenguajes de destino.

Para el presente proyecto se decidió crear todas las interfaces estructuradas en un nuevo paquete llamado $ALLIANCE\_INTERFACES$. En él se definen los siguientes tipos de datos estructurados utilizados por los tópicos del paquete $ALLIANCE$:

- `Comando.msg`: Este mensaje se utiliza para pasar los comandos desde el nodo `Motivacional` al nodo `Executer`. El número que se transmite corresponde al enumerado `ComandoEjecutable`, que está definido en el archivo auxiliar `constantesGlobales.py`. Los valores de este enumerado son: (`NOTHING=-1`, `RUN=1`, `PAUSE=2`, `RESUME=3`, `STOP=4`). Su estructura es

la siguiente:

```
int32   comando
```

- `InterRobotComm.msg`: Este mensaje se utiliza para la comunicación entre robots. El tipo de mensaje, `message_type`, se corresponde con el enumerado `TipoMensajeTarea` definido en el archivo auxiliar `constantesGlobales.py`. Los valores del enumerado son: (`TAREA_ACTIVA=1`, `TAREA_FINALIZADA=2`). Su estructura es:

```
uint32                    seq
builtin_interfaces/Time   timestamp
uint8                     robot_id
string                    robot_name
uint8                     task_id
uint8                     task_type
string                    task_name
uint8                     message_type
string                    message
```

- `Tareas.msg`: utilizado para indicar que hay una nueva tarea disponible. Su estructura es:

```
int32                       task_id
string                      task_name
int32                       task_time
geometry_msgs/PoseStamped   destination
```

5.4. Arquitectura General

La arquitectura del motor de Alliance se basa en una serie de nodos de ROS 2 que colaboran en la asignación y ejecución de tareas. Los componentes principales del sistema pueden verse en la Figura 5.1.

5.5. Interacción entre Componentes

A modo de ejemplo, se muestra el flujo entre componentes desde que se crea hasta que finaliza una tarea, en el *diagrama de secuencia UML* que puede verse en la Figura 5.2.

Dicho flujo de interacción entre los componentes se puede resumir de la siguiente manera:

1. El `Generador de Tareas` (o una fuente externa) introduce una nueva tarea en el sistema.
2. El `Controller` recibe la tarea y crea los nodos `Motivacional` y `Executer` correspondientes.
3. Cada nodo `Motivacional` calcula la motivación de cada robot para la tarea.

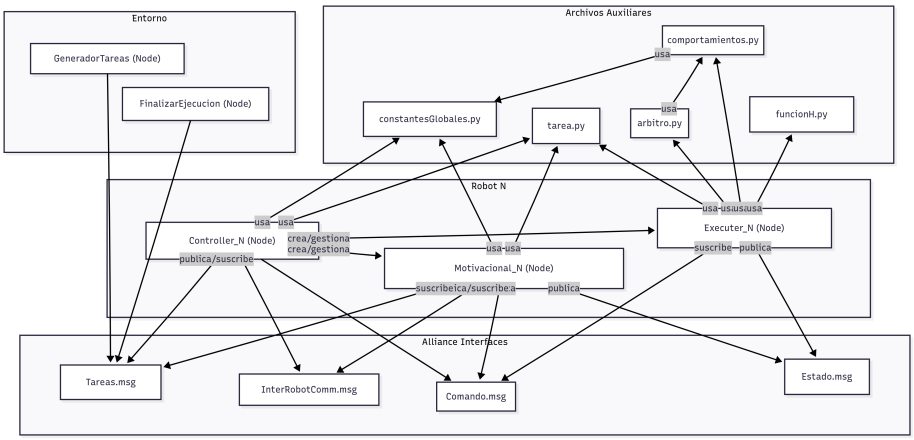

Figura 5.1: Diagrama de componentes

4. El `Motivacional`, basándose en el cálculo de la motivación, asigna la tarea al conjunto de comportamientos correspondiente al `Arbitro` que contiene el nodo `Executer`.

5. El nodo `Executer` del robot asignado despierta al `Arbitro` y sus comportamientos asignados.

6. El `Arbitro` despierta y duerme los comportamientos necesarios para completar la tarea dependiendo del estado de los sensores.

7. Un comportamiento reconoce que finalizó la tarea, da aviso al `Executer` y finaliza.

8. El `Arbitro` finaliza el resto de los comportamientos. El `Executer` avisa al `Motivacional` que terminó la tarea y finaliza.

9. El `Motivacional` finaliza y se liberan los recursos utilizados por la tarea.

5.6. Nodo Controller

La necesidad de contar con un nodo `Controller` surge al tomar la decisión de aceptar la aparición de tareas de forma dinámica, en conjunción con la definición de que los nodos que las van a resolver también serán creados y eliminados dinámicamente.

La función del nodo `Controller` es permanecer toda la ejecución escuchando en el tópico donde se publican las nuevas tareas, `'/tareas'`. Cada vez que aparece una nueva tarea t_j para resolver, el nodo $Controller_i$ del robot r_i es el encargado de crear dinámicamente un nodo $Motivacional_{ij}$ y un nodo $Executer_{ij}$, previa verificación de que el robot r_i es capaz de resolver dicha tarea, es decir,

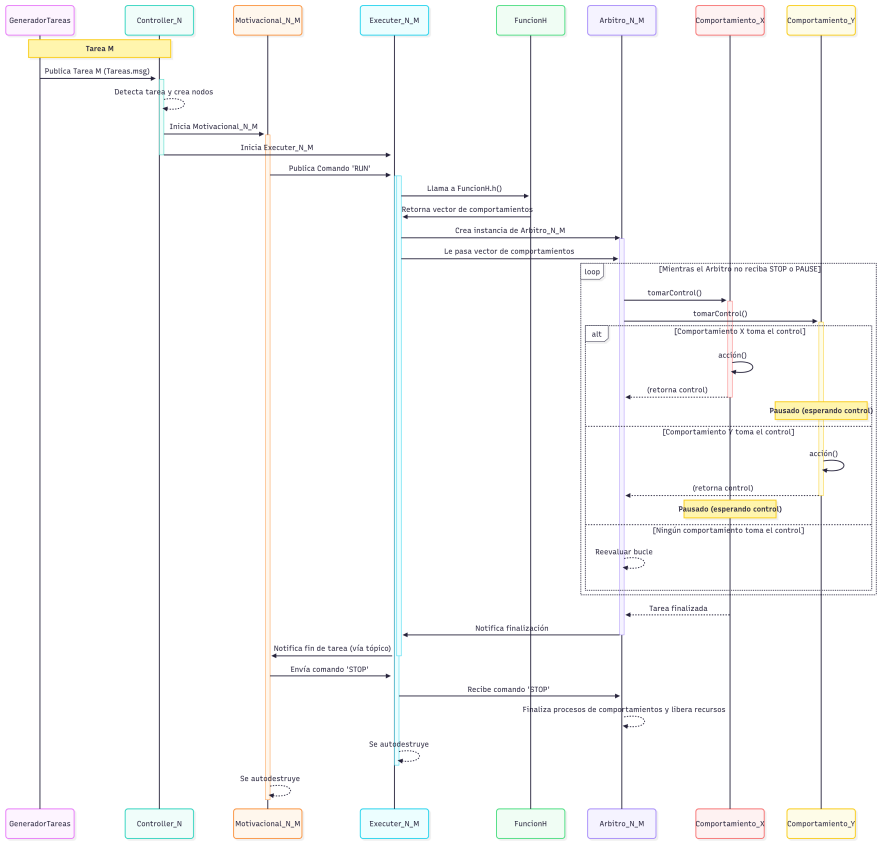

Figura 5.2: Diagrama de Secuencia para una tarea.

se valida que el robot r_i tiene un conjunto adecuado de comportamientos que permite resolver la tarea t_j.

Por último, el nodo `Controller` define y utiliza un ejecutor del tipo `MultiThreadedExecutor` de la biblioteca Python `rclpy`, para gestionar la ejecución de código en cada nodo.

En ROS 2, un *executor* es el componente responsable de gestionar la ejecución de las devoluciones de llamada (*callbacks*) generadas por los nodos en respuesta a eventos como la recepción de mensajes, la activación de temporizadores o la invocación de servicios. La biblioteca `rclpy` provee diferentes variantes de ejecutores que permiten definir el modelo de concurrencia:

- **SingleThreadedExecutor**: procesa todas las devoluciones de llamada en un único hilo de ejecución. Es sencillo de utilizar y adecuado en aplicaciones con baja concurrencia, pero puede convertirse en un cuello de botella cuando múltiples callbacks deben atenderse de manera casi simultánea.

- **MultiThreadedExecutor**: emplea un conjunto de hilos para ejecutar callbacks de manera concurrente. Esto permite que eventos independientes, como la recepción de tareas y la creación de nuevos nodos, se procesen en paralelo, reduciendo el tiempo de respuesta del sistema.

- **Executores personalizados**: ROS 2 permite extender la clase base de los ejecutores para diseñar estrategias específicas de planificación, como la asignación de prioridades o el control explícito del número de hilos disponibles.

Se selecciona un **MultiThreadedExecutor** para el nodo `Controller`, ya que este debe atender simultáneamente varios tipos de eventos: por un lado, la suscripción al tópico `/tareas` para la recepción de nuevas asignaciones, y por otro, la creación y supervisión de los nodos `Motivacional` y `Executer`. El uso de múltiples hilos asegura que la llegada de una nueva tarea no quede bloqueada por operaciones que requieren mayor tiempo de procesamiento, manteniendo así la capacidad de respuesta del sistema.

En resumen, las principales responsabilidades del nodo `Controller` son:

- Recibir nuevas tareas.

- Crear dinámicamente nodos `Motivacional` y `Executer` para cada tarea.

- Gestionar la ejecución concurrente de los nodos.

5.7. Nodo Executer

El nodo `Executer` es responsable de llevar a cabo las acciones necesarias para completar una tarea. Contiene una instancia de la clase `Arbitro` y utiliza instancias de la clase `Comportamiento`.

La instancia de la clase `Arbitro` es la que se encarga de gestionar los comportamientos del conjunto aplicable.

Este nodo conoce el id, nombre y tipo de tarea que debe resolver y para la cual fue creado.

Se invoca la función $h()$ de la clase `FuncionH` que, dado el tipo de tarea a resolver, retorna el conjunto aplicable de comportamientos. Luego, el conjunto de comportamientos devuelto pasa a la instancia del `Arbitro` y éste se encarga de instanciar cada comportamiento y gestionar su ejecución.

Se suscribe al tópico '`CommandExecuter_T`$[idTipoTarea]$_$[idTarea]$' donde se reciben los comandos desde el nodo `Motivacional` correspondiente a la tarea para la que fue creado.

Dependiendo del valor de la motivación y del estado actual del robot, el nodo `Motivacional` envía el comando que el nodo `Executer` debe pasar al `Arbitro`. Los comandos son: `RUN`, `PAUSE`, `RESUME`, `STOP`. Estos comandos ejecutan las respectivas funciones de la clase `Arbitro` y aplican a todo el conjunto de comportamientos que el `Arbitro` gestiona.

5.7.1. Clase Árbitro

Esta clase gestiona la ejecución de un conjunto seleccionado de comportamientos en un robot, que operan en conjunto para resolver una tarea. Cada comportamiento se ejecuta de forma concurrente en un hilo separado. De esta forma, cada vez que el `Arbitro` define pausar un comportamiento, éste se detiene sin ocupar procesador. Los comportamientos son mutuamente excluyentes, es decir, la activación de un comportamiento pausa la ejecución de los demás.

Al inicio, cuando el `Arbitro` recibe el comando `RUN`, comienza a arbitrar, se ejecutan todos los hilos de forma concurrente, cada uno correspondiente a un comportamiento, e inmediatamente el `Arbitro` los pausa a todos. Luego el nodo `Executer` corre un *timer* cuyo evento *tic* es atendido por una función `timerCallback()` que invoca a la función `arbitrar()` de la instancia del `Arbitro`. De esta forma se simula un bucle dentro de la función `arbitrar()`. En cada iteración se consulta la función `takeControl()` de cada comportamiento, que retorna `true` si el aparato sensorial del robot indica que el comportamiento es aplicable, o `false` en caso contrario. En caso de que la respuesta sea `true`, el `Arbitro` activa dicho comportamiento, despertando la ejecución del hilo mediante la función `resume()` de la instancia que debe activar. En caso de que la respuesta sea `false`, el `Arbitro` ejecuta la función `pause()` del comportamiento y lo pone a dormir. Puede verse un pseudocódigo de la función `arbitrar()` en la Figura 5.3.

Las condiciones de parada son:

- Un comportamiento detecta que la tarea está completa, informa al `Executer` y al `Árbitro` correspondiente y finaliza. Luego, el `Arbitro` sale del bucle principal, termina todos los demás comportamientos, indica al nodo `Executer` que debe enviar un mensaje de tarea finalizada al `Motivacional` correspondiente, y luego finaliza. Después de notificar al `Motivacional`, el `Executer` también finaliza. Por último, el nodo `Motivacional`, luego de recibir el mensaje, también finaliza su ejecución y así se liberan todos los recursos utilizados para resolver la tarea que finalizó.

- ídem al caso anterior, pero esta vez originado por un comando `STOP` del nodo `Motivacional` que indica que otro robot finalizó la misma tarea.

- El nodo `Motivacional` decide que el robot se va a dedicar a resolver otra tarea, por lo tanto va a activar otro conjunto de comportamientos de otro nodo `Executer`, y le envía a este nodo `Executer` el comando `PAUSE`. De esta forma el `Arbitro` pausa y pone a dormir a todos los hilos correspondientes a todos los comportamientos de su conjunto.

5.7.2. Clase Comportamiento

Genéricamente cada instancia de esta clase se encarga de invocar a las funciones motoras del robot para exhibir un comportamiento específi-

Algorithm 1: Procedimiento `arbitrar()`

```
1  Procedimiento arbitrar():
2      if no stopped then
3          if paused then
               // Si está pausado, se pausa el comportamiento
                  actual
4              current_comp.pause()
5          else
6              foreach c en comps do
7                  if c no está detenido then
8                      if c.takeControl() es verdadero then
                           // Este comportamiento toma el control
9                          mensaje ← ''<nombre>toma el control'' ;
10                         current_comp ← c ;
11                         current_comp.reset() ;
12                         current_comp.resume()
13                     else
                           // El comportamiento no puede tomar el
                              control
14                         c.pause()
15                 else
                       // El comportamiento terminó su ejecución
16                     if c.tareaFinalizada es verdadero then
17                         stop()
18     else
           // El árbitro está detenido, detener todos los
              comportamientos
19         foreach c en comps do
20             c.stop()
```

Figura 5.3: Lógica del procedimiento de arbitraje para la selección de comportamientos.

Algorithm 2: Procedimiento `run()`

```
1 Procedimiento run():
2     while no stopped do
3         if paused then
              // Si está pausado, espera antes de continuar
4             wait()
5         if no stopped y no paused then
              // Ejecuta la acción principal
6             action() ;
7             sleep(refresh_rate)
      // Al salir del bucle, realiza tareas de finalización
8     post_stop()
```

Figura 5.4: Lógica del algoritmo principal de cada comportamiento.

co, por ejemplo: `AvanzarSegs`, `RotarAngulo`, `Detenerse`, `EvitarObstáculo`, `NavegarDeFormaSegura`, `BordearPared`, etc.

Debido al diseño de la interacción que se definió para el `Arbitro` y los comportamientos, la clase `Comportamiento` es una clase abstracta. Cada hilo de ejecución de un comportamiento está regido por la función `run()` que se invoca una única vez desde el `Arbitro`. Luego, el hilo del comportamiento puede ser pausado (dormido) o reactivado hasta que finaliza la ejecución. Cada vez que el hilo se despierta, ejecuta la función `action()` del comportamiento correspondiente.

Puede verse un pseudocódigo de la función `run` de cada comportamiento en la Figura 5.4.

Como la clase `Comportamiento` es abstracta, en cada implementación es necesario extender las siguientes funciones:

- `getNombre()`: Devuelve el nombre del comportamiento.
- `takeControl()`: Devuelve `true` si los sensores indican que el comportamiento es aplicable y puede tomar el control o `false` en caso contrario.
- `action()`: Realiza las acciones motoras necesarias para que se ejecute el comportamiento.
- `post_stop()`: Realiza las tareas de finalización necesarias al culminar una tarea.

La idea de *Alliance* es que esas funciones se utilicen para implementar comportamientos relativamente simples, que acoplen fuertemente sensado con actuación, es decir, que respondan al paradigma *Reactivo*.

