Libro ASR

Curso de

Arquitecturas Software para Robots

Libro Universitario de Ingeniería de Robótica Software

Francisco Martín Rico Rodrigo Pérez Rodríguez

v27.0.0

23 de junio de 2026

Intelligent Robotics Lab

Copyright
© 2026 Francisco Martín Rico y Rodrigo Pérez Rodríguez.

Licencia
Este libro se distribuye bajo la licencia *Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0).

Se permite **compartir** (copiar y redistribuir) el material en cualquier medio o formato, siempre que se cumpla: **atribución** (reconocer la autoría), **uso no comercial** y **sin obras derivadas** (no se permite remezclar, transformar ni crear a partir del material).

Texto completo de la licencia:
https://creativecommons.org/licenses/by-nc-nd/4.0/

Colofón
Este documento se compuso con KOMA-Script y LaTeX, usando la clase kaobook.

Editorial
Primera edición: enero de 2026. Publicado por Intelligent Robotics Lab

Writing good code is like brushing your teeth: you should do it without thinking too much about it.
– Davide Faconti

Prefacio

La robótica es, por naturaleza, una disciplina integradora. Un sistema robótico moderno combina percepción, decisión y actuación sobre un entorno físico incierto, y lo hace mediante software distribuido, concurrente y fuertemente acoplado al tiempo. Sin embargo, en muchos planes de estudio, el software para robots se aborda de forma fragmentada: algoritmos por un lado, herramientas por otro, y con frecuencia sin un marco arquitectónico que permita entender cómo se construyen sistemas completos, mantenibles y escalables.

Este libro nace con el objetivo de cubrir ese vacío desde una perspectiva clara: formar ingenieros e ingenieras en robótica especializados en software, capaces no solo de programar componentes aislados, sino de diseñar, razonar, evaluar y evolucionar arquitecturas software para robots reales.

Sobre los contenidos teóricos

La primera parte del libro está dedicada a los fundamentos de la arquitectura software en robótica. No pretende ser una enciclopedia exhaustiva ni un catálogo de patrones abstractos, sino un recorrido estructurado por los conceptos que un ingeniero robótico especializado en software debe dominar para enfrentarse a sistemas reales.

Los capítulos iniciales introducen el concepto de arquitectura software en robótica y discuten el papel de los *middlewares* de programación como infraestructura que materializa las decisiones arquitectónicas. En particular, se presenta cómo un middleware (por ejemplo ROS 2, *Robot Operating System 2*, y su base DDS, *Data Distribution Service*) condiciona y habilita aspectos clave como la comunicación, el despliegue, la observabilidad y los modelos de ejecución. A partir de ello, se introducen los modelos de ejecución reactivos, el flujo de datos, los eventos y la concurrencia, poniendo especial énfasis en el tiempo como una dimensión fundamental del comportamiento robótico.

En los capítulos centrales se abordan los aspectos estructurales del software: componentes, interfaces, comunicación, estado y representación espacial. Estos temas no se presentan como detalles de implementación, sino como decisiones arquitectónicas con implicaciones directas en la robustez, la depuración y la escalabilidad del sistema. La percepción, la gestión del estado distribuido y el uso de infraestructuras como TF (*Transform Frames*) se analizan desde esta perspectiva, evitando un enfoque puramente algorítmico.

La generación de comportamiento ocupa un lugar destacado en el libro, mediante el estudio de máquinas de estados finitos (FSM) y Behavior Trees (BT). Ambos modelos se presentan no como soluciones universales, sino como herramientas complementarias, adecuadas para distintos niveles de decisión. Esta distinción resulta clave para entender cómo se estructuran sistemas robóticos complejos en la práctica.

El bloque teórico culmina con un capítulo de síntesis y análisis que combina tres perspectivas complementarias: el estudio de arquitecturas de referencia en ROS 2, la discusión de criterios transversales de evaluación arquitectónica, y una mirada más profunda a mecanismos internos de ROS 2 como executors, callback groups, lifecycle y consideraciones de tiempo real. De este modo, el cierre de la parte teórica no se limita a introducir nuevos conceptos, sino que conecta los capítulos anteriores con problemas reales de integración, observabilidad, robustez, despliegue y evolución del sistema.

Es importante señalar también lo que este libro no pretende cubrir en profundidad. No se abordan aquí algoritmos avanzados de planificación, SLAM (*Simultaneous Localization and Mapping*) desde un punto de vista matemático, control óptimo, verificación formal o seguridad funcional. Estos temas pertenecen a asignaturas específicas, a estudios de posgrado o a contextos profesionales concretos. El objetivo de este libro no es formar especialistas en cada uno de esos ámbitos, sino proporcionar una base sólida para integrarlos correctamente dentro de una arquitectura software robótica.

Sobre la parte práctica

La segunda parte del libro está organizada como una progresión de seis prácticas de complejidad creciente, diseñada para acompañar y reforzar los conceptos teóricos. Estas prácticas no deben entenderse como ejercicios aislados ni como recetas cerradas, sino como experiencias de diseño arquitectónico con un grado de apertura creciente.

Las primeras prácticas tienen un carácter claramente guiado: fijan el entorno de trabajo, consolidan herramientas y establecen patrones básicos de ejecución reactiva, percepción y control. En este tramo inicial, cada enunciado ofrece una guía detallada y, cuando es necesario, una sección específica de teoría y herramientas para facilitar la implementación.

Las prácticas intermedias mantienen objetivos funcionales observables, pero abren progresivamente el espacio de decisiones de diseño: se integran capacidades más complejas, se explicita la separación entre misión, tareas y capacidades, y se comparan modelos de comportamiento (FSM y Behavior Trees) en niveles arquitectónicos distintos. De este modo, la teoría no se aplica de forma estrictamente lineal, sino cuando resulta necesaria para resolver problemas concretos de integración.

En el tramo final de la parte práctica, el enfoque se vuelve más integrador y abierto: se exige diseñar y justificar una arquitectura completa Misión–Tarea–Capacidad, combinando los elementos trabajados en el recorrido anterior. En esta etapa, el énfasis recae menos en seguir pasos prescritos y más en argumentar decisiones, validar el comportamiento global y demostrar coherencia arquitectónica entre diseño e implementación.

Sobre ROS 2

ROS 2 ocupa un papel central en este libro, no como un mero entorno de desarrollo o una colección de herramientas, sino como una infraestructura software que materializa muchos de los conceptos arquitectónicos que se estudian a lo largo del texto. La elección de ROS 2 no responde únicamente a su adopción generalizada en investigación e industria, sino a su adecuación como marco pedagógico para razonar sobre arquitecturas software en robótica.

A lo largo del libro, ROS 2 se utiliza como soporte concreto para ilustrar conceptos abstractos de arquitectura software. Nociones como componentes, interfaces, comunicación asíncrona, ejecución reactiva, concurrencia, estado distribuido o ciclo de vida no se presentan de forma aislada, sino siempre conectadas con los mecanismos que ROS 2 proporciona para implementarlas. De este modo, ROS 2 actúa como un puente entre la teoría arquitectónica y los sistemas robóticos reales.

Es importante subrayar que este libro no pretende enseñar ROS 2 como un fin en sí mismo. No se concibe como un manual de referencia exhaustivo ni como una guía paso a paso de todas sus APIs (*Application Programming Interfaces*). En su lugar, ROS 2 se presenta como un ejemplo representativo de una infraestructura moderna de robótica software, cuyas decisiones de diseño permiten discutir ventajas, limitaciones y trade-offs arquitectónicos.

El uso de ROS 2 permite, además, introducir de forma natural aspectos que son esenciales en sistemas robóticos reales y que a menudo quedan fuera de enfoques más abstractos: ejecución distribuida, comunicación basada en eventos, gestión explícita del tiempo, configuración dinámica, observabilidad del estado y robustez frente a fallos. Estos elementos no se tratan como detalles técnicos, sino como partes integrales del diseño arquitectónico.

Finalmente, el enfoque adoptado en este libro busca que el lector adquiera una comprensión transferible. Aunque los ejemplos y las prácticas se basan en ROS 2, los principios arquitectónicos subyacentes no dependen de una herramienta concreta. El objetivo es que el estudiante sea capaz de reconocer estos principios en otros frameworks, plataformas o contextos profesionales, y de aplicar el mismo criterio arquitectónico más allá de ROS 2.

Alcance y filosofía del libro

Este libro está pensado para un Grado en Ingeniería de Robótica Software, y su alcance es deliberadamente formativo. No busca ofrecer soluciones cerradas ni imponer una arquitectura concreta, sino proporcionar un lenguaje común, un marco de razonamiento y un criterio técnico que permita a los futuros profesionales enfrentarse a sistemas robóticos reales con solvencia.

Si al finalizar el curso el lector es capaz de analizar una arquitectura robótica existente, comprender sus decisiones, identificar sus limitaciones y proponer mejoras fundamentadas, entonces este libro habrá cumplido su objetivo.

Lista de acrónimos

API	Application Programming Interface
AMCL	Adaptive Monte Carlo Localization
BT	Behavior Trees
CPU	Central Processing Unit
DDS	Data Distribution Service
FSM	Finite State Machine
IMU	Inertial Measurement Unit
LIDAR	Light Detection and Ranging
NPC	Non-Player Character
PDDL	Planning Domain Definition Language
PID	Proportional-Integral-Derivative
RGBD	RGB-Depth
RMW	ROS Middleware Interface
ROS	Robot Operating System
ROS 2	Robot Operating System 2
SLAM	Simultaneous Localization and Mapping
TF	Transform Frames
YAML	YAML Ain't Markup Language

Índice general

Lista de acrónimos vii

Índice general viii

Teoría de Arquitecturas Software para Robots 1

1 Arquitectura software en robótica 2
1.1 Principios teóricos . . . 2
Qué es arquitectura software . . . 2
Qué es arquitectura software en robótica . . . 3
Middlewares de programación de robot . . . 5
1.2 Aplicación . . . 9
ROS 2 como materialización de una arquitectura software robótica . . . 9
Componentes, procesos y comunicación en ROS 2 . . . 10

2 Ejecución y comunicación básica: nodos, datos y eventos 13
2.1 Principios teóricos . . . 13
Modelos de ejecución reactivos . . . 13
Flujo de datos, eventos y callbacks . . . 15
2.2 Aplicación . . . 18
Nodos . . . 18
Publishers y subscribers . . . 19
Topics . . . 21
Primeros mensajes . . . 22
Servicios y acciones . . . 24

3 Percepción robótica y representación espacial 27
3.1 Principios teóricos . . . 27
Sensores crudos . . . 27
Percepción como transformación de datos . . . 29
Marco espacial y referencias . . . 30
Fusión sensorial básica . . . 32
3.2 Aplicación . . . 33
Gestión de sensores en ROS 2 . . . 33
Sistema de transformaciones en ROS 2 . . . 35
Sensores habituales en ROS 2 . . . 39
De sensores a detecciones: reducción de complejidad . . . 46

4 Arquitecturas deliberativas e híbridas: capas y modelos del mundo 50
4.1 Principios teóricos . . . 50
De lo reactivo a lo deliberativo e híbrido . . . 50
Arquitecturas en capas: misión, tarea y capacidad . . . 52
Aproximaciones por comportamientos: subsumption y arbitraje reactivo . . . 55
Modelos del mundo . . . 56
4.2 Aplicación . . . 57
Mapeo de capas a un grafo de ROS 2 . . . 57
Lifecycle Nodes y subsumption . . . 59
Construcción del modelo del mundo en ROS 2 . . . 61

5 Máquinas de estados finitos **65**
5.1 Principios teóricos . . . 65
La máquina de estados como arquitectura de decisión . . . 65
Anatomía de un estado en robótica . . . 66
Semántica de una transición . . . 67
Topologías . . . 68
El problema de la concurrencia . . . 70
Modelado de misiones y tareas . . . 72
Ventajas y problemas de escalabilidad . . . 72
Máquinas de estados jerárquicas . . . 73
5.2 Aplicación . . . 73
FSM a nivel de tarea . . . 74
FSM a nivel de misión . . . 75
Diseño compositivo: FSM de misión y tarea . . . 76
5.3 Conclusiones . . . 76

6 Árboles de comportamiento **78**
6.1 Principios teóricos . . . 78
Origen histórico: de los videojuegos a la robótica . . . 78
Motivación de los Behavior Trees . . . 79
Anatomía de un Behavior Tree . . . 80
Nodos de control fundamentales . . . 82
Decoradores . . . 88
Modelado de misiones y tareas . . . 89
Reutilización de comportamientos . . . 90
Comparación con FSM . . . 91
6.2 Aplicación . . . 92
Behavior Trees en ROS 2 . . . 92
Blackboard y puertos . . . 93
Ejemplo ilustrativo e integración con capacidades . . . 94

7 Análisis de arquitecturas de referencia y Deep ROS 2 **97**
7.1 Arquitecturas de referencia en ROS 2 . . . 97
Nav2 (Navigation2) . . . 97
PlanSys2 . . . 101
EasyNav (EasyNavigation) . . . 105
7.2 Síntesis de criterios arquitectónicos transversales . . . 108
Patrones arquitectónicos y antipatrones . . . 108
Cómo evaluar una arquitectura robótica . . . 109
Testing, validación y depuración arquitectónica . . . 110
Gestión de errores y tolerancia a fallos . . . 111
Arquitecturas distribuidas reales . . . 111
Seguridad y *safety* . . . 112
Diseño de interfaces y contratos . . . 113
Configuración y despliegue . . . 113
Arquitectura y sistema operativo . . . 114
Ingeniería y evaluación comparativa de arquitecturas robóticas . . . 114
7.3 Deep ROS 2 . . . 115
Diseño de ROS 2 . . . 115
Gestión de ejecución en ROS 2 . . . 117
Executors en práctica: un ejemplo de *starvation* . . . 119
Callback groups . . . 122
Tiempo real en ROS 2: conceptos y latencias . . . 125
Planificador del sistema operativo (Linux) y `SCHED_FIFO` . . . 126
Estrategias de tiempo real en ROS 2 con executors y callback groups . . . 127

Proyectos de Arquitecturas Software para Robots 136

1 Entorno de desarrollo en ROS 2 **137**
1.1 Objetivo . . . 138
1.2 Guión de desarrollo . . . 138
1.3 Teoría y herramientas para el ejercicio . . . 139
Workspace, underlay y overlay . . . 139
Activación del entorno y persistencia . . . 140
Compilación del workspace y opción `-symlink-install` . . . 140
Nodos y su ejecución: `ros2 run` vs `ros2 launch` . . . 141
Interfaces y parámetros . . . 141
Herramientas de línea de comandos (ros2cli) . . . 141
Instalación y ejecución del simulador del Kobuki . . . 145
1.4 Errores comunes y resolución rápida . . . 146
Olvido de activar el entorno . . . 146
Dependencias no instaladas . . . 147
Confusión de terminales . . . 147

2 Ejecución reactiva dirigida por eventos **148**
2.1 Objetivo . . . 149
2.2 Guión de desarrollo . . . 149
2.3 Teoría y herramientas para el ejercicio . . . 151
Estructura del paquete `ch2_examples` . . . 151
El nodo `LoggerNode`: timers y logging . . . 152
El nodo `PublisherNode`: publicación periódica . . . 153
El nodo `SubscriberNode`: callbacks por evento . . . 154
Varios nodos en el mismo proceso: el papel del *executor* . . . 154

3 Percepción, seguimiento y evitación **156**
3.1 Objetivo . . . 157
3.2 Guión de desarrollo . . . 157
3.3 Teoría y herramientas para el ejercicio . . . 159
Launch files y despliegue . . . 159
Configuración mediante parámetros . . . 160
Procesado de `sensor_msgs/msg/LaserScan` . . . 161
Publicación y consulta de TF . . . 162
Procesado de imágenes . . . 164
Reconstrucción 3D con imagen de profundidad . . . 167
Reconstrucción 3D con `PointCloud2` . . . 169

4 Coordinación de misiones con FSM **170**
4.1 Objetivo . . . 171
4.2 Guión de desarrollo . . . 171
4.3 Teoría y herramientas para el ejercicio . . . 174
Implementación de FSM en C++ para ROS 2 . . . 174
Capacidad vs misión . . . 184
Interfaz de capacidad: patrón general y ejemplo con NavigateToPose . . . 184

5 Coordinación de misiones con Behavior Trees **193**
5.1 Objetivo . . . 194
5.2 Guión de desarrollo . . . 194
5.3 Teoría y herramientas para el ejercicio . . . 196
Asignación de comportamiento a arquitectura . . . 196
Behavior Trees: secuencias, selectores y nodos hoja . . . 196
Nodos reutilizables: patrones mínimos . . . 196

5.4 Ejemplo de referencia: Bump-and-Go con Behavior Trees . . . 197
Diseño del árbol . . . 197
Comunicación entre nodos mediante puertos y blackboard . . . 198
Organización del paquete de referencia . . . 198
Implementación: Definición XML del árbol . . . 199
Implementación: Nodos personalizados . . . 201
Resumen de conceptos clave . . . 208
5.5 Ejemplo de capacidad de interacción . . . 209

6 Arquitectura Misión–Tarea–Capacidad **214**
6.1 Objetivo . . . 215
6.2 Guión de desarrollo . . . 215
6.3 Teoría y herramientas para el ejercicio . . . 216
Asignación de modelos por nivel . . . 216
Pila de implementación y criterios prácticos . . . 216
6.4 Ejemplo de referencia: `mission_task_example` . . . 217
Arquitectura del ejemplo de referencia . . . 217
Patrón reutilizable: TaskLifecycleNode . . . 217
Patrón genérico: BTTaskNode . . . 218
Coordinación FSM: MissionExecutor . . . 219
Programa principal: instanciación y ejecución . . . 220
Archivos XML de Behavior Trees . . . 221
Compilación y ejecución del ejemplo . . . 221

Appendix **222**

Bibliography **223**

Alphabetical Index **224**

Índice de figuras

1.1 Requisitos funcionales y no funcionales. 2
1.2 Componentes funcionales compuestos en jerarquía y composición, intercambiando mensajes. . 3
1.3 Procesos de percepción, decisión y actuación en un robot. 4
1.4 Subsistemas clásicos en un sistema robótico. 4
1.5 Implementación de la arquitectura AiboWare sobre Open-R (imagen de https://www.aibohack.com/openr_sdk/index.html). 6
1.6 NaoQi como middleware para robots Nao y Pepper (imagen de http://doc.aldebaran.com/1-14/getting_started/software_in_and_out.html). 7
1.7 Player/Stage como middleware y simulador para robótica móvil (imagen de https://playerstage.sourceforge.net/stage/stage.html). 7
1.8 YARP como middleware para robótica (imagen de http://www.yarp.it/). 8
1.9 Orocos como middleware para control robótico en tiempo real (imagen de https://www.orocos.org/). 8
1.10 Logo de ROS 2, el estandar *de facto* en programación de robots. 8
1.11 Grafo de computación de un sistema ROS 2. (Fig.de [1][2]) 10
1.12 Arquitectura software de un subsistema de percepción (Fig.de [1][2]) 10
1.13 Arquitectura software varios subsistemas (Fig.de [1][2]) 11

2.1 Ejecución iterativa. 13
2.2 Ejecución dirigida por eventos. 14
2.3 Componentes distribuidos que reaccionan de forma autónoma. 15
2.4 Un nodo ROS 2 como proceso reactivo, donde el usuario define callbacks para eventos específicos. La gestión de los eventos y la ejecución de los callbacks la realiza la infraestructura de ROS 2. Aunque no es exactamente así en la realidad, podemos entender que el círculo amarillo representa el mecanismo que despacha los mensajes a los callback, del que ya daremos detalles en próximos capítulos. 18
2.5 Sistema compuesto por distintos nodos con funciones claras que colaboran para percibir, deliberar y actuar. 18
2.6 Sistema distribuido entre varias máquinas y procesos distintos. 19
2.7 Publicadores y suscriptores en un nodo ROS 2. 20
2.8 Topics como canales de comunicación entre nodos. 21
2.9 Composición de mensajes. 23
2.10 Mecanismo de servicios en ROS 2. 24
2.11 Mecanismo de acciones en ROS 2. 25

3.1 Robot iCreate equipado con un laser 2D y con una cámara RGB-D (*RGB-Depth*). 27
3.2 Calibración de los parámetros intrínsecos de una cámara monocular (imagen de https://docs.ros.org/en/jazzy/p/camera_calibration/doc/tutorial_mono.html). 28
3.3 Un mismo punto P tiene coordenadas diferentes según el marco de referencia en el que se exprese. 30
3.4 Fusión sensorial de imágenes y LIDAR (imagen de https://github.com/CDonosoK/ros2_camera_lidar_fusion). 32
3.5 Sistema de TFs con sus topics, y nodos que lo usan a través de la infraestructura común. 35
3.6 Marcos de referencia y transformaciones en un robot móvil (izquierda) y convención dextrógira (derecha). 36
3.7 Convenciones estándar para la organización del árbol de transformaciones en robots móviles según REP 105. 37
3.8 El nodo Simulador y Localizador publican TFs en diferentes instantes temporales. Cuando se solicita una transformación en un instante intermedio, el sistema interpola cada tramo del camino para obtener una consulta coherente. 38

3.9 Modelo pin-hole de formación de imagen. Un punto 3D L se proyecta sobre el plano de imagen en un punto a través del centro óptico C. 42
3.10 Modelo HSV de coloración. Un punto en el espacio de color se representa mediante los componentes Matiz, Saturación y Valor. (imagen adaptada de `https://es.wikipedia.org/wiki/Modelo_de_color_HSV`). 44
3.11 PointCloud2 representa una nube de puntos 3D, donde cada punto puede contener coordenadas y atributos adicionales como intensidad o color. 45
3.12 Tres ejemplos de pipelines de percepción que transforman datos sensoriales crudos en detecciones operativas. 49

4.1 Modelos deliberativos: el robot mantiene una representación interna sobre la que planifica acciones. 50
4.2 Roles y responsabilidades en un sistema híbrido: gestor de objetivos, deliberador/planificador, ejecutor y monitor. 51
4.3 Arquitectura en capas: misión, tarea y capacidad. Cada capa opera con un horizonte y un ritmo diferentes, manteniendo un modelo del mundo consistente. 53
4.4 Arquitectura en capas con interfaces entre ellas. 54
4.5 Los comportamientos emergen de la activación de comportamientos (en verde) y la supresión/inhibición de otros (en rojo). 55
4.6 Arquitectura en capas interactiva con modelo del mundo. 56
4.7 Arquitectura en capas implementada en ROS con máquinas de estados para la capa de misión, Behavior Trees para tareas y *frameworks* para capacidades. 58
4.8 Arquitectura en capas implementada en ROS con PlanSys2 para la capa de misión, Behavior Trees para tareas y *frameworks* para capacidades. 59
4.9 LifeCycle Node: estados y transiciones. 60
4.10 Nodo de ROS 2 que implementa el modelo del mundo, mostrando las interfaces de estado publicado y consultas. 61
4.11 Modelo del mundo basado en vistas federadas. 63
4.12 Modelo del mundo basado en modelo dual (geométrico + simbólico) con traductor. 64

5.1 Ejemplo de FSM para un robot de patrullaje con gestión de batería. Los estados representan modos de operación, las transiciones indican eventos o condiciones, y las guardas (expresiones booleanas) determinan cuándo ocurre cada cambio. 66
5.2 Ciclo de vida de un estado en una FSM robótica. La fase `on_entry` inicializa recursos, `on_do` se ejecuta cíclicamente monitorizando condiciones de transición, y `on_exit` garantiza la limpieza segura antes de abandonar el estado. 67
5.3 Comparativa de latencia: En Moore, la acción se ejecuta al entrar en el estado en el siguiente ciclo ($T + 1$). En Mealy, la acción ocurre en la transición (T). 68
5.4 Diferencia arquitectónica entre Moore y Mealy. En Moore, la acción `stop_robot()` se ejecuta dentro del estado destino `STOPPED` (ciclo siguiente). En Mealy, la acción forma parte de la transición misma (ciclo actual). 69
5.5 Explosión combinatoria en una FSM tradicional. Para combinar navegación (3 estados) con batería (2 estados), se requieren $3 \times 2 = 6$ estados explícitos. Cada estado codifica información de ambas dimensiones (`IDLE_BAT_OK`, `MOVING_BAT_LOW`, etc.), creando una estructura compleja y difícil de mantener. Añadir una tercera dimensión (ej. luces) multiplicaría el número a 12 estados. 71
5.6 Descomposición ortogonal en dos FSM independientes. La NavigationFSM gestiona la navegación (3 estados) y la BatteryFSM monitoriza la batería (2 estados). Ambas se ejecutan en paralelo conceptual en cada ciclo de control, evitando la explosión combinatoria ($3 + 2 = 5$ estados en lugar de $3 \times 2 = 6$ estados combinados). 71
5.7 Ejemplo de FSM jerárquica para un robot de limpieza. El superestado `CLEANING` (rectángulo morado) encapsula tres subestados. La transición de emergencia por batería baja se define una única vez desde el superestado, aplicándose automáticamente a todos los subestados internos. Esto evita tener que dibujar tres flechas separadas. 73

5.8 FSM a nivel de tarea: `TransportObject`. Coordina la secuencia de estados, algunos de los cuales invocan capacidades del robot (navegación, agarre, liberación) cuando se requieren operaciones complejas. Las transiciones principales (negro sólido) indican flujo nominal, mientras que las transiciones de error (rojo discontinuo) conducen a recuperación. 74
5.9 FSM a nivel de misión: `WarehouseInspection`. Coordina fases de larga duración, cada una ejecutando múltiples tareas. Las transiciones naranjas indican eventos especiales que alteran el flujo, mientras que las rojas discontinuas son abortos de emergencia. 75
5.10 Composición jerárquica en la arquitectura misión–tarea–capacidad. La misión contiene una FSM que activa tareas (flecha morada). Cada tarea tiene su propia FSM que invoca capacidades según sea necesario (flechas azules). Las capacidades son módulos complejos reutilizables, no necesariamente modelados con FSM. 76

6.1 Tipos de nodos en un Behavior Tree y su representación gráfica 81
6.2 Propagación de tick y agregación de resultados en un Sequence 82
6.3 Nodo Sequence: todas las acciones deben completarse en orden 82
6.4 Comparación de comportamiento entre Sequence y ReactiveSequence: el árbol contiene un nodo que verifica la condición de batería (IsBatteryOK?) seguido de una acción de navegación (Navigate). El Sequence estándar mantiene memoria y no reevalúa la condición mientras Navigate está en ejecución, mientras que ReactiveSequence reevalúa la condición en cada tick, permitiendo detectar inmediatamente cuando la batería se agota durante la navegación. 83
6.5 Nodo Fallback: intenta alternativas hasta encontrar una que funcione 84
6.6 Comparación de comportamiento entre Fallback y ReactiveFallback: el árbol verifica si la batería está crítica (BatteryCritical?) antes de continuar con la tarea normal (PerformTask). El Fallback estándar mantiene memoria y no reevalúa BatteryCritical? mientras PerformTask está en ejecución, mientras que ReactiveFallback reevalúa en cada tick, permitiendo interrumpir la tarea inmediatamente cuando la batería se vuelve crítica. 85
6.7 Nodo Parallel: ejecución concurrente con monitorización 86
6.8 Decoradores añadiendo robustez a un árbol básico 89
6.9 Arquitectura de BehaviorTree.CPP integrada con ROS 2 92
6.10 Arquitectura del blackboard y sistema de puertos. Los nodos declaran puertos de entrada (InputPort) y salida (OutputPort) que se conectan al blackboard mediante claves. Un nodo puede escribir un valor en el blackboard y otros nodos pueden leerlo, permitiendo comunicación sin acoplamiento directo. La figura muestra un ejemplo donde un nodo DetectObject escribe la posición del objeto detectado en el blackboard, y posteriormente los nodos NavigateToObject y GraspObject leen esa posición para ejecutar sus acciones. 93
6.11 Behavior Tree para bump-and-go integrando control directo con capacidades Nav2 95

7.1 Arquitectura de Nav2. Fuente: [2]. 98
7.2 Ejemplo de *Behavior Tree* (BT) simple usado por el *BT Navigator* de Nav2. 99
7.3 Ejemplo de costmap global y local en Nav2 (representaciones espaciales por capas usadas por el planificador global y el controlador local). 100
7.4 Arquitectura general de PlanSys2. Fuente: `https://plansys2.github.io/_images/plansys2_arch.png` 102
7.5 Vista ampliada de la arquitectura de PlanSys2. Fuente: `https://plansys2.github.io/_images/plansys2_arch2.png` 103
7.6 Ejemplo de árbol de comportamiento asociado a una acción/plan en PlanSys2. Fuente: `https://plansys2.github.io/_images/action_bt.png` 103
7.7 Protocolo de ejecución y comunicación de acciones en PlanSys2. Fuente: `https://plansys2.github.io/_images/protocol.png` 104
7.8 Diseño general de EasyNav. Fuente: `https://easynavigation.github.io/_images/easynav_simple_design_v2.png` 106
7.9 Combinaciones de plugins en EasyNav. Fuente: `https://easynavigation.github.io/_images/plugin_combinations.png` 106
7.10 Más combinaciones de plugins en EasyNav. Fuente: `https://easynavigation.github.io/_images/plugin_combinations_2.png` 107

7.11 Interfaz TUI de EasyNav. Fuente: https://easynavigation.github.io/_images/easynav_simple_tui.png . . . 107
7.12 Diseño en capas de ROS 2. . . . 116
7.13 Gestión de ejecución por capas en ROS 2 (adaptado de [2]). . . . 118
7.14 Pasos de procesado de mensajes mediante wait-set (adaptado de [2]). . . . 118
7.15 Semántica general de un executor basado en wait-set (adaptado de [2]). . . . 119
7.16 Procesado de un wait-set: un mensaje por cola por ventana (adaptado de [2]). . . . 119
7.17 Semántica del `StaticSingleThreadedExecutor` (adaptado de [2]). . . . 120
7.18 Traza con `SingleThreadedExecutor`: no hay concurrencia entre callbacks (adaptado de [2]). . . . 122
7.19 Traza con `MultiThreadedExecutor`: sin configuración adicional, el nodo no ejecuta callbacks en paralelo y puede haber *starvation* (adaptado de [2]). . . . 122
7.20 Callback groups en un nodo: arriba todo en el grupo por defecto; abajo el timer en un grupo separado (adaptado de [2]). . . . 123
7.21 Ejecución con dos callback groups: el timer puede correr concurrentemente con las suscripciones (adaptado de [2]). . . . 124
7.22 Callback group reentrant con sincronización explícita: puede haber concurrencia, pero se controla con locks (adaptado de [2]). . . . 125
7.23 Ejemplos de casos de uso en el plano severidad/latencia (adaptado de [2]). . . . 125
7.24 Fuentes típicas de latencia en un sistema de control (adaptado de [2]). . . . 126
7.25 Interpretación de `nice` y `rtprio` en políticas `SCHED_OTHER` y `SCHED_FIFO` (adaptado de [2]). . . . 127
7.26 Histograma de periodos entre activaciones sin tiempo real (adaptado de [2]). . . . 127
7.27 Histograma de periodos entre activaciones usando `SCHED_FIFO` (adaptado de [2]). . . . 127
7.28 Caso no deseado: todas las callbacks comparten la misma prioridad (adaptado de [2]). . . . 128
7.29 Idea: callbacks críticas en hilos con mayor prioridad, capaces de preemptar a las no críticas (adaptado de [2]). . . . 128
7.30 Estrategia 1: un executor en tiempo real para nodos críticos (adaptado de [2]). . . . 129
7.31 Beneficios esperados de la estrategia 1: mejorar la regularidad temporal y reducir el tiempo de respuesta. . . . 129
7.32 Estrategia 1: histograma del tiempo entre ejecuciones de timers en nodo no crítico vs. crítico (adaptado de [2]). . . . 130
7.33 Estrategia 1: histograma del tiempo de ejecución de timers (adaptado de [2]). . . . 130
7.34 Estrategia 2: callbacks del mismo nodo en executors distintos mediante callback groups (adaptado de [2]). . . . 131
7.35 Beneficios esperados de la estrategia 2: mejorar la regularidad temporal y reducir el tiempo de respuesta de callbacks seleccionadas. . . . 132
7.36 Estrategia 2: tiempo de ejecución del callback crítico antes/después de elevar prioridad (adaptado de [2]). . . . 132
7.37 Estrategia 3: cadena crítica distribuida (ejemplo tipo freno automático) (adaptado de [2]). . . . 133
7.38 Estrategia 3: latencia extremo-a-extremo antes/después de ejecutar la cadena crítica en tiempo real (adaptado de [2]). . . . 134

1.1 Estructura de underlay y overlay en ROS 2. . . . 140

5.1 Behavior Tree para bump-and-go con control directo y recuperación ante obstáculos . . . 197

Índice de cuadros

6.1 Comparación entre tipos de nodos en Behavior Trees . . . 81
6.2 Comportamiento de nodos de control según resultados de sus hijos . . . 87
6.3 Comparación entre nodos de control estándar y reactivos . . . 88

6.4 Comparación arquitectónica entre FSM y Behavior Trees 91

Teoría de Arquitecturas Software para Robots

1 Arquitectura software en robótica

1.1 Principios teóricos 2
Qué es arquitectura software 2
Qué es arquitectura software en robótica 3
Middlewares de programación de robot 5
1.2 Aplicación 9
ROS 2 como materialización de una arquitectura software robótica 9
Componentes, procesos y comunicación en ROS 2 . . . 10

1.1. Principios teóricos

Qué es arquitectura software

La arquitectura software de un sistema se refiere a la estructura fundamental que organiza dicho sistema, definiendo sus componentes principales, sus responsabilidades, y las relaciones e interacciones entre ellos. Describe cómo se descompone un sistema complejo en partes manejables, cómo se comunican entre sí y bajo qué principios se toman las decisiones de diseño que condicionan su evolución. No se trata de una descripción de bajo nivel ni de la implementación concreta, sino de una visión de alto nivel que establece el marco sobre el que se construye y mantiene el software.

Desde un punto de vista ingenieril, la arquitectura software cumple un papel estratégico: actúa como puente entre los requisitos del sistema y su implementación técnica. A través de la arquitectura se traducen necesidades funcionales y no funcionales (Fig. 1.1) en decisiones estructurales explícitas. Por ello, una arquitectura no es simplemente un diagrama estático, sino un conjunto de decisiones fundamentales que restringen y guían el diseño del sistema a lo largo de su ciclo de vida.

La arquitectura traduce requisitos funcionales y no funcionales en decisiones estructurales.

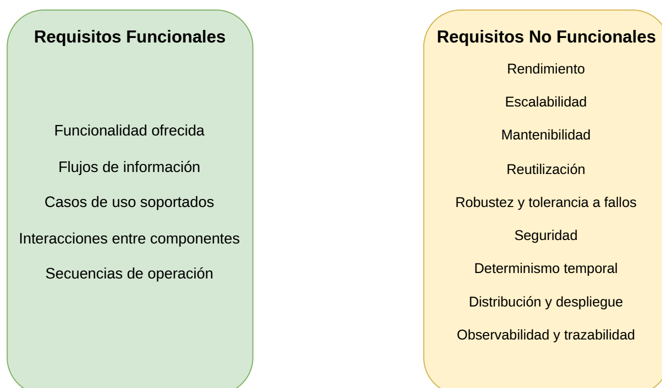


Figura 1.1: Requisitos funcionales y no funcionales.

Una arquitectura software se compone, de manera general, de varios elementos clave. En primer lugar, los componentes o unidades estructurales

Una arquitectura se define por componentes, mecanismos de interacción y principios organizativos.

(Fig. 1.2), que encapsulan funcionalidades concretas y delimitan responsabilidades claras. En segundo lugar, los mecanismos de interacción, que definen cómo se comunican dichos componentes (llamadas directas, intercambio de mensajes, eventos, flujos de datos, etc.). En tercer lugar, los principios y patrones arquitectónicos que gobiernan la organización global del sistema, como la separación de responsabilidades, el desacoplamiento, la modularidad o la jerarquía. Estos elementos permiten razonar sobre el sistema sin necesidad de conocer los detalles de cada algoritmo.

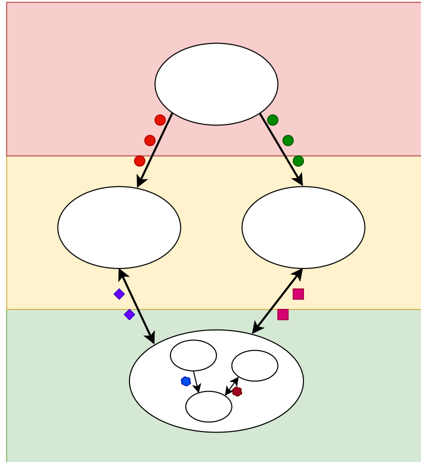


Figura 1.2: Componentes funcionales compuestos en jerarquía y composición, intercambiando mensajes.

Para un ingeniero, y en particular para un ingeniero de robótica, resulta esencial comprender que la arquitectura software condiciona de forma directa la capacidad del sistema para adaptarse al cambio. Decisiones arquitectónicas tempranas influyen en aspectos como la facilidad para integrar nuevos componentes, la reutilización de código, la posibilidad de realizar pruebas aisladas o la gestión de fallos. Una arquitectura bien diseñada facilita la evolución incremental del sistema y reduce el coste de mantenimiento, mientras que una arquitectura deficiente tiende a generar sistemas rígidos, difíciles de extender y propensos a errores.

Las decisiones arquitectónicas influyen directamente en la evolución y mantenibilidad del sistema.

Finalmente, es importante destacar que no existe una arquitectura software universalmente correcta. La arquitectura debe ser adecuada al contexto, al dominio de aplicación y a los requisitos del sistema. Diseñar una arquitectura implica evaluar compromisos entre alternativas, priorizar ciertos atributos de calidad frente a otros y asumir restricciones tecnológicas u organizativas. Entender qué problemas resuelve una arquitectura, qué problemas introduce y bajo qué supuestos ha sido diseñada es una competencia clave para cualquier ingeniero que participe en el desarrollo de sistemas software complejos.

La idoneidad de una arquitectura depende del contexto y de los compromisos asumidos.

Qué es arquitectura software en robótica

La arquitectura software en robótica es la organización estructural del software que gobierna el comportamiento de un robot, definiendo cómo se distribuyen, coordinan e integran los distintos procesos de percepción, decisión y actuación (Fig. 1.3) que permiten al sistema interactuar con el mundo físico. A diferencia de otros sistemas software, un robot no opera únicamente sobre información digital, sino que debe enfrentarse de manera continua a un entorno dinámico, incierto y parcialmente

La arquitectura software robótica estructura los procesos que permiten al robot percibir, decidir y actuar.

observable, lo que convierte a la arquitectura en un elemento crítico para garantizar un comportamiento coherente, robusto y seguro.

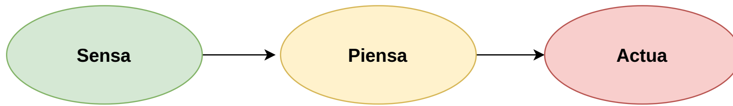


Figura 1.3: Procesos de percepción, decisión y actuación en un robot.

Las arquitecturas software para robots heredan los principios generales de la ingeniería del software —modularidad, separación de responsabilidades, desacoplamiento y reutilización—, pero los adaptan a un dominio con requisitos especialmente exigentes. En robótica, los componentes software no son independientes del tiempo ni del espacio: procesan datos sensoriales con caducidad temporal, generan decisiones bajo restricciones de latencia y controlan actuadores físicos cuyas acciones tienen consecuencias irreversibles. Por ello, la arquitectura debe gestionar explícitamente aspectos como la sincronización temporal, la concurrencia, la priorización de tareas y la tolerancia a fallos.

Los principios clásicos de arquitectura deben adaptarse a las particularidades físicas y temporales de la robótica.

Una característica distintiva de las arquitecturas robóticas es su estructura funcional basada en grandes bloques conceptuales. De forma clásica, un sistema robótico se organiza en subsistemas de percepción, modelado del entorno, planificación, control y supervisión, (Fig 1.4) cada uno con responsabilidades bien definidas pero altamente interdependientes. La arquitectura establece cómo fluye la información entre estos subsistemas, qué representaciones se comparten, qué decisiones se toman de manera reactiva y cuáles requieren deliberación, y cómo se coordinan comportamientos de distinta naturaleza y frecuencia.

Las arquitecturas robóticas se organizan clásicamente en subsistemas funcionales interdependientes.

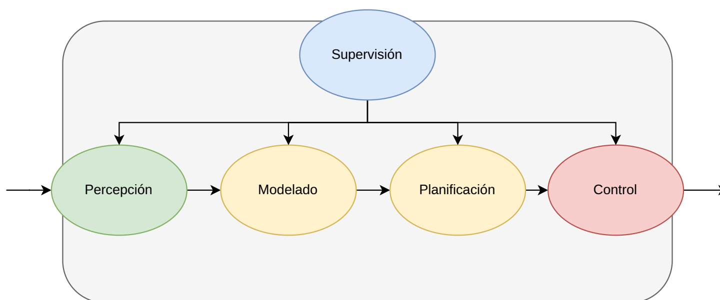


Figura 1.4: Subsistemas clásicos en un sistema robótico.

Otro rasgo fundamental es la necesidad de concurrencia y distribución. Un robot moderno ejecuta múltiples procesos en paralelo: adquisición de sensores, estimación del estado, generación de trayectorias, control de bajo nivel, comunicaciones y monitorización. La arquitectura debe permitir que estos procesos se ejecuten de forma concurrente, potencialmente distribuidos en varios nodos de cómputo, manteniendo coherencia en los datos y garantizando propiedades temporales mínimas. Esto introduce retos adicionales relacionados con la comunicación, la sincronización y la gestión de recursos computacionales.

La arquitectura debe gestionar la ejecución concurrente y distribuida de múltiples procesos robóticos.

Además, las arquitecturas software en robótica deben ser adaptables y extensibles. Los robots suelen evolucionar a lo largo de su vida útil: se añaden nuevos sensores, se sustituyen algoritmos, se modifican comportamientos o se integran capacidades adicionales. Una arquitectura adecuada facilita estas modificaciones sin necesidad de reescribir el sistema completo, promoviendo la reutilización de componentes y la experimentación con distintas soluciones algorítmicas. Esta capacidad resulta especialmente relevante en entornos de investigación y desarrollo, pero también en sistemas industriales y de servicio.

La adaptabilidad es clave para la evolución funcional de los sistemas robóticos.

Un aspecto cada vez más relevante en las arquitecturas software para robots es la incorporación explícita de mecanismos de monitorización del estado interno del sistema y de su interacción con el entorno. Un robot no solo debe actuar, sino también ser capaz de conocer y exponer su propio estado: qué sensores están operativos, qué decisiones se están tomando, qué componentes presentan degradaciones y qué objetivos se encuentran activos en cada momento. Desde el punto de vista arquitectónico, esto implica definir canales de observación, estados internos bien estructurados y mecanismos de supervisión que permitan detectar fallos, incoherencias o situaciones anómalas durante la ejecución.

La arquitectura debe permitir monitorizar explícitamente el estado interno y operativo del robot.

Relacionado con lo anterior, la trazabilidad de las decisiones se ha convertido en un requisito fundamental, especialmente en sistemas autónomos complejos. La arquitectura debe permitir reconstruir por qué el robot ha tomado una determinada decisión, qué información sensorial la ha motivado y qué componentes han intervenido en el proceso. Esta trazabilidad se incorpora mediante el diseño de flujos de datos explícitos, el registro estructurado de eventos y decisiones, y la separación clara entre percepción, deliberación y actuación. Más allá de su utilidad para depuración y validación, estos mecanismos son esenciales para la explicabilidad del comportamiento del robot, el cumplimiento de requisitos de seguridad y la evolución controlada del sistema.

La arquitectura debe permitir reconstruir y explicar las decisiones del robot.

Por último, la arquitectura software en robótica cumple una función integradora entre el nivel algorítmico y el nivel de ejecución real. No basta con disponer de algoritmos correctos; es necesario que estos puedan convivir en un sistema que funcione de manera continua, bajo incertidumbre y con interacción física. La arquitectura define los contratos entre componentes, los flujos de datos, los mecanismos de coordinación y los puntos de control del sistema, convirtiéndose en el marco que hace posible que un conjunto de algoritmos aislados se transforme en un robot operativo.

La arquitectura integra algoritmos en un sistema robótico operativo y coherente.

Middlewares de programación de robot

El desarrollo de sistemas robóticos complejos hace inviable construir todo el software desde cero para cada plataforma o aplicación. La necesidad de integrar múltiples componentes concurrentes, gestionar comunicaciones, abstraer el hardware y mantener propiedades temporales mínimas conduce de manera natural al uso de middlewares de programación de robots. Un middleware actúa como una capa intermedia entre la arquitectura software del sistema y la infraestructura subyacente, proporcionando mecanismos comunes que permiten implementar, desplegar y coordinar los distintos componentes del robot de forma estructurada y reutilizable.

Los sistemas robóticos complejos requieren infraestructuras comunes que eviten desarrollos ad hoc.

Desde el punto de vista arquitectónico, el middleware no sustituye a la arquitectura software, sino que la materializa. Mientras que la arquitectura define qué componentes existen, qué responsabilidades tienen y cómo deben interactuar, el middleware proporciona los mecanismos concretos para que esas interacciones sean posibles en tiempo de ejecución. En robótica, esta relación es especialmente estrecha: muchas decisiones arquitectónicas —como el grado de desacoplamiento entre módulos, el modelo de concurrencia o el tipo de comunicación— vienen condicionadas por las capacidades y restricciones del middleware elegido.

El middleware proporciona los mecanismos de ejecución que hacen operativa la arquitectura software.

De manera general, un middleware de programación de robots suele componerse de varios elementos fundamentales. En primer lugar, mecanismos de comunicación, que permiten el intercambio de datos entre componentes mediante mensajes, servicios o flujos de eventos. En segundo lugar, abstracciones de hardware, que desacoplan el software de los detalles concretos de sensores y actuadores. En tercer lugar, modelos de ejecución y concurrencia, que determinan cómo se planifican las tareas y cómo se gestionan procesos paralelos. Finalmente, muchos middlewares incorporan herramientas de configuración, despliegue y depuración, esenciales para el desarrollo y mantenimiento de sistemas robóticos reales.

Los middlewares robóticos integran comunicación, abstracción hardware y modelos de ejecución.

A lo largo de la historia de la robótica han surgido distintos middlewares, cada uno reflejando las necesidades, limitaciones y supuestos de su contexto tecnológico:

- **Open-R (Sony Aibo).**
  Open-R es un middleware diseñado específicamente para sistemas robóticos embebidos con fuertes requisitos de tiempo real. El software se estructura en *OPEN-R objects*, donde cada objeto es una entidad concurrente que ejecuta su propio *thread*. La comunicación entre objetos se realiza exclusivamente mediante paso de mensajes tipados a través de canales unidireccionales, con un protocolo explícito de sincronización que permite controlar cuándo un componente está preparado para recibir nuevos datos.

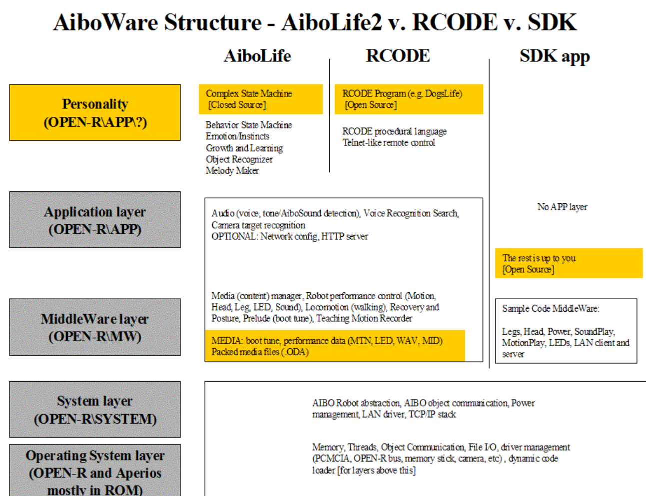


Figura 1.5: Implementación de la arquitectura AiboWare sobre Open-R (imagen de `https://www.aibohack.com/openr_sdk/index.html`).

  Desde el punto de vista arquitectónico, Open-R impone un modelo muy estricto y determinista: la ejecución está organizada en ciclos temporales discretos y existe una separación clara entre objetos de aplicación y objetos que interfazan directamente con el hardware. Esto favorece arquitecturas altamente predecibles, pero también rígidas, donde la estructura del sistema y su comportamiento temporal están fuertemente condicionados por el middleware.
- **NaoQi (Nao y Pepper).** NaoQi es un middleware orientado a la programación de robots humanoides mediante el uso de módulos preexistentes. La mayor parte de las funcionalidades del robot se ofrecen como servicios ya implementados, a los que los programas de usuario acceden mediante *proxies*. Estos proxies permiten invocar métodos, suscribirse a eventos y acceder al estado del robot sin interactuar directamente con sensores o actuadores.
  Internamente, NaoQi mantiene una *blackboard* compartida donde los distintos módulos publican y consumen información. El mo-

delo de ejecución es multihilo, pero la gestión de la concurrencia queda en gran medida oculta al desarrollador. Herramientas como *Choregraphe* permiten definir comportamientos de alto nivel, mientras que para desarrollos más avanzados se emplean las APIs en C++ o Python. Arquitectónicamente, NaoQi facilita el desarrollo rápido, pero limita la libertad para redefinir la estructura interna del sistema.

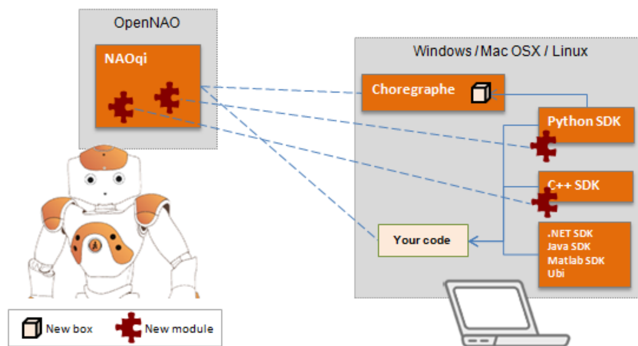


Figura 1.6: NaoQi como middleware para robots Nao y Pepper (imagen de `http://doc.aldebaran.com/1-14/getting_started/software_in_and_out.html`).

- **Player/Stage.** Player/Stage adopta un modelo cliente–servidor claramente definido. Player actúa como un servidor que expone interfaces estandarizadas para sensores y actuadores, mientras que los algoritmos se implementan como clientes independientes que se conectan a dichos servicios. Stage complementa este enfoque proporcionando un simulador 2D ligero que permite ejecutar el mismo software tanto en simulación como en el robot real.

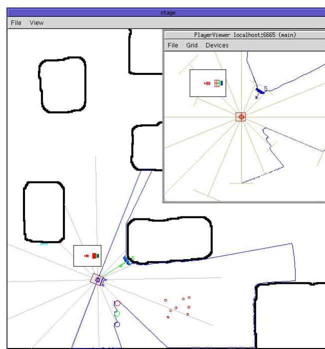


Figura 1.7: Player/Stage como middleware y simulador para robótica móvil (imagen de `https://playerstage.sourceforge.net/stage/stage.html`).

Desde una perspectiva arquitectónica, Player/Stage promueve una separación clara entre hardware y algoritmos de alto nivel, favoreciendo sistemas modulares y reutilizables. Sin embargo, su modelo de comunicación y sus capacidades de concurrencia resultan limitados para arquitecturas robóticas complejas o altamente distribuidas.

- **YARP (Yet Another Robot Platform).** YARP es un middleware orientado explícitamente a flujos de datos y a la interconexión flexible de componentes. El elemento central del sistema es el *port*, un canal de comunicación tipado que permite conectar módulos de forma dinámica, incluso en tiempo de ejecución. YARP no impone una arquitectura funcional fija ni una jerarquía de componentes.
Arquitectónicamente, YARP favorece sistemas débilmente acoplados, donde la topología de comunicación forma parte de la propia arquitectura. Es especialmente utilizado en robótica humanoide y cognitiva, donde múltiples procesos intercambian flujos continuos de información. Esta flexibilidad ofrece gran libertad de diseño, a costa de una menor estandarización.

Figura 1.8: YARP como middleware para robótica (imagen de `http://www.yarp.it/`).

- **Orocos (Open Robot Control Software).** Orocos es un middleware orientado al control robótico con requisitos estrictos de tiempo real. Su núcleo, el *Real-Time Toolkit*, proporciona componentes con interfaces bien definidas, puertos de datos y puntos de ejecución que se integran directamente con planificadores de tiempo real.

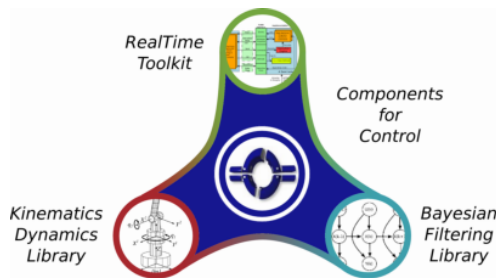


Figura 1.9: Orocos como middleware para control robótico en tiempo real (imagen de `https://www.orocos.org/`).

Desde el punto de vista arquitectónico, Orocos condiciona fuertemente el diseño del sistema: los componentes están pensados para ejecutar tareas periódicas con latencias y jitter controlados. Esto lo hace especialmente adecuado para capas de control de bajo nivel, aunque menos flexible para arquitecturas robóticas de alto nivel o de carácter exploratorio.

ROS (*Robot Operating System*) y su evolución ROS 2 se han consolidado como el *estándar de facto* para el desarrollo de software robótico a nivel internacional. Más que un framework monolítico, ROS constituye un ecosistema de herramientas, librerías y convenciones que permiten construir sistemas robóticos complejos de forma modular, reutilizable y distribuida. Su adopción masiva en investigación, industria y educación ha hecho que muchas arquitecturas software en robótica se diseñen hoy directamente en términos de los conceptos que ROS proporciona.

ROS y ROS 2 constituyen el estándar dominante para el desarrollo de software robótico.

Figura 1.10: Logo de ROS 2, el estandar *de facto* en programación de robots.

ROS 2 surge como respuesta a las limitaciones arquitectónicas de ROS en entornos reales y productivos. Introduce soporte nativo para sistemas distribuidos, multi-robot y de tiempo real, basándose en tecnologías estándar de comunicaciones como DDS. Esto permite definir políticas de calidad de servicio, controlar el comportamiento temporal del sistema y desplegar arquitecturas más robustas y escalables. En este sentido, ROS 2 no solo amplía las capacidades técnicas del middleware, sino que refuerza su papel como infraestructura arquitectónica para robots modernos.

ROS 2 amplía el modelo original para soportar sistemas distribuidos, robustos y en tiempo real.

Un factor clave en el éxito de ROS y ROS 2 es su comunidad. Existe un ecosistema muy amplio de paquetes reutilizables que cubren percepción, navegación, manipulación, simulación, planificación y control, lo que permite construir sistemas complejos combinando componentes existentes. Esta reutilización influye directamente en la arquitectura de los robots, ya que muchas decisiones estructurales vienen condicionadas por la integración de estos paquetes y por las convenciones establecidas por la comunidad.

La comunidad y el ecosistema de paquetes condicionan y facilitan el diseño arquitectónico en ROS.

En la práctica, ROS 2 se utiliza en una gran variedad de aplicaciones, que van desde robótica móvil y manipuladores industriales hasta vehículos autónomos, robots de servicio y plataformas de investigación avanzada. Su flexibilidad arquitectónica permite adaptar el mismo middleware a robots muy distintos, desde prototipos académicos hasta sistemas comerciales. Por ello, comprender ROS 2 desde una perspectiva arquitectónica resulta fundamental para diseñar, analizar y mantener sistemas robóticos actuales.

ROS 2 se aplica a un amplio abanico de robots y contextos con requisitos arquitectónicos diversos.

1.2. Aplicación

ROS 2 como materialización de una arquitectura software robótica

ROS 2 se ha consolidado como la infraestructura de referencia para implementar arquitecturas software en robótica, actuando como una capa intermedia que conecta el diseño arquitectónico con la ejecución real del sistema. Desde esta perspectiva, ROS 2 no debe entenderse como una arquitectura funcional cerrada, sino como un conjunto de mecanismos que permiten materializar decisiones arquitectónicas previamente definidas. Es el ingeniero quien decide cómo se descompone el sistema, qué componentes existen y cómo interactúan, mientras que ROS 2 proporciona las abstracciones necesarias para llevar esas decisiones a la práctica.

ROS 2 actúa como infraestructura que materializa decisiones arquitectónicas en sistemas robóticos reales.

Desde el punto de vista arquitectónico, ROS introduce un modelo basado en procesos independientes, denominados *nodos*, que se comunican mediante mecanismos explícitos como *topics*, *servicios* y *acciones*. Este modelo favorece la separación de responsabilidades, el desacoplamiento entre componentes y la ejecución concurrente, alineándose de manera natural con los principios generales de la arquitectura software en robótica. A diferencia de otros middlewares más cerrados, ROS no impone una arquitectura funcional concreta, sino que proporciona las primitivas necesarias para que el ingeniero defina la estructura del sistema.

ROS propone un modelo arquitectónico basado en nodos desacoplados que se comunican explícitamente.

Una característica fundamental de ROS 2 es que integra de forma natural los requisitos propios de los sistemas robóticos. La concurrencia, la distribución y la gestión del tiempo no se tratan como aspectos secundarios, sino como elementos centrales del modelo de ejecución. Esto permite diseñar arquitecturas que operan de forma continua, reaccionan a eventos del entorno y se distribuyen entre múltiples procesos o dispositivos, manteniendo una coherencia global. En este sentido, ROS 2 actúa como un habilitador de arquitecturas robustas y escalables.

ROS 2 integra concurrencia, distribución y gestión del tiempo como elementos arquitectónicos centrales.

Finalmente, ROS 2 incorpora mecanismos que facilitan la observabilidad del sistema y la trazabilidad de su comportamiento, aspectos clave en arquitecturas robóticas modernas. La posibilidad de inspeccionar el estado de los componentes, analizar las interacciones entre ellos y

ROS 2 incorpora observabilidad y trazabilidad como capacidades arquitectónicas explícitas.

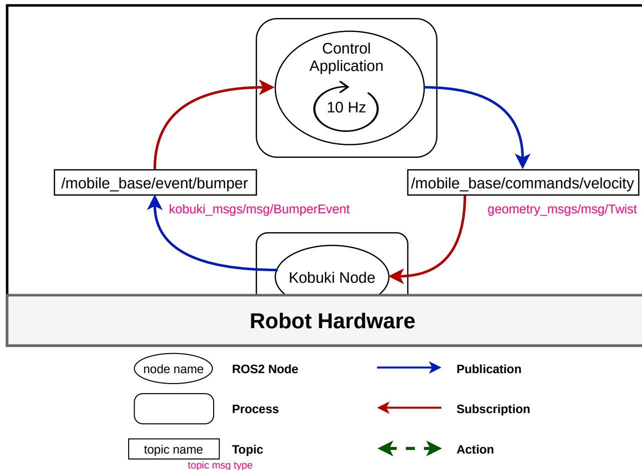


Figura 1.11: Grafo de computación de un sistema ROS 2. (Fig.de [1][2])

registrar eventos relevantes convierte a la arquitectura en un elemento explícito y verificable, no solo en una abstracción de diseño. De este modo, ROS 2 permite que las decisiones arquitectónicas se reflejen claramente en el sistema en ejecución, cerrando el ciclo entre teoría, diseño e implementación.

Componentes, procesos y comunicación en ROS 2

En ROS 2, la descomposición estructural de un sistema robótico se articula fundamentalmente en torno a los conceptos de nodo y proceso. El nodo constituye la unidad arquitectónica básica, encapsulando una funcionalidad concreta y delimitando claramente sus responsabilidades. Desde el punto de vista arquitectónico, diseñar un sistema en ROS 2 implica decidir qué responsabilidades se asignan a cada nodo y cómo se distribuyen estas responsabilidades entre distintos procesos de ejecución.

La estructura de un sistema ROS 2 se basa en nodos distribuidos en procesos.

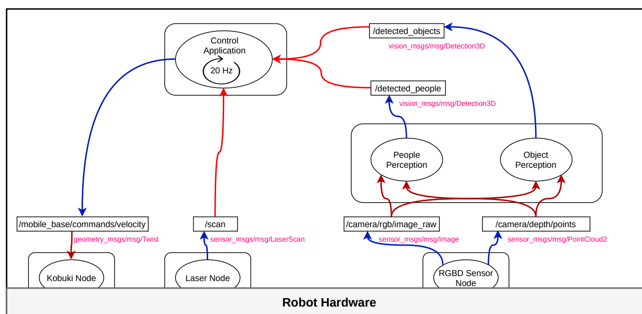


Figura 1.12: Arquitectura software de un subsistema de percepción (Fig.de [1][2])

La separación entre nodos y procesos permite introducir distintos grados de aislamiento y robustez en el sistema. Ejecutar nodos en procesos independientes facilita la detección y contención de fallos, mientras que agrupar varios nodos en un mismo proceso puede reducir latencias y consumo de recursos. ROS 2 no impone una única estrategia, sino que ofrece la flexibilidad necesaria para que estas decisiones se tomen en función de los requisitos del sistema, como el rendimiento, la fiabilidad o las restricciones de despliegue.

La separación entre nodos y procesos permite equilibrar robustez, rendimiento y aislamiento.

La interacción entre componentes en ROS 2 se realiza mediante mecanismos de comunicación explícitos que materializan los flujos de información definidos en la arquitectura. Los topics permiten el intercambio asíncrono

Los mecanismos de comunicación en ROS 2 reflejan decisiones arquitectónicas sobre el flujo de información.

de datos, favoreciendo el desacoplamiento entre productores y consumidores. Los servicios introducen interacciones síncronas orientadas a peticiones puntuales, mientras que las acciones permiten modelar tareas de mayor duración con seguimiento de su progreso. La elección entre estos mecanismos no es únicamente técnica, sino arquitectónica, ya que condiciona la estructura del sistema y la naturaleza de las dependencias entre componentes.

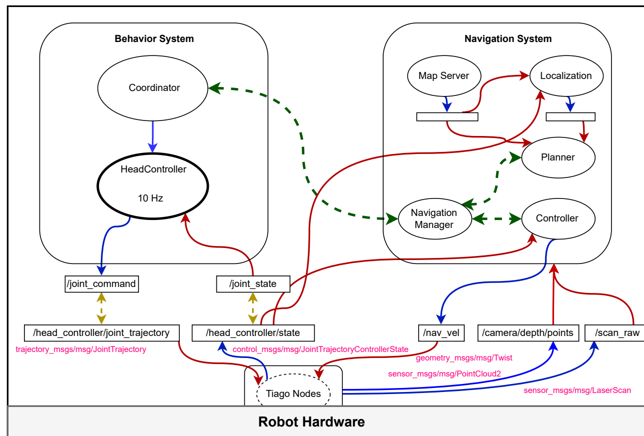


Figura 1.13: Arquitectura software varios subsistemas (Fig.de [1][2])

El comportamiento dinámico de un sistema ROS 2 está gobernado por un modelo de ejecución orientado a eventos. Los nodos reaccionan a la llegada de datos, temporizadores o peticiones externas, lo que se ajusta de forma natural a la naturaleza reactiva de los sistemas robóticos. Este enfoque evita bucles de control monolíticos y facilita la ejecución concurrente de múltiples actividades, como percepción, procesamiento, control y supervisión, dentro de un mismo sistema.

ROS 2 adopta un modelo de ejecución reactivo basado en eventos.

ROS 2 está concebido desde su diseño inicial como un sistema distribuido. Los nodos pueden ejecutarse en procesos y máquinas distintas, comunicándose de forma transparente a través de la red. Esta capacidad permite escalar el sistema, separar cargas computacionales y desplegar arquitecturas multi-robot, pero introduce también retos relacionados con la latencia, la sincronización y la gestión de fallos. Desde el punto de vista arquitectónico, estas cuestiones deben considerarse desde las primeras fases del diseño.

La distribución es una propiedad nativa de las arquitecturas ROS 2.

Un aspecto distintivo de ROS 2 es que la estructura del sistema es explícita y observable en tiempo de ejecución. El middleware mantiene información estructurada sobre nodos, procesos y mecanismos de comunicación, lo que permite inspeccionar el estado del sistema, analizar el grafo de computación y verificar que la arquitectura en ejecución corresponde con el diseño previsto. Esta capacidad resulta esencial para la supervisión, la depuración y la validación de sistemas robóticos complejos.

La estructura explícita del sistema permite observabilidad y supervisión arquitectónica.

Finalmente, la explicitación de la estructura y de las interacciones facilita la trazabilidad del comportamiento del sistema. El registro de eventos, mensajes y estados permite reconstruir la secuencia de decisiones y acciones que conducen a un determinado comportamiento del robot. De este modo, la arquitectura software no solo organiza el sistema, sino que también proporciona el soporte necesario para su comprensión, análisis y evolución a lo largo del tiempo.

La trazabilidad emerge como consecuencia de una arquitectura explícita y observable.

En conjunto, la combinación de nodos, procesos, comunicación explícita, ejecución reactiva y observabilidad convierte a ROS 2 en una infraestructura especialmente adecuada para implementar arquitecturas robóticas modulares, distribuidas y mantenibles. Los principios generales de la ar-

ROS 2 ofrece una base arquitectónica coherente para sistemas robóticos complejos y evolutivos.

quitectura software descritos en la parte teórica se reflejan así de manera directa y tangible en la estructura y el comportamiento de los sistemas desarrollados con ROS 2.

2 Ejecución y comunicación básica: nodos, datos y eventos

2.1. Principios teóricos

2.1 **Principios teóricos** **13**
Modelos de ejecución reactivos 13
Flujo de datos, eventos y callbacks 15
2.2 **Aplicación** **18**
Nodos 18
Publishers y subscribers . . 19
Topics 21
Primeros mensajes 22
Servicios y acciones 24

Modelos de ejecución reactivos

En los sistemas robóticos modernos, la ejecución del software suele organizarse siguiendo un **modelo reactivo**, en el que el comportamiento del sistema emerge como respuesta a **estímulos externos e internos**. Estos estímulos pueden provenir de **sensores** físicos, de **mensajes** emitidos por otros componentes software o de **eventos** generados por el propio sistema, como temporizadores, cambios de estado o detecciones de error. En este enfoque, el robot no sigue un guion rígido previamente establecido, sino que **adapta su comportamiento** continuamente en función de la información que recibe.

La ejecución reactiva hace que el comportamiento emerja de eventos.

A diferencia de los modelos secuenciales clásicos, en los que el flujo de ejecución está completamente determinado de antemano (Fig. 2.1), los modelos reactivos (Fig. 2.2) se basan en la idea de que el **orden de ejecución** no puede predecirse con exactitud. En un programa secuencial tradicional, el desarrollador controla explícitamente qué instrucciones se ejecutan y en qué orden. En robótica, este supuesto rara vez se cumple: los sensores producen datos a ritmos variables, los eventos pueden ocurrir de forma asincrónica y múltiples procesos deben ejecutarse de manera concurrente. El modelo reactivo acepta esta realidad y la convierte en un principio de diseño.

La asincronía y la concurrencia rompen el control secuencial.

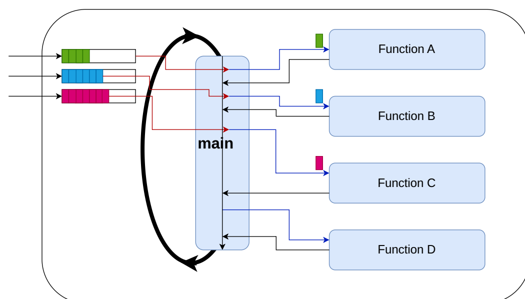


Figura 2.1: Ejecución iterativa.

Desde el punto de vista arquitectónico, un modelo reactivo implica que el sistema está estructurado como un conjunto de **componentes** que permanecen esencialmente **inactivos** hasta que ocurre algo relevante.

Los componentes se activan solo ante estímulos relevantes.

Cuando llega nueva información, el componente correspondiente se activa, **procesa el evento** y, en muchos casos, genera nuevos eventos que activan a otros componentes. El comportamiento global del robot surge entonces de la interacción de estas **reacciones locales**, más que de una **secuencia centralizada** de control.

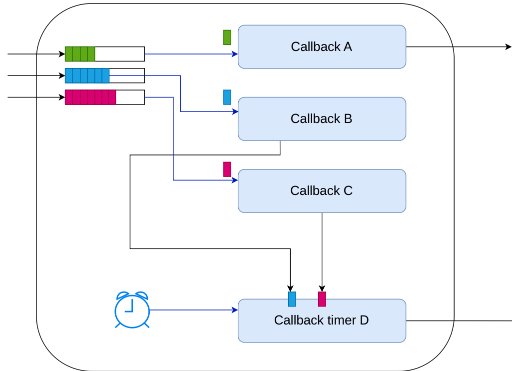


Figura 2.2: Ejecución dirigida por eventos.

Este enfoque resulta especialmente adecuado en robótica porque el entorno físico es **dinámico e impredecible**. Un obstáculo puede aparecer de repente, una persona puede entrar en el campo de visión del robot o un sensor puede dejar de producir datos temporalmente. En un modelo reactivo, estos cambios se traducen en eventos que disparan reacciones inmediatas, sin necesidad de esperar a que un bucle principal alcance un punto concreto de ejecución. Esto permite reducir latencias de respuesta y mejorar la capacidad del sistema para adaptarse a situaciones imprevistas.

El modelo reactivo reduce latencia ante cambios del entorno.

Adoptar un modelo de ejecución reactivo también tiene implicaciones importantes sobre cómo se **distribuye la lógica** del sistema. En lugar de concentrar el control en un único punto, la responsabilidad se reparte entre múltiples componentes que reaccionan de forma autónoma. Esto favorece la modularidad y la escalabilidad, ya que nuevos comportamientos pueden incorporarse añadiendo nuevos componentes reactivos, sin necesidad de reescribir el flujo de ejecución global. Sin embargo, también introduce nuevos retos, como la necesidad de coordinar reacciones concurrentes y evitar comportamientos emergentes no deseados.

La modularidad aumenta, pero exige coordinar reacciones.

Desde una perspectiva de ingeniería, el modelo reactivo obliga a replantear la noción tradicional de control del programa. El diseñador ya no controla directamente cuándo se ejecuta cada parte del código, sino que define bajo qué condiciones debe reaccionar cada componente. Esto requiere un cambio de mentalidad: en lugar de pensar en términos de "qué hace el sistema paso a paso", es necesario pensar en términos de "qué hace el sistema cuando ocurre X". Esta forma de razonar es fundamental para diseñar arquitecturas robóticas robustas y adaptativas.

Se diseña por condiciones de reacción, no por pasos.

Es importante señalar que un modelo reactivo no implica ausencia de estructura. Aunque la ejecución esté gobernada por eventos, la arquitectura debe definir claramente qué tipos de eventos existen, qué componentes reaccionan a ellos y cómo se resuelven posibles conflictos entre reacciones simultáneas. Sin este marco, el sistema puede volverse difícil de entender y de depurar. Por ello, los modelos de ejecución reactivos deben apoyarse siempre en una arquitectura explícita que establezca responsabilidades claras y límites bien definidos.

Sin arquitectura explícita, los eventos generan complejidad.

En resumen, los modelos de ejecución reactivos constituyen una respuesta natural a las características del dominio robótico: **incertidumbre**,

asincronía y **concurrencia**. Comprender estos modelos es esencial para interpretar cómo funcionan los sistemas robóticos modernos y para diseñar arquitecturas capaces de responder de forma eficaz y segura a un entorno cambiante. A lo largo de este capítulo y de los siguientes, se analizará cómo estos modelos se materializan en infraestructuras concretas y qué implicaciones tienen sobre el diseño del software robótico.

El modelo reactivo encaja con incertidumbre y tiempo real.

Es importante destacar que adoptar un modelo de ejecución reactivo no excluye la existencia de **lazos de control periódicos**. Estos lazos pueden entenderse como componentes reactivos activados por eventos temporales y son habituales en niveles bajos de la arquitectura, como en las capacidades de control. Lo que caracteriza al modelo reactivo no es la ausencia de bucles, sino la ausencia de un control secuencial centralizado del sistema.

Los bucles periódicos pueden verse como eventos temporales.

Flujo de datos, eventos y callbacks

El **flujo de datos** constituye uno de los **principios estructurales** más importantes en la arquitectura software de sistemas robóticos. En este enfoque, los componentes se organizan como **productores y consumidores** de información, conectados mediante **canales de comunicación** bien definidos (Fig. 2.3). En lugar de invocar explícitamente a otros componentes, cada módulo **publica** los datos que genera y **reacciona** a los datos que recibe. El comportamiento global del sistema emerge entonces de la **circulación y transformación** continua de información a través de la red de componentes.

El sistema se estructura como productores y consumidores de datos.

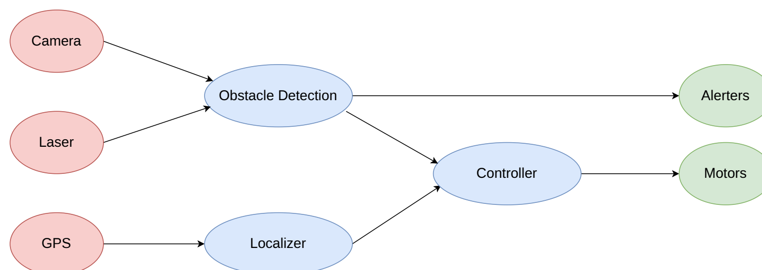


Figura 2.3: Componentes distribuidos que reaccionan de forma autónoma.

Este modelo favorece el **desacoplamiento** entre componentes, ya que los productores de datos no necesitan conocer quién los consume ni con qué finalidad. Un sensor publica medidas, un estimador publica estados, un controlador publica comandos; cada uno cumple su función sin depender de la lógica interna de los demás. Esta separación reduce las dependencias directas y permite modificar o sustituir componentes sin necesidad de reestructurar el sistema completo, siempre que se mantengan las interfaces de datos acordadas.

Los contratos de datos desacoplan a productores y consumidores.

Desde un punto de vista arquitectónico, el flujo de datos impone una forma particular de estructurar el software: los componentes se conciben como **transformadores de información**. Cada componente recibe uno o varios flujos de entrada, realiza algún tipo de procesamiento —filtrado, fusión, estimación, decisión— y produce uno o varios flujos de salida. Este patrón es especialmente natural en robótica, donde gran parte del procesamiento consiste precisamente en transformar datos sensoriales en representaciones cada vez más abstractas, hasta llegar a decisiones de actuación.

Cada componente transforma flujos de entrada en flujos de salida.

Pensar el sistema como una **red de flujos de datos** permite razonar de forma explícita sobre **dependencias** y **acoplamientos**. Es posible

Los flujos hacen visibles dependencias y puntos críticos.

identificar qué componentes dependen de qué información, qué datos son críticos para el funcionamiento del sistema y qué consecuencias tendría la pérdida o el retraso de un determinado flujo. Esta visión resulta esencial para analizar la robustez del sistema y para introducir mecanismos de supervisión o degradación funcional cuando parte de la información no está disponible.

Otro aspecto clave del enfoque basado en flujo de datos es su relación con el **tiempo**. Cada flujo tiene asociadas características temporales: frecuencia de producción, latencia, variabilidad y posible obsolescencia de los datos. Desde una perspectiva arquitectónica, no todos los flujos son iguales ni tienen la misma importancia. Algunos flujos, como los relacionados con la percepción inmediata de obstáculos, pueden requerir latencias muy bajas; otros, como mapas o configuraciones, pueden tolerar retrasos mayores. Modelar explícitamente el sistema en términos de flujos de datos permite identificar estos requisitos y tomar decisiones informadas sobre su implementación.

Cada flujo impone requisitos de frecuencia y latencia.

El enfoque de flujo de datos también favorece la **escalabilidad** y la **extensibilidad** del sistema. Añadir un nuevo comportamiento puede consistir simplemente en introducir un nuevo componente que consuma flujos existentes y produzca otros nuevos. Del mismo modo, un mismo flujo puede ser consumido por múltiples componentes con propósitos distintos, sin que el productor tenga que modificarse. Esta propiedad resulta especialmente útil en robótica, donde es habitual reutilizar información sensorial o estimaciones internas para distintos fines, como control, visualización o registro.

Nuevos comportamientos se añaden conectándose a flujos existentes.

Sin embargo, adoptar un enfoque basado en flujo de datos no elimina la necesidad de **diseño arquitectónico** cuidadoso. Un uso indiscriminado de flujos puede conducir a sistemas difíciles de comprender, con dependencias implícitas y rutas de datos poco claras. Es responsabilidad del arquitecto decidir qué flujos existen, qué significado tienen y qué garantías ofrecen. En particular, es importante evitar que los componentes asuman supuestos implícitos sobre el orden de llegada de los datos o sobre la coherencia global del sistema, ya que estos supuestos pueden romperse fácilmente en un entorno concurrente y distribuido.

Los flujos deben tener semántica clara y garantías explícitas.

En muchos sistemas robóticos, el flujo de datos convive con otros mecanismos de interacción, como llamadas a servicios o acciones de larga duración. No obstante, incluso en estos casos, el flujo de datos suele constituir la columna vertebral del sistema, especialmente en los niveles relacionados con percepción y control. Comprender este modelo y sus implicaciones permite diseñar arquitecturas más claras, modulares y adaptadas a la naturaleza continua y dinámica de la información en robótica.

El flujo de datos convive con servicios, pero suele ser la base.

En resumen, concebir la arquitectura de un sistema robótico como una red de flujos de datos proporciona un marco poderoso para estructurar el software, analizar sus propiedades temporales y gestionar la complejidad. Este enfoque no prescribe una implementación concreta, pero establece un modo de razonar que será recurrente a lo largo del libro y que se materializará de forma explícita en las herramientas y prácticas basadas en ROS 2.

Pensar en flujos ayuda a estructurar y escalar la arquitectura.

En la práctica, esta arquitectura orientada a flujos se implementa mediante **mecanismos reactivos**: la llegada de un dato, la expiración de un temporizador o la finalización de una operación interna se interpretan como **eventos** que disparan la ejecución del software asociado. Los **callbacks** constituyen, por tanto, **puntos de entrada** al sistema y deben

Los flujos suelen materializarse como eventos que activan callbacks.

diseñarse con criterio arquitectónico para que la **concurrencia implícita** no degrade la **claridad** ni la **robustez**.

Desde una perspectiva arquitectónica, el uso de **eventos** y **callbacks** tiene implicaciones profundas. En primer lugar, introduce asincronía de forma natural: los callbacks pueden ejecutarse en momentos imprevisibles, dependiendo de cuándo se produzcan los eventos. En segundo lugar, introduce concurrencia implícita: distintos eventos pueden dar lugar a la ejecución de callbacks diferentes de manera simultánea o intercalada. Estas propiedades no son opcionales ni excepcionales; forman parte intrínseca del modelo de ejecución y deben ser tenidas en cuenta en el diseño del sistema.

Asincronía y concurrencia son inherentes al diseño reactivo.

Uno de los riesgos habituales al trabajar con eventos y callbacks es perder la **visión global** del comportamiento del sistema. Cuando la lógica se fragmenta en múltiples callbacks distribuidos entre distintos componentes, puede resultar difícil entender qué ocurre en conjunto o en qué orden se ejecutan determinadas acciones. Por esta razón, es fundamental que los callbacks sean conceptualmente simples, con responsabilidades bien acotadas, y que la arquitectura defina claramente qué tipo de decisiones deben tomarse dentro de ellos y cuáles deben delegarse a niveles superiores.

Callbacks simples y responsabilidades claras preservan la visión global.

Otro aspecto crítico es la gestión del **estado compartido**. Dado que los callbacks pueden ejecutarse de forma concurrente, acceder o modificar estado común sin las debidas precauciones puede dar lugar a condiciones de carrera, inconsistencias o comportamientos no deterministas. Desde el punto de vista arquitectónico, esto refuerza la importancia de diseñar componentes que minimicen el estado compartido y que expongan su estado de forma explícita mediante mecanismos de comunicación bien definidos, en lugar de mediante variables globales o dependencias implícitas.

Minimizar estado compartido evita carreras e inconsistencias.

Los eventos y callbacks también influyen en cómo se razona sobre el **tiempo** en el sistema. La latencia entre la ocurrencia de un evento y la ejecución de su callback, así como la duración de dicho callback, afectan directamente a la capacidad de respuesta del robot. Por ello, es habitual imponer restricciones arquitectónicas, como mantener los callbacks cortos, evitar operaciones bloqueantes y delegar tareas complejas a otros componentes o a procesos asincrónicos. Estas decisiones no son meras recomendaciones de estilo, sino elementos clave para garantizar un comportamiento predecible.

Callbacks cortos y no bloqueantes mantienen baja latencia.

Es importante destacar que el uso de eventos y callbacks no elimina la necesidad de **estructura** ni de **planificación**. Aunque la ejecución sea reactiva, la arquitectura debe definir qué eventos existen, qué callbacks se asocian a ellos y cómo se coordinan las reacciones resultantes. En muchos casos, los callbacks actúan como mecanismos de entrada a comportamientos más estructurados, como máquinas de estados o árboles de comportamiento, que permiten organizar la lógica de decisión de forma más clara y controlada.

Los callbacks suelen alimentar comportamientos más estructurados.

En resumen, los eventos y callbacks constituyen el mecanismo fundamental mediante el cual se implementa un modelo de ejecución reactivo. Proporcionan flexibilidad y capacidad de respuesta, pero introducen asincronía y concurrencia que deben gestionarse cuidadosamente. Comprender sus implicaciones arquitectónicas es esencial para diseñar sistemas robóticos que sean no solo funcionales, sino también comprensibles, robustos y mantenibles a lo largo del tiempo.

Eventos y callbacks dan flexibilidad, pero exigen disciplina.

2.2. Aplicación

Nodos

En ROS 2, los **nodos** son las **unidades básicas de ejecución** y constituyen la materialización directa del modelo de componentes reactivos introducido en la parte teórica de este capítulo. Un nodo es un **proceso software** que permanece activo dentro del sistema y que **reacciona a eventos** mediante la ejecución de **callbacks** asociados a mecanismos de comunicación, temporizadores u otros estímulos internos 2.4. Desde esta perspectiva, los nodos no deben entenderse como simples contenedores de código, sino como **entidades activas** que participan de forma continua en el comportamiento del robot.

Los nodos son procesos reactivos que ejecutan callbacks.

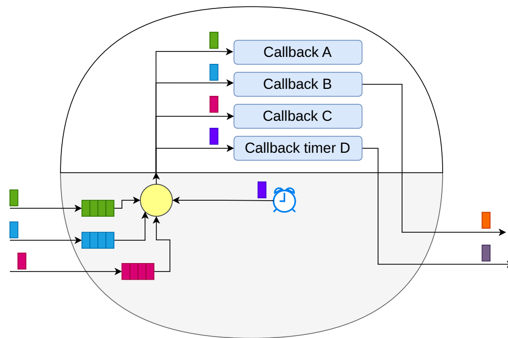


Figura 2.4: Un nodo ROS 2 como proceso reactivo, donde el usuario define callbacks para eventos específicos. La gestión de los eventos y la ejecución de los callbacks la realiza la infraestructura de ROS 2. Aunque no es exactamente así en la realidad, podemos entender que el círculo amarillo representa el mecanismo que despacha los mensajes a los callback, del que ya daremos detalles en próximos capítulos.

Cada nodo **encapsula** una funcionalidad concreta y ofrece una **interfaz explícita** hacia el resto del sistema, normalmente en forma de **flujos de datos** o **servicios**. Esta encapsulación es uno de los elementos clave del diseño arquitectónico en ROS 2. Un nodo bien diseñado tiene un propósito claro, procesa un tipo de información bien definido y produce resultados que pueden ser consumidos por otros nodos sin necesidad de conocer su implementación interna. Este enfoque favorece el desacoplamiento y permite que los distintos componentes del sistema evolucionen de forma independiente.

Cada nodo encapsula funcionalidad y expone interfaces explícitas.

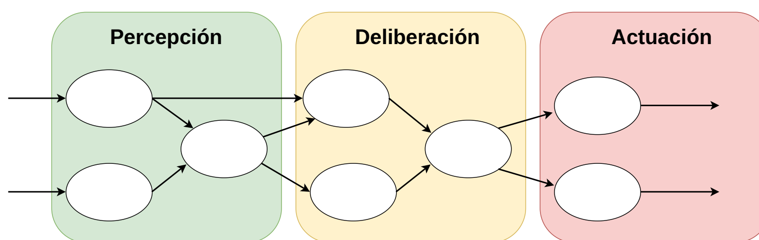


Figura 2.5: Sistema compuesto por distintos nodos con funciones claras que colaboran para percibir, deliberar y actuar.

Desde un punto de vista arquitectónico, resulta esencial asignar **responsabilidades claras** y bien delimitadas a cada nodo. Diseñar nodos excesivamente grandes, que mezclan percepción, decisión y actuación (Fig. 2.5), suele conducir a sistemas monolíticos difíciles de comprender, depurar y reutilizar. Del mismo modo, nodos con múltiples roles tienden a acumular dependencias implícitas y a convertirse en puntos críticos del sistema. En contraste, nodos con responsabilidades bien acotadas facilitan el razonamiento arquitectónico y permiten reorganizar el sistema con menor coste cuando cambian los requisitos.

La cohesión de cada nodo evita monolitos difíciles de mantener.

El diseño de nodos también está estrechamente relacionado con el **modelo de ejecución reactivo**. Un nodo no controla cuándo se ejecuta

La lógica interna debe tolerar asincronía y concurrencia.

su código; simplemente declara qué callbacks deben ejecutarse cuando ocurren determinados eventos. Esto implica que la lógica interna del nodo debe estar preparada para ejecutarse en un contexto asincrónico y potencialmente concurrente. Desde el punto de vista arquitectónico, esto refuerza la necesidad de evitar estados compartidos innecesarios y de estructurar el código interno de forma que cada callback tenga una responsabilidad clara y limitada.

Otro aspecto relevante es que los nodos constituyen unidades naturales de **despliegue** y **distribución**. En ROS 2, distintos nodos pueden ejecutarse en procesos separados, en diferentes máquinas o incluso en distintos dispositivos dentro de un sistema robótico distribuido (Fig. 2.6). Esta posibilidad no debe considerarse un detalle técnico, sino una característica arquitectónica fundamental. Diseñar nodos con interfaces limpias y sin dependencias implícitas facilita su redistribución y escalado, algo habitual en sistemas robóticos complejos.

Interfaces limpias facilitan despliegue distribuido y escalado.

Entender los nodos como componentes reactivos permite trasladar de forma directa los principios teóricos de este capítulo a sistemas ROS 2 reales. Los conceptos de eventos, callbacks, flujo de datos y desacoplamiento encuentran en los nodos su expresión concreta. A lo largo del libro, se utilizarán los nodos como unidad básica para ejemplificar decisiones arquitectónicas, analizar diseños existentes y construir sistemas progresivamente más complejos, siempre manteniendo el foco en la claridad, la modularidad y la capacidad de evolución del sistema.

Los nodos concretan eventos, datos y desacoplamiento en ROS 2.

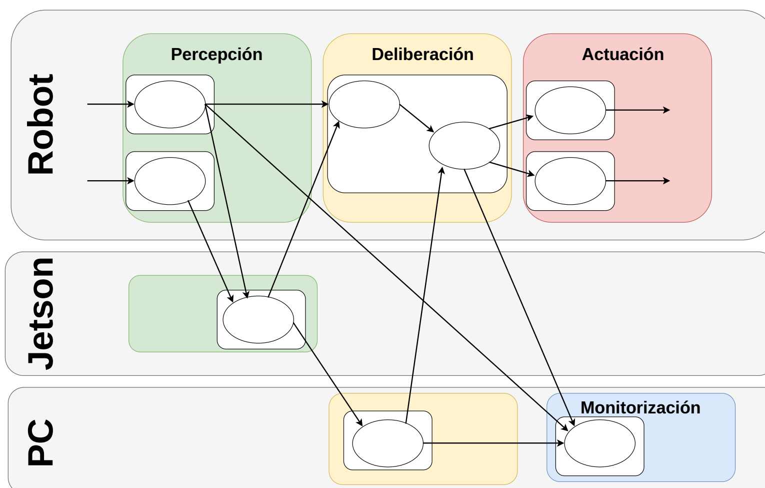


Figura 2.6: Sistema distribuido entre varias máquinas y procesos distintos.

Publishers y subscribers

Los mecanismos de **publicación y subscripción** constituyen la materialización directa del modelo de **flujo de datos** en ROS 2. Mediante este esquema, los *publishers* generan datos y los publican en **canales de comunicación** bien definidos, mientras que los *subscribers* **reaccionan** a la llegada de esos datos ejecutando **callbacks** asociados (Fig. 2.7). Este patrón permite que la información fluya de manera continua entre los distintos componentes del sistema, sin necesidad de que exista una relación explícita de invocación entre ellos.

Pub/sub implementa el flujo de datos en ROS 2.

Una de las principales ventajas de este modelo es que favorece la **comunicación asíncrona** y el **desacoplamiento** entre componentes. Un publisher no necesita conocer qué nodos están consumiendo los datos que produce,

El desacoplamiento se logra mediante contratos de mensajes.

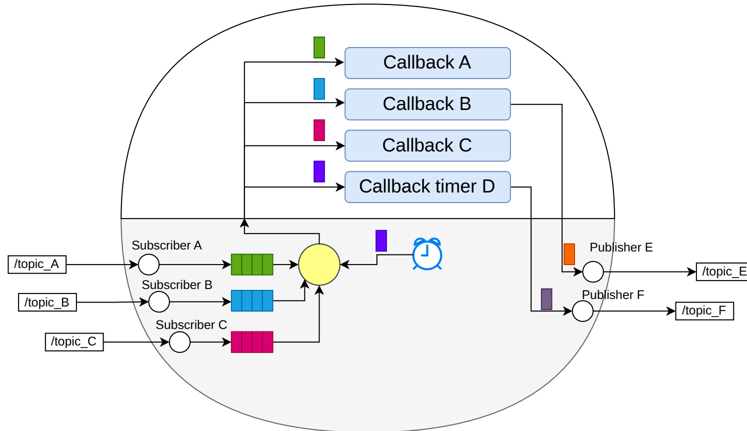


Figura 2.7: Publicadores y suscriptores en un nodo ROS 2.

ni con qué finalidad se utilizan. De forma análoga, un subscriber no depende de un productor concreto, sino del tipo de información que recibe. Esta independencia permite que productores y consumidores evolucionen de forma separada, siempre que se mantenga estable el contrato de datos representado por el tipo de mensaje.

Este esquema resulta especialmente adecuado para representar información que cambia con frecuencia o que debe difundirse a múltiples componentes, como datos sensoriales, estimaciones de estado o comandos de control. En muchos sistemas robóticos, gran parte del comportamiento se articula precisamente en torno a la transformación progresiva de estos flujos de datos, desde medidas crudas hasta decisiones de actuación. El uso de publishers y subscribers proporciona una forma natural de expresar este tipo de arquitecturas.

Es idóneo para sensórica, estado y comandos frecuentes.

Desde un punto de vista arquitectónico, el diseño de los publishers y subscribers define gran parte de la **estructura del sistema** y de sus **dependencias funcionales**. Decidir qué información se publica, con qué granularidad y con qué frecuencia es una decisión arquitectónica, no meramente técnica. Un flujo demasiado grueso puede forzar a los consumidores a procesar información innecesaria; un flujo demasiado fino puede fragmentar la lógica y dificultar la comprensión del sistema. De forma similar, publicar información que no es conceptualmente estable puede generar dependencias frágiles entre componentes.

Granularidad y frecuencia de los flujos son decisiones arquitectónicas.

Otro aspecto relevante es que la comunicación basada en publicación y subscripción introduce una **separación clara** entre el momento en que se genera la información y el momento en que se consume. Esta separación tiene implicaciones temporales importantes: los datos pueden llegar con cierto retraso, pueden perderse o pueden llegar a distintos consumidores con latencias diferentes. Desde el punto de vista arquitectónico, es fundamental que los componentes estén diseñados para tolerar estas condiciones y no asuman un comportamiento perfectamente sincronizado del sistema.

Los componentes deben tolerar latencias, pérdidas y desincronización.

El uso de publishers y subscribers también facilita la **observabilidad** y el **diagnóstico** del sistema. Al estar la información distribuida en flujos explícitos, resulta sencillo inspeccionar qué datos se están produciendo y consumiendo en cada momento, así como introducir componentes adicionales para visualización, registro o depuración sin alterar el funcionamiento del sistema principal. Esta propiedad resulta especialmente valiosa en robótica, donde la interacción con el mundo físico puede producir comportamientos difíciles de reproducir.

Los flujos explícitos facilitan inspección y depuración.

En resumen, los publishers y subscribers constituyen el eje central de la comunicación basada en flujo de datos en ROS 2. Proporcionan desacoplamiento, flexibilidad y escalabilidad, pero también imponen la necesidad de tomar decisiones arquitectónicas conscientes sobre la estructura y el significado de los flujos de información. Comprender estos mecanismos y sus implicaciones es esencial para diseñar sistemas robóticos claros, robustos y preparados para crecer en complejidad.

Pub/sub es central, pero requiere diseño consciente de los flujos.

Topics

Los *topics* son los **canales de comunicación** a través de los cuales se intercambian datos en ROS 2 y constituyen el elemento central del modelo de publicación y subscripción. Cada topic define un flujo de información de un tipo de mensaje concreto, al que pueden conectarse uno o varios publishers y uno o varios subscribers (Fig. 2.8). Desde el punto de vista conceptual, un topic no pertenece a un nodo en particular, sino que existe como una entidad lógica compartida dentro del sistema.

Los topics son canales lógicos compartidos de un tipo de mensaje.

Arquitectónicamente, los topics representan ***interfaces públicas*** entre componentes. Publicar información en un topic equivale a declarar que ese dato forma parte del contrato del sistema y que puede ser utilizado por otros componentes presentes o futuros. Por esta razón, el diseño de los topics no debe abordarse como una decisión local de implementación, sino como una decisión estructural que condiciona la interoperabilidad, la reutilización y la evolución del sistema robótico.

Un topic es una interfaz pública que condiciona la evolución.

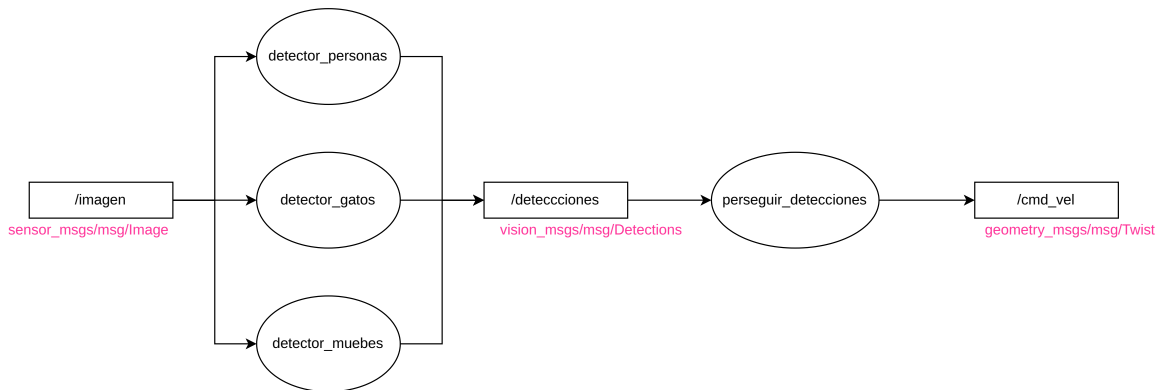


Figura 2.8: Topics como canales de comunicación entre nodos.

Un aspecto clave del diseño de topics es la definición del **tipo de mensaje** asociado. El tipo de mensaje establece qué información se intercambia, con qué estructura y con qué significado. Un mensaje demasiado específico puede limitar la reutilización del flujo de datos; uno demasiado genérico puede generar ambigüedad y forzar a los consumidores a interpretar información que no les corresponde. Desde una perspectiva arquitectónica, un buen diseño de mensajes busca un equilibrio entre expresividad y estabilidad, capturando el significado esencial del dato sin imponer supuestos innecesarios sobre su uso.

El tipo de mensaje fija el contrato y su reutilización.

Además del tipo de mensaje, el propio **nombre** del topic tiene una relevancia arquitectónica. Los nombres de topics actúan como puntos de acoplamiento semántico entre componentes: reflejan qué información se intercambia y con qué propósito. Convenciones claras y consistentes en el nombrado facilitan la comprensión del sistema y reducen la probabilidad de malentendidos. En sistemas grandes, un esquema de nombres

Un buen nombrado de topics actúa como documentación viva.

bien diseñado puede servir como documentación viva de la arquitectura, permitiendo inferir relaciones entre componentes sin necesidad de inspeccionar el código.

Elegir qué información se publica, con qué frecuencia y en qué topics es una decisión estructural que afecta al comportamiento global del robot. Publicar información con demasiada frecuencia puede saturar el sistema y aumentar latencias; hacerlo con poca frecuencia puede reducir la capacidad de reacción. Publicar información derivada en lugar de datos crudos puede simplificar el diseño de los consumidores, pero también reducir la flexibilidad para usos futuros. Estas decisiones deben tomarse considerando tanto los requisitos actuales como la posible evolución del sistema.

Frecuencia y contenido de topics determinan latencia y flexibilidad.

Desde el punto de vista del desacoplamiento, los topics permiten que múltiples componentes consuman la misma información sin que el productor tenga conocimiento de ello. Esta propiedad es especialmente valiosa en robótica, donde una misma fuente de datos —por ejemplo, una estimación de estado— puede ser utilizada simultáneamente por controladores, módulos de planificación, sistemas de supervisión o herramientas de visualización. El uso de topics evita duplicar lógica y favorece una arquitectura en la que los componentes se conectan a través de flujos de información bien definidos.

Un mismo topic puede alimentar múltiples consumidores sin acoplarlos.

No obstante, el uso indiscriminado de topics también puede introducir complejidad innecesaria. Un exceso de topics, flujos poco claros o mensajes con significados implícitos pueden dificultar la comprensión del sistema y generar dependencias ocultas. Desde el punto de vista arquitectónico, es preferible contar con un número moderado de topics bien definidos, con semántica clara y responsabilidades explícitas, que con una proliferación de canales difíciles de razonar.

Evitar proliferación de topics y semánticas implícitas reduce complejidad.

Finalmente, los topics constituyen un mecanismo fundamental para la observabilidad del sistema. Al ser flujos de datos explícitos, pueden inspeccionarse, registrarse o visualizarse sin alterar el comportamiento del sistema principal. Esta característica resulta esencial para el diagnóstico, la depuración y la validación de sistemas robóticos, especialmente en entornos donde reproducir exactamente las condiciones de operación es complicado.

Los topics habilitan observabilidad sin intrusión.

En conjunto, los topics son mucho más que un detalle técnico de ROS 2: son el elemento arquitectónico que define cómo fluye la información dentro del sistema. Diseñarlos con cuidado es una de las tareas más importantes en la construcción de arquitecturas software para robots, ya que de ellos depende en gran medida la claridad, la robustez y la capacidad de evolución del sistema completo.

Diseñar topics bien es clave para claridad y robustez.

Primeros mensajes

Los mensajes (Fig. 2.9) definen la estructura de los datos que circulan por los topics y constituyen el **contrato explícito** entre productores y consumidores de información dentro del sistema. En un sistema basado en publicación y subscripción, los mensajes son el único conocimiento compartido entre componentes: un nodo no conoce la implementación interna de otro, pero sí conoce el tipo y el significado de los mensajes que intercambian. Desde esta perspectiva, los mensajes desempeñan un papel arquitectónico central, ya que materializan las interfaces de datos del sistema.

Los mensajes son el contrato de datos entre nodos.

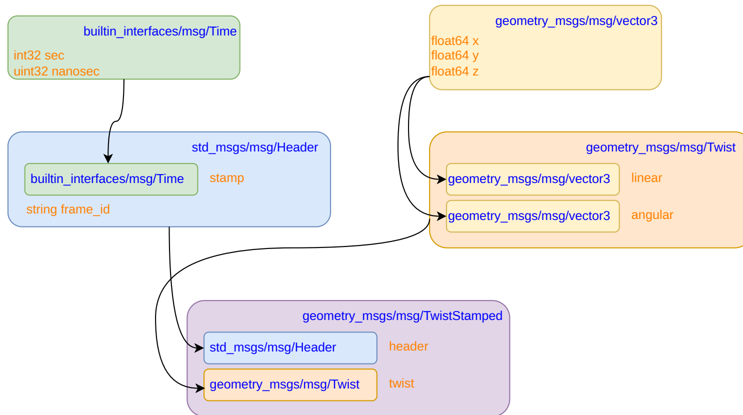


Figura 2.9: Composición de mensajes.

Un mensaje no es simplemente una estructura de datos. Define qué información es relevante, cómo se representa y con qué nivel de detalle se expone al resto del sistema. Decidir qué campos incluye un mensaje, qué unidades se utilizan, qué convenciones temporales se asumen o qué información se omite es una decisión de diseño que condiciona la forma en que otros componentes pueden interactuar con ese flujo de datos. Por esta razón, el diseño de mensajes debe abordarse con el mismo cuidado que el diseño de interfaces en otros dominios del software.

El contenido del mensaje condiciona cómo se integra el sistema.

Un diseño adecuado de mensajes facilita la claridad conceptual y la extensibilidad del sistema. Mensajes bien definidos, con semántica clara y campos coherentes, permiten que nuevos componentes se integren fácilmente sin necesidad de conocer el contexto completo del sistema. Por ejemplo, un mensaje que representa una estimación de estado debe ser interpretable por distintos consumidores —controladores, planificadores, sistemas de supervisión— sin que cada uno tenga que reinterpretar su significado. Esta propiedad resulta esencial para la reutilización y la evolución del sistema robótico.

Mensajes claros facilitan extensibilidad e integración.

Por el contrario, mensajes mal definidos pueden introducir **ambigüedades** y **acoplamientos** innecesarios. Incluir información irrelevante, mezclar conceptos de distinto nivel de abstracción o depender de supuestos implícitos sobre el origen o el uso de los datos puede dificultar la comprensión del sistema y generar errores sutiles. Un ejemplo habitual es incorporar en un mensaje detalles específicos de un algoritmo concreto, lo que limita su reutilización y obliga a modificar múltiples componentes cuando dicho algoritmo cambia.

Mensajes ambiguos introducen acoplamientos y deuda técnica.

Desde un punto de vista arquitectónico, los mensajes también influyen en la gestión del **tiempo** y del **estado**. Muchos mensajes en robótica incluyen marcas temporales, identificadores de referencia espacial o información de calidad de los datos. Estos elementos permiten razonar sobre la validez y la coherencia de la información en un sistema distribuido y concurrente. Aunque en este primer capítulo no se profundizará en estos aspectos, es importante reconocer que los mensajes son el vehículo a través del cual se propagan tanto los datos como el contexto necesario para interpretarlos correctamente.

Timestamps y referencias espaciales aportan contexto y coherencia.

En este primer contacto, el objetivo es familiarizarse con el uso de mensajes estándar proporcionados por el ecosistema ROS 2 y comprender su papel arquitectónico. El uso de mensajes ampliamente adoptados facilita la interoperabilidad con herramientas y paquetes existentes, y permite centrarse en el diseño del sistema en lugar de en detalles de bajo

Usar mensajes estándar mejora interoperabilidad y criterio.

nivel. Al mismo tiempo, analizar estos mensajes desde una perspectiva arquitectónica ayuda a desarrollar criterio sobre qué información debe formar parte de un contrato público y qué información debe permanecer encapsulada dentro de un componente.

A lo largo de los capítulos posteriores y de las prácticas, se profundizará progresivamente en el diseño de mensajes propios y en la evaluación de sus implicaciones arquitectónicas. En este punto inicial, basta con comprender que los mensajes no son un detalle accesorio, sino uno de los pilares sobre los que se construyen las arquitecturas basadas en flujo de datos. Sentar estas bases desde el principio permitirá abordar diseños más complejos con una comprensión sólida de cómo se estructura y se intercambia la información en sistemas robóticos reales.

El diseño de mensajes propios será un eje de las prácticas.

Servicios y acciones

Aunque el modelo de publicación y subscripción basado en topics constituye la base del **flujo de datos** en ROS 2, no todos los patrones de interacción entre componentes pueden expresarse de forma adecuada mediante **flujos continuos** de información. Para cubrir otros tipos de comunicación, ROS 2 proporciona mecanismos adicionales: los *servicios* y las *acciones*. Ambos permiten estructurar interacciones más dirigidas y con semántica explícita, complementando el modelo puramente reactivo basado en datos.

Servicios y acciones complementan a los flujos continuos.

Los *servicios* implementan un modelo de comunicación de tipo **petición–respuesta** (Fig. 2.10). Un nodo cliente realiza una solicitud concreta y un nodo servidor responde con un resultado. Desde el punto de vista arquitectónico, este modelo resulta apropiado para operaciones puntuales, acotadas en el tiempo y con un resultado bien definido, como consultar un estado, modificar una configuración o solicitar una operación rápida. A diferencia de los topics, los servicios establecen una relación explícita entre solicitante y proveedor, lo que introduce un mayor acoplamiento pero también una semántica más clara para determinadas interacciones.

Los servicios son adecuados para operaciones puntuales con respuesta.

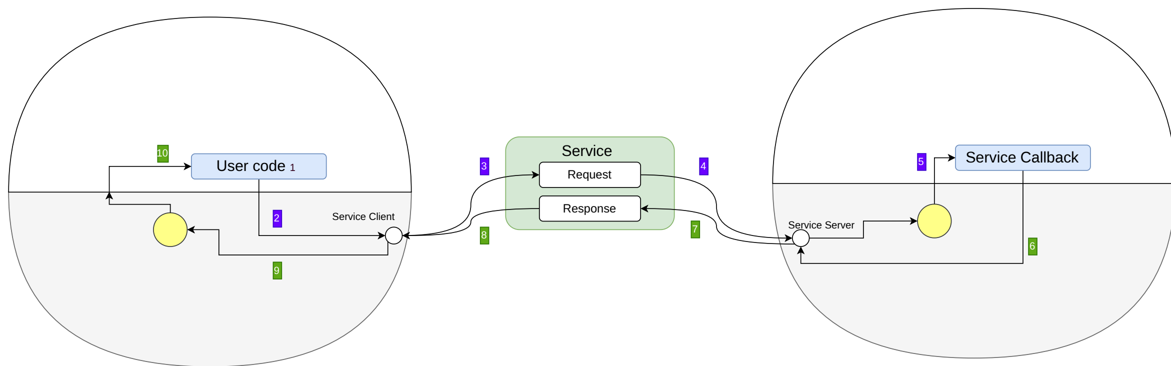


Figura 2.10: Mecanismo de servicios en ROS 2.

El uso de servicios implica un cambio en el modo de razonar sobre la ejecución del sistema. Mientras que en el flujo de datos los componentes reaccionan de forma continua a la llegada de información, en los servicios existe una intención explícita por parte del cliente: se solicita algo concreto y se espera una respuesta. Arquitectónicamente, esto obliga a reflexionar sobre qué operaciones deben exponerse como servicios y cuáles deben mantenerse como flujos de datos, evitando utilizar servicios

Elegir servicio o topic depende de la semántica de la información.

para intercambiar información que debería modelarse como un estado continuo del sistema.

Las *acciones* extienden el modelo de servicios para representar operaciones de **larga duración** (Fig. 2.11). Una acción permite iniciar una tarea que puede tardar un tiempo significativo en completarse, proporcionando además mecanismos para recibir retroalimentación intermedia y para cancelar la operación si es necesario. Desde un punto de vista arquitectónico, las acciones resultan especialmente adecuadas para representar capacidades complejas del robot, como navegar a una posición, seguir una trayectoria o ejecutar una maniobra prolongada.

Las acciones modelan tareas largas con feedback y cancelación.

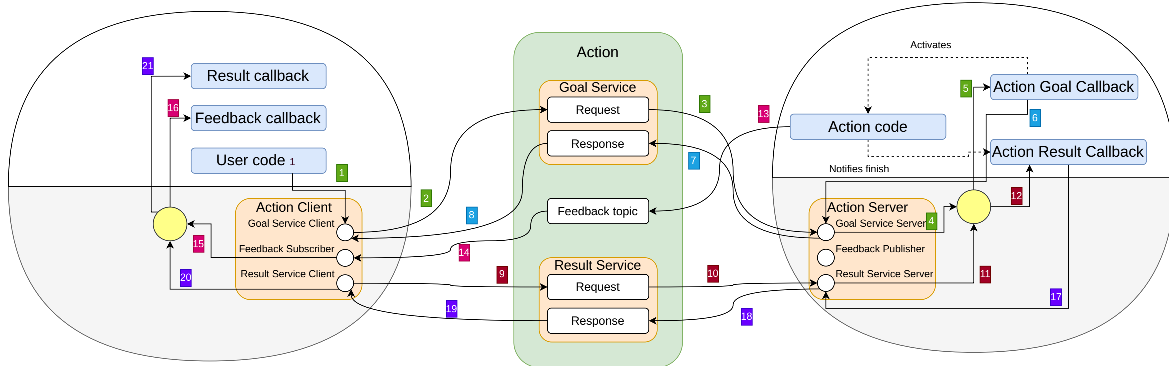


Figura 2.11: Mecanismo de acciones en ROS 2.

A diferencia de los servicios, las acciones encajan de forma natural con el modelo misión–tarea–capacidad introducido en el capítulo anterior. Una acción suele corresponder a la invocación de una capacidad: se solicita su ejecución, se monitoriza su progreso y se recibe un resultado final. Este patrón permite desacoplar la lógica de alto nivel, que decide cuándo y por qué ejecutar una acción, de la lógica de bajo nivel, que se encarga de cómo llevarla a cabo. De este modo, las acciones actúan como interfaces explícitas entre tareas y capacidades.

Las acciones exponen capacidades a la lógica de alto nivel.

Desde la perspectiva del modelo de ejecución reactivo, tanto servicios como acciones siguen siendo mecanismos basados en eventos. La recepción de una solicitud, la llegada de una respuesta, la actualización de feedback o la finalización de una acción se traducen en eventos que activan callbacks dentro de los nodos correspondientes. Sin embargo, su semántica es distinta de la de los topics: mientras que estos representan flujos continuos de información, los servicios y acciones representan interacciones intencionales y estructuradas.

Servicios y acciones también se implementan mediante eventos.

El **diseño arquitectónico** de servicios y acciones debe realizarse con el mismo cuidado que el de los topics. Exponer demasiadas operaciones como servicios puede introducir dependencias rígidas y dificultar la evolución del sistema. Por el contrario, utilizar únicamente topics para todo tipo de interacción puede llevar a diseños confusos, donde la intención de una operación no queda clara. Un uso equilibrado de estos mecanismos permite expresar de forma explícita distintos tipos de relaciones entre componentes: flujos de información, consultas puntuales y ejecuciones de tareas complejas.

Un uso equilibrado evita acoplamientos rígidos o diseños confusos.

En conjunto, los servicios y acciones completan el conjunto de paradigmas de comunicación disponibles en ROS 2. Comprender cuándo utilizar cada uno —topics para datos continuos, servicios para operaciones puntuales y acciones para tareas de larga duración— es fundamental para diseñar arquitecturas robóticas claras y coherentes. En los capítulos y prácticas

Topic, servicio y acción se eligen según la interacción deseada.

posteriores, estos mecanismos se emplearán de forma combinada para ilustrar cómo una arquitectura bien diseñada utiliza distintos patrones de comunicación según la naturaleza de la interacción que se desea modelar.

3 Percepción robótica y representación espacial

3.1. Principios teóricos

3.1 Principios teóricos **27**
Sensores crudos 27
Percepción como transformación de datos 29
Marco espacial y referencias 30
Fusión sensorial básica . . . 32
3.2 Aplicación **33**
Gestión de sensores en ROS 2 33
Sistema de transformaciones en ROS 2 35
Sensores habituales en ROS 2 39
De sensores a detecciones: reducción de complejidad . 46

Sensores crudos

Los sensores (Fig. 3.1) constituyen el punto de contacto físico entre el robot y su entorno, y representan la única vía mediante la cual el sistema puede acceder a información externa o interna. A través de ellos, el robot obtiene acceso a variables físicas tales como distancia, intensidad luminosa, aceleración, velocidad angular, presión o temperatura. En esta etapa inicial, el sistema únicamente dispone de **mediciones directas de magnitudes físicas** transformadas en datos digitales mediante procesos de muestreo y cuantificación. Dichas mediciones reflejan fenómenos del mundo real, pero lo hacen en forma estrictamente numérica y dependiente del dispositivo que las genera. No existe todavía estructuración geométrica, modelado ni interpretación conceptual: el sistema no "sabe" que un conjunto de distancias corresponde a una pared ni que una imagen contiene un objeto. Solo dispone de valores numéricos organizados según el formato propio del sensor y asociados inequívocamente a un hardware concreto.

Los sensores crudos constituyen el único acceso físico al entorno y producen mediciones digitalizadas sin estructura ni interpretación.

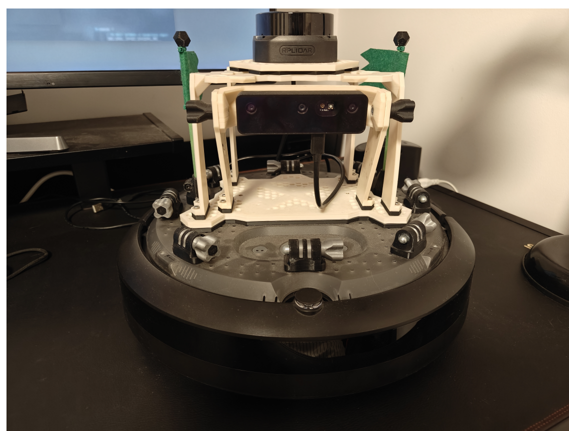


Figura 3.1: Robot iCreate equipado con un laser 2D y con una cámara RGB-D (*RGB-Depth*).

Desde una perspectiva arquitectónica, la capa de sensores crudos tiene una responsabilidad claramente delimitada: **adquirir, encapsular y publicar datos tal como los entrega el hardware**. Esto implica interactuar

con drivers, buses de comunicación y protocolos específicos, así como gestionar la inicialización y configuración básica del dispositivo.

La responsabilidad de esta capa es adquirir y encapsular datos sin interpretarlos.

Los datos generados en esta etapa están definidos por las características físicas del sensor: resolución angular en un LIDAR (*Light Detection and Ranging*), frecuencia de muestreo en una IMU (*Inertial Measurement Unit*), tamaño de imagen en una cámara, rango dinámico en un micrófono. Estas propiedades determinan la forma y dimensionalidad del flujo de datos, que debe preservarse fielmente.

La estructura de los datos viene determinada por las propiedades físicas del sensor.

Un aspecto esencial de esta capa es la **gestión del ciclo de vida del dispositivo**. Esto incluye su inicialización, verificación de estado, detección de errores y recuperación ante fallos. Desde el punto de vista arquitectónico, estos mecanismos deben quedar encapsulados, evitando que los niveles superiores dependan de detalles operativos del hardware.

La gestión del ciclo de vida del dispositivo pertenece exclusivamente a esta capa.

Asimismo, la capa de sensores crudos debe encargarse de la **configuración paramétrica** del sensor: resolución, rango, frecuencia, modos de operación, etc. Estos parámetros no forman parte del procesamiento perceptivo, sino de la correcta operación del dispositivo físico. Su definición explícita mejora la trazabilidad y reproducibilidad del sistema.

La configuración del sensor es responsabilidad de la capa de adquisición.

Un aspecto crítico dentro de la capa de sensores crudos es la **calibración** (Fig. 3.2). Todo sensor introduce desviaciones sistemáticas respecto a la magnitud física real debido a tolerancias de fabricación, alineamientos imperfectos o variaciones ambientales. La calibración consiste en estimar y compensar estos parámetros con el objetivo de garantizar que los datos publicados representen fielmente la realidad física medida. Se distingue habitualmente entre **calibración intrínseca**, que corrige propiedades internas del sensor (factores de escala, sesgos, matrices de distorsión óptica, ruido característico), y **calibración extrínseca**, que determina la transformación rígida entre el marco del sensor y el marco del robot. Desde el punto de vista arquitectónico, ambos tipos de parámetros deben gestionarse como configuración explícita de la capa sensorial, evitando que compensaciones geométricas o correcciones sistemáticas se dispersen en módulos posteriores. La calibración no introduce interpretación ni abstracción, pero sí garantiza coherencia física y consistencia geométrica en la base del sistema.

La calibración, intrínseca y extrínseca, garantiza coherencia física y pertenece a la capa sensorial.

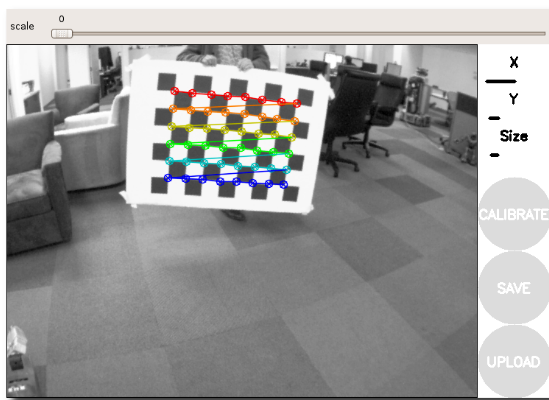


Figura 3.2: Calibración de los parámetros intrínsecos de una cámara monocular (imagen de `https://docs.ros.org/en/jazzy/p/camera_calibration/doc/tutorial_mono.html`).

Otro criterio clave es la definición de un **contrato de datos estable**. Aunque el hardware pueda cambiar, la interfaz publicada por el componente debe mantener coherencia estructural. Esto permite sustituir dispositivos físicos sin modificar el resto del sistema, siempre que el contrato se preserve.

Un contrato de datos estable permite sustituir hardware sin afectar al resto del sistema.

Desde el punto de vista de diseño, esta capa debe minimizar dependencias externas y evitar introducir cualquier forma de semántica. No debe clasificar, filtrar conceptualmente ni inferir estructura. Su función es estrictamente instrumental: proporcionar datos fiables y consistentes.

La capa de sensores crudos no introduce semántica ni estructura conceptual.

En síntesis, los sensores crudos representan la base material sobre la que se construye la percepción. Su correcta encapsulación permite aislar la complejidad del hardware y establecer una frontera arquitectónica clara entre adquisición física y procesamiento estructural posterior.

La encapsulación de sensores crudos establece una frontera clara entre hardware y procesamiento.

Percepción como transformación de datos

Una vez adquiridos los datos sensoriales, comienza la etapa de percepción. La percepción no consiste en adquirir nuevas medidas, sino en **transformar datos existentes en representaciones más estructuradas**. Es un proceso de abstracción progresiva que convierte magnitudes físicas en entidades manipulables por los módulos de decisión.

La percepción transforma datos en representaciones estructuradas y útiles.

Cada etapa perceptiva puede entenderse como una transformación formal entre espacios de datos. Si $\mathcal{D}_0$ representa datos en bruto y $\mathcal{D}_1$ una representación estructurada, entonces la percepción implementa funciones del tipo:

$$\mathcal{T} : \mathcal{D}_0 \to \mathcal{D}_1$$

donde la salida posee mayor organización interna que la entrada.

La percepción puede modelarse como transformaciones formales entre espacios de datos.

Este proceso suele implicar operaciones como **cambio de marco de referencia**, **extracción de características**, **agrupamiento**, **estimación de estado** o **construcción de representaciones espaciales**. Cada transformación reduce ambigüedad estructural y aumenta la capacidad del sistema para razonar sobre el entorno.

Las transformaciones perceptivas añaden estructura y capacidad de razonamiento.

Es importante destacar que la percepción introduce **modelos explícitos**. Cuando se estima una pose, se adopta un modelo geométrico; cuando se construye un mapa, se adopta un modelo espacial; cuando se detecta un objeto, se adopta un modelo de clase. La arquitectura debe garantizar que estos modelos estén bien encapsulados en módulos claramente definidos.

La percepción incorpora modelos explícitos que deben quedar encapsulados.

La organización típica de la percepción es en forma de **pipeline modular**. Cada componente recibe una representación estructurada y produce otra de mayor nivel. Esta composición favorece la verificabilidad y permite validar cada etapa de manera independiente.

La organización modular facilita validación y mantenibilidad.

En procesos más complejos, la percepción puede incluir **realimentación interna**. Por ejemplo, una estimación previa puede condicionar transformaciones posteriores. En estos casos, el sistema perceptivo mantiene estado interno y estructura iterativa, lo que exige una arquitectura que soporte dependencias temporales explícitas.

Algunos procesos perceptivos requieren estado interno y dependencias temporales.

Otro elemento central es la **reducción de dimensionalidad**. Transformar una imagen de millones de píxeles en un conjunto reducido de características relevantes constituye un ejemplo claro de cómo la percepción compacta la información para hacerla computacionalmente manejable.

La percepción reduce dimensionalidad y hace tratable la información.

Finalmente, la salida de la percepción no necesariamente es semántica en sentido humano. Puede ser una representación geométrica, probabilística o topológica que facilite planificación o control. La clave arquitectónica no es el nivel semántico alcanzado, sino la coherencia estructural de la transformación realizada.

La percepción produce representaciones coherentes, no necesariamente semánticas.

En conjunto, la percepción constituye la etapa en la que los datos adquieren estructura operativa. Es el mecanismo mediante el cual el robot pasa de disponer de mediciones a disponer de representaciones sobre las que puede decidir.

La percepción convierte mediciones en representaciones operativas para la decisión.

Marco espacial y referencias

Todo sistema robótico necesita expresar posiciones y orientaciones de manera coherente. Para ello se definen explícitamente **marcos de referencia**, cada uno caracterizado por un origen y un conjunto de ejes ortogonales que permiten describir coordenadas en el espacio. Sin una definición clara de referencias, no es posible integrar mediciones, estimaciones y acciones dentro de un mismo modelo geométrico.

El razonamiento espacial requiere marcos de referencia explícitos y coherentes.

Un mismo punto físico puede representarse mediante coordenadas distintas según el marco elegido (Fig. 3.3). Por ejemplo, la posición de un objeto puede expresarse respecto al sensor que lo detecta, respecto al centro del robot o respecto a un marco global fijo en el entorno. Estas representaciones son matemáticamente diferentes, aunque describan la misma entidad física.

Las coordenadas dependen del marco en el que se expresan.

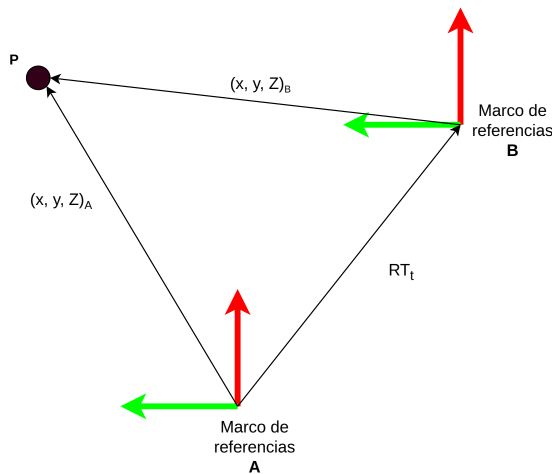


Figura 3.3: Un mismo punto P tiene coordenadas diferentes según el marco de referencia en el que se exprese.

La relación entre dos marcos rígidos tridimensionales se modela mediante una **transformación rígida**, compuesta por una rotación y una traslación. Esta transformación se representa mediante una matriz homogénea 4×4:

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

donde la submatriz $R = [r_{ij}] \in SO(3)$ describe la rotación y el vector $t = (t_x, t_y, t_z)^T$ describe la traslación.

Las transformaciones entre marcos se representan mediante matrices homogéneas RT 4×4.

Si un punto $p_A = (x, y, z, 1)^T$ está expresado en el marco A, su representación en el marco B se obtiene mediante:

$$p_B = T_{B \leftarrow A}\, p_A$$

Este mecanismo permite cambiar de referencia de manera algebraicamente consistente.

El cambio de referencia se realiza mediante multiplicación matricial homogénea.

Una propiedad fundamental es la **composición**. Si existen tres marcos A, B y C, se cumple:

$$T_{C\leftarrow A} = T_{C\leftarrow B}\, T_{B\leftarrow A}$$

La multiplicación matricial garantiza coherencia geométrica siempre que las transformaciones individuales sean consistentes.

Las transformaciones se componen mediante multiplicación matricial.

En robótica, las relaciones espaciales no son estáticas. La pose relativa entre marcos móviles depende del tiempo. Por tanto, la transformación debe considerarse como una función discreta:

$$T(t)$$

El sistema solo dispone de transformaciones en instantes $t_0, t_1, \ldots, t_n$.

Las transformaciones espaciales son funciones discretas del tiempo.

Cuando se requiere una transformación en un instante intermedio t', se emplea **interpolación**. La traslación puede interpolarse linealmente y la rotación mediante interpolación esférica para preservar ortonormalidad. Esta operación mantiene coherencia dentro del intervalo temporal conocido.

La interpolación permite estimar transformaciones entre muestras temporales.

La **extrapolación**, en cambio, implica proyectar la transformación hacia el futuro utilizando un modelo dinámico del movimiento. Esta operación introduce hipótesis adicionales y, por tanto, mayor incertidumbre estructural.

Extrapolar requiere modelos dinámicos y aumenta la incertidumbre.

Árbol de transformaciones

Aunque conceptualmente podría definirse un grafo arbitrario de relaciones espaciales, arquitectónicamente es preferible organizar los marcos como un **árbol dirigido**. En esta estructura, cada marco —excepto uno raíz— tiene exactamente un único padre.

Las referencias espaciales se organizan preferentemente como un árbol dirigido.

La restricción de árbol garantiza que entre cualquier par de marcos exista un único camino de composición. Esto elimina ambigüedades y asegura consistencia algebraica en el cálculo de transformaciones indirectas.

La estructura en árbol garantiza un único camino de composición.

Si el árbol está enraizado en un marco global W, la transformación entre dos marcos A y B puede obtenerse recorriendo el camino único hasta la raíz:

$$T_{B\leftarrow A} = T_{B\leftarrow W}\, T_{W\leftarrow A}$$

Cada término se obtiene mediante composición de transformaciones elementales padre–hijo.

Las transformaciones se calculan componiendo el camino único hasta la raíz.

Esta organización refleja tanto la estructura física del robot —sensores montados sobre enlaces, enlaces sobre una base— como su estructura lógica dentro del sistema software.

El árbol espacial refleja la estructura física y lógica del sistema.

Integración en la arquitectura software

El sistema de marcos y transformaciones no debe implementarse como utilidades dispersas, sino como un **servicio estructural centralizado**. Todos los módulos que produzcan o consuman información espacial deben apoyarse en una infraestructura común responsable de:

- Mantener el árbol coherente de marcos.
- Almacenar transformaciones asociadas a marcas temporales.
- Resolver consultas espaciales mediante composición automática.
- Gestionar interpolación temporal cuando sea necesario.

Este servicio actúa como capa transversal que conecta percepción, localización, planificación y control.

La infraestructura de transformaciones debe ser un servicio central transversal.

Los módulos funcionales no deben almacenar ni recomponer manualmente transformaciones espaciales globales. Delegar esta responsabilidad en la infraestructura común garantiza coherencia geométrica y reduce errores.

La resolución espacial debe delegarse en la infraestructura común.

Los principios aquí descritos —representación homogénea, composición algebraica, dependencia temporal y organización en árbol— constituyen la base formal sobre la que se construyen infraestructuras prácticas de gestión de transformaciones espaciales en sistemas robóticos modernos. En la parte aplicada del libro se estudiará cómo estos fundamentos se materializan en un sistema software capaz de mantener y consultar dinámicamente dicho árbol de referencias en tiempo real.

Los fundamentos teóricos preparan la comprensión de la infraestructura práctica de transformaciones.

Fusión sensorial básica

Un sistema robótico raramente opera con una única fuente de información. Sensores distintos observan el entorno desde posiciones físicas diferentes, con frecuencias de actualización distintas y con campos de visión parcialmente solapados. La **fusión sensorial básica** consiste en integrar estas observaciones dentro de una representación espacial y temporal coherente (Fig. 3.4).

La fusión básica integra múltiples sensores en un marco espacial y temporal común.

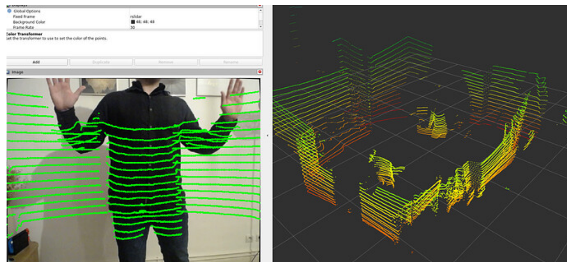


Figura 3.4: Fusión sensorial de imágenes y LIDAR (imagen de `https://github.com/CDonosoK/ros2_camera_lidar_fusion`).

Antes de cualquier integración, deben resolverse dos condiciones fundamentales: **coherencia espacial** y **coherencia temporal**. Las mediciones producidas por sensores distintos están expresadas inicialmente en sus propios marcos locales. Para poder combinarlas, es necesario transformarlas a un mismo marco de referencia utilizando la infraestructura de transformaciones descrita en la sección anterior.

La coherencia espacial exige expresar todas las mediciones en un mismo marco.

Por ejemplo, una nube de puntos procedente de un sensor tridimensional y una estimación de posición obtenida mediante odometría solo pueden integrarse si ambas se expresan respecto a una referencia común. Esta transformación no añade interpretación, pero garantiza que las magnitudes geométricas sean comparables.

La transformación a un marco común permite comparar e integrar mediciones.

Además de la coherencia espacial, la fusión exige **alineamiento temporal**. Sensores distintos pueden operar a frecuencias diferentes y con latencias variables. Integrar datos adquiridos en instantes distintos puede producir inconsistencias físicas si el robot está en movimiento. Por ello, es necesario consultar o interpolar transformaciones correspondientes al instante exacto en que cada medición fue generada.

La coherencia temporal es esencial cuando el sistema está en movimiento.

En esta etapa básica, la fusión no implica todavía inferencia estadística compleja. Consiste principalmente en **alinear y superponer información** procedente de distintas fuentes para construir una representación espacial unificada. Por ejemplo, combinar una lectura de proximidad frontal con información lateral para generar una visión más completa del entorno inmediato.

La fusión básica alinea y superpone información sin inferencia compleja.

Desde el punto de vista arquitectónico, esta integración debe apoyarse completamente en la infraestructura común de marcos y transformaciones. Ningún módulo debería realizar transformaciones espaciales ad hoc o asumir relaciones geométricas implícitas. La fusión debe operar sobre datos ya expresados en un marco compartido.

La fusión debe apoyarse en la infraestructura común de transformaciones.

Otro aspecto relevante es la sincronización de flujos de datos. Cuando múltiples sensores publican información de manera asíncrona, la arquitectura debe definir políticas claras: seleccionar la medición más reciente disponible, interpolar estados intermedios o descartar datos fuera de una ventana temporal válida. Estas decisiones forman parte del diseño del sistema.

La arquitectura debe definir políticas explícitas de sincronización de sensores.

En términos estructurales, la fusión básica puede entenderse como un módulo que recibe múltiples flujos coherentes en espacio y tiempo y produce una representación integrada, también coherente. Esta representación puede ser una estructura geométrica ampliada, un mapa local o una estimación compuesta de la situación actual.

La fusión produce una representación integrada coherente en espacio y tiempo.

Sobre esta base —alineamiento espacial, consistencia temporal y representación unificada— se podrán introducir posteriormente mecanismos más avanzados de estimación y modelado. La fusión básica constituye, por tanto, el primer nivel de integración estructural dentro de la arquitectura perceptiva.