Luego, también se debe extender la función $h()$ de la clase `FuncionH`. Esa función, como se explica en la Sección 5.7, recibe como parámetro un robot y el

tipo de tarea a resolver, y retorna el conjunto aplicable de comportamientos que resuelven la tarea, que son los comportamientos implementados específicamente mediante la extensión de las funciones explicadas anteriormente.

5.8. Nodo Motivacional

El nodo `Motivacional` calcula el nivel de motivación del robot para realizar una tarea. Para realizar el cálculo, se implementan las funciones explicadas en la Sección 4.9.

Estas son:

- $impatience_{ij}(t)$
- $sensory\_feedback_{ij}(t)$
- $activity\_suppression_{ij}(t)$
- $impatience\_reset_{ij}(t)$
- $acquiescence_{ij}(t)$

Además `Motivacional` define un `Timer` que genera un evento `timerCallback()` a una tasa ρ_i. Cada evento del timer invoca a la función `cicloPrincipalAlliance()`, y de esta forma se simula un bucle a una tasa predeterminada.

En cada iteración del ciclo principal del nodo `Motivacional` se calcula la motivación $m_{ij}(t)$ de la siguiente forma:

$$\begin{aligned} m_{ij}(0) &= 0 \\ m_{ij}(t) &= [m_{ij}(t-1) + impatience_{ij}(t)] \\ &\quad \times sensory\_feedback_{ij}(t) \\ &\quad \times activity\_suppression_{ij}(t) \\ &\quad \times impatience\_reset_{ij}(t) \\ &\quad \times acquiescence_{ij}(t) \end{aligned} \tag{5.1}$$

Si el valor de la motivación $m_{ij}(t)$ (que es el valor m definido en el nodo $Motivacional_{ij}$ para el robot r_i y la tarea $h_i(a_{ij})$) supera el umbral θ, entonces se envía por broadcast aviso de que el robot r_i tomó la tarea t_j y se envía el comando `RESUME` al nodo $Executer_{ij}$ correspondiente. La tarea t_j se expresa como $h_i(a_{ij})$, ya que la función $h_i(a_{ij})$ del robot r_i devuelve la tarea correspondiente al conjunto de comportamientos a_{ij} (del robot r_i para la tarea t_j).

Se utilizan las siguientes funciones auxiliares:

- *broadcast* (*messageType*): Esta función es la que publica el mensaje broadcast indicando o bien que se tomó la tarea con el `messageType`

`= TipoMensajeTarea.TAREA_ACTIVA` o que se finaliza la tarea pasando `messageType = TipoMensajeTarea.TAREA_FINALIZADA`. Los mensajes son enviados al tópico "`/interRobotComm`".

- *interRobotCommCallback* (*data* : *InterRobotComm*): Esta función implementa el evento que recibe un mensaje del tópico "`/interRobotComm`", indicando que otro robot tomó o finalizó alguna tarea. También se reciben los mensajes enviados por el propio robot mediante *broadcast*().

- *commReceived* (*idRobot*, t_1, t_2): Esta función indica si el robot recibió un mensaje de broadcast del robot con id = *idRobot* en el intervalo de tiempo t_1, t_2. Para esto, se mantienen vectores que indican la primera y última comunicación de cada robot. Estos vectores se actualizan en la función *interRobotCommCallback* ().

5.8.1. Parámetros dinámicos

Los parámetros siguientes, explicados en la Sección 4.7 y la Sección 4.8, se implementaron como funciones dependientes del tiempo. De esta forma, extendiendo dichas funciones es posible ajustar y calcular los parámetros dinámicamente. Esto permite por ejemplo, utilizar algoritmos de aprendizaje automático para mejorar el rendimiento.

Los parámetros son:

- $\phi_{ij}\,(k,t)$
- $\delta\_slow_{ij}\,(k,t)$
- $\delta\_fast_{ij}\,(t)$
- $\psi_{ij}\,(t)$
- $\lambda_{ij}\,(t)$

Las funciones que se pueden extender son respectivamente:

- `phi(idRobot, t)`
- `deltaSlow(idRobot, t)`
- `deltaFast(t)`
- `psi(t)`
- `parmLambda(t)`

5.9. Decisiones de Diseño Clave

5.9.1. Arquitectura de Dos Nodos (Motivacional/Executer)

Se optó por separar la lógica de motivación y ejecución en dos nodos distintos (`Motivacional` y `Executer`) para mejorar la modularidad, la claridad del código y la flexibilidad del sistema. El diagrama mostrado en la Figura 5.5 y propuesto en la definición de la arquitectura Alliance así lo sugiere.

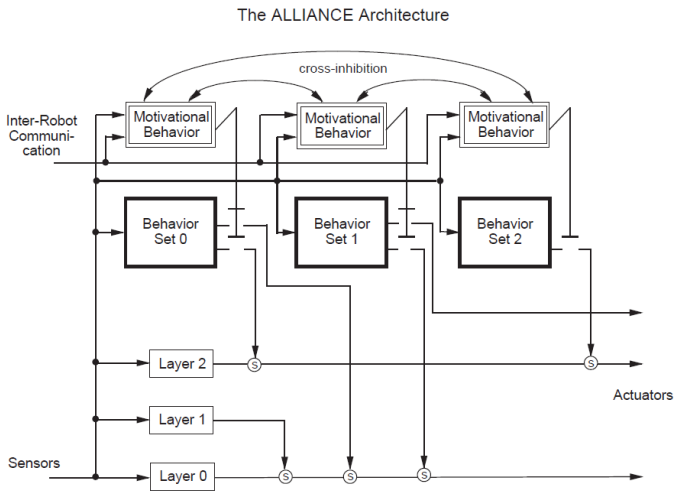

Figura 5.5: Diagrama de la Arquitectura ALLIANCE presentado en el artículo de Parker 1995

5.9.2. Dinamismo de Tareas y Gestión Dinámica de Nodos

Observando que existen muchos casos en los que no se puede disponer de las tareas al inicio de la ejecución, se decidió modificar levemente Alliance para dar soporte a la creación dinámica de tareas. Una posible implementación de esta funcionalidad es permitir la creación y eliminación de nodos dinámicamente a medida que las tareas aparecen. Esa fue la forma en que se resolvió. De esta manera surge la necesidad de utilizar un nodo `Controller`. Además, para lograr una correcta liberación de los recursos computacionales al finalizar una tarea, se debió modificar la función de *broadcast*() para que utilizara un parámetro que indique si un nodo está tomando una tarea o la acaba de finalizar.

Nodo Generador de Tareas

El nodo Generador de Tareas es un componente opcional utilizado para simular la aparición dinámica de tareas en el sistema, lo cual facilita las pruebas

y la experimentación.

5.10. Ventajas y Limitaciones

El diseño implementado ofrece las siguientes ventajas:

- **Reusabilidad:** El motor de Alliance es reutilizable en diferentes aplicaciones.
- **Modularidad:** La arquitectura modular facilita el mantenimiento y la extensión del sistema.
- **Flexibilidad:** El sistema se adapta a diferentes tipos de tareas y robots.

Las limitaciones incluyen:

- **Complejidad:** La gestión dinámica de nodos añade complejidad al diseño.
- **Sobrecarga:** La creación y eliminación frecuente de nodos puede generar sobrecarga.

Capítulo 6

Caso de estudio: Granja de manzanas

En este capítulo se describe la solución al problema planteado en el caso de estudio. Se plantea el problema a resolver, se explica de qué forma se reutilizó el "Motor de Alliance" para resolverlo y se discuten las decisiones de diseño más relevantes.

6.1. Introducción

El problema de la optimización de la recolección de manzanas en una granja ofrece un nivel de dificultad interesante en el que se puede aplicar un equipo de robots que cooperan para resolverlo. En particular, este tipo de problemas presenta un ambiente parcialmente observable, estocástico, secuencial, dinámico y continuo. Estas son características de complejidad suficientes como para hacer del problema interesante para ser abordado en un proyecto de grado como el presente.

6.2. Planteo del problema

El caso de estudio plantea el problema de optimizar la recolección de manzanas en una granja mediante el uso de un conjunto de robots que operan coordinándose y colaborando entre sí. Los robots tienen la capacidad de transportar una cierta cantidad de manzanas desde el punto de recolección hasta ciertas bases predefinidas. La forma del robot es similar a la de un tráiler que puede ser con o sin barandas. La forma de transportarlas es a priori irrelevante para el problema, es decir, las manzanas pueden volcarse y transportarse a granel en el tráiler o ser transportadas en los mismos canastos que utilizan los recolectores sobre el tráiler. Los manzanos (árboles de manzanas) están dispuestos en filas. En cada fila, los árboles están a una cierta distancia fija cada uno del siguiente.

Entre filas hay un espacio que forma un sendero donde pueden circular tanto los robots como los recolectores. No se especifica si los recolectores son humanos u otros robots; es irrelevante para el problema, ya que los robots deben desplazarse evitando colisiones con: árboles, otros robots, recolectores (sean humanos o no). Los puntos de recolección se ubican cada uno al lado de algún árbol, ya que se supone que es ahí el lugar donde un recolector llenó su canasto con manzanas. Las bases en las que los robots llenos vaciarán su carga se encuentran en los bordes de la granja o en las esquinas de ésta, o en las cabeceras de las sendas.

Entonces, los pasos del proceso son:

- Cada recolector junta manzanas de los árboles en un canasto. En determinado momento su canasto se llena y emite una señal que llega al equipo de robots transportadores. En términos del proyecto, se genera una nueva tarea de recolección.
- Los robots que tienen disponibilidad deben definir cuál de ellos toma cada tarea.
- Los robots visitan uno o más puntos de recolección y se cargan con las manzanas de los recolectores.
- Los robots que estén lo suficientemente llenos pueden elegir una tarea de retorno a base para descargar.

6.2.1. Supuestos

Como hay ciertos puntos indefinidos en el planteo del problema, se realizaron los siguientes supuestos para acotarlo:

- Cada robot tiene una capacidad limitada de carga y ese valor es un parámetro del robot.
- Cada robot tiene un umbral de carga a partir del cual ya puede ir a una base a descargar. No es necesario que el robot esté lleno para ir a descargar, sólo que el nivel de carga supere el umbral. Ese valor es un parámetro del robot.
- Cada recolector que llena su canasto, tendrá una cierta cantidad de unidades de carga para subir a un robot. Esta cantidad no es fija y puede variar con cada tarea (recolector que llena su canasto de manzanas).
- Un robot dependiendo de su nivel de carga actual y lo que le falta para llenarse podrá tener capacidad para aceptar una tarea de recolección o no. Si el robot no puede aceptar una tarea de recolección, el aparato sensorial del mismo deberá indicarlo.
- Cada vez que un robot llega a una base, se descarga completamente su carga de manzanas y su nivel de carga actual pasa a ser cero.

- Un robot cuyo nivel de carga superó el umbral, comenzará a competir por las tareas de retorno a base como x tareas más junto a las otras, siendo x la cantidad de bases. Es decir, un robot que superó su umbral de carga todavía puede aceptar otras tareas de recolección. Tanto las tareas de recolección y las de retorno a base competirán para aumentar su motivación.

- Cargar las manzanas en el robot, es una tarea que se ejecuta automáticamente por el robot que llegó al punto de recolección pero esta tarea demora cierto tiempo predefinido. Esto se supone así porque en algún caso podría haber un comportamiento con un brazo robótico por ejemplo que se encargue de resolver la tarea. Aún siendo el humano quien pone las manzanas sobre el robot, esto requiere de cierto tiempo y no se resuelve instantáneamente. Ese tiempo de carga es un tiempo que el robot no podrá dedicar a ninguna otra tarea.

- Ídem al anterior pero para la tarea de descarga en una base.

- Las tareas de carga en un punto de recolección y descarga en una base se adjudican instantáneamente al robot que llegó. No hay que competir por las tareas de carga y descarga.

- Cada vez que un robot llega al destino de una tarea de recolección, se cargará con toda la carga de manzanas que tenga el recolector y ese valor se sumará al nivel de carga del robot. No habrá cargas parciales.

6.3. Detalles de la solución propuesta

Se implementó un paquete de ROS 2 al que se llamó `cocoro2` que es un acrónimo de "Cooperación y Coordinación entre Robots". El número 2 indica que la solución se realizó en ROS 2, ya que hubo una implementación previa en ROS llamada `cocoro`. Para la implementación se utilizó el "Motor de Alliance" implementado en el paquete `alliance` como base, sobre el que se extendieron algunas clases. Para ello, fue necesario definir correctamente las tareas y programar los conjuntos de comportamientos en cada robot que las resuelven. También se debió estudiar el tema de los sensores utilizados y la creación de una API para controlar el uso de los motores.

6.3.1. Tareas y sus características

Se identifican tres grupos de tareas:

- Tareas de $Recoleccion\_i\_j(x, y)$, siendo i el id del robot y j el id de la tarea. El robot debe dirigirse de forma segura hacia una ubicación pasada por parámetro en la tarea. Los parámetros de la tarea serán las coordenadas (x, y) del punto de destino donde se solicitó la tarea. Las tareas de recolección serán generadas dinámicamente.

- Tareas de retorno a base. Se parte de un número fijo y conocido de x bases. Al inicio de la misión se crea en cada robot una tarea de $Retorno\_a\_Base\_i\_x$, siendo i el id del robot y x el id de la base. La tarea vivirá durante toda la misión y podrá ser reutilizada. Habrá momentos en los que las tareas de retorno a base no compitan y permanezcan con su motivación en cero y otros casos en los que todas las tareas de retorno a base compitan con el resto.

- Tareas de $Carga\_i\_j$ y $Descarga\_i$, siendo i el id del robot y j el id de la tarea. Estas tareas no compiten, son asignadas automáticamente al robot que llega a un destino de recolección o a una base y tenían asignada la tarea de recolección o retorno a base correspondiente. Es decir, se asignan al robot que corresponde y no a otro que pasó de casualidad. Estas tareas demoran un tiempo predefinido en ejecutarse y terminan, liberando al robot para competir por otras tareas. Los robots que están ejecutando una tarea de $Carga$ o $Descarga$, no podrán competir por ninguna otra tarea ni abandonar la que están realizando hasta no finalizarla. Al inicio de la misión, cada robot crea una tarea $Descarga\_i$ que vivirá durante toda la misión y se activará solamente cuando el aparato sensorial del robot lo indique. Cada vez que aparece una nueva tarea de recolección, se crea además una nueva tarea de $Carga\_i\_j$ que una vez finalizada se eliminará y se activará solamente cuando el aparato sensorial del robot lo indique.

6.3.2. Comportamientos implementados

Dado que las tareas a resolver son, en esencia, iguales; esto es, trasladarse con seguridad hacia una ubicación objetivo dada por coordenadas (x, y), los comportamientos implementados solamente involucran algoritmos para el control del movimiento.

De los algoritmos para el control del movimiento estudiados, se implementó el `BUG 0`, presentado a continuación y se explican `BUG 1` y `BUG 2` a modo de referencia.

6.3.3. Algoritmos de control de movimiento

Se investigaron algoritmos de control de movimiento que contemplan las características de los robots y del escenario, cambiante y desconocido.

Algoritmo Bug 0

El algoritmo implementado fue el `Bug 0`, un enfoque clásico en la navegación robótica reactiva. Este algoritmo sigue un conjunto de reglas simples para alcanzar un objetivo desde un punto de inicio en un entorno desconocido, conociendo únicamente la posición del objetivo y basándose en sensores locales; en el caso del presente proyecto, el sensor de proximidad.

Las reglas principales de `Bug 0` son:

- Movimiento hacia el objetivo: El robot se desplaza directamente hacia el objetivo siempre que no detecte obstáculos.
- Evasión de obstáculos: Al encontrar un obstáculo, el robot lo rodea hasta regresar a una línea recta hacia el objetivo.
- Retorno hacia el objetivo: Una vez superado el obstáculo, el robot retoma el movimiento directo hacia el objetivo.

Puede verse el pseudocódigo del algoritmo `Bug 0` en la Figura 6.1 y la máquina de estados correspondiente en la Figura 6.2.

Para la implementación del algoritmo `Bug 0` se implementaron los comportamientos `NavegarDeFormaSegura` y `Esquivar` que heredan de la clase `Comportamiento` y extienden las funciones: `getNombre()`, `action()`, `takeControl()` y `post_stop()`.

`NavegarDeFormaSegura` se activa cuando no hay un obstáculo delante; es decir, si la función `takeControl()` no detecta un obstáculo próximo, devuelve `true`.

La velocidad pasada al robot consta de un componente lineal (velocidad lineal) y un componente angular (velocidad angular). Luego, la función `action()` avanza con el componente de velocidad lineal mayor que cero (valor preestablecido según la calibración) hacia el objetivo. Además, si el ángulo hacia el objetivo es mayor que cierto umbral, corrige la trayectoria añadiendo una pequeña velocidad angular en sentido horario o antihorario, según corresponda.