La fusión básica es el primer nivel estructural de integración perceptiva.

3.2. Aplicación

Gestión de sensores en ROS 2

En ROS 2, los sensores se integran en la arquitectura mediante el mecanismo de **publicación–suscripción** sobre **topics**. Cada sensor se encapsula típicamente en un nodo que adquiere datos del hardware y los publica periódicamente utilizando un tipo de mensaje estándar o específico.

Los sensores en ROS 2 se integran mediante publicación en topics.

El diseño sigue el principio de separación de responsabilidades descrito en la parte teórica: el nodo del sensor se encarga exclusivamente de adquirir y publicar datos, sin introducir interpretación ni procesamiento avanzado. Esta publicación se realiza con una frecuencia determinada por el dispositivo físico o por la configuración del nodo.

El nodo del sensor publica datos crudos sin añadir semántica.

Estandarización de mensajes

Uno de los pilares del ecosistema ROS es la **estandarización de tipos de mensaje** para familias de sensores. Por ejemplo, todos los escáneres láser 2D publican típicamente mensajes del tipo `sensor_msgs/LaserScan`, las cámaras utilizan `sensor_msgs/Image`, y las IMU emplean `sensor_msgs/Imu`.

ROS estandariza mensajes para familias completas de sensores.

Esta estandarización implica que cualquier sensor perteneciente a una misma familia debe poder expresar toda su información relevante dentro de la estructura del mensaje común. El diseño de estos mensajes no es arbitrario: es el resultado de discusiones técnicas y consensos dentro de

la comunidad ROS, buscando equilibrio entre generalidad, claridad y eficiencia.

Los mensajes estándar surgen de consensos comunitarios y buscan generalidad.

Gracias a esta uniformidad, los módulos consumidores —por ejemplo, algoritmos de percepción o localización— pueden operar de manera agnóstica respecto al fabricante o modelo concreto del sensor. El desacoplamiento entre hardware y software de alto nivel se apoya directamente en esta estandarización.

La estandarización permite desacoplar hardware y algoritmos.

Calidad de Servicio (QoS)

En ROS 2, la comunicación entre publicadores y suscriptores está gobernada por políticas de **Quality of Service (QoS)**. Estas políticas determinan el comportamiento de la transmisión en términos de fiabilidad, almacenamiento en cola, durabilidad y gestión histórica de mensajes.

La QoS define el comportamiento de la comunicación entre nodos.

Entre las políticas más relevantes se encuentran:

- **Reliability**: *reliable* o *best effort*.
- **History**: conservación del último mensaje o de un número limitado.
- **Depth**: tamaño de la cola asociada.
- **Durability**: persistencia frente a suscriptores tardíos.

Cada publicador ofrece un perfil de QoS y cada suscriptor solicita uno. La comunicación solo se establece si ambas configuraciones son compatibles. Cuando existe compatibilidad, la **QoS efectiva** es el resultado de la intersección entre lo ofrecido y lo solicitado.

La QoS efectiva resulta de la intersección entre oferta y solicitud.

Fiabilidad (Reliability)

En el caso de la política de fiabilidad, la compatibilidad puede resumirse como sigue:

Publicador	Suscriptor	QoS efectiva
Best Effort	Best Effort	Best Effort
Best Effort	Reliable	Sin conexión
Reliable	Best Effort	Best Effort
Reliable	Reliable	Reliable

La fiabilidad efectiva depende de la oferta del publicador y la solicitud del suscriptor.

Si el publicador ofrece *reliable* y el suscriptor solicita *best effort*, la comunicación se establece, pero la fiabilidad efectiva es *best effort*. En cambio, si el publicador ofrece *best effort* y el suscriptor solicita *reliable*, no existe conexión.

Esto muestra que el publicador establece el límite superior de garantías posibles, mientras que el suscriptor puede aceptar un nivel igual o inferior.

Durabilidad (Durability)

La política de durabilidad determina si el publicador conserva mensajes para suscriptores que se conectan con posterioridad. Existen dos modos principales:

- **Volatile**: los mensajes no se almacenan; los suscriptores tardíos no reciben datos anteriores.
- **Transient Local**: el publicador conserva los últimos mensajes y los entrega a nuevos suscriptores.

La compatibilidad se comporta del siguiente modo:

Publicador	Suscriptor	QoS efectiva
Volatile	Volatile	Volatile
Volatile	Transient Local	Sin conexión
Transient Local	Volatile	Volatile
Transient Local	Transient Local	Transient Local

La durabilidad efectiva también resulta de la intersección compatible.

Si el publicador ofrece *transient local* y el suscriptor solicita *volatile*, la comunicación se establece, pero la durabilidad efectiva es *volatile*. En cambio, si el publicador ofrece *volatile* y el suscriptor solicita *transient local*, no existe conexión.

Desde el punto de vista arquitectónico, *transient local* resulta especialmente útil para información estructural o casi estática, como parámetros globales o transformaciones fijas, ya que permite que nodos que se inician posteriormente reciban automáticamente el último estado publicado. Por el contrario, los sensores de alta frecuencia suelen utilizar *volatile*, dado que el interés reside en datos actuales y no en el histórico previo.

Transient Local es adecuado para información estructural; Volatile para flujos dinámicos.

En conjunto, estas tablas muestran que la QoS no solo afecta al rendimiento, sino también a la conectividad efectiva del sistema. Una configuración inadecuada puede impedir la comunicación incluso cuando el tipo de mensaje y el topic coinciden.

Por ello, la definición de QoS debe considerarse una decisión arquitectónica explícita y coherente con el papel funcional de cada componente dentro del sistema.

La QoS condiciona la conectividad y debe definirse arquitectónicamente.

Sistema de transformaciones en ROS 2

El sistema de transformaciones espaciales en ROS 2 materializa los principios teóricos presentados anteriormente: matrices homogéneas, dependencia temporal y organización en árbol. Conceptualmente, puede entenderse como una **infraestructura independiente de los nodos funcionales** cuya responsabilidad es mantener y resolver el árbol dinámico de marcos de referencia. En el ecosistema ROS 2, esta infraestructura se conoce habitualmente como TF (*Transform Frames*).

El sistema TF materializa la infraestructura espacial descrita teóricamente.

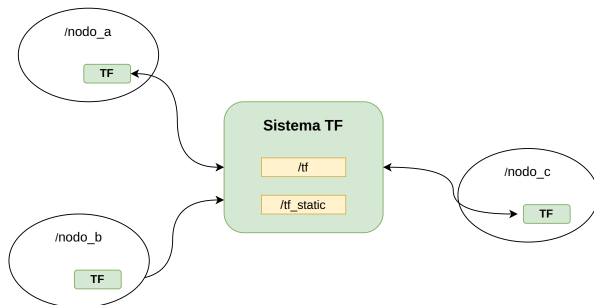


Figura 3.5: Sistema de TFs con sus topics, y nodos que lo usan a través de la infraestructura común.

Los nodos no intercambian transformaciones directamente entre sí. En su lugar, publican relaciones espaciales en topics específicos y consultan la infraestructura común cuando necesitan resolver una transformación entre marcos.

Los nodos publican y consultan transformaciones a través de la infraestructura común.

Fuentes de transformaciones

El árbol de transformaciones se construye a partir de múltiples fuentes que publican relaciones espaciales entre marcos. Estas fuentes pueden clasificarse en dos grandes categorías: transformaciones estructurales del robot y transformaciones dinámicas asociadas a su movimiento.

El árbol TF se construye a partir de fuentes estructurales y dinámicas.

Geometría estructural del robot

La estructura geométrica del robot (Fig. 3.6) se describe habitualmente mediante formatos como **URDF** o **SDF**. Estos modelos especifican enlaces rígidos, articulaciones y transformaciones fijas entre componentes físicos.

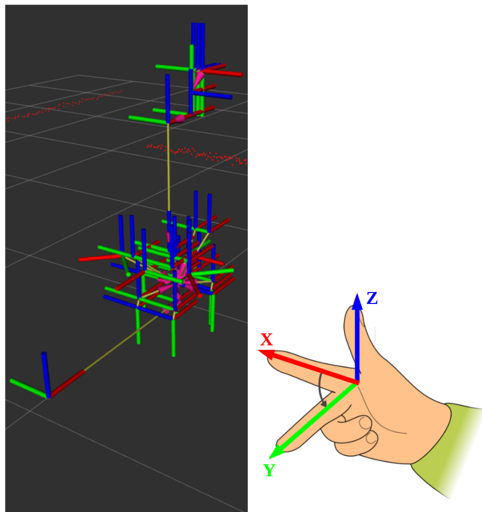


Figura 3.6: Marcos de referencia y transformaciones en un robot móvil (izquierda) y convención dextrógira (derecha).

El componente **robot_state_publisher** utiliza esta descripción junto con el estado actual de las articulaciones para publicar automáticamente las transformaciones correspondientes entre los distintos enlaces del robot. De este modo, la estructura interna del robot queda representada de manera sistemática dentro del árbol de referencias.

URDF y robot_state_publisher generan las transformaciones internas del robot.

Transformaciones dinámicas

Las relaciones espaciales que varían en el tiempo —como la pose del robot respecto al entorno— son publicadas por nodos encargados de estimación o control, tales como fuentes de odometría, sistemas de localización o estimadores de estado.

En entornos simulados, los simuladores desempeñan exactamente el mismo papel, publicando transformaciones dinámicas que representan la evolución del robot en el mundo virtual. Desde el punto de vista del sistema TF, no existe diferencia conceptual entre una transformación generada por hardware real o por simulación: ambas alimentan el mismo árbol espacial.

Controladores, estimadores y simuladores publican las transformaciones dinámicas.

Convenciones estándar: REP 105

La interoperabilidad entre componentes software requiere no solo coherencia geométrica, sino también coherencia semántica en la denominación de marcos de referencia. El documento REP 105 (`https://ros.org/reps/rep-0105.html`) establece una convención ampliamente adoptada para la organización del árbol de transformaciones en robots móviles (Fig. 3.7).

REP 105 define una convención semántica para estructurar el árbol de referencias.

Esta convención distingue explícitamente entre referencias globales, referencias locales continuas y referencias asociadas al cuerpo físico del robot. Entre los marcos más relevantes se encuentran:

- **map**: marco global fijo en el entorno, asociado a una representación absoluta del espacio.
- **odom**: marco local continuo que evoluciona suavemente en el tiempo, pero que puede acumular error.
- **base_link**: marco rígidamente unido al cuerpo principal del robot, describiendo su pose completa en el espacio tridimensional.
- **base_footprint**: marco planar derivado de `base_link`, que elimina componentes de inclinación respecto al suelo.

map, odom, base_link y base_footprint estructuran la referencia espacial del robot móvil.

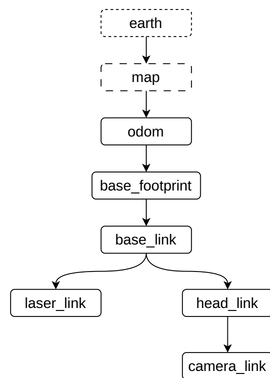


Figura 3.7: Convenciones estándar para la organización del árbol de transformaciones en robots móviles según REP 105.

La separación entre `map` y `odom` refleja una distinción arquitectónica fundamental: continuidad frente a corrección global. El marco `odom` proporciona una referencia suave y localmente consistente, adecuada para control y estimación incremental. El marco `map`, en cambio, introduce correcciones absolutas que pueden generar discontinuidades cuando se actualiza la estimación global.

odom garantiza continuidad local; map introduce corrección global.

Por su parte, la distinción entre `base_link` y `base_footprint` responde a una separación entre representación tridimensional completa y representación planar simplificada. Mientras que `base_link` conserva roll, pitch y yaw, el marco `base_footprint` proyecta el robot sobre el plano del suelo, facilitando el razonamiento en $SE(2)$ cuando el problema es esencialmente bidimensional.

base_footprint permite razonamiento planar desacoplado de la orientación 3D.

Estas convenciones no son meramente nominales. Constituyen un acuerdo estructural que permite que distintos módulos —localización, planificación, control— operen sobre referencias compartidas sin ambigüedad. La adopción consistente de esta jerarquía favorece la modularidad y reduce acoplamientos implícitos en arquitecturas complejas.

Las convenciones estándar refuerzan modularidad y coherencia arquitectónica.

Publicación y recepción de TF

Las transformaciones se publican en dos topics diferenciados:

- **/tf**: transformaciones dinámicas.
- **/tf_static**: transformaciones estáticas.

Las transformaciones estáticas se publican una sola vez y permanecen válidas durante toda la ejecución, mientras que las dinámicas se actualizan periódicamente.

Las TF estáticas y dinámicas se publican en topics distintos.

Cada nodo que necesita consultar transformaciones mantiene internamente un **buffer temporal** donde se almacenan las transformaciones recibidas junto con su marca temporal. Este buffer conserva un historial finito de datos, permitiendo realizar consultas retrospectivas dentro de una ventana temporal determinada (10 segundos, por defecto).

Cada nodo mantiene un buffer temporal con el historial reciente de transformaciones.

Cuando se solicita una transformación entre dos marcos en un instante concreto t, el sistema no se limita a buscar una relación directa. En primer lugar, identifica el camino único en el árbol que conecta ambos marcos. Este camino puede involucrar múltiples transformaciones intermedias.

Para cada una de esas relaciones padre–hijo, el sistema recupera la transformación válida en el instante solicitado. Dado que las transformaciones se publican de manera discreta y a frecuencias potencialmente distintas (Fig. 3.8), es habitual que no exista una muestra exacta en el instante t. En ese caso, se realiza **interpolación temporal** entre las muestras más próximas disponibles.

Cada transformación intermedia se interpola temporalmente si no existe muestra exacta.

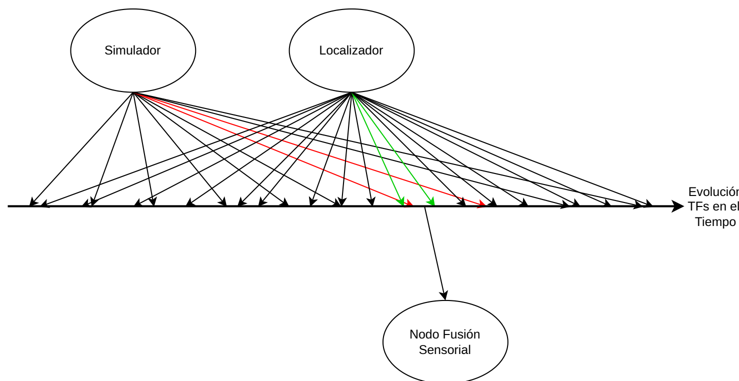


Figura 3.8: El nodo Simulador y Localizador publican TFs en diferentes instantes temporales. Cuando se solicita una transformación en un instante intermedio, el sistema interpola cada tramo del camino para obtener una consulta coherente.

Una vez interpoladas todas las transformaciones elementales en el instante t, se procede a su composición para obtener la transformación final solicitada. Es importante destacar que la interpolación se aplica individualmente a cada tramo del árbol antes de realizar la composición global.

La interpolación se realiza por tramo antes de la composición final.

Este procedimiento refleja la naturaleza distribuida del sistema: distintas transformaciones pueden haber sido publicadas por nodos diferentes, con frecuencias distintas y con ligeros desfases temporales. El buffer temporal permite armonizar estas publicaciones discretas en una consulta coherente.

El sistema armoniza publicaciones discretas y asíncronas mediante interpolación.

Sin embargo, el sistema no permite **extrapolaciones hacia el futuro**. Si la consulta solicita una transformación en un instante posterior a la última muestra disponible de cualquiera de los tramos intermedios, la operación falla y se genera un error. Esta restricción evita introducir hipótesis dinámicas implícitas sobre el movimiento del robot o del entorno.

No se permiten extrapolaciones futuras para evitar hipótesis dinámicas implícitas.

En consecuencia, el uso correcto del sistema de TF exige coherencia temporal entre los datos sensoriales y las consultas espaciales. La infraestructura garantiza consistencia geométrica dentro del intervalo temporal disponible, pero no introduce predicciones ni modelos de movimiento más allá de los datos publicados.

El sistema garantiza coherencia dentro del intervalo disponible, pero no realiza predicción.

Publicación de TF propias

El sistema permite que cualquier usuario publique sus propias transformaciones. Esto resulta especialmente útil para representar entidades percibidas, como objetos detectados o referencias externas.

Por ejemplo, un módulo de percepción puede publicar un nuevo marco asociado a un objeto detectado, posicionándolo respecto a `base_link` o `map`. De este modo, el objeto pasa a formar parte del árbol espacial y puede ser utilizado por planificadores o módulos de control.

Los usuarios pueden añadir nuevos marcos al árbol publicando TF propias.

Esta capacidad convierte al sistema de transformaciones en una infraestructura extensible que no solo describe el robot, sino también su interpretación dinámica del entorno.

El sistema TF permite extender dinámicamente el modelo espacial del entorno.

En conjunto, el sistema de transformaciones de ROS 2 materializa los principios matemáticos y arquitectónicos estudiados previamente: representación homogénea, composición en árbol, dependencia temporal e interpolación coherente. Constituye una infraestructura transversal esencial para integrar sensores, estimaciones y decisiones dentro de un espacio geométrico común.

El sistema TF conecta teoría y práctica y soporta la integración de sensores.

Sensores habituales en ROS 2

Escáner láser 2D

Los escáneres láser 2D, comúnmente denominados LIDAR, miden distancias mediante la emisión de pulsos láser y la medición del tiempo que tarda la señal reflejada en regresar al sensor. A partir del tiempo de vuelo se calcula la distancia a los objetos presentes en el plano de escaneo.

Un LIDAR 2D mide distancias mediante tiempo de vuelo en un plano fijo.

El sensor realiza un barrido angular, generando un conjunto ordenado de mediciones polares. Cada lectura corresponde a una distancia r_i asociada a un ángulo θ_i dentro de un rango definido. El resultado no es una imagen ni un mapa, sino una secuencia de distancias expresadas en coordenadas polares en el marco del sensor.

El láser produce distancias polares ordenadas en el marco del sensor.

En ROS 2, estos sensores publican típicamente mensajes del tipo `sensor_msgs/LaserScan`. Este mensaje incluye:

- Ángulo inicial del barrido (`angle_min`).
- Ángulo final (`angle_max`).
- Incremento angular (`angle_increment`).
- Vector de distancias (`ranges`).
- Marca temporal asociada al inicio del escaneo.

LaserScan incluye metadatos angulares, distancias y marca temporal.

Estructura del mensaje `sensor_msgs/msg/LaserScan` Esta es la estructura del mensaje `LaserScan`, con un resumen de sus campos más relevantes:

```
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

float32 angle_min
float32 angle_max
float32 angle_increment

float32 time_increment
```

```
float32 scan_time

float32 range_min
float32 range_max

float32[] ranges
float32[] intensities
```

Fragmento C++ 3.1: Estructura de sensor_msgs/msg/LaserScan (resumen de campos)

La razón de cada campo es la siguiente:

- `header.stamp`: referencia temporal de adquisición del escaneo. Es esencial para fusionar el láser con TF y con otros sensores.
- `header.frame_id`: marco del sensor donde se interpretan los ángulos y las distancias. Permite proyectar el escaneo a `base_link`, `odom` o `map` usando TF.
- `angle_min`, `angle_max`, `angle_increment`: definen la geometría del barrido y permiten asociar un ángulo a cada índice i del vector `ranges`.
- `time_increment`: tiempo entre rayos consecutivos dentro del mismo escaneo; es clave para corregir distorsiones si el robot se mueve durante el barrido.
- `scan_time`: periodo aproximado entre escaneos completos; útil para sincronización y para estimar frecuencia efectiva del sensor.
- `range_min`, `range_max`: límites físicos/operativos del sensor; ayudan a filtrar lecturas inválidas o fuera de rango.
- `ranges`: distancias medidas (en metros) asociadas a los ángulos del barrido; es el dato principal para evitación, SLAM 2D o navegación planar.
- `intensities`: medida adicional (dependiente del dispositivo) asociada al retorno del rayo; se usa cuando el sensor la proporciona (p.ej., para detectar reflectores).

Dado un índice i del vector de distancias, el ángulo asociado se calcula como:

$$\theta_i = \texttt{angle_min} + i \cdot \texttt{angle_increment}$$

y la coordenada polar correspondiente es (r_i, θ_i).

Para convertir esta medición a coordenadas cartesianas en el marco del sensor se emplean las relaciones geométricas básicas:

$$x_i = r_i \cos(\theta_i)$$
$$y_i = r_i \sin(\theta_i)$$

De este modo, cada medición polar se transforma en un punto (x_i, y_i) en el plano del escaneo.

Cada medición polar se convierte en coordenadas cartesianas mediante trigonometría básica.

Interpretación espacial y temporal

Supongamos que en el índice $i = 180$ de un escaneo con incremento angular de 0,5° se obtiene una distancia de $r_{180} = 2{,}0\,\text{m}$. Si el ángulo inicial es −90°, entonces:

$$\theta_{180} = -90^\circ + 180 \cdot 0{,}5^\circ = 0^\circ$$

En este caso, el punto detectado se encuentra a dos metros directamente frente al sensor en su propio marco de referencia.

No obstante, la interpretación correcta requiere considerar también la dimensión temporal. En el mensaje `LaserScan`, la marca temporal almacenada en `header.stamp` corresponde al instante de adquisición del primer rayo del escaneo. El mensaje incluye además dos parámetros temporales adicionales:

- `scan_time`: tiempo total necesario para completar un barrido completo.
- `time_increment`: intervalo temporal entre dos mediciones consecutivas.

Por tanto, el instante exacto en el que se adquirió la medición i se calcula como:

$$t_i = t_0 + i \cdot \texttt{time_increment}$$

donde t_0 es la marca temporal del encabezado.

En el ejemplo anterior, si `time_increment` = 0,0001 s, entonces la medición correspondiente al índice 180 se obtuvo en:

$$t_{180} = t_0 + 180 \cdot 0{,}0001 = t_0 + 0{,}018\,\text{s}$$

Esto implica que el punto no representa la posición del entorno en el instante t_0, sino en un instante ligeramente posterior dentro del mismo barrido.

Cada medición del láser tiene un instante temporal propio dentro del escaneo.

Si el robot está en movimiento durante el escaneo, cada rayo corresponde a una pose ligeramente distinta del sensor. Para expresar correctamente el punto en el marco del robot o en un marco global, debe utilizarse la transformación espacial válida en el instante t_i, no simplemente en t_0.

Desde el punto de vista arquitectónico, esto obliga a consultar la infraestructura de transformaciones con resolución temporal suficiente para interpolar la pose del sensor en el instante exacto de cada medición.

La correcta proyección espacial exige utilizar la transformación correspondiente al instante exacto de cada rayo.

Este detalle, aparentemente menor, resulta crítico en sistemas dinámicos: ignorar la evolución temporal durante el barrido puede introducir distorsiones geométricas apreciables, especialmente a altas velocidades de movimiento.

Ignorar la dimensión temporal del escaneo introduce errores geométricos.

El LIDAR 2D constituye así una fuente estructurada y ligera de información espacial, especialmente adecuada para navegación planar y evitación de obstáculos en entornos bidimensionales.

El LIDAR 2D es clave en navegación planar y evitación de obstáculos.

Cámaras e imágenes

Las cámaras capturan información luminosa proyectada sobre un sensor bidimensional. Desde el punto de vista geométrico, el modelo más utilizado es el **modelo pin-hole** (Fig. 3.9), que aproxima el proceso de formación de imagen como una proyección perspectiva desde el espacio tridimensional al plano de imagen.

El modelo pin-hole describe la proyección perspectiva 3D→2D.

En este modelo, la relación entre un punto 3D y su proyección 2D depende de los parámetros intrínsecos de la cámara, que incluyen distancia focal, centro óptico y posibles coeficientes de distorsión. Estos parámetros se estiman mediante un proceso de **calibración intrínseca**.

La calibración intrínseca estima parámetros ópticos de la cámara.

La calibración no corrige el contenido de la imagen en sí, sino que proporciona los parámetros necesarios para interpretar geométricamente

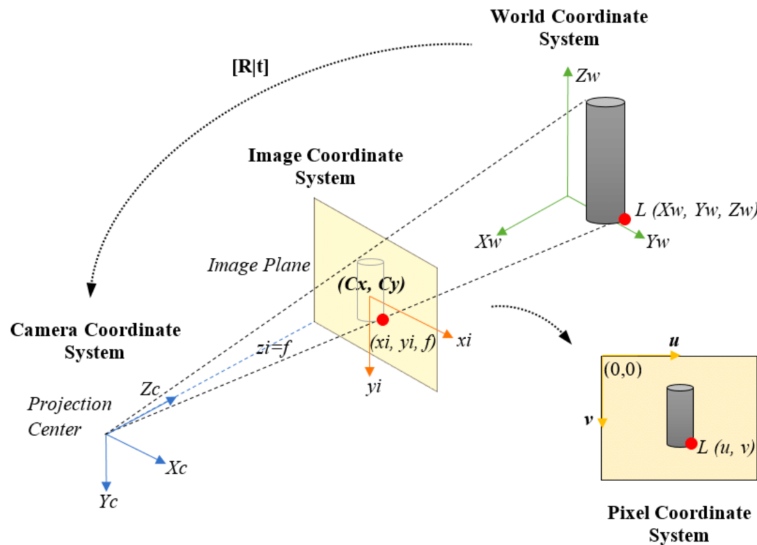


Figura 3.9: Modelo pin-hole de formación de imagen. Un punto 3D L se proyecta sobre el plano de imagen en un punto a través del centro óptico C.

los píxeles. Estos parámetros se almacenan en mensajes del tipo `sensor_msgs/CameraInfo`, que acompañan al topic de imagen.

CameraInfo contiene los parámetros intrínsecos necesarios para interpretación geométrica.

Las imágenes propiamente dichas se publican mediante mensajes `sensor_msgs/Image`, que contienen una matriz de píxeles junto con información sobre codificación y formato.

Image contiene la matriz de píxeles y su codificación.

Estructura del mensaje `sensor_msgs/msg/Image` Esta es la estructura del mensaje `Image`, con un resumen de sus campos principales:

```
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

uint32 height
uint32 width

string encoding

uint8 is_bigendian
uint32 step
uint8[] data
```

Fragmento C++ 3.2: Estructura de sensor_msgs/msg/Image (resumen de campos)

La razón de cada campo es la siguiente:

- `header.stamp`: instante de adquisición de la imagen. Es crítico para sincronización con `CameraInfo`, TF y otros sensores.
- `header.frame_id`: marco óptico asociado a la cámara. Permite relacionar píxeles/medidas con la geometría 3D del robot mediante TF.
- `height`, `width`: tamaño de la imagen en píxeles (filas y columnas). Determina la dimensión de la matriz.
- `encoding`: describe cómo interpretar los bytes (número de canales, orden, profundidad). Ejemplos típicos: `rgb8`, `bgr8`, `mono8`, `16UC1` (profundidad).
- `is_bigendian`: orden de bytes para codificaciones con palabras de más de 8 bits; relevante en imágenes de 16 bits o superiores.
- `step`: número de bytes por fila (*row stride*). Permite representar filas con padding y calcular el inicio de cada fila dentro de `data`.

- `data`: *blob* de bytes con los píxeles, de tamaño aproximado `step * height`. Su interpretación depende completamente de `encoding`.

ImageTransport y transporte eficiente de imágenes Las imágenes crudas (`sensor_msgs/Image`) pueden ser muy costosas de transmitir: incluso resoluciones moderadas a decenas de Hz saturan el ancho de banda de red o el bus interno del robot. Para abordar esto, ROS proporciona el mecanismo **ImageTransport** (paquete `image_transport`), que permite publicar una misma fuente de imagen bajo un **topic base** y, de forma simultánea, ofrecer **transportes alternativos** mediante topics derivados.

ImageTransport permite publicar un topic base de imagen y ofrecer transportes alternativos.

Conceptualmente, el productor publica en un topic como `/camera/image` (transporte `raw`) y, mediante plugins, pueden aparecer variantes como `/camera/image/compressed`, `/camera/image/theora` (y, en algunas configuraciones, transportes específicos para profundidad como `/camera/depth/image/compressedDepth`). El punto clave es que el consumidor puede elegir el transporte **sin cambiar su lógica perceptiva**: usando `image_transport` en el suscriptor, la selección se hace por configuración y el sistema resuelve de manera transparente qué topic y qué codificación se emplean.

La elección de transporte se realiza en el suscriptor por configuración y de forma transparente.

Desde el punto de vista arquitectónico, esto introduce una separación explícita entre **contrato de información** ("una imagen con su `header`") y **contrato de transporte** (forma de serialización/compresión y coste asociado). Las implicaciones principales son:

ImageTransport separa el significado de la imagen de los detalles del transporte y la serialización.

- **Intercambio CPU–ancho de banda**: compresión reduce tráfico, pero añade latencia y carga de cómputo (en el emisor y/o receptor).
- **Desacoplamiento y despliegue**: permite mover nodos entre máquinas o redes distintas manteniendo el mismo topic base, eligiendo el transporte según el enlace disponible.
- **Observabilidad y reproducibilidad**: al grabar con rosbag o depurar, es relevante saber qué transporte se está consumiendo, ya que determina calidad, pérdidas y retardos.
- **Coherencia temporal**: aunque cambie el transporte, deben preservarse `header.stamp` y `frame_id` para mantener sincronización con `CameraInfo`, TF y otros sensores.

En resumen, ImageTransport permite optimizar la comunicación de imágenes sin acoplar a los consumidores al detalle de cómo se transportan, convirtiendo el transporte en una decisión de arquitectura y despliegue.

ImageTransport desacopla el contrato de imagen del transporte y permite elegir compresión según requisitos de red, CPU y latencia.

Espacios de color y robustez frente a iluminación En robótica, muchas tareas perceptivas iniciales (seguimiento de objetos por color, detección de marcas, segmentación rápida) se basan en umbrales y métricas de similitud de color. En un espacio RGB (o BGR, común en OpenCV), las componentes mezclan color y luminancia: cambios de iluminación, sombras o autoexposición afectan simultáneamente a R, G y B, haciendo que umbrales simples sean poco estables.

Una alternativa habitual es transformar la imagen a **HSV** (Hue–Saturation–Value) (Fig. 3.10), donde el componente de matiz (H) captura el tono puro del color, mientras que la saturación (S) y el valor (V) representan la pureza y la intensidad respectivamente. Esta separación permite diseñar detectores más robustos a variaciones de iluminación, ya que

el matiz puede permanecer relativamente constante incluso cuando el brillo cambia.:

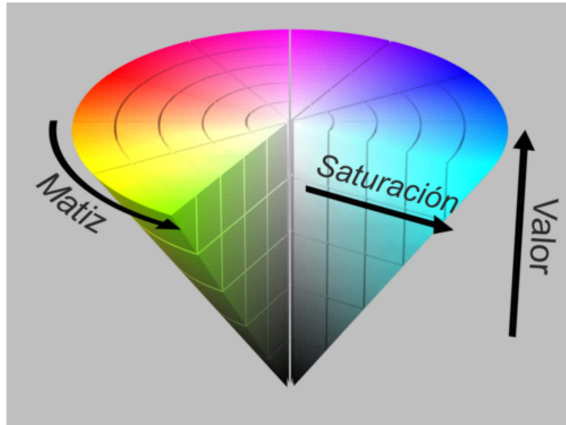


Figura 3.10: Modelo HSV de coloración. Un punto en el espacio de color se representa mediante los componentes Matiz, Saturación y Valor. (imagen adaptada de `https://es.wikipedia.org/wiki/Modelo_de_color_HSV`).

- **H (Hue)** captura el *tono* ("qué color") y es relativamente estable cuando cambia la intensidad global.
- **S (Saturation)** captura cuán "puro.[es] el color (frente a grises). Ayuda a descartar regiones poco informativas.
- **V (Value)** captura la luminancia/brillo, donde se concentran muchas variaciones por iluminación.

Esta separación permite diseñar detectores más robustos: por ejemplo, segmentar por rangos de H y S y tolerar variaciones en V suele funcionar mejor que umbralizar directamente en RGB. Arquitectónicamente, esto es una **transformación perceptiva** temprana que aumenta la invariancia a condiciones externas y reduce sensibilidad a cambios de exposición.

En la práctica, el procesamiento de imágenes no se realiza directamente sobre los mensajes ROS, sino mediante bibliotecas especializadas como **OpenCV**. Los mensajes se convierten a estructuras compatibles con OpenCV, donde se aplican algoritmos de filtrado, detección o reconstrucción.

El procesamiento de imágenes se realiza habitualmente mediante OpenCV.

Desde el punto de vista arquitectónico, la cámara introduce un flujo de datos de alta dimensionalidad que requiere procesamiento intensivo, lo que influye directamente en decisiones de QoS, sincronización y distribución computacional.

Las cámaras generan grandes volúmenes de datos que impactan la arquitectura.

Sensores láser 3D y nubes de puntos

Los sensores láser tridimensionales extienden el principio del LIDAR 2D generando mediciones en múltiples planos o mediante rotación mecánica. El resultado es una representación espacial en tres dimensiones del entorno.

El LIDAR 3D genera mediciones espaciales completas en 3D.

En ROS 2, estos sensores publican típicamente mensajes del tipo `sensor_msgs/PointCloud2`. Este mensaje representa una colección estructurada de puntos, donde cada punto puede contener no solo coordenadas (x, y, z), sino también intensidad u otros atributos.

PointCloud2 representa colecciones estructuradas de puntos 3D.

Estructura del mensaje `sensor_msgs/msg/PointCloud2` Esta es la estructura del mensaje `PointCloud2`, con un resumen de sus campos principales:

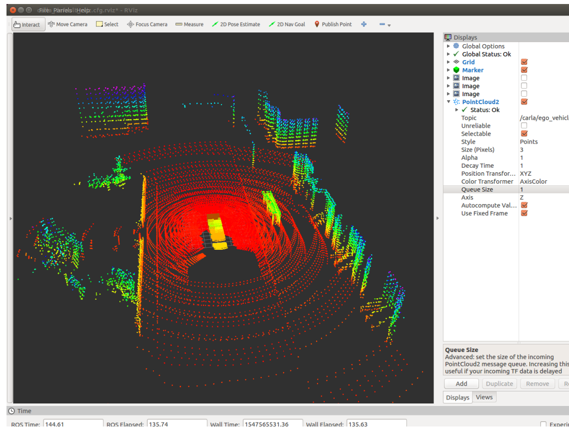


Figura 3.11: PointCloud2 representa una nube de puntos 3D, donde cada punto puede contener coordenadas y atributos adicionales como intensidad o color.

```
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

uint32 height
uint32 width

PointField[] fields
    uint8 INT8    = 1
    uint8 UINT8   = 2
    uint8 INT16   = 3
    uint8 UINT16  = 4
    uint8 INT32   = 5
    uint8 UINT32  = 6
    uint8 FLOAT32 = 7
    uint8 FLOAT64 = 8
    uint8 INT64   = 9
    uint8 UINT64  = 10
    uint8 BOOL    = 11
    string name
    uint32 offset
    uint8  datatype
    uint32 count

bool    is_bigendian
uint32  point_step
uint32  row_step
uint8[] data

bool is_dense
```

Fragmento C++ 3.3: Estructura de sensor_msgs/msg/PointCloud2 (resumen de campos)

La razón de cada campo es la siguiente:

- `header.stamp` y `header.frame_id`: sitúan la nube en tiempo y en un marco. Permiten transformar puntos a otros marcos con TF y fusionar con odometría/localización.
- `height` y `width`: describen si la nube es organizada (2D, tipo imagen) o desordenada (habitualmente `height=1`, `width=N`).
- `fields`: define el *layout* del dato binario. Cada `PointField` describe un canal (p.ej. `x`, `y`, `z`, `intensity`) indicando nombre, desplazamiento (`offset`), tipo (`datatype`) y multiplicidad (`count`).
- `is_bigendian`: orden de bytes del *blob* `data`.
- `point_step`: tamaño en bytes de un punto (stride entre puntos consecutivos).
- `row_step`: tamaño en bytes de una fila completa; en nubes organizadas permite saltar filas eficientemente.

- `data`: *blob* con todos los puntos; se interpreta usando `fields`, `point_step` y `row_step`.
- `is_dense`: indica si hay puntos inválidos (NaN/Inf). Si es `false`, los algoritmos deben estar preparados para filtrar.

A diferencia de `LaserScan`, que es esencialmente un vector de distancias, `PointCloud2` es una estructura más general y flexible, capaz de representar distintos tipos de sensores tridimensionales.

PointCloud2 es una estructura general para sensores tridimensionales.

El procesamiento de nubes de puntos no se realiza directamente sobre el mensaje, sino mediante bibliotecas especializadas como la **Point Cloud Library (PCL)**. Esta librería proporciona herramientas para filtrado, segmentación, extracción de planos y registro entre nubes.

El procesamiento de nubes de puntos se realiza mediante PCL.

Desde el punto de vista arquitectónico, las nubes de puntos representan uno de los flujos de datos más pesados del sistema, lo que exige decisiones cuidadosas sobre frecuencia, filtrado previo y distribución computacional.

Las nubes de puntos imponen fuertes requisitos computacionales.

ROSBag

ROS 2 proporciona un mecanismo para registrar y reproducir flujos de datos denominado **rosbag**. Un rosbag permite almacenar mensajes publicados en topics durante la ejecución del sistema y reproducirlos posteriormente como si provinieran de sensores reales.

ROSBag permite registrar y reproducir flujos de mensajes.

Esta herramienta resulta fundamental para depuración, experimentación reproducible y validación de algoritmos. Permite separar adquisición física de desarrollo algorítmico, facilitando pruebas repetibles sin necesidad de disponer del hardware en tiempo real.

ROSBag facilita experimentación reproducible y desacoplada del hardware.

Arquitectónicamente, rosbag actúa como una fuente alternativa de datos dentro del sistema, pudiendo reemplazar sensores físicos durante fases de desarrollo o evaluación.

ROSBag puede sustituir temporalmente a sensores físicos en la arquitectura.

De sensores a detecciones: reducción de complejidad

Una vez que el sistema dispone de datos sensoriales coherentes en espacio y tiempo (sensores crudos, sincronización y TF), un paso natural es introducir un procesamiento posterior cuyo objetivo no sea "ver más", sino **reducir la complejidad** de la información para que pueda ser consumida por módulos de **control** o **deliberación**. Esta idea conecta con el principio general de procesamiento de la información: transformar representaciones de alta dimensionalidad en representaciones compactas, con estructura y significado operativo.

Tras la etapa sensorial, es útil reducir complejidad para habilitar control y deliberación.

En la práctica, esto equivale a sustituir flujos densos (imágenes, nubes de puntos) por **conjuntos pequeños de entidades** que resuman lo relevante para la tarea: "hay una persona aquí", "hay una señal allí", "hay un obstáculo a 1.2 m". Existen muchas opciones posibles (mapas de ocupación, líneas/planos segmentados, landmarks, tracks), pero una opción especialmente extendida en percepción moderna es producir **detecciones de objetos**.

La reducción de complejidad sustituye datos densos por entidades compactas y relevantes.

Opción: publicar detecciones con `vision_msgs`

ROS 2 dispone del paquete `vision_msgs`, que define mensajes pensados para comunicar resultados típicos de percepción basada en visión o en 3D: listas de detecciones con hipótesis de clase, puntuaciones y localización.

Estos mensajes permiten desacoplar la etapa de percepción (costosa y dependiente del sensor) de los módulos que toman decisiones (que necesitan entradas compactas y estables).

vision_msgs permite expresar resultados perceptivos compactos y desacoplados del sensor.

Los tipos más habituales para este propósito son:

- `vision_msgs/msg/Detection2D` y `vision_msgs/msg/Detection2DArray`.
- `vision_msgs/msg/Detection3D` y `vision_msgs/msg/Detection3DArray`.

Estructura de `vision_msgs/msg/Detection2D` (resumen) Esta es la estructura del mensaje `Detection2D`, con un resumen de sus campos más relevantes.

```
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

ObjectHypothesisWithPose[] results
    ObjectHypothesis hypothesis
        string class_id
        float64 score
    geometry_msgs/PoseWithCovariance pose

BoundingBox2D bbox
    vision_msgs/Pose2D center
        vision_msgs/Point2D position
            float64 x
            float64 y
        float64 theta
    float64 size_x
    float64 size_y

string id
```

Fragmento C++ 3.4: Estructura de vision_msgs/msg/Detection2D (resumen de campos)

Este mensaje permite describir una detección 2D como una **caja en imagen** (`bbox`) y un conjunto de **hipótesis de clase** con puntuaciones (`results`). En algunos pipelines, la pose asociada en `results.pose` puede usarse para adjuntar una estimación espacial (por ejemplo, proyectando con profundidad), aunque el uso dominante de `Detection2D` es como salida de detectores sobre imagen.

Detection2D resume imagen en cajas y clases; opcionalmente puede asociar una pose.

Estructura de `vision_msgs/msg/Detection2DArray` (resumen) Esta es la estructura del mensaje `Detection2DArray`, con un resumen de sus campos más relevantes.

```
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

Detection2D[] detections
```

Fragmento C++ 3.5: Estructura de vision_msgs/msg/Detection2DArray (resumen de campos)

El mensaje `Detection2DArray` es el contenedor habitual para detectores multi-objeto: un **conjunto de propuestas** derivadas de una misma observación.

Estructura de `vision_msgs/msg/Detection3D` (resumen) Esta es la estructura del mensaje `Detection3D`, con un resumen de sus campos más relevantes.

```
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

ObjectHypothesisWithPose[] results
    ObjectHypothesis hypothesis
        string class_id
        float64 score
    geometry_msgs/PoseWithCovariance pose

BoundingBox3D bbox
    geometry_msgs/Pose center
    geometry_msgs/Vector3 size

string id
```

Fragmento C++ 3.6: Estructura de vision_msgs/msg/Detection3D (resumen de campos)

En 3D, la salida ya está directamente formulada como **pose** y **caja 3D** (tamaño y orientación), lo que resulta especialmente útil para manipulación, navegación en 3D o para integrar objetos en un modelo del entorno.

Estructura de `vision_msgs/msg/Detection3DArray` (resumen) Esta es la estructura del mensaje `Detection3DArray`, con un resumen de sus campos más relevantes.

```
std_msgs/Header header
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id

Detection3D[] detections
```

Fragmento C++ 3.7: Estructura de vision_msgs/msg/Detection3DArray (resumen de campos)

Implicaciones arquitectónicas

Publicar detecciones en lugar de datos densos cambia el contrato entre módulos:

- **Reducción drástica de dimensionalidad**: de millones de píxeles o cientos de miles de puntos a decenas de detecciones.
- **Consumibilidad**: control y deliberación operan sobre listas pequeñas, con semántica explícita (clase, score, caja, pose).
- **Trazabilidad temporal y espacial**: `header.stamp` y `header.frame_id` permiten sincronizar con TF y con otras fuentes.
- **Incertidumbre explícita**: el uso de `PoseWithCovariance` permite representar que la salida es una estimación.
- **Separación de responsabilidades**: el detector asume el coste y la complejidad de interpretar el sensor; los consumidores no necesitan conocer el formato del sensor.

Las detecciones son una interfaz compacta, trazable y desacoplada para decisiones.

Un detalle importante es que estos mensajes **no incluyen los datos fuente** (imagen, nube de puntos). Esto es intencional: evita duplicar información pesada y fuerza a que la asociación con el dato original se haga mediante **sincronización temporal** (misma marca temporal) si fuese necesaria para depuración o para tareas posteriores.

Las detecciones no arrastran datos densos; se relacionan con el origen por sincronización temporal.

En resumen, producir `Detection2DArray` o `Detection3DArray` constituye un ejemplo claro de procesamiento posterior al sensor cuyo objetivo es **hacer la información operativa** para el resto de la arquitectura, reduciendo complejidad y estabilizando contratos de datos entre percepción y decisión. . En la Fifura 3.12 se muestran tres ejemplos de pipelines

Publicar detecciones es un paso posterior al sensor que vuelve operativa la información.

de percepción que transforman datos sensoriales crudos en detecciones operativas, utilizando los mensajes y mecanismos descritos en esta sección.

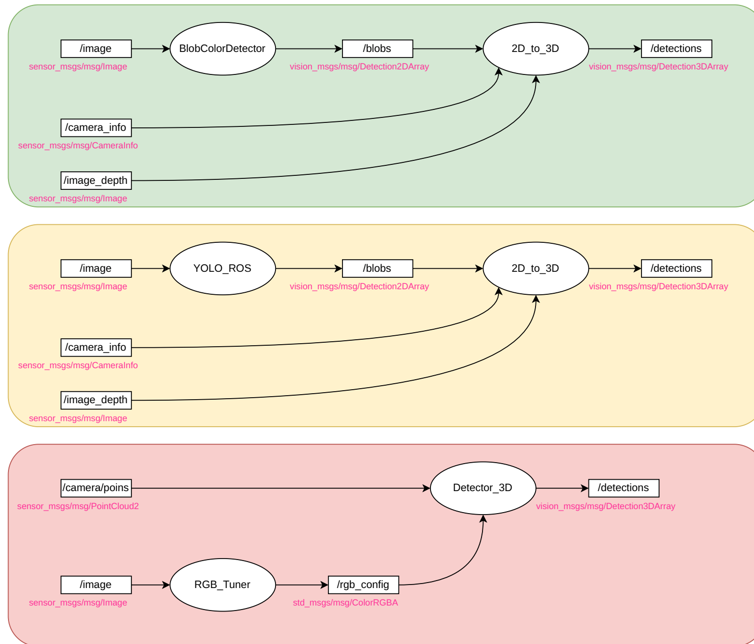


Figura 3.12: Tres ejemplos de pipelines de percepción que transforman datos sensoriales crudos en detecciones operativas.

4 Arquitecturas deliberativas e híbridas: capas y modelos del mundo

4.1. Principios teóricos

4.1 Principios teóricos 50
De lo reactivo a lo deliberativo e híbrido 50
Arquitecturas en capas: misión, tarea y capacidad . 52
Aproximaciones por comportamientos: subsumption y arbitraje reactivo 55
Modelos del mundo 56
4.2 Aplicación 57
Mapeo de capas a un grafo de ROS 2 57
Lifecycle Nodes y subsumption 59
Construcción del modelo del mundo en ROS 2 61

De lo reactivo a lo deliberativo e híbrido

En los capítulos anteriores hemos trabajado con modelos de ejecución mayoritariamente reactivos: el robot transforma eventos y flujos de datos en acciones con poca o ninguna representación explícita de objetivos futuros. Este enfoque es natural en robótica porque los sensores son asíncronos, el entorno cambia, y muchos bucles de control exigen respuesta inmediata. Arquitectónicamente, lo reactivo suele expresarse como "si ocurre X, haz Y", lo cual simplifica la implementación y funciona bien como base de reflejos.

El punto débil aparece cuando el sistema debe sostener comportamiento coherente durante minutos u horas, u optimizar decisiones bajo restricciones globales. Surgen fenómenos típicos: **miopía temporal** (lo local no conduce a lo global), **oscilaciones** (alternar entre respuestas incompatibles) y **deuda de estado** (contexto implícito repartido en callbacks y flags). En esos casos, el sistema puede estar "siempre reaccionando" sin avanzar hacia un objetivo.

En misiones largas, la reactividad pura induce miopía, oscilaciones y "deuda de estado" difícil de mantener.

Los modelos **deliberativos** introducen una idea clave: el robot mantiene una representación interna del problema (estado, objetivos, restricciones) y utiliza mecanismos de planificación o razonamiento para seleccionar cursos de acción (Fig. 4.1). No significa planificar una vez y ejecutar sin mirar: el valor está en poder comparar alternativas con criterio y justificar por qué se eligió una, dado un estado del mundo y una política.

Lo deliberativo explicita estado, objetivos y restricciones para poder elegir y justificar cursos de acción.

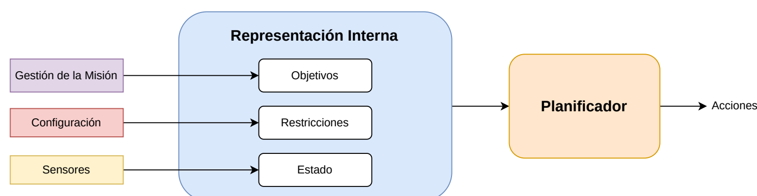


Figura 4.1: Modelos deliberativos: el robot mantiene una representación interna sobre la que planifica acciones.

Para que la deliberación sea útil, el sistema debe responder tres preguntas: (1) *¿qué sé?* (modelo del mundo, con incertidumbre y vigencia), (2) *¿qué quiero?* (objetivos, prioridades, restricciones), y (3) *¿qué puedo hacer?* (capacidades disponibles y condiciones). Lo que en un sistema reactivo

Deliberar exige convertir conocimiento, objetivos y capacidades en entidades de diseño consultables y auditables.

queda disuelto en el flujo de ejecución pasa a ser diseño explícito: interfaces, estados y contratos.

Por **contratos** nos referimos a acuerdos explícitos entre componentes/capas sobre qué se pide y qué se garantiza: qué significa éxito/fracaso, qué feedback o progreso se expone, y cómo se cancela o degrada una acción. Estos contratos evitan inferencias a partir de telemetría y facilitan supervisión y replanificación.

La deliberación no sustituye a la reactividad: la complementa. Por eso hablamos de arquitecturas **híbridas**, donde conviven bucles rápidos (seguridad, control, percepción) y decisiones de más alto nivel (planificación, asignación, estrategia). El reto no es "mezclar" módulos, sino conectar ambos mundos con reglas claras: qué se comparte, qué se impone y cuándo se revisa.

Una arquitectura híbrida sana define prioridades, flujos de información y autoridad para gestionar el desacuerdo plan–mundo.

Una guía práctica es pensar en **escalas temporales** y **responsabilidades** explícitas. Hay procesos en milisegundos (límites de seguridad, control), en segundos (ejecución con contexto) y en minutos (misión). Esto suele traducirse en roles: gestor de objetivos (misión), deliberador/planificador (decisión), ejecutor (acciones sostenidas) y monitor (progreso y seguridad).

Separar por ritmos y roles evita bloqueos: decisiones lentas no frenan reflejos y reflejos no destruyen la intención.

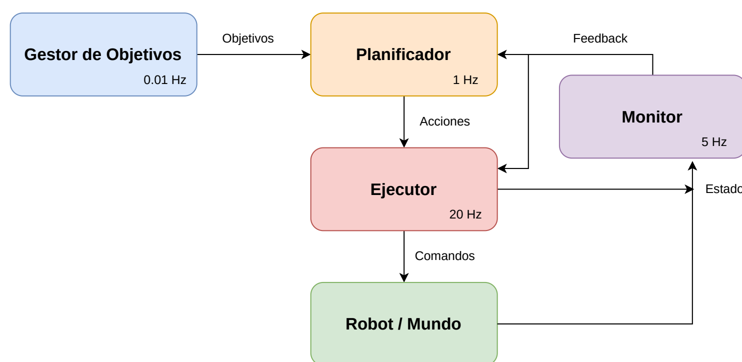


Figura 4.2: Roles y responsabilidades en un sistema híbrido: gestor de objetivos, deliberador/planificador, ejecutor y monitor.

El salto a lo deliberativo obliga también a tratar el **estado** como un activo: consultable, consistente y con semántica, algo que se concreta en la noción de **modelo del mundo** que desarrollaremos en la subsección correspondiente. No implica centralizarlo todo, sino decidir qué parte del mundo se representa y *cómo se consulta*, con qué resolución y con qué noción de incertidumbre y vigencia; la regla práctica es: modela explícitamente aquello que *cambia decisiones*.

El estado deliberativo debe modelar lo que cambia decisiones, incluyendo incertidumbre y vigencia, sin pretender modelarlo todo.

Para que tareas largas sean gobernables, la arquitectura debe definir **contratos de ejecución**: no basta con "ordenar" una acción, también hay que acordar cómo se observa su evolución y cómo termina. Un contrato típico incluye (i) condiciones de *éxito* y *fracaso* con significado operativo, (ii) señales de *progreso* para supervisar si avanza o está atascada, (iii) *cancelación* para ceder autoridad cuando cambia el objetivo o aparece un riesgo, y (iv) *timeout* para acotar cuánto tiempo se tolera la ausencia de avance. Por ejemplo, en una capacidad "navegar a una pose", el progreso puede ser "distancia restante" o "porcentaje de ruta completada"; el éxito es "llegó dentro de tolerancia"; y los fallos se clasifican como "objetivo inalcanzable por obstáculo persistente", "sin localización fiable" o "plan no converge". Con esta información, la capa deliberativa deja de adivinar a partir de telemetría y puede decidir con criterio: reintentar, cambiar de estrategia, pedir más información o replantear la misión.

Contratos con progreso y causas de fallo vuelven las capacidades controlables y replanificables desde arriba.

La robustez de un sistema híbrido depende de **supervisión**, **recuperación** y **arbitraje** entre capas. La supervisión detecta estancamientos, incoherencias y degradación; recuperación aplica reintentos acotados, alternativas o degradación segura; y el arbitraje fija prioridades (seguridad primero), cancelación y autoridad para evitar "peleas" entre planificador, ejecutor y reflejos.

Monitorizar, recuperar y arbitrar con reglas explícitas es más importante que "decidir bien" en condiciones ideales.

Finalmente, una deliberación operable necesita **replanificación por eventos**, **presupuestos de tiempo** y **trazabilidad**. En vez de replanificar "por si acaso", define disparadores (cambios significativos del mundo, fallos, desviación de progreso) y presupuestos (cuánto tiempo decidir, cada cuánto revisar salvo emergencia), con políticas de degradación cuando el tiempo se agota. Exponer objetivos, creencias y razones reduce el coste de depuración y ayuda a validar seguridad.

Disparadores, presupuestos y trazabilidad estabilizan la deliberación: menos oscilaciones y decisiones explicables.

Una forma de materializar todo lo anterior es seguir una receta de diseño en pasos, antes incluso de elegir algoritmos concretos:

Un checklist previo fija ritmos, contratos, estado y arbitraje antes de discutir algoritmos concretos.

- **Fija el horizonte y el ritmo** de cada tipo de decisión: qué se decide en milisegundos (reactivo), qué en segundos (ejecución) y qué en minutos (misión) (p. ej., "evitar colisión" vs. "navegar a una pose" vs. "entregar").
- **Define objetivos y restricciones** en términos que puedan verificarse: condiciones de éxito, límites de seguridad, recursos (tiempo, energía) y prioridades (p. ej., éxito: "objeto entregado"; restricción: "no entrar en zona X").
- **Enumera capacidades** como "servicios con contrato": entradas, feedback esperado, resultados, y catálogo de fallos con significado (p. ej., "navegar" con progreso; "manipular" con agarre/no agarre; "diálogo" con confirmación).
- **Decide qué estado debe ser explícito** (modelo del mundo) y qué puede permanecer como señal reactiva; especifica vigencia e incertidumbre (p. ej., "puerta abierta/cerrada" con timestamp y confianza).
- **Diseña el bucle de supervisión**: qué se monitoriza, qué umbrales disparan recuperación y cuándo se replanifica (p. ej., "progreso = 0 durante 10 s" → reintento o alternativa).
- **Establece reglas de arbitraje** entre capas: prioridades, cancelación, y comportamiento seguro por defecto (p. ej., una alarma de seguridad cancela "navegar" y fuerza parada).

El resultado buscado es un sistema simultáneamente **responsivo** (capaz de reaccionar) e **intencional** (capaz de sostener un propósito). En las siguientes subsecciones concretaremos dos piezas estructurales para conseguirlo: la organización por capas misión–tarea–capacidad y el diseño de modelos del mundo que permitan deliberar sobre conocimiento consistente.

La hibridación efectiva combina reflejos con deliberación, apoyándose en capas y en un modelo del mundo consistente.

Arquitecturas en capas: misión, tarea y capacidad

Una forma práctica de materializar lo discutido en la subsección anterior (escalas temporales, roles, contratos y supervisión) es separar el sistema en capas según el **nivel de abstracción** de sus decisiones. En este capítulo usaremos tres capas: **misión**, **tarea** y **capacidad**. No es un estándar universal, pero sí un vocabulario útil para no caer en monolitos.

Separar por capas convierte la hibridación en diseño: ritmo, responsabilidades y contratos quedan claros.

La idea central es que cada capa toma decisiones con un **horizonte** y un **ritmo** diferentes (Fig. 4.3). Misión opera en minutos u horas (inten-

Cada capa decide con distinto horizonte; por eso consume distinto estado y produce distintas garantías.

ción/política), tarea en segundos (ejecución con contexto) y capacidades en milisegundos o pocos segundos (acción segura). Si no se respeta esta separación aparecen bloqueos y "tiras y aflojas" entre intención y reflejos: oscilaciones, estado implícito y reglas de emergencia dispersas.

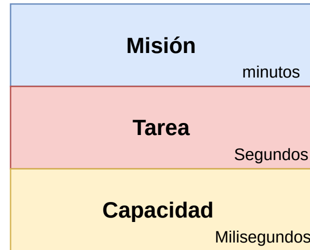


Figura 4.3: Arquitectura en capas: misión, tarea y capacidad. Cada capa opera con un horizonte y un ritmo diferentes, manteniendo un modelo del mundo consistente.

Una regla de diseño sencilla para dibujar fronteras es preguntar: *¿qué pregunta responde este componente?* "¿Qué queremos ahora y por qué?" (misión), "¿cómo lo consigo y qué hago si falla?" (tarea), "¿puedo ejecutar esta acción de forma segura y medible?" (capacidad). Esta regla evita interfaces difusas basadas en tamaño de módulos u organización del equipo.

Dibujar fronteras por preguntas (qué/ cómo/ con qué) reduce acoplamiento y ambigüedad.

Diseñar por capas obliga a decidir **qué se comparte** y **qué se encapsula**. En general, hacia abajo deberían bajar decisiones **declarativas** (objetivos, restricciones, prioridades) y hacia arriba deberían subir evidencias (progreso, resultado, causa de fallo). Invertir ese flujo (capacidad decide objetivos, misión decide control fino) reduce explicabilidad y aumenta fragilidad.

La tarea es el puente: la misión no debe bajar a sensores y las capacidades no deben conocer el objetivo global.

Misión. La misión define *qué* debe lograrse: objetivos globales, prioridades, restricciones y criterios de éxito (Fig. 4.4). Por ejemplo, en "entregar un objeto" decide *qué* entregar, *a quién* y *con qué política* (priorizar seguridad, minimizar tiempo o conservar batería), además de imponer restricciones como "no entrar en zona X". Su interfaz ideal expresa objetivo activo y contexto, criterios de éxito y límites (por ejemplo, zonas prohibidas o presupuesto de tiempo), además de una política de revisión (cuándo replantear, cuándo mantener) que trate explícitamente la incertidumbre y el riesgo.

La misión produce intención verificable: objetivos, restricciones y política de revisión, no comandos de bajo nivel.

Tarea. La tarea traduce intención en un **curso de acción ejecutable** y la sostiene en el tiempo: orquesta capacidades, mantiene contexto y gestiona contingencias. Por ejemplo, para "entregar un objeto" puede secuenciar "navegar al destino → solicitar confirmación al usuario → entregar → verificar", con reintentos acotados y alternativas cuando algo falla. Una guía útil es que declare estados de progreso y transiciones (preparar, ejecutar, verificar, recuperar, finalizar) y que distinga fallos recuperables de fallos fatales, reportando hacia misión causas y evidencias en un vocabulario coherente con la intención.

La tarea convierte intención en ejecución con estado de progreso y recuperación acotada.

En interfaces, la tarea consume **feedback** y **resultados** de capacidades y produce **eventos** y **resúmenes** para misión. Un anti-patrón es que misión "espíe" telemetría para inferir progreso; en su lugar, la tarea debe reportar progreso y causas de fallo en un vocabulario coherente con la intención.

La tarea consolida señales de bajo nivel en eventos de alto nivel para replanificar con criterio.

Capacidad. La capacidad ofrece servicios reutilizables (navegar, detectar, localizar, manipular) con garantías de seguridad y observabilidad. Por ejemplo, "navegar a una pose", "coger un objeto" o "realizar una

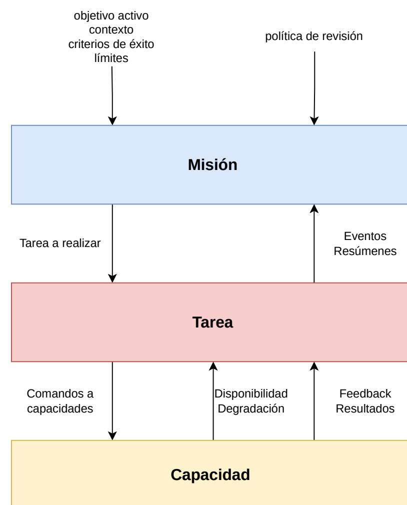


Figura 4.4: Arquitectura en capas con interfaces entre ellas.

pregunta y esperar respuesta" son capacidades distintas, cada una con sus parámetros y su propia noción de progreso. Además, debe declarar **disponibilidad** y **degradación** (por ejemplo, "baja confianza", "modo seguro lento", "sin calibración"), evitando que tarea y misión decidan a ciegas.

Una capacidad debe ser gobernable: precondiciones, parámetros, feedback, cancelación y causas de fallo con significado.

Contratos entre capas. La relación misión–tarea–capacidad debe estar definida por **contratos explícitos**, no por llamadas informales. Como mínimo, cada frontera acuerda entradas/salidas, semántica temporal (duración esperada, timeouts) y catálogo de fallos. Diseña contratos alrededor de **decisiones verificables** (éxito, progreso, razón) y usa una taxonomía mínima (no disponible, condición no cumplida, riesgo de seguridad, no converge, tiempo excedido) para habilitar recuperación.

Contratos explícitos separan intención, orquestación y acción, y permiten evolucionar cada capa sin colapsar el diseño.

Conflictos, prioridades y anti-oscilación. En sistemas híbridos el conflicto es normal: seguridad frena, ejecución se estanca, misión cambia prioridades. Una guía simple es: (1) seguridad en capacidades tiene precedencia, (2) la tarea tiene autoridad para pausar/cancelar y mantener coherencia de ejecución, (3) la misión cambia objetivos y políticas. Para evitar oscilaciones, introduce histéresis: umbrales de significancia, tiempos mínimos en estado y reglas de "cooldown" antes de replanificar.

Prioridades explícitas y cancelación, más histéresis, evitan "peleas" y oscilaciones entre capas.

Por ejemplo, imagina una misión "entregar" que ordena a la tarea "navegar a la puerta". La capacidad de navegación detecta personas muy cerca y activa un modo seguro que reduce velocidad o fuerza parada; la tarea observa que el progreso cae por debajo de un umbral durante unos segundos y cancela "navegar" para ejecutar una recuperación (esperar, retroceder, o rodear). Si tras uno o dos intentos el riesgo persiste, la tarea reporta un fallo con causa ("obstáculo persistente") y la misión cambia el objetivo (por ejemplo, "esperar" o "ir a otro acceso"). La histéresis evita que el sistema alterne sin fin entre "reintentar" y "replanificar": exige evidencia sostenida (tiempo mínimo, cooldown) antes de tomar decisiones drásticas.

Qué información consume cada capa. La coherencia depende de alinear cada capa con un **modelo del mundo** de resolución adecuada. Misión trabaja con zonas/recursos/riesgo, tarea con estado semántico y progreso (subobjetivos, disponibilidad), y capacidad con señales y estimaciones necesarias para actuar. Si una capa consume un mundo inadecuado

Cada capa necesita un "mundo" distinto: misión usa abstracciones, tarea usa estado semántico, capacidad usa señales locales.

(demasiado crudo o demasiado abstracto), aparecen decisiones incoherentes.

Una forma de aplicar estas ideas es usar un checklist breve cuando se diseña una nueva misión, tarea o capacidad.

Un checklist por capa fuerza a declarar propósito, ritmo, contratos, fallos y supervisión antes de implementar.

- **Para misión:** define objetivos, prioridades, restricciones y disparadores de revisión (cuándo replanificar y cuándo mantener).
- **Para tarea:** define estados de progreso, eventos que disparan recuperación, y qué resumen reporta a misión (progreso, razón de fallo, alternativa elegida).
- **Para capacidad:** define precondiciones, parámetros, feedback, resultado, cancelación y catálogo de fallos con significado operativo.
- **Para fronteras:** define timeouts, invariantes de seguridad, y quién tiene autoridad para pausar/cancelar.

Si estas piezas están bien definidas, el diseño gana **coherencia**, **reutilización** y **operabilidad**. A continuación introduciremos una aproximación clásica por comportamientos (subsumption) para entender su valor y sus límites, y después veremos cómo los *modelos del mundo* aportan el estado consistente que estas capas necesitan para deliberar sin depender de streams de bajo nivel.

Capas bien definidas sostienen intención y supervisión; falta completar el arbitraje por comportamientos y el estado.

Aproximaciones por comportamientos: subsumption y arbitraje reactivo

Antes de entrar de lleno en modelos del mundo, conviene mencionar una familia de arquitecturas que históricamente han capturado muy bien el espíritu de lo reactivo y lo híbrido: las aproximaciones por **comportamientos**, y en particular la arquitectura de *subsumption*. Aunque su foco no es deliberar con un estado global, sigue siendo relevante como patrón para diseñar reflejos, prioridades y arbitraje dentro de capas inferiores.

Subsumption organiza comportamientos en capas reactivas con arbitraje, útil para reflejos robustos dentro de un diseño híbrido.

La idea fundamental es construir el comportamiento como capas de módulos simples que se ejecutan en paralelo y compiten por el control, usando mecanismos de **supresión** o **inhibición** (Fig. 4.5). Por ejemplo, un comportamiento “evitar colisiones” puede suprimir temporalmente “seguir rumbo”, y un comportamiento “recuperar de atasco” puede suprimir “avanzar”. Esto produce sistemas muy responsivos, donde la seguridad y la estabilidad local pueden imponerse sin esperar a una decisión deliberativa.

El control se resuelve por supresión/inhibición: capas superiores pueden anular salidas de capas inferiores bajo condiciones.