Cuando los sensores detectan un obstáculo próximo, la función `takeControl()` de `NavegarDeFormaSegura` retorna `false` y la correspondiente de `Esquivar` retorna `true`. Luego, la función `action()` de `Esquivar` produce un giro con una velocidad lineal baja o nula para evitar colisiones. Cuando el obstáculo queda fuera del alcance del sensor, se pausa el comportamiento `Esquivar` y se vuelve a activar `NavegarDeFormaSegura`.

Para poder calibrar los robots, se utilizaron las siguientes constantes que definen los valores de velocidad, entre otros:

- `VEL_LINEAL_ESQUIVA`, velocidad lineal mientras se está esquivando.
- `VEL_ANGULAR_ESQUIVA`, velocidad angular mientras se está esquivando.
- `VEL_LINEAL_RECTO`, velocidad lineal mientras se avanza hacia el objetivo en línea recta.
- `VEL_LINEAL_ORIENTA`, velocidad lineal mientras se avanza hacia el objetivo, corrigiendo la trayectoria en forma de arco.
- `VEL_ANGULAR_HORARIA`, velocidad angular mientras se navega hacia el objetivo, corrigiendo el ángulo.
- `VEL_ANGULAR_ANTIHORARIA`, velocidad angular mientras se navega hacia el objetivo, corrigiendo el ángulo.

Algorithm 3: Algoritmo `Bug0`

```
1 Algoritmo Bug0(inicio, objetivo):
2     posicion ← inicio ;
3     while posicion ≠ objetivo do
4         if camino directo hacia objetivo está libre then
              // Avanza en línea recta hacia el objetivo
5             ir directo hacia objetivo
6         else
7             while existe obstáculo en el camino do
                  // Seguir el contorno del obstáculo hasta
                     despejar el camino
8                 rodear obstáculo
      // Fin del algoritmo cuando se alcanza el objetivo
```

Figura 6.1: Lógica del algoritmo `Bug0` para el control del movimiento.

- `ANGULO_PARA_ORIENTAR`, ángulo en radianes a partir del cual se avanza en forma de arco, corrigiendo la trayectoria.

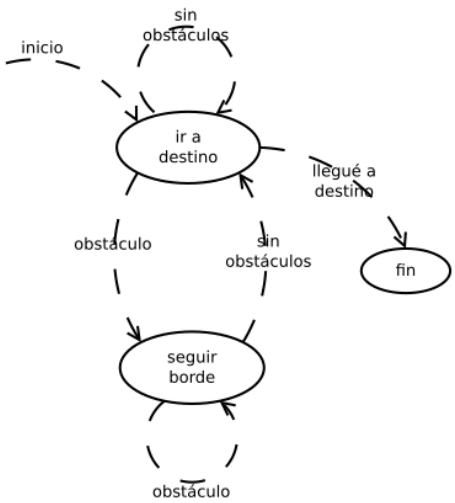

Figura 6.2: Máquina de estados del algoritmo Bug 0

Algoritmo Bug 1

El algoritmo de navegación `Bug1` es un enfoque simple para resolver el problema de mover un robot desde una posición inicial hasta un objetivo en un entorno con obstáculos. Puede verse un diagrama de la máquina de estados del algoritmo `Bug1` en la Figura 6.3. Se basa en las siguientes ideas:

- Movimiento directo al objetivo: El robot intenta avanzar directamente hacia el objetivo siguiendo una línea recta.
- Contorno de obstáculos: Si encuentra un obstáculo en su camino, el robot rodea el obstáculo siguiendo su contorno.
- Retorno hacia el objetivo: Después de rodear el obstáculo, el robot reanuda el movimiento directo hacia el objetivo desde el punto más cercano que alcanzó mientras bordeaba el obstáculo.

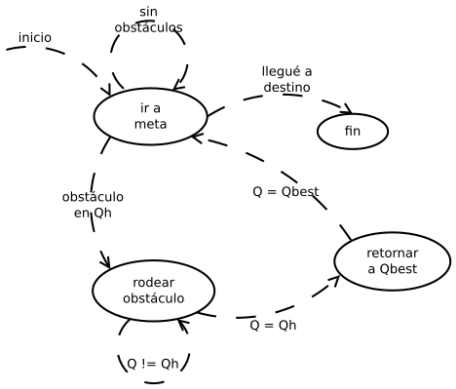

Figura 6.3: Máquina de estados del algoritmo Bug 1

La diferencia principal entre el algoritmo anterior y el actual se encuentra en la forma de evitar los obstáculos.

Algoritmo Bug 2

El algoritmo `Bug2` es una mejora sobre `Bug1` y `Bug0`, diseñado para optimizar la navegación hacia el objetivo en entornos con obstáculos. A diferencia de `Bug1`, no rodea completamente los obstáculos antes de decidir avanzar; en su lugar, sigue una estrategia basada en una línea recta llamada línea objetivo. Puede verse un diagrama de la máquina de estados del algoritmo `Bug2` en la Figura 6.4.

Características principales

- Línea objetivo: El robot sigue una línea recta entre la posición inicial y el objetivo, siempre que no haya obstáculos en el camino.

- Rodeo de obstáculos: Si encuentra un obstáculo, el robot rodea el obstáculo siguiendo su contorno.
- Reanudación desde la línea objetivo: Mientras rodea el obstáculo, el robot busca puntos donde pueda regresar a la línea objetivo y continuar su movimiento hacia el objetivo.

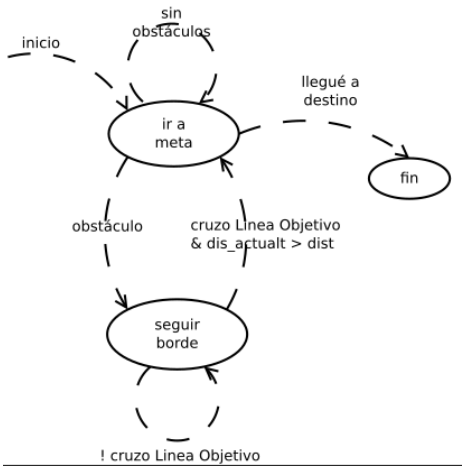

Figura 6.4: Máquina de estados del algoritmo Bug 2

Esta estrategia es más eficiente porque evita rodear completamente los obstáculos, siempre que exista un punto de retorno a la línea objetivo.

Diferencias con Bug1

- `Bug2` no almacena ni utiliza el punto más cercano al objetivo; en su lugar, prioriza volver a la línea objetivo tan pronto como sea posible.
- Es más eficiente en entornos donde el obstáculo no bloquea completamente la línea objetivo.

6.3.4. Entorno virtual de pruebas

Para la simulación se utilizó el entorno virtual CoppeliaSim Edu V4.8.0 rev0 (Coppelia Robotics AG, 2025) (por detalles, referirse al Apéndice B), en el cual se diseñaron robots similares al Butiá 2.0 (InCo, FIng, UDELAR, 2012a) disponibles en el Instituto de Computación de la Facultad de Ingeniería. La Figura 6.5 muestra una imagen del Butiá 2.0.

La escena creada en el simulador CoppeliaSim puede verse en la Figura 6.6.

La Figura 6.7 muestra los modelos de Butiá azul, verde y amarillo, creados en CoppeliaSim.

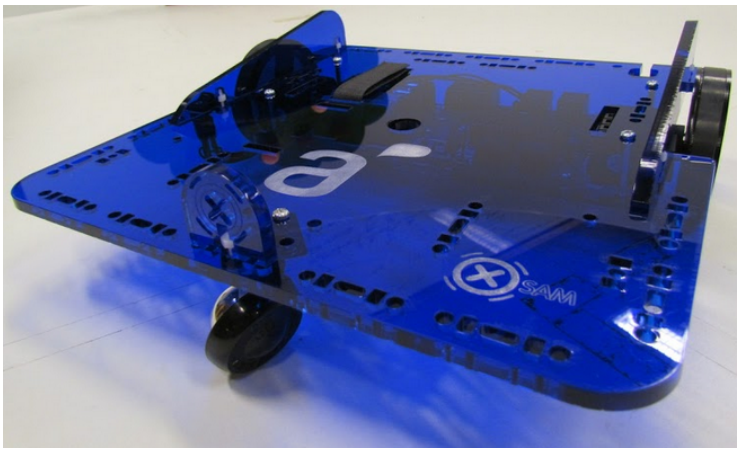

Figura 6.5: Butiá 2.0. Fuente:(InCo, FIng, UDELAR, 2012b)

Otro simulador de uso muy extendido es Gazebo (Open Source Robotics Foundation, 2025a). Debido a la experiencia previa del equipo, se definió utilizar CoppeliaSim.

6.3.5. Sensores de los robots

A cada uno de los robots creados en el simulador se le agregó un sensor GPS y un sensor de proximidad que retorna un valor correspondiente a la distancia al objeto más cercano detectado.

Para comunicar los valores sensados, el GPS del simulador escribe sus datos en el tópico `'Butia[numRobot]/gpsPosition'`. Por su parte, el sensor de proximidad transmite sus datos en el tópico `'Butia[numRobot]/distanceSensorData'`. Para lograr esto, se utilizaron scripts escritos en Lua en CoppeliaSim para cada sensor.

El `[numRobot]` no se corresponde exactamente con el id del robot. Esto se debe a la forma en que CoppeliaSim nombra objetos que se repiten. Para el *ButiaAzul* con $id = 1$ el sensor GPS transmite en el tópico `'Butia/gpsPosition'`. Para el *ButiaVerde* con $id = 2$ el sensor GPS transmite en el tópico `'Butia0/gpsPosition'`. Para el *ButiaAmarillo* con $id = 3$ el sensor GPS transmite en el tópico `'Butia1/gpsPosition'`. Se numera de forma análoga a los tópicos para el sensor de proximidad.

Para conectar la implementación actual con un robot real, es necesario contar con controladores (drivers) para cada sensor (GPS y sensor de proximidad) que escriba en los mismos tópicos, asegurando que se utilicen los mismos tipos de datos en el mismo rango. El resto del sistema podrá utilizar el aparato sensorial de los robots reales sin más.

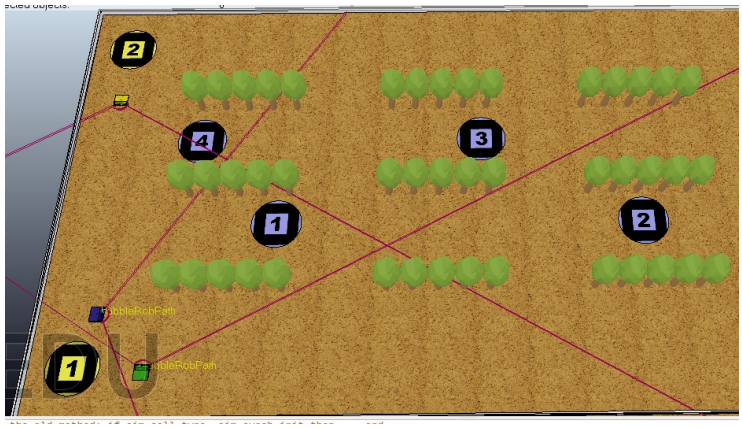

Figura 6.6: Escena simulada en CoppeliaSim de un monte de manzanas. Incluye tres robots, cuatro tareas y dos bases.

6.3.6. Actuadores de los robots

Cada robot cuenta con dos motores, uno para la rueda izquierda y otro para la rueda derecha. El sistema admite comunicarse con un robot real si se ejecutan: `python pybot_server.py` y `butia_ros_server.py` para recibir los valores que activan los motores.

Se utiliza el nodo `pilotoVRep` como interfaz entre los actuadores y el resto del sistema. En los archivos de arranque se pasa el parámetro `usarVRep`, que indica si se conecta con el simulador o con robots reales.

Este nodo recibe la velocidad que se debe enviar a los motores en el tópico `'Butia[numRobot]/cmd_vel'` del tipo Twist. Twist es un tipo de dato que expresa la velocidad dada por sus componentes (velocidad lineal y velocidad angular): `Vector3 linear`, `Vector3 angular`. Luego, para los robots reales, transmite los valores adaptados al controlador, o para CoppeliaSim, envía la velocidad de cada rueda en los tópicos: `'Butia[numRobot]/LeftMotorData'` y `'Butia[numRobot]/RightMotorData'`. Los valores recibidos en esos tópicos serán procesados por los scripts Lua correspondientes a los motores en CoppeliaSim.

6.3.7. Implementación de nodos

Los nodos utilizados para resolver el caso de estudio son los siguientes:

- `NodoSensores`. Se agrega este nodo para recibir la información de cada uno de los sensores y publicar el estado del robot, que incluye la ubicación y la distancia al objeto próximo más cercano (si es detectable). Se suscribe a los siguientes tópicos:
  - `'Butia[numRobot]/distanceSensorData'`. Recibe la distancia al

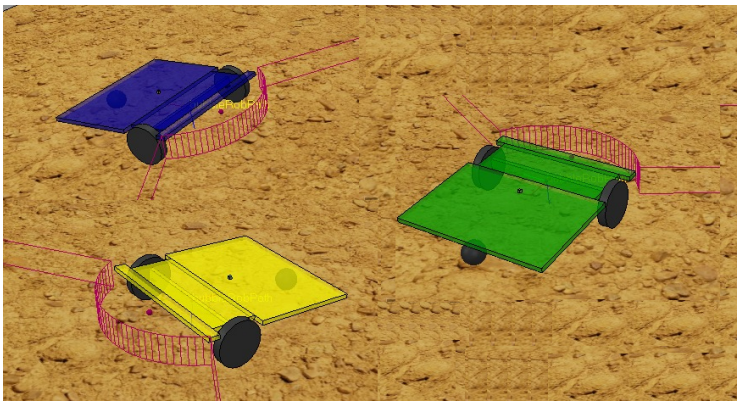

Figura 6.7: Modelos de Butiá 2.0 creados en CoppeliaSim.

objeto más cercano detectado.

- `'Butia[numRobot]/gpsPosition'`. Recibe la ubicación del robot dada por el GPS.
- `'Butia[numRobot]/Cargar'`. Recibe la cantidad a cargar y la suma a la carga actual del robot.
- `'Butia[numRobot]/Descargar'`. Recibe la notificación de descarga y pone la carga actual en cero.
- `'Butia[numRobot]/ActualizarEstado'`. Recibe el nuevo estado del robot y lo actualiza.

Publica información en los siguientes tópicos:

- `'Butia[numRobot]/EstadoSensores'`. Publica el estado interno completo del robot cada cierto tiempo.

- `Piloto`: Este nodo actúa como el controlador cinemático del robot. Su función principal es recibir la velocidad deseada para el chasis (lineal v y angular ω) y realizar la transformación al espacio de los actuadores mediante el modelo de tracción diferencial.

  Se suscribe a los siguientes tópicos:

  - `'Butia[numRobot]/cmd_vel'`: Recibe la velocidad que debe adoptar el robot, separada en sus componentes lineal y angular.
  - `'/finalizar'`: Indica si se debe finalizar la ejecución.

  Cálculo de velocidades: Para el robot Butiá 2.0, se utiliza el modelo cinemático diferencial para obtener las velocidades lineales de las ruedas derecha (v_R) e izquierda (v_L) a partir de los parámetros físicos R (radio de la rueda) y L (distancia entre ruedas):

$$v_R = v + \frac{\omega \cdot L}{2} \quad ; \quad v_L = v - \frac{\omega \cdot L}{2} \tag{6.1}$$

Finalmente, el nodo calcula la velocidad angular necesaria para cada motor ($\dot{\phi}$):

$$\dot{\phi}_R = \frac{v_R}{R} \quad ; \quad \dot{\phi}_L = \frac{v_L}{R} \tag{6.2}$$

Publica información en los siguientes tópicos:

- `'Butia[numRobot]/LeftMotorData'`: Envía el valor calculado $\dot{\phi}_L$ para la rueda izquierda.
- `'Butia[numRobot]/RightMotorData'`: Envía el valor calculado $\dot{\phi}_R$ para la rueda derecha.
- `'Butia[numRobot]/odom'`: Publica la odometría estimada del robot basada en el desplazamiento.

Interfaz con el Hardware Real: Aunque el nodo está configurado por defecto para publicar comandos de velocidad en los tópicos para el simulador *CoppeliaSim*, la arquitectura permite la integración con el robot físico Butiá 2.0. En este escenario, en lugar de publicar mensajes de tipo `Float64`, el nodo utiliza un cliente de servicio para comunicarse con el driver de bajo nivel. Específicamente, se emplea el servicio `butia_set_2_motor_speed`, enviando una cuaterna de valores (a, b, c, d). Aquí, a y c representan la dirección de giro de las ruedas derecha e izquierda respectivamente, mientras que b y d indican la potencia aplicada (mapeada mediante el factor de calibración de 125,98 obtenido experimentalmente) en un rango de 0 a 1023.

Cálculo de Odometría: El nodo implementa un bucle de control a una frecuencia de 10 Hz para la estimación del estado del robot. El cálculo de la odometría se basa en la integración temporal de las velocidades; se determina el diferencial de tiempo (dt) entre ciclos y se calculan los desplazamientos parciales mediante las ecuaciones:

$$\Delta x = (v \cdot \cos(\theta)) \cdot dt \quad ; \quad \Delta y = (v \cdot \sin(\theta)) \cdot dt \quad ; \quad \Delta\theta = \omega \cdot dt \tag{6.3}$$

No obstante, para mitigar el error acumulado propio de la integración numérica (drift), el sistema actualiza de forma síncrona las variables de estado (x, y, θ) utilizando los datos provistos por el sensor GPS y la brújula integrados en el objeto `sensores`. Finalmente, esta información se empaqueta en un mensaje de tipo `nav_msgs/Odometry`, que incluye tanto la pose en el marco de referencia `/odom` como las velocidades lineales y angulares actuales en el marco `base_link`.

- `Controller`. Este nodo fue reescrito a partir de `controller.py` del paquete `alliance` para soportar la creación de nuevos nodos.
- `MotivacionalRetornoBase1`. Este nodo hereda del nodo `Motivacional` y reescribe las funciones `interRobotCallback` y `sensoryFeedback`. Los nodos se nombran `MOTIV_Rob[idRobot]_TarRB1`.
- `MotivacionalRetornoBase2`. Idem `MotivacionalRetornoBase1`. Los nodos se nombran `MOTIV_Rob[idRobot]_TarRB2`.
- `MotivacionalDescargar`. Este nodo hereda del nodo `Motivacional` y reescribe las funciones `interRobotCallback`, `sensoryFeedback`, `impatience`, `impatienceReset` y `acquiescence`. Los nodos se nombran `MOTIV_Rob[idRobot]_TarDESCARGA`.
- `MotivacionalCargar`. Idem `MotivacionalDescargar`. Los nodos se nombran `MOTIV_Rob[idRobot]_TarCARGA_[idTarea]`.
- `MotivacionalRecoleccion`. Este nodo hereda del nodo `Motivacional` y reescribe las funciones `interRobotCallback` y `sensoryFeedback`. Los nodos se nombran `MOTIV_Rob[idRobot]_TarRECOLECC_[idTarea]`.

6.3.8. Manejo de dependencias entre tareas

Una limitación que presenta esta implementación de Alliance es la dificultad para lidiar con tareas secuenciales, es decir, aquellas que deben cumplirse en un determinado orden; por ejemplo, la $Tarea\_2$ no puede comenzar hasta haber finalizado la $Tarea\_1$.

Si ese tipo de tareas compitieran de igual a igual, podría darse el caso de que se activara una tarea antes de que se haya finalizado una predecesora, lo que daría lugar a inconsistencias o mal funcionamiento.

En este caso de estudio se presenta la situación de tareas secuenciales, a saber:

- No debería tomar una tarea de $Retorno\_a\_Base$ un robot vacío.
- No debería tomar una tarea de $Carga$ un robot que no haya llegado a un punto de recolección.
- No debería tomar una tarea de $Descarga$ un robot que no haya llegado a una base.
- No debería tomar una tarea de Recolección un robot totalmente cargado.

Como no fue posible resolver este problema sin pasarle información adicional a la función `sensoryFeedback`, para resolver esta casuística, se implementó una máquina de estados.

Se extendió y se realizaron modificaciones en la función `interRobotCallback` del nodo `Motivacional` para mantener y actualizar el estado del robot.

La función `sensoryFeedback` habilitará al comportamiento motivacional para competir por una tarea o no, dependiendo del estado actual del robot.

La máquina de estados que mantiene cada robot se puede ver en la Figura 6.8. Los estados definidos son los siguientes: `INICIAL`, `RECOLECTANDO`, `EN_DESTINO_DE_CARGA`, `CARGANDO`, `CARGA_FINALIZADA`, `RETORNANDO_A_BASE`, `EN_DESTINO_DE_DESCARGA`, `DESCARGANDO`, `DESCARGA_FINALIZADA`.

6.4. Conclusiones de la Implementación del Caso de Estudio

La implementación del caso de estudio "Granja de Manzanas" permitió validar la modularidad y extensibilidad del motor de Alliance desarrollado en el capítulo anterior. El paquete `cocoro2` demostró la capacidad de la arquitectura para ser adaptada a un problema concreto, mediante la herencia y la implementación de comportamientos de bajo nivel específicos (como `NavegarDeFormaSegura` y `Esquivar`).

Un gran desafío técnico de esta implementación fue la gestión de dependencias secuenciales entre las tareas, una limitación reconocida de la arquitectura Alliance. La solución adoptada, detallada en la Subsección 6.3.8, consistió en una máquina de estados finitos que regula la elegibilidad de las tareas a través de la función `sensoryFeedback`.

Si bien este enfoque resolvió funcionalmente la restricción, permitiendo que un robot solo compita por tareas válidas para su estado actual (por ejemplo, que no compita por "Descarga" si no está en la base), introdujo una capa significativa de complejidad en la coordinación. La integridad del estado se convirtió en un punto crítico y frágil del sistema.

Esta fragilidad se manifestó en forma de condiciones de carrera inherentes al entorno asíncrono y *multihilo* de ROS 2. Se abordaron estas condiciones de carrera aplicando ciertos retrasos en lugares específicos.

Si bien no se puede asegurar que el sistema quedó 100 % libre de bloqueos, dado que en las ejecuciones realizadas para las pruebas y la experimentación no se reportaron fallas ni bloqueos, se puede argumentar que el caso de estudio de la "Granja de Manzanas" validó con éxito la implementación de la arquitectura para un problema de dominio específico. Sin embargo, también expuso las dificultades prácticas de la gestión del estado en un sistema distribuido que carece de garantías transaccionales.

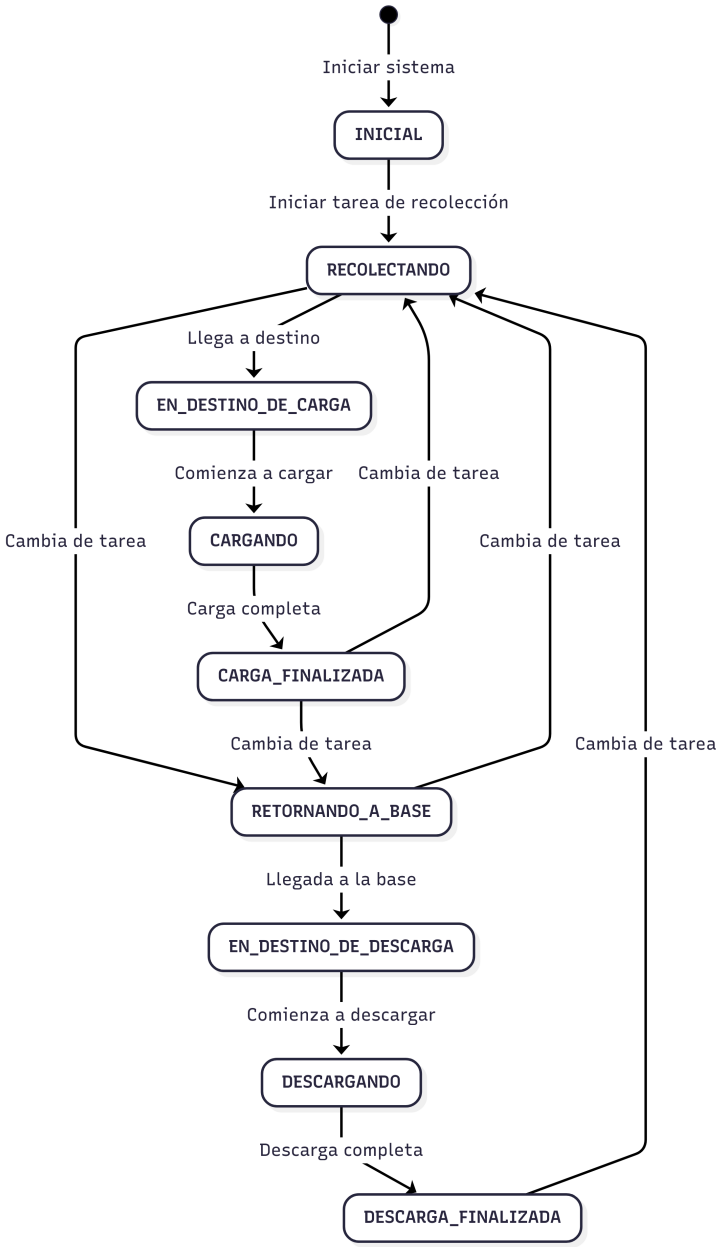

Figura 6.8: Máquina de estados del robot

Capítulo 7

Experimentación y Validación

En este capítulo se presentan los resultados de los experimentos realizados para validar empíricamente la implementación del motor de Alliance, desarrollado sobre ROS 2.

7.1. Introducción

Se diseñaron conjuntos de pruebas orientadas a verificar el correcto funcionamiento de la asignación de tareas, la distribución de carga entre robots y la respuesta del sistema bajo cargas de trabajo y condiciones de equipo variables, en particular, en condiciones de alta demanda.

Se desea mostrar como escala el rendimiento del sistema al añadir más robots o más tareas y si el sistema es robusto ante fallos de comunicación y cambios dinámicos en la composición del equipo, cumpliendo así las premisas de la arquitectura Alliance.

Los experimentos se ejecutaron utilizando cierta cantidad de tareas iguales en tiempo de finalización y cuya resolución activa un único conjunto de comportamientos compuesto por los comportamientos de prueba genéricos `DefaultBehavior1` y `DefaultBehavior2`, explicados en la Subsección 7.2.1. Estos comportamientos simulan la ejecución de una tarea consumiendo un intervalo de tiempo predefinido, lo que permite eliminar las variables de navegación (por ejemplo, la evasión de obstáculos).

7.2. Contexto de ejecución

Para obtener los tiempos, se utilizó el timestamp que muestra el mecanismo de logs utilizado en ROS 2. Entonces el tiempo de realización de una tarea se mide desde que el nodo `GeneradorTareas` envía al log que hay una tarea

disponible, inmediatamente después de publicarla en el tópico correspondiente, hasta que el nodo `Motivacional` que la finaliza, indica en el log que finalizó dicha tarea inmediatamente después de enviar el mensaje en el tópico correspondiente.

7.2.1. Comportamientos de prueba DefaultBehavior1 y DefaultBehavior2

Para el *Motor de Alliance* se implementaron a modo de prueba las clases `DefaultBehavior1` y `DefaultBehavior2` que heredan de `Comportamiento` para poder testear el correcto funcionamiento del motor. Dichos comportamientos corren en hilos separados y operan en conjunto para resolver una tarea, es decir, es el conjunto de comportamientos que se activa para resolver una tarea.

`DefaultBehavior1` es el comportamiento que avanza hacia la realización de una tarea mientras que `DefaultBehavior2` se supone un comportamiento auxiliar que se activa bajo determinadas condiciones.

Los objetivos perseguidos al implementar estos comportamientos son:

- Para validar que un comportamiento toma el control y los demás no, se debe lograr que todos los comportamientos se activen más de una vez antes de finalizar una tarea.
- Para validar que los comportamientos quedan pausados y se reactivan, se debe lograr que se pase el control de un comportamiento a otro, e incluso que haya momentos en los que no se permite activar a ninguno de ellos.
- Para validar que se termina una tarea y se liberan los recursos, se debe lograr que uno de los comportamientos detecte que se finalizó la tarea y realice las operaciones necesarias.
- Se debe validar que el sistema cambia de tarea y activa diferente conjunto de comportamientos sin que se bloqueen. Para esto se requiere que cada conjunto se active al menos un par de veces antes de finalizar.

Para el contexto de ejecución se definen los siguientes valores constantes:

- `TIEMPO_TERMINAR_TAREA_DEFAULT_1` Es el tiempo que se espera para que el `DefaultBehavior1` finalice una tarea. Solamente `DefaultBehavior1` puede finalizar una tarea. Para la experimentación este valor se puso en 5 segundos.
- `REFRESH_RATE` Es la tasa de ejecución de la función `run()` del nodo `Comportamiento`. También es la tasa de invocación de la función `arbitrar()` de la clase `Arbitro`. Es el nodo `Executer` el que define un *timer* y en cada callback (cada `REFRESH_RATE` segundos) invoca a `arbitrar()`.

Para lograr los objetivos planteados anteriormente, se eligió una combinación particular y arbitraria de valores que se detallan a continuación:

- Se define que la función `takeControl()` de la clase `Comportamiento` incremente un contador de iteraciones.
- Se definió que la instancia de `DefaultBehavior1` tomara el control si el resto de dividir el número de iteración entre 50, es mayor o igual a 10 y menor que 40. Esto significa que en estas condiciones la función `takeControl()` retornará `true`.
- Se definió que la instancia de `DefaultBehavior2` tomara el control si el resto de dividir el número de iteración entre 50, es mayor o igual a 40. Análogamente, la función `takeControl()` retornará `true` en estas condiciones.
- Se definió que la instancia de `DefaultBehavior1` sea la que detecta la finalización de la tarea.
- Se definió que para finalizar una tarea, el comportamiento `DefaultBehavior1` debe sumar un tiempo de actividad superior a lo que demoran 30 iteraciones, para asegurar que `DefaultBehavior1` se active al menos en 2 períodos diferentes.

De esta forma, cada 50 iteraciones que se incrementan en cada llamada a la función `takeControl()` (invocada por la función `arbitrar()` de la clase `Arbitro`), en las primeras 9 iteraciones no se debe activar ningún comportamiento, entre las iteraciones 10 y 39 se debe activar `DefaultBehavior1` pero no `DefaultBehavior2` y entre las iteraciones 40 y 50 se debe activar `DefaultBehavior2` y no `DefaultBehavior1`. La instancia de `DefaultBehavior1` es la que reconoce el fin de la tarea si se ejecutó un tiempo acumulado mayor o igual a `TIEMPO_TERMINAR_TAREA_DEFAULT_1` recibido como parámetro.

Puede verse el diagrama de colaboración entre el nodo `Executer`, el `Arbitro`, y los comportamientos `DefaultBehavior1` y `DefaultBehavior2` en la Figura 7.1.

Todos los valores anteriores son arbitrarios y se eligieron para facilitar la experimentación.

7.2.2. Limitaciones y parámetros

Los nodos de ROS 2 no son entidades "livianas" similares a threads u objetos, que pueden crearse y destruirse rápidamente para gestionar una tarea. Al contrario, un nodo es una entidad "pesada" (heavyweight).

Al crear un nuevo nodo, internamente ocurren varias tareas complejas y lentas (considerando tiempos computacionales):

- **Inicialización de RCL:** Se inicializa la biblioteca cliente de ROS, (RCL por su sigla en inglés, ROS Client Library).
- **Participante de red:** El nodo debe crear un participante en la red gestionada por DDS (Data Distribution Service).

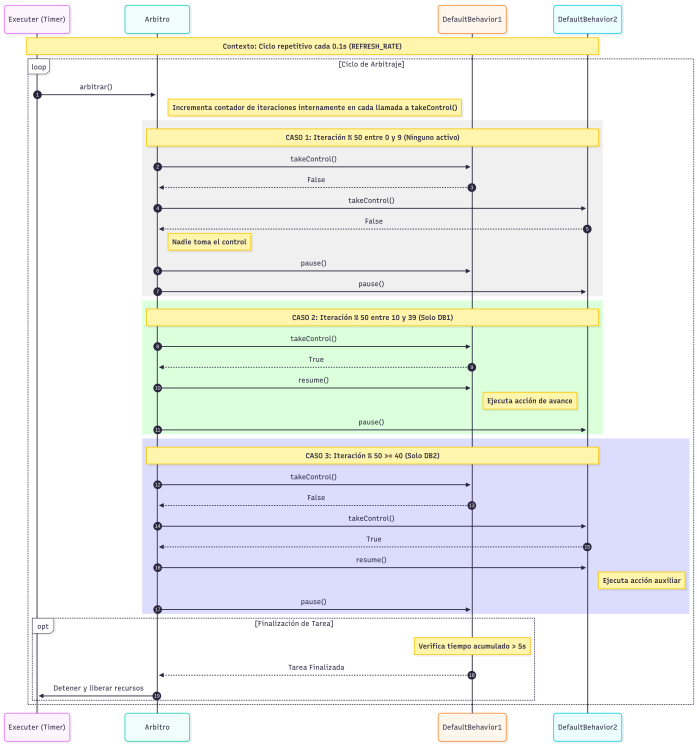

Figura 7.1: Diagrama de colaboración DefaultBehavior1 y DefaultBehavior2

- **Descubrimiento de Red:** El nuevo nodo debe anunciar su presencia a todos los demás nodos del sistema. A su vez, todos los demás nodos deben descubrir a este nuevo participante.
- **Creación de Tópicos/Servicios:** Se configuran todos sus publicadores, suscriptores, etc., lo cual implica más negociación en la red.