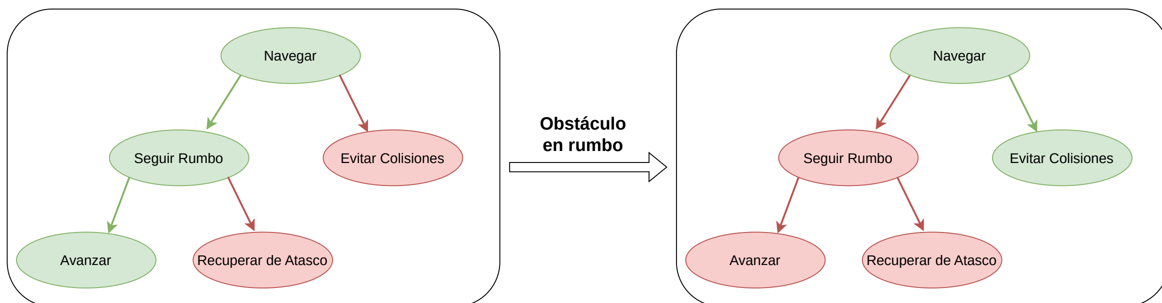


Figura 4.5: Los comportamientos emergen de la activación de comportamientos (en verde) y la supresión/inhibición de otros (en rojo).

Su ventaja es práctica: permite construir complejidad incremental y conservar robustez ante ruido y cambios rápidos del entorno. Además,

El valor de subsumption es incrementalidad y robustez: se añaden capas sin reescribir las anteriores.

fuerza a pensar en términos de prioridad y autoridad, que es precisamente una de las dificultades en arquitecturas híbridas.

Su límite, en el contexto deliberativo, es que el objetivo global y el estado suelen quedar implícitos o dispersos: cuesta justificar decisiones, garantizar coherencia a largo plazo y evitar oscilaciones sutiles entre comportamientos. Por eso es mejor entenderlo como una solución local (reflejos y arbitraje de bajo nivel) que como una arquitectura completa para misiones largas.

Sin estado y objetivos explícitos, subsumption escala peor en misiones largas: aumenta opacidad, oscilaciones y deuda de estado.

En un diseño misión–tarea–capacidad, la forma más sana de integrar estas ideas es encapsularlas *dentro* de capacidades o en un estrato reactivo de seguridad, manteniendo contratos claros hacia arriba. Así, el arbitraje reactivo puede decidir rápidamente cómo actuar ante un obstáculo, pero la tarea y la misión siguen gobernando la intención, la cancelación y la recuperación a escala larga. Esta separación evita que un patrón útil para reflejos se convierta en una fuente de opacidad para la deliberación.

Integra subsumption como mecanismo interno de capacidades: hacia arriba debe exponer progreso, límites y fallos con semántica.

Modelos del mundo

Para deliberar, el robot necesita algo más que streams de sensores: necesita un **modelo del mundo**. Este término engloba estructuras de datos y componentes que integran percepciones en una representación de mayor nivel: mapa, poses del robot, entidades del entorno, zonas navegables, objetos, personas, estados de puertas, etc. El objetivo no es "ver más", sino **poder decidir** sobre información consistente (Fig. 4.6).

El modelo del mundo integra percepciones en conocimiento de alto nivel para habilitar decisiones consistentes.

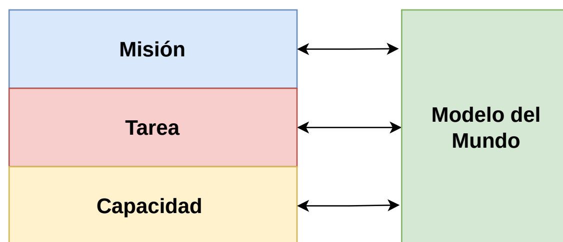


Figura 4.6: Arquitectura en capas interactiva con modelo del mundo.

Arquitectónicamente, un modelo del mundo actúa como frontera entre percepción y decisión. Recibe evidencia (mediciones, detecciones, estimaciones), la fusiona y produce conocimiento explotable por la capa de misión y tarea. Esta fusión suele incluir **asociación de datos** (qué observación corresponde a qué entidad), **estimación** (filtrado, seguimiento) y **gestión temporal** (caducidad, actualización, consistencia entre fuentes).

Como frontera percepción–decisión, el modelo del mundo fusiona evidencia con asociación, estimación y gestión temporal.

Un aspecto importante es el tratamiento de la **incertidumbre**. Las mediciones son ruidosas, las detecciones fallan y el entorno cambia; por ello el modelo del mundo suele mantener creencias: probabilidades, covarianzas, puntuaciones de confianza o etiquetas de validez. En sistemas deliberativos, estas señales de incertidumbre influyen en decisiones como "replanificar", "pedir más información" o "escoger una alternativa más segura".

Incertidumbre, confianza y vigencia deben modelarse explícitamente porque condicionan replanificación y seguridad.

El modelo del mundo también define **interfaces de consulta**. No solo publica datos; debe permitir responder preguntas relevantes para decisión: "¿dónde estoy?", "¿qué obstáculos hay cerca?", "¿cuál es el objeto objetivo y dónde estaba hace 2 segundos?". Diseñar esas consultas ayuda a que la capa deliberativa no se acople a detalles perceptivos (formatos, tasas, fuentes concretas).

Interfaces de consulta orientadas a decisiones reducen acoplamiento y evitan reconstruir el mundo desde streams.

En resumen, los modelos del mundo convierten una arquitectura basada en flujos en una arquitectura basada en **estado y conocimiento**. Esto habilita deliberación, permite supervisión y mejora la trazabilidad: se puede explicar por qué el sistema decidió lo que decidió, en qué creencia se basó y qué evidencia la sustentaba.

Pasar de flujos a estado y conocimiento habilita deliberación, supervisión y explicabilidad de decisiones.

4.2. Aplicación

Mapeo de capas a un grafo de ROS 2

En ROS 2, una arquitectura se hace tangible como un **grafo** de nodos y comunicaciones. Para reflejar la separación misión–tarea–capacidad conviene diseñar el grafo con una intención clara: nodos que representan *roles* (supervisor de misión, ejecutor de tareas, proveedores de capacidades) y conexiones que representan *contratos* (órdenes, feedback, estado, eventos).

En ROS 2, el grafo materializa la arquitectura: nodos como roles y conexiones como contratos explícitos.

Sin embargo, el grafo no solo dice *quién habla con quién*: también expresa *cómo se ejecuta* cada rol. En una capa de misión esperas decisiones lentas y orientadas a objetivos; en una capa de tarea esperas orquestación con estado de progreso y recuperación; y en capacidades esperas actividades gobernables, preemptables y observables. En términos de ROS 2, esto se traduce en nodos con (i) un **estado interno** bien definido, (ii) interfaces que permitan **progreso y cancelación**, y (iii) un modelo de activación y degradación que facilite operación.

En ROS 2, el diseño por capas también es un diseño de ejecución: estado, ritmos, preempción y ciclo de vida por nodo.

Una práctica habitual es que la capa de **capacidad** se materialice como nodos o composiciones de nodos que exponen interfaces estables y reutilizables. En ROS 2, esto se alinea muy bien con interacciones de larga duración: una capacidad no es "un mensaje", sino una actividad que progresa con el tiempo. A nivel de diseño, interesa que las capacidades publiquen estado y diagnósticos, y que puedan ser activadas/desactivadas sin reiniciar el sistema; por eso suelen beneficiarse de nodos *managed* (ciclo de vida) y de separar claramente estado continuo (topics) de ejecución (acciones). Un ejemplo típico de capacidad es **navegación** mediante Nav2: aunque se perciba como "una capacidad", en el grafo suele ser un *conjunto* de nodos (planificación, control, recuperación) en un namespace propio, con acciones como *NavigateToPose* y estado para supervisión.

Las capacidades se diseñan como subsistemas gobernables: acciones, estado, diagnósticos y ciclo de vida para activación y recuperación.

Del mismo modo, una capacidad de **manipulación** suele apoyarse en MoveIt 2: el nodo (o conjunto de nodos) responsable de planificar y ejecutar trayectorias actúa como proveedor de capacidad con su propio estado, y se consume desde arriba mediante objetivos que pueden cancelarse y que reportan resultado. Visto así, la capa de capacidad es donde "vive" gran parte de la complejidad técnica, pero encapsulada de manera que tarea y misión no dependan de detalles internos (sensores concretos, controladores, parámetros finos), sino de contratos.

Capacidades como Nav2 o MoveIt 2 no son "llamadas": son servidores con estado, precondiciones y degradación, integrables en tareas.

La capa de **tarea** suele adoptar el papel de "orquestador": integra varias capacidades, decide su orden, maneja errores y mantiene el contexto de ejecución. En ROS 2, este nodo puede consumir estado publicado por capacidades (topics), solicitar operaciones (servicios o acciones) y escuchar eventos del sistema (timers, cambios de modo, alarmas). Un patrón muy extendido para materializar tareas es usar **Behavior Trees** (BT): la tarea se define como un árbol que "tiquea" (evalúa) condiciones y acciones, consulta estado y decide cuándo reintentar, retroceder o pedir ayuda a misión. Esto encaja bien con requisitos operativos: progreso

La tarea es un ejecutor con memoria: orquesta, mide progreso, aplica recuperación y preempta sin bloquear al resto.

observable, cancelación, *timeouts* por rama, y recuperación local antes de escalar el problema.

Por ejemplo, una tarea "entregar un objeto" puede implementarse como un BT que secuencia "ir al destino" (capacidad Nav2), "alinearse/alcanzar" (capacidad MoveIt 2 o control local), y "confirmar" (capacidad HRI/diálogo), con nodos de condición que consultan si la localización es fiable o si el usuario respondió. En el grafo, esto se ve como un nodo de tarea que actúa como *cliente* de varias acciones y como productor de eventos/resúmenes hacia misión.

Behavior Trees permiten expresar tareas como políticas de ejecución con recuperación: secuencias, condiciones y fallback sobre capacidades.

La capa de **misión** se modela como un supervisor que gestiona objetivos, prioridades y políticas. A este nivel interesan mecanismos para cambiar el objetivo activo, pausar, cancelar o replanificar; también interesa registrar decisiones y razones. En ROS 2 es común materializar misión de dos formas, según la complejidad: (i) una **FSM** (máquina de estados, Fig. 4.7) cuando el conjunto de situaciones es acotado y el comportamiento puede enumerarse con claridad, o (ii) un **planificador** cuando conviene razonar sobre objetivos y recursos de forma más flexible. Un ejemplo del segundo caso (Fig. 4.8) es PlanSys2: misión expresa objetivos (por ejemplo, "entregar(X,personaY)") y obtiene un plan; luego delega en tareas que ejecutan pasos y reportan resultados.

La misión gestiona objetivos y políticas, no secuencias finas: planifica o gobierna una FSM y replanifica por eventos.

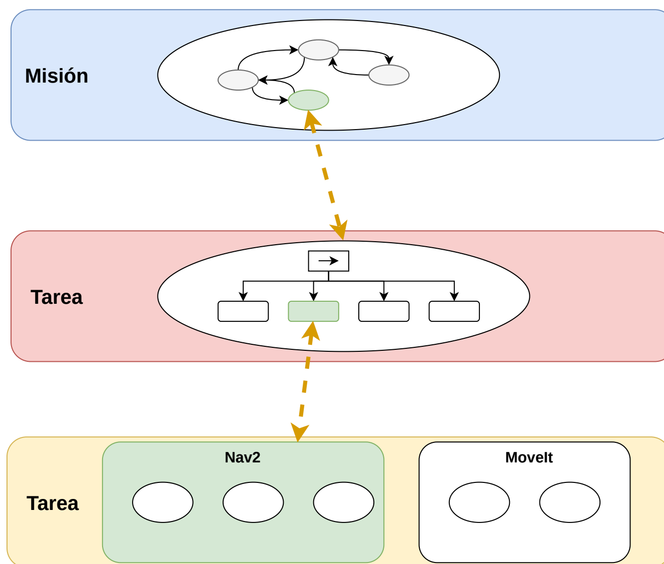


Figura 4.7: Arquitectura en capas implementada en ROS con máquinas de estados para la capa de misión, Behavior Trees para tareas y *frameworks* para capacidades.

En ambos enfoques, la misión suele ser un nodo con *bajo ritmo* (decisiones cada segundos/minutos), fuertemente orientado a **preempción** (cambiar objetivo, pausar o abortar) y a **toma de decisión basada en evidencia** (resultados de tareas, diagnósticos, estado del mundo). En el grafo, su salida principal no es teleoperar capacidades, sino disparar y cancelar tareas, mantener la política activa (prioridades, riesgo tolerable) y coordinar replanificación.

Misión debe ser "lenta pero soberana": cambia objetivos y políticas, cancela tareas y registra razones con trazabilidad.

Para mantener el desacoplamiento, ROS 2 ofrece herramientas de organización: **namespaces** para agrupar subsistemas, **parámetros** para configurar roles sin recompilar, y **composición** para integrar nodos cuando se necesite eficiencia sin perder modularidad. El diseño por capas se refleja así no solo en el diagrama conceptual, sino también en la topología y convenciones del sistema ROS 2.

Namespaces, parámetros y composición permiten modularidad y eficiencia sin romper la separación por capas.

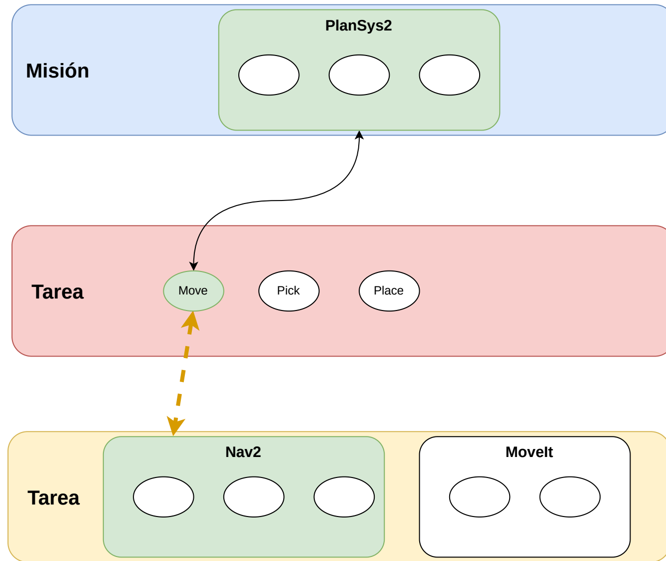


Figura 4.8: Arquitectura en capas implementada en ROS con PlanSys2 para la capa de misión, Behavior Trees para tareas y *frameworks* para capacidades.

Un detalle práctico al pasar de la arquitectura al despliegue es asignar **concurrencia y aislamiento** de forma coherente con las capas. Aunque ROS 2 permite múltiples configuraciones de *executors* y grupos de callbacks, la intención arquitectónica es simple: capacidades no deberían quedar bloqueadas por la lógica deliberativa; tareas no deberían depender de temporizaciones implícitas; y misión debería poder cancelar y replanificar sin quedar atrapada en esperas. Estos aspectos se afinan en el diseño detallado y en la fase de integración, pero conviene anticiparlos al diseñar el grafo.

Alinear concurrencia con capas evita interferencias: capacidades con callbacks críticos, tareas con orquestación, misión con control de alto nivel.

Finalmente, conviene que cada frontera del grafo use el tipo de interacción que mejor expresa su contrato. En términos generales, las **acciones** encajan con actividades largas y cancelables (tareas y capacidades), los **topics** con estado y señales continuas (telemetría, estimaciones, alertas) y los **servicios** con consultas u operaciones puntuales (configurar, preguntar, reiniciar). Los **parámetros** permiten fijar políticas (prioridades, umbrales, presupuestos de tiempo) sin reescribir la lógica, y se complementan con ciclo de vida y QoS cuando se necesita control explícito de disponibilidad, fiabilidad y latencias.

Acciones, servicios, topics y parámetros son una decisión arquitectónica: clarifican qué es comando, qué es estado y qué es consulta.

Lifecycle Nodes y subsumption

En ROS 2 existe un tipo de nodo gestionado llamado **Lifecycle Node** (nodo con ciclo de vida). La idea es separar claramente *crear el nodo* de *ponerlo a trabajar*: un nodo puede arrancar, cargar parámetros, reservar recursos y anunciar interfaces, pero permanecer *inactivo* hasta que el sistema decida activarlo. Esto es especialmente útil en robótica porque los subsistemas suelen tener dependencias (TF, mapas, controladores, sensores) y porque conviene poder *recuperar* de fallos sin reiniciar todo.

Lifecycle Nodes separan creación y activación: arrancan "preparados" y se activan con control, facilitando recuperación sin reinicios globales.

Conceptualmente, un Lifecycle Node implementa una **máquina de estados** bien definida (Fig. 4.9). Los estados típicos incluyen *unconfigured*, *inactive*, *active* y *finalized*, y se recorren mediante **transiciones** (por ejemplo, *configure*, *activate*, *deactivate*, *cleanup* o *shutdown*). Cada transición dispara **callbacks** donde se ejecuta la lógica de inicialización o liberación de recursos: en *configure* se valida configuración y se crean conexiones; en

El ciclo de vida formaliza estados y transiciones, habilitando activación/desactivación segura y arranques/paradas orquestados y auditables.

activate se empieza a publicar o a procesar; en *deactivate* se deja de actuar manteniendo el nodo disponible; y en *cleanup* se vuelve a un estado previo liberando lo que sea necesario. El resultado es que el despliegue puede orquestar arranques ordenados, activaciones controladas, y paradas seguras.

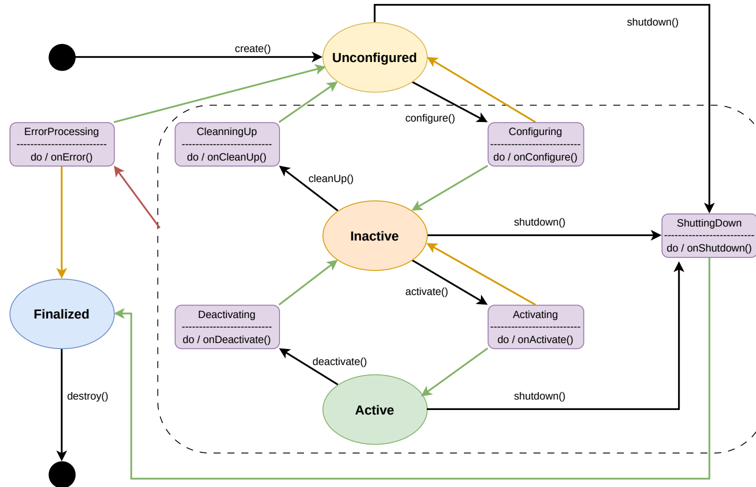


Figura 4.9: LifeCycle Node: estados y transiciones.

Desde el punto de vista arquitectónico, el ciclo de vida aporta **operabilidad**: (i) facilita comprobar si un subsistema está listo antes de depender de él, (ii) permite degradar capacidades desactivándolas, y (iii) ofrece un mecanismo claro de recuperación (desactivar, limpiar y volver a configurar). Esto encaja con lo discutido en este capítulo: contratos explícitos no solo para *ejecutar* acciones, sino también para *gestionar* disponibilidad y modos del sistema.

El ciclo de vida mejora operabilidad: readiness, degradación por desactivación y recuperación controlada, integradas como parte del contrato del sistema.

Subsumption con Cascade Lifecycle Node. Una arquitectura de *subsumption* organiza comportamientos reactivos en capas con prioridad, donde unas capas pueden *suprimir* o *inhibir* a otras. En ROS 2, una forma práctica de materializar ese arbitraje es hacer que cada çomportamiento"(o cada capa) sea un conjunto de nodos gobernables por ciclo de vida, y que la activación de una capa implique la activación (o desactivación) de los nodos que le dan soporte.

Subsumption puede materializarse como capas gobernables: activar una capa implica activar su soporte y suprimir capas inferiores de forma explícita y operable.

El paquete **cascade_lifecycle** (`https://github.com/fmrico/cascade_lifecycle`) propone un patrón útil: un *Cascade Lifecycle Node* declara qué otros Lifecycle Nodes deben estar activos si él está activo, produciendo activaciones y supresiones en cascada. A nivel de diseño, esto permite expresar dependencias y prioridades sin dispersar lógica de arranque/parada por todo el grafo: si una capa de alta prioridad entra en *active*, puede forzar que su "subárbol"de nodos relevantes se active; y, simétricamente, al desactivarse puede liberar (o devolver a *inactive*) los nodos que ya no deben influir.

Cascade Lifecycle Node captura dependencias y prioridades: activaciones y supresiones en cascada evitan lógica de arranque/parada dispersa por el grafo.

Por ejemplo, imagina tres capas: (i) *parada de seguridad*, (ii) *evitación reactiva* y (iii) *seguimiento de ruta*. Puedes modelar cada capa como un Cascade Lifecycle Node que .ªrrastra"los nodos que necesita (sensado cercano, filtro, control local) y, cuando se activa, **suprime** la capa inferior desactivando sus nodos de control. El arbitraje queda entonces como una secuencia de transiciones de ciclo de vida (activar/desactivar) con prioridad explícita. Además, la histéresis puede implementarse a nivel de activación: tiempos mínimos en *active* o *cooldown* antes de permitir volver a una capa inferior, evitando oscilaciones.

El arbitraje se vuelve una política de activación/desactivación por prioridad, con histéresis (tiempos mínimos y cooldown) para evitar oscilaciones entre capas.

Este enfoque no sustituye a una capa deliberativa, pero aporta una forma robusta y operable de materializar reactividad y arbitraje: hace visibles los modos del sistema, hace gobernable qué subsistemas están actuando, y encaja con contratos de cancelación y recuperación discutidos anteriormente.

Como complemento a la deliberación, hace visibles y gobernables los modos del sistema y refuerza contratos de cancelación y recuperación.

Construcción del modelo del mundo en ROS 2

Un modelo del mundo en ROS 2 suele organizarse como un subsistema que **ingiere** evidencia de múltiples fuentes y **publica** una representación integrada. Es habitual separar nodos que realizan estimación (por ejemplo, localización, seguimiento) de nodos que gestionan conocimiento (listas de entidades, mapas semánticos, estados discretos), evitando crear un "nodo monolítico" difícil de validar.

El modelo del mundo se estructura como subsistema de ingestión y publicación, evitando nodos monolíticos.

Las fuentes de evidencia llegan típicamente como topics de sensores, detecciones o estimaciones parciales. La fusión se apoya en convenciones temporales y espaciales: marcas de tiempo coherentes, transformaciones de referencia y consistencia entre marcos. Diseñar bien estas entradas significa especificar qué campos son obligatorios, qué frecuencia mínima se espera y qué ocurre cuando una fuente se degrada o desaparece.

La fusión requiere convenciones temporales y espaciales, y un plan explícito ante degradación o pérdida de fuentes.

El modelo del mundo (Fig. 4.10) debería exponer dos caras: **estado publicado** (para visualización, monitorización y consumo reactivo) y **consultas** (para consumo deliberativo). En ROS 2, las consultas pueden mapearse a servicios o acciones dependiendo de si son instantáneas o costosas. Esta separación evita que la capa de misión tenga que "reconstruir" el mundo a partir de streams y le permite pedir exactamente la información que necesita.

Separar estado publicado y consultas permite consumo reactivo y deliberativo sin reconstrucciones ad hoc.

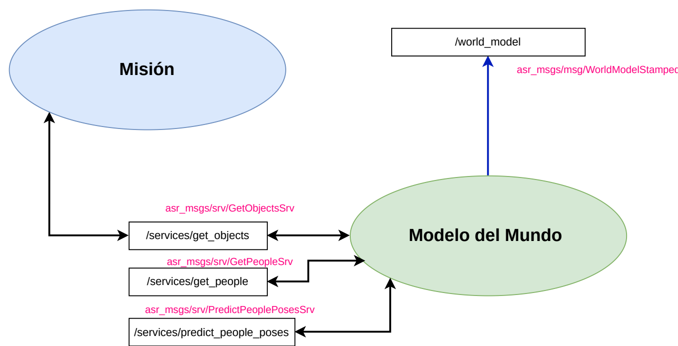


Figura 4.10: Nodo de ROS 2 que implementa el modelo del mundo, mostrando las interfaces de estado publicado y consultas.

Otra decisión de diseño es cómo representar incertidumbre y validez. A nivel de arquitectura, interesa que el modelo del mundo publique no solo "qué cree", sino también **cómo de seguro está** y **cuán reciente** es la información. Esto habilita políticas deliberativas como replanificación bajo alta incertidumbre o exploración dirigida para reducir ambigüedad.

Publicar incertidumbre, validez y frescura habilita políticas deliberativas robustas ante ambigüedad.

Finalmente, el modelo del mundo sirve como punto de integración con herramientas de operación: visualización (por ejemplo, RViz), logging, y diagnóstico. Si el sistema falla, una buena arquitectura permite inspeccionar el estado del mundo, entender qué evidencia llegó y en qué etapa se perdió consistencia. Esa trazabilidad es un requisito práctico para que la deliberación sea operable en un robot real.

La integración con visualización y diagnóstico hace el modelo del mundo inspeccionable y la deliberación operable.

En la práctica no hay un único "modelo del mundo" correcto: hay **diseños** con propiedades distintas, y es útil conocer alternativas antes de elegir. A continuación se describen tres diseños concretos (sin bajar

Un modelo del mundo no es una pieza única: existen diseños con diferentes garantías de consistencia, latencia y complejidad operativa.

a implementación) que cubren un espectro amplio: un servidor central, una federación basada en vistas, y un modelo dual que separa geometría y símbolos.

Diseño 1: servidor central de modelo del mundo (*World Model Server*). En este diseño (Fig. 4.10) existe un nodo (o composición) responsable de mantener un **estado canónico** de alto nivel y de servir consultas. Los nodos "productores de evidencia" publican observaciones (detecciones, estimaciones, estados discretos) hacia el servidor; el servidor realiza asociación/fusión y actualiza entidades (por ejemplo, personas, objetos, zonas, puertas), además de mantener metadatos como *timestamp*, fuente, confianza y caducidad. El resultado se expone en dos formas: (i) topics para consumo reactivo y visualización (por ejemplo, "lista de entidades" o "estado semántico del entorno"), y (ii) interfaces de consulta para decisión (por ejemplo, "dame el objeto más probable que corresponde a X", "qué puertas están abiertas y recientes", "qué zonas están bloqueadas y con qué evidencia").

Servidor central: una fuente de verdad consultable, con fusión y semántica en un único punto, a costa de ser un cuello de botella.

Un patrón útil en ROS 2 es que el servidor central tenga **interfaces separadas por coste**: consultas ligeras como servicios (responden rápido y son deterministas) y consultas costosas como acciones (permiten progreso y cancelación). Por ejemplo, "consulta de entidad por id" o "estado de una puerta" sería servicio; "resolver una referencia ambigua" ("¿qué objeto es el que el usuario señaló?") o "reconstruir un mapa semántico local" podría ser acción. Además, el servidor suele beneficiarse de ciclo de vida: inicializa fuentes, valida configuración, pasa a activo y puede reiniciarse controladamente.

Interfaces por coste: servicios para consultas rápidas, acciones para razonamientos caros; así la decisión puede supervisar y preemptar.

Ventajas. Es fácil de entender, depurar y documentar: hay un lugar donde buscar "qué cree el robot". Reduce duplicación de lógica (fusión, caducidad, asociación), ayuda a imponer vocabularios coherentes (taxonomía de entidades/estados), y facilita que misión/tarea no se acoplen a streams. *Desventajas.* Puede convertirse en cuello de botella (CPU/memoria/latencia) y en punto único de fallo; además, si su modelo crece sin control, tiende a absorber responsabilidades que deberían estar en estimadores especializados.

Ventaja clave: trazabilidad y coherencia; es fácil auditar decisiones porque existe una fuente de verdad explícita.

Diseño 2: federación de estimadores + nodo de "vista" (*World Model Views*). En este diseño (Fig. 4.11) no existe una sola base de conocimiento central: cada subsistema mantiene su propia "verdad" de dominio (localización, mapas, seguimiento de objetos, seguimiento de personas, estados de interacción), y publica resultados como topics con semántica clara. Encima se construyen uno o varios nodos de **vista** que suscriben, sincronizan en el tiempo, transforman a marcos comunes (TF) y *cachean* estado reciente para responder consultas. Por ejemplo, una vista para navegación puede combinar mapa + coste local + obstáculos; una vista para manipulación combina poses de objetos + accesibilidad + estado del gripper; una vista para HRI combina identidad del interlocutor + intención detectada + confirmaciones.

Federado: cada subsistema publica su mejor estimación; una "vista" compone y cachea para consulta sin imponer un estado global rígido.

Las vistas son, en esencia, **adaptadores deliberativos**: convierten streams heterogéneos en respuestas a preguntas del planificador o del ejecutor de tareas, sin que estos consuman directamente decenas de topics. Una vista típica ofrece (i) estado publicado para monitorización (qué está cacheando, qué fuentes están vivas, qué incertidumbre tiene) y (ii) consultas (servicios o acciones) que devuelven una respuesta estable aunque las fuentes internas sean asíncronas. Este enfoque suele convivir bien con sistemas grandes: las vistas pueden ser específicas por tarea/capacidad y evolucionar sin forzar un esquema global.

Las vistas reducen acoplamiento: la decisión pregunta "lo que necesita", y la vista se encarga de sincronizar, transformar y filtrar.

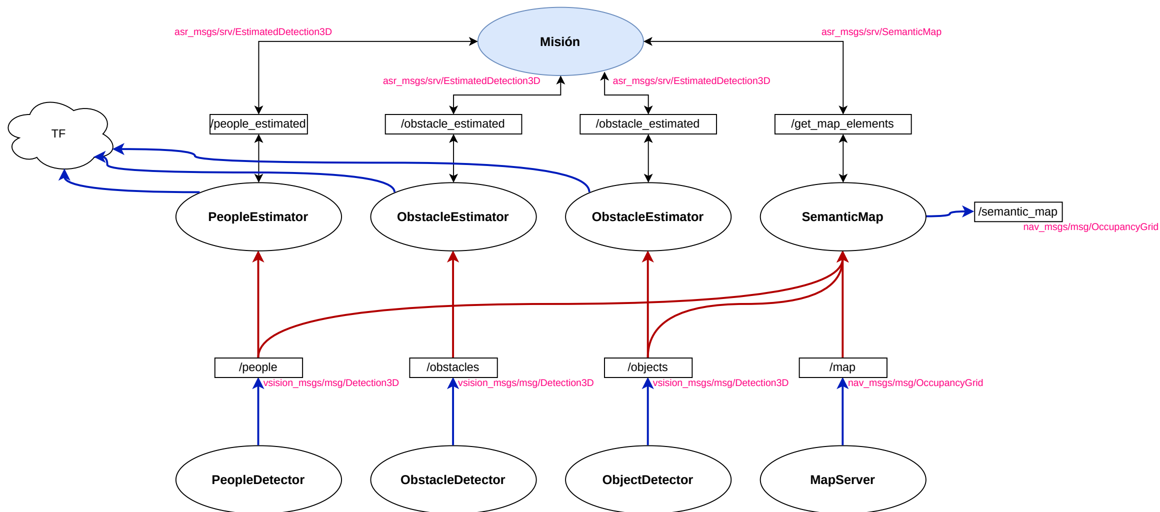


Figura 4.11: Modelo del mundo basado en vistas federadas.

Ventajas. Escala mejor por equipos y dominios: cada subsistema evoluciona con su ritmo, y las vistas encapsulan integración. Es robusto a degradación parcial: si cae el seguidor de personas, la vista de navegación puede seguir funcionando. *Desventajas.* La consistencia es más débil: puede haber discrepancias temporales entre fuentes y vistas, y la trazabilidad requiere buenas herramientas de logging/diagnóstico para reconstruir qué vista vio qué dato en qué momento.

Ventaja clave: escalabilidad y modularidad; evita el "mega-modelo" y reparte carga computacional y responsabilidad.

Diseño 3: modelo dual (métrico/geométrico + simbólico) con traductor. Este diseño (Fig. 4.12) separa explícitamente dos niveles de "mundo": (i) un mundo **geométrico** (poses, mapas, obstáculos, trayectorias, accesibilidad), y (ii) un mundo **simbólico** (predicados, estados discretos, recursos, permisos) consumible por misión. El mundo geométrico suele estar dominado por componentes ya estándar (TF, localización, mapas, Nav2, MoveIt 2, tracking), mientras que el simbólico se mantiene como una base de hechos con vigencia y confianza, por ejemplo: "en(herramienta,mesa)", "abierta(puerta1)", "interlocutor(personaY)" u "objeto(objetoX)".

Modelo dual: geometría para ejecutar con seguridad y símbolos para planificar; un traductor mantiene consistencia entre ambos.

La pieza clave es un **traductor** (o varios) que observa el mundo geométrico y actualiza el simbólico con reglas explícitas. Por ejemplo, si el tracker estima que una puerta está abierta con confianza alta durante un intervalo, el traductor afirma "abierta(puerta1)"; si la localización pierde calidad, afirma "localizacion(no-fiable)"; y si un objeto se detecta en una zona, afirma "en(objetoX,zonaA)". Este diseño encaja especialmente bien cuando la misión se implementa con un planificador (por ejemplo, PlanSys2 u otro), porque permite razonar sobre objetivos y precondiciones en un lenguaje estable, sin acoplarse a métricas concretas.

El traductor convierte evidencia continua en hechos discretos con umbrales y vigencia, haciendo el puente entre percepción y planificación.

Ventajas. Clarifica responsabilidades y reduce acoplamiento semántico: misión consume símbolos, tarea consume vistas geométricas, y ambos se sincronizan mediante reglas. Es muy explicable: se puede trazar qué hechos estaban presentes cuando se tomó una decisión. *Desventajas.* Exige diseñar bien los umbrales, la caducidad y los conflictos de evidencia: discretizar un mundo continuo puede introducir cambios espurios u oscilaciones si no se incorpora histéresis y gestión temporal.

Ventaja clave: separación limpia por niveles; facilita planificación, validación de precondiciones y explicaciones ("qué hecho habilitó qué acción").

Una recomendación transversal a los tres diseños es tratar el modelo del mundo como un componente **operable**: debe publicar diagnóstico de fuentes, métricas de frescura, y un resumen de incertidumbre; y debe

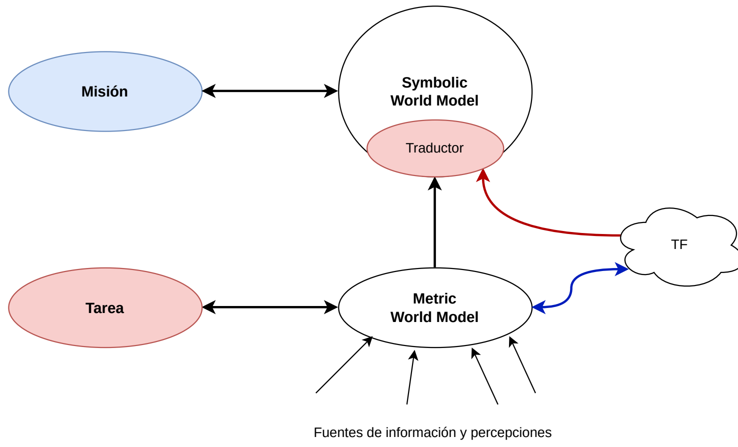


Figura 4.12: Modelo del mundo basado en modelo dual (geométrico + simbólico) con traductor.

poder degradar con seguridad (por ejemplo, “sin localización fiable” → restringir tareas). En una arquitectura deliberativa, estas señales no son “decoración”: son parte del contrato que permite a misión y tarea decidir cuándo replanificar, cuándo pedir más información y cuándo abortar.

Sea cual sea el diseño, el modelo del mundo debe ser operable: diagnósticos, frescura, incertidumbre y degradación explícita.

5 Máquinas de estados finitos

5.1. Principios teóricos

5.1 Principios teóricos 65
La máquina de estados como arquitectura de decisión . . 65
Anatomía de un estado en robótica 66
Semántica de una transición 67
Topologías 68
El problema de la concurrencia 70
Modelado de misiones y tareas 72
Ventajas y problemas de escalabilidad 72
Máquinas de estados jerárquicas 73
5.2 Aplicación 73
FSM a nivel de tarea 74
FSM a nivel de misión . . . 75
Diseño compositivo: FSM de misión y tarea 76
5.3 Conclusiones 76

La máquina de estados como arquitectura de decisión

Las máquinas de estados finitos (FSM) constituyen uno de los mecanismos más utilizados para modelar el comportamiento de robots. Desde un punto de vista arquitectónico, una FSM define explícitamente los estados posibles del sistema y las transiciones entre ellos, proporcionando una estructura clara para la toma de decisiones. A diferencia de enfoques puramente reactivos, las FSM permiten representar secuencias de comportamiento y modos de operación persistentes. Cada estado encapsula una lógica específica y las transiciones definen cuándo y cómo el sistema cambia de comportamiento. Esta explicitación del comportamiento resulta especialmente útil en robótica, donde es necesario coordinar acciones a lo largo del tiempo y reaccionar de forma controlada ante eventos del entorno.

Para ilustrar este concepto, consideremos un ejemplo simple pero representativo: un robot de patrullaje que debe navegar entre puntos de interés mientras monitoriza su batería. La figura 5.1 muestra la estructura de esta FSM. El robot comienza en el estado `IDLE` (reposo), a la espera de una orden de inicio. Una vez activado, transita a `PATROLLING`, donde se desplaza autónomamente entre waypoints. Si durante el patrullaje la batería desciende por debajo de un umbral crítico (guarda: `battery <20 %`), el sistema transita automáticamente a `CHARGING`, dirigiéndose a la estación de carga. Una vez recargado (guarda: `battery >90 %`), el robot puede retomar el patrullaje o volver a reposo según la orden del usuario.

Este diagrama ilustra los elementos fundamentales de una FSM: estados bien definidos (círculos), transiciones explícitas (flechas), eventos que las disparan (etiquetas en las flechas) y guardas que condicionan la transición (expresiones booleanas entre corchetes). La arquitectura es declarativa: basta observar el diagrama para comprender el comportamiento completo del sistema, sin necesidad de analizar líneas de código.

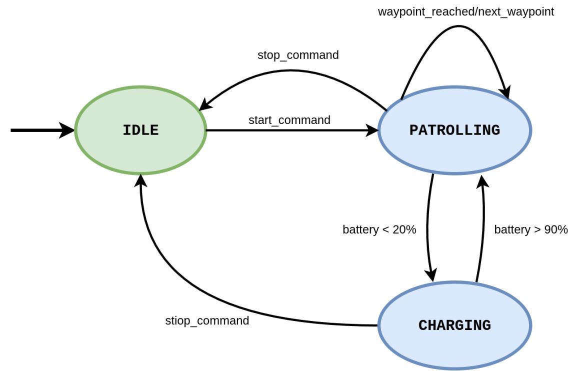


Figura 5.1: Ejemplo de FSM para un robot de patrullaje con gestión de batería. Los estados representan modos de operación, las transiciones indican eventos o condiciones, y las guardas (expresiones booleanas) determinan cuándo ocurre cada cambio.

Anatomía de un estado en robótica

En sistemas puramente computacionales, un estado puede ser simplemente una etiqueta. Sin embargo, en robótica, un estado suele controlar recursos físicos (motores, sensores). Para gestionar esto de forma segura, es fundamental estructurar cada estado con tres fases diferenciadas:

Un aspecto clave en el diseño de FSMs en robótica es la distinción entre el estado interno del sistema y el estado observable desde el exterior. El estado interno incluye variables o temporizadores que solo la máquina de estados conoce y utiliza para la toma de decisiones. El estado observable, en cambio, es la manifestación externa del sistema: lo que un usuario, otro sistema o un sensor puede percibir (por ejemplo, la posición de un actuador, el encendido de una luz o la emisión de un sonido).

Esta distinción es fundamental para el diagnóstico y la interacción: un robot puede estar en un estado interno de .espera.aunque externamente parezca inactivo, o puede mostrar un estado observable de .en movimiento.aunque internamente esté gestionando una transición. Diseñar FSMs que hagan explícita esta diferencia mejora la trazabilidad y la depuración de comportamientos complejos.

Distinguir entre estado interno y observable facilita el diagnóstico y la trazabilidad.

En robótica, un estado controla recursos físicos y requiere fases bien definidas.

- **on_entry (entrada):** Acciones que se ejecutan una única vez al transitar hacia este estado. Es el lugar ideal para inicializar temporizadores, reiniciar contadores, encender sensores o configurar parámetros de control.
- **on_do / on_run (ejecución):** Lógica que se ejecuta de forma cíclica mientras el sistema permanece en el estado. Aquí reside la «inteligencia» continua: calcular señales de control, monitorizar condiciones de seguridad y verificar si se cumplen las condiciones para transitar a otro estado.
- **on_exit (salida):** Acciones de limpieza que se ejecutan una única vez justo antes de abandonar el estado. Son críticas para la seguridad (detener motores, apagar un láser o liberar recursos).

Cada estado se estructura en entrada, ejecución cíclica y salida.

Esta estructura garantiza que, independientemente de por qué se abandone un estado (éxito, error o interrupción), el robot siempre deja sus recursos en un estado conocido y seguro. La figura 5.2 ilustra el ciclo de vida completo de un estado y cómo se orquesta la ejecución de estas tres fases.

La estructura de fases asegura la limpieza y seguridad de los recursos.

Analicemos con detalle el diagrama de la figura 5.2. Al entrar en un estado, se ejecuta automáticamente la fase **on_entry** (verde), donde se inicializan recursos: temporizadores, contadores, sensores o parámetros de control. Esta fase se ejecuta una única vez, garantizando que el estado comience desde una configuración conocida.

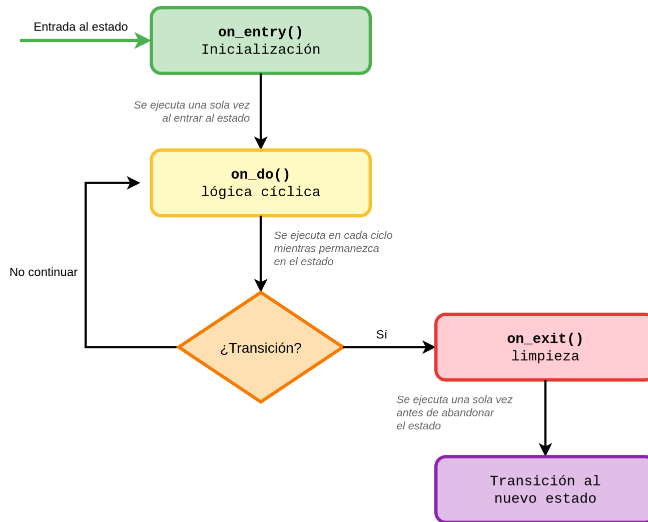


Figura 5.2: Ciclo de vida de un estado en una FSM robótica. La fase `on_entry` inicializa recursos, `on_do` se ejecuta cíclicamente monitorizando condiciones de transición, y `on_exit` garantiza la limpieza segura antes de abandonar el estado.

A continuación, el sistema entra en un bucle cíclico ejecutando repetidamente la fase **on_do** (amarillo). En cada iteración del ciclo de control, esta fase evalúa las condiciones del entorno, calcula señales de control y determina si se cumplen las condiciones para transitar a otro estado. La flecha que regresa desde la decisión «¿Transición?» hacia `on_do` representa este comportamiento cíclico: mientras no se detecte una condición de transición, el estado permanece activo ejecutando su lógica en cada ciclo.

Cuando `on_do` detecta que debe cambiar de estado (por ejemplo, se alcanza un objetivo, se detecta un error o expira un temporizador), el sistema invoca la fase **on_exit** (rojo). Esta fase es crítica para la seguridad: detiene motores, apaga sensores, libera recursos y garantiza que el robot quede en un estado seguro antes de proceder a la transición. Solo después de completar la limpieza, el sistema transita formalmente al nuevo estado.

Esta secuencia estricta (entrada → ejecución cíclica → salida → transición) es fundamental para evitar condiciones de carrera y estados inconsistentes en los actuadores del robot. Independientemente de por qué se abandone un estado (éxito, error o interrupción externa), el sistema siempre ejecuta `on_exit`, garantizando que los recursos físicos queden en un estado conocido y seguro.

La fase `on_exit` garantiza la seguridad y consistencia al cambiar de estado.

Semántica de una transición

Para que el diseño de una FSM sea robusto, es necesario formalizar qué provoca un cambio de estado. Arquitectónicamente, una transición se define mediante la tupla:

$$\text{Transición} = \text{Evento} + [\text{Guarda}]/\text{Acción}$$

Una transición se define por evento, guarda y acción.

- **Evento:** Es el suceso (externo o interno) que «despierta» la posibilidad de cambio. En ROS 2, suele ser la llegada de un mensaje, un timeout o un resultado de una Acción.
- **Guarda:** Es una expresión lógica que debe ser verdadera para que la transición ocurra dado el evento. Por ejemplo, recibir un mensaje de láser (evento) no provoca un cambio a menos que la distancia medida sea menor a un umbral (guarda).

Distinguir entre evento y guarda es crucial para la depuración: un robot puede no cambiar de estado porque el evento no llega (fallo de comunicación) o porque la guarda no se cumple (fallo de lógica).

Distinguir evento y guarda facilita la depuración de FSM.

Topologías

Desde la teoría de autómatas, la diferencia entre Moore y Mealy parece sutil, pero en la implementación software de un robot tiene un impacto directo en la **latencia** (tiempo de reacción). La clave está en determinar *en qué ciclo de control* se envía el comando a los motores.

La diferencia Moore/Mealy afecta la latencia de reacción.

- **Máquinas de Moore (salida en el estado):** Las acciones de control se ejecutan dentro del código del estado.
- **Máquinas de Mealy (salida en la transición):** Las acciones de control se ejecutan en el momento exacto de detectar el evento, antes incluso de cambiar de estado.

Veamos la diferencia con un ejemplo de frenada de emergencia ante un obstáculo, asumiendo un ciclo de control de 100ms. La figura 5.3 ilustra mediante un diagrama temporal cómo difieren ambos enfoques en términos de latencia de respuesta:

Enfoque Moore (1 ciclo de retraso) La lógica es: «*Si detecto obstáculo, cambio el estado a* `PARADO`*. En el estado* `PARADO`*, envío velocidad cero.*»

1. **Ciclo T:** El robot está en `AVANZANDO`. Detecta el obstáculo. La FSM decide cambiar `next_state = PARADO`. (Nota: En este ciclo, todavía no se ha ejecutado el código de `PARADO`).
2. **Ciclo $T + 1$ (100ms después):** La FSM entra en `PARADO`. Ahora sí, se ejecuta `publish_velocity(0)`.

Consecuencia: Hay una latencia de un ciclo completo. Es más seguro (código ordenado), pero más lento.

Moore: acción más segura pero con mayor latencia.

Enfoque Mealy (Reacción instantánea) La lógica es: «*Si detecto obstáculo, envío velocidad cero INMEDIATAMENTE y cambio el estado a* `PARADO`*.* »

1. **Ciclo T:** El robot está en `AVANZANDO`. Detecta el obstáculo. La transición incluye la acción: se ejecuta `publish_velocity(0)` en ese mismo instante y se marca `next_state = PARADO`.

Consecuencia: El robot frena 100ms antes que en el caso Moore.

Mealy: reacción instantánea ante eventos críticos.

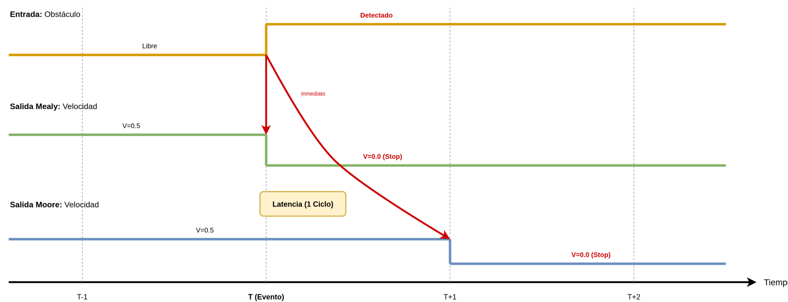


Figura 5.3: Comparativa de latencia: En Moore, la acción se ejecuta al entrar en el estado en el siguiente ciclo ($T + 1$). En Mealy, la acción ocurre en la transición (T).

Analicemos en detalle el diagrama temporal de la figura 5.3. En la parte superior se muestra la señal del sensor de obstáculo (naranja): en el instante T detecta un objeto, pasando de «Libre» a «Detectado».

Este evento dispara la lógica de ambas máquinas de estados, pero con consecuencias temporales diferentes.

La traza verde (salida Mealy) muestra que la velocidad del robot cae a cero *inmediatamente* en el ciclo T. La flecha roja vertical indica esta reacción instantánea: el comando de parada se envía en el mismo ciclo en que se detecta el obstáculo. Esto es posible porque en Mealy la acción forma parte de la transición, no del estado destino.

Por el contrario, la traza azul (salida Moore) mantiene la velocidad original durante todo el ciclo T, y solo en el ciclo $T + 1$ (100ms después) envía el comando de parada. La flecha roja curva ilustra este retardo: el evento se detecta en T, pero la acción solo se ejecuta al entrar en el nuevo estado en $T + 1$. El recuadro amarillo resalta visualmente esta latencia de un ciclo completo.

Esta diferencia de 100ms puede parecer insignificante, pero para un robot que se desplaza a 0.5 $\mathrm{m/s}$, representa 5 centímetros adicionales de desplazamiento antes de detenerse. En aplicaciones críticas (frenado de emergencia, detección de bordes), la arquitectura Mealy es preferible. En cambio, para tareas donde el orden y la claridad del código son prioritarios, Moore ofrece un diseño más modular y mantenible. La figura 5.4 muestra cómo se traduce esta diferencia conceptual en términos de estructura de FSM.

La elección Moore/Mealy depende de los requisitos temporales.

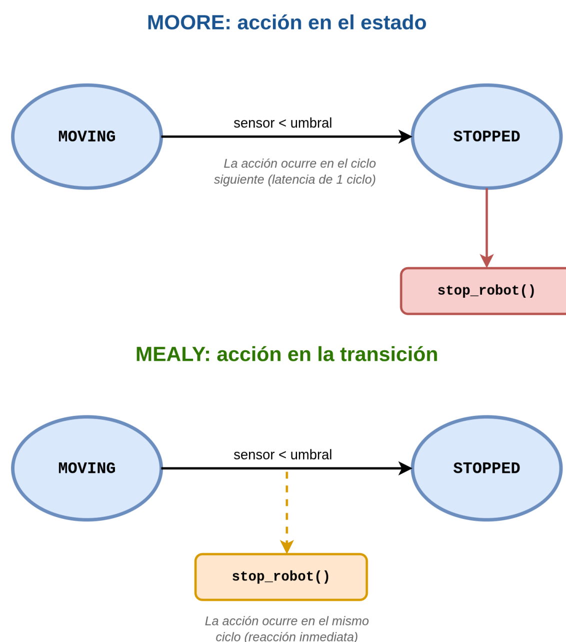


Figura 5.4: Diferencia arquitectónica entre Moore y Mealy. En Moore, la acción `stop_robot()` se ejecuta dentro del estado destino `STOPPED` (ciclo siguiente). En Mealy, la acción forma parte de la transición misma (ciclo actual).

Observemos la diferencia clave en el diagrama de la figura 5.4. En la arquitectura Moore (parte superior), los dos estados `MOVING` y `STOPPED` están conectados por una transición disparada por la condición `sensor <umbral`. La acción crítica `stop_robot()` (recuadro rojo) está ubicada *dentro* del estado `STOPPED`, lo que significa que solo se ejecutará cuando el sistema haya transitado formalmente a ese estado en el ciclo siguiente.

En contraste, en la arquitectura Mealy (parte inferior), aunque la estructura de estados es idéntica, la acción `stop_robot()` (recuadro naranja) está asociada directamente a la transición misma, no al estado destino. La flecha discontinua indica que esta acción se ejecuta *durante* el cambio de estado, es decir, en el mismo ciclo en que se detecta la condición `sensor <umbral`, antes incluso de entrar formalmente al estado `STOPPED`.

Esta diferencia de ubicación de la acción (dentro del estado destino vs. en la transición) es lo que genera la latencia de un ciclo en Moore. Para el alumno, la pregunta clave al implementar una FSM en robótica es: *¿necesito que esta acción ocurra instantáneamente (Mealy) o puedo permitirme esperar un ciclo para mantener el código más organizado (Moore)?* La respuesta depende de los requisitos temporales de la aplicación.

Ubicación de la acción determina la latencia y modularidad.

El problema de la concurrencia

Las FSM clásicas presentan una limitación arquitectónica severa: son secuenciales. El sistema solo puede estar en un único estado en un instante dado. Esto plantea un problema cuando el robot debe realizar múltiples tareas independientes simultáneamente, como «Navegar hacia un punto» y «Monitorizar el nivel de batería».

FSM clásicas no permiten concurrencia de tareas independientes.

Si intentamos modelar esto con una FSM clásica, sufrimos una explosión combinatoria de estados conocida como *producto de estados*. Tendríamos que crear estados artificiales combinados. El fragmento de código 5.1 ilustra esta problemática con un ejemplo concreto.

```
// Si queremos navegar (3 estados) y vigilar la batería (2 estados)
     necesitamos 3x2 = 6 estados explícitos
enum State {
    IDLE_BAT_OK,
    IDLE_BAT_LOW,
    MOVING_BAT_OK,
    MOVING_BAT_LOW,
    RECOVERING_BAT_OK,
    RECOVERING_BAT_LOW
};
// Cada vez que añadimos una funcionalidad, los estados se multiplican.
```

Fragmento C++ 5.1: La pesadilla combinatoria

Analicemos el problema expuesto en el fragmento de código 5.1. Imaginemos que queremos que el robot navegue (con tres estados posibles: `IDLE`, `MOVING`, `RECOVERING`) mientras simultáneamente monitoriza su batería (con dos estados: batería OK o batería baja). Con una FSM plana tradicional, nos vemos obligados a crear el producto cartesiano de ambos comportamientos: $3 \times 2 = 6$ estados combinados.

El resultado es una enumeración artificial donde cada nombre de estado debe codificar la información de *ambas* dimensiones: `IDLE_BAT_OK`, `IDLE_BAT_LOW`, etc. Esta explosión combinatoria tiene consecuencias devastadoras para el mantenimiento del código. Si añadimos una tercera funcionalidad ortogonal (por ejemplo, el estado de las luces del robot: encendidas/apagadas), el número de estados se vuelve a multiplicar: $6 \times 2 = 12$ estados. Con cuatro dimensiones independientes llegaríamos a 24 estados, y así sucesivamente.

Este enfoque es completamente insostenible desde el punto de vista arquitectónico. Cada nueva funcionalidad independiente multiplica el número de estados en lugar de sumarlo, violando el principio de modularidad y haciendo el código prácticamente imposible de mantener y depurar. La figura 5.5 ilustra visualmente esta explosión combinatoria con los 6 estados resultantes del producto cartesiano.

La explosión combinatoria dificulta el mantenimiento y la escalabilidad.

La solución de Harel: Statecharts y regiones ortogonales

En 1987, David Harel propuso una extensión a las FSM llamada **statecharts**. Harel introdujo el concepto de **ortogonalidad** (estados AND).

Statecharts introducen ortogonalidad para evitar la explosión de estados.

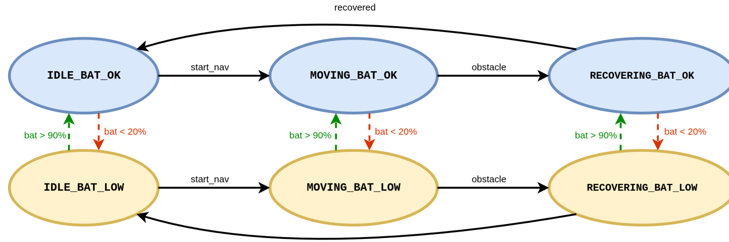


Figura 5.5: Explosión combinatoria en una FSM tradicional. Para combinar navegación (3 estados) con batería (2 estados), se requieren $3 \times 2 = 6$ estados explícitos. Cada estado codifica información de ambas dimensiones (`IDLE_BAT_OK`, `MOVING_BAT_LOW`, etc.), creando una estructura compleja y difícil de mantener. Añadir una tercera dimensión (ej. luces) multiplicaría el número a 12 estados.

En un statechart, un superestado puede dividirse en regiones paralelas. El robot está, simultáneamente, en un estado de la región A **Y** en un estado de la región B. En código, esto se traduce en descomponer el sistema en múltiples máquinas de estados más pequeñas que se ejecutan secuencialmente en el mismo ciclo. La figura 5.6 ilustra esta descomposición ortogonal como solución al problema mostrado en la figura 5.5.

La ortogonalidad permite modelar comportamientos independientes en paralelo.

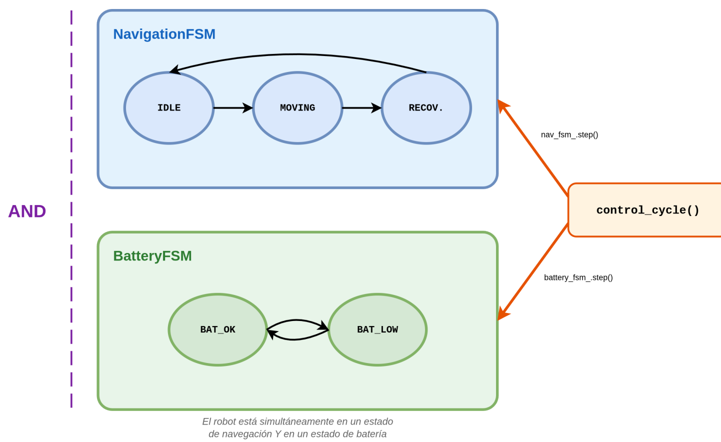


Figura 5.6: Descomposición ortogonal en dos FSM independientes. La NavigationFSM gestiona la navegación (3 estados) y la BatteryFSM monitoriza la batería (2 estados). Ambas se ejecutan en paralelo conceptual en cada ciclo de control, evitando la explosión combinatoria ($3 + 2 = 5$ estados en lugar de $3 \times 2 = 6$ estados combinados).

La descomposición ortogonal reduce la complejidad de la FSM.

Analicemos la solución presentada en la figura 5.6, que contrasta directamente con el problema de la figura 5.5. En lugar de crear un enum con 6 estados combinados, definimos dos FSM completamente independientes: `NavigationFSM` (región superior azul) y `BatteryFSM` (región inferior verde). Cada una encapsula su propia lógica de estados sin conocer la existencia de la otra.

La clave arquitectónica está en el concepto de ejecución ortogonal, representado por el símbolo AND y la línea discontinua vertical. En cada iteración del bucle de control (`control_cycle()`), se invoca el método `step()` de ambas FSM. Aunque estas llamadas ocurren secuencialmente en el código, desde el punto de vista conceptual y de diseño, ambas máquinas están activas *simultáneamente* en el mismo ciclo de control. El robot está, en todo momento, en un estado de navegación **Y** en un estado de batería.

Esta descomposición ortogonal tiene una ventaja matemática fundamental sobre el enfoque tradicional: en lugar de multiplicar estados ($3 \times 2 = 6$ estados combinados como en la figura 5.5), los sumamos ($3 + 2 = 5$ estructuras de estado independientes). La complejidad crece linealmente, no exponencialmente. Si añadimos el sistema de luces (2 estados), pasamos de 5 a 7 componentes en el enfoque ortogonal, frente a los 12 estados combinados del enfoque tradicional. Este enfoque respeta el principio de separación de responsabilidades: cada FSM gestiona una dimensión ortogonal del comportamiento del robot.

La ortogonalidad respeta la modularidad y escalabilidad del diseño.

Modelado de misiones y tareas

En arquitecturas deliberativas modernas, el lugar más natural para una máquina de estados finitos es la capa de **misión**. La FSM modela la macrosecuencia del sistema: define las fases principales de la misión, los criterios de transición entre ellas y las condiciones de éxito o fallo global. Cada estado de la FSM de misión representa una *fase* o *modo* operativo relevante (por ejemplo, `PLANNING`, `EXECUTING`, `REPORTING`), y las transiciones reflejan eventos o condiciones que justifican el cambio de fase.

La misión, así modelada, se descompone en **tareas** que constituyen las unidades ejecutables concretas. Cada vez que la FSM de misión entra en un nuevo estado, activa la tarea correspondiente (por ejemplo, `INSPECT_AREA`, `TRANSPORT_OBJECT`, etc.). Las tareas pueden implementarse mediante otros modelos de control, siendo los Behavior Trees (BT) una opción habitual por su capacidad de orquestar acciones concurrentes y gestionar recuperación. En sistemas sencillos, una tarea también puede ser una FSM, pero en arquitecturas escalables es preferible reservar la FSM para la misión y emplear otros mecanismos más adecuados para las tareas.

Las **capacidades** son los servicios gobernables y reutilizables que implementan el "cómo" de cada acción elemental (navegar, manipular, percibir, etc.), con contratos claros de precondiciones, feedback, resultado y cancelación. Las tareas consumen capacidades y consolidan su feedback, reportando progreso y causas de fallo hacia la misión. Así, la misión no necesita espiar telemetría de bajo nivel, sino que recibe información semántica relevante para decidir si replanificar, cambiar de objetivo o modificar la política.

Este diseño compositivo habilita arquitecturas híbridas: la FSM de misión gestiona el flujo global y la persistencia de estado, mientras que las tareas orquestan capacidades y gestionan la ejecución concreta, pudiendo emplear BT, sub-FSM (FSM anidada dentro de una tarea) u otros modelos según convenga. Por ejemplo, la fase `EXPLORE_AREAS` de la misión puede activar una tarea implementada como BT que coordina navegación, inspección visual y monitorización de batería de forma reactiva, mientras la FSM de misión permanece en ese estado hasta que la tarea reporte completitud o fallo.

FSM modela la misión y delega la ejecución a tareas especializadas.

La contribución arquitectónica fundamental de la FSM en este contexto es definir el *cuándo* y el *en qué fase* se encuentra el sistema, y bajo qué condiciones debe transitar a otra. No define el *cómo* se implementan las operaciones complejas, esa responsabilidad reside en las tareas y capacidades de bajo nivel. Esta separación de responsabilidades mantiene un diseño modular, escalable y trazable.

FSM orquesta el flujo global; tareas y capacidades implementan las operaciones.

Ventajas y problemas de escalabilidad

Las FSM ofrecen ventajas claras, como la simplicidad conceptual, la facilidad de implementación y la transparencia del comportamiento. Resultan especialmente adecuadas para sistemas con un número limitado de estados y transiciones bien definidas. Sin embargo, a medida que el número de estados crece, las FSM tienden a volverse difíciles de mantener y extender. La explosión de estados y transiciones puede conducir a diseños complejos y poco legibles, conocidos como *state explosion*.

FSM: simples y transparentes, pero poco escalables para sistemas grandes.

Máquinas de estados jerárquicas

Además de la ortogonalidad, Harel formalizó la **jerarquía** (estados OR). Un estado puede contener dentro de sí otra máquina de estados completa. Este enfoque aporta tres ventajas arquitectónicas fundamentales:

1. **Abstracción:** Visto desde fuera, un superestado como `NAVEGAR` es una caja negra.
2. **Reutilización:** Una sub-máquina bien diseñada puede encapsularse y reutilizarse.
3. **Factorización de transiciones:** Si se define una transición que sale del superestado (ej. `ERROR_GLOBAL`), esta aplica a todos sus subestados internos, simplificando drásticamente el diagrama.

Para ilustrar este concepto, consideremos un robot de limpieza doméstico. La figura 5.7 muestra cómo un superestado `CLEANING` encapsula tres subestados diferentes (`VACUUMING`, `MOPPING`, `DRYING`). La ventaja clave es que la transición de emergencia (batería baja) no necesita repetirse desde cada subestado individual: basta con definirla una vez desde el superestado. Esto reduce drásticamente el número de flechas en el diagrama y facilita el mantenimiento (si añadimos un nuevo subestado dentro de `CLEANING`, automáticamente hereda la transición de emergencia).

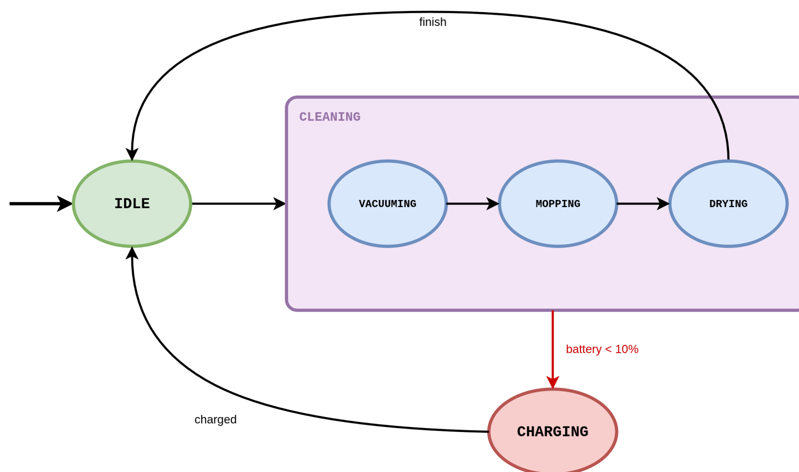


Figura 5.7: Ejemplo de FSM jerárquica para un robot de limpieza. El superestado `CLEANING` (rectángulo morado) encapsula tres subestados. La transición de emergencia por batería baja se define una única vez desde el superestado, aplicándose automáticamente a todos los subestados internos. Esto evita tener que dibujar tres flechas separadas.

Observemos cómo en la figura 5.7, la flecha roja gruesa sale directamente del rectángulo morado que representa el superestado `CLEANING`, no de cada círculo individual. Esto significa que *independientemente de si el robot está aspirando, fregando o secando*, si la batería cae por debajo del 10 %, el sistema transita inmediatamente a `CHARGING`. Esta factorización es especialmente valiosa en sistemas complejos: si el superestado `CLEANING` tuviera 10 subestados en lugar de 3, seguiríamos necesitando una única transición de emergencia.