El nodo `Controller` debe crear dos nodos por cada tarea que aparece. Por ejemplo, cuando se publican 10 tareas a la vez, se le está pidiendo al `Controller` que inicie 20 nuevos nodos (10 `Motivacional` y 10 `Executer`). Esto se multiplica por la cantidad de robots en el sistema y se vuelve especialmente pesado si se corre una simulación en una sola PC. El sistema se ve inundado por una "tormenta de descubrimiento" en la red, donde todos los nodos intentan hablar con todos a la vez. Al ejecutar las simulaciones en una única PC, el sistema se ve limitado principalmente por la memoria y los procesadores disponibles.

Cuando el `MultiThreadedExecutor` que usa `Controller` se satura intentando gestionar la creación de nodos, se ralentiza todo el sistema.

Se determinó experimentalmente que para tres o cuatro robots el sistema comienza a degradar su performance con más de 10 tareas concurrentes, porque

el executor se satura cuando se ejecuta en una única PC.

Por las razones recién expuestas se definió una constante `CANT_MAX_TAREAS_EVITA_SOBRECARGA` y se programó la aparición de tareas de forma que no supere esa cantidad de tareas concurrentes. La estrategia utilizada para crear más tareas fue que cada `CANT_MAX_TAREAS_EVITA_SOBRECARGA` tareas, se pausa la generación de nuevas tareas por un tiempo de `ESPERA_EVITAR_SOBRECARGA` segundos, obtenido experimentalmente, para darle tiempo al nodo `Controller` para que finalice la creación de los nodos pendientes. De esta forma se realizaron todos los experimentos.

Se entendió que esta forma de realizar los experimentos no los invalida, dado que por un lado se encontró la limitación de ejecución en paralelo para una única PC, y el resto de experimentos miden otros factores, por ejemplo la robustez de agregar/quitar elementos al conjunto de robots, o la capacidad del sistema para finalizar misiones con muchas tareas (sin sobrepasar el límite de ejecución en paralelo).

7.3. Metodología y Métricas

La metodología consiste en ejecutar un conjunto de escenarios predefinidos, variando el número de robots y tareas, y registrar el rendimiento del sistema utilizando tres métricas clave:

- **Tasa de Finalización de Misión (F_{ratio})**
  - **Definición:** El porcentaje de tareas publicadas que fueron completadas exitosamente por el equipo de robots.
  - **Fórmula:**
$$F_{ratio} = \left(\frac{\text{Tareas Completadas}}{\text{Tareas Totales}} \right) \times 100 \tag{7.1}$$
  - **Propósito:** Es la métrica principal de **robustez**. Un valor inferior al 100 % indica un fallo crítico en el sistema. Se espera un 100 % en todos los experimentos.
- **Tiempo Total de Misión (T_{mision})**
  - **Definición:** El tiempo total transcurrido desde la publicación de la primera tarea hasta que el sistema registra la finalización de la última tarea.
  - **Propósito:** Mide la **eficiencia global** y la **escalabilidad** del equipo.
- **Tiempo de Resolución de Tarea (T_{tarea_j})**
  - **Definición:** El tiempo transcurrido para una tarea individual j, desde que aparece como disponible (en este contexto, es desde que se publica en el tópico `/tareas`) hasta que un robot avisa que la ha finalizado.

- **Propósito:** Mide la **eficiencia a nivel de tarea** y la latencia del sistema. Para el análisis, se utilizarán sus valores estadísticos:
  - **Tiempo Promedio de Resolución ($\mu(T_{tarea})$):** Indica qué tan rápido, en promedio, se resuelve una tarea.
  - **Desviación Estándar del Tiempo de Resolución ($\sigma(T_{tarea})$):** Mide la consistencia del rendimiento.

7.4. Diseño de Experimentos

Se diseñaron tres conjuntos de experimentos para evaluar el sistema.

7.4.1. Experimento 1: Escalabilidad del Sistema (Carga Fija)

- **Objetivo:** Analizar la escalabilidad horizontal del sistema, es decir, cómo impacta añadir más robots a una carga de trabajo constante.
- **Variable independiente:** Número de robots ($N_{robots} \in \{1, 2, 3, ..., 10\}$).
- **Constantes:** Número de tareas ($N_{tareas} = 20$).

Para el experimento 1, se utilizaron los valores:

- `CANT_MAX_TAREAS_EVITA_SOBRECARGA` $= 100$.
- `ESPERA_EVITAR_SOBRECARGA` $= 1$ *segundos*.
- `TIEMPO_TERMINAR_TAREA_DEFAULT_1` $= 5$ *segundos*.
- `REFRESH_RATE` $= 0{,}1$ *segundos*.

De esta forma, se lanzaron las 20 tareas al inicio de la ejecución con una diferencia aleatoria de hasta un segundo entre una y la siguiente.

Análisis de los Resultados (Experimento 1)

Se muestran como ejemplo los tiempos obtenidos de los logs para los dos primeros casos (1 Robot y 2 Robots) en: Tabla 7.1 y Tabla 7.2 y por claridad se excluyen del informe el resto de las tablas (3 Robots a 10 Robots) con valores detallados ya que esos datos se presentan agrupados en las tablas subsiguientes.

Los resultados generales del Experimento 1, presentados en la Tabla 7.3, permiten evaluar la escalabilidad horizontal del sistema.

El objetivo de esta prueba era medir el impacto en la eficiencia global al incrementar el número de agentes (robots) frente a una carga de trabajo constante de 20 tareas.

Tabla 7.1: Detalle de tiempos de ejecución por tarea individual (Experimento 1, para 1 Robot y 20 Tareas).

ID Tarea	Hora de Inicio	Hora de Fin	Tiempo (s)
1	*00:45:29.467216*	*00:45:51.824686*	*22.357*
2	*00:45:30.269367*	*00:46:04.474448*	*34.205*
3	*00:45:30.886632*	*00:46:14.668241*	*43.782*
4	*00:45:31.233304*	*00:46:27.902181*	*56.669*
5	*00:45:31.923591*	*00:46:42.196196*	*70.273*
6	*00:45:32.894056*	*00:46:47.702708*	*74.809*
7	*00:45:33.495283*	*00:46:46.293506*	*72.798*
8	*00:45:34.044094*	*00:47:00.843640*	*86.800*
9	*00:45:34.940228*	*00:48:49.534029*	*194.594*
10	*00:45:35.612770*	*00:47:13.569341*	*97.957*
11	*00:45:36.495687*	*00:47:18.208710*	*101.713*
12	*00:45:37.387627*	*00:47:29.828412*	*112.441*
13	*00:45:37.820273*	*00:48:48.266089*	*190.446*
14	*00:45:38.570051*	*00:47:43.517159*	*124.947*
15	*00:45:38.785746*	*00:47:54.563948*	*135.778*
16	*00:45:39.657645*	*00:48:05.043383*	*145.386*
17	*00:45:40.226653*	*00:48:16.740930*	*156.514*
18	*00:45:41.111456*	*00:48:22.383675*	*161.272*
19	*00:45:41.702885*	*00:48:29.945336*	*168.242*
20	*00:45:42.366067*	*00:48:41.456304*	*179.090*

Robustez del Sistema

La métrica F_{ratio} (Tasa de Finalización de Misión) se mantuvo en 100.0 % en la totalidad de las pruebas. Este dato valida que, independientemente del número de agentes, la arquitectura implementada es robusta y garantiza la finalización de todas las tareas asignadas.

Eficiencia y Escalabilidad

El Tiempo Total de Misión (T_{mision}), visualizado en la Figura Figura 7.2, demuestra una clara mejora en el rendimiento al añadir agentes, hasta alcanzar un punto de saturación. Al pasar de uno a dos robots, se observó la mejora más drástica en el rendimiento, reduciendo el T_{mision} en 52.4 segundos (una optimización del 26.2 %). Esto valida el beneficio fundamental de la paralelización. Al añadir un tercer robot, la mejora continuó (6.2 segundos de reducción), aunque

Tabla 7.2: Detalle de tiempos de ejecución por tarea individual (Experimento 1, para 2 Robots y 20 Tareas).

ID Tarea	Hora de Inicio	Hora de Fin	Tiempo (s)
1	*00:58:13.826669*	*00:58:33.796662*	*19.970*
2	*00:58:13.963607*	*00:58:39.747560*	*25.784*
3	*00:58:14.467478*	*00:58:47.569469*	*33.102*
4	*00:58:14.605082*	*00:58:52.681929*	*38.077*
5	*00:58:15.534228*	*00:59:00.111278*	*44.577*
6	*00:58:15.995265*	*00:59:05.241462*	*49.246*
7	*00:58:16.171510*	*00:59:18.442650*	*62.271*
8	*00:58:17.101586*	*00:59:22.145095*	*65.044*
9	*00:58:17.960811*	*00:59:32.187662*	*74.227*
10	*00:58:18.942412*	*00:59:32.641861*	*73.699*
11	*00:58:19.798072*	*00:59:43.077318*	*83.279*
12	*00:58:20.366699*	*00:59:48.307659*	*87.941*
13	*00:58:20.705000*	*00:59:55.040271*	*94.335*
14	*00:58:21.374381*	*00:59:59.988430*	*98.614*
15	*00:58:21.907110*	*01:00:08.486380*	*106.579*
16	*00:58:22.455108*	*01:00:13.718066*	*111.263*
17	*00:58:23.339434*	*01:00:24.190712*	*120.851*
18	*00:58:23.576326*	*01:00:27.647220*	*124.071*
19	*00:58:24.259438*	*01:00:34.837771*	*130.578*
20	*00:58:25.212768*	*01:00:41.488720*	*136.276*

con menor magnitud. Finalmente, al incorporar un cuarto robot, la mejora fue marginal (2.8 segundos de reducción), alcanzando el punto de mejor rendimiento para esta carga de trabajo.

Saturación y Rendimientos Marginales Decrecientes

El hallazgo más crítico se observa al pasar de cuatro a cinco robots. El T_{mision} no solo dejó de mejorar, sino que aumentó de 138.6 s a 143.2 s. Esto demuestra que el sistema alcanzó su punto de saturación.

Son muchas las variables a tener en cuenta para explicar este comportamiento, a saber:

- La forma en que se asigna el comportamiento `DefaultBehavior1`, esperando una cierta cantidad de iteraciones. Si se asignara de otra forma, no se observaría este resultado, podría asignarse siempre, o iteración por

Tabla 7.3: Resultados del Experimento 1: Escalabilidad del Sistema para 20 tareas.

N_{robots}	F_{ratio} (%)	T_{mision} (seg)	$\mu(T_{tarea})$ (seg)	$\sigma(T_{tarea})$ (seg)
1	*100.0*	*200.067*	*111.504*	*53.456*
2	*100.0*	*147.662*	*78.989*	*36.228*
3	*100.0*	*141.485*	*73.659*	*34.358*
4	*100.0*	*138.640*	*72.506*	*33.650*
5	*100.0*	*143.250*	*73.656*	*35.511*
6	*100.0*	*141.071*	*73.564*	*34.751*
7	*100.0*	*139.145*	*74.450*	*35.219*
8	*100.0*	*141.865*	*74.778*	*35.404*
9	*100.0*	*143.750*	*74.843*	*35.008*
10	*100.0*	*140.449*	*75.153*	*35.109*

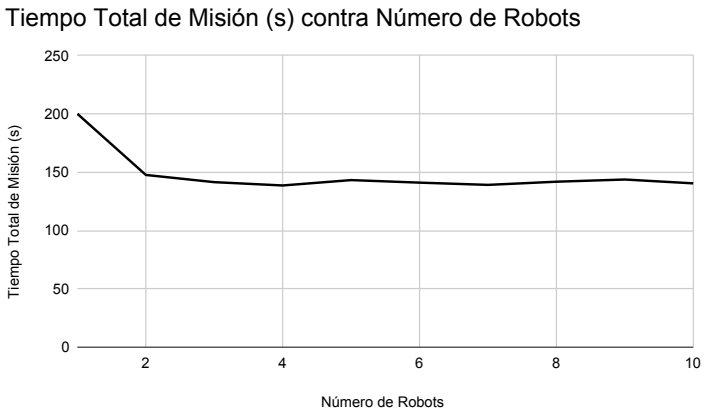

Figura 7.2: Impacto del número de robots en el Tiempo Total de Misión (T_{mision}) para una carga fija de 20 tareas.

medio, etc. En un escenario más real, la tarea sería asignable salvo que la hubiera tomado otro robot antes, o que el propio robot haya seleccionado otra tarea.

- La definición de que haya iteraciones en las que `DefaultBehavior1` no sea aplicable. Esto afecta porque en el momento en que `DefaultBehavior1` no es aplicable, la motivación vuelve a cero y debe reiniciarse el conteo hasta superar el umbral.

- Los valores asignados a ciertas constantes del sistema. Afectan, por ejemplo, el tiempo activo de una tarea para darla por concluida, los refresh rate de los timers y ciertos delays utilizados para desincronizar los nodos.

- Los valores asignados a los parámetros de Alliance. Afecta el refresh rate del ciclo del cálculo de la motivación, así como la frecuencia de envío de mensajes por broadcast.
- Los tiempos de espera añadidos para resolver condiciones de carrera.
- Otras particularidades de la implementación. Posibles detalles de implementación que no fueron analizados.
- La ejecución en una única PC. Porque se satura el uso de memoria y procesador.
- El *overhead* computacional y de red al crear muchos nodos, causado principalmente por la ejecución en una única PC.

El *overhead* computacional y de red generado por la creación dinámica y coordinación de un número excesivo de nodos (un total de 200 nodos para 5 robots y 20 tareas, más los 5 nodos controladores), sumado a las variables mencionadas anteriormente, supera el beneficio de paralelización que el quinto robot podría haber aportado. Como se observa en la Figura 7.2, de 5 a 10 robots, el rendimiento se estabiliza en una meseta, con fluctuaciones menores (entre 139 s y 144 s), donde el costo de la coordinación anula cualquier ganancia marginal.

Si bien un rendimiento marginal decreciente era esperable al agregar robots, se estima que la causa principal de la saturación es la capacidad de cómputo de una única PC que ejecuta todo el entorno, simulando el paralelismo que tendría un escenario más real en el que los procesos de cada robot se ejecutarían en máquinas separadas.

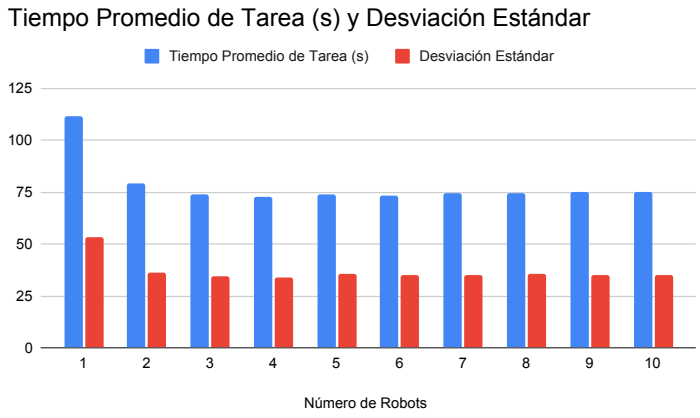

Figura 7.3: Impacto del número de robots en el Tiempo Promedio de Resolución de Tarea ($\mu(T_{tarea})$).

Eficiencia y Consistencia a Nivel de Tarea (μ y σ)

El Tiempo Promedio de Resolución de Tarea ($\mu(T_{tarea})$) corrobora este análisis, como puede verse en la Figura 7.3. El tiempo que una tarea espera desde su publicación hasta su finalización se reduce significativamente, pasando de 111.5 s (1 robot) a un mínimo de 72.5 s (4 robots). Esto indica una resolución de tareas considerablemente más rápida. A partir de 5 robots, el tiempo promedio vuelve a incrementarse ligeramente, reflejando el *overhead* del sistema. Asimismo, la desviación estándar ($\sigma(T_{tarea})$) se reduce abruptamente de 53.5 s a 33.7 s (con 4 robots). Esto demuestra que el sistema no solo es más rápido con más agentes, sino también significativamente más consistente y predecible en su operación.