5.2. Aplicación

Las FSM encuentran aplicación natural en la arquitectura robótica, principalmente en los niveles de tarea y misión. Esta sección ilustra mediante ejemplos conceptuales cómo las FSM modelan comportamientos a diferentes escalas temporales, desde secuencias de operaciones (tareas) hasta objetivos de largo plazo (misiones).

Es importante destacar que desde los estados de estas FSM se *pueden invocar e interactuar* con las capacidades del robot (navegación, manipulación, percepción, etc.), aunque no todos los estados necesariamente invocan capacidades. Algunos estados pueden realizar procesamiento interno, cálculos, o simplemente esperar eventos. Las capacidades en sí son módulos complejos que encapsulan comportamientos de bajo nivel y no necesariamente se modelan con FSM propias. Por ejemplo, una tarea puede activar la capacidad de navegación Nav2 desde uno de sus estados, o invocar un sistema de visión para detectar objetos, mientras que otros estados pueden limitarse a validar condiciones o realizar transiciones de control.

FSM a nivel de tarea

En el nivel de tarea, las FSM coordinan la ejecución de comportamientos para lograr objetivos más complejos. Una tarea coordina el *cuándo* y *en qué orden* ocurren las acciones, invocando capacidades cuando se requieren operaciones complejas, pero sin implementar su lógica interna.

Consideremos la tarea `TransportObject`, que debe recoger un objeto de una ubicación y llevarlo a otra. La figura 5.8 muestra su estructura. El robot comienza navegando hacia la ubicación del objeto (`NAVIGATE_TO_PICKUP`). Al llegar, invoca la capacidad de agarre (`GRASPING`). Si el agarre tiene éxito, navega hacia la ubicación destino (`NAVIGATE_TO_PLACE`). Una vez allí, libera el objeto (`RELEASING`). Si cualquier paso falla, transita a un estado de recuperación (`RECOVERY`) que puede reintentar o abortar.

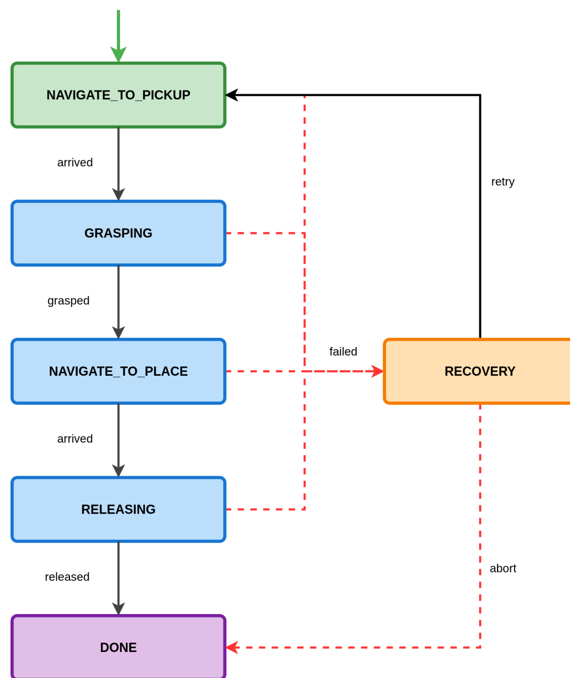


Figura 5.8: FSM a nivel de tarea: `TransportObject`. Coordina la secuencia de estados, algunos de los cuales invocan capacidades del robot (navegación, agarre, liberación) cuando se requieren operaciones complejas. Las transiciones principales (negro sólido) indican flujo nominal, mientras que las transiciones de error (rojo discontinuo) conducen a recuperación.

Observemos cómo algunos estados de la tarea *invocan* capacidades del robot sin implementarlas, mientras que otros pueden realizar procesamiento interno o simplemente gestionar transiciones. Por ejemplo:

- El estado `NAVIGATE_TO_PICKUP` activa la capacidad de navegación (por ejemplo, Nav2 en ROS 2) con la pose objetivo del objeto.
- El estado `GRASPING` invoca la capacidad de manipulación del gripper y espera su resultado.

- El estado `RELEASING` utiliza nuevamente la capacidad de manipulación para abrir el gripper.
- El estado `DONE` simplemente señaliza la finalización de la tarea sin invocar capacidades.

Si una capacidad reporta éxito, la tarea avanza al siguiente estado; si reporta error, la tarea gestiona la recuperación. Esta separación de responsabilidades es fundamental: la FSM de la tarea decide la *estrategia* (qué hacer y cuándo), mientras que las capacidades ejecutan las *operaciones* (cómo hacerlo). Las capacidades son módulos complejos y reutilizables que pueden ser invocados desde múltiples tareas diferentes.

FSM a nivel de misión

En el nivel de misión, las FSM coordinan múltiples tareas de larga duración para lograr objetivos de alto nivel. Una misión puede persistir durante horas o días, transitando entre fases bien diferenciadas.

Consideremos la misión `WarehouseInspection`, donde un robot debe inspeccionar un almacén completo, detectar anomalías y generar un informe. La figura 5.9 muestra su estructura de alto nivel. La misión comienza con una fase de planificación (`PLANNING`), donde se genera la ruta óptima de inspección. Luego transita a `EXECUTING`, donde se ejecutan múltiples tareas de inspección (navegar a zonas, capturar imágenes, analizar). Si se detecta una anomalía crítica, puede transitar a `INVESTIGATING` para inspección detallada. Al completar el recorrido, transita a `REPORTING`, donde genera y envía el informe final.

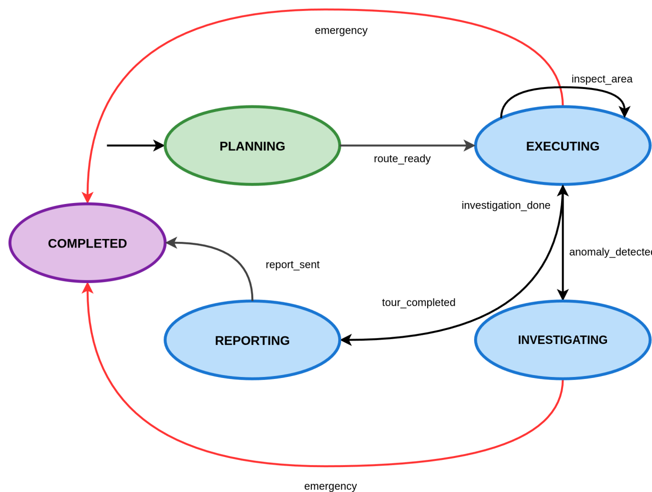


Figura 5.9: FSM a nivel de misión: `WarehouseInspection`. Coordina fases de larga duración, cada una ejecutando múltiples tareas. Las transiciones naranjas indican eventos especiales que alteran el flujo, mientras que las rojas discontinuas son abortos de emergencia.

Lo característico de las FSM de misión es que cada estado puede persistir durante largos periodos (minutos u horas), ejecutando internamente múltiples tareas. Es importante destacar que *cada estado de una FSM de misión es en sí mismo una máquina de estados (tarea) o un behavior tree* (que se estudia en el tema siguiente). El estado `EXECUTING`, por ejemplo, invoca repetidamente la tarea `InspeccionarZona` para diferentes ubicaciones del almacén, donde cada invocación ejecuta su propia FSM de tarea. La FSM de misión opera en un nivel de abstracción superior: no gestiona los detalles de cada inspección, simplemente decide cuándo iniciar cada fase, cuándo investigar anomalías, y cuándo generar el informe, delegando la ejecución concreta a las tareas subyacentes.

Diseño compositivo: FSM de misión y tarea

En sistemas robóticos reales, los niveles de misión y tarea se componen jerárquicamente. Una misión contiene una FSM de alto nivel cuyos estados activan tareas. Cada tarea tiene su propia FSM cuyos estados pueden invocar capacidades del robot cuando sea necesario.

La figura 5.10 ilustra esta arquitectura compositiva para el ejemplo de inspección de almacén.

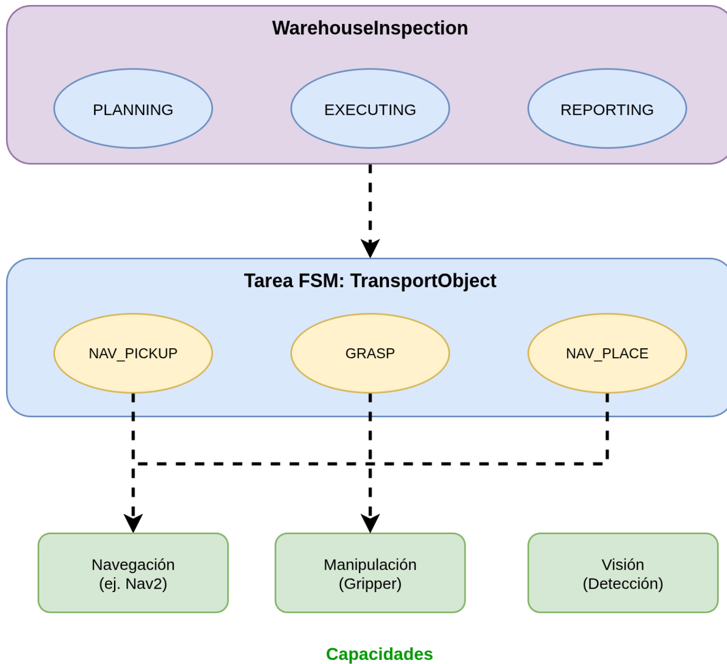


Figura 5.10: Composición jerárquica en la arquitectura misión–tarea–capacidad. La misión contiene una FSM que activa tareas (flecha morada). Cada tarea tiene su propia FSM que invoca capacidades según sea necesario (flechas azules). Las capacidades son módulos complejos reutilizables, no necesariamente modelados con FSM.

Esta composición jerárquica proporciona modularidad y reutilización. Las capacidades de navegación y manipulación pueden utilizarse en múltiples tareas diferentes (transportar, clasificar, inspeccionar, apilar). La tarea `TransportObject` puede invocarse desde múltiples misiones (inspección, reabastecimiento, ordenación). Cada nivel opera a su propia escala temporal: las capacidades responden en tiempo real (milisegundos a segundos), las tareas se completan en segundos a minutos, y las misiones persisten durante minutos a horas.

La implementación práctica de estos patrones, incluyendo integración con ROS 2 y ejemplos de código funcional, se desarrolla en detalle en las prácticas del curso.

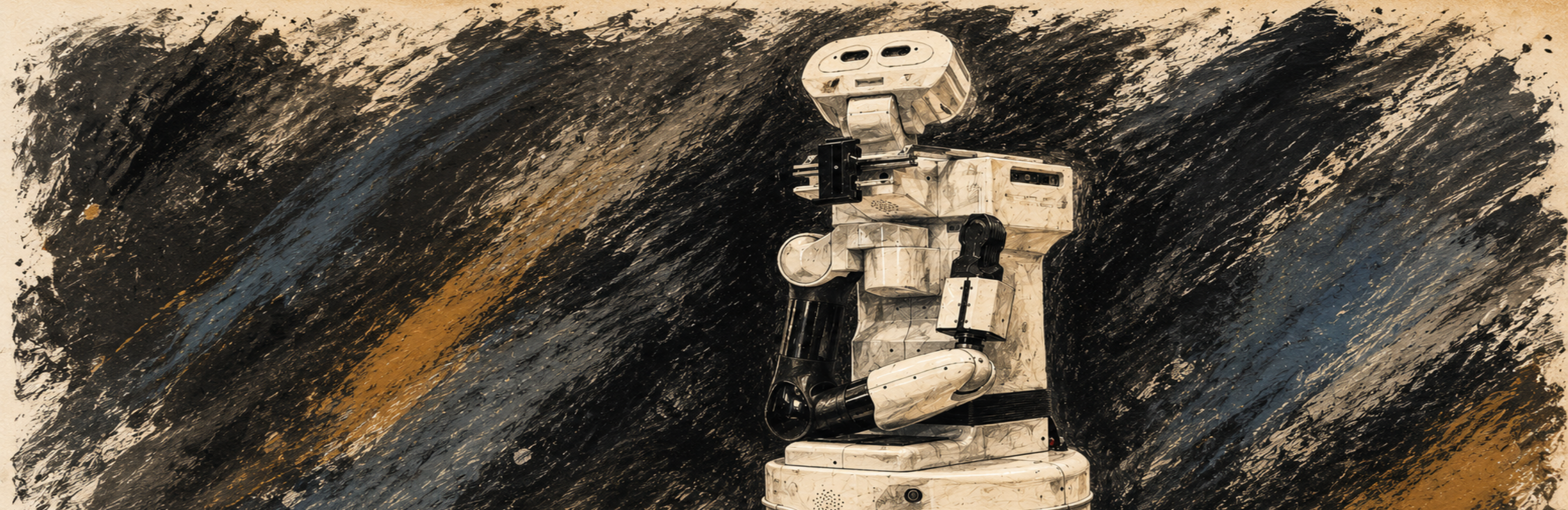
5.3. Conclusiones

Las máquinas de estados finitos constituyen una herramienta fundamental en la ingeniería de software robótico. Su valor arquitectónico reside en hacer explícito el comportamiento del sistema mediante estados bien definidos y transiciones claras.

Las FSM constituyen la base sobre la cual se construyen sistemas más sofisticados como árboles de comportamiento (Behavior Trees) o grafos de comportamiento jerárquicos. Sin embargo, para tareas de complejidad media, una FSM bien diseñada proporciona un equilibrio óptimo entre expresividad, eficiencia y facilidad de depuración, convirtiéndola en

una herramienta indispensable en el arsenal del ingeniero de robótica autónoma.

Los ejemplos prácticos de implementación y casos de uso se desarrollan en detalle en las prácticas del curso, donde se aplican estos conceptos teóricos en sistemas robóticos reales.



6 Árboles de comportamiento

6.1. Principios teóricos

6.1 **Principios teóricos** **78**
Origen histórico: de los videojuegos a la robótica . . 78
Motivación de los Behavior Trees 79
Anatomía de un Behavior Tree 80
Nodos de control fundamentales 82
Decoradores 88
Modelado de misiones y tareas 89
Reutilización de comportamientos 90
Comparación con FSM . . . 91
6.2 **Aplicación** **92**
Behavior Trees en ROS 2 . . 92
Blackboard y puertos 93
Ejemplo ilustrativo e integración con capacidades 94

Origen histórico: de los videojuegos a la robótica

Los Behavior Trees (BT, *Behavior Trees*) no nacieron en el campo de la robótica, sino en la industria de los videojuegos, específicamente para resolver el problema de modelar el comportamiento de personajes no jugadores (NPCs, *Non-Player Characters*). A principios de los años 2000, los desarrolladores de videojuegos enfrentaban el mismo desafío que hoy afrontan los ingenieros de robótica: cómo diseñar comportamientos complejos, adaptativos y reactivos de forma escalable.

Los Behavior Trees nacen en videojuegos para modelar comportamientos complejos y adaptativos.

Las FSM tradicionales dominaban el diseño de IA en videojuegos, pero presentaban limitaciones críticas. En juegos AAA con docenas de enemigos diferentes, cada uno con múltiples comportamientos contextuales (patrullar, perseguir, atacar, buscar cobertura, pedir refuerzos, huir), las FSM resultaban en grafos de estados inmanejables con cientos de transiciones. Modificar un comportamiento requería rastrear y actualizar transiciones dispersas por todo el grafo, haciendo el mantenimiento extremadamente costoso.

Las FSM sufrían explosión de estados y difícil mantenimiento en IA compleja.

El punto de inflexión llegó con el desarrollo de Halo 2 (Bungie Studios, 2004), donde el equipo de IA necesitaba modelar comportamientos tácticos sofisticados para los enemigos Covenant. Damián Isla, diseñador de IA del equipo, propuso una arquitectura jerárquica basada en árboles que permitía componer comportamientos complejos a partir de bloques reutilizables. Esta arquitectura inicial evolucionó en Halo 3 (2007), donde Isla formalizó completamente el concepto de BT y lo presentó públicamente en la *Game Developers Conference* (GDC) de 2005.

La presentación de Isla, titulada «*Handling Complexity in the Halo 2 AI*», tuvo un impacto revolucionario en la industria. Explicaba cómo los BT permitían:

- Expresar jerarquías de decisión de forma natural (objetivo estratégico → táctica → acción concreta)
- Reutilizar subárboles completos entre diferentes tipos de enemigos
- Iterar rápidamente en el diseño sin refactorizar toda la arquitectura
- Visualizar el comportamiento de forma intuitiva para diseñadores no programadores

Tras Halo, los BT se adoptaron rápidamente en la industria: *Spore* (2008), *Crysis* (2007), y posteriormente prácticamente todos los motores de videojuegos modernos (Unreal Engine, Unity, CryEngine) incorporaron sistemas BT nativos.

La transición a la robótica ocurrió alrededor de 2010-2012, cuando investigadores en robótica de servicio y vehículos autónomos comenzaron a experimentar con BT para orquestar misiones complejas. Los pioneros fueron Michele Colledanchise y Petter Ögren del KTH Royal Institute of Technology (Suecia), quienes en 2014 publicaron el primer trabajo académico riguroso sobre BT aplicados a robótica: *«Behavior Trees in Robotics and AI: An Introduction»*. Este trabajo formalizó matemáticamente las propiedades de los BT (reactividad, modularidad, determinismo) y demostró su aplicabilidad en sistemas robóticos reales.

Desde entonces, los BT se han consolidado como una alternativa arquitectónica a las FSM en robótica autónoma, especialmente en el ecosistema ROS 2, donde librerías como BehaviorTree.CPP proporcionan implementaciones industriales robustas. La trayectoria histórica es notable: una técnica desarrollada para que aliens virtuales ataquen de forma convincente en un videojuego ahora orquesta robots reales en almacenes, hospitales y vehículos autónomos.

Los BT se consolidan en robótica por su modularidad y reactividad.

Motivación de los Behavior Trees

Los BT surgen como una respuesta a las limitaciones de escalabilidad y mantenibilidad de otros modelos de generación de comportamiento, en particular las máquinas de estados finitos. A medida que los sistemas robóticos crecen en complejidad, resulta necesario disponer de mecanismos que permitan estructurar el comportamiento de forma jerárquica y modular.

Los BT resuelven la escalabilidad y mantenibilidad en sistemas complejos.

Un BT organiza el comportamiento como un árbol de nodos que se evalúan de manera recurrente, permitiendo combinar decisiones reactivas con estructuras de control más complejas. Este enfoque facilita la representación explícita de prioridades, secuencias y condiciones de ejecución.

El BT estructura el comportamiento como un árbol jerárquico y reactivo.

Desde un punto de vista arquitectónico, los BT proporcionan un marco claro para describir comportamientos complejos sin caer en la explosión de estados característica de otros modelos.

Los BT evitan la explosión de estados mediante composición modular.

Consideremos un ejemplo motivador: un robot de servicio debe recoger un objeto de una mesa. La tarea involucra múltiples subtareas: navegar hasta la mesa, detectar el objeto, aproximar el brazo, agarrar, y verificar el agarre. En una FSM, necesitaríamos definir estados explícitos para cada fase y transiciones entre ellos. Si añadimos requisitos como «verificar batería antes de cada acción» o «reintentar si falla el agarre», la FSM crece exponencialmente en complejidad.

Un BT modela esta misma tarea como una jerarquía: un nodo `Sequence` que coordina las subtareas, con nodos `Fallback` que gestionan los reintentos. Añadir nuevos comportamientos (verificar batería) se reduce a insertar un nodo en la jerarquía, sin modificar la estructura existente. Esta composición modular es la ventaja arquitectónica fundamental de los BT.

La modularidad permite añadir o modificar comportamientos fácilmente.

Anatomía de un Behavior Tree

Un BT se compone de nodos organizados jerárquicamente. Cada nodo representa una unidad de comportamiento que, al ejecutarse, devuelve uno de tres estados posibles: `SUCCESS`, `FAILURE` o `RUNNING`. Este modelo de ejecución simple pero poderoso permite expresar comportamientos complejos mediante composición.

Un BT se basa en nodos jerárquicos con tres posibles estados de retorno.

Estados de retorno

Los tres estados de retorno definen el protocolo de comunicación entre nodos:

- **SUCCESS:** El nodo completó su tarea satisfactoriamente. Por ejemplo, una acción `GrabObject` retorna `SUCCESS` cuando el objeto ha sido agarrado correctamente.
- **FAILURE:** El nodo no pudo completar su tarea. Por ejemplo, `GrabObject` retorna `FAILURE` si no detecta ningún objeto en el gripper tras intentar el agarre.
- **RUNNING:** El nodo está en ejecución y requiere más tiempo. Las acciones que tardan múltiples ciclos (navegación, movimiento de brazo) retornan `RUNNING` en cada tick hasta completarse.

Este protocolo asíncrono es fundamental: permite que acciones de larga duración se ejecuten sin bloquear la evaluación del árbol. En cada tick, el árbol consulta el estado de los nodos `RUNNING` y actualiza el flujo de control según sus respuestas.

El protocolo asíncrono permite acciones largas sin bloquear el árbol.

Tipos de nodos

Los nodos se clasifican en cuatro categorías según su función en el árbol:

Nodos de control: Coordinan la ejecución de sus hijos según reglas específicas. No realizan acciones directamente, solo gestionan el flujo. Los principales son:

- `Sequence`: Ejecuta hijos en orden hasta que uno falla
- `Fallback` (o `Selector`): Ejecuta hijos hasta que uno tiene éxito
- `Parallel`: Ejecuta múltiples hijos simultáneamente

Nodos decoradores: Modifican el comportamiento de un único hijo. Actúan como transformadores o filtros:

- `Inverter`: Invierte el resultado (`SUCCESS`↔`FAILURE`)
- `Retry`: Reintenta el hijo N veces si falla
- `ForceSuccess`: Convierte cualquier resultado en `SUCCESS`
- `ForceFailure`: Convierte cualquier resultado en `FAILURE`
- `Timeout`: Falla si el hijo no completa en tiempo límite

Nodos de acción: Hojas que ejecutan operaciones concretas: mover el robot, activar un actuador, enviar un mensaje. Interfaz con las capacidades del sistema.

Nodos condicionales: Hojas que evalúan predicados sin efectos secundarios: verificar nivel de batería, comprobar si un objeto está visible, evaluar distancia a objetivo. Retornan inmediatamente (`SUCCESS/FAILURE`), nunca `RUNNING`.

La figura 6.1 muestra la representación gráfica estándar de cada tipo de nodo y sus símbolos asociados.

Los nodos se clasifican en control, decoradores, acción y condición.

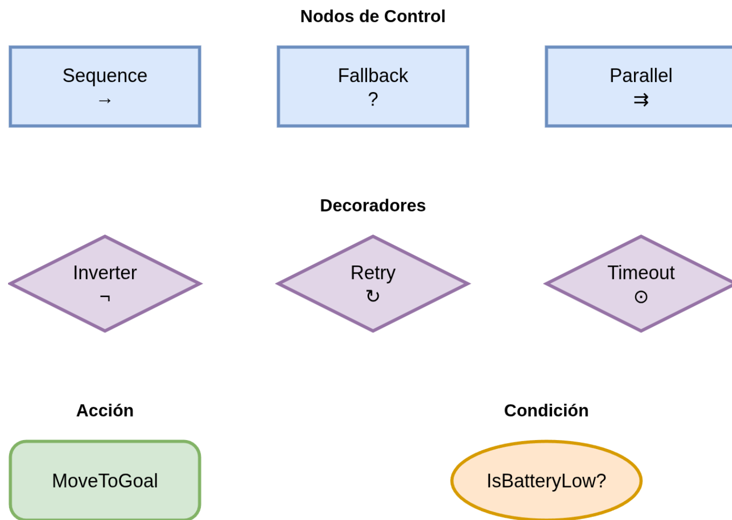


Figura 6.1: Tipos de nodos en un Behavior Tree y su representación gráfica

La tabla 6.1 resume las diferencias fundamentales entre los tipos de nodos.

Tabla 6.1: Comparación entre tipos de nodos en Behavior Trees

Tipo de nodo	**Número de hijos**	**Estados que puede retornar**	**Función principal**
Nodos de Control	1:N	`SUCCESS`, `FAILURE`, `RUNNING`	Orquestar la ejecución de múltiples hijos según reglas de control de flujo
Decoradores	1	`SUCCESS`, `FAILURE`, `RUNNING`	Modificar o transformar el comportamiento/resultado del hijo
Nodos de Acción	0	`SUCCESS`, `FAILURE`, `RUNNING`	Ejecutar operaciones concretas con efectos en el sistema o entorno
Nodos de Condición	0	`SUCCESS`, `FAILURE`	Evaluar predicados sin efectos secundarios; evaluación instantánea

Ejecución: el concepto de tick

El árbol se evalúa mediante **ticks** periódicos. En cada tick, se envía una señal desde la raíz que se propaga recursivamente por el árbol siguiendo las reglas de cada nodo de control. Esta evaluación reactiva garantiza que el comportamiento responda inmediatamente a cambios en el entorno.

El tick periódico permite reactividad inmediata ante cambios.

El algoritmo de tick sigue este patrón recursivo:

1. El nodo raíz recibe un tick
2. Según su tipo, decide qué hijos tickear y en qué orden

3. Cada hijo retorna `SUCCESS`, `FAILURE` o `RUNNING`
4. El nodo padre agrega estos resultados según su lógica
5. El resultado agregado se propaga hacia arriba

La figura 6.2 ilustra la propagación de un tick en un árbol simple. El `Sequence` recibe un tick desde arriba. Ejecuta secuencialmente sus hijos: primero evalúa `HasGoal?` (1), que retorna `SUCCESS`. Continúa con el `Fallback` (2), que intenta `Navigate` (3). Como `Navigate` retorna `RUNNING`, el `Fallback` también retorna `RUNNING` y el `Sequence` no evalúa `AtGoal?`. En el siguiente tick, el `Sequence` recordará que debe continuar desde el `Fallback`.

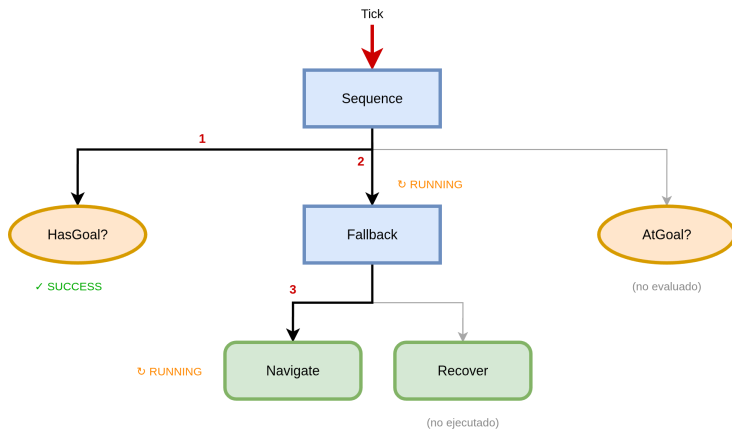


Figura 6.2: Propagación de tick y agregación de resultados en un Sequence

Nodos de control fundamentales

Los nodos de control definen el flujo de ejecución del árbol. Comprender su semántica precisa es esencial para diseñar comportamientos correctos.

Secuencias

Sequence: versión con memoria Un nodo `Sequence` (símbolo: $\longrightarrow$) ejecuta sus hijos de izquierda a derecha. Retorna:

- `SUCCESS` si **todos** los hijos retornan `SUCCESS`
- `FAILURE` tan pronto como un hijo retorna `FAILURE`
- `RUNNING` si el hijo actual retorna `RUNNING`

La semántica es equivalente al operador lógico AND: todas las subtareas deben tener éxito para que la secuencia tenga éxito. Se utiliza para expresar requisitos en cadena: «primero haz A, luego B, luego C».

La figura 6.3 muestra un ejemplo concreto de secuencia de navegación con tres pasos obligatorios.

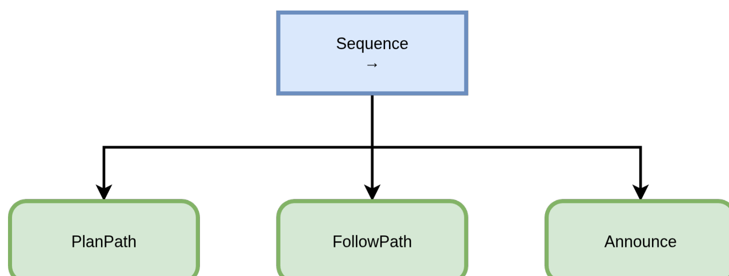


Figura 6.3: Nodo Sequence: todas las acciones deben completarse en orden

Caso de uso típico: Secuencias de acciones que deben completarse en orden estricto, donde el fallo de cualquier paso invalida toda la operación. Ejemplo: «verificar precondiciones, ejecutar acción principal, verificar postcondiciones».

ReactiveSequence: versión sin memoria El `ReactiveSequence` elimina la memoria de estado: en cada tick, reevalúa desde el primer hijo, independientemente de dónde estaba en el tick anterior. Esta diferencia es fundamental para la reactividad del sistema.

Consideremos un ejemplo: un robot debe mantener una condición de seguridad (`IsBatteryOK?`) mientras navega (`Navigate`). La figura 6.4 muestra la estructura del árbol y el comportamiento comparado entre `Sequence` estándar y `ReactiveSequence`.

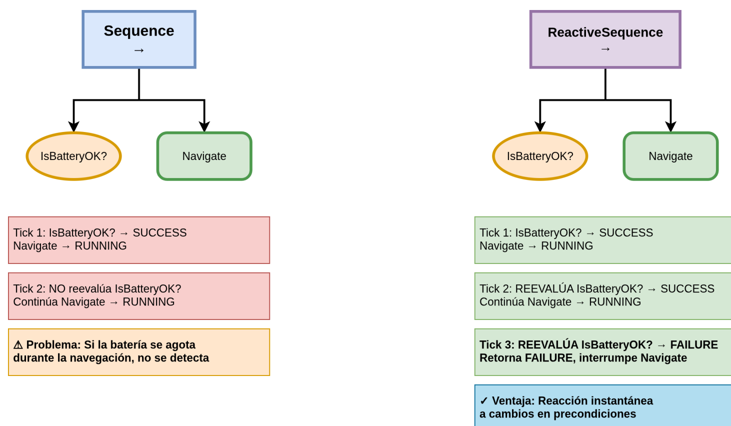


Figura 6.4: Comparación de comportamiento entre Sequence y ReactiveSequence: el árbol contiene un nodo que verifica la condición de batería (IsBatteryOK?) seguido de una acción de navegación (Navigate). El Sequence estándar mantiene memoria y no reevalúa la condición mientras Navigate está en ejecución, mientras que ReactiveSequence reevalúa la condición en cada tick, permitiendo detectar inmediatamente cuando la batería se agota durante la navegación.

Comportamiento con Sequence:

1. Tick 1: Evalúa `IsBatteryOK?` → `SUCCESS`. Continúa a `Navigate` → `RUNNING`
2. Tick 2: El `Sequence` recuerda que está en `Navigate`, **no reevalúa** `IsBatteryOK?`. Continúa con `Navigate` → `RUNNING`
3. Ticks 3-N: Mismo comportamiento. `IsBatteryOK?` nunca se vuelve a verificar
4. Problema: Si la batería se agota durante la navegación, el sistema no reacciona hasta que `Navigate` complete

Comportamiento con ReactiveSequence:

1. Tick 1: Evalúa `IsBatteryOK?` → `SUCCESS`. Continúa a `Navigate` → `RUNNING`
2. Tick 2: **Reevalúa** `IsBatteryOK?` → `SUCCESS`. Continúa con `Navigate` → `RUNNING`
3. Tick 3: **Reevalúa** `IsBatteryOK?` → `FAILURE` (batería baja)
4. El `ReactiveSequence` retorna `FAILURE` inmediatamente, interrumpiendo `Navigate`
5. Ventaja: Reacción instantánea a cambios en las precondiciones

El `ReactiveSequence` verifica precondiciones continuamente, lo que resulta muy útil cuando las condiciones iniciales pueden invalidarse durante la ejecución de acciones largas.

Selectores

Fallback: versión con memoria Un nodo `Fallback` (también llamado `Selector`, símbolo: **?**) ejecuta sus hijos de izquierda a derecha buscando uno que tenga éxito. Retorna:

- `SUCCESS` tan pronto como un hijo retorna `SUCCESS`
- `FAILURE` si **todos** los hijos retornan `FAILURE`
- `RUNNING` si el hijo actual retorna `RUNNING`

La semántica es equivalente al operador lógico OR: basta que una subtarea tenga éxito. Se utiliza para expresar *alternativas con prioridad*: «intenta A; si falla, intenta B; si falla, intenta C».

La figura 6.5 muestra un nodo `Fallback` raíz que coordina tres estrategias para agarrar un objeto ordenadas por preferencia. En el diagrama, las flechas indican el orden de ejecución: de izquierda a derecha, de más óptima a menos óptima. El comportamiento es el siguiente:

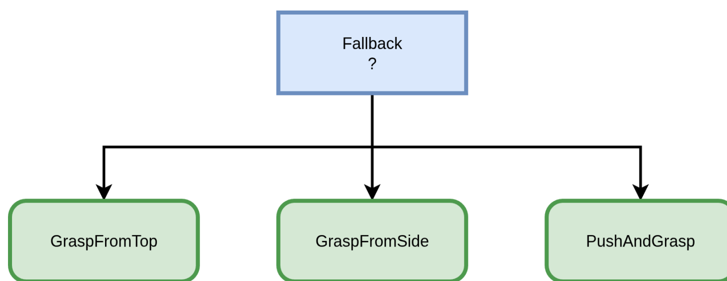


Figura 6.5: Nodo Fallback: intenta alternativas hasta encontrar una que funcione

1. El `Fallback` intenta primero `GraspFromTop`, agarrar el objeto desde arriba con el gripper estándar. Si el agarre tiene éxito (por ejemplo, el objeto tiene una geometría favorable), el nodo retorna `SUCCESS` inmediatamente y los nodos hermanos a la derecha no se ejecutan.
2. Si `GraspFromTop` retorna `FAILURE` (por ejemplo, el objeto es demasiado plano o está pegado a la superficie), el `Fallback` procede a evaluar el segundo hijo: `GraspFromSide`, que intenta agarrar el objeto lateralmente.
3. Si `GraspFromSide` también falla (por ejemplo, no hay espacio lateral suficiente), el `Fallback` recurre a la última opción: `PushAndGrasp`, que primero empuja el objeto para crear espacio y luego intenta agarrarlo.
4. Solo si las tres estrategias fallan, el nodo `Fallback` retorna `FAILURE` a su padre, indicando que el objeto no es agarrable con las técnicas disponibles.

Este patrón implementa degradación elegante: el sistema intenta primero la solución óptima y recurre progresivamente a alternativas menos eficientes pero más robustas. La jerarquía de prioridades está codificada implícitamente en el orden de los hijos, sin necesidad de lógica condicional explícita.

Caso de uso típico: Implementar tolerancia a fallos mediante estrategias de respaldo. Los hijos se ordenan de más eficiente/preferible a menos. Ejemplo: «intenta plan A (óptimo); si falla, plan B (subóptimo); si falla, plan C (seguro pero lento)».

ReactiveFallback: versión sin memoria El `ReactiveFallback` no mantiene memoria: en cada tick, reevalúa desde el primer hijo. Esto permite cambios de prioridad instantáneos cuando aparecen condiciones de mayor precedencia.

Consideremos un robot que debe priorizar la carga de batería sobre otras tareas cuando la batería está crítica. La figura 6.6 muestra la estructura del árbol y el comportamiento comparado entre `Fallback` estándar y `ReactiveFallback`.

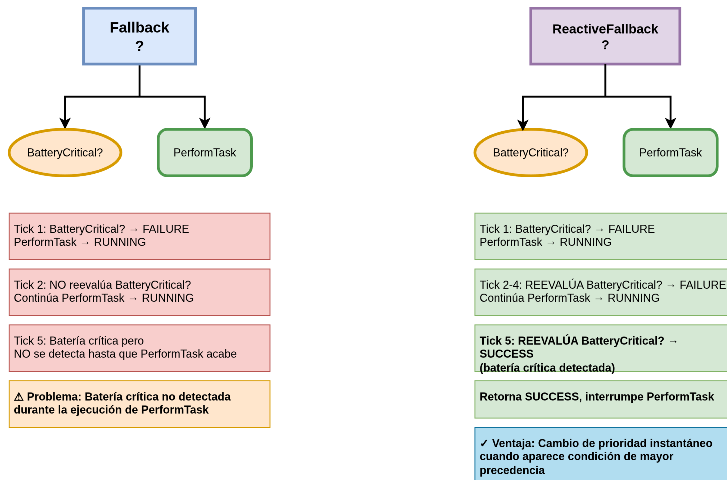


Figura 6.6: Comparación de comportamiento entre Fallback y ReactiveFallback: el árbol verifica si la batería está crítica (BatteryCritical?) antes de continuar con la tarea normal (PerformTask). El Fallback estándar mantiene memoria y no reevalúa BatteryCritical? mientras PerformTask está en ejecución, mientras que ReactiveFallback reevalúa en cada tick, permitiendo interrumpir la tarea inmediatamente cuando la batería se vuelve crítica.

Comportamiento con Fallback estándar:

1. Tick 1: Evalúa `BatteryCritical?` → `FAILURE` (batería OK). Continúa a `PerformTask` → `RUNNING`
2. Tick 2: El `Fallback` recuerda que está en `PerformTask`, **no reevalúa** `BatteryCritical?`. Continúa con `PerformTask` → `RUNNING`
3. Tick 5: La batería se vuelve crítica, pero `BatteryCritical?` no se reevalúa. La tarea continúa hasta completar
4. Problema: La condición crítica no se detecta hasta que `PerformTask` termina, arriesgando apagado inesperado

Comportamiento con ReactiveFallback:

1. Tick 1: Evalúa `BatteryCritical?` → `FAILURE` (batería OK). Continúa a `PerformTask` → `RUNNING`
2. Tick 2-4: **Reevalúa** `BatteryCritical?` → `FAILURE`. Continúa con `PerformTask` → `RUNNING`
3. Tick 5: **Reevalúa** `BatteryCritical?` → `SUCCESS` (batería crítica detectada)
4. El `ReactiveFallback` retorna `SUCCESS` inmediatamente, interrumpiendo `PerformTask`
5. Ventaja: El robot puede reaccionar inmediatamente para ir a cargar, evitando apagado inesperado

El `ReactiveFallback` monitoriza prioridades continuamente, que es de gran utilidad en sistemas donde pueden aparecer condiciones de alta prioridad que deben prevalecer sobre el comportamiento en curso.

Ejecución concurrente

Parallel Un nodo `Parallel` (símbolo: $\rightrightarrows$) ejecuta todos sus hijos simultáneamente en cada tick. Su política de terminación se configura mediante umbrales:

- Retorna `SUCCESS` cuando al menos M hijos retornan `SUCCESS`
- Retorna `FAILURE` cuando al menos N hijos retornan `FAILURE`
- Retorna `RUNNING` mientras no se alcancen los umbrales

Los casos comunes son:

- M = todos, $N = 1$: Todos deben tener éxito, falla al primer fallo (AND paralelo)
- $M = 1$, N = todos: Basta un éxito, falla solo si todos fallan (OR paralelo)

La figura 6.7 muestra un ejemplo de monitorización concurrente durante navegación.

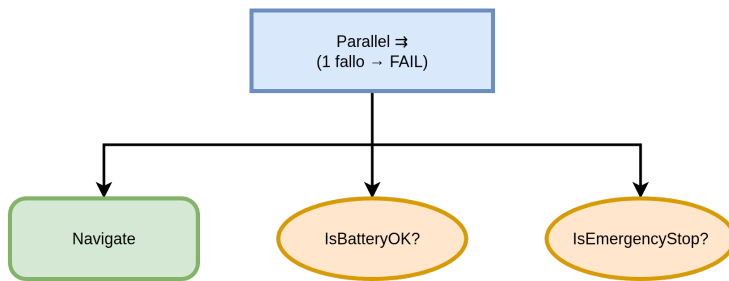


Figura 6.7: Nodo Parallel: ejecución concurrente con monitorización

La figura 6.7 ilustra un nodo `Parallel` raíz configurado con política de fallo temprano: basta que un hijo falle para que el nodo completo retorne `FAILURE`. El diagrama muestra tres hijos que se evalúan simultáneamente en cada tick:

- **Navigate:** Acción de larga duración que mueve el robot hacia el objetivo. Mientras está activa, retorna `RUNNING` en cada tick. Este es el comportamiento principal que queremos ejecutar.
- **IsBatteryOK?:** Condición que verifica continuamente que el nivel de batería sea suficiente. Retorna `SUCCESS` si la batería está bien, `FAILURE` si está críticamente baja. Al ser una condición, su evaluación es instantánea (nunca retorna `RUNNING`).
- **IsEmergencyStop?:** Condición que detecta si se ha activado un botón de parada de emergencia o si se ha recibido una señal de detención externa. Retorna `SUCCESS` si no hay emergencia, `FAILURE` si se detecta parada de emergencia.

El funcionamiento del nodo `Parallel` en este ejemplo es el siguiente: en cada tick, evalúa los tres hijos. Mientras `Navigate` esté en ejecución (`RUNNING`) y ambas condiciones de seguridad retornen `SUCCESS`, el nodo `Parallel` retorna `RUNNING`, permitiendo que la navegación continúe. Sin embargo, en el instante en que cualquiera de las condiciones de seguridad retorne `FAILURE`, el nodo `Parallel` retorna `FAILURE` inmediatamente, provocando que `Navigate` sea interrumpido (su método `onHalted()` es invocado). Este mecanismo garantiza que el robot se detenga instantáneamente ante situaciones de riesgo.

La política configurada (anotada como «1 fallo ⟶ FAIL» en el diagrama) especifica el umbral de fallo: $N = 1$. Esto significa que el `Parallel` no espera a que todos los hijos fallen, sino que reacciona ante el primer fallo. Esta configuración es típica para guardas de seguridad, donde cualquier condición adversa debe abortar la operación.

Caso de uso típico: Ejecutar una acción principal mientras se monitorizan condiciones de seguridad. Si una condición falla, se aborta la acción principal inmediatamente. Este patrón implementa «ejecución vigilada».

Importante: `Parallel` no implica verdadero paralelismo a nivel de hilos del sistema operativo. Todos los hijos se tickean secuencialmente dentro del mismo ciclo de evaluación. La «concurrencia» es lógica: el árbol no espera a que un hijo `RUNNING` complete antes de evaluar los demás.

Resumen comparativo de nodos de control

La tabla 6.2 resume el comportamiento de los nodos de control fundamentales (estándar y reactivos) según los resultados retornados por sus hijos.

Tabla 6.2: Comportamiento de nodos de control según resultados de sus hijos

Nodo	Retorna SUCCESS	Retorna FAILURE	Retorna RUNNING
`Sequence`	**Todos** los hijos retornan `SUCCESS`	Tan pronto como **un hijo** retorna `FAILURE`	El hijo actual retorna `RUNNING`
`Reactive Sequence`	**Todos** los hijos retornan `SUCCESS`	Tan pronto como **un hijo** retorna `FAILURE`	El hijo actual retorna `RUNNING`
`Fallback`	Tan pronto como **un hijo** retorna `SUCCESS`	**Todos** los hijos retornan `FAILURE`	El hijo actual retorna `RUNNING`
`Reactive Fallback`	Tan pronto como **un hijo** retorna `SUCCESS`	**Todos** los hijos retornan `FAILURE`	El hijo actual retorna `RUNNING`
`Parallel`	Al menos M hijos retornan `SUCCESS`	Al menos N hijos retornan `FAILURE`	No se han alcanzado los umbrales M o N

La tabla 6.3 resume las diferencias arquitectónicas entre nodos estándar y reactivos.

La elección entre nodos estándar y reactivos responde a un compromiso fundamental entre eficiencia y reactividad. Los nodos reactivos son apropiados cuando se trabaja con precondiciones que pueden invalidarse durante la ejecución, como el nivel de batería, la conectividad de red o la visibilidad de objetos. Su capacidad de reevaluación continua resulta esencial para monitorizar condiciones de seguridad tales como la detección de obstáculos o paradas de emergencia, y constituyen la elección natural en sistemas con prioridades dinámicas donde eventos de alta prioridad pueden necesitar interrumpir tareas en curso.

Por el contrario, los nodos estándar son preferibles en secuencias de acciones donde las condiciones iniciales se mantienen válidas durante toda la ejecución. Su ventaja radica en la optimización de rendimiento, ya que evitan reevaluaciones innecesarias de condiciones costosas. Estos nodos resultan adecuados para comportamientos donde la persistencia hasta completar la tarea es deseable, garantizando que no se interrumpan ante cambios en condiciones que ya fueron validadas inicialmente.

Los nodos reactivos pueden combinarse con `Parallel` para obtener lo mejor de ambos mundos: un `Parallel` con una acción principal y condiciones de guarda proporciona monitorización continua sin la sobrecarga de reevaluar toda la jerarquía en cada tick. La elección entre

Aspecto	Sequence/Fallback estándar	ReactiveSequence/Fallback
Memoria de estado	Recuerda qué hijo está `RUNNING`	No mantiene memoria, reevalúa desde el inicio
Reevaluación	Hijos anteriores **no** se reevalúan mientras hay uno `RUNNING`	**Todos** los hijos se reevalúan en cada tick
Reactividad	Baja: cambios en hijos anteriores ignorados	Alta: respuesta inmediata a cambios
Eficiencia	Alta: evalúa solo el hijo activo	Baja: reevalúa todos los hijos
Uso típico	Secuencias/alternativas sin cambios de contexto	Monitorización continua de condiciones

Tabla 6.3: Comparación entre nodos de control estándar y reactivos

nodos reactivos y `Parallel` depende de si las condiciones deben abortar la acción (`Parallel`) o simplemente fallar el nodo de control (`Reactive`).

Decoradores

Los decoradores envuelven un único hijo y modifican su comportamiento o resultado. Actúan como transformadores funcionales del flujo de control.

El conjunto de decoradores posibles es extensible y depende de la implementación concreta del motor BT. Aquí presentamos los decoradores más comunes y fundamentales, que aparecen en prácticamente todas las librerías BT. Otras implementaciones pueden incluir decoradores adicionales como `RepeatUntilFail`, `Delay`, `RateController`, o `Cooldown`, entre otros.

Inverter

El decorador `Inverter` (símbolo: ¬) invierte el resultado del hijo: `SUCCESS` se convierte en `FAILURE` y viceversa. `RUNNING` permanece sin cambios.

Caso de uso típico: Convertir condiciones afirmativas en negativas sin modificar el nodo original. Por ejemplo, transformar `IsBatteryLow?` en `!IsBatteryLow?` para expresar «si la batería *no* está baja, continúa».

Retry

El decorador `Retry` (símbolo: ↺) reintenta el hijo N veces si retorna `FAILURE`. Solo propaga `FAILURE` si se agotan todos los reintentos.

Caso de uso típico: Acciones que pueden fallar temporalmente por condiciones transitorias. Ejemplo: `GraspObject` puede fallar por ruido en los sensores; reintentarlo 3 veces aumenta la robustez sin complicar la lógica del nodo de acción.

ForceSuccess/ForceFailure

Estos decoradores reemplazan el resultado del hijo por un valor fijo. `ForceSuccess` siempre retorna `SUCCESS` (salvo si el hijo está `RUNNING`), independientemente del resultado real del hijo.

Caso de uso típico: Ejecutar una acción «de mejor esfuerzo» sin que su fallo afecte la secuencia principal. Ejemplo: en una secuencia de evacuación, queremos `SendAlert` pero no debemos abortar si el envío falla.

Timeout

El decorador `Timeout` (símbolo: ⊙) ejecuta el hijo pero lo aborta retornando `FAILURE` si no completa en un tiempo límite.

Caso de uso típico: Garantizar que acciones de larga duración no bloqueen indefinidamente el sistema. Ejemplo: una navegación con timeout de 5 minutos permite recuperarse si el robot queda atascado.

La figura 6.8 muestra cómo los decoradores enriquecen un árbol básico para añadir robustez.

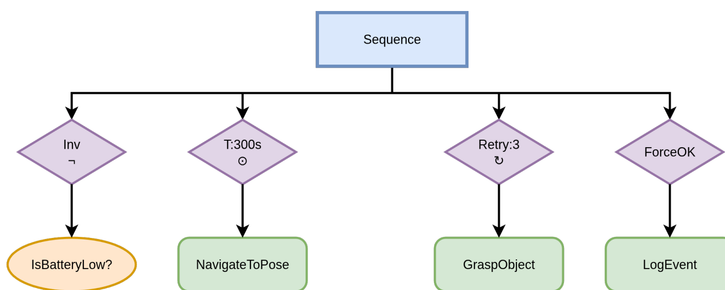


Figura 6.8: Decoradores añadiendo robustez a un árbol básico

En la figura 6.8, el `Sequence` combina cuatro operaciones protegidas: verifica que la batería *no* esté baja (invertida), navega con límite de 5 minutos, agarra con hasta 3 reintentos, y registra el evento (sin bloquear si falla). Esta composición ilustra cómo los decoradores permiten expresar políticas complejas de manejo de errores sin modificar las acciones subyacentes.

Modelado de misiones y tareas

Recordemos que el modelo arquitectónico misión–tarea–capacidad estructura el sistema robótico en tres niveles de abstracción: las *capacidades* son habilidades atómicas del robot (navegar, agarrar, detectar objetos); las *tareas* combinan capacidades para lograr objetivos específicos (transportar un objeto de A a B); y las *misiones* orquestan tareas para completar objetivos complejos de alto nivel (inspeccionar un almacén completo).

Los BT son lo suficientemente versátiles como para operar tanto a nivel de tarea como de misión. A nivel de *tarea*, un BT puede encapsular la lógica de coordinación de capacidades para resolver un objetivo concreto. Por ejemplo, una tarea `TransportObject` puede implementarse como un BT que coordina las capacidades `Navigate`, `Grasp` y `Release`, incluyendo manejo de errores y estrategias de recuperación. Este BT de tarea se convierte en una unidad reutilizable que puede invocarse desde niveles superiores.

A nivel de *misión*, los BT pueden actuar como orquestadores de alto nivel que coordinan múltiples tareas. Una misión de «inspección de planta industrial» podría modelarse como un `Sequence` raíz que ejecuta secuencialmente: `PlanRoute`, `CompleteNavigation`, `VisualInspection`, y `GenerateReport`. Si cada una de estas tareas está implementada como un BT independiente, estaríamos hablando de un *behavior forest*: una colección de BT especializados orquestados por un BT maestro.

Esta arquitectura compositiva admite esquemas híbridos que resultan especialmente efectivos: usar FSM a nivel de misión y BT a nivel de tarea. Las misiones típicamente se estructuran en fases de larga duración con transiciones bien definidas (preparación, ejecución, finalización), precisamente el dominio donde las FSM sobresalen. Cada fase puede permanecer activa durante periodos significativos (minutos u horas) y no requiere reactividad constante, solo transiciones claras cuando se completan ciertos objetivos o se detectan condiciones específicas.

Por el contrario, las tareas suelen requerir reactividad continua: deben responder inmediatamente a cambios en el entorno (obstáculos inesperados, pérdida de visibilidad del objeto), monitorizar múltiples condiciones simultáneamente (batería, seguridad, precondiciones), y coordinar capacidades de forma jerárquica. Este es precisamente el dominio donde los BT destacan. Un BT de tarea reevalúa continuamente su árbol, puede interrumpir acciones ante cambios de contexto, y expresa naturalmente estrategias de recuperación mediante nodos `Fallback`.

Esta combinación FSM-en-misión + BT-en-tarea constituye una de las mejores prácticas arquitectónicas en robótica moderna. La FSM de misión selecciona qué fase ejecutar y cuándo transitar (lógica de alto nivel, persistente), mientras que cada estado de la FSM activa un BT que implementa las tareas de esa fase con toda la reactividad necesaria. Por ejemplo, en una misión de logística, la FSM podría tener estados `PICKUP`, `TRANSPORT`, `DELIVER`, y dentro del estado `TRANSPORT`, un BT coordina navegación reactiva, detección de obstáculos, y replanning dinámico. La FSM permanece en `TRANSPORT` hasta que el BT señale éxito o fallo crítico, momento en el que la FSM decide la transición apropiada.

La clave arquitectónica está en la separación de responsabilidades: independientemente del nivel, un BT define el *qué* y el *cuándo* (qué acciones/tareas ejecutar y bajo qué condiciones), mientras que los niveles inferiores definen el *cómo* (la implementación técnica de capacidades). Esta separación, combinada con la capacidad de composición jerárquica, permite construir sistemas complejos de forma modular y mantenible.

Reutilización de comportamientos

Una de las principales ventajas de los BT es la facilidad para reutilizar comportamientos. Subárboles completos pueden encapsular patrones de comportamiento comunes y reutilizarse en distintas misiones o robots.

Esta reutilización no se limita al código, sino también al diseño del comportamiento. Los BT permiten construir bibliotecas de comportamientos que pueden combinarse y adaptarse a nuevos contextos con un esfuerzo reducido.

Arquitectónicamente, esta capacidad de reutilización favorece la escalabilidad del sistema y reduce la duplicación de lógica, alineándose con los objetivos de modularidad y mantenibilidad.

Comparación con FSM

Aunque las FSM y los BT persiguen el mismo objetivo general, modelar el comportamiento del robot, lo hacen desde perspectivas distintas. Las FSM se centran en estados persistentes y transiciones explícitas, mientras que los BT describen el comportamiento como un proceso continuo de evaluación de condiciones y acciones.

Las FSM resultan adecuadas para tareas bien delimitadas con pocos estados, pero se vuelven difíciles de gestionar cuando el número de combinaciones crece. Los BT, en cambio, favorecen la composición jerárquica y la reutilización de subestructuras.

FSM: simples pero poco escalables; BT: jerárquicos y reutilizables.

La tabla 6.4 resume las diferencias arquitectónicas fundamentales entre ambos modelos:

Tabla 6.4: Comparación arquitectónica entre FSM y Behavior Trees

Aspecto	**FSM**	**Behavior Tree**
Unidad básica	Estado persistente	Nodo de ejecución
Flujo de control	Transiciones entre estados	Evaluación recursiva del árbol
Reactividad	Mediante eventos/condiciones en transiciones	Reevaluación continua desde la raíz
Composición	Ortogonalidad (estados paralelos)	Jerarquía de subárboles
Escalabilidad	Explosión de estados en sistemas complejos	Crecimiento logarítmico por jerarquía
Persistencia	El estado persiste hasta transición explícita	Cada tick recalcula el flujo completo
Modularidad	Media (estados acoplados por transiciones)	Alta (subárboles independientes)

Arquitectónicamente, la elección entre FSM y BT no es excluyente. Una FSM es óptima cuando:

- El comportamiento tiene estados claramente diferenciados con persistencia temporal significativa
- Las transiciones están bien definidas y son relativamente escasas
- Se requiere control fino sobre cuándo y cómo ocurren las transiciones

Un BT es preferible cuando:

- El comportamiento se estructura naturalmente como una jerarquía de decisiones
- Se requiere alta reactividad ante cambios del entorno
- La reutilización y composición modular son prioritarias
- El sistema debe escalar a decenas o cientos de comportamientos distintos

En sistemas robóticos reales, es común encontrar FSM controlando comportamientos de bajo nivel (ej. control de un actuador) mientras que BT orquestan misiones de alto nivel que coordinan múltiples FSM.

FSM y BT pueden coexistir en sistemas robóticos reales.

6.2. Aplicación

Behavior Trees en ROS 2

En ROS 2, los BT pueden implementarse mediante librerías específicas que integran la evaluación del árbol con el modelo de ejecución basado en eventos. La librería de referencia es **BehaviorTree.CPP**, desarrollada por Davide Faconti, que proporciona una infraestructura robusta y extensible para implementar BT en C++.

BehaviorTree.CPP ofrece:

- Motor de ejecución eficiente con soporte para árboles grandes
- Mecanismo de puertos (*blackboard*) para compartir datos entre nodos
- Carga de árboles desde archivos XML para facilitar la iteración de diseño
- Sistema extensible para implementar nodos personalizados
- Herramientas de visualización (`Groot`) para depuración en tiempo real

BehaviorTree.CPP es agnóstico al framework de robótica, por lo que la integración con ROS 2 se realiza mediante la implementación de nodos personalizados. El patrón común consiste en pasar un nodo ROS 2 (`rclcpp::Node::SharedPtr`) a través del blackboard o del constructor de los nodos del BT, permitiendo que estos accedan a las interfaces de comunicación de ROS 2.

Desde un punto de vista arquitectónico, esta integración debe realizarse de forma que el árbol actúe como un orquestador de alto nivel, sin asumir responsabilidades propias de las capacidades de bajo nivel. Los nodos del árbol invocan acciones ROS 2, servicios o interactúan con topics, pero la lógica de negocio reside en los nodos independientes que implementan las capacidades.

Esta separación facilita la integración de los BT en sistemas existentes y su coexistencia con otros mecanismos de generación de comportamiento como FSM a nivel de actuadores individuales.

La figura 6.9 ilustra la arquitectura de BehaviorTree.CPP en el contexto de ROS 2.

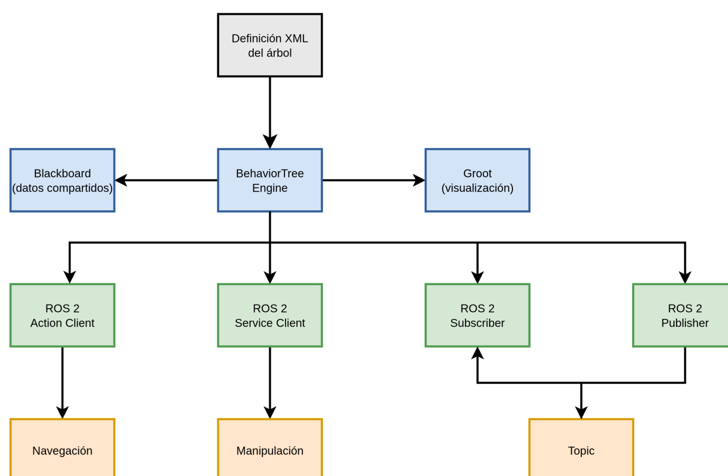


Figura 6.9: Arquitectura de BehaviorTree.CPP integrada con ROS 2

El motor BehaviorTree.CPP carga la definición del árbol (típicamente desde XML), proporciona un blackboard para compartir datos entre

nodos, y ofrece integración con Groot para visualización. Los nodos del árbol interactúan con capacidades ROS 2 mediante los mecanismos estándar: clientes de acciones para operaciones de larga duración, clientes de servicios para peticiones síncronas, publicadores y suscriptores.

Blackboard y puertos

Uno de los mecanismos fundamentales de BehaviorTree.CPP es el **blackboard**, un espacio de memoria compartida que permite la comunicación de datos entre nodos del árbol sin acoplarlos directamente. El blackboard actúa como un diccionario clave-valor tipado que persiste durante toda la ejecución del árbol. Los nodos acceden al blackboard mediante **puertos**, que son interfaces declarativas que especifican qué datos necesita leer un nodo (puertos de entrada) y qué datos puede escribir (puertos de salida).

Arquitectura del blackboard

El blackboard resuelve un problema arquitectónico crítico: ¿cómo comparten datos nodos que están en diferentes partes del árbol sin crear dependencias explícitas entre ellos? La solución es la indirección: los nodos no se comunican directamente, sino a través de un espacio compartido.

La figura 6.10 ilustra la arquitectura del blackboard y el sistema de puertos.

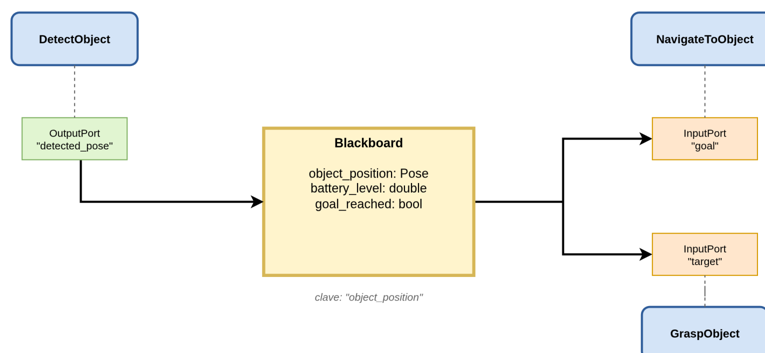


Figura 6.10: Arquitectura del blackboard y sistema de puertos. Los nodos declaran puertos de entrada (InputPort) y salida (OutputPort) que se conectan al blackboard mediante claves. Un nodo puede escribir un valor en el blackboard y otros nodos pueden leerlo, permitiendo comunicación sin acoplamiento directo. La figura muestra un ejemplo donde un nodo DetectObject escribe la posición del objeto detectado en el blackboard, y posteriormente los nodos NavigateToObject y GraspObject leen esa posición para ejecutar sus acciones.

Cada nodo del árbol declara estáticamente qué puertos proporciona mediante el método `providedPorts()`. Un puerto de entrada (`InputPort`) indica que el nodo necesita leer un dato del blackboard, mientras que un puerto de salida (`OutputPort`) señala que el nodo escribirá un dato. Existe también un tipo bidireccional (`BidirectionalPort`) que permite ambas operaciones.

Los puertos no se conectan directamente entre nodos, sino mediante claves del blackboard. Cuando se define el árbol (típicamente en XML), se especifica qué clave del blackboard se conecta a cada puerto. Esta indirección permite que múltiples nodos lean de la misma clave, y que cualquier nodo con un `OutputPort` pueda escribir en ella sin conocer quiénes serán sus consumidores.

El blackboard es tipado: cada entrada tiene un tipo asociado (números, cadenas de texto, posiciones, etc.). BehaviorTree.CPP valida en tiempo de construcción del árbol que los tipos coincidan. Si un nodo declara un puerto de entrada numérico y el XML conecta ese puerto a una

clave del blackboard que contiene texto, se genera un error en la fase de inicialización, antes de ejecutar el árbol.

Durante la ejecución, los nodos leen y escriben datos del blackboard mediante operaciones seguras respecto al tipo que fallan explícitamente si el puerto no está conectado o el tipo no coincide. Consideremos un escenario donde el nodo `DetectObject` ejecuta detección visual y escribe las coordenadas del objeto detectado en el blackboard usando su puerto de salida conectado a la clave `object_position`. Posteriormente, `NavigateToObject` lee esa misma clave mediante su puerto de entrada, obteniendo la posición sin conocer qué nodo la generó. Esta indirección permite reemplazar `DetectObject` por otro método de detección sin modificar `NavigateToObject`.

Este sistema de comunicación indirecta proporciona desacoplamiento estructural: los nodos no necesitan referencias directas entre sí. Un nodo que necesita una posición objetivo simplemente declara un puerto de entrada para ese dato, sin importar qué nodo lo proporciona. Esta característica facilita la reutilización, ya que un nodo bien diseñado con puertos claros puede emplearse en diferentes contextos. El mismo nodo `Navigate` puede usarse con objetivos provenientes de detección visual, entrada del usuario, o planificación, sin modificación alguna.

La composición se vuelve flexible: los árboles pueden reconfigurarse cambiando las conexiones en el XML sin recompilar código. Si queremos que `NavigateToObject` use una posición alternativa, solo cambiamos la clave del blackboard a la que se conecta su puerto. El blackboard también proporciona un punto de observación único para todos los datos compartidos, lo que facilita la depuración. Herramientas como Groot pueden inspeccionar el estado del blackboard en tiempo real, mostrando qué valores están disponibles y cómo fluyen entre nodos.

Finalmente, la validación temprana es posible porque los puertos se declaran estáticamente. El motor puede verificar antes de ejecutar que todos los puertos necesarios están conectados y que los tipos coinciden, detectando errores de configuración en la fase de inicialización en lugar de descubrirlos durante la ejecución.

Ejemplo ilustrativo e integración con capacidades

Para consolidar los conceptos teóricos presentados, resulta útil visualizar cómo se estructuraría un behavior tree completo que combine navegación con reactividad ante eventos del entorno. El comportamiento clásico de *bump-and-go* con recuperación ante obstáculos ilustra perfectamente las ventajas arquitectónicas de los BT frente a FSM: expresión jerárquica de prioridades, recuperación automática ante fallos, composición modular de comportamientos reutilizables, e integración natural con capacidades externas.

La figura 6.11 muestra la estructura completa del comportamiento. En este escenario, el robot debe avanzar hacia un objetivo mientras monitoriza su sensor láser. Si detecta un obstáculo cercano (umbral típico: 0.5 metros), debe ejecutar inmediatamente una maniobra de recuperación: retroceder para alejarse y girar para cambiar la orientación, antes de reintentar el avance.

Un árbol BT que implementa este comportamiento se estructuraría con un `Fallback` raíz que establece la política de control: intenta primero gestionar obstáculos cercanos (prioridad de seguridad), luego ejecutar la

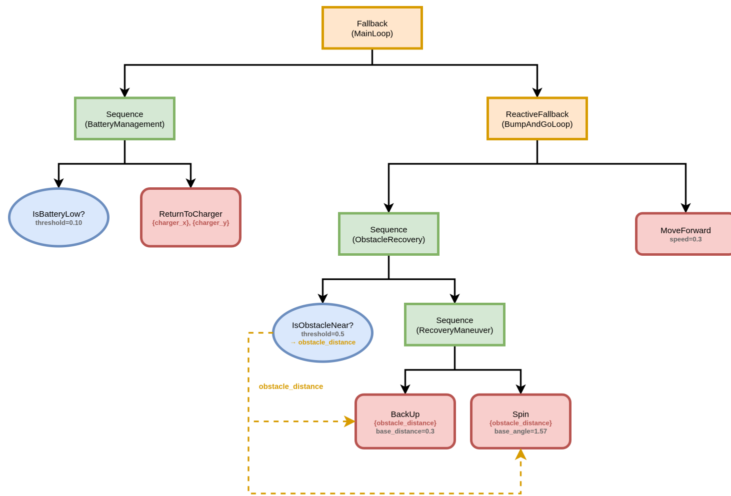


Figura 6.11: Behavior Tree para bump-and-go integrando control directo con capacidades Nav2

navegación (objetivo principal), y finalmente abortar si todo falla (último recurso). Este patrón expresa explícitamente que la seguridad tiene prioridad sobre el objetivo de navegación. La gestión de obstáculos se implementa mediante un `Sequence` que verifica si hay un obstáculo cercano y, de ser así, ejecuta la recuperación (`BackUp` seguida de `Spin`). La navegación estaría protegida con un decorador `Retry(3)` y un `Timeout(120s)` para robustez.

Comunicación entre nodos mediante puertos Este ejemplo ilustra tres patrones de comunicación mediante el blackboard:

1. **Datos globales de misión**: El programa principal establece `goal_x`, `goal_y` y `goal_theta` en el blackboard global. El nodo `MoveTowardsGoal` lee estos valores mediante puertos de entrada, accediendo a datos compartidos establecidos externamente.
2. **Conexión directa entre nodos**: El nodo `IsObstacleNear` detecta la distancia mínima al obstáculo más cercano y escribe este valor en el puerto de salida `obstacle_distance`. Esta información fluye directamente hacia los nodos `BackUp` y `Spin`, que la utilizan para parametrizar su comportamiento de recuperación. Si el obstáculo está muy cerca (<0.3m), `Spin` gira 180°; si está más lejos, solo 90°. `BackUp` retrocede proporcionalmente a la distancia detectada.
3. **Parámetros con valores por defecto**: Los nodos `BackUp`, `Spin` e `IsObstacleNear` declaran parámetros configurables (`base_distance`, `base_angle`, `threshold`) con valores por defecto razonables, permitiendo ajustar el comportamiento desde el XML sin modificar el código.

Esta arquitectura de comunicación demuestra cómo el blackboard permite que los nodos produzcan información que otros consumen, creando un flujo de datos desde sensores (`IsObstacleNear`) hacia actuadores (`BackUp`, `Spin`), mientras que los datos de misión global (`goal_x/y`) permanecen accesibles para todos los nodos que los necesiten.

Comparemos con una FSM equivalente: necesitaríamos estados explícitos para `IDLE`, `NAVIGATING`, `OBSTACLE_DETECTED`, `BACKING_UP`, `SPINNING`, `RETRY_1`, `RETRY_2`, `RETRY_3`, `ABORTING`, y transiciones complejas entre todos ellos, además de variables globales para compartir la distancia del obstáculo. El BT expresa el mismo comportamiento de forma mucho más compacta y legible, con comunicación explícita mediante puertos.

Integración con capacidades externas El ejemplo anterior muestra nodos simples de control directo (`BackUp`, `Spin`), pero en sistemas robóticos reales, las capacidades se suelen implementar como acciones o servicios ROS 2 que encapsulan tareas complejas. Como se ilustra en la figura 6.11, el nodo `MoveTowardsGoal` puede conectarse con el sistema de navegación Nav2, que proporciona el servidor de acción `/navigate_to_pose` implementando navegación global robusta con planificación A* o Theta*, control DWB, y capas de costmap para evitación de obstáculos.

El patrón arquitectónico consiste en que los nodos del BT actúen como clientes de estas acciones o servicios, delegando la ejecución mientras supervisan el progreso y deciden cuándo activar, cancelar o reintentar. Este patrón establece una separación arquitectónica clara aplicable a cualquier capacidad compleja: el nodo BT decide *cuándo* invocar la capacidad y *qué hacer* con el resultado (reintentar, abortar, ejecutar recuperación), mientras que el servidor de acción decide *cómo* ejecutar la tarea. El nodo actúa como adaptador entre el árbol BT (evaluación síncrona con estados discretos) y las acciones ROS 2 (ejecución asíncrona con feedback continuo).

La arquitectura asíncrona es fundamental: el método `onStart()` envía el objetivo y retorna `RUNNING` inmediatamente sin bloquear, el servidor de acción ejecuta en paralelo, y `onRunning()` consulta periódicamente si ha completado. Este diseño basado en polling garantiza que el árbol mantenga control sobre la frecuencia de evaluación, permitiendo que otros nodos se evalúen mientras la capacidad progresa.

La gestión de preemption permite que el BT cancele la ejecución si una condición de mayor prioridad lo requiere. Por ejemplo, si el árbol detecta batería baja, puede interrumpir la capacidad en curso y ejecutar un comportamiento de retorno a estación de carga. Este mecanismo es genérico y aplicable a cualquier servidor de acción: navegación (Nav2), manipulación (MoveIt 2), percepción (detección de objetos), interacción humano-robot (diálogo), etc.

El BT proporciona supervisión, recuperación ante fallos, y orquestación de múltiples capacidades en comportamientos complejos, mientras cada capacidad encapsula su implementación específica. El BT orquesta, las capacidades ejecutan.

7 Análisis de arquitecturas de referencia y Deep ROS 2

7.1 Arquitecturas de referencia en ROS 2 97
Nav2 (Navigation2) 97
PlanSys2 101
EasyNav (EasyNavigation) 105
7.2 Síntesis de criterios arquitectónicos transversales . 108
Patrones arquitectónicos y antipatrones 108
Cómo evaluar una arquitectura robótica 109
Testing, validación y depuración arquitectónica . . . 110
Gestión de errores y tolerancia a fallos 111
Arquitecturas distribuidas reales 111
Seguridad y *safety* 112
Diseño de interfaces y contratos 113
Configuración y despliegue 113
Arquitectura y sistema operativo 114
Ingeniería y evaluación comparativa de arquitecturas robóticas 114
7.3 Deep ROS 2 115
Diseño de ROS 2 115
Gestión de ejecución en ROS 2 117
Executors en práctica: un ejemplo de *starvation* . . . 119
Callback groups 122
Tiempo real en ROS 2: conceptos y latencias . . . 125
Planificador del sistema operativo (Linux) y `SCHED_FIFO` 126
Estrategias de tiempo real en ROS 2 con executors y callback groups 127

7.1. Arquitecturas de referencia en ROS 2

En los capítulos anteriores hemos discutido conceptos arquitectónicos (capas misión–tarea–capacidad, modelos del mundo, máquinas de estados, Behavior Trees, contratos de interacción, y razonamiento temporal). En ROS 2, muchos de esos conceptos aparecen cristalizados en *frameworks* que resuelven problemas recurrentes de forma reutilizable: navegación, planificación deliberativa o control en tiempo real.

Los frameworks de ROS 2 concretan patrones arquitectónicos recurrentes.

En esta sección revisamos tres arquitecturas de referencia ampliamente usadas en el ecosistema: Nav2 (navegación), PlanSys2 (planificación a nivel de misión) y EasyNav (navegación en un único proceso con composición y ciclos diferenciados). El objetivo no es detallar APIs, sino entender qué decisiones de diseño toman y cómo encajan con los conceptos ya vistos.

El objetivo es leer los frameworks como decisiones de arquitectura.

Nav2 (Navigation2)

Nav2 es el *stack* de navegación de referencia en ROS 2 y, en nuestro esquema misión–tarea–capacidad, se sitúa principalmente en el nivel de capacidad o *skill*. Fue uno de los grandes *frameworks* desarrollados específicamente para ROS 2, por lo que ha servido como banco de pruebas y adopción temprana de muchos de los mecanismos que ROS 2 ha ido incorporando. Su arquitectura separa responsabilidades en servidores especializados (planificación global, control local, comportamientos de recuperación, suavizado, etc.) y utiliza un *Behavior Tree* (BT) como mecanismo de orquestación para la navegación de alto nivel. La documentación oficial describe el diseño general y sus componentes principales en `https://docs.nav2.org/`.

Nav2 es la referencia de navegación y se sitúa en la capa de capacidad.

Visión general y componentes. La figura 7.1 resume la arquitectura típica. En el centro aparece el *BT Navigator*, que ejecuta un árbol de comportamiento para resolver una petición de navegación (por ejemplo, ir a una pose o seguir una ruta). Las hojas del BT suelen invocar acciones (`NavigateToPose`, `FollowWaypoints`, etc.). La figura 7.2 muestra un ejemplo simple del árbol (BT) que ejecuta el *BT Navigator*, donde se aprecia la

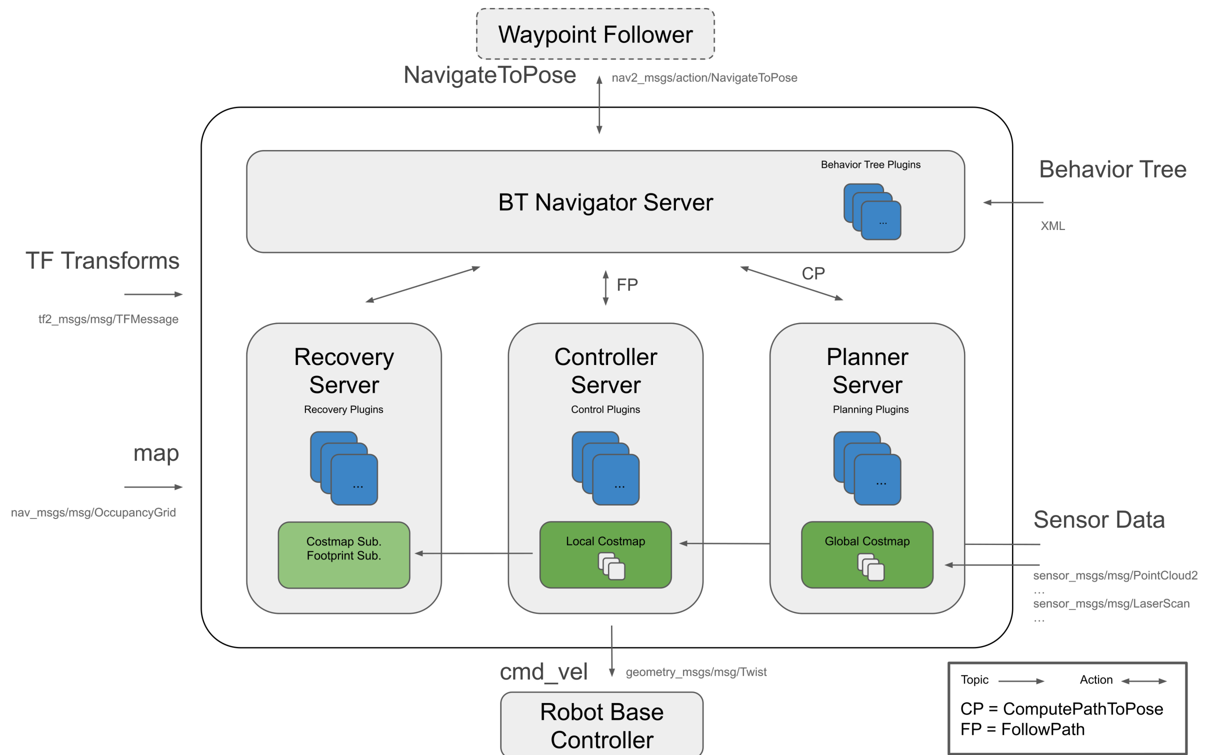


Figura 7.1: Arquitectura de Nav2. Fuente: [2].

estructura de navegación con replanificación y recuperaciones. Las hojas del BT suelen invocar acciones que delegan en servidores específicos:

- **Planner Server:** calcula planes globales en el mapa (frecuentemente sobre un *costmap*) usando plugins de planificación.
- **Controller Server:** convierte el plan global en comandos de velocidad, ejecutando un bucle de control local a una frecuencia configurada.
- **Recovery Server:** encapsula comportamientos de recuperación (p.ej., girar, retroceder, limpiar costmap) también como plugins.
- **Costmaps (global y local):** mantienen representaciones espaciales por capas (obstáculos, inflado, coste) que alimentan al planificador y al controlador.

Todos estos componentes se integran mediante los mecanismos básicos de ROS 2 (nodos, topics, servicios y especialmente *acciones*) y hacen un uso intensivo de *plugins* (`pluginlib`) para permitir sustituir algoritmos sin modificar el *framework*. Arquitectónicamente, Nav2 adopta con bastante pureza el estilo distribuido que promueve ROS 2: muchos nodos especializados, cada uno con sus propias interfaces, estados y ciclo de vida.

Nav2 materializa una arquitectura distribuida y basada en plugins.

Puntos de extensión y variabilidad (plugins). Nav2 se apoya en plugins para capturar la variabilidad algorítmica sin romper la arquitectura. En un despliegue típico, se configura (más que se programa) el sistema eligiendo implementaciones concretas en:

- **Planificación global:** plugins de planificador (por ejemplo, variantes de búsqueda o planificadores basados en rejilla).
- **Control local:** plugins de controlador (modelos cinemáticos distintos, controladores tipo *pure pursuit*, DWB, etc.).

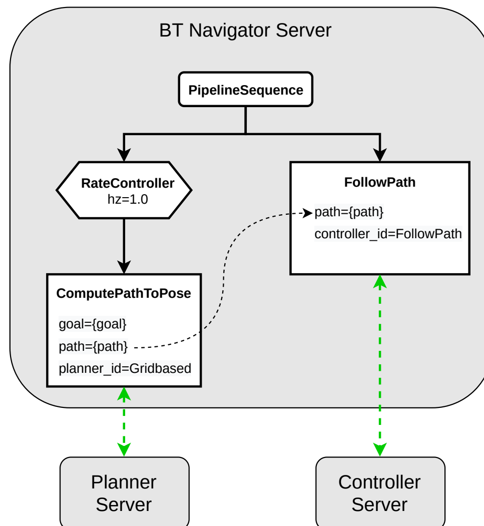


Figura 7.2: Ejemplo de *Behavior Tree* (BT) simple usado por el *BT Navigator* de Nav2.

- **Costmap:** plugins de capas (obstáculos, inflado, coste social, zonas prohibidas, etc.).
- **Comportamientos de recuperación:** plugins de comportamientos (rotar, retroceder, limpiar capas, etc.).
- **BT Navigator:** el propio árbol (y sus nodos) es intercambiable y actúa como pegamento entre capacidades.

Esto ilustra un principio arquitectónico clave: definir interfaces estables para componentes variables (capítulo 1).

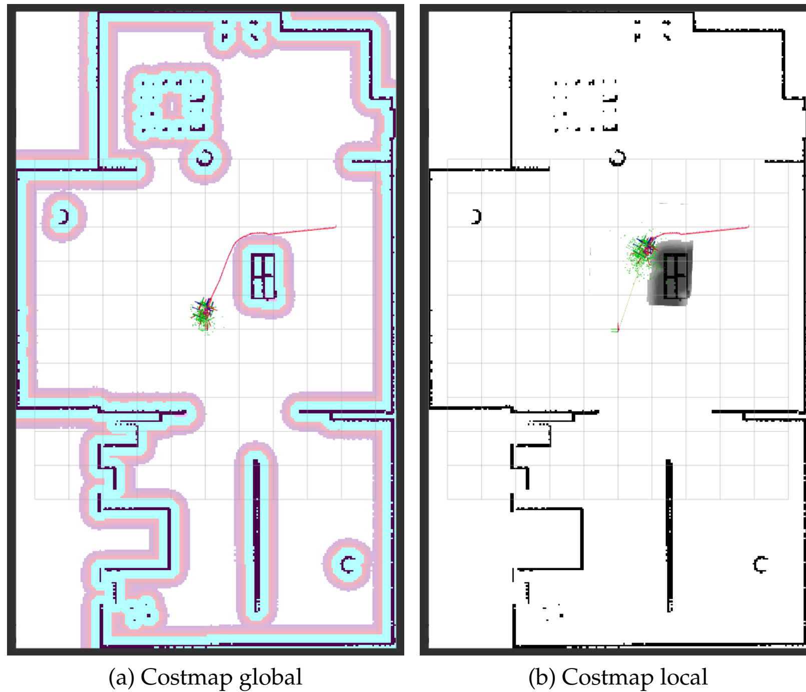
Composición, ejecutores y control de concurrencia. Aunque Nav2 puede componerse en un proceso para reducir sobrecarga y simplificar despliegue, su forma más reconocible es la de un sistema distribuido en múltiples nodos. Esto sigue bien los patrones de diseño de ROS 2, pero también introduce un coste apreciable: cada nodo levanta topics, servicios, acciones y tráfico de descubrimiento/monitorización que termina cargando el RMW/DDS incluso cuando la funcionalidad de aplicación apenas ha cambiado. En sistemas grandes o redes limitadas, parte de la sobrecarga proviene simplemente de mantener vivo el ecosistema distribuido.

La distribución mejora modularidad, pero añade sobrecarga en middleware y red.

Esta decisión vuelve a poner en primer plano lo discutido en Deep ROS: la elección de *executors*, *callback groups* y la separación de hilos afecta al aislamiento de cargas (costmap, planificación, control) y al cumplimiento de frecuencias. Arquitectónicamente, lo importante es que la concurrencia sea una decisión explícita y verificable, no un efecto colateral.

Integración con localización, mapas y sensores. Nav2 se apoya en fuentes externas para el mapa y la localización (p.ej., servidores de mapa, AMCL, SLAM), que alimentan tanto a TF como a los costmaps. Aquí aparece una conexión directa con el capítulo de percepción y representación espacial (capítulo 3): una navegación robusta requiere definir bien marcos de referencia, sincronización temporal y calidad de la estimación. Desde el punto de vista de arquitectura, Nav2 delimita claramente qué necesita del mundo (costmaps + TF) y deja el cómo de estimarlo a componentes especializados.

Nav2 depende de una representación espacial coherente basada en costmaps y TF.



(a) Costmap global (b) Costmap local

Figura 7.3: Ejemplo de costmap global y local en Nav2 (representaciones espaciales por capas usadas por el planificador global y el controlador local).

Relación con conceptos previos: BTs, capas y modelo del mundo. Nav2 ejemplifica muy bien las ideas del capítulo de *Behavior Trees* (capítulo 6). El BT actúa como un programa de alto nivel donde cada hoja representa una capacidad (planificar, controlar, comprobar progreso, ejecutar recuperación). El uso del BT aporta (i) trazabilidad del flujo de decisión, (ii) reactividad (replanificación y recuperaciones ante fallos), y (iii) modularidad: el árbol puede reconfigurarse sin tocar los servidores.

Desde el punto de vista de capas (capítulo 4), Nav2 suele situarse como una capacidad reutilizable de navegación sobre la que capas superiores construyen tareas y misiones. Su memoria o representación interna del mundo emerge principalmente de los *costmaps* y de las transformaciones (`TF`): la navegación depende de tener una representación espacial coherente (capítulo 3). En la práctica, cualquier conocimiento del entorno que Nav2 deba usar para navegar acaba codificándose, de un modo u otro, en el costmap: se parte típicamente de un mapa estático y se le añaden capas de obstáculos, inflado, zonas prohibidas u otros costes. El propio costmap es así un ejemplo de diseño por *capas* donde cada fuente de información añade una contribución al coste.

El costmap actúa como memoria espacial por capas de la navegación.

Contratos de interacción y operabilidad. En términos de comunicación (capítulo 2), Nav2 descansa de manera muy fuerte sobre ROS 2 actions. Muchos de los nodos del BT Navigator se comunican con el *planner*, el *controller* o los comportamientos de recuperación mediante acciones, lo que conceptualmente es correcto porque expresa bien objetivos, feedback, resultado y *preemption*. Sin embargo, en la práctica esto introduce un *overhead* de comunicación nada despreciable y, sobre todo, una pérdida de determinismo: la entrega efectiva de mensajes y callbacks queda en manos de los *executors* y del *runtime*, lo que puede convertirse en un punto débil cuando la CPU está muy cargada.

Las actions clarifican contratos, pero introducen sobrecarga y menos determinismo.

Nav2 también utiliza *Lifecycle Nodes*: cada servidor tiene estados gestionados (configurar, activar, desactivar, limpiar), lo que mejora la operabilidad del sistema y facilita construir secuencias de arranque controladas

(capítulo 4). Como contrapartida, la configuración global del sistema, fuertemente apoyada en ficheros YAML, es potente pero compleja: buena parte de la dificultad real de Nav2 está menos en programarlo que en parametrizar de forma coherente todos sus componentes.

El lifecycle mejora operabilidad, pero la configuración global sigue siendo compleja.

Aspectos temporales. La navegación combina procesos con frecuencias distintas: el controlador local suele ejecutarse a decenas de Hz, la actualización de costmaps puede ser diferente, y el BT se tickea a una frecuencia propia. Esta separación ilustra el principio percepción–decisión–acción y hace explícita la necesidad de razonar sobre latencias y sincronización (capítulo 7). En la práctica, el componente más claramente consciente de requisitos de tiempo real es el controlador; el resto del sistema prioriza modularidad y flexibilidad sobre determinismo estricto.

Solo parte de Nav2 está pensada con disciplina de tiempo real estricta.

Supervisión y recuperación como arquitectura, no como *parches*. Un punto fuerte de Nav2 es que los mecanismos de supervisión (comprobar progreso, detectar bloqueos, validar que se sigue el plan) y de recuperación (limpiar costmaps, girar, retroceder, replanificar) se modelan explícitamente en el BT. Esto encaja con la idea de *control supervisor* basada en BTs (capítulo 6): ante condiciones observables, el árbol activa ramas alternativas sin necesidad de codificar lógica de control de flujo dispersa en callbacks.

Observabilidad y depuración. Al estar la lógica de alto nivel concentrada en un BT y las capacidades encapsuladas en servidores, el sistema se presta a instrumentación: se puede inspeccionar el estado de los servidores, el árbol activo, el resultado de chequeos y el estado de costmaps. Esto es relevante porque en sistemas reales los fallos suelen ser *emergentes* y dependen de interacciones (capítulo 7); disponer de puntos claros de observación mejora el mantenimiento.

Al mismo tiempo, es razonable cuestionar hasta qué punto conviene implementar comportamientos cada vez más complejos dentro del BT Navigator o exponer ese árbol al usuario final como lugar natural donde resolver cualquier problema. Muchas veces esos comportamientos pertenecen más bien a capas superiores de misión o supervisión, dejando a Nav2 centrado en la capacidad de navegar con robustez.

PlanSys2

PlanSys2 es un *framework* de planificación deliberativa para ROS 2 basado en PDDL (Planning Domain Definition Language). En nuestro esquema misión–tarea–capacidad, ocupa claramente el nivel de misión en lo que respecta a planificación y control deliberativo, mientras que las acciones PDDL que ejecuta se corresponden con tareas. Nació tomando a Nav2 como una referencia importante en cuanto a organización de componentes para ROS 2, pero trasladando el foco desde la navegación hacia la deliberación simbólica. La descripción y diagramas de referencia están disponibles en `https://plansys2.github.io/`.

PlanSys2 lleva la deliberación simbólica a la capa de misión en ROS 2.

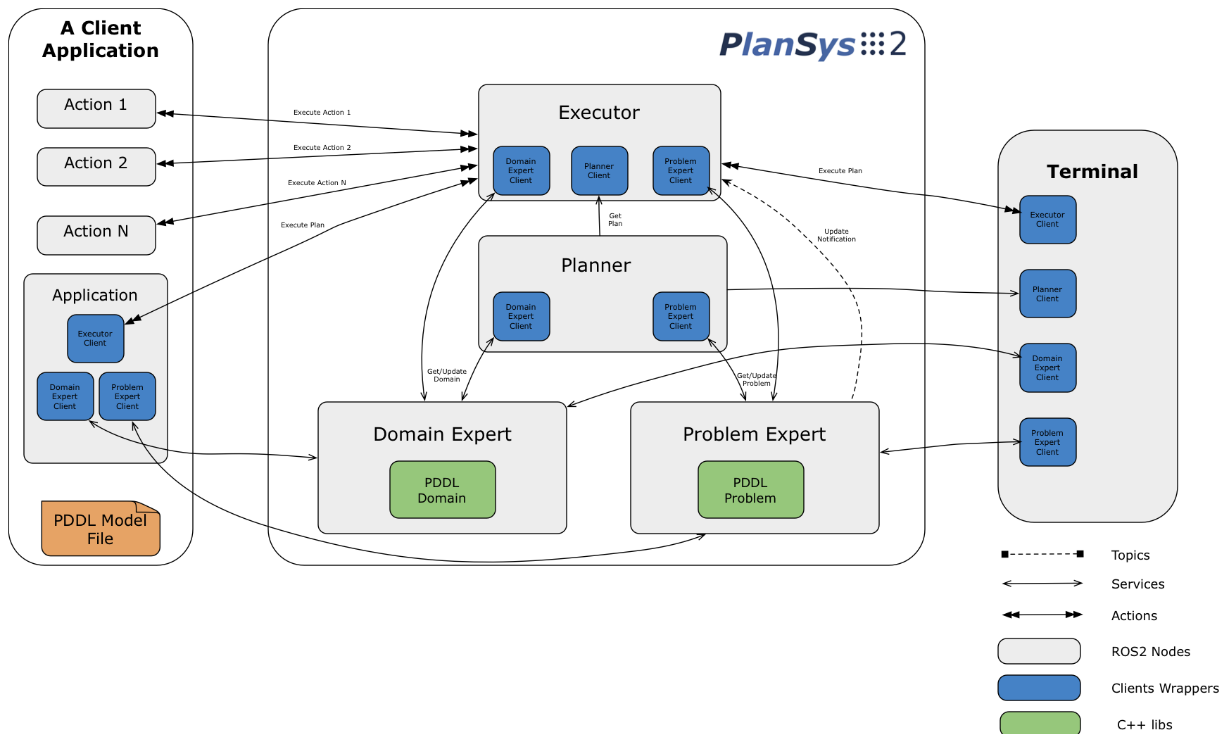


Figura 7.4: Arquitectura general de PlanSys2. Fuente: `https://plansys2.github.io/_images/plansys2_arch.png`

Arquitectura: conocimiento, planificación y ejecución. PlanSys2 separa responsabilidades en cuatro nodos principales:

- **Domain Expert:** mantiene el dominio PDDL (tipos, predicados, acciones y sus precondiciones/efectos).
- **Problem Expert:** mantiene el problema actual (objetos, estado inicial, metas), es decir, el *estado del mundo simbólico*.
- **Planner:** genera un plan (secuencia/estructura temporal de acciones) a partir del dominio, el estado y la meta. Suele ofrecer plugins para distintos planificadores.
- **Executor:** ejecuta el plan y supervisa su avance, integrándolo con acciones ROS 2 en la capa inferior.

La figura 7.4 ilustra esta separación, que encaja directamente con la idea de arquitectura deliberativa por capas (capítulo 4): (i) una capa de misión define metas, (ii) la planificación calcula una secuencia de tareas, y (iii) la ejecución conecta con capacidades reales. Igual que Nav2, el sistema se organiza en componentes bien aislados y, en su implementación, esos componentes se despliegan como *Lifecycle Nodes*, lo que ayuda a controlar inicialización, activación y parada.

PlanSys2 separa misión, planificación y ejecución en componentes aislados.

La documentación también ofrece una vista alternativa de la arquitectura y su interacción con el ecosistema ROS 2, mostrada en la figura 7.5.

Modelado del dominio: separar qué se quiere de cómo se implementa. El modelado en PDDL fuerza una disciplina de diseño: las acciones definen precondiciones y efectos sobre un conjunto de predicados, lo que convierte el *estado del mundo* en un artefacto explícito. Arquitectónicamente, esto es una forma de *contrato* entre la capa de misión y las capacidades: una capacidad no se describe por su API concreta, sino por cómo transforma el estado simbólico. Esto ayuda a mantener independencia entre el razonamiento deliberativo y la implementación en

PDDL convierte el estado simbólico en el contrato entre misión y capacidades.

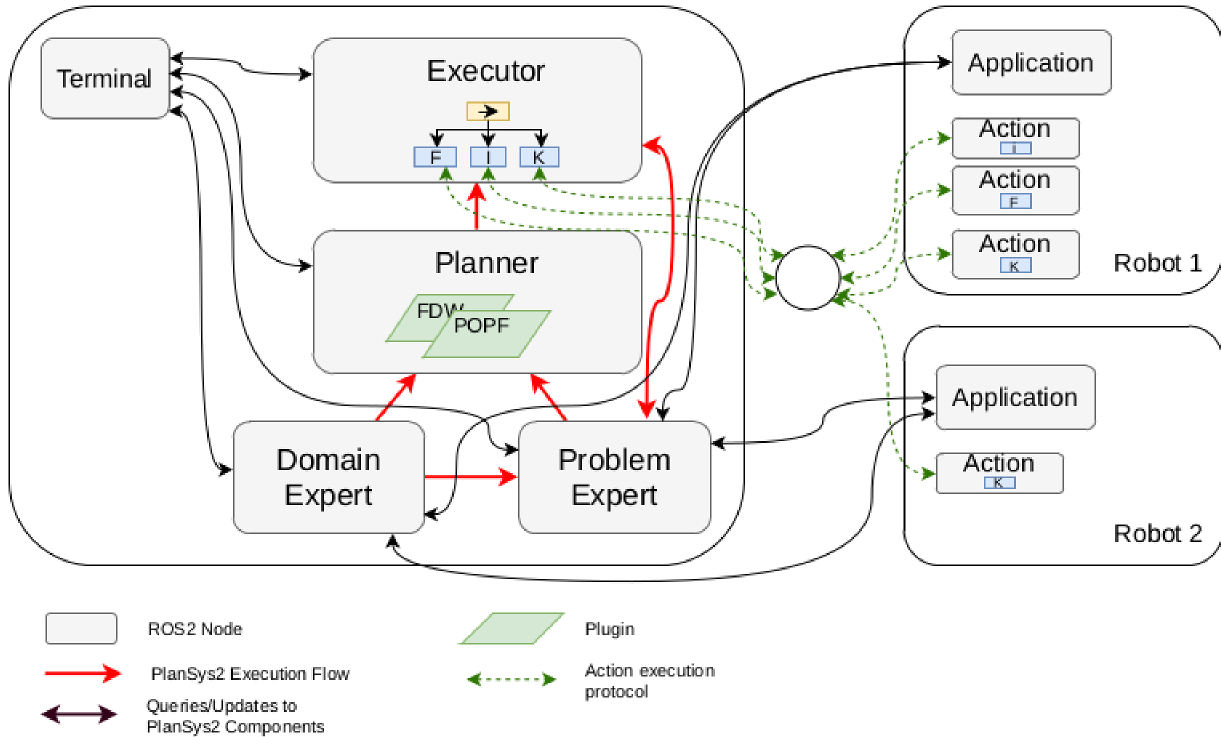


Figura 7.5: Vista ampliada de la arquitectura de PlanSys2. Fuente: https://plansys2.github.io/_images/plansys2_arch2.png

ROS 2. Esa memoria es esencialmente simbólica y está materializada en el *Problem Expert*, que mantiene los hechos del problema activo.

De un plan a un controlador reactivo: PlanSys2 y BTs. Un aspecto especialmente relevante (y didáctico) es que PlanSys2 usa *Behavior Trees* de dos maneras. Por un lado, el *Executor* no ejecuta el plan como una lista estrictamente secuencial de llamadas: construye una estructura de ejecución basada en *Behavior Trees*, donde cada acción del plan se representa como un nodo del BT y la lógica de control refleja dependencias, secuencias y paralelismos. La figura 7.6 muestra las hojas de este árbol. Por otro lado, las propias acciones pueden implementarse internamente como BTs cuando interesa encapsular comportamiento reactivo dentro de una tarea.

PlanSys2 usa BTs tanto para ejecutar planes como para implementar tareas.

Esta decisión conecta explícitamente con el capítulo de BTs (capítulo 6): la ejecución del plan se beneficia de reactividad, supervisión y capacidad de recuperación. En una arquitectura real, el planificador proporciona intención (qué hacer) y el BT proporciona política de ejecución (cómo hacerlo ante eventos y fallos).

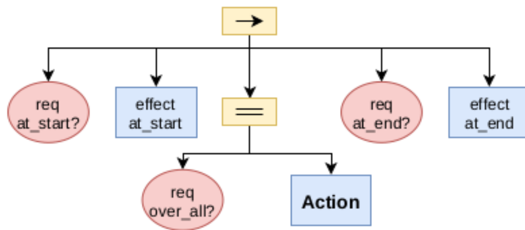


Figura 7.6: Ejemplo de árbol de comportamiento asociado a una acción/plan en PlanSys2. Fuente: https://plansys2.github.io/_images/action_bt.png

Contratos entre capa deliberativa y capacidades. PlanSys2 define un protocolo para que las acciones simbólicas (del dominio PDDL)

se materialicen en *performers* ROS 2 capaces de ejecutarlas. Un punto interesante es que esta interacción no se articula mediante ROS 2 actions, sino a través de un topic bidireccional usado por el *Executor* y las acciones para solicitar ejecución, aceptar o rechazar trabajos, enviar feedback y comunicar éxito o fallo (Fig. 7.7). Arquitectónicamente, es una alternativa muy razonable a las *actions*: en este contexto aporta más flexibilidad y reduce parte de la rigidez del contrato estándar.

PlanSys2 usa un protocolo propio más flexible que las actions estándar.

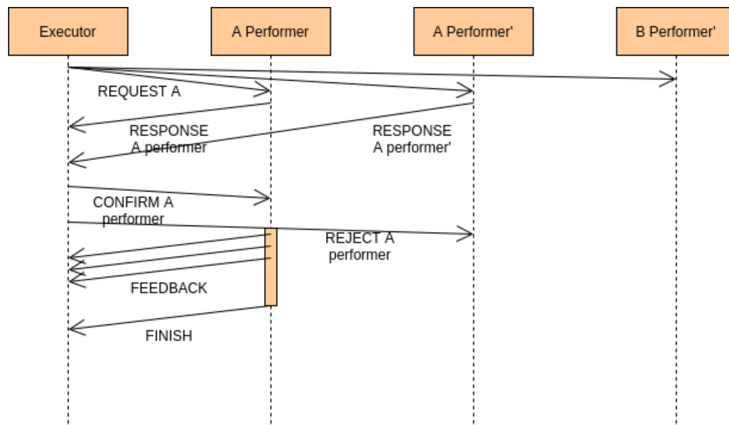


Figura 7.7: Protocolo de ejecución y comunicación de acciones en PlanSys2. Fuente: `https://plansys2.github.io/_images/protocol.png`

Además, PlanSys2 proporciona *proxies* en C++ y Python para acceder a la funcionalidad de cada componente desde fuera. Esto simplifica mucho el trabajo del programador, porque lo aísla de la complejidad de las comunicaciones ROS 2 internas y le permite pensar en términos de operaciones de alto nivel sobre dominio, problema, planificación y ejecución.

Gestión de fallos, monitorización y *replanning*. Una arquitectura deliberativa no es solo planificar: también necesita monitorización y adaptación. PlanSys2 aporta mecanismos para detectar fallos de ejecución (acciones que no llegan a éxito, condiciones que dejan de cumplirse) y reaccionar: reintentar, ejecutar recuperaciones o volver a planificar con un estado actualizado. Este patrón conecta con lo visto en máquinas de estados (capítulo 5) y BTs (capítulo 6): el sistema de misión suele combinar estructuras deliberativas (plan) con mecanismos reactivos (árbol/recuperaciones) para mantener robustez.

La robustez deliberativa exige supervisión y capacidad de replanning.

Aspectos temporales y sincronización con el mundo real. En la práctica, las acciones tienen duraciones, deadlines y dependencias temporales, y el estado del mundo cambia de forma asíncrona por percepción. El reto arquitectónico es que el modelo simbólico no se quede obsoleto: la actualización del Problem Expert debe conectarse con sensores y estimadores (capítulo 3) y con mecanismos de concurrencia seguros (capítulo 7).

Integración con capacidades complejas (por ejemplo, navegación). Un patrón común es implementar un *performer* de PlanSys2 que encapsule una capacidad existente, como navegar con Nav2. Desde la perspectiva de la capa de misión, la acción simbólica puede ser algo como `move(robot, wp)`; el performer traduce ese objetivo a una acción ROS 2 concreta (por ejemplo, `NavigateToPose`) y reporta feedback/resultados al Executor. Esto ilustra cómo se conectan, en una arquitectura por capas,

Los performers conectan misión deliberativa con capacidades concretas reutilizables.

un plan deliberativo (capítulo 4) con capacidades reactivas y temporizadas (capítulo 7) sin que el planificador conozca detalles del controlador local.

Deliberación + reactividad: una combinación pragmática. En robótica, la deliberación pura es frágil: el mundo cambia y las suposiciones se rompen. PlanSys2 muestra una solución pragmática: usar planificación para decidir una estrategia global, y BTs/acciones para ejecutar de forma reactiva, detectando desviaciones y adaptándose. Esta combinación conecta con el motivo por el que en arquitecturas reales aparecen múltiples modelos de control (FSM, BT, planificador) a distintos niveles (capítulos 5 y 6).

Modelo del mundo y mantenimiento de consistencia. El *Problem Expert* actúa como un modelo del mundo simbólico: predicados que describen hechos relevantes para el planificador. Este componente ejemplifica la distinción entre (i) representaciones métricas/geométricas (capítulo 3) y (ii) representaciones simbólicas usadas para deliberación (capítulo 4). Un reto arquitectónico es mantener coherencia entre ambos niveles: por ejemplo, que el predicado `at(robot, roomA)` se actualice por percepción/localización.

El Problem Expert mantiene la memoria simbólica que usa la deliberación.

EasyNav (EasyNavigation)

EasyNav (EasyNavigation) es un *framework* de navegación para ROS 2 que, igual que Nav2, se sitúa en el nivel de capacidad o *skill*. Su objetivo es ofrecer una alternativa a Nav2 en la que la representación del entorno no quede incrustada en el propio *framework*, permitiendo escoger la más adecuada para cada dominio: costmaps para interiores planos, *grid maps* de elevación, superficies navegables como NavMap o representaciones 3D como Bonxai u OctoMap para drones o robots marinos. Su documentación y ejemplos se encuentran en `https://easynavigation.github.io/` y el repositorio en `https://github.com/EasyNavigation/EasyNavigation`.

EasyNav propone una navegación componible sin fijar una única representación espacial.

Arquitectura en un proceso: composición y *SystemNode*. La figura 7.8 muestra el diseño a alto nivel: un *SystemNode* (*Lifecycle*) compone e inicializa módulos internos responsables de sensores, mapas, localización, planificación y control. El despliegue se concentra en un único proceso con tres hilos bien diferenciados: uno de tiempo real para las tareas críticas de navegación, uno de no tiempo real para tareas menos críticas y otro, también de tiempo real, dedicado a mantener actualizado un *buffer* interno de TFs que comparten nodos y plugins. Esta organización reduce fricción de despliegue (menos procesos que coordinar) y permite una configuración relativamente compacta.

EasyNav concentra la navegación en un proceso con hilos de propósito explícito.

Desde el punto de vista de operabilidad, un *SystemNode* con *lifecycle* facilita secuencias de arranque/parada controladas y una inicialización ordenada de dependencias (sensores antes que localización; localización antes que planificación; etc.). Esto conecta con el uso de *lifecycle nodes* como herramienta arquitectónica (capítulo 4).

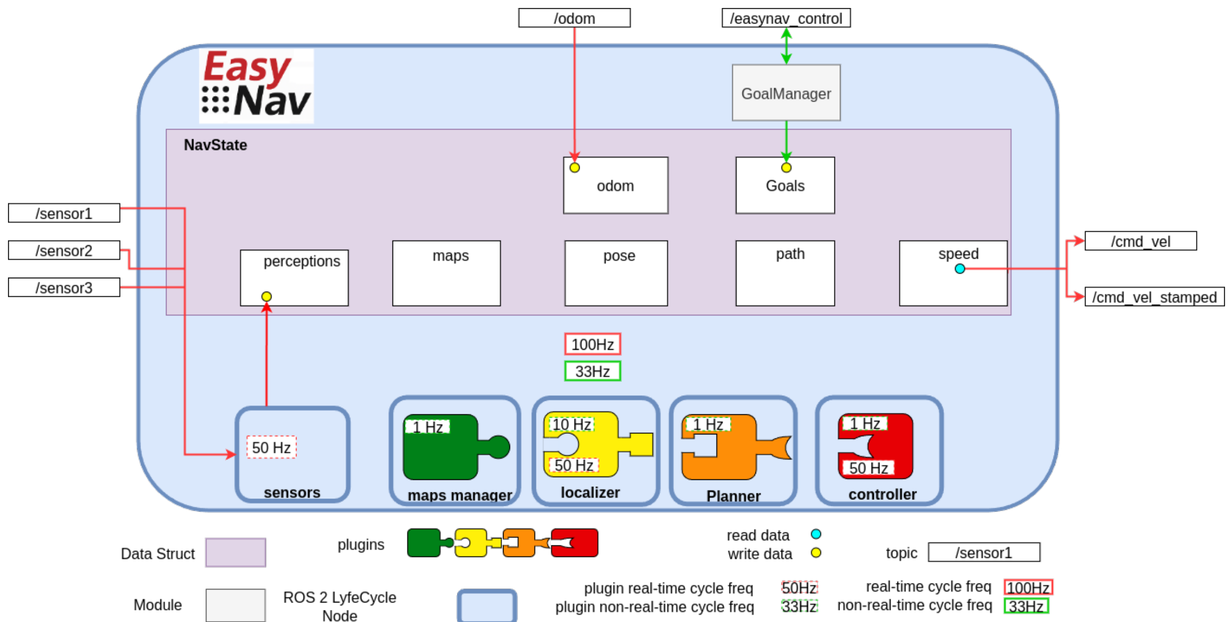


Figura 7.8: Diseño general de EasyNav. Fuente: `https://easynavigation.github.io/_images/easynav_simple_design_v2.png`

***Blackboard* compartido: *NavState*.** Un elemento arquitectónico distintivo es el uso de un estado compartido (*NavState*) como *memoria* del sistema. Este patrón recuerda al *blackboard* de los BTs (capítulo 6): diferentes módulos leen/escriben información sobre el estado actual de navegación, objetivos, progreso, etc. Internamente, los nodos y plugins no se comunican mediante topics, servicios o acciones ROS 2, sino mediante una *blackboard thread-safe*. Esto reduce de forma apreciable la carga del sistema y evita inyectar tráfico innecesario al middleware para coordinar componentes que ya viven dentro del mismo proceso. El beneficio es una integración directa entre módulos; el coste es que aumenta la necesidad de definir contratos claros (qué campos produce cada módulo, y cuándo) para evitar acoplamiento implícito.

La blackboard interna reduce carga middleware a cambio de contratos más cuidados.

Variabilidad mediante combinaciones de plugins. EasyNav también está diseñado para ser altamente configurable mediante plugins (planificadores, controladores, localizadores, gestores de mapa, etc.). La figura 7.9 ilustra cómo se combinan componentes para construir una solución concreta. Lo importante aquí es que la arquitectura no fuerza una única representación del entorno: deja espacio para seleccionar la estructura de datos espacial que mejor encaje con el problema. Esto conecta con el principio de modularidad por sustitución (plugins) ya visto en Nav2: se mantiene una arquitectura estable y se cambia el *interior* de los módulos.

La modularidad de EasyNav está en desacoplar arquitectura y representación espacial.

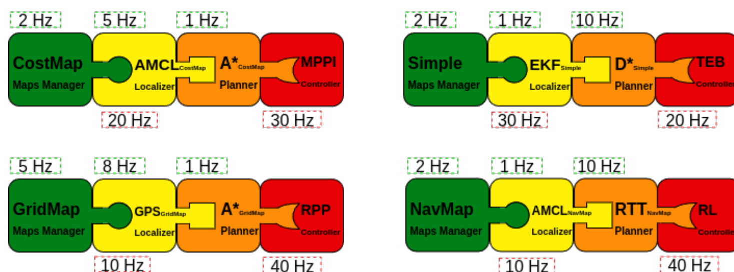


Figura 7.9: Combinaciones de plugins en EasyNav. Fuente: `https://easynavigation.github.io/_images/plugin_combinations.png`

La figura 7.10 complementa el diagrama anterior con más combinaciones

típicas.

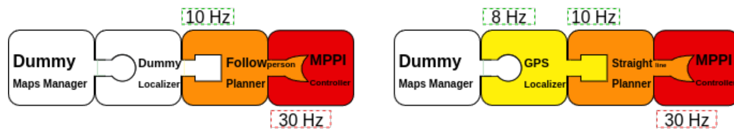


Figura 7.10: Más combinaciones de plugins en EasyNav. Fuente: https://easynavigation.github.io/_images/plugin_combinations_2.png

Timing: ciclos RT/no-RT. EasyNav explicita dos ciclos de ejecución con frecuencias distintas (parámetros típicos `rt_freq` y `freq`), separando tareas con requisitos temporales más estrictos de otras de carácter más deliberativo. Además, en lugar de confiar en *timers* ROS 2 para ejecutar cada módulo, el sistema realiza llamadas explícitas a los componentes en el orden adecuado. Esto mejora el determinismo y la latencia extremo a extremo, porque evita que la infraestructura de ROS 2 consuma tiempo de computación y predictibilidad en la coordinación interna.

EasyNav busca determinismo con ciclos explícitos y ejecución ordenada de módulos.

Consistencia del *NavState* bajo concurrencia. Si el *NavState* es un blackboard compartido, su diseño debe contemplar concurrencia: quién escribe qué, a qué frecuencia y con qué garantías. Una forma de verlo es como un contrato de datos entre módulos: campos inmutables vs. campos actualizados, *timestamps* para evitar lecturas obsoletas, y mecanismos de exclusión cuando varias tareas compiten por actualizar el mismo recurso.

Interfaz externa y contratos. Aunque el sistema se integra internamente en un proceso, ofrece una interfaz ROS 2 hacia el exterior para aquello que sí necesita cruzar el límite del *framework*: información sensorial, TFs externas, depuración o publicación de velocidades/comandos. Esta es una decisión práctica: permite conectar con otras capas (misión, UI, supervisión) usando el modelo de comunicación estándar de ROS 2 (capítulo 2), sin obligar a que todo el sistema sea monolítico.

En particular, la presencia de herramientas de operación (como una interfaz textual) refuerza la idea de que una arquitectura de referencia debe contemplar no solo *funcionalidad*, sino también despliegue, diagnóstico y control humano en el bucle.

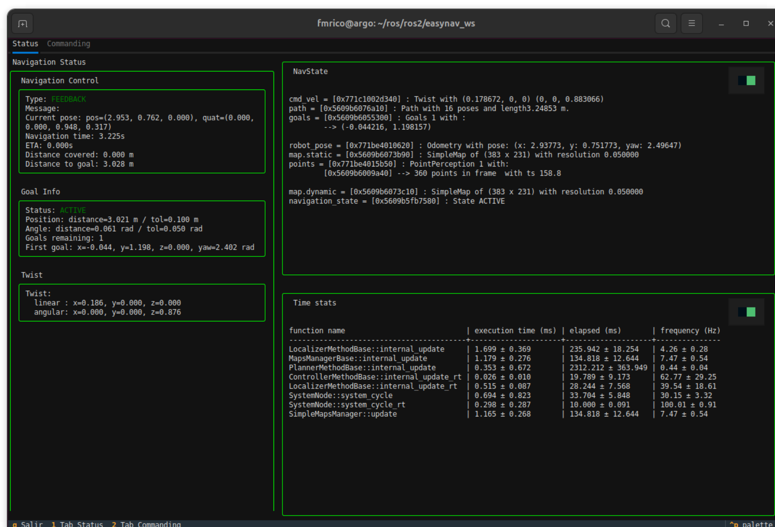


Figura 7.11: Interfaz TUI de EasyNav. Fuente: https://easynavigation.github.io/_images/easynav_simple_tui.png

Trade-offs: monolito compuesto vs. sistema distribuido. Comparado con Nav2, EasyNav se decanta por componer módulos en un único proceso y reservar ROS 2 principalmente para la frontera externa del sistema. Esto reduce latencias de comunicación, simplifica la configuración y evita mucha carga interna sobre el RMW/DDS, pero aumenta el acoplamiento operativo: un fallo en un módulo puede afectar al conjunto y la asignación de recursos (hilos, prioridades, afinidad) pasa a ser crítica. Desde el punto de vista de arquitectura, ambos enfoques son válidos; la clave es que el diseño haga explícitos los supuestos de timing, concurrencia y aislamiento (capítulo 7).

El monoproceso reduce latencia, pero exige más cuidado con acoplamiento y recursos.

7.2. Síntesis de criterios arquitectónicos transversales

Hasta aquí el libro ha presentado conceptos teóricos repartidos entre capas, comunicación, representación, supervisión y modelos de control; y en la sección anterior los hemos vuelto a ver encarnados en arquitecturas de referencia concretas. Conviene, por tanto, hacer una síntesis explícita antes de entrar en Deep ROS 2, no para añadir una nueva teoría desconectada, sino para nombrar con precisión criterios que ya habían aparecido de forma implícita en capítulos anteriores y en el análisis de Nav2, PlanSys2 y EasyNav.

Esta sección recapitula criterios transversales ya usados en el libro y los conecta con las arquitecturas de referencia.

Patrones arquitectónicos y antipatrones

Un patrón arquitectónico es una solución estructural recurrente para un problema también recurrente; un antipatrón, por el contrario, es una solución aparentemente cómoda que degrada propiedades globales del sistema cuando el sistema crece, se distribuye o se mantiene durante mucho tiempo. En robótica esto es especialmente importante porque los sistemas raras veces son "planos": combinan percepción, planificación, control, supervisión, teleoperación, registro de datos y despliegue distribuido. En ese contexto, patrones como *blackboard*, *pipeline* o *dataflow*, *publish–subscribe*, arquitectura basada en componentes, arquitectura por plugins y el espectro entre microservicios y monolito compuesto ayudan a organizar responsabilidades, dependencias y ritmos de ejecución.

Patrones y antipatrones permiten nombrar regularidades estructurales de sistemas robóticos complejos.

El patrón *blackboard* resulta útil cuando varios módulos deben compartir una representación común del estado sin acoplarse directamente entre sí; el patrón *pipeline* aparece cuando el valor está en la cadena completa de transformación de datos y no en un componente aislado. El patrón *publish–subscribe* desacopla productores y consumidores a costa de introducir asincronía y menor control temporal, la arquitectura basada en componentes ayuda a encapsular responsabilidades, la arquitectura por plugins captura variabilidad sin romper la estructura principal y la discusión microservicios frente a monolito compuesto recuerda que distribuir funcionalidad no siempre equivale a mejorar el sistema. Cada uno de estos patrones resuelve un problema diferente y, por tanto, conviene nombrarlos de forma explícita para no mezclar decisiones que pertenecen a planos distintos del diseño.

Cada patrón responde a un problema distinto y no debería confundirse con una idea genérica de modularidad.

En paralelo, los antipatrones aparecen cuando una solución local parece conveniente pero acaba degradando claridad, determinismo o mantenibilidad. El abuso de topics consiste en convertir cualquier interacción

en una conversación distribuida aunque los módulos compartan proceso, memoria o ciclo de trabajo; el acoplamiento oculto aparece cuando un módulo depende de nombres, frecuencias, orden de publicación o convenciones que no están escritas en ningún contrato; la lógica dispersa en callbacks rompe la trazabilidad del flujo de control; y el estado distribuido inconsistente aparece cuando varias representaciones del mundo dejan de coincidir sin que el sistema tenga reglas claras para reconciliarlas. Son antipatrones particularmente dañinos porque muchas veces "funcionan" durante las primeras iteraciones del proyecto y solo se vuelven visibles cuando aparecen carga real, depuración compleja o necesidad de evolución.

Los antipatrones suelen funcionar al principio, pero degradan el sistema cuando crece o se vuelve difícil de depurar.

En este libro esos patrones ya habían aparecido, aunque a menudo sin ser nombrados explícitamente: el *pipeline* se veía en percepción y representación (capítulo 3), el *publish–subscribe* en comunicación (capítulo 2), el *blackboard* en BTs (capítulo 6) y la descomposición por componentes y capas en los capítulos 1 y 4. La sección de arquitecturas de referencia les ha dado además cuerpo práctico: Nav2 es fuertemente componente y plugin-oriented, PlanSys2 hace explícitos contratos deliberativos entre piezas bien diferenciadas y EasyNav se aproxima a un monolito compuesto con *blackboard* interno precisamente para evitar parte de los costes de una distribución excesiva. Leer esas arquitecturas con este lenguaje permite entender mejor no solo cómo están construidas, sino por qué tomaron esas decisiones.

Las arquitecturas de referencia del capítulo pueden releerse como ejemplos concretos de esos patrones y antipatrones.

Cómo evaluar una arquitectura robótica

Evaluar una arquitectura robótica significa medir propiedades globales del sistema, no solo verificar que resuelve un caso feliz. Una arquitectura puede funcionar en una demo y, sin embargo, ser mala si bajo carga crece la latencia, si el sistema se vuelve opaco al depurarlo o si cualquier cambio obliga a reescribir medio código. Por eso, además de la funcionalidad, interesa medir latencia extremo a extremo, *throughput*, determinismo temporal, escalabilidad, acoplamiento, observabilidad y testabilidad.

La evaluación arquitectónica se centra en propiedades sistémicas y no sólo en funcionalidad aparente.

La latencia extremo a extremo es especialmente importante en robótica porque el comportamiento útil rara vez termina en el primer callback: empieza en una percepción o una orden, atraviesa varias capas y solo tiene sentido cuando el robot actúa o cuando el supervisor recibe confirmación. El *throughput* importa cuando hay flujos continuos de datos o cuando varios subsistemas compiten por los mismos recursos. El determinismo mide la variabilidad temporal, no sólo la media; la escalabilidad pregunta qué ocurre al añadir más sensores, más nodos, más robots o más volumen de información; el acoplamiento revela cuánto conocimiento oculto se ha filtrado entre módulos; y la testabilidad mide hasta qué punto una pieza puede aislarse, reproducirse e instrumentarse.

Los criterios relevantes abarcan tiempo, capacidad de proceso, crecimiento, acoplamiento y facilidad de prueba.

Lo importante es que estos criterios no apuntan siempre en la misma dirección. Una arquitectura muy distribuida puede mejorar reutilización y aislamiento organizativo, pero aumentar latencia y sobrecarga de coordinación; una arquitectura más integrada puede mejorar el control temporal, pero exigir contratos internos más finos y mecanismos más fuertes de aislamiento. Por eso no existe una arquitectura "óptima" en abstracto: existe una arquitectura más apropiada para un tipo de plataforma, un perfil de misión, un presupuesto computacional y un nivel de criticidad determinados.

No existe una arquitectura universalmente mejor, sino compromisos distintos según el contexto del robot y la misión.

En el libro estos criterios ya habían aparecido repartidos entre comunicación, capas, FSMs y BTs (capítulos 2, 4, 5 y 6), pero aquí pueden leerse sobre ejemplos completos. Nav2 prioriza reutilización, extensibilidad y claridad funcional aun pagando más sobrecarga distribuida; EasyNav empuja más hacia latencia y control del flujo interno; y PlanSys2 añade otra dimensión de evaluación, la consistencia entre estado simbólico, ejecución y capacidad de replanning. La comparación entre estos tres casos deja claro que evaluar arquitectura es, en gran medida, comparar *trade-offs* defendibles y relacionarlos con los objetivos reales del sistema.

Nav2, PlanSys2 y EasyNav sirven como casos de estudio para comparar compromisos arquitectónicos reales.

Testing, validación y depuración arquitectónica

El *testing* arquitectónico se ocupa de validar interacciones entre módulos, tiempos, contratos y modos de despliegue, no solo funciones aisladas. En robótica, una enorme parte de los fallos no aparece dentro de una clase o de un nodo individual, sino en la frontera entre percepción, decisión, sincronización temporal, red y actuadores. Por eso el testing de integración, la validación en simulación, el contraste con ejecución real, la reproducibilidad de experimentos, el *logging* estructurado, el *tracing* y la introspección del sistema en marcha son técnicas de arquitectura, no meras herramientas auxiliares de desarrollo.

La validación arquitectónica debe centrarse en interacciones y no sólo en unidades de software.

La simulación, por ejemplo, permite explorar escenarios, forzar fallos, repetir ejecuciones y diseñar campañas de validación de forma económica, pero nunca sustituye del todo al robot real porque cambian ruido, retardos, reloj, CPU, red y contacto con el entorno. La reproducibilidad obliga a fijar configuración, versiones, semillas, datos de entrada y condiciones de despliegue; de lo contrario, el sistema se vuelve imposible de comparar entre pruebas. El *logging* estructurado aporta contexto semántico y temporal sobre qué ocurrió y por qué; el *tracing* permite seguir eventos a bajo nivel y reconstruir latencias reales; y la introspección del grafo, de TF, de lifecycle, de topics y de actions convierte al sistema en algo observable, no en una caja negra.

Simulación, reproducibilidad, logging, tracing e introspección son instrumentos complementarios para validar sistemas robóticos.

Esto es particularmente importante porque una buena arquitectura no sólo debería permitir operar al robot, sino también explicar su comportamiento cuando algo no va como se esperaba. Si no puede saberse qué módulo estaba activo, qué datos manejaba, qué callback quedó bloqueado o en qué tramo de la cadena se degradó el tiempo de respuesta, entonces la arquitectura está fallando también como instrumento de depuración. En entornos académicos e industriales, la diferencia entre un sistema mantenible y uno frágil suele empezar precisamente ahí: en la capacidad de observar y repetir lo que ha ocurrido.

Una arquitectura mantenible debe hacer observable y repetible el comportamiento del sistema cuando falla.

En capítulos anteriores ya se había intuido esta necesidad al discutir percepción, representación y supervisión; y en esta misma sección aparece de forma natural en las arquitecturas de referencia. Nav2 concentra mucha observabilidad en lifecycle, costmaps y BTs; PlanSys2 obliga a validar la coherencia entre mundo simbólico y ejecución real; EasyNav gana control interno a costa de exigir mejores puntos de instrumentación dentro del proceso. La transición a Deep ROS 2 hace además más relevante el *tracing*, porque muchos problemas ya no serán puramente funcionales sino temporales y concurrentes.

Las arquitecturas de referencia ilustran distintas formas de observabilidad y validación del comportamiento global.

Gestión de errores y tolerancia a fallos

La tolerancia a fallos consiste en definir cómo se detecta, aísla y recupera el sistema cuando algo sale mal. Esto incluye manejo explícito de fallos en nodos, *retries* para errores transitorios, *watchdogs* para detectar ausencia de actividad o violaciones temporales, degradación funcional cuando no puede mantenerse toda la capacidad y estrategias de recuperación que devuelvan al sistema a un estado operativo o seguro. La idea clave es que el fallo no es una anécdota, sino una situación prevista que condiciona la estructura del sistema.

La arquitectura debe tratar el fallo como un caso normal de operación y no como una excepción marginal.

No todos los fallos son iguales, y por eso tampoco todas las respuestas deberían ser iguales. Hay fallos transitorios de comunicación, fallos persistentes de hardware, errores lógicos en una acción, divergencia entre estimación y realidad, saturación temporal o simplemente ausencia de información necesaria para avanzar. Sin esta distinción, el tratamiento del error se convierte en un conjunto de *ifs* locales y mensajes de log, pero no en una política coherente de operación.

Una política de robustez útil distingue entre fallos recuperables, tolerables con degradación y fallos que obligan a reconfigurar o detener el sistema.

La degradación funcional es especialmente relevante en robótica porque muchas veces lo razonable no es "seguir como si nada" ni "apagar todo", sino mantener una parte útil del comportamiento mientras se limita el riesgo. Un robot puede dejar de planificar globalmente y seguir en modo seguro, o puede perder una fuente de percepción y reconfigurarse temporalmente con otra. Del mismo modo, un *watchdog* no sirve solo para detectar que algo no responde, sino para activar mecanismos de transición a otros modos o para escalar el problema a un supervisor.

La robustez arquitectónica implica conservar comportamiento útil y seguro en presencia de fallos parciales.

En el libro esta perspectiva ya se había preparado en los capítulos de máquinas de estados y BTs (capítulos 5 y 6), donde la recuperación se modela como parte del control. En la sección de arquitecturas de referencia, Nav2 lo hace muy visible con sus ramas de recuperación y chequeo de progreso, mientras que PlanSys2 lo expresa mediante monitorización, fallo de acciones y replanning. EasyNav, por su enfoque más integrado, obliga a pensar con especial cuidado cómo degradar o aislar módulos sin que el monoproceso entero se vuelva frágil.

Las arquitecturas de referencia muestran formas distintas de incorporar recuperación y degradación como parte del diseño.

Arquitecturas distribuidas reales

Una arquitectura distribuida real no es simplemente una arquitectura con varios nodos, sino una organización en la que reloj, red y replicación del estado pasan a ser restricciones de primer orden. Eso obliga a tratar explícitamente la sincronización entre máquinas, la latencia y la pérdida de red, la consistencia del estado distribuido y la partición entre cómputo cercano al robot (*edge*) y cómputo remoto (*cloud*). Cuando el sistema cruza fronteras físicas o de red, la arquitectura deja de ser sólo una cuestión de organización lógica y pasa a ser también una cuestión de disponibilidad y confianza.

Distribuir cómputo implica diseñar también con tiempo, red y consistencia, no sólo con módulos funcionales.

En robótica, esta cuestión es particularmente delicada porque el robot suele operar en tiempo físico mientras parte del sistema puede estar distribuida entre su propio computador embarcado, estaciones de supervisión, máquinas de desarrollo o servicios remotos. No basta con que los mensajes "lleguen": importa cuándo llegan, qué ocurre si se pierden, cómo se detecta el retraso excesivo y qué representación del estado se considera autoritativa cuando hay varias copias del mundo en circulación. Aquí surgen tensiones entre validez local y global del estado que no pueden ignorarse en el diseño.

Los problemas clásicos de sincronización, jitter, particiones y réplicas de estado aparecen de forma natural en robots distribuidos.

La oposición entre *edge* y *cloud* no debería entenderse solo como una cuestión de infraestructura, sino como una decisión arquitectónica sobre dónde reside la criticidad temporal y dónde puede aceptarse mayor incertidumbre. Las funciones cercanas al lazo del robot suelen exigir proximidad computacional y mayor control sobre latencia y disponibilidad; otras tareas pueden tolerar más variabilidad a cambio de mayor capacidad de cómputo o almacenamiento. En un sistema distribuido maduro, red y tiempo forman parte del diseño desde el principio.

La partición entre edge y cloud redistribuye latencia, observabilidad, seguridad y modos de fallo.

El capítulo de comunicación (capítulo 2) ya preparaba esta lectura al explicar middleware, QoS y descubrimiento; y en Deep ROS 2 reaparecerá al hablar de DDS y Zenoh. En la sección de arquitecturas de referencia, Nav2 representa bien la lógica distribuida tradicional de ROS 2, PlanSys2 añade el problema de sincronizar mundo simbólico y mundo ejecutado, y EasyNav muestra el movimiento inverso: reabsorber parte de la distribución dentro de un proceso para reducir incertidumbre de red y de coordinación interna.

Las tres arquitecturas del capítulo ilustran respuestas distintas al problema de la distribución real.

Seguridad y *safety*

Aunque este libro no se centra en *safety* ni en ciberseguridad, arquitectónicamente conviene distinguir ambos planos. La *safety* se preocupa por evitar daño físico o comportamientos inseguros del robot; la seguridad se ocupa de proteger comunicaciones, interfaces y componentes frente a accesos no autorizados o manipulación. En ambos casos la arquitectura importa porque decide qué queda aislado, qué puede escalar un fallo al resto del sistema y qué modos seguros existen cuando alguna hipótesis deja de cumplirse.

Safety y seguridad condicionan directamente cómo se aíslan componentes y cómo se recupera el sistema.

Desde el punto de vista de *safety*, conceptos como *fail-safe*, estados seguros por defecto, limitación del radio de impacto y recuperación supervisada son centrales. Un robot no debería depender de que “todo salga bien” para comportarse razonablemente: debería tener modos de parada, de degradación o de retirada controlada cuando fallan sensores, comunicaciones o subsistemas críticos. Desde el punto de vista de seguridad, la cuestión se traslada a autenticación, confidencialidad, control de acceso, segmentación de red y protección de interfaces.

La safety exige modos seguros explícitos y la seguridad exige proteger interfaces, red y accesos.

La conexión con el resto del libro es directa aunque no siempre se haya explicitado con esta terminología. Hablar de supervisión, modos de operación, aislamiento de responsabilidades o separación entre control crítico y no crítico ya era, implícitamente, hablar de *safety* arquitectónica. Del mismo modo, discutir contratos de interacción, middleware distribuido y políticas de despliegue apuntaba ya a preocupaciones de seguridad; en un sistema real, ambas dimensiones se cruzan continuamente.

Muchos conceptos previos del libro ya contenían, de forma implícita, preocupaciones de safety y seguridad.

En la sección de arquitecturas de referencia estas cuestiones pueden leerse con más concreción: Nav2 necesita límites claros para que una recuperación no degrade otras capacidades, PlanSys2 depende de contratos fiables entre deliberación y ejecución, y EasyNav, al concentrar más elementos dentro de un proceso, obliga a cuidar aún más el aislamiento interno. La discusión posterior sobre hilos, prioridades y tiempo real también toca este punto, porque seguridad y *safety* dependen en parte de poder predecir cómo se comporta el sistema bajo carga.

Las arquitecturas de referencia muestran que aislamiento, recuperación y previsibilidad temporal también son decisiones de safety y seguridad.

Diseño de interfaces y contratos

Una interfaz arquitectónica es el contrato que fija qué ofrece un módulo, qué espera del resto y cómo puede evolucionar sin romper el sistema. Eso incluye tipos de datos, precondiciones y postcondiciones, semántica temporal, manejo de errores, versionado y compatibilidad. Cuanto más grande y duradero es un sistema, más valioso resulta que sus interfaces sean explícitas, estables y verificables.

Una interfaz buena define responsabilidades y hace posible la evolución compatible del sistema.

La importancia de una buena interfaz se hace evidente cuando un sistema evoluciona. Si un módulo obliga a conocer detalles internos de otro, si la validez de un mensaje depende de convenciones no documentadas o si los tiempos esperados de respuesta no forman parte del contrato, entonces la arquitectura queda apoyada en suposiciones tácitas. En robótica esto es todavía más crítico porque los contratos no se limitan a datos: incluyen también estado del mundo, hipótesis temporales y consecuencias sobre el comportamiento físico.