Conclusión (Experimento 1)

El motor de Alliance implementado demuestra una escalabilidad horizontal exitosa; sin embargo no se exhiben claros rendimientos marginales decrecientes como era de esperar (salvo hasta los 4 robots) y se llega rápidamente al caso de mejor rendimiento. El caso de mejor rendimiento para esta carga de trabajo (20 tareas), bajo la arquitectura actual, se encuentra en 4 robots. Más allá de este punto, el costo de la creación dinámica de nodos y la sobrecarga de coordinación saturan el sistema y degradan el rendimiento general.

Si bien llama la atención por qué se demora un tiempo de alrededor de 20 segundos en resolver la primera tarea, esto se explica por la forma en que fue diseñado el experimento. Cuando una tarea queda disponible (se envió el mensaje al tópico correspondiente), se publica inmediatamente en el log y se toma el tiempo inicial. Luego, el overhead de creación de nodos es quizás excesivo. Una vez que los nodos *Motivacional* y *Executer* están operativos, comienza a aumentar la motivación a una tasa ρ_i de entre 0,2 y 0,4 segundos. Luego, la motivación debe llegar a 100 para que el *Motivacional* le avise al *Executer* que habilite el conjunto de comportamientos y comience a arbitrar. Recién ahí operan las iteraciones de los comportamientos. Los 5 segundos que se esperan para finalizar la tarea son de actividad del comportamiento, no en total; entonces, subir la motivación hasta 100 ocurre 2 veces por la forma en que están dispuestos.

Otro punto que llama la atención es por qué los tiempos para resolver una misma tarea se incrementan a medida que avanza la misión. Por los valores utilizados en las constantes para este experimento, se inician las 20 tareas con una diferencia mínima de tiempo entre ellas, a lo sumo un segundo entre una tarea y la siguiente (lapso asignado aleatoriamente). Luego, se crean los nodos que las resuelven y, a medida que éstos aparecen, se van resolviendo las tareas. El tiempo de resolución de cada tarea es similar, pero al comenzar a resolverse cada vez más tarde, el tiempo total que la tarea permanece sin resolver es cada vez mayor.

Para casos generales, podría esperarse que si el tiempo para resolver una misión por un robot es t, el tiempo esperado para la misma misión por dos robots es $t/2$ aproximadamente, sabiendo de antemano que no es una relación lineal. Para este caso, el overhead inicial es muy grande, de los 200 segundos

que le lleva a un robot terminar la misión, hay aproximadamente 100 segundos en total de overhead y de los otros 100 segundos de trabajo efectivo se observa que pasan a la mitad con 2 robots, o sea que el tiempo total pasa de 200 a 150 segundos aproximadamente.

Lo que llama la atención es que el sistema se degrada muy rápido y se atribuye al uso de memoria y procesador en una única PC que se va multiplicando con cada robot.

7.4.2. Experimento 2: Gestión de Carga de Trabajo (Robots Fijos)

- **Objetivo:** Analizar la escalabilidad vertical, es decir, cómo responde el sistema ante un incremento en la carga de trabajo.
- **Variable Independiente:** Número de tareas ($N_{tareas} \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$).
- **Constantes:** Número de robots ($N_{robots} = 3$).

Para el experimento 2, se utilizaron los valores:

- `CANT_MAX_TAREAS_EVITA_SOBRECARGA` $= 9$.
- `ESPERA_EVITAR_SOBRECARGA` $= 80$ *segundos*.
- `TIEMPO_TERMINAR_TAREA_DEFAULT_1` $= 5$ *segundos*.
- `REFRESH_RATE` $= 0{,}1$ *segundos*.

Análisis de los Resultados (Experimento 2) Gestión de Carga de Trabajo

Este experimento se diseñó para evaluar la **escalabilidad vertical** del sistema, analizando la respuesta de un equipo de composición fija (3 robots) ante un incremento progresivo de la carga de trabajo (de 10 a 100 tareas). Los resultados se presentan en la Tabla 7.4.

Robustez del Sistema (F_{ratio}) La Tasa de Finalización de Misión (F_{ratio}) se mantuvo en un 100,0 % en todas las configuraciones probadas. Este resultado es fundamental, ya que valida la robustez de la implementación. El sistema completó exitosamente la totalidad de las tareas asignadas, independientemente de la carga de trabajo, demostrando que el mecanismo de coordinación y la gestión dinámica de nodos no presentan fallos catastróficos ni pérdida de tareas al aumentar la demanda (teniendo en cuenta que se ejecuta en las condiciones descritas en la Sección 7.2).

Tabla 7.4: Resultados del Experimento 2: Gestión de Carga de Trabajo para 3 Robots.

N_{tareas}	F_{ratio} **(%)**	T_{mision} **(seg)**	$\mu(T_{tarea})$ **(seg)**	$\sigma(T_{tarea})$ **(seg)**
10	*100.0*	*102.226*	*42.492*	*18.594*
20	*100.0*	*197.044*	*40.465*	*15.763*
30	*100.0*	*287.328*	*40.166*	*15.185*
40	*100.0*	*379.513*	*41.583*	*14.830*
50	*100.0*	*470.327*	*42.695*	*16.110*
60	*100.0*	*564.600*	*42.785*	*15.398*
70	*100.0*	*656.471*	*43.135*	*15.579*
80	*100.0*	*750.893*	*42.903*	*15.618*
90	*100.0*	*837.581*	*42.857*	*15.493*
100	*100.0*	*952.614*	*43.130*	*15.732*

Escalabilidad y Predictibilidad del Rendimiento (T_{mision}) El Tiempo Total de Misión (T_{mision}), ilustrado en la Figura 7.4, exhibe un **crecimiento marcadamente lineal** en función del número de tareas (N_{tareas}). Este comportamiento lineal es el mejor resultado que se puede esperar de un sistema con una capacidad de procesamiento fija (un equipo de 3 robots).

Esta linealidad confirma que el *throughput* del sistema (la tasa de finalización de tareas) se mantiene aproximadamente constante. El sistema demuestra ser predecible: no se observan cuellos de botella ni una degradación exponencial del rendimiento. El incremento en T_{mision} es una consecuencia directa y proporcional del aumento de la carga de trabajo, no de una pérdida de eficiencia operativa. Nuevamente, se debe tener en cuenta que el experimento se ejecuta en las condiciones descritas en la Sección 7.2. En otras condiciones, se podría esperar una degradación del rendimiento al aumentar significativamente el número de tareas.

Eficiencia y Consistencia a Nivel de Tarea (μ y σ) El hallazgo más significativo de este experimento reside en la **excepcional estabilidad** del Tiempo Promedio de Resolución ($\mu(T_{tarea})$) y su Desviación Estándar ($\sigma(T_{tarea})$), visualizada en la Figura 7.5.

1. **Tiempo Promedio ($\mu(T_{tarea})$):** A pesar de que la carga de trabajo total se multiplicó por diez, el tiempo promedio de resolución de tarea permaneció notablemente constante, fluctuando en un rango acotado entre 40,1 y 43,1 segundos.

2. **Desviación Estándar ($\sigma(T_{tarea})$):** De forma análoga, la desviación estándar (excluyendo la variabilidad esperada de la muestra pequeña de $N = 10$) se estabilizó en un valor constante de $\approx 15{,}5 \pm 0{,}9$ segundos.

Esta estabilidad demuestra que la eficiencia del sistema a nivel de tarea individual **no se degrada con el aumento de la carga**. El tiempo promedio

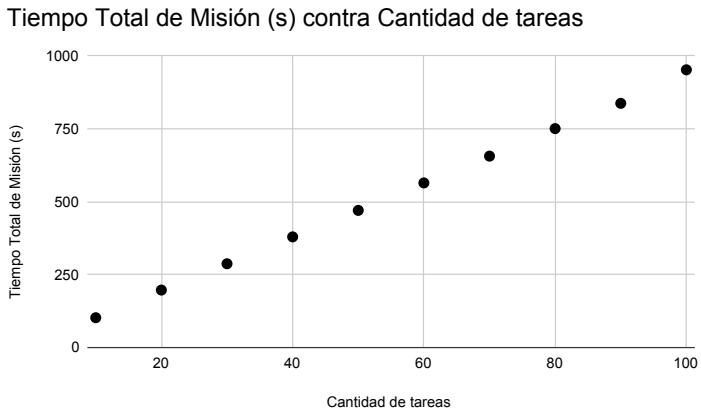

Figura 7.4: Impacto del número de tareas en el Tiempo Total de Misión (T_{mision}) para un equipo fijo de 3 robots.

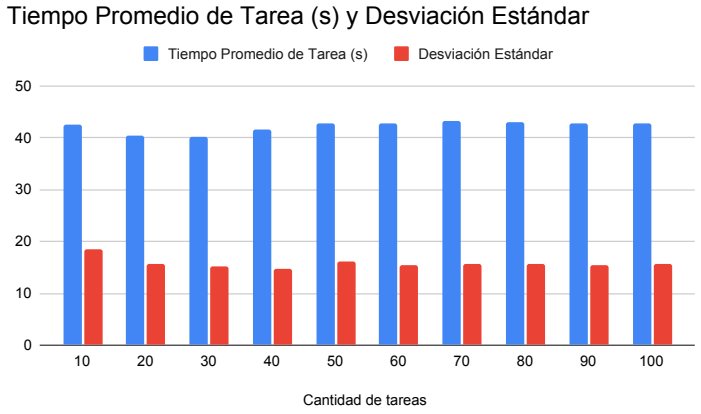

Figura 7.5: Impacto del número de tareas en el Tiempo Promedio de Resolución ($\mu(T_{tarea})$).

que transcurre desde que una tarea es publicada hasta que es completada por un robot es independiente de la duración total de la misión. Esto indica que la arquitectura maneja eficientemente la concurrencia de los nodos de tarea y que el equipo de 3 robots opera dentro de su capacidad nominal sin alcanzar un punto de saturación que degrade la latencia del servicio.

Al realizar el experimento en las condiciones descritas en la Sección 7.2, se evita, por un lado, la saturación de la gestión de recursos en la PC; además, se evita la espera excesiva de tareas sin resolver. Si este experimento se reali-

zara, por ejemplo, disponibilizando todas las tareas al inicio de la misión, sería esperable un aumento en el tiempo promedio de resolución.

Existe una diferencia notable entre la Tabla 7.3 y la Tabla 7.4, en particular para el caso que es similar, línea 3 de la Tabla 7.3 (3 Robots y 20 tareas aumentando robots), y la línea 2 de la Tabla 7.4 (3 Robots y 20 tareas aumentando tareas). Lo que explica esas diferencias es que las ejecuciones se realizaron con diferentes parámetros. Para el primer caso, 20 tareas incrementando robots, se iniciaron todas las tareas al comienzo. Eso bajó el tiempo total pero aumentó el tiempo promedio por tarea. Para el segundo caso 3 robots incrementando tareas 10..100, las tareas iban apareciendo de a grupos (como puede apreciarse según los valores en las constantes). Esto hizo que el tiempo inicial para las tareas 10-18 comenzaran 80 segundos después, y las 19-20 comenzaran 160 segundos después. Cada grupo arrancó el tiempo en cero (cuando apareció). Esto provocó que bajara bastante el tiempo promedio de ejecución, pero como tiene una espera de 80 segundos entre grupos, el overhead de creación hizo que el tiempo total aumentara.

7.4.3. Experimento 3: Robustez y Adaptabilidad Dinámica

- **Objetivo:** Validar la capacidad del sistema para manejar fallos de agentes y la adición dinámica de nuevos agentes en tiempo real.
- **Escenarios:**
  1. **Base (Control):** $N_{robots} = 3$, $N_{tareas} = 20$. Ejecución sin interrupciones.
  2. **Fallo:** $N_{robots} = 3$, $N_{tareas} = 20$. Se finaliza manualmente el proceso de un robot a mitad de la misión (al completarse 10 tareas).
  3. **Adición:** $N_{robots} = 2$, $N_{tareas} = 20$. Se inicia un tercer robot a mitad de la misión (al completarse 10 tareas).
- **Condiciones del experimento:** Para este experimento, no se espera que el sistema llegue a saturarse; por lo tanto, para el caso base y el caso de fallo, se disponibilizaron las 20 tareas al inicio de la misión, con una diferencia de hasta un segundo como máximo entre una y la siguiente. Sin embargo, por la forma en que se implementó que un robot detecta que hay tareas disponibles escuchando en un tópico, para el caso de adición fue necesario demorar la aparición de las tareas desde la 11 hasta la 20, para que el tercer robot (el que se añade luego de finalizar la tarea 10) pueda enterarse de las tareas que faltan por resolver.

Análisis de Resultados (Experimento 3) Robustez y Adaptabilidad Dinámica

Este experimento fue diseñado para probar la resiliencia y la adaptabilidad del sistema frente a cambios dinámicos en la composición del equipo. Se eva-

luaron tres escenarios sobre una carga constante de 20 tareas. Los resultados consolidados se presentan en la Tabla 7.5.

Tabla 7.5: Resultados del Experimento 3: Robustez y Adaptabilidad para 20 tareas.

Escenario	F_{ratio} **(%)**	T_{mision} **(seg)**	$\mu(T_{tarea})$ **(seg)**	$\sigma(T_{tarea})$ **(seg)**
Base (3 Robots)	*100.0*	*145.425*	*77.896*	*35.950*
Fallo (3 → 2 Robots)	*100.0*	*146.532*	*77.236*	*35.505*
Adición (2 → 3 Robots)	*100.0*	*150.030*	*47.828*	*18.013*

Robustez y Tolerancia a Fallos (F_{ratio})

Un resultado importante de esta prueba es que la Tasa de Finalización de Misión (F_{ratio}) fue del **100.0 % en los tres escenarios**. Esto demuestra de manera concluyente la robustez fundamental de la arquitectura. Incluso en el escenario de *Fallo*, donde un agente fue eliminado abruptamente, los robots restantes reevaluaron sus motivaciones, se adjudicaron las tareas huérfanas y completaron la misión con éxito.

Análisis del Tiempo de Misión (T_{mision})

Tal como se ve en la Tabla 7.5, los Tiempos Totales de Misión son notablemente similares en los tres casos, fluctuando todos en un rango estrecho (145-150 segundos).

- **Caso Base (3 Robots)** este caso sirvió como referencia, con un T_{mision} de **145.4 s**.
- **Caso Fallo (3 → 2 Robots)** finalizó en **146.5 s**. Este tiempo es casi idéntico al del Caso Base y al T_{mision} de un equipo de 2 robots sanos del Experimento 1 (147.6 s). Esto demuestra una **degradación controlada**: el sistema funcionó eficientemente con 3 robots en la primera mitad y, tras el fallo, su rendimiento general simplemente se ajustó al de un equipo de 2 robots.
- **Caso Adición (2 → 3 Robots)** completó la misión en **150.0 s**, siendo el escenario más lento de los tres. Este resultado es contraintuitivo, pero se explica por la metodología de la prueba. Para que el tercer robot (añadido dinámicamente) pudiera tomar tareas, las tareas 11-20 debían publicarse después de su incorporación, una vez finalizada la `Tarea_10`. La adición en caliente de un nuevo robot y la creación simultánea de 10 nuevas tareas (60+ nodos generados en tiempo de ejecución) impuso una penalización de rendimiento significativa que resultó en el T_{mision} más largo.

Análisis del Cronograma de Tareas (Throughput)

La Figura 7.6 proporciona la visualización de la adaptabilidad del sistema. El eje X representa las tareas completadas acumulativamente, y el eje Y representa el tiempo transcurrido. La pendiente de la curva representa el *throughput* (tasa de completado) del equipo.

- La progresión del **Caso Base** (azul) muestra una pendiente baja. Sirve como referencia del rendimiento básico: cada nueva tarea completada (un paso en el eje X) añade una cantidad de tiempo (un salto en el eje Y) pequeña y predecible.

- La progresión del **Caso Fallo** (rojo) inicia con una pendiente baja, idéntica a la del Caso Base, ya que ambos escenarios comienzan con 3 robots. Después de la tarea 10 (mitad de la misión), era esperable observar que la curva sufriría una **inflexión positiva**: la pendiente **aumentaría** (se volvería visiblemente más empinada). Eso reflejaría la pérdida de capacidad de trabajo porque cada tarea restante (paso en X) requeriría un salto de tiempo (Y) mayor, ya que solo 2 robots estárían disponibles. Sin embargo, se observa que las curvas permanecen prácticamente superpuestas.

- Para la progresión del **Caso Adición** (negro) cabía esperar que iniciara con la pendiente **más alta** de las tres (la más empinada) ya que los 2 robots iniciales tardarían más tiempo (saltos en Y más grandes) en completar las primeras 10 tareas. Sin embargo, se observó una curva superpuesta al caso base. A mitad de la misión (X=10), se esperaba que experimentara una **inflexión negativa**: la pendiente **decrecería** (se aplanaría), igualando la pendiente del Caso Base. Eso visualizaría perfectamente cómo el sistema absorbería la capacidad del 3$^{\text{er}}$ robot, reduciendo el tiempo necesario para completar cada tarea restante. Sin embargo, el resultado observado fue que la pendiente creció. Este resultado se dio por la forma en que se realizó el experimento, iniciando con 10 tareas y creando las restantes 10 en la mitad de la ejecución. Eso era necesario para que el tercer robot (el que se añadía) pudiera ver esas nuevas tareas y crear los nodos correspondientes. Se evidencia que la penalización generada por ejecutar el experimento en una única PC superó el beneficio de incorporar un nuevo robot al equipo.

Análisis de Métricas por Tarea (μ y σ)

Se observa que el $\mu(T_{tarea})$ del Caso Base (77.9 s) y del Caso Fallo (77.2 s) son muy similares.

Notablemente, el $\mu(T_{tarea})$ del Caso Adición es significativamente menor (47.8 s). Esto se debe a que la segunda mitad de las tareas (11-20) fue publicada cuando el tercer robot ya estaba activo y la `Tarea_10` había finalizado. De esta forma, el tiempo inicial para las tareas (11-20) comenzó en un momento muy avanzado de la misión y, por lo tanto, el tiempo de ejecución de esas tareas se

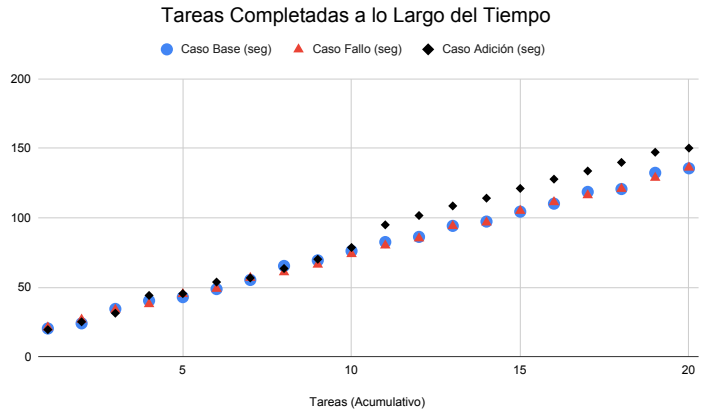

Figura 7.6: Cronograma de tareas completadas (acumulativo) para los tres escenarios de adaptabilidad dinámica.

reinicia a tiempos bajos. La baja desviación estándar (18.0 s) corrobora que los tiempos de espera fueron consistentes en este escenario.

Conclusión (Experimento 3)

El Experimento 3 valida exitosamente las propiedades de **robustez** y **adaptabilidad dinámica** de la arquitectura. El sistema demostró ser tolerante a fallos, completando el 100 % de las tareas tras la pérdida de un agente. Asimismo, demostró ser adaptable, incorporando un nuevo agente de manera dinámica con la intención de mejorar el rendimiento.

El análisis de los Tiempos de Misión (T_{mision}), del Cronograma de Tareas (Throughput) y de las Métricas por Tarea (μ y σ) no es concluyente en el escenario utilizado.

Capítulo 8

Conclusiones y Trabajo Futuro

En este capítulo, se presentan las conclusiones del trabajo realizado y se mencionan algunas ideas para trabajos futuros que utilicen como base y extiendan lo desarrollado en el presente proyecto.

8.1. Conclusiones

El objetivo principal de este proyecto de grado fue **implementar la arquitectura Alliance** de control para sistemas robóticos cooperativos, generando un paquete accesible y reutilizable. Este objetivo fue plenamente alcanzado, entregando un motor de Alliance funcional y extensible, capaz de aplicarse en diferentes escenarios.

- **Plataforma y Robustez:** El desarrollo se construyó sobre **ROS 2 (Robot Operating System)** utilizando **Python**. Esta elección fue fundamental, ya que su infraestructura basada en el *middleware* **DDS (Data Distribution Service)** proporciona un entorno robusto, escalable y distribuido, eliminando la dependencia de un nodo maestro (punto de fallo crítico en ROS).

- **Innovación Clave:** Una innovación relevante del trabajo fue la implementación de la **gestión dinámica de tareas y nodos**. Esta característica, que permite la creación y eliminación dinámica de nodos Motivacional y Executer en respuesta a la aparición de nuevas tareas, añade una flexibilidad crucial al sistema.

- **Validación del Sistema:** El motor fue validado a través de un caso de estudio simulado en una **granja de manzanas** utilizando **CoppeliaSim Edu**.

- **Escalabilidad:** Los resultados de la experimentación demostraron que el sistema presenta **escalabilidad horizontal**, aunque está limitada por la frecuencia de generación de tareas y los recursos de cada robot, ya que la incorporación de más robots reduce los tiempos de ejecución en las condiciones en que se realizaron los experimentos. La distribución de tareas se confirmó como eficiente y equilibrada.

- **Tolerancia a Fallos:** Las pruebas de robustez confirmaron que **todas las tareas fueron completadas**. Esto valida la resiliencia inherente a la arquitectura Alliance, demostrando su capacidad de adaptación y reorganización frente a fallos.

- **Limitaciones Identificadas:** Se identificaron limitaciones relacionadas con la **sobrecarga computacional** generada por la gestión dinámica de nodos en una única PC. Para la implementación en robots reales, se detectó la necesidad de contar con algoritmos de navegación más avanzados.

8.2. Trabajo Futuro

Para empezar a mencionar el trabajo futuro, se deben considerar mejoras al resultado obtenido; por ejemplo, en algunas de las decisiones tomadas, por opciones que **mejoren la eficiencia de los algoritmos implementados**.

A continuación, se proponen diversas líneas de extensión, muchas de las cuales estaban originalmente fuera del alcance del presente proyecto:

- **Mejoras en la implementación detectadas:**

  - La implementación actual utiliza solamente tópicos para la comunicación entre nodos. En un trabajo futuro que necesite mejorar el rendimiento del sistema, podría investigarse la posibilidad de sustituir algunos tópicos por servicios y/o acciones de ROS 2 en determinados lugares, según sea conveniente.
  - Con el objetivo de facilitar la calibración del sistema para la ejecución en diferentes escenarios, se detecta una posible mejora para un trabajo futuro que consiste en leer las constantes y parámetros de Alliance desde un archivo de configuración. Actualmente, estos valores están almacenados en constantes del sistema, y al cambiarlos, es necesario recompilar el sistema para que surtan efecto.

- **Robustez de la Comunicación:** Durante el desarrollo del proyecto, se revisaron distintos enfoques presentes en la literatura, entre ellos la tesis resumida en el capítulo de antecedentes, Sección 3.2, "*Multi-robot exploration under non-ideal communication conditions*". Si bien sus planteamientos no fueron incorporados en la implementación actual, considerarlos en futuras extensiones podría aportar una mayor robustez comunicacional al sistema Alliance. En particular, permitirían mejorar la coordinación en

entornos con comunicación limitada y reducir trayectos innecesarios de los robots.

- **Expansión del Equipo Heterogéneo:** Se pueden incluir los **recolectores robotizados** en el mismo equipo de Alliance, aprovechando que la arquitectura está diseñada para equipos heterogéneos con diferentes capacidades y que implementan distintos comportamientos.

- **Mejoras en la Percepción y la Navegación:**

  - Se debe mejorar la obtención de la posición, orientación y velocidad del robot en condiciones no ideales que simulen escenarios más reales.
  - Se podría incorporar **ruido a los datos del GPS** para simular condiciones menos ideales.
  - Es necesaria la integración de **sensores adicionales**, como los sensores láser, para implementar algoritmos de navegación más complejos.
  - Se deben implementar **algoritmos de navegación avanzada**, como el recorrido paralelo a filas de árboles.
  - El sistema y, en particular, la navegación, podrían verse beneficiados si se utilizaran técnicas de mapeo y localización como SLAM (*Simultaneous Localization and Mapping*).
  - **Integración con el Stack de Navegación (Nav2)** Actualmente, la ejecución del movimiento se basa en primitivas de control directo. La integración del stack *Navigation 2* (Nav2) para gestionar la movilidad de los agentes permitiría delegar la planificación de trayectorias (global y local) y la evasión de obstáculos dinámicos a un estándar industrial robusto. Esto habilitaría a los robots para operar en entornos no estructurados y desconocidos mediante técnicas de SLAM, desacoplando la lógica de decisión: la arquitectura Alliance mantendría la responsabilidad de determinar *qué* tarea realizar (nivel motivacional), mientras que Nav2 resolvería *cómo* navegar eficientemente hacia el objetivo (nivel de ejecución), actuando como un servidor de acciones para los comportamientos del sistema.

- **Jerarquización de tareas:** Es posible jerarquizar las tareas dentro de cierto rango para obtener resultados específicos; por ejemplo, algunos robots podrían competir solamente por las tareas más cercanas para evitar largos desplazamientos innecesarios, y otros por las más antiguas para evitar postposiciones indefinidas.

- **Aprendizaje de los parámetros:** Se podrían aplicar algoritmos de aprendizaje para los parámetros de Alliance. En particular, otra posible línea de trabajo a futuro sería extender Alliance con el modelo L-Alliance (Parker, 1997) cuyo objetivo es aprender y ajustar dinámicamente los valores de los parámetros de Alliance para mejorar el rendimiento.

- **Adopción de Árboles de Comportamiento (Behavior Trees)** Se propone también como trabajo futuro la migración de la lógica de ejecución interna hacia Árboles de Comportamiento (*Behavior Trees*) (Colledanchise y Ogren, 2018). Este paradigma constituye el estándar *de facto* en el ecosistema moderno de ROS 2 (siendo el núcleo de stacks como Nav2) y ofrece ventajas significativas sobre la implementación actual basada puramente en subsunción. Su estructura jerárquica y modular permitiría una gestión más eficiente de secuencias complejas y dependencias entre tareas —una limitación conocida de la arquitectura Alliance original— mejorando la legibilidad del código, la reactividad del sistema y la facilidad de integración con herramientas de navegación avanzadas.

Glosario

actuador Dispositivo que permite al robot modificar su entorno con efectos que pueden ser físicos o lógicos. 9

agente Entidad capaz de percibir su entorno y actuar en él. 83

agente autónomo Se dice que un agente carece de autonomía cuando se apoya más en el conocimiento inicial que en sus propias percepciones. Un agente racional debe ser autónomo. Incorporar aprendizaje facilita el diseño de agentes racionales. 85

agente robótico Agente robótico o agente situado, es el que tiene interacción bidireccional con el entorno físico. El agente no es solo un observador pasivo del entorno, sino que está inmerso en él. Sus acciones tienen un impacto directo y son influenciadas por las condiciones del entorno. 8

AI IA (Inteligencia Artificial) o AI por su sigla en inglés (Artificial Intelligence) es un campo de la informática dedicado al desarrollo de sistemas informáticos capaces de realizar tareas que normalmente requieren inteligencia humana. Estas tareas incluyen aprendizaje, percepción, toma de decisiones, resolución de problemas y comprensión del lenguaje natural. El objetivo de la AI es permitir que las máquinas razonen, aprendan y actúen para lograr objetivos complejos. 2

ambiente parcialmente observable Si el aparato sensorial del agente le permite tener acceso al estado total de un ambiente se dice que éste es accesible al agente o totalmente observable, sino es un ambiente parcialmente observable. 45

API Del inglés Application Programming Interface, es un conjunto de reglas, protocolos y herramientas que permiten que diferentes aplicaciones de software se comuniquen entre sí. Funciona como un contrato de servicio, definiendo cómo una aplicación puede solicitar servicios o intercambiar datos con otra. 47

basada en comportamientos Arquitectura basada en comportamientos (Behavior-Based Architecture): Modelo de control en robótica que organiza la inteligencia del sistema en módulos reactivos independientes. A

diferencia de los modelos lineales, cada "comportamiento" conecta directamente sensores con actuadores, permitiendo respuestas rápidas y una inteligencia emergente. Se caracteriza por ser distribuida, asíncrona y, a menudo, jerarquizada mediante mecanismos de supresión o inhibición.. 13

comportamiento Correspondencia entre estímulos y respuestas en un agente. El comportamiento corresponde tanto a las acciones observables de un individuo, como a los procesos internos subyacentes a las mismas. La conducta es un subconjunto (el visible) del comportamiento total del individuo. 9, 13

comportamiento emergente Aparición de nuevas propiedades observadas a nivel del sistema. Funcionalidades globales emergen a partir de la interacción paralela de comportamientos locales. Se alcanzan funcionalidades emergentes en virtud de la interacción entre componentes que no fueron diseñados para brindarla. 13

conducta Son aquellas acciones que se producen después de una determinada secuencia de percepciones. La conducta corresponde a las acciones observables de un individuo. 13, 84

continuo Si existe una cantidad limitada de percepciones y acciones claramente discernibles, se dice que el ambiente es discreto. Si la cantidad es ilimitada, se trata de un ambiente continuo. 45

controlador Es un programa de software que permite a un sistema operativo interactuar y controlar un dispositivo de hardware específico, como una impresora, una tarjeta gráfica, una tarjeta de sonido o un ratón. 53

DDS DDS (Data Distribution Service) Es un estándar de comunicación Machine-to-Machine (M2M), orientado a la publicación-suscripción, diseñado específicamente para sistemas distribuidos en tiempo real y de alto rendimiento. DDS permite a los nodos de un sistema (como robots, sensores o controladores) compartir datos de forma directa, fiable y con calidad de servicio (QoS) predefinida, sin la necesidad de un broker o servidor central. 2, 63

dinámico Si existe la posibilidad de que el ambiente sufra modificaciones mientras el agente se encuentra deliberando se dice que el ambiente se comporta de forma dinámica con relación al agente. En ambientes estáticos el agente no se debe preocupar por lo que sucede mientras delibera; no se debe preocupar por el paso del tiempo. Ambiente semi-dinámico: no cambia el ambiente pero sí se modifica la calificación asignada al desempeño. 45

entorno El ambiente o entorno, en robótica se refiere al espacio físico o virtual en el que el robot opera. Este puede ser tan simple como una mesa de trabajo o tan complejo como una ciudad entera. Las características del

ambiente influyen en la forma en que el robot percibe el mundo, interactúa con él y toma decisiones. Los sensores del robot deben ser capaces de percibir las características relevantes del ambiente, como la distancia a los objetos, la presencia de obstáculos y otros datos de interés dependiendo del problema a resolver. El controlador del robot debe ser capaz de generar comandos a los actuadores para que el robot se mueva y manipule objetos en el ambiente de forma segura y eficiente. 9

estocástico Si el estado siguiente se determina completamente a partir del estado actual y la acción elegida por el agente, el ambiente es determinista, sino es un ambiente estocástico (No determinista). Si el medio es determinista excepto por la acción de otros agentes el entorno es estratégico. 45

GPS El GPS (Global Positioning System) es un sistema de navegación global por satélite. Utiliza una red de satélites en órbita alrededor de la Tierra para proporcionar información precisa sobre la posición, velocidad y tiempo a un receptor en cualquier lugar del planeta que tenga línea de visión con al menos cuatro de estos satélites. 53

IoT IoT (Internet de las Cosas) es una red de objetos físicos (o "cosas") cotidianos que están interconectados y son capaces de recopilar e intercambiar datos a través de Internet. Estos objetos están equipados con sensores, software y otras tecnologías que les permiten comunicarse, percibir su entorno o ser controlados remotamente. El objetivo principal del IoT es aumentar la eficiencia, automatización y el control sobre los sistemas del mundo físico. 2

Lua Es un lenguaje de programación multiparadigma, ligero y de scripting, diseñado para ser embebido en otras aplicaciones. Creado en 1993 por el grupo de Tecnología en Computación Gráfica (Tecgraf) de la Pontificia Universidad Católica de Río de Janeiro, su nombre significa "Luna" en portugués. 53

PC PC Siglas en inglés de Personal Computer (Computadora Personal). Se refiere a una computadora diseñada para el uso individual, en contraposición a un servidor o una supercomputadora. 64

racional Un agente autónomo es racional cuando "hace lo correcto". En cada posible secuencia de percepciones, un agente racional deberá realizar aquella acción que supuestamente maximice su medida de rendimiento, basándose en las evidencias aportadas y el conocimiento almacenado. 8

RCL ROS Client Library (Biblioteca Cliente de ROS) es la biblioteca de software fundamental, escrita en lenguaje C, que proporciona la interfaz central de comunicación para ROS 2.. 63

secuencia de percepciones Es una recopilación de datos del ambiente a través de los sensores. 85

secuencial En un ambiente episódico, la experiencia del agente se divide en episodios. La calidad de su actuación dependerá únicamente del episodio actual, y los episodios subsiguientes no dependerán de las acciones tomadas en episodios previos. Si el ambiente no es episódico, es un ambiente secuencial. 45

sensor Dispositivo que permite al robot medir el valor de una variable de interés. Opera sobre el entorno o sobre el propio robot. 9

SLAM SLAM (Simultaneous Localization and Mapping) Término que describe un conjunto de algoritmos y técnicas computacionales que permiten a un agente autónomo (como un robot o vehículo) construir un mapa de un entorno desconocido (Mapping) y, de forma simultánea, determinar su propia ubicación y orientación (Localization) dentro de ese mapa que está creando. Es un problema fundamental en la robótica móvil, ya que una localización precisa requiere un mapa preciso, y la construcción de un mapa preciso requiere una localización precisa. 81

unidad de control UC (Unidad de Control) Se define como el componente central del sistema de control de un robot, encargado de procesar la información proveniente de los sensores y ejecutar los algoritmos de control para coordinar el movimiento y las acciones de los actuadores. Actúa como el “cerebro” del sistema. 9

Referencias

Arkin, R. C. (1989). Motor schema—based mobile robot navigation. *The International journal of robotics research*, *8*(4), 92–112.