Diseñar interfaces no es sólo fijar tipos, sino también semántica, tiempos, errores y evolución futura.

Además, las interfaces cumplen una función epistemológica: hacen visible qué sabe cada módulo y qué no sabe. Una buena arquitectura reduce el conocimiento innecesario entre piezas, de forma que cada una dependa de acuerdos mínimos pero suficientes. Esto ayuda tanto a la mantenibilidad como a la posibilidad de sustituir implementaciones, añadir plugins o rediseñar internamente un subsistema sin propagar cambios por todo el sistema.

Las interfaces bien diseñadas reducen acoplamiento y facilitan sustitución, extensión y mantenimiento.

Esta idea ya había atravesado el libro desde el capítulo de comunicación (capítulo 2) y las capas misión–tarea–capacidad (capítulo 4). La sección de arquitecturas de referencia la concreta de tres maneras distintas: Nav2 define contratos de plugins y de actions; PlanSys2 usa contratos simbólicos y protocolos de ejecución entre planificador, executor y *performers*; y EasyNav desplaza parte de ese contrato al *NavState* y a la interacción entre módulos internos.

Las arquitecturas de referencia muestran contratos distintos, pero en todas ellas la calidad depende de la claridad de esos límites.

Configuración y despliegue

La configuración y el despliegue forman parte de la arquitectura porque determinan cómo se materializa el diseño lógico en un sistema repetible y operable. Esto incluye parametrización sistemática, configuración reproducible, orquestación con *launch* complejo y despliegue en robots reales con dependencias, modos y recursos bien definidos. Una arquitectura que sólo existe "en el papel" pero no puede arrancarse de forma fiable, inspeccionarse o modificarse sin romperse está incompleta.

La arquitectura no termina en la estructura lógica: también incluye cómo se arranca, configura y reproduce el sistema.

En muchos sistemas robóticos, buena parte de la complejidad real no está en la lógica de negocio, sino en cómo se combinan parámetros, recursos, rutas, políticas de middleware, modos del sistema y secuencias de arranque. Una parametrización mal planteada lleva a configuraciones opacas, a interacciones difíciles de reproducir y a errores que dependen del entorno en el que se ejecuta el robot. Por eso conviene decidir explícitamente qué puede cambiar y con qué impacto sobre el comportamiento global.

La configuración debe pensarse como diseño de variabilidad con límites, valores por defecto y efectos controlados.

El despliegue, además, hace visibles aspectos que en el código pueden pasar desapercibidos: dependencias temporales de arranque, recursos que compiten por CPU, diferencias entre laboratorio y robot real, necesidad de recuperación tras reinicio parcial o coexistencia de módulos críticos y no críticos. Una arquitectura madura no sólo define qué componentes existen, sino cómo se ponen en marcha, en qué orden, con qué dependencias y

con qué garantías de reproducibilidad. De ahí que estas decisiones pesen mucho más de lo que a veces se les concede.

Launch, lifecycle y disciplina de parametrización son herramientas arquitectónicas y no simples detalles operativos.

En el libro esto ya se intuía al hablar de capas, modos de operación y contratos, y la sección de arquitecturas de referencia lo confirma con claridad. Nav2 concentra mucha complejidad precisamente en la configuración coherente de costmaps, planners y lifecycle; PlanSys2 exige alinear dominio, problema, planificador y acciones ejecutables; EasyNav hace visible la relación entre parámetros, hilos y frecuencias internas.

Las arquitecturas de referencia muestran que la reproducibilidad del despliegue es también una métrica de calidad arquitectónica.

Arquitectura y sistema operativo

La arquitectura de un robot no termina en ROS 2: también depende de cómo el software se apoya en el scheduler, la memoria y las prioridades del sistema operativo. Cuando crecen la carga y las restricciones temporales, la política de planificación, las afinidades de CPU, la gestión de memoria y el soporte de tiempo real pasan a influir directamente en latencia, interferencia y predecibilidad. Por eso la frontera entre "arquitectura" e "implementación" se vuelve más difusa de lo que parece.

El comportamiento arquitectónico real depende también de decisiones del sistema operativo y no sólo de los módulos ROS 2.

En sistemas pequeños es fácil pensar que el sistema operativo "simplemente está ahí" y que basta con que ROS 2 procese callbacks. Sin embargo, cuando hay varios flujos de datos, control periódico, tareas críticas y procesamiento no crítico compitiendo por recursos, el scheduler deja de ser un detalle. En otras palabras, el rendimiento arquitectónico no se explica sólo por diagramas de módulos, sino también por cómo esos módulos viven sobre el sistema operativo.

Prioridades, afinidades, memoria y políticas temporales condicionan si la arquitectura cumple o no sus propiedades.

Esta dimensión es además importante porque conecta directamente estructura y ejecución. Una arquitectura muy desacoplada pero fuertemente dependiente del *runtime* puede ser excelente en mantenibilidad y, sin embargo, mostrar más variabilidad temporal; una arquitectura más integrada puede explotar mejor el control local de hilos, prioridades y afinidades, pero exigir al diseñador más responsabilidad sobre recursos compartidos.

La relación con el sistema operativo obliga a pensar en latencia real y comportamiento bajo carga, no sólo en diseño idealizado.

En capítulos anteriores esto ya se insinuaba al hablar de temporización y separación de responsabilidades, y en la sección de arquitecturas de referencia puede leerse como una diferencia de estilo. Nav2 delega bastante en el *runtime* distribuido de ROS 2; EasyNav hace más explícita la gestión local de ciclos e hilos; y PlanSys2 hereda ambos mundos, porque combina deliberación de más alto nivel con ejecución concreta en ROS 2. La sección siguiente sobre Deep ROS 2 y `SCHED_FIFO` hará explícito lo que aquí solo estamos anticipando como criterio arquitectónico.

Las arquitecturas del capítulo anticipan distintas relaciones entre diseño software, runtime y sistema operativo.

Ingeniería y evaluación comparativa de arquitecturas robóticas

La ingeniería de arquitecturas robóticas consiste en justificar, con criterios y métricas, por qué una determinada organización del sistema es adecuada para un problema concreto. Eso obliga a trabajar con criterios de calidad, métricas observables y *trade-offs* explícitos entre modularidad, determinismo, escalabilidad, observabilidad, mantenibilidad o rendimiento. Una arquitectura no se elige sólo por elegancia conceptual, sino por la adecuación de sus compromisos al contexto operativo.

Ingenierizar una arquitectura significa justificarla con evidencia y no sólo con intuición o gusto técnico.

Esta perspectiva es importante porque evita dos errores frecuentes. El primero es el fetichismo de la solución "bonita", donde una estructura

se adopta por moda o por afinidad tecnológica sin comprobar si encaja con las necesidades del robot; el segundo es el pragmatismo sin criterio, donde una arquitectura se va construyendo por acumulación de decisiones locales hasta que ya nadie puede explicar por qué el sistema está organizado así. Por eso usa conceptos y patrones, pero exige también comparación, evidencia y capacidad de explicar las decisiones tomadas.

La ingeniería arquitectónica se sitúa entre la abstracción vacía y la improvisación sin justificación.

Aquí las métricas y los ejemplos comparativos desempeñan un papel clave. Las métricas convierten intuiciones en argumentos defendibles; los *trade-offs* impiden vender como universal una solución que sólo era buena bajo ciertos supuestos; y los ejemplos comparativos permiten ver que la misma familia de problemas robóticos puede resolverse con estilos arquitectónicos distintos. Por eso resulta tan valioso cerrar la parte teórica del libro con arquitecturas de referencia reales.

El valor de las arquitecturas de referencia está en hacer comparables decisiones que, de otro modo, quedarían demasiado abstractas.

Ese ha sido, en el fondo, el hilo conductor del libro: pasar de conceptos aislados a comparaciones razonadas entre alternativas. Los capítulos iniciales aportaron vocabulario y modelos; los capítulos de FSMs, BTs y capas aportaron mecanismos de estructuración; y la primera parte de este capítulo ha servido para aterrizar todo ello en tres casos de estudio reconocibles. Estas arquitecturas no son únicamente herramientas útiles, sino referencias que permiten conectar teoría, criterios de calidad y decisiones de implementación.

Nav2, PlanSys2 y EasyNav cierran la parte teórica del libro como ejemplos comparativos antes de entrar en Deep ROS 2.

7.3. Deep ROS 2

En capítulos anteriores hemos usado ROS 2 como una caja negra: creamos nodos, publicamos y suscribimos, y dejamos que el *runtime* haga el resto. Sin embargo, cuando un sistema empieza a crecer (más nodos, más carga, más sensores y más requisitos de temporización), aparecen efectos que no se explican solo con la API superficial: *jitter*, colas que se llenan, *starvation* de callbacks, y dificultades para garantizar frecuencias.

Deep ROS estudia los efectos temporales y de concurrencia ocultos bajo la API.

Esta sección profundiza en lo que ocurre “por debajo” cuando un nodo ejecuta callbacks y cómo eso se conecta con concurrencia y tiempo real. La explicación sigue de cerca el material de referencia del libro de ROS 2 [2].

Diseño de ROS 2

La figura 7.12 resume el diseño en capas de ROS 2. Justo por debajo de los nodos escritos por el usuario aparece la capa que ofrece al programador la API con la que interactúa con el sistema. Los paquetes y nodos implementados en C++ utilizan la librería cliente *rclcpp* , mientras que los desarrollados en Python usan *rclpy* .

ROS 2 se organiza como una pila de capas que separa APIs, núcleo y middleware.

Ni *rclcpp* ni *rclpy* son implementaciones completas e independientes de ROS 2. Si lo fueran, un nodo escrito en Python podría exhibir un comportamiento distinto del mismo nodo escrito en C++. Ambas librerías se construyen sobre *rcl*, que implementa la funcionalidad básica de los elementos de ROS 2; sobre esa base, cada una adapta la API a las particularidades del lenguaje y a decisiones propias de ese nivel, como el modelo de hilos o las abstracciones idiomáticas.

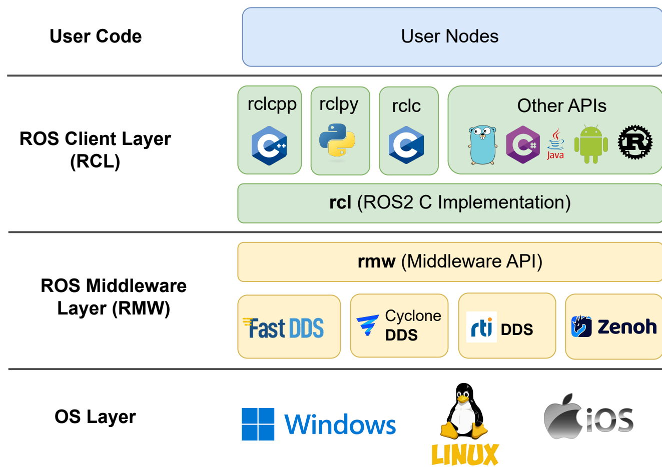


Figura 7.12: Diseño en capas de ROS 2.

Cualquier librería cliente para otros lenguajes debería construirse también sobre *rcl*. Ese es el enfoque seguido por proyectos para Rust*, Go**, Java*** o .NET****, entre otros.

rcl es el núcleo funcional de ROS 2, aunque no suele emplearse directamente al escribir aplicaciones. Para C existe además *rclc*, una librería cliente específica que facilita el desarrollo de nodos en ese lenguaje sin obligar al programador a trabajar con la interfaz de bajo nivel de *rcl*.

Un componente decisivo en el diseño de ROS 2 es su capa de comunicaciones. El grafo de cómputo puede repartirse entre varias máquinas, y eso ocurre incluso cuando el robot parece ejecutarse de forma local: es habitual tener nodos en el PC del operador para supervisar, depurar o teleoperar el sistema.

ROS 2 eligió inicialmente *Data Distribution Service (DDS)* (https://www.omg.org/omg-dds-portal) como base de su middleware de comunicaciones. DDS, implementado típicamente sobre UDP, ofrece publicación/suscripción con descubrimiento automático de publicadores y suscriptores sin depender de un servicio centralizado. Ese descubrimiento inicial suele usar multicast, mientras que las conexiones de datos posteriores se establecen normalmente en unicast. Además, DDS incorpora políticas de calidad de servicio, mecanismos de seguridad y capacidades adecuadas para sistemas distribuidos con restricciones de latencia y fiabilidad.

DDS aporta descubrimiento distribuido, QoS y soporte para requisitos temporales exigentes.

Existen varios proveedores de DDS compatibles en distinta medida con el estándar definido por la OMG (https://www.omg.org). Entre los más conocidos están FastDDS (https://github.com/eProsima/Fast-DDS), CycloneDDS (https://github.com/eclipse-cyclonedds/cyclonedds) y RTI Connext (https://www.rti.com/products/dds-standard). Desde el punto de vista de la mayoría de desarrolladores, cambiar de uno a otro apenas altera la programación diaria; sin embargo, cuando importan la latencia, el volumen de datos o el consumo de recursos, las diferencias entre implementaciones sí obligan a elegir con criterio.

ROS 2 desacopla su API del proveedor concreto, pero el rendimiento real sí depende de la implementación elegida.

En distribuciones recientes ha aparecido *Zenoh* como alternativa a DDS dentro del middleware de ROS 2. Introducido inicialmente en Iron y esta-

Zenoh amplía el espacio de diseño de ROS 2 para escenarios distribuidos con requisitos de red más duros.

* https://github.com/ros2-rust/ros2_rust
** https://github.com/tiiuae/rclgo
*** https://github.com/ros2-java/ros2_java
**** https://github.com/ros2-dotnet/ros2_dotnet

bilizado en Jazzy mediante *rmw_zenoh* (https://github.com/ros2/rmw_zenoh), Zenoh combina publicación/suscripción, consulta/respuesta y almacenamiento distribuido, y puede apoyarse en distintos transportes como UDP o TCP. Su interés arquitectónico está en que mejora la difusión eficiente de datos a través de redes complejas y resulta especialmente atractivo en escenarios de *edge computing*, despliegues geográficamente distribuidos o sistemas donde DDS no encaja bien por coste de red o escalabilidad.

Como las APIs nativas de los distintos middlewares no coinciden, ROS 2 introduce una capa intermedia llamada *rmw*. Esa capa presenta a *rcl* una interfaz común para acceder a las capacidades del middleware subyacente, de modo que cambiar de implementación se reduce en la práctica a seleccionar otra opción mediante la variable de entorno `RMW_IMPLEMENTATION`.

La implementación oficial de middleware puede cambiar entre distribuciones. En Jazzy la referencia oficial basada en DDS es FastDDS, mientras que en Galactic lo fue CycloneDDS; ese cambio deja claro que ROS 2 no fija de forma permanente un único proveedor. Las llamadas informalmente *DDS Wars* reflejan precisamente esa competencia entre implementaciones, que idealmente termina mejorando las prestaciones y la madurez del ecosistema.

Gestión de ejecución en ROS 2

La unidad básica de ejecución en ROS 2 es el nodo. Un nodo no "corre" de forma monolítica: su lógica vive en callbacks que se activan por eventos: temporizadores (código periódico), suscripciones (datos entrantes), servicios/acciones (peticiones/respuestas), etc. Quien coordina estos eventos y llama a los callbacks es el *executor*.

El executor coordina la ejecución de callbacks disparados por eventos.

Aunque a menudo no lo escribimos explícitamente, siempre hay un executor implicado. Por ejemplo, el siguiente programa parece no mencionar ninguno:

```
int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);

    auto node = rclcpp::Node::make_shared("listener");
    auto sub = node->create_subscription<std_msgs::msg::String>(
        "/chatter", 10, callback);

    rclcpp::spin(node);
    rclcpp::shutdown();
}
```

Fragmento C++ 7.1: Uso típico de `rclcpp::spin` (conceptualmente implícito un executor). label

En realidad, `rclcpp::spin` crea internamente un `SingleThreadedExecutor` para añadir el nodo y ejecutar el bucle:

```
void rclcpp::spin(rclcpp::node_interfaces::NodeBaseInterface::SharedPtr
     node_ptr)
{
    rclcpp::ExecutorOptions options;
    options.context = node_ptr->get_context();
    rclcpp::executors::SingleThreadedExecutor exec(options);
    exec.add_node(node_ptr);
    exec.spin();
    exec.remove_node(node_ptr);
}
```

Fragmento C++ 7.2: Implementación simplificada de `rclcpp::spin` (idea). label

Capas relevantes. Para entender qué controla un executor y dónde aparecen las colas y el orden de procesado, conviene separar capas conceptuales (Figura 7.13): (i) middleware (colas por topic), (ii) librerías cliente (rclcpp/rclpy) donde vive el executor, y (iii) capa de usuario donde residen los callbacks.

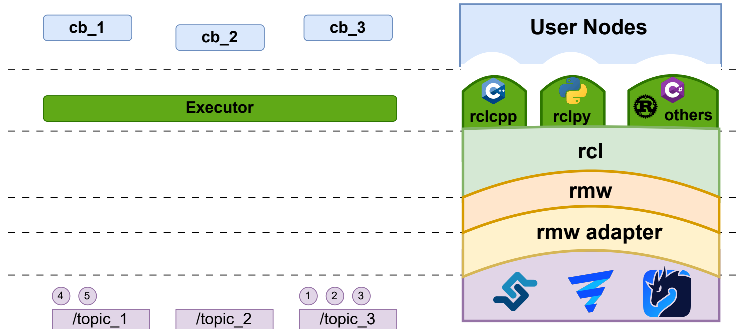


Figura 7.13: Gestión de ejecución por capas en ROS 2 (adaptado de [2]).

Wait-sets y ventana de procesado. En los executors "clásicos", el núcleo es un mecanismo de *wait-set*: el executor "duerme" hasta que alguno de los elementos que le interesan (suscripciones, timers, servicios) tenga trabajo pendiente. Cuando el wait-set se activa, el executor sabe qué colas contienen mensajes y entra en una ventana de procesado. La secuencia típica es:

El wait-set abre ventanas de procesado cuando detecta trabajo pendiente.

1. esperar en el wait-set a que haya eventos/mensajes,
2. extraer un mensaje/evento de una cola marcada,
3. ejecutar el callback asociado.

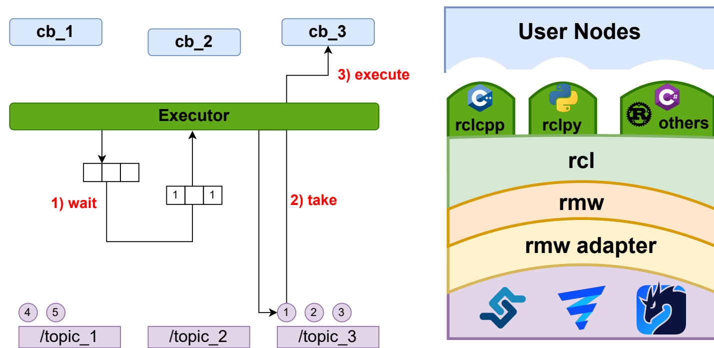


Figura 7.14: Pasos de procesado de mensajes mediante wait-set (adaptado de [2]).

La figura 7.15 resume esa lógica de forma más abstracta, separando la fase de detección de trabajo pendiente de la ejecución efectiva de callbacks.

Semántica del executor. De forma simplificada, el executor itera sobre el conjunto de elementos listos en el wait-set y ejecuta callbacks en un orden definido. Tras procesar lo que está listo, vuelve a recolectar entidades (timers, subscripciones, etc.) y arma un nuevo wait-set para el siguiente ciclo.

FIFO dentro de una ventana y efectos de orden. Cuando hay varios topics con mensajes pendientes, el executor suele procesar como máximo un mensaje por cola en cada ventana de procesado y en orden FIFO dentro de cada cola. Eso implica que mensajes "antiguos" pueden quedar para el siguiente *polling point* si la ventana actual se consume en otros

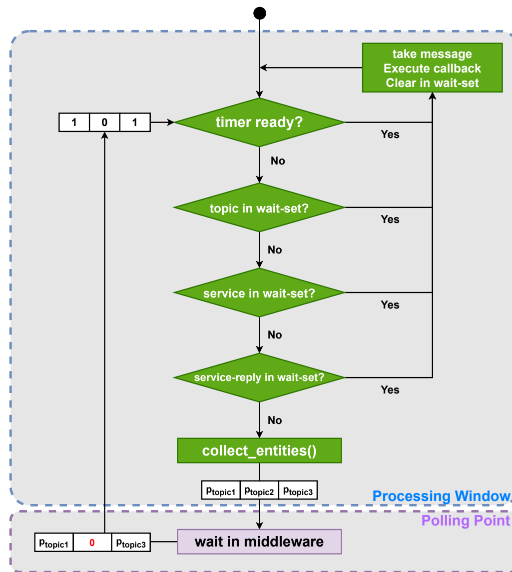


Figura 7.15: Semántica general de un executor basado en wait-set (adaptado de [2]).

callbacks. La figura 7.16 ilustra precisamente ese efecto: una ventana de ejecución no vacía todas las colas, sino que deja trabajo pendiente para iteraciones posteriores.

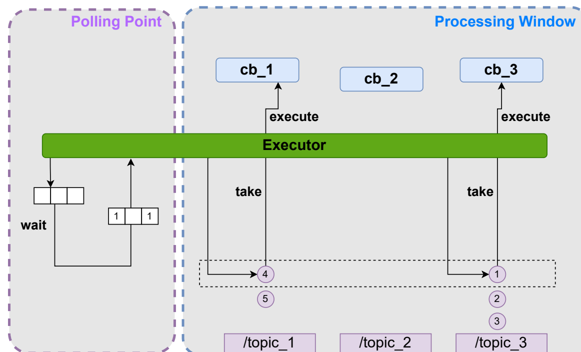


Figura 7.16: Procesado de un wait-set: un mensaje por cola por ventana (adaptado de [2]).

StaticSingleThreadedExecutor y EventsExecutor. Además de `SingleThreadedExecutor` y `MultiThreadedExecutor`, existen ejecutores con semánticas distintas (por ejemplo, `StaticSingleThreadedExecutor` , que evita rescaneos continuos a costa de exigir que las entidades estén creadas al inicio). La figura 7.17 resume esta variante, donde el conjunto de entidades se fija desde el comienzo y se reduce la sobrecarga de reconstruir continuamente el wait-set. Conceptualmente:

Executors en práctica: un ejemplo de *starvation*

Para visualizar el efecto del orden y de la carga, consideremos el patrón productor/consumidor (el ejemplo completo se describe en [2]). El productor publica a dos topics a alta frecuencia; el consumidor tiene dos suscriptores y un timer "pesado". En un `SingleThreadedExecutor`, todos los callbacks se ejecutan secuencialmente. En un `MultiThreadedExecutor`, si no hacemos nada más, dentro de un mismo nodo no aparece concurrencia por defecto, y puede producirse *starvation* de un callback.

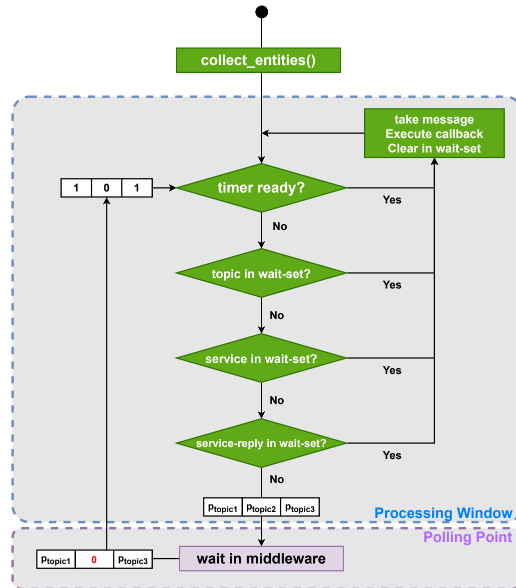


Figura 7.17: Semántica del `StaticSingleThreadedExecutor` (adaptado de [2]).

Como este mismo ejemplo se reutiliza después para entender callback groups y estrategias de priorización, conviene ver primero el código base de los dos nodos. El `ProducerNode` genera carga constante publicando en dos topics cada 1 ms:

```cpp
class ProducerNode : public rclcpp::Node
{
public:
    ProducerNode() : Node("producer_node")
    {
        pub_1_ = create_publisher<std_msgs::msg::Int32>("topic_1", 100);
        pub_2_ = create_publisher<std_msgs::msg::Int32>("topic_2", 100);
        timer_ = create_wall_timer(
            1ms, std::bind(&ProducerNode::timer_callback, this));
    }

    void timer_callback()
    {
        message_.data += 1;
        pub_1_->publish(message_);
        message_.data += 1;
        pub_2_->publish(message_);
    }

private:
    rclcpp::Publisher<std_msgs::msg::Int32>::SharedPtr pub_1_, pub_2_;
    rclcpp::TimerBase::SharedPtr timer_;
    std_msgs::msg::Int32 message_;
};
```

Fragmento C++ 7.3: Nodo productor del ejemplo productor/consumidor. label

El `ConsumerNode` es el nodo realmente interesante: tiene dos suscripciones, cada una con una carga simulada de 500 μs, y un timer periódico de 10 ms cuya callback consume 5 ms. Esa combinación hace que sus colas tiendan a llenarse y permite observar con claridad el efecto del orden de ejecución:

```cpp
class ConsumerNode : public rclcpp::Node
{
public:
    ConsumerNode() : Node("consumer_node")
    {
        sub_2_ = create_subscription<std_msgs::msg::Int32>(
            "topic_2", 100, std::bind(&ConsumerNode::cb_2, this, _1));
```

```cpp
        sub_1_ = create_subscription<std_msgs::msg::Int32>(
            "topic_1", 100, std::bind(&ConsumerNode::cb_1, this, _1));

        timer_ = create_wall_timer(
            10ms, std::bind(&ConsumerNode::timer_cb, this));
    }

    void cb_1(const std_msgs::msg::Int32::SharedPtr msg)
    {
        TRACE_EVENT(session);
        waste_time(500us);
    }

    void cb_2(const std_msgs::msg::Int32::SharedPtr msg)
    {
        TRACE_EVENT(session);
        waste_time(500us);
    }

    void timer_cb()
    {
        TRACE_EVENT(session);
        waste_time(5ms);
    }

    void waste_time(const rclcpp::Duration & duration)
    {
        auto start = now();
        while (now() - start < duration);
    }

private:
    rclcpp::Subscription<std_msgs::msg::Int32>::SharedPtr sub_1_;
    rclcpp::Subscription<std_msgs::msg::Int32>::SharedPtr sub_2_;
    rclcpp::TimerBase::SharedPtr timer_;
};
```

Fragmento C++ 7.4: Nodo consumidor base usado en los ejemplos de starvation, callback groups y tiempo real. label

Con estos dos nodos, basta con cambiar el tipo de executor en `main()` para pasar de la ejecución monohilo a la multihilo. Las figuras siguientes muestran precisamente qué ocurre con este mismo código cuando el executor es `SingleThreadedExecutor` y cuando pasa a ser `MultiThreadedExecutor`.

El caso monohilo se construye con un `main()` completamente convencional: se crean ambos nodos, se añaden al executor y se llama a `spin()`.

```cpp
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);

    auto node_pub = std::make_shared<ProducerNode>();
    auto node_sub = std::make_shared<ConsumerNode>();

    rclcpp::executors::SingleThreadedExecutor executor;

    executor.add_node(node_pub);
    executor.add_node(node_sub);

    executor.spin();

    rclcpp::shutdown();
    return 0;
}
```

Fragmento C++ 7.5: Caso base con texttSingleThreadedExecutor. label

La figura 7.18 muestra la traza correspondiente a este primer caso: aunque hay varias fuentes de trabajo, todas las callbacks del consumidor se ejecutan secuencialmente.

Para pasar al caso multihilo no hace falta tocar ni `ProducerNode` ni

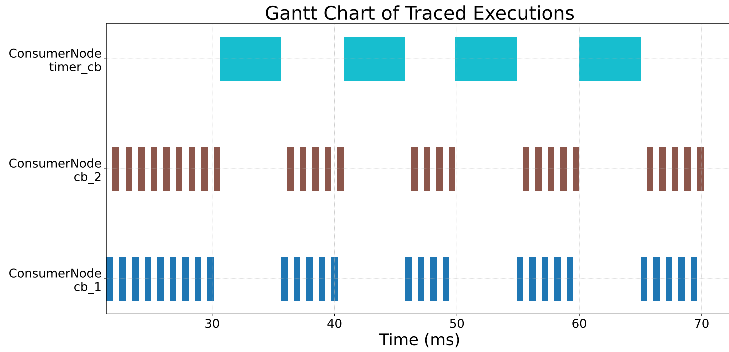


Figura 7.18: Traza con `SingleThreadedExecutor`: no hay concurrencia entre callbacks (adaptado de [2]).

`ConsumerNode`; basta con cambiar el tipo de executor y el número de hilos. Precisamente ese detalle es el que hace interesante el ejemplo: con el mismo nodo consumidor, el mero cambio de executor altera el patrón observable de ejecución.

```cpp
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);

    auto node_pub = std::make_shared<ProducerNode>();
    auto node_sub = std::make_shared<ConsumerNode>();

    rclcpp::executors::MultiThreadedExecutor executor(
        rclcpp::ExecutorOptions(), 8);

    executor.add_node(node_pub);
    executor.add_node(node_sub);

    executor.spin();

    rclcpp::shutdown();
    return 0;
}
```

Fragmento C++ 7.6: Variante con texttMultiThreadedExecutor. label

La figura 7.19 muestra el resultado de ese cambio aislado: aparece *starvation* aunque el executor disponga de múltiples hilos, porque el nodo sigue agrupando sus callbacks de la forma por defecto.

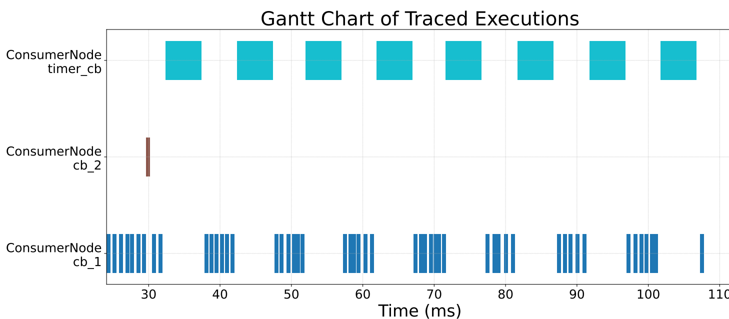


Figura 7.19: Traza con `MultiThreadedExecutor`: sin configuración adicional, el nodo no ejecuta callbacks en paralelo y puede haber *starvation* (adaptado de [2]).

Callback groups

Hasta este punto hemos simplificado deliberadamente el modelo de ejecución, pero ahora conviene ser precisos: los executors no operan realmente a nivel de nodo, sino a nivel de *callback group*. Esta diferencia suele pasar desapercibida porque cada nodo tiene un callback group por defecto, y todos los timers, suscripciones y servicios que creamos se

añaden a ese grupo si no indicamos otra cosa. Además, ese grupo por defecto suele ser *mutually exclusive*, lo que garantiza que no se ejecuten simultáneamente dos callbacks pertenecientes al mismo grupo.

La figura 7.20 visualiza esa diferencia entre dejar todas las callbacks en el grupo por defecto y separar explícitamente el timer en un grupo adicional.

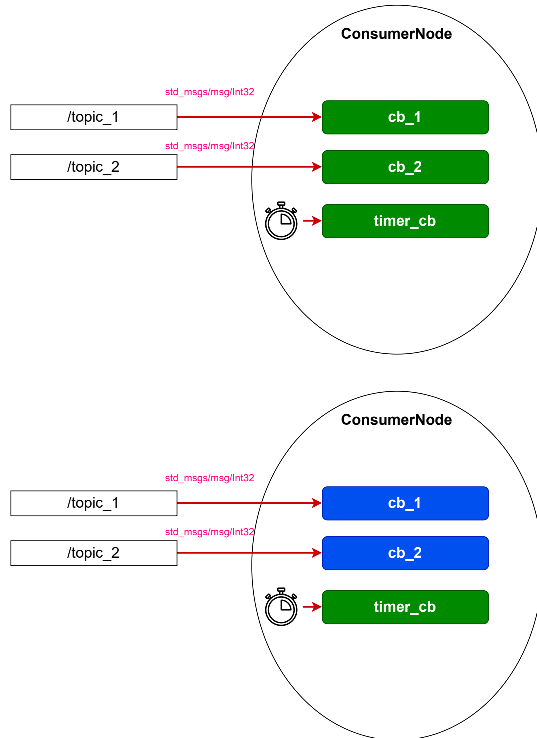


Figura 7.20: Callback groups en un nodo: arriba todo en el grupo por defecto; abajo el timer en un grupo separado (adaptado de [2]).

En ROS 2 podemos crear callback groups adicionales y decidir qué timers, suscripciones o servicios pertenecen a cada uno. Esto permite separar cargas y habilitar concurrencia controlada cuando usamos un `MultiThreadedExecutor`. El ejemplo más sencillo es mover el timer a un callback group distinto y dejar las suscripciones en otro. Así, el timer podrá ejecutarse en paralelo con las suscripciones, aunque estas seguirán sin concurrencia entre sí mientras permanezcan en un grupo *mutually exclusive*. Como el `ConsumerNode` base ya se presentó en la sección anterior, aquí basta con mostrar el constructor modificado:

```
1  ConsumerNode() : Node("consumer_node")
2  {
3      custom_cb_ = create_callback_group(
4          rclcpp::CallbackGroupType::MutuallyExclusive);
5
6      rclcpp::SubscriptionOptions options;
7      options.callback_group = custom_cb_;
8
9      sub_1_ = create_subscription<std_msgs::msg::Int32>(
10         "topic_1", 100, std::bind(&ConsumerNode::cb_1, this, _1), options)
       ;
11     sub_2_ = create_subscription<std_msgs::msg::Int32>(
12         "topic_2", 100, std::bind(&ConsumerNode::cb_2, this, _1), options)
       ;
13
14     timer_ = create_wall_timer(10ms, std::bind(&ConsumerNode::timer_cb,
       this));
15 }
```

El punto importante de este código es que no hace falta cambiar nada en `main()`. Cuando añadimos el nodo al executor, todos sus callback groups se añaden a ese executor. En este caso, el resultado es que el timer queda en un grupo separado y puede correr concurrentemente con las suscripciones, mientras que `cb_1` y `cb_2` siguen excluyéndose entre sí por pertenecer al mismo grupo.

Fragmento C++ 7.7: Cambios sobre el constructor del `ConsumerNode` para usar un callback group explícito. label

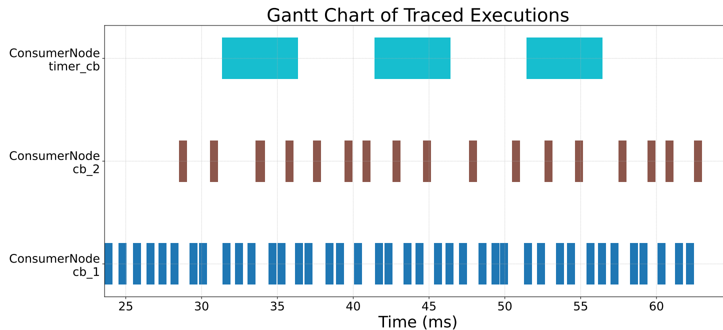


Figura 7.21: Ejecución con dos callback groups: el timer puede correr concurrentemente con las suscripciones (adaptado de [2]).

La figura 7.21 debe leerse junto con el código anterior: el timer puede ejecutarse a la vez que las callbacks de los topics precisamente porque ya no está en el mismo callback group que ellas. Sin ese detalle en la construcción del nodo, la figura parecería un comportamiento “mágico” del executor, cuando en realidad depende de cómo hemos particionado internamente las callbacks.

También existen callback groups *reentrant*, donde un `MultiThreadedExecutor` puede invocar el mismo callback simultáneamente (con datos distintos) si hay carga suficiente. Esto debe usarse con mucho cuidado, porque la concurrencia ya no viene limitada por el executor: pasa a ser responsabilidad del programador proteger el acceso a datos compartidos.

El siguiente fragmento muestra solo las diferencias respecto al `ConsumerNode` base. Todas las callbacks pasan a un callback group *reentrant*, pero se introduce un `std::mutex` para impedir que el timer y `cb_1` se ejecuten al mismo tiempo. La callback `cb_2` permanece igual que en el ejemplo inicial:

```
1  ConsumerNode() : Node("consumer_node")
2  {
3      custom_cb_ = create_callback_group(
4          rclcpp::CallbackGroupType::Reentrant);
5  
6      rclcpp::SubscriptionOptions options;
7      options.callback_group = custom_cb_;
8  
9      sub_2_ = create_subscription<std_msgs::msg::Int32>(
10         "topic_2", 100, std::bind(&ConsumerNode::cb_2, this, _1), options)
           ;
11     sub_1_ = create_subscription<std_msgs::msg::Int32>(
12         "topic_1", 100, std::bind(&ConsumerNode::cb_1, this, _1), options)
           ;
13 
14     timer_ = create_wall_timer(
15         10ms, std::bind(&ConsumerNode::timer_cb, this), custom_cb_);
16 }
17 
18 std::mutex mutex_;
19 
20 void cb_1(const std_msgs::msg::Int32::SharedPtr msg)
21 {
22     std::unique_lock<std::mutex> lock(mutex_);
```

```cpp
    TRACE_EVENT(session);
    waste_time(500us);
}

void timer_cb()
{
    std::unique_lock<std::mutex> lock(mutex_);
    TRACE_EVENT(session);
    waste_time(5ms);
}
```

Fragmento C++ 7.8: Cambios sobre el `ConsumerNode` para usar un callback group reentrant con mutex. label

Este código aclara tres ideas importantes. Primero, al crear el callback group como `Reentrant` estamos permitiendo que el executor ejecute callbacks de ese grupo en paralelo si dispone de hilos. Segundo, `create_wall_timer()` también permite especificar explícitamente el callback group del timer. Tercero, si no protegemos adecuadamente los recursos compartidos, el paralelismo adicional puede romper la lógica del nodo; de ahí el uso del mutex en `cb_1` y en `timer_cb`.

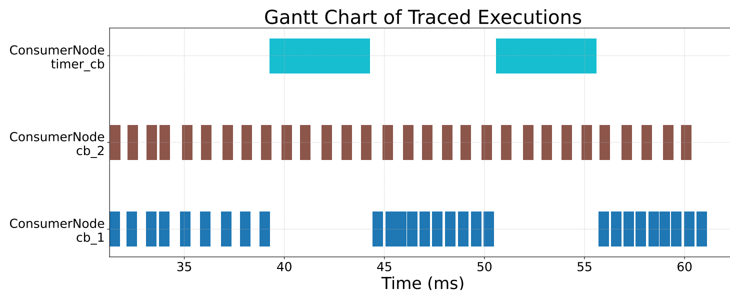


Figura 7.22: Callback group reentrant con sincronización explícita: puede haber concurrencia, pero se controla con locks (adaptado de [2]).

La figura 7.22 ya se entiende a la luz de este código: el executor sí puede lanzar callbacks concurrentes porque el grupo es *reentrant*, pero la concurrencia real queda modulada por el mutex. Por eso no basta con mostrar la traza; hay que ver también el mecanismo de sincronización que la hace posible y segura.

Tiempo real en ROS 2: conceptos y latencias

El término *tiempo real* es ambiguo fuera del contexto de sistemas embebidos/control, pero aquí lo usaremos en el sentido estricto: *responder dentro de restricciones temporales máximas*. No se trata de minimizar latencia media, sino de acotar latencia máxima. La figura 7.23 sitúa distintos casos de uso en el plano severidad/latencia y ayuda a entender por qué no todas las aplicaciones requieren el mismo rigor temporal.

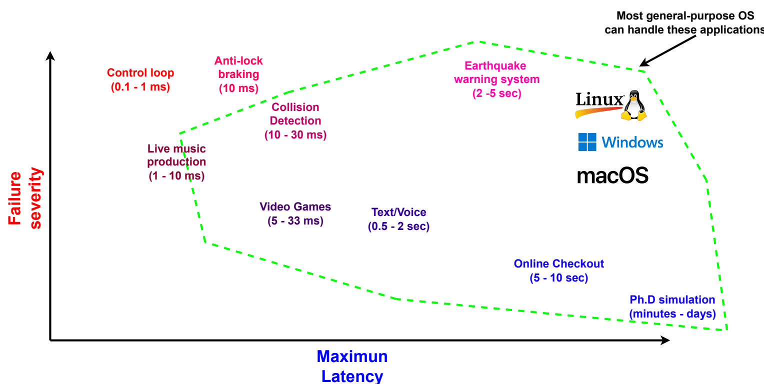


Figura 7.23: Ejemplos de casos de uso en el plano severidad/latencia (adaptado de [2]).

La latencia de reacción se compone de múltiples contribuciones (aplicación, sistema operativo, hardware), y es importante identificar dónde se introduce variabilidad. La figura 7.24 descompone precisamente esas fuentes de retardo para recordar que ROS 2 es solo una parte del problema temporal total:

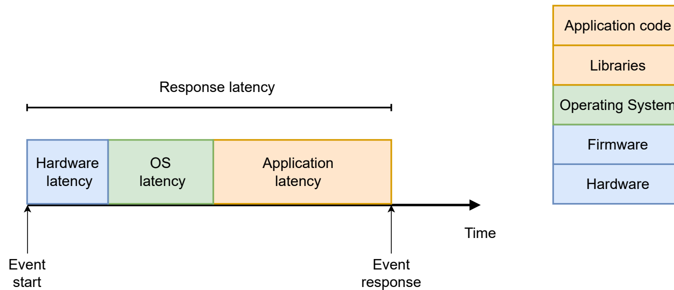


Figura 7.24: Fuentes típicas de latencia en un sistema de control (adaptado de [2]).

Planificador del sistema operativo (Linux) y `SCHED_FIFO`

En Linux, el planificador por defecto (CFS, política `SCHED_OTHER`) optimiza latencia media, pero no garantiza acotación de latencia máxima. Para tareas críticas es habitual recurrir a políticas de tiempo real como `SCHED_FIFO` o `SCHED_DEADLINE` . En particular, `SCHED_FIFO` es simple de usar y se controla con prioridades `rtprio`.

Para que un usuario con *userid*(que puede ser fmrico, por ejemplo) pueda probar de verdad el código con `SCHED_FIFO`, antes necesita permisos para elevar la prioridad de sus procesos. En un sistema Linux típico esto se configura mediante PAM creando un fichero en `/etc/security/limits.d/`. Por ejemplo, para permitir que el usuario `alumno` alcance prioridad de tiempo real 98, podría añadirse un fichero como el siguiente:

```
userid - rtprio 98
```

Fragmento C++ 7.9: Permiso de `rtprio` para probar ejemplos con `SCHED_FIFO`. Esto tiene que ir en el fichero /etc/security/limits.d/20-userid-rtprio.conf label

Tras hacer ese cambio, normalmente hay que cerrar sesión y volver a entrar para que el límite quede aplicado. Si este paso no se realiza, el ejemplo siguiente puede compilar y ejecutarse, pero la llamada a `sched_setscheduler()` fallará aunque el código sea correcto.

En la práctica, cuando queremos ejecutar un subconjunto del grafo ROS 2 con prioridad, lo habitual es lanzar el `spin()` de un executor dentro de un hilo configurado como tiempo real:

```cpp
auto rt_thread = std::thread([&]() {
    sched_param sch;
    sch.sched_priority = 90;

    if (sched_setscheduler(0, SCHED_FIFO, &sch) == -1) {
        throw std::runtime_error("failed to set scheduler");
    }

    rt_executor.spin();
});

no_rt_executor.spin();
rt_thread.join();
```

Fragmento C++ 7.10: Lanzar un executor en un hilo `SCHED_FIFO` (patrón típico). label

La figura 7.25 ayuda a interpretar cómo conviven las prioridades normales y las prioridades de tiempo real en Linux, y por qué `SCHED_FIFO` cambia de forma tan visible la capacidad de preempción.

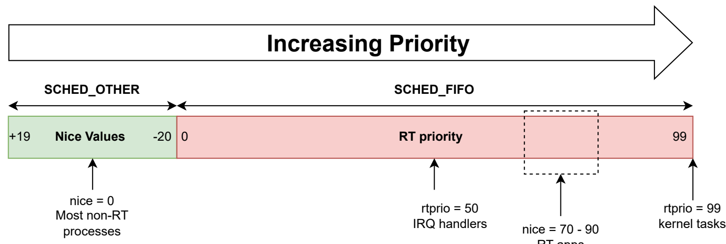


Figura 7.25: Interpretación de `nice` y `rtprio` en políticas `SCHED_OTHER` y `SCHED_FIFO` (adaptado de [2]).

Una forma típica de observar *jitter* del planificador es medir el tiempo entre activaciones periódicas de un hilo. El histograma de la figura 7.26 ilustra cómo, bajo carga, las iteraciones pueden alargarse de forma ocasional si la tarea no es prioritaria; al elevarla a `SCHED_FIFO`, la figura 7.27 muestra un comportamiento mucho más determinista.

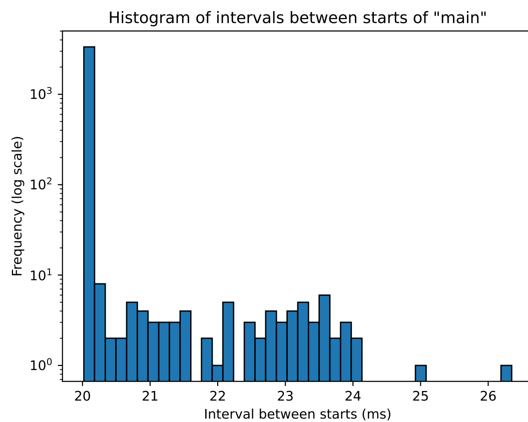


Figura 7.26: Histograma de periodos entre activaciones sin tiempo real (adaptado de [2]).

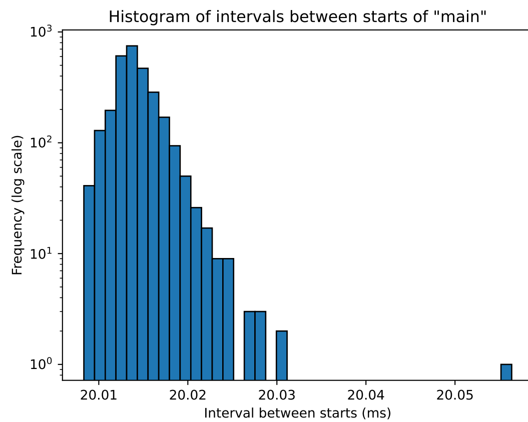


Figura 7.27: Histograma de periodos entre activaciones usando `SCHED_FIFO` (adaptado de [2]).

Estrategias de tiempo real en ROS 2 con executors y callback groups

En ROS 2, el soporte de tiempo real no viene "de serie" en el sentido de poder asignar prioridades a cada callback desde la API estándar. En la práctica, la estrategia consiste en: (i) separar callbacks críticos en callback groups dedicados, (ii) ejecutar esos grupos en executors que corren en hilos con política/prioridad de tiempo real, y (iii) dejar el resto en hilos normales.

El problema de partida es sencillo: normalmente todas las callbacks acaban ejecutándose en hilos con la misma prioridad y bajo las mismas condiciones que el resto de procesos del sistema. La figura 7.28 representa exactamente ese caso: varias callbacks comparten ejecución sin distinguir cuáles tienen requisitos temporales estrictos y cuáles no. Si el sistema se carga, todas compiten por CPU en igualdad de condiciones.

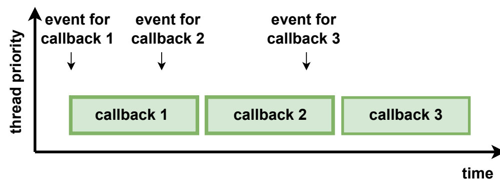


Figura 7.28: Caso no deseado: todas las callbacks comparten la misma prioridad (adaptado de [2]).

Las estrategias que siguen persiguen justo lo contrario: garantizar que las callbacks que consideramos críticas se ejecuten en hilos de mayor prioridad, mientras que las menos importantes permanecen en hilos normales. La figura 7.29 resume ese objetivo. Cuando llega el evento que debe procesar una callback crítica, una callback de menor prioridad puede incluso ser interrumpida para respetar la restricción temporal.

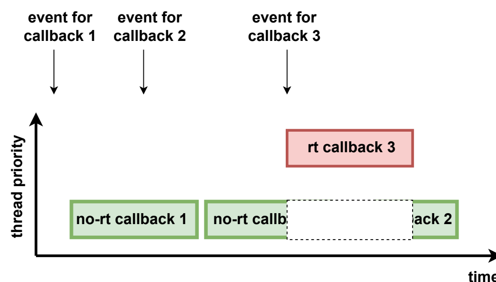


Figura 7.29: Idea: callbacks críticas en hilos con mayor prioridad, capaces de preemptar a las no críticas (adaptado de [2]).

Estrategia 1: nodos con distinta prioridad

La primera estrategia es la más simple: identificar qué nodos completos tienen requisitos más estrictos que otros y ejecutarlos en un executor diferente, asociado a un hilo de alta prioridad. Esta solución encaja bien cuando la frontera entre lo crítico y lo no crítico coincide con la frontera entre nodos.

El ejemplo considera tres nodos: un productor de datos, un consumidor crítico y un nodo registrador sin requisitos de tiempo real. El productor y el logger se asignan a un executor normal; el consumidor crítico se ejecuta en un executor que correrá en un hilo `SCHED_FIFO`. La figura 7.30 representa exactamente esa distribución.

Lo esencial del código no está en la lógica interna de los nodos, sino en cómo se crean y emparejan nodos, executors e hilos. El siguiente fragmento muestra el patrón mínimo que hace posible la estrategia:

```
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);

    auto node_producer = std::make_shared<ProducerNode>();
    auto node_consumer = std::make_shared<ConsumerNode>();
    auto node_logger = std::make_shared<LoggerNode>();

    rclcpp::executors::SingleThreadedExecutor no_rt_executor;
```

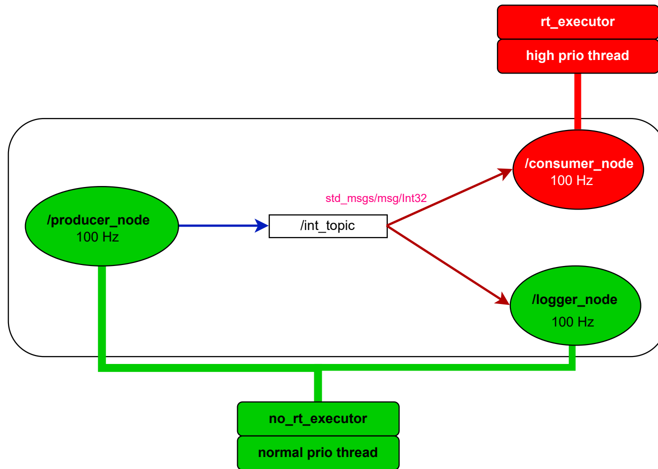


Figura 7.30: Estrategia 1: un executor en tiempo real para nodos críticos (adaptado de [2]).

```cpp
    rclcpp::executors::SingleThreadedExecutor rt_executor;

    no_rt_executor.add_node(node_producer);
    no_rt_executor.add_node(node_logger);
    rt_executor.add_node(node_consumer);

    auto rt_thread = std::thread([&]() {
        sched_param sch;
        sch.sched_priority = 90;

        if (sched_setscheduler(0, SCHED_FIFO, &sch) == -1) {
            throw std::runtime_error("failed to set scheduler");
        }

        rt_executor.spin();
    });

    no_rt_executor.spin();
    rt_thread.join();
    rclcpp::shutdown();
    return 0;
}
```

Fragmento C++ 7.11: Estrategia 1: separar nodos críticos y no críticos en executors distintos. label

Este código deja clara la idea arquitectónica: la criticidad se expresa en el despliegue. No hace falta que el nodo crítico "sepa" que es crítico; basta con ubicarlo en el executor correcto y hacer que ese executor se ejecute con prioridad elevada.

El objetivo de esta estrategia es doble: mejorar la regularidad temporal de las callbacks periódicas y reducir el tiempo efectivo de ejecución cuando el sistema está cargado. La figura 7.31 resume ambos beneficios.

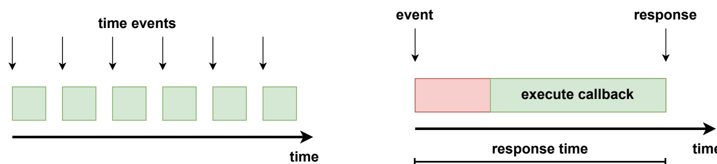


Figura 7.31: Beneficios esperados de la estrategia 1: mejorar la regularidad temporal y reducir el tiempo de respuesta.

La figura 7.32 muestra el tiempo entre activaciones de los temporizadores del nodo no crítico y del nodo crítico. En el nodo no crítico aparece una dispersión mucho mayor: algunas activaciones se retrasan y otras llegan demasiado pronto porque las anteriores quedaron desplazadas. En cambio, el nodo crítico mantiene una cadencia mucho más constante

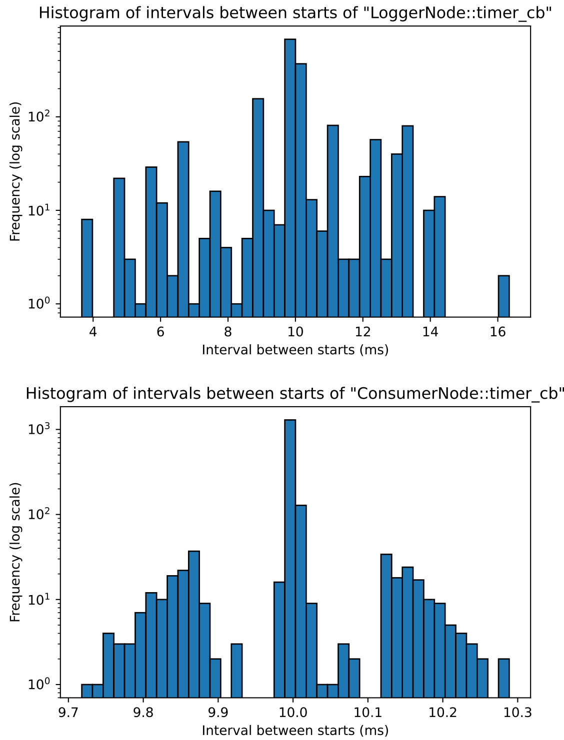


Figura 7.32: Estrategia 1: histograma del tiempo entre ejecuciones de timers en nodo no crítico vs. crítico (adaptado de [2]).

incluso bajo carga. La estrategia no elimina la variabilidad del sistema completo; la desplaza fuera de la parte que no puede degradarse.

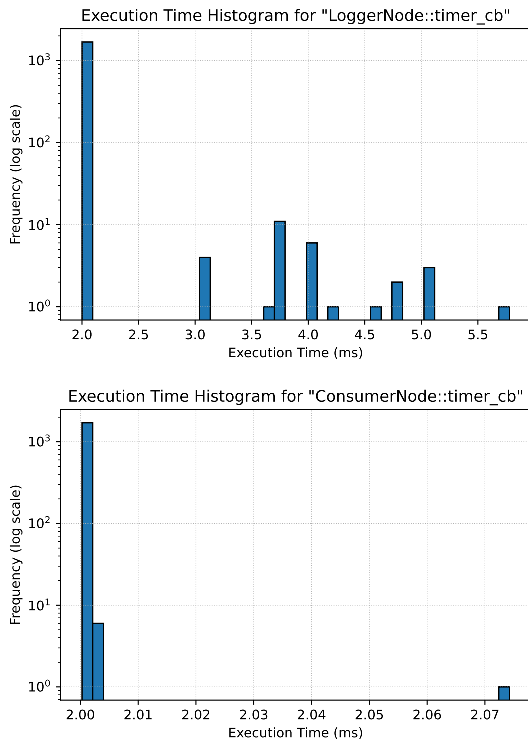


Figura 7.33: Estrategia 1: histograma del tiempo de ejecución de timers (adaptado de [2]).

La figura 7.33 completa la lectura anterior mostrando que también mejora el tiempo efectivo de ejecución de las callbacks críticas: una callback que idealmente debería durar 2 ms puede degradarse mucho en un hilo normal, mientras que en el hilo de tiempo real permanece muy cerca de

su valor esperado. La limitación de esta estrategia es también evidente: si dentro de un mismo nodo conviven callbacks críticas y no críticas, separar solo por nodos resulta demasiado grueso.

Estrategia 2: callback groups en el mismo nodo con distinta prioridad

Cuando un nodo mezcla callbacks críticas y no críticas, conviene bajar un nivel y trabajar directamente con callback groups. En lugar de mover el nodo entero a un executor de tiempo real, se crea un callback group específico para la callback prioritaria y se deja el resto en el grupo por defecto. Eso permite un control más fino sin romper artificialmente el diseño lógico del nodo.

La figura 7.34 representa precisamente ese caso: el mismo nodo publica callbacks en executors diferentes gracias a la separación por callback groups.

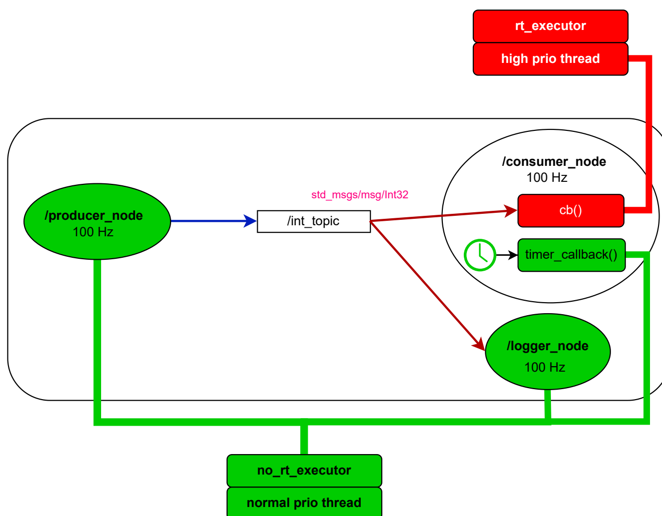


Figura 7.34: Estrategia 2: callbacks del mismo nodo en executors distintos mediante callback groups (adaptado de [2]).

El primer paso en código es modificar el `ConsumerNode` base para declarar y exponer el callback group que queremos ejecutar con prioridad elevada. Como solo cambian unas pocas líneas, tiene más sentido mostrar únicamente esas diferencias:

```
ConsumerNode() : Node("consumer_node")
{
    rt_callback_group_ = this->create_callback_group(
        rclcpp::CallbackGroupType::MutuallyExclusive, false);

    rclcpp::SubscriptionOptions sub_options;
    sub_options.callback_group = rt_callback_group_;

    sub_ = create_subscription<std_msgs::msg::Int32>(
        "int_topic", 100,
        std::bind(&ConsumerNode::cb, this, _1), sub_options);

    timer_ = create_wall_timer(10ms,
        std::bind(&ConsumerNode::timer_cb, this));
}

rclcpp::CallbackGroup::SharedPtr get_rt_callback_group()
{
    return rt_callback_group_;
}

```

```
rclcpp::CallbackGroup::SharedPtr rt_callback_group_;
```

Fragmento C++ 7.12: Estrategia 2: cambios mínimos sobre el `ConsumerNode` para exponer el callback group crítico. label

El detalle importante es el segundo parámetro de `create_callback_group(..., false)`: evita que ese grupo se añada automáticamente al executor cuando incorporamos el nodo completo. Así mantenemos el control explícito sobre qué grupo irá al executor de tiempo real.

El segundo paso es conectar el nodo entero al executor normal y añadir solo el callback group crítico al executor de tiempo real:

```
no_rt_executor.add_node(node_producer);
no_rt_executor.add_node(node_logger);
no_rt_executor.add_node(node_consumer);

rt_executor.add_callback_group(
    node_consumer->get_rt_callback_group(),
    node_consumer->get_node_base_interface());
```

Fragmento C++ 7.13: Estrategia 2: añadir el callback group crítico al executor de tiempo real. label

Este patrón refleja bien la semántica real de ROS 2: la unidad de planificación no es el nodo completo, sino el callback group. Una vez entendido esto, la figura 7.34 deja de ser solo un diagrama y pasa a ser una guía de implementación.

Como en la estrategia 1, lo que se persigue es mejorar regularidad temporal y tiempo de respuesta, pero ahora de una callback concreta dentro de un nodo mixto. La figura 7.35 resume ese objetivo.

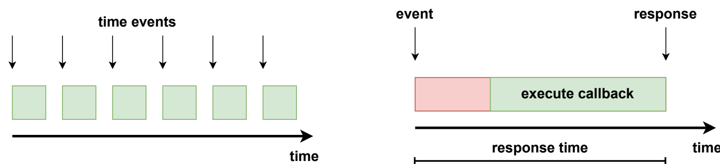


Figura 7.35: Beneficios esperados de la estrategia 2: mejorar la regularidad temporal y reducir el tiempo de respuesta de callbacks seleccionadas.

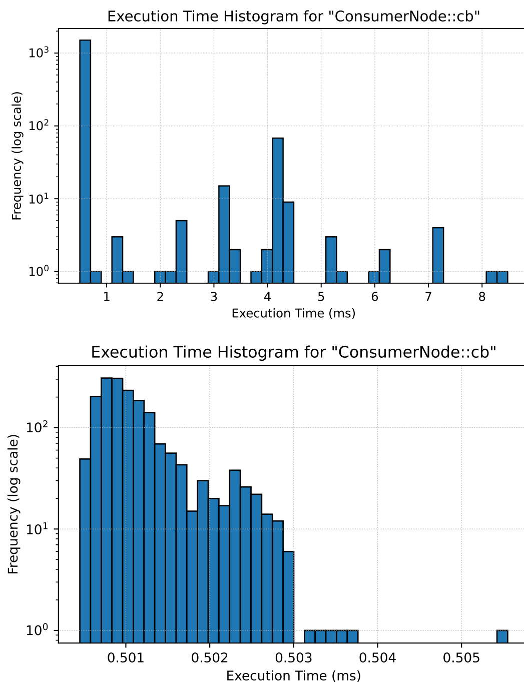


Figura 7.36: Estrategia 2: tiempo de ejecución del callback crítico antes/después de elevar prioridad (adaptado de [2]).

La figura 7.36 compara el tiempo de ejecución de la callback crítica antes y después de moverla al executor de tiempo real. El cambio relevante no es solo la media, sino la desaparición de ejecuciones ocasionalmente muy

lentas. A cambio, esta estrategia exige más disciplina interna: al introducir separación y potencial concurrencia dentro del mismo nodo, también aumentan las posibilidades de contención y errores de sincronización.

Estrategia 3: callback groups críticos distribuidos en varios nodos

En muchos sistemas robóticos, lo verdaderamente crítico no es una callback aislada, sino un flujo completo de datos y procesamiento a través de varios nodos. Si queremos reaccionar ante un obstáculo en una imagen, importa el tiempo total desde que se adquiere la imagen hasta que se envía la orden de frenado. Por eso, la tercera estrategia consiste en identificar el camino crítico extremo a extremo y colocar en callback groups priorizados todas las callbacks que forman parte de ese recorrido.

La figura 7.37 ilustra este caso con un ejemplo de frenado automático. El camino crítico aparece marcado explícitamente y atraviesa varios nodos: producción de datos, detección y actuación. Lo demás, como logs o funciones periódicas de estado, se mantiene fuera del hilo de tiempo real.

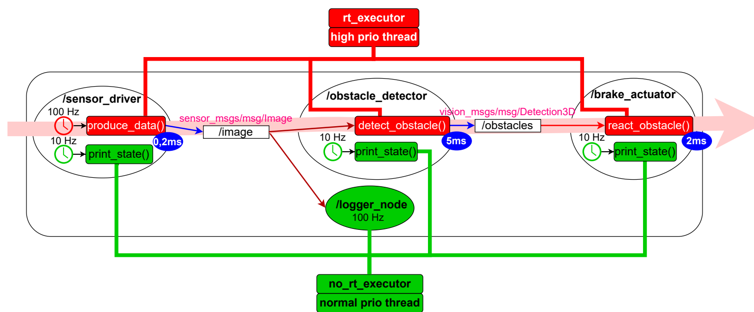


Figura 7.37: Estrategia 3: cadena crítica distribuida (ejemplo tipo freno automático) (adaptado de [2]).