Arkin, R. C. (1998). *Behavior-based robotics*. Cambridge, MA: MIT Press.

Arkin, R. C., Riseman, E. M., y Hanson, A. (1987). Aura: An architecture for vision-based robot navigation. En *Darpa image understanding workshop* (pp. 417–431).

Benavides, F. (2019). *Multi-robot exploration under non-ideal communication conditions* (Tesis de Doctorado). Universidad de Toulouse, Institut Supérieur de l'Aéronautique et de l'Espace (ISAE), Toulouse, Francia. (Cotutela internacional: PEDECIBA, Universidad de la República (Uruguay))

Brooks, R. (1986). A robust layered countrol system for a mobil robot. *IEEE Journal of Robotics and Automation*, *2*, 1.

Cao, Y. U., Fukunaga, A. S., y Kahng, A. B. (1997). Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, *4*(1), 7–27. doi: 10.1023/A:1008855018923

Colledanchise, M., y Ogren, P. (2018). *Behavior trees in robotics and ai: An introduction*. CRC Press. doi: 10.1201/9780429489105

Coppelia Robotics AG. (2025). *Coppeliasim v4.8.0 rev0.* Descargado de https://www.coppeliarobotics.com/ (enero de 2025)

Dudek, G., Jenkin, M., Milios, E., y Wilkes, D. (1996). A taxonomy for multi-agent robotics. *Autonomous Robots*, *3*(4), 375–397. doi: 10.1007/BF00240651

Gat, E. (1991). Integrating reaction and planning in a heterogeneous asynchronous architecture for mobile robot navigation. *ACM SIGART Bulletin*, *2*(4), 70–74.

Georgeff, M. P., Lansky, A. L., y Schoppers, M. J. (1987). *Reasoning and planning in dynamic domains: An experiment with a mobile robot* (Inf. Téc.). 333 Ravenswood Avenue,Menlo Park,CA,94025: SRI International.

Gerkey, B. P., y Matarić, M. J. (2002). Sold!: Auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, *18*(5), 758–768. doi: 10.1109/TRA.2002.803462

InCo, FIng, UDELAR. (2012a). *Butiá 2.0: Diseño educativo de plataforma robótica móvil para estudiantes escolares.* Proyecto de extensión / investigación. Montevideo, Uruguay. Descarga-

do de https://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php?title=Butia2 (Financiado por ANTEL – Fundación Julio Ricaldoni; grupo de investigación "Network Management & Artificial Intelligence". Años 2012 – 2013)

InCo, FIng, UDELAR. (2012b). *Mecánica butiá v2.0.* Repositorio de proyectos INCO, MediaWiki. Montevideo, Uruguay. Descargado de https://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php?title=Mec%C3%A1nica_Buti%C3%A1_V2.0 ([Imagen]. Años 2012 – 2013)

Lyons, D. M., y Hendriks, A. J. (1992). Planning for reactive robot behavior. En *Proceedings 1992 ieee international conference on robotics and automation* (pp. 2675–2676).

Matarić, M. J. (2007). *The robotics primer.* Cambridge, MA: The MIT Press.

Mottini, M. (2021). *Alliance sobre torocó: Arquitecturas cooperativas basadas en comportamientos* (Tesis de grado en Ingeniería en Computación, Universidad de la República (UDELAR), Facultad de Ingeniería). Descargado de https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/29194/1/MOT21.pdf

Nilsson, N. J. (1984). *Shakey the robot* (Inf. Téc. n.º 323). Menlo Park, CA: SRI International, Artificial Intelligence Center. Descargado de https://www.ai.sri.com/shakey/

Olfati-Saber, R., Fax, J. A., y Murray, R. M. (2007). Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, *95*(1), 215–233. doi: 10.1109/JPROC.2006.887293

Open Source Robotics Foundation, Inc. (2025). *Ros kinetic kame.* Descargado de http://wiki.ros.org/kinetic (enero de 2025)

Open Source Robotics Foundation, I. (2025a). *Gazebo.* Descargado de https://gazebosim.org/home/ (setiembre de 2025)

Open Source Robotics Foundation, I. (2025b). *Ros humble hawksbill.* Descargado de https://docs.ros.org/en/foxy/Releases/Release-Humble-Hawksbill.html (enero de 2025)

Pardo-Castellote, G. (2003). Omg data-distribution service: Architectural overview. En *23rd international conference on distributed computing systems workshops, 2003. proceedings.* (pp. 200–206).

Parker, L. E. (1995, 2). *Alliance: An architecture for fault tolerant multi-robot cooperation* (Inf. Téc.). Oak Ridge National Laboratory. Descargado de https://www.osti.gov/biblio/34318 doi: 10.2172/34318

Parker, L. E. (1997). L-alliance: Task-oriented multi-robot learning in behavior-based systems. *Journal of Advanced Robotics*, *11*(4), 305–322. doi: 10.1163/156855397X00344

Parker, L. E. (1998). Alliance: An architecture for fault tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, *14*(2), 220–240. doi: 10.1109/70.681240

Python Software Foundation. (2025). Python developer's guide [Manual de software informático]. Descargado de https://devguide.python.org/

Quigley, M. (2009). Ros: an open-source robot operating system. En *Icra workshop on open source software* (Vol. 3, p. 5).

Shoham, Y., y Leyton-Brown, K. (2009). *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press.

Siegwart, R., Nourbakhsh, I. R., y Scaramuzza, D. (2011). *Introduction to autonomous mobile robots* (2nd ed.). Cambridge, MA: MIT Press.

Stanford Artificial Intelligence Laboratory et al. (2022). *Robotic operating system.* Descargado de https://www.ros.org (Software)

Tejera, G. D. (2004). *Contribución al diseño de sistemas multi robots utilizando alliance* (Master's thesis, Universidad de la República, Montevideo, Uruguay). Descargado de https://hdl.handle.net/20.500.12008/2933 (Tesis de maestría)

Anexo A

ROS - Robot Operating System

ROS(Quigley, 2009) (Robot Operating System) es una plataforma de software de código abierto la cual proporciona un conjunto de herramientas, bibliotecas y convenciones diseñadas para facilitar el desarrollo de aplicaciones en robótica. Originalmente fue lanzado en 2007 por el laboratorio de robótica de Willow Garage, ROS facilita la creación de sistemas modulares y reutilizables que se pueden distribuir y ejecutar en múltiples dispositivos. La plataforma es ampliamente utilizada en la comunidad de la robótica gracias a su arquitectura modular, que incluye características como nodos, tópicos, mensajes y servicios, los cuales permiten la comunicación eficiente entre distintos componentes en un sistema robótico.

ROS permite a los desarrolladores construir sistemas complejos y colaborativos al proporcionar infraestructura de comunicación, simuladores como Gazebo y herramientas de visualización como Rviz. Además, gracias a su amplio soporte y comunidad activa, ROS ha evolucionado en una segunda versión, ROS 2(Stanford Artificial Intelligence Laboratory et al., 2022), que aborda limitaciones en aspectos de seguridad, comunicación en tiempo real y distribución de sistemas en entornos industriales y de alta complejidad.

A.1. Introducción a ROS 2

ROS 2 es una infraestructura de middleware diseñada para desarrollar aplicaciones robóticas de manera modular y distribuida. Se desarrolló como sucesor de ROS, con el objetivo de mejorar la flexibilidad, confiabilidad y escalabilidad, especialmente para sistemas robóticos colaborativos y aplicaciones industriales.

A.2. Motivación y Evolución desde ROS 1

ROS 2 surgió para superar algunas limitaciones de ROS1, particularmente en aspectos como la seguridad, la escalabilidad en entornos distribuidos, y la compatibilidad en aplicaciones en tiempo real. ROS1 no estaba diseñado para estos contextos, lo que limitaba su adopción en entornos de producción. Con ROS 2, se incorporaron mejoras en la arquitectura de comunicación, haciendo posible su uso en contextos industriales y en proyectos robóticos de gran escala.

A.3. Arquitectura y Componentes Clave

ROS 2 se basa en el Data Distribution Service (DDS)(Pardo-Castellote, 2003), un estándar de middleware de comunicación orientado a datos que facilita la interoperabilidad y la comunicación distribuida sin necesidad de un servidor centralizado. Esto permite a ROS 2 soportar aplicaciones en redes heterogéneas, lo que resulta esencial para sistemas multi-robot.

Los componentes clave de ROS 2 incluyen:

Nodos: Unidades de ejecución que representan procesos independientes. La arquitectura distribuida permite que los nodos se ejecuten en distintas máquinas.

Topologías de comunicación: ROS 2 emplea el paradigma Publisher-Subscriber para la transmisión de mensajes y el paradigma Cliente-Servidor para llamadas a servicios, ambos gestionados a través de DDS.

Parámetros y Launch Files: Los nodos en ROS 2 pueden ser configurados con parámetros dinámicos, y los Launch Files permiten gestionar el despliegue de múltiples nodos simultáneamente.

A.4. Comunicación y DDS

DDS permite la comunicación descentralizada entre nodos, asegurando calidad de servicio (QoS) con opciones como confiabilidad, prioridad y durabilidad de mensajes. Esto permite que ROS 2 soporte escenarios en los que los robots deben adaptarse a redes de baja calidad o a la falta intermitente de conectividad, algo común en sistemas multi-robot distribuidos.

A.5. Herramientas de ROS 2

ROS 2 ofrece un conjunto de herramientas y bibliotecas que son muy útilies para el desarrollo y la simulación de aplicaciones robóticas:

- rcl y rclcpp/rclpy: Son capas de abstracción que permiten que el código sea independiente del middleware, haciéndolo portable.

- Gazebo y rviz: Permiten la simulación física y visualización 3D, ayudando en el desarrollo, prueba y depuración.
- Colcon: Herramienta de compilación específica para proyectos ROS 2, facilitando la gestión de múltiples paquetes de software.

Anexo B

CoppeliaSim

CoppeliaSim es un simulador de robots ampliamente utilizado en investigación y desarrollo debido a su flexibilidad, facilidad de uso y soporte para múltiples plataformas. Ofrece un entorno de simulación física donde se pueden modelar y probar robots, sistemas mecánicos y entornos interactivos de manera eficiente. Su arquitectura basada en scripts y su capacidad de integración lo hacen ideal para simulaciones avanzadas.

B.1. Características principales de CoppeliaSim

1. Motor de simulación:
   - Utiliza motores físicos como Bullet, ODE, Vortex y Newton para simular dinámicas reales.
   - Permite simulaciones cinemáticas, dinámicas y robóticas de alta precisión.
2. Programación y automatización:
   - Soporte para múltiples lenguajes de programación (Lua, Python, C/C++, Java, etc.).
   - Scripts integrados en la simulación para personalizar el comportamiento del entorno y los robots.
3. Interactividad:
   - Interfaz gráfica intuitiva para la creación y manipulación de escenas.
   - Herramientas para visualizar trayectorias, sensores y estados del sistema en tiempo real.
4. Biblioteca de robots y objetos:

- Incluye modelos predefinidos de robots populares como UR5, KUKA, y Pioneer.
- Facilita la creación de sistemas robóticos personalizados.

5. Extensibilidad:
   - Compatible con APIs remotas para controlar la simulación desde aplicaciones externas.
   - Soporte para intercambio de datos con otros programas y simuladores.

B.2. Interacción con ROS 2

CoppeliaSim puede integrarse con ROS 2 (Robot Operating System 2), lo que permite a los desarrolladores usar el simulador como un entorno de pruebas para aplicaciones robóticas basadas en ROS 2.

1. Características de la integración:
   - Publicadores y suscriptores de ROS 2: Permite publicar y recibir datos desde tópicos ROS 2.
   - Soporte para servicios y acciones ROS 2: La simulación puede interactuar con nodos ROS 2 utilizando servicios y acciones.
   - Intercambio de datos en tiempo real: Los sensores y actuadores simulados en CoppeliaSim pueden ser controlados por nodos ROS 2.
   - Integración modular: Se pueden conectar escenas simuladas a pipelines de ROS 2, como SLAM, navegación y planificación de trayectorias.
2. Ventajas de la integración:
   - Proporciona un entorno controlado para probar algoritmos de ROS 2 sin necesidad de hardware físico.
   - Permite simular robots completos con sensores virtuales y sistemas dinámicos antes de implementarlos en el mundo real.
3. Configuración para ROS 2:
   - Requiere instalar los complementos específicos para ROS 2 en CoppeliaSim.
   - Es necesario configurar los nodos ROS 2 para interactuar con la API de CoppeliaSim a través de mensajes y servicios.

Anexo C

Manual de Usuario

En este apéndice se describe el procedimiento para configurar el ambiente de trabajo necesario para ejecutar el código desarrollado en el presente proyecto.

C.1. Requisitos Previos

Antes de comenzar, asegúrese de contar con lo siguiente:

- **Sistema operativo:** Ubuntu Linux (preferentemente Ubuntu 22.04).
- **Conexión a internet:** necesaria para descargar dependencias.
- **Otros:** ver Sección C.1.1.

C.1.1. Software Requerido

Asegúrese de tener instalado lo siguiente:

- **Git:** para clonar el repositorio del proyecto.
- **Python 3.8 o superior:** intérprete del lenguaje utilizado.
- **Pip:** gestor de paquetes de Python (incluido por defecto).
- **Editor de código:** VS Code o cualquier IDE compatible.

C.2. Instalación de ROS 2

Siga la guía oficial de instalación de ROS 2 Humble Hawksbill disponible en `https://docs.ros.org/en/humble/Installation.html`. Asegúrese de que los comandos `ros2` y `colcon` estén disponibles en la terminal antes de continuar.

C.3. Guía de Configuración del Entorno

C.3.1. Crear el Workspace

Abra una terminal y ejecute:

```
mkdir -p ~/ros2_ws/src
```

C.3.2. Clonar el Repositorio

Existen dos repositorios posibles, uno para las pruebas de validación y otro para el simulador:

1. Para las pruebas de validación:

   ```
   git clone --single-branch --branch validarArqSinVRep \
   git@gitlab.fing.edu.uy:proysistrobcoop/ProySistRobCoop.git
   ```

2. Para pruebas con el simulador:

   ```
   git clone git@gitlab.fing.edu.uy:proysistrobcoop/ProySistRobCoop.git
   ```

3. Copie el contenido al workspace creado:

   ```
   cp -Rp alliance* ~/ros2_ws/src
   cp -Rp cocoro* ~/ros2_ws/src
   ```

C.3.3. Compilar y Configurar Variables de Entorno

Ubíquese en el workspace y compile:

```
cd ~/ros2_ws
colcon build
```

Luego, para que los nodos sean accesibles desde la terminal, ejecute:

```
source install/local_setup.bash
```

C.4. Ejecución del Proyecto

C.4.1. Modo Validación

Abra dos terminales:

1. **Consola 1:** para lanzar las tareas:

   ```
   ros2 launch alliance gen_tareas_n_tareas.launch.py num_tareas:=10
   ```

2. **Consola 2:** para ejecutar el código principal:

   ```
   ros2 launch alliance 3RobotNoVRep.launch.xml
   ```

C.4.2. Modo Simulación con CoppeliaSim

Abra tres terminales:

1. **Consola 1:**

   ```
   ros2 launch alliance gen_tareas_n_tareas.launch.py num_tareas:=10
   ```

2. **Consola 2:**

   ```
   ros2 launch alliance 3RobotNoVRep.launch.xml
   ```

3. **Consola 3:** ejecute el simulador:

   ```
   cd CoppeliaSimEdu
   ./coppeliasim
   ```

   Dentro de CoppeliaSim, cargue la escena correspondiente para la simulación.