En esta estrategia, la medición se hace con trazas compartidas que marcan el inicio y el final del proceso completo. El código mínimo para hacerlo visible es tan simple como arrancar la traza en el nodo sensor y cerrarla en el actuador:

```cpp
void SensorDriverNode::produce_data()
{
    SHARED_TRACE_START("brake_process");
    waste_time(shared_from_this(), 200us);
    pub_->publish(image_msg);
}

void BrakeActuatorNode::react_obstacle(
    vision_msgs::msg::Detection3D::SharedPtr msg)
{
    waste_time(shared_from_this(), 2ms);
    SHARED_TRACE_END("brake_process");
}
```

Fragmento C++ 7.14: Estrategia 3: medir la latencia extremo a extremo del camino crítico. label

Cada nodo del camino crítico necesita además su propio callback group priorizado. El siguiente ejemplo, tomado del detector de obstáculos, muestra el patrón:

```cpp
ObstacleDetectorNode() : Node("obstacle_detector")
{
    rt_callback_group_ = create_callback_group(
        rclcpp::CallbackGroupType::MutuallyExclusive, false);

    rclcpp::SubscriptionOptions sub_options;
    sub_options.callback_group = rt_callback_group_;
```

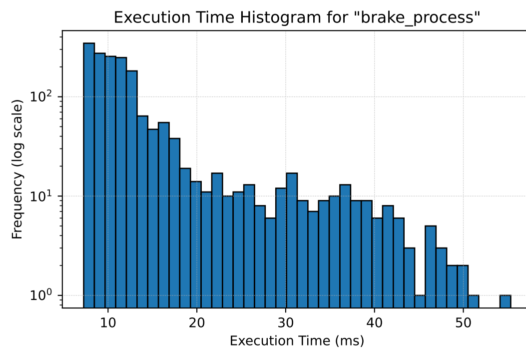
```cpp

    sub_ = create_subscription<sensor_msgs::msg::Image>(
        "image", 100,
        std::bind(&ObstacleDetectorNode::detect_obstacle, this, _1),
        sub_options);

    timer_state_ = create_wall_timer(
        100ms, std::bind(&ObstacleDetectorNode::print_state, this));
}
```

Fragmento C++ 7.15: Estrategia 3: callback group crítico dentro de un nodo del camino extremo a extremo. label

Finalmente, la asignación a executors se hace mezclando nodos completos en el executor normal y callback groups críticos en el executor de tiempo real:

```cpp
rclcpp::executors::SingleThreadedExecutor no_rt_executor;
rclcpp::executors::MultiThreadedExecutor rt_executor(
    rclcpp::ExecutorOptions(), 3);

no_rt_executor.add_node(node_sensor_driver);
no_rt_executor.add_node(node_obstacle_detector);
no_rt_executor.add_node(node_logger);
no_rt_executor.add_node(node_brake_actuator);

rt_executor.add_callback_group(
    node_sensor_driver->get_rt_callback_group(),
    node_sensor_driver->get_node_base_interface());
rt_executor.add_callback_group(
    node_obstacle_detector->get_rt_callback_group(),
    node_obstacle_detector->get_node_base_interface());
rt_executor.add_callback_group(
    node_brake_actuator->get_rt_callback_group(),
    node_brake_actuator->get_node_base_interface());
```

Fragmento C++ 7.16: Estrategia 3: distribuir callback groups críticos de varios nodos en un executor de tiempo real. label



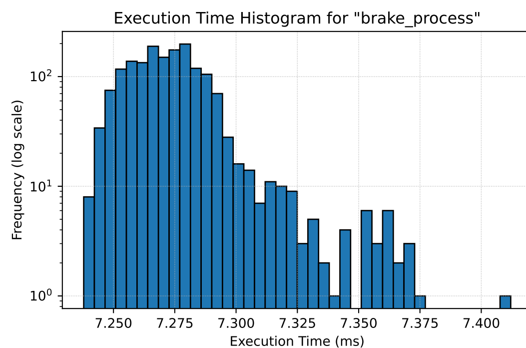


Figura 7.38: Estrategia 3: latencia extremo-a-extremo antes/después de ejecutar la cadena crítica en tiempo real (adaptado de [2]).

Este es el punto en el que el tiempo real deja de ser una propiedad local de un nodo y pasa a ser una propiedad del flujo completo. La estrategia exige identificar qué callbacks forman parte del camino crítico y tratarlas como una unidad de diseño y de medición.

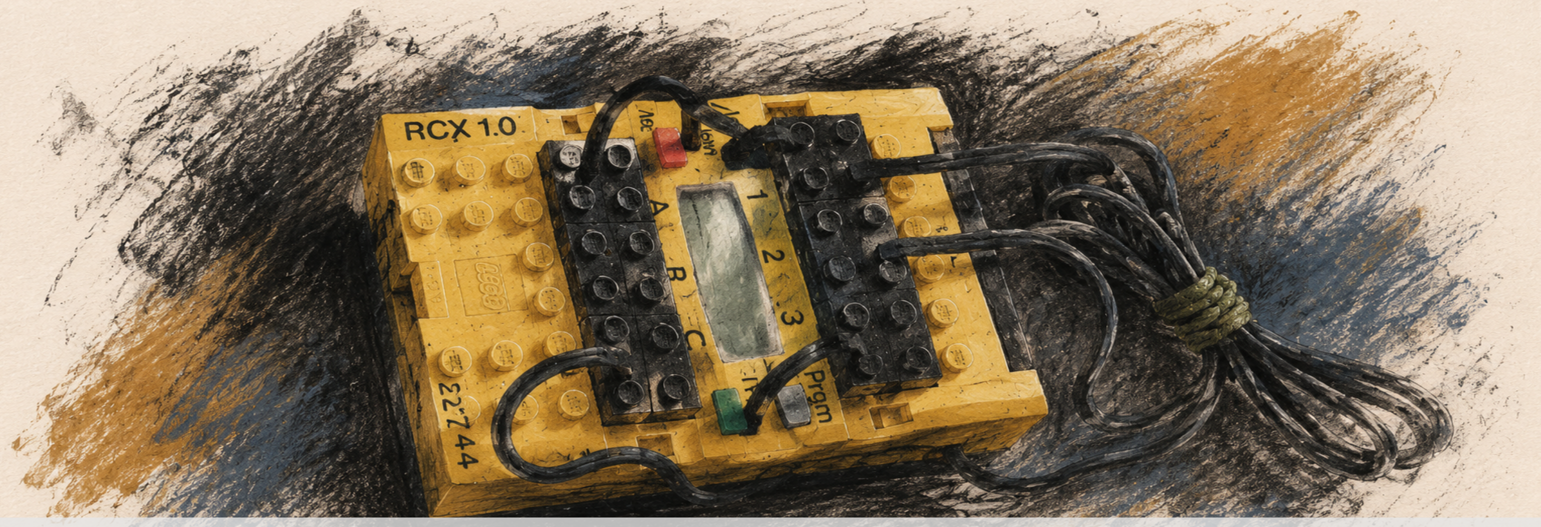
La figura 7.38 muestra el tiempo total desde la percepción hasta la

reacción. Es la figura decisiva de esta estrategia, porque deja ver con claridad que optimizar callbacks individuales no basta: lo que importa es el comportamiento del recorrido completo. En el caso no priorizado, la latencia extremo a extremo puede dispararse de forma inaceptable; al aplicar la estrategia, el sistema queda acotado mucho más cerca del mínimo impuesto por el propio tiempo de cómputo.

En esta estrategia, por tanto, el beneficio principal no es solo mejorar la regularidad de una callback o reducir la dispersión de un temporizador, sino reducir la latencia extremo a extremo del proceso crítico.

La lección común a las tres estrategias es que el tiempo real en ROS 2 no aparece por accidente. Hay que diseñarlo explícitamente: decidir qué es crítico, dónde vive esa criticidad, en qué executor se ejecuta y con qué prioridad corre el hilo que lo soporta.

Proyectos de Arquitecturas Software para Robots



1 Entorno de desarrollo en ROS 2

Este ejercicio se centra en disponer de un entorno de desarrollo funcional antes de comenzar el trabajo arquitectónico propiamente dicho. No se persigue todavía el diseño ni la implementación de arquitecturas software, sino asegurar que las herramientas básicas están correctamente instaladas y comprendidas.

Desde el punto de vista técnico, en este ejercicio se trabajará en:

- Crear y organizar un workspace de ROS 2.
- Activar correctamente el entorno de ROS 2 y del workspace.
- Clonar un repositorio con paquetes ROS 2 en `src`.
- Resolver dependencias con `rosdep`.
- Compilar con `colcon` y comprender la estructura `build` /`install`/`log`.
- Lanzar un sistema existente (simulación) y observarlo mediante herramientas de introspección.
- Inspeccionar nodos, topics, tipos de mensajes, frecuencias y parámetros con `ros2cli`.

Estas competencias serán imprescindibles para todos los ejercicios posteriores. Este ejercicio no introduce programación desde cero ni modificación de código existente.

A diferencia de los ejercicios integradores posteriores, este ejercicio tiene un carácter deliberadamente guiado. El objetivo es consolidar el entorno de trabajo y las herramientas mínimas antes de abordar ejercicios con mayor autonomía de diseño.

Este ejercicio se apoya principalmente en los conceptos introducidos en el Capítulo 1 de la Parte I, en particular en la visión de ROS 2 como infraestructura para sistemas robóticos y en la organización básica de workspaces, paquetes y nodos.

No se requiere todavía una comprensión profunda de modelos de ejecución, concurrencia o arquitectura de componentes. El objetivo es empezar a reconocer estos elementos en sistemas reales, aunque aún no los diseñe ni implemente.

El ejercicio puede abordarse al comienzo del libro o durante la primera semana, de forma guiada, para garantizar un punto de partida común en el recorrido de aprendizaje.

1.1 Objetivo **138**
1.2 Guión de desarrollo . . . **138**
1.3 Teoría y herramientas para el ejercicio **139**
Workspace, underlay y overlay 139
Activación del entorno y persistencia 140
Compilación del workspace y opción `-symlink-install` 140
Nodos y su ejecución: `ros2 run` vs `ros2 launch` 141
Interfaces y parámetros . . 141
Herramientas de línea de comandos (ros2cli) 141
Instalación y ejecución del simulador del Kobuki . . . 145
1.4 Errores comunes y resolución rápida **146**
Olvido de activar el entorno 146
Dependencias no instaladas 147
Confusión de terminales . 147

1.1. Objetivo

El objetivo de este ejercicio es verificar que el entorno de desarrollo de ROS 2 está correctamente configurado y que es posible ejecutar y observar un sistema robótico básico ya existente.

Al finalizar el ejercicio, deberá ser posible lanzar nodos de ROS 2, interactuar con ellos mediante herramientas estándar y comprender de forma preliminar qué tipos de información circulan en un sistema robótico real.

Se trata, por tanto, de un ejercicio guiado orientado a fijar una base común de trabajo para el resto del libro.

1.2. Guión de desarrollo

El desarrollo del ejercicio se estructura como una secuencia guiada de pasos que deben seguirse de forma ordenada. Deben ejecutarse exactamente las acciones indicadas en los apartados siguientes. La explicación y el contexto de cada herramienta se proporciona en la Sección 1.3.

Paso 0: verificación de instalación y entorno

1. Abrir una terminal y verificar que ROS 2 está instalado.
2. Identificar la distribución activa (por ejemplo, `rolling`, `jazzy` u otra).
3. Activar el entorno base de ROS 2.

Paso 1: creación de un workspace de trabajo

1. Crear un directorio de workspace con la estructura estándar `src`.
2. Acceder al directorio raíz del workspace.

Paso 2: clonación del repositorio del Kobuki

1. Clonar el repositorio `https://github.com/IntelligentRoboticsLabs/ASR-Software` en el directorio `src`.
2. Verificar que se han incorporado paquetes ROS 2.

Paso 3: compilación del workspace con colcon

1. Compilar el workspace con `colcon build`.
2. Verificar la generación de los directorios `build`, `install` y `log`.

Paso 4: activación del workspace

1. Activar el entorno generado por el workspace mediante el script `install/setup.bash`.
2. Verificar que los paquetes del workspace son visibles para ROS 2.

Paso 5: lanzamiento de la simulación

1. Lanzar el sistema de simulación del Kobuki usando `ros2 launch`.
2. Mantener el sistema en ejecución en una terminal dedicada.

Paso 6: análisis del sistema con ros2cli

En una segunda terminal (con el entorno correctamente activado), debe:

1. Listar nodos en ejecución.
2. Identificar topics relevantes (sensores, odometría, comandos).
3. Inspeccionar el tipo de mensajes en los topics principales.
4. Observar mensajes en tiempo real en al menos dos topics (uno sensorial y uno de estado).
5. Medir la frecuencia de publicación de un topic (por ejemplo, odometría o láser).
6. Localizar el topic de comandos de velocidad y publicar un comando de prueba (si procede según el sistema).
7. Inspeccionar el estado del sistema y visualizar los sensores del robot con RViz2.

Paso 7: dibujar el diagrama de procesos, nodos y topics

Debe elaborarse un diagrama que represente los nodos en ejecución, su agrupamiento en procesos, y los topics que utilizan para comunicarse. Para ello se usará la herramienta web `https://app.diagrams.net/`.

El uso de esta herramienta es sencillo y se puede aprender en pocos minutos de manera autónoma.

1.3. Teoría y herramientas para el ejercicio

En este ejercicio se trabajará por primera vez con un entorno de desarrollo completo en ROS 2. El objetivo de esta sección es proporcionar el contexto técnico necesario para comprender qué está haciendo en cada paso, y no se limite a ejecutar comandos de forma mecánica.

Workspace, underlay y overlay

ROS 2 organiza el software en *workspaces*. Un workspace es una estructura de directorios que contiene código fuente, resultados de compilación y artefactos de ejecución.

Desde el punto de vista del entorno, es fundamental distinguir entre dos conceptos:

- **Underlay**: es el conjunto de paquetes ROS 2 ya instalados en el sistema, normalmente mediante paquetes binarios. En Ubuntu 24.04 con ROS 2 Jazzy, el underlay se encuentra en: `/opt/ros/jazzy` (Fig. 1.1).
- **Overlay**: es el workspace del usuario, donde se desarrollan y compilan paquetes propios o de terceros a partir de código fuente. Este overlay se superpone al underlay.

Cuando se activa un overlay correctamente, los paquetes compilados en él *tienen prioridad* sobre los paquetes del underlay con el mismo nombre. Este mecanismo permite modificar o extender funcionalidad sin tocar la instalación base del sistema. Activar un overlay que se ha compilado con un underlay hace que, al activar el overlay, también se active automáticamente el underlay.

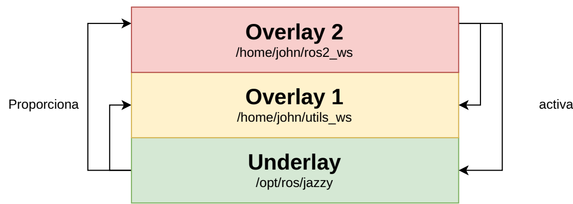


Figura 1.1: Estructura de underlay y overlay en ROS 2.

Activación del entorno y persistencia

Para poder usar ROS 2 es necesario *activar* el entorno, lo que equivale a configurar una serie de variables de entorno (PATH, LD_LIBRARY_PATH, AMENT_PREFIX_PATH, etc.).

La activación del underlay se realiza mediante:

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

Tras compilar un workspace, también debe activarse el overlay:

```
source install/setup.bash
```

Para evitar tener que ejecutar estos comandos manualmente cada vez que se abre una terminal, es habitual añadirlos al fichero `$HOME/.bashrc`, de modo que la activación sea persistente:

```
echo "source /opt/ros/jazzy/setup.bash" >> ~/.bashrc echo "source ~/
     ros2_ws/install/setup.bash" >>
~/.bashrc
```

A partir de ese momento, cualquier nueva terminal tendrá ROS 2 y el workspace de trabajo correctamente configurados.

Compilación del workspace y opción `-symlink-install`

La compilación del workspace se realiza con la herramienta `colcon`, siempre desde el directorio raíz del workspace:

```
colcon build --symlink-install
```

La opción `-symlink-install` indica que los ficheros generados en el directorio `install` no se copian físicamente, sino que se crean enlaces simbólicos a los ficheros originales.

Esta opción:

- Ahorra espacio en disco.
- Permite modificar ciertos ficheros (por ejemplo, scripts Python o ficheros de configuración) sin necesidad de recompilar.

En este libro se recomienda usar siempre `-symlink-install`.

Nodos y su ejecución: `ros2 run` vs `ros2 launch`

La unidad básica de ejecución en ROS 2 es el *nodo*. Un nodo es un proceso que realiza una función concreta dentro del sistema robótico.

Ejecución directa de nodos Para ejecutar un nodo individual se utiliza el comando:

```
ros2 run <paquete> <ejecutable>
```

Este comando localiza el ejecutable dentro del workspace activo (underlay u overlay) y lo lanza, sin necesidad de conocer su ruta exacta en el sistema de ficheros.

Ejecución mediante launch files Cuando una aplicación robótica está formada por múltiples nodos, con parámetros, remapeos o configuraciones específicas, se utilizan *launch files*. Estos se ejecutan con:

```
ros2 launch <paquete> <fichero_launch.py>
```

La diferencia fundamental es que:

- `ros2 run` lanza un único nodo.
- `ros2 launch` lanza uno o varios nodos coordinados, normalmente con parámetros, remapeos y dependencias entre ellos.

Interfaces y parámetros

Antes de usar los comandos de introspección de ROS 2 es importante aclarar algunos conceptos:

- **Interfaces**: en ROS 2 engloban mensajes, servicios y acciones. Definen los tipos de datos con los que los nodos se comunican.
- **Parámetros**: son valores configurables asociados a un nodo, que permiten modificar su comportamiento sin cambiar el código.

ROS 2 proporciona herramientas para inspeccionar estos elementos en tiempo de ejecución, lo que resulta esencial para comprender sistemas complejos.

Herramientas de línea de comandos (ros2cli)

ROS 2 proporciona una interfaz de línea de comandos extensible denominada `ros2cli`, que permite inspeccionar, depurar y controlar un sistema robótico en tiempo de ejecución.

El comando principal es `ros2`, y su sintaxis general es:

```
ros2 <comando> <verbo> [opciones]
```

Cada `comando` corresponde a un dominio del sistema (nodos, topics, interfaces, parámetros, etc.), y cada `verbo` indica la acción concreta que se desea realizar.

Una característica fundamental de `ros2cli` es que soporta **autocompletado con la tecla TAB**, lo que permite explorar el sistema de forma interactiva y descubrir opciones sin necesidad de memorizarlas.

Paquetes y ejecutables

Antes de ejecutar nodos, es importante comprender cómo se organizan los programas en ROS 2.

Un **paquete** es la unidad básica de distribución de software en ROS 2. Un paquete puede contener uno o varios ejecutables (nodos), librerías, interfaces y ficheros de configuración.

Para listar todos los paquetes disponibles en el entorno activo (underlay y overlay), se utiliza:

```
ros2 pkg list
```

Para ver los ejecutables que proporciona un paquete concreto:

```
ros2 pkg executables <nombre_del_paquete>
```

Este comando resulta especialmente útil para descubrir qué nodos se pueden ejecutar sin necesidad de consultar la documentación o el código fuente.

Nodos

Un **nodo** es una entidad de ejecución que realiza una función concreta dentro del sistema robótico. Los nodos se comunican entre sí mediante topics, servicios y acciones.

Para listar los nodos que están activos en un sistema en ejecución:

```
ros2 node list
```

Para obtener información detallada de un nodo concreto:

```
ros2 node info <nombre_del_nodo>
```

Este comando muestra, entre otros datos:

- Topics a los que el nodo se suscribe.
- Topics que publica.
- Servicios y acciones que ofrece o utiliza.
- Parámetros asociados.

Es una herramienta clave para comprender el grafo de computación del sistema.

Topics

Los **topics** son canales de comunicación asíncronos usados para el intercambio continuo de datos entre nodos.

Para listar todos los topics activos:

```
ros2 topic list
```

Para conocer el tipo de mensaje asociado a un topic:

```
ros2 topic type <nombre_del_topic>
```

Para mostrar información más detallada sobre un topic:

```
ros2 topic info <nombre_del_topic>
```

Durante el ejercicio, será habitual inspeccionar los datos que circulan por un topic. Para ello se utiliza:

```
ros2 topic echo <nombre_del_topic>
```

Este comando imprime por pantalla los mensajes que se publican, lo que permite verificar rápidamente si un nodo está funcionando correctamente.

Interfaces

Las **interfaces** definen los tipos de datos que se utilizan en ROS 2. Incluyen:

- Mensajes (`msg`)
- Servicios (`srv`)
- Acciones (`action`)

Para listar todas las interfaces disponibles en el sistema:

```
ros2 interface list
```

Para mostrar la definición de una interfaz concreta:

```
ros2 interface show geometry_msgs/msg/Twist
```

Este comando es fundamental para entender qué campos tiene un mensaje y cómo deben interpretarse los datos que se observan en un topic.

Parámetros

Los **parámetros** permiten configurar el comportamiento de un nodo en tiempo de ejecución, sin modificar el código fuente.

Para listar los parámetros de un nodo:

```
ros2 param list <nombre_del_nodo>
```

Para consultar el valor de un parámetro concreto:

```
ros2 param get <nombre_del_nodo> <nombre_del_parametro>
```

Para modificar el valor de un parámetro en tiempo de ejecución:

```
ros2 param set <nombre_del_nodo> <nombre_del_parametro> <valor>
```

En sistemas simulados, es habitual encontrar parámetros como `use_sim_time`, que indican al nodo que utilice el tiempo proporcionado por el simulador en lugar del reloj del sistema.

Launch y run

Aunque ya se han introducido anteriormente, desde el punto de vista de `ros2cli` conviene remarcar que:

- `ros2 run` se utiliza para ejecutar un único nodo.
- `ros2 launch` se utiliza para ejecutar uno o varios nodos coordinados mediante un fichero `launch`.

Para listar los ficheros `launch` disponibles en un paquete:

```
ros2 launch <nombre_del_paquete> --show-args
```

Este comando permite además descubrir qué argumentos y parámetros admite un fichero de lanzamiento, algo especialmente útil al trabajar con simuladores y sistemas complejos.

Resumen operativo para el ejercicio

Durante este ejercicio, conviene familiarizarse al menos con los siguientes comandos:

- `ros2 pkg list`, `ros2 pkg executables`
- `ros2 run`, `ros2 launch`
- `ros2 node list`, `ros2 node info`
- `ros2 topic list`, `ros2 topic info`, `ros2 topic echo`
- `ros2 interface show`
- `ros2 param list`, `ros2 param get`

Estos comandos constituyen la base para analizar y comprender cualquier sistema robótico desarrollado con ROS 2.

Instalación y ejecución del simulador del Kobuki

En este ejercicio se utilizará un simulador del robot Kobuki. El simulador se distribuye como un conjunto de paquetes ROS 2 que deben ser compilados dentro del workspace de trabajo.

Obtención del código fuente

El primer paso consiste en clonar el repositorio que contiene el playground del Kobuki dentro del directorio `src` del workspace:

```
git clone https://github.com/IntelligentRoboticsLabs/ASR-Software.
    git
```

Este repositorio contiene los paquetes principales del simulador, así como un fichero que describe dependencias externas necesarias para su funcionamiento.

Compilación del workspace

Con todas las dependencias disponibles, se compila el workspace:

```
colcon build --symlink-install
```

Tal y como se ha indicado anteriormente, la opción `-symlink-install` permite trabajar de forma más ágil durante el desarrollo y es la recomendada para este libro.

Recarga del workspace

Tras la compilación, los nuevos paquetes **no están automáticamente disponibles** en el entorno actual. Esto es un punto crítico que suele generar confusión al principio.

Para que los paquetes recién compilados puedan utilizarse, es necesario **recargar el workspace**, lo cual se puede hacer de dos maneras:

- Ejecutando explícitamente:

  ```
  source install/setup.bash
  ```

- Cerrando la terminal actual y abriendo una nueva (si el workspace se activa desde el `.bashrc`).

Si este paso no se realiza, ROS 2 no podrá localizar los nuevos paquetes y los comandos `ros2 run` o `ros2 launch` fallarán indicando que el paquete no existe.

Ejecución del simulador

Una vez recargado el entorno, el simulador del Kobuki puede lanzarse mediante un fichero `launch`:

```
ros2 launch kobuki simulation.launch.py
```

Este comando inicia:

- El simulador.
- El modelo del robot Kobuki.
- Los nodos necesarios para la navegación básica y los sensores.

El simulador incluye una interfaz gráfica opcional. Si se desea ejecutar sin GUI, por ejemplo en máquinas con recursos limitados, puede utilizarse:

```
ros2 launch kobuki simulation.launch.py gui:=false
```

Visualización con RViz2

Para inspeccionar el estado del sistema y visualizar los sensores del robot, se utilizará `rviz2`, la herramienta estándar de visualización en ROS 2.

RViz2 se puede lanzar en una terminal independiente:

```
rviz2
```

Una vez abierto RViz2, conviene:

- Fijar el `Fixed Frame` al frame de referencia adecuado (por ejemplo, `map` o `odom`).
- Añadir una visualización de tipo `LaserScan`.
- Seleccionar el topic correspondiente al láser del Kobuki (por ejemplo, `/scan`).

Si todo está correctamente configurado, se observarán los datos del sensor láser del robot actualizándose en tiempo real, lo que confirma que el simulador y el sistema ROS 2 están funcionando correctamente.

1.4. Errores comunes y resolución rápida

Olvido de activar el entorno

Síntoma: `ros2` no encuentra paquetes o launch files, o `ros2 node list` aparece vacío cuando debería haber nodos.

Acción correctiva:

```
source /opt/ros/$ROS_DISTRO/setup.bash cd ~/ros2_ws source install/
    setup.bash
```

Dependencias no instaladas

Síntoma: errores durante `colcon build` indicando paquetes faltantes o librerías no encontradas.

Acción correctiva:

```
cd ~/ros2_ws rosdep update rosdep install --from-paths src --ignore-
    src -r -y
```

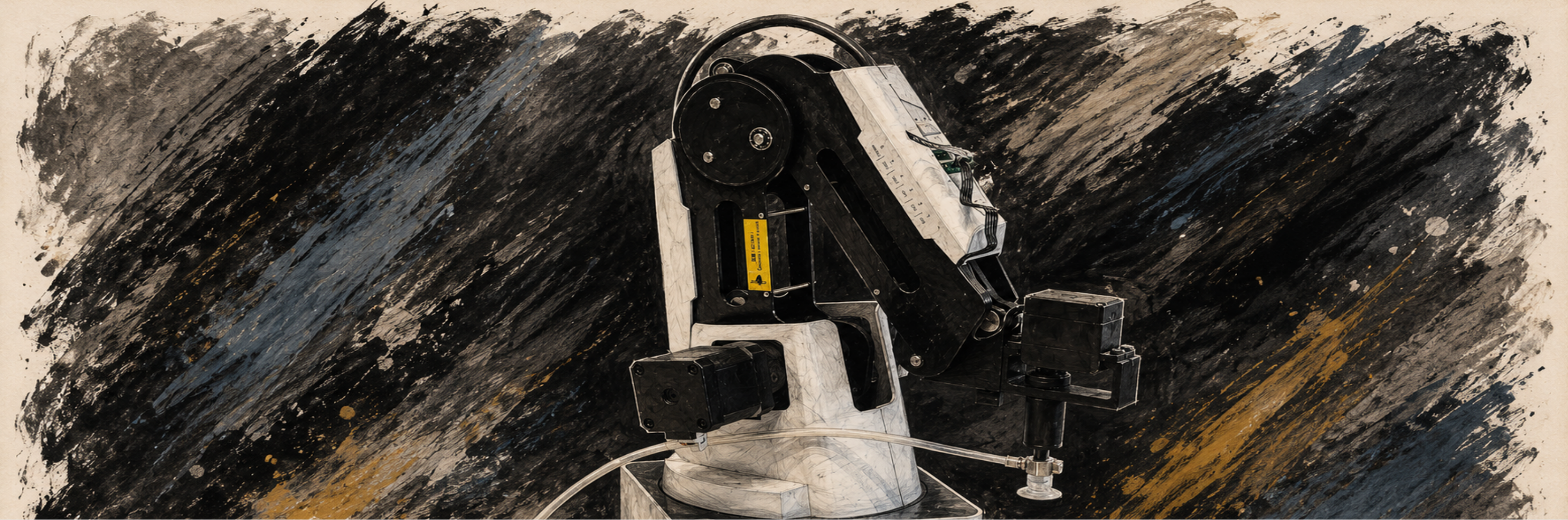
Confusión de terminales

Síntoma: la simulación está en ejecución en una terminal, pero las herramientas de introspección en otra terminal no ven nodos ni topics.

Causa probable: en la segunda terminal no se activó el mismo entorno.

Acción correctiva: repetir la activación de ROS 2 y del workspace en esa terminal.

```
source /opt/ros/$ROS_DISTRO/setup.bash cd ~/ros2_ws source install/
    setup.bash
```



2 Ejecución reactiva dirigida por eventos

2.1 Objetivo **149**
2.2 Guión de desarrollo . . . **149**
2.3 Teoría y herramientas para el ejercicio **151**
Estructura del paquete `ch2_examples` 151
El nodo `LoggerNode`: timers y logging 152
El nodo `PublisherNode`: publicación periódica . . . 153
El nodo `SubscriberNode`: callbacks por evento . . . 154
Varios nodos en el mismo proceso: el papel del *executor* 154

Este ejercicio se centra en implementar un primer nodo completo en ROS 2 basado en un modelo de ejecución reactivo: el comportamiento del robot emergerá de eventos del entorno (pulsación del bumper) y de un temporizador periódico.

Desde el punto de vista técnico, en este ejercicio se trabajará en:

- Identificación de *topics* relevantes y análisis de tipos de mensajes.
- Implementación de un **suscriptor** y programación en callbacks.
- Implementación de un **publicador** de velocidades (`cmd_vel`).
- Diseño de un lazo de control con **timer** a 10 Hz.
- Gestión de estado interno mínimo (estado del bumper) en un nodo concurrente.
- Representación del diseño mediante un esquema (draw.io) de entradas, salidas y eventos.

Desde el punto de vista arquitectónico, el ejercicio refuerza el flujo de datos y el diseño dirigido por eventos: un callback actualiza el estado, y un timer genera la acción de control.

Al igual que en los primeros ejercicios del bloque, se trata de un ejercicio guiado. El énfasis no está en abrir decisiones amplias de diseño, sino en comprender con claridad un patrón básico de ejecución reactiva en ROS 2.

Este ejercicio se apoya principalmente en los conceptos introducidos en el Capítulo 2 de la Parte I, en particular:

- modelos de ejecución reactivos y arquitectura dirigida por eventos,
- nodos como unidad de ejecución en ROS 2,
- publicación/subscripción y contratos de mensajes,
- diseño de topics y mensajes como interfaces públicas.

El objetivo no es obtener un comportamiento autónomo complejo, sino comprender el *pipeline* mínimo “evento → estado → control → actuación” y las herramientas de introspección que permiten validarlo.

2.1. Objetivo

El objetivo del ejercicio es implementar un **teleoperador reactivo** usando el bumper del Kobuki.

El ejercicio consiste en programar un nodo que hace que el robot empiece **parado**. A partir de ese momento, el nodo publicará comandos de velocidad según el estado del bumper:

- Si el bumper **central** está pulsado, el robot **avanza**.
- Si el bumper central **no** está pulsado, el robot **se para** (velocidad cero), salvo que se indique giro.
- Si está pulsado el bumper **izquierdo**, el robot **gira a la izquierda** *sin avanzar*.
- Si está pulsado el bumper **derecho**, el robot **gira a la derecha** *sin avanzar*.

Regla de prioridad (seguridad y determinismo) Si hay pulsaciones simultáneas, se recomienda:

- priorizar el giro lateral frente a avanzar (si hay lateral, girar),
- si izquierdo y derecho están pulsados a la vez, parar (no girar ni avanzar).

El comportamiento debe ser observable en simulación (o en el robot real) y debe mantenerse estable: el nodo nunca debe dejar el robot "sin comando" en un estado indeterminado.

Como ejercicio guiado, el recorrido está pensado para construir el comportamiento paso a paso, aislando primero la percepción del evento, después el estado interno y finalmente la acción de control.

2.2. Guión de desarrollo

El desarrollo del ejercicio se estructura como una secuencia guiada de pasos.

Paso 1: identificar topics y tipos de mensajes

Antes de programar, deben identificarse:

- el topic del bumper del Kobuki (evento de pulsación),
- el topic de comandos de velocidad (habitualmente `/cmd_vel`),
- el tipo de mensaje en ambos topics.

Para ello, se recomienda usar `ros2cli`:

```
ros2 topic list ros2 topic info <topic_bumper> ros2 interface show <
     tipo_del_bumper> ros2 topic info
/cmd_vel ros2 interface show geometry_msgs/msg/Twist
```

Paso 2: suscriptor que muestre cambios del bumper

Implementar un nodo con un suscriptor al topic del bumper que imprima con `RCLCPP_INFO` **solo cuando cambia el estado**.

- Debe mostrarse qué bumper (izq/centro/der) cambia y si se pulsa o se suelta.
- La salida debe ser clara para depurar: que pueda verse una secuencia de eventos.

Nota para simulación. Si el simulador no genera eventos de bumper de forma directa, puede crearse un pequeño publicador de prueba o usar `ros2 topic pub` para inyectar eventos de bumper y validar el callback.

Paso 3: lazo de control con timer a 10 Hz

Añadir al nodo un temporizador a 10 Hz que imprima el estado actual del bumper (el último estado conocido) aunque no haya eventos nuevos.

El objetivo de este paso es distinguir:

- el **evento** (callback del bumper) que actualiza estado,
- el **ciclo de control** periódico que consulta ese estado.

Paso 4: generación y publicación de velocidades

Modificar el lazo de control para que, en lugar de solo imprimir, publique las velocidades correctas en `/cmd_vel` (`geometry_msgs/msg/Twist`) según las reglas del objetivo.

Se recomienda parametrizar al menos:

- velocidad lineal de avance v,
- velocidad angular de giro ω.

Paso 5: esquema de diseño (draw.io)

Debe dibujarse un esquema del diseño de la aplicación, mostrando:

- el nodo desarrollado,
- sus entradas (topic del bumper),
- sus salidas (`/cmd_vel`),
- y los eventos internos (callback del bumper y evento del timer a 10 Hz).

Se recomienda usar la herramienta web `https://app.diagrams.net/` y exportar el resultado como PDF o SVG.

2.3. Teoría y herramientas para el ejercicio

Esta sección introduce las piezas mínimas de ROS 2 en C++ con `rclcpp` (nodos, *publishers*, *subscribers*, temporizadores y ejecutores), usando el paquete `ch2_examples` incluido en el repositorio. Es importante entender el **modelo de ejecución**: los nodos no "corren" por sí mismos, sino que un **executor** despacha callbacks cuando hay eventos (mensajes recibidos, timers, etc.).

Un **executor** es el componente de `rclcpp` que implementa el bucle de espera y *despacho* de callbacks: monitoriza las fuentes de eventos (suscripciones, timers, etc.) y ejecuta sus funciones asociadas cuando corresponde. Intuitivamente, es el "planificador" que decide **cuándo** se ejecuta cada callback (y, según el tipo de executor, en cuántos hilos). En este ejercicio, usar un executor monohilo ayuda a mantener el comportamiento simple y determinista.

Cómo compilar y ejecutar los ejemplos (referencia rápida) En un workspace ROS 2 estándar, estos comandos suelen ser suficientes:

```
colcon build --packages-select ch2_examples source install/setup.
    bash ros2 run ch2_examples logger
```

En el ejercicio, usaremos herramientas de introspección para comprobar topics, tipos y flujo:

```
ros2 node list ros2 topic list ros2 topic info /counter ros2 topic
    echo /counter
```

Estructura del paquete `ch2_examples`

Este paquete está organizado de forma "canónica" para C++ en ROS 2:

- `package.xml`: metadatos y dependencias del paquete.
- `CMakeLists.txt`: cómo se compila y se instala (targets, librerías, ejecutables).
- `include/ch2_examples/*.hpp`: cabeceras públicas (API del paquete).
- `src/ch2_examples/*.cpp`: implementación de los nodos (lógica).
- `src/*_main.cpp`: programas `main` que despliegan (instancian) los nodos.

Una idea clave para el ejercicio: **separa el nodo (clase) del `main`**. Así puedes reutilizar el nodo en otros ejecutables (por ejemplo, varios nodos en el mismo proceso) sin duplicar la lógica.

`package.xml`: dependencias del paquete

En ROS 2, las dependencias se declaran en `package.xml`. Para `ch2_examples` las relevantes son `rclcpp` (API C++) y `std_msgs` (tipos de mensajes estándar usados en el ejemplo):

```
<buildtool_depend>ament_cmake</buildtool_depend>

<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

Fragmento C++ 2.1: Fragmento de package.xml: dependencias principales

Por qué es importante: si en este ejercicio se usan `geometry_msgs` (para `Twist`) o el mensaje del bumper (por ejemplo `kobuki_ros_interfaces` u otro, según el simulador/robot), tendrás que añadir también esas dependencias.

`CMakeLists.txt`: targets (librería + ejecutables)

En CMake (ament_cmake), lo habitual es:

- localizar dependencias con `find_package(... REQUIRED)`,
- compilar una librería con tus nodos (reutilizable),
- compilar uno o varios ejecutables con `main`,
- instalar targets para poder usar `ros2 run`.

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_library(${PROJECT_NAME}
    src/ch2_examples/LoggerNode.cpp
    src/ch2_examples/PublisherNode.cpp
    src/ch2_examples/SubscriberNode.cpp
)

add_executable(logger src/logger_main.cpp)
target_link_libraries(logger ${PROJECT_NAME})
```

Fragmento C++ 2.2: Fragmento de CMakeLists.txt: librería y ejecutables

Lectura recomendada:

- `add_library`: el "código del nodo" se compila una vez.
- `add_executable`: cada ejecutable elige qué nodos instanciar.
- `install(TARGETS ... RUNTIME DESTINATION lib/${PROJECT_NAME})`: es lo que hace que `ros2 run ch2_examples logger` encuentre el binario.

El nodo `LoggerNode`: timers y logging

Este nodo es el "hola mundo" de `rclcpp`: un temporizador periódico que ejecuta un callback.

```
#include "rclcpp/rclcpp.hpp"

class LoggerNode : public rclcpp::Node
{
public:
    LoggerNode();
    void timer_callback();

private:
    rclcpp::TimerBase::SharedPtr timer_;
    int counter_;
};
```

Fragmento C++ 2.3: LoggerNode.hpp: un nodo con timer y estado interno

La implementación crea un timer con `create_wall_timer` (basado en tiempo real de pared, no tiempo simulado) y registra mensajes con `RCLCPP_INFO`:

¿Qué es el "tiempo real de pared"? Es el tiempo del reloj del sistema (el que usarías para mirar la hora), y avanza "de forma natural" aunque no haya simulación. En ROS 2 suelen convivir tres nociones: *wall time* (reloj del sistema), *steady time* (reloj monótono que no salta si cambia la hora del sistema) y *ROS time* o tiempo simulado (controlado por un simulador vía `/clock`, que puede acelerarse, pausarse o reiniciarse). Usamos `create_wall_timer` aquí porque los ejemplos deben funcionar igual en una terminal sin depender de simulación, y queremos que el periodo sea estable y fácil de interpretar. En proyectos con simulación, a menudo interesa usar timers basados en ROS time para que el comportamiento quede sincronizado con el tiempo del simulador.

```
using namespace std::chrono_literals;

LoggerNode::LoggerNode()
: Node("logger_node")
{
    counter_ = 0;
    timer_ = create_wall_timer(
        500ms, std::bind(&LoggerNode::timer_callback, this));
}

void LoggerNode::timer_callback()
{
    RCLCPP_INFO(get_logger(), "Hello %d", counter_++);
}
```

Fragmento C++ 2.4: LoggerNode.cpp: create_wall_timer + RCLCPP_INFO

Ideas clave (ROS 2)

- `rclcpp::Node`: es la unidad de ejecución; encapsula parámetros, logging, publishers/subscribers, etc.
- `create_wall_timer(periodo, callback)`: registra un evento periódico. No "lanza un hilo" por sí mismo; el timer se dispara cuando el executor lo procesa.
- `RCLCPP_INFO`: logging con niveles. Otros niveles típicos: `RCLCPP_DEBUG`, `RCLCPP_WARN`, `RCLCPP_ERROR`.

Despliegue del nodo: `main` mínimo con `spin`

La clase del nodo no tiene por qué conocer cómo se ejecuta el proceso. El ejecutable (`main`) se encarga de inicializar ROS, crear el nodo y **hacer spin**:

```
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<LoggerNode>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

Fragmento C++ 2.5: logger_main.cpp: init + spin + shutdown

Qué significa `spin`: entra en un bucle del executor que atiende callbacks (timers, mensajes entrantes, etc.) hasta que el proceso termina.

El nodo `PublisherNode`: publicación periódica

Este nodo publica un contador en el topic `counter` con tipo `std_msgs/msg/Int32`. La estructura es la misma: un timer periódico llama a un callback, que actualiza estado y publica.

```
PublisherNode::PublisherNode()
: Node("publisher_node")
{
    publisher_ = create_publisher<std_msgs::msg::Int32>("counter", 10);
    timer_ = create_wall_timer(
        100ms, std::bind(&PublisherNode::timer_callback, this));

    counter_message_.data = 0;
}

void PublisherNode::timer_callback()
{
    RCLCPP_INFO(get_logger(), "Publishing %d", counter_message_.data++);
    publisher_->publish(counter_message_);
}
```

Fragmento C++ 2.6: PublisherNode.cpp: create_publisher + publish

Ideas clave (ROS 2)

- `create_publisher<T>(topic, qos)` crea una salida. El `10` es un perfil QoS sencillo (profundidad del historial) para empezar.
- `publish(msg)` envía un mensaje al middleware. En este ejercicio, la publicación importante será `/cmd_vel` con `geometry_msgs/msg/Twist`.

El nodo `SubscriberNode`: callbacks por evento

El suscriptor se activa cuando **llega un mensaje** al topic `counter`. Esto es exactamente el patrón que usarás con el bumper: el callback actualiza estado interno y el resto del nodo reacciona.

```
SubscriberNode::SubscriberNode()
: Node("subscriber_node")
{
    counter_subscription_ = create_subscription<std_msgs::msg::Int32>(
        "counter", 10,
        std::bind(&SubscriberNode::subscription_callback, this, std::
    placeholders::_1));
}

void SubscriberNode::subscription_callback(const std_msgs::msg::Int32::
    SharedPtr msg)
{
    RCLCPP_INFO(get_logger(), "Received %d", msg->data);
}
```

Fragmento C++ 2.7: SubscriberNode.cpp: create_subscription + callback

Ideas clave (ROS 2)

- Un suscriptor es **reactivo**: no haces un `while` leyendo; registras un callback.
- El argumento `msg` (shared_ptr) contiene los datos del mensaje recibido.
- El executor decide **cuándo** ejecutar el callback. En un executor monohilo, los callbacks se ejecutan secuencialmente.

Varios nodos en el mismo proceso: el papel del *executor*

No es obligatorio ejecutar un nodo por proceso. Puedes instanciar varios nodos y dejarlos bajo el control de un mismo executor. En `pub_sub_main.cpp` se hace con un `SingleThreadedExecutor`:

```cpp
auto pub_node = std::make_shared<PublisherNode>();
auto sub_node = std::make_shared<SubscriberNode>();

rclcpp::executors::SingleThreadedExecutor executor;
executor.add_node(pub_node);
executor.add_node(sub_node);
executor.spin();
```

Fragmento C++ 2.8: pub_sub_main.cpp: dos nodos, un executor

Relación directa con este ejercicio Tu teleoperador reactivo tendrá el mismo patrón estructural:

- **Callback de suscripción** (evento del bumper) → actualiza estado interno (qué bumper está pulsado).
- **Timer a 10 Hz** → lee ese estado y publica un `Twist` en `/cmd_vel`.

Esto separa claramente **evento** (bumper) de **control periódico** (timer), y hace que el robot nunca se quede sin comando.

3 Percepción, seguimiento y evitación

Este ejercicio se centra en diseñar e implementar un sistema robótico capaz de seguir de forma autónoma un objeto/persona detectado con la cámara, mientras evita colisiones con obstáculos detectados mediante un sensor láser.

3.1 Objetivo **157**
3.2 Guión de desarrollo . . . **157**
3.3 Teoría y herramientas para el ejercicio **159**
Launch files y despliegue 159
Configuración mediante parámetros 160
Procesado de `sensor_msgs/msg/LaserScan` 161
Publicación y consulta de TF 162
Procesado de imágenes . . 164
Reconstrucción 3D con imagen de profundidad . 167
Reconstrucción 3D con `PointCloud2` 169

Desde el punto de vista arquitectónico, en este ejercicio se trabajará en:

- Separar percepción, decisión y control en módulos con responsabilidades claras.
- Componer capacidades perceptivas y de actuación mediante flujos de datos explícitos.
- Mantener observabilidad de las salidas intermedias del sistema para validar cada bloque.

Desde el punto de vista técnico, se trabajará en:

- Procesado de `LaserScan` para localizar obstáculos relevantes.
- Publicación de información perceptiva compacta con `vision_msgs`.
- Reconstrucción 3D a partir de profundidad y `CameraInfo`.
- Publicación y consulta de TF propias.
- Control PID y combinación de comportamientos perceptivos y reactivos.

Aunque este ejercicio introduce más componentes que PR1 y PR2, sigue planteado como un ejercicio guiado. El objetivo es consolidar una cadena completa de percepción y control antes de pasar a ejercicios con mayor libertad arquitectónica.

Este ejercicio se apoya directamente en los conceptos de la percepción y su integración en ROS 2:

- Capítulo 3: sensores en ROS 2 (`LaserScan`, `Image`, `CameraInfo`), TF y publicación de información ya procesada (detecciones).

La misión del sistema es seguir un objetivo visual de forma segura. Las tareas incluyen: detección 2D, inferencia 3D, control de orientación y distancia, y evitación reactiva de obstáculos.

3.1. Objetivo

El objetivo del ejercicio es implementar un robot móvil (simulado o real) que siga un objeto/persona detectado con la cámara, manteniéndolo en su campo de visión y aproximándose hasta una distancia objetivo de 1–2 metros, al mismo tiempo que evita obstáculos utilizando información de un sensor láser.

El comportamiento debe ser continuo, estable y observable. El robot debe moverse de forma suave, reaccionar a cambios en la detección y evitar colisiones incluso cuando estas interfieren con el seguimiento.

Este comportamiento representa un salto cualitativo respecto a ejercicios anteriores, al introducir control continuo y composición de capacidades.

El desarrollo propuesto sigue una secuencia guiada de integración, de modo que cada bloque pueda validarse de forma aislada antes de componer el comportamiento completo.

3.2. Guión de desarrollo

El ejercicio se desarrolla de forma incremental. Cada paso debe poder demostrarse de manera aislada (con `ros2 topic echo`, RViz o el simulador) antes de pasar al siguiente.

Paso 1: obstáculo más cercano con láser

En este paso se implementará un nodo que:

- Reciba un `sensor_msgs/msg/LaserScan`.
- Localice el obstáculo más cercano (mínimo rango válido) y calcule su posición 2D en el marco del robot.
- Publique el resultado en `/nearest_obstacle` como `geometry_msgs/msg/PointStamped`.
- Publique además una TF propia asociada al obstáculo más cercano.

Esto implica usar el sistema de TF: el obstáculo se obtiene inicialmente en el marco del sensor (`LaserScan.header.frame_id`) y debe transformarse al marco del robot (por ejemplo, `base_link`) usando la transformación correspondiente al instante `LaserScan.header.stamp`.

Requisitos:

- El `PointStamped` publicado en `/nearest_obstacle` debe estar expresado en el marco del robot (`header.frame_id` igual al marco elegido, p.ej. `base_link`).
- El `header.stamp` debe ser coherente con el `LaserScan` de entrada.
- La TF del obstáculo más cercano debe publicarse con el mismo instante de tiempo (`TransformStamped.header.stamp` coherente con el `LaserScan`) y como marco hijo fijo (por ejemplo, `nearest_obstacle`) respecto al marco del robot (por ejemplo, `base_link`).
- Deben ignorarse rangos inválidos (`NaN`, `Inf` o fuera de `range_min` / `range_max`).
- Si no existe ningún rango válido en el mensaje, no se publicará nada.

- Si no se puede resolver la transformación TF necesaria en ese instante (por ejemplo, porque no está en el buffer), no se publicará nada.

Paso 2: detección 2D de un objeto/persona

En este paso se implementará un nodo de detección 2D que:

- Reciba una imagen `sensor_msgs/msg/Image` (topic a elegir/remapear según el robot o simulador).
- Publique una detección como `vision_msgs/msg/Detection2D` (por ejemplo, en `/detection_2d`).

Puede escogerse el método:

- Segmentación por color en HSV (solución ligera y explicable).
- Deep Learning (por ejemplo, mediante un wrapper tipo `Yolo_ROS`, si está disponible en el entorno).

Regla importante: si no se detecta nada, **no se publica nada**.

Paso 3: detección 3D

A partir de:

- Una `vision_msgs/msg/Detection2D`.
- Una imagen de profundidad (`sensor_msgs/msg/Image`).
- Los parámetros intrínsecos de cámara (`sensor_msgs/msg/CameraInfo`).

en este paso se implementará un nodo que publique:

- Una `vision_msgs/msg/Detection3D` (por ejemplo, en `/detection_3d`).
- Una TF propia asociada al objeto/persona detectado.

Si en un instante dado no hay `Detection2D` (porque el detector 2D no ha publicado), entonces este nodo tampoco publicará `Detection3D`.

Pista: un enfoque mínimo consiste en tomar el centro de la caja 2D, leer su profundidad Z y proyectar a 3D usando los intrínsecos de `CameraInfo`. La TF puede publicarse como un marco hijo (p.ej. `target`) respecto al marco óptico de la cámara o respecto a `base_link`, siempre que se indique claramente qué marcos se usan.

Paso 4: control de orientación hacia la detección

En este paso se implementará un nodo de control que:

- Reciba una `vision_msgs/msg/Detection3D`.
- Genere la velocidad angular necesaria para orientar el robot hacia el objetivo.
- Publique comandos de velocidad (`geometry_msgs/msg/Twist`) en `/cmd_vel`.

Si no hay detección, el robot debe girar hacia un lado hasta encontrarla.

Se recomienda implementar el control angular como un PID sobre el error de orientación (por ejemplo, el ángulo hacia el objetivo en el marco del robot).

Paso 5: control de distancia (1–2 m)

Se ampliará el nodo anterior para que el robot:

- Se acerque al objeto/persona hasta una distancia objetivo (entre 1 y 2 metros).
- Si el objeto/persona se acerca demasiado, el robot retroceda para mantener la distancia.

Paso 6: evitación con el obstáculo más cercano

El robot deberá tener en cuenta `/nearest_obstacle` para evitar colisiones **cuando el obstáculo interfiera en el seguimiento**. En particular:

- La evitación puede inhibir el avance, modificar la velocidad angular o imponer una maniobra reactiva.
- Debe demostrarse que el robot evita colisiones incluso cuando el objetivo visual está "detrás" de un obstáculo.

3.3. Teoría y herramientas para el ejercicio

En este ejercicio aparecerán cuatro herramientas que, en muchos cursos introductorios, no se trabajan con detalle: **archivos launch**, **parámetros**, uso práctico de **`sensor_msgs/msg/LaserScan`** y **TF** (publicación y consulta). En esta sección se introducen con ejemplos reales del repositorio (`ch3_examples`).

Launch files y despliegue

Un **archivo launch** es un programa (normalmente en Python) que describe **cómo lanzar** un conjunto de nodos y acciones en ROS 2: qué ejecutables arrancar, con qué parámetros, qué *remappings* aplicar, qué otros archivos launch incluir, etc. A nivel práctico, un archivo launch permite reemplazar una secuencia larga de comandos de terminal por una descripción reproducible.

Por ejemplo, el archivo launch `ch3_examples/sensors/laser/launch/laser.launch.py` lanza el nodo detector y le pasa un fichero de parámetros, además de remapear el topic de entrada:

```
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    pkg_dir = get_package_share_directory('laser')
    param_file = os.path.join(pkg_dir, 'config', 'params.yaml')

    detector_cmd = Node(
        package='laser',
        executable='obstacle_detector',
        output='screen',
        parameters=[param_file],
        remappings=[
          ('input_scan', '/scan_raw')
        ])

    ld = LaunchDescription()
    ld.add_action(detector_cmd)
    return ld
```

Ideas clave del ejemplo:

- `LaunchDescription()` es el "contenedor" de acciones que se van a ejecutar.
- `Node(...)` describe un proceso ROS 2 a lanzar (paquete + ejecutable) y su configuración.
- `parameters=[param_file]` permite cargar parámetros desde un YAML (se detalla en el punto 2).
- `remappings=[('input_scan','/scan_raw')]` reescribe nombres: el nodo se subscribe a `input_scan`, pero realmente recibirá del topic `/scan_raw`.

Fragmento C++ 3.1: Archivo launch mínimo para un nodo con parámetros y remapping (`laser.launch.py`)

Configuración mediante parámetros

Los **parámetros** en ROS 2 son valores de configuración asociados a un nodo (por ejemplo, un umbral, una ganancia, un nombre de frame, etc.). La idea es **no hardcodear** constantes en el código: se definen valores por defecto, se declaran y luego se pueden sobreescribir desde launch o desde YAML.

Fichero de parámetros (YAML)

El fichero `ch3_examples/sensors/laser/config/params.yaml` configura el nodo `obstacle_detector_node`:

```
obstacle_detector_node:
  ros__parameters:
    use_sim_time: true
    min_distance: 0.75
```

Fragmento C++ 3.2: Ejemplo de YAML de parámetros (`params.yaml`)

Cómo leer este YAML:

- La clave superior `obstacle_detector_node` coincide con el **nombre del nodo**. Esto hace que los parámetros se apliquen a ese nodo (si el nombre coincide).
- Bajo `ros__parameters` aparecen pares `clave: valor`.
- `use_sim_time` fuerza al nodo a usar el reloj simulado (útil en Gazebo/RViz).
- `min_distance` es un umbral (en metros) usado por el nodo en su lógica.

Declaración y lectura en el nodo (C++)

En el nodo `ch3_examples/sensors/laser/src/laser/ObstacleDetectorNode.cpp` se observa el patrón típico: **declarar** parámetros con un valor por defecto y luego **leer** el valor que finalmente se haya configurado (por defecto o sobreescrito desde YAML/launch):

```
ObstacleDetectorNode::ObstacleDetectorNode()
: Node("obstacle_detector_node")
{
  declare_parameter("min_distance", min_distance_);
  get_parameter("min_distance", min_distance_);

  RCLCPP_INFO(get_logger(), "ObstacleDetectorNode set to %f m",
     min_distance_);
```

```
  // ...
}
```

Notas para el ejercicio:

- `declare_parameter` crea el parámetro (si no existía) y fija un valor por defecto.
- `get_parameter` lee el valor efectivo (tras aplicar YAML/launch).
- Si el nombre del nodo en C++ no coincide con el nombre usado en YAML, esos parámetros no se aplicarán (por eso en el ejemplo el nodo se llama "`obstacle_detector_node`").

Fragmento C++ 3.3: Declarar y leer un parámetro en un nodo (`ObstacleDetectorNode.cpp`)

Procesado de `sensor_msgs/msg/LaserScan`

`LaserScan` es el mensaje estándar para láser 2D planar: proporciona una serie de distancias medidas (vector `ranges`) y la geometría del barrido (`angle_min`, `angle_increment`, etc.).

En el ejemplo
`ch3_examples/sensors/laser/src/laser/ObstacleDetectorNode.cpp`
se ve cómo:

- el nodo se subscribe al scan,
- calcula el mínimo rango,
- y convierte índice → ángulo → coordenadas (x, y) en el plano del láser.

Suscripción al láser

```cpp
laser_sub_ = create_subscription<sensor_msgs::msg::LaserScan>(
  "input_scan", rclcpp::SensorDataQoS().reliable(),
  std::bind(&ObstacleDetectorNode::laser_callback, this, _1));
```

Fragmento C++ 3.4: Suscripción a `LaserScan` con QoS de sensor

Extraer el obstáculo más cercano y su posición 2D

El ejemplo utiliza el mínimo de `scan.ranges` y calcula el ángulo del rayo asociado con:

$$\theta = \texttt{angle_min} + i\,\texttt{angle_increment}$$

y las coordenadas en el plano del sensor con:

$$x = r\cos\theta, \quad y = r\sin\theta$$

```cpp
int min_idx = std::min_element(scan.ranges.begin(), scan.ranges.end()) -
    scan.ranges.begin();
float distance_min = scan.ranges[min_idx];

float obstacle_angle = scan.angle_min + min_idx * scan.angle_increment;
float obstacle_x = distance_min * std::cos(obstacle_angle);
float obstacle_y = distance_min * std::sin(obstacle_angle);
```

Fragmento C++ 3.5: Cálculo de ángulo y (x,y) del obstáculo más cercano (extracto)

La primera línea puede resultar poco intuitiva si no se ha trabajado antes con algoritmos de la STL de C++:

- `scan.ranges` es un `std::vector<float>` con las distancias medidas; cada entrada `ranges[i]` corresponde a un rayo del láser.

- `std::min_element(begin, end)` recorre el vector y devuelve un **iterador** (una "referencia avanzada") al elemento con menor valor, no un índice.
- Para convertir ese iterador en un índice entero, se calcula cuántas posiciones hay desde el comienzo del vector: `it_min - begin`. Ese resultado es precisamente el índice `i` tal que `ranges[i]` es el mínimo.

Una forma equivalente y a veces más legible es separar los pasos:

```
auto it_min = std::min_element(scan.ranges.begin(), scan.ranges.end());
int min_idx = std::distance(scan.ranges.begin(), it_min);
```

Fragmento C++ 3.6: Mismo cálculo, en dos pasos (alternativa legible)

En ambos casos, el objetivo es obtener el índice del rayo con menor distancia para, a partir de `angle_min` y `angle_increment`, recuperar el ángulo y proyectar a coordenadas (x, y).

Importante para el ejercicio: en un sistema real debes filtrar rangos inválidos (`NaN`, `Inf` y valores fuera de `range_min` /`range_max`) antes de buscar el mínimo. Además, recuerda que estas coordenadas están inicialmente en el **frame del sensor** (`LaserScan.header.frame_id`) y normalmente tendrás que transformarlas al frame del robot (p.ej. `base_link`) usando TF.

Publicación y consulta de TF

TF (`tf2`) es el sistema estándar de ROS 2 para manejar **transformaciones entre marcos de referencia** (frames): permite publicar transformaciones (padre → hijo) y consultarlas para expresar puntos/poses en el frame que necesites.

Para esta explicación usaremos el ejemplo `ch3_examples/tf_seeker`, que incluye:

- un nodo que **publica** una TF (un objetivo "target" en `odom`),
- y un nodo que **lee** esa TF desde `base_footprint` para generar control.

Publicar una TF (TransformBroadcaster)

En `ch3_examples/tf_seeker/src/tf_seeker/TFPublisherNode.cpp` se instancia un `TransformBroadcaster` y se envía periódicamente un `TransformStamped`:

```
tf_broadcaster_ = std::make_shared<tf2_ros::TransformBroadcaster>(*this);

transform_.header.frame_id = "odom";
transform_.child_frame_id = "target";

transform_.transform.translation.x = pos_x(generator_);
transform_.transform.translation.y = pos_y(generator_);
transform_.transform.translation.z = 0.0;

transform_.header.stamp = now();
tf_broadcaster_->sendTransform(transform_);
```

Fragmento C++ 3.7: Publicación periódica de una TF (extracto de `TFPublisherNode.cpp`)

Ideas clave:

- Una TF es siempre **entre dos frames**: `frame_id` (padre) y `child_frame_id` (hijo).
- El `stamp` determina en qué instante es válida esa transformación.

- En este tipo de ejercicios, cuando publiques un frame propio (p.ej. `target` o `nearest_obstacle`), elige nombres consistentes y documenta claramente la relación padre/hijo.

Leer una TF (Buffer + TransformListener)

En `ch3_examples/tf_seeker/src/tf_seeker/TFSeekerNode.cpp` se crea un `tf2_ros::Buffer` y un `TransformListener`. En cada ciclo de control, el nodo comprueba si puede transformar y, si es posible, consulta la transformación para obtener la posición del objetivo en el frame del robot:

```cpp
if (tf_buffer_.canTransform("base_footprint", "target", tf2::TimePointZero
    , &error)) {
  auto bf2target_msg = tf_buffer_.lookupTransform(
    "base_footprint", "target", tf2::TimePointZero);

  tf2::fromMsg(bf2target_msg, bf2target);

  double x = bf2target.getOrigin().x();
  double y = bf2target.getOrigin().y();
  // ... control a partir de (x,y)
}
```

Fragmento C++ 3.8: Consulta de TF y obtención de (x,y) del objetivo (extracto de `TFSeekerNode.cpp`)

Notas para el ejercicio:

- `canTransform` evita excepciones si aún no hay TF en el buffer.
- `TimePointZero` pide "la última disponible". En este ejercicio, cuando necesites coherencia temporal (p.ej. con un `LaserScan.header.stamp`), deberías consultar con ese instante.
- Una vez expresado el objetivo/obstáculo en el frame del robot, la geometría es sencilla: con (x, y) puedes calcular ángulo y distancia ($\mathrm{atan2}(y, x)$ y $\sqrt{x^2 + y^2}$) para control y toma de decisiones.

Transformar un punto entre marcos (PointStamped + doTransform)

En el Paso 1 (láser), el obstáculo más cercano se calcula inicialmente en el **frame del sensor** (`LaserScan.header.frame_id`). Sin embargo, para que el resultado sea directamente consumible por otros módulos, suele interesar expresarlo en el **frame del robot** (por ejemplo, `base_link`).

La idea operativa es: construir un `geometry_msgs/msg/PointStamped` con el punto en el marco del sensor, consultar la transformación en el instante del sensor y aplicar dicha transformación al punto.

Este patrón se ve en `ch3_examples/sensors/laser/src/laser/ObstacleDetectorNode.cpp`:

```cpp
geometry_msgs::msg::PointStamped obstacle_point;
obstacle_point.header = scan.header;  // frame_id + stamp del sensor
obstacle_point.point.x = obstacle_x;
obstacle_point.point.y = obstacle_y;
obstacle_point.point.z = 0.0;

try {
  auto tf = tf_buffer_.lookupTransform(
    "base_link",
    obstacle_point.header.frame_id,
    obstacle_point.header.stamp,
    tf2::durationFromSec(0.1));

  geometry_msgs::msg::PointStamped obstacle_point_base;
```

```
    tf2::doTransform(obstacle_point, obstacle_point_base, tf);

    // obstacle_point_base.point.* ya está expresado en base_link
} catch (const tf2::TransformException & ex) {
    // Puede fallar si falta información en el buffer o si la consulta
       extrapola.
}
```

Fragmento C++ 3.9: Transformar un punto desde el frame del sensor a `base_link` (extracto)

Notas para el ejercicio:

- Preservar `header.stamp` y consultar TF con ese instante evita incoherencias temporales.
- Evita recomponer geometría a mano: el buffer TF se encarga de la composición e interpolación necesarias.
- Si la transformación no está disponible, no publiques resultados derivados de ese punto (mejor no publicar que publicar mal).

Procesado de imágenes

En ROS 2, las imágenes se transportan como mensajes `sensor_msgs/msg/Image`. Sin embargo, la mayoría de algoritmos de visión (segmentación, filtrado, detección por color, etc.) están implementados sobre estructuras de OpenCV como `cv::Mat`. Por ello, el patrón típico es:

1. convertir `Image` → `cv::Mat` con `cv_bridge`,
2. manipular la imagen con OpenCV (por ejemplo, en HSV),
3. publicar el resultado (por ejemplo, como detección 2D).

Usaremos como referencia el nodo `ch3_examples/sensors/camera/src/camera/HSVFilterNode.cpp`.

De `sensor_msgs::msg::Image` a `cv::Mat` (cv_bridge)

En el callback de imagen, el ejemplo convierte el mensaje ROS a un `cv::Mat` BGR8:

```
cv_bridge::CvImagePtr cv_ptr;
try {
  cv_ptr = cv_bridge::toCvCopy(image, sensor_msgs::image_encodings::BGR8);
} catch (cv_bridge::Exception & e) {
  RCLCPP_ERROR(get_logger(), "cv_bridge exception: %s", e.what());
  return;
}
cv::Mat & image_cv = cv_ptr->image;
```

Fragmento C++ 3.10: Conversión de `Image` a `cv::Mat` con `cv_bridge` (extracto)

Qué hace `cv_bridge::toCvCopy`:

- **Entrada 1** (`image`): el mensaje ROS con píxeles y metadatos (`width`, `height`, `step`, `encoding`, etc.).
- **Entrada 2** (`BGR8`): el *encoding* deseado para trabajar en OpenCV. Aquí se pide BGR de 8 bits por canal (formato típico en OpenCV).
- **Salida**: un `cv_bridge::CvImagePtr` cuyo campo `image` es un `cv::Mat` listo para procesado.
- **Por qué puede fallar**: si el *encoding* de entrada no es convertible al solicitado, lanza una excepción; por eso se usa `try/catch`.

Por qué es necesario: el mensaje `Image` contiene un búfer de bytes y metadatos (ancho, alto, *encoding*, etc.), pero no ofrece operaciones de visión. `cv::Mat` sí permite aplicar transformaciones de color, máscaras, momentos, etc.

Transformación a HSV y filtrado por rango

El filtrado por color suele hacerse en HSV porque separa mejor tono (H) de iluminación (V) que RGB/BGR. El ejemplo realiza:

- `cvtColor` para pasar de BGR a HSV,
- `inRange` para crear una máscara binaria (pixeles que están dentro del rango HSV).

```cpp
cv::Mat img_hsv, out;
cv::cvtColor(in, img_hsv, cv::COLOR_BGR2HSV);
cv::inRange(img_hsv, cv::Scalar(h, s, v), cv::Scalar(H, S, V), out);
```

Fragmento C++ 3.11: Filtrado por HSV: BGR→HSV + máscara binaria (extracto)

Qué hace `cv::cvtColor(src, dst, code)`:

- `src`: imagen de entrada (aquí, BGR).
- `dst`: imagen de salida (aquí, HSV).
- `code`: el tipo de conversión (aquí, `cv::COLOR_BGR2HSV`).

En OpenCV, el canal H suele representarse en el rango `[0,179]` (la mitad de `0..359`), y S/V en `[0,255]`. Por eso en muchos ejemplos se ven trackbars de H hasta `360/2`.

Qué hace `cv::inRange(src, lowerb, upperb, dst)`:

- `src`: la imagen sobre la que se umbraliza (aquí, HSV).
- `lowerb/upperb`: límites inferior y superior (aquí, `cv::Scalar(h,s,v)` y `cv::Scalar(H,S,V)`).
- `dst`: máscara binaria (matriz 1 canal, típicamente `uint8`) donde vale 255 si el pixel cumple el rango y 0 si no.

Con esa máscara, el nodo estima un centro con `cv::moments(mask, true)` (centroide de los píxeles activos) y construye una caja con `cv::boundingRect(mask)` (rectángulo mínimo que encierra los píxeles no-cero). Si no hay píxeles activados, no publica nada (regla coherente con el resto del ejercicio).

Visualización: `cv::imshow`, `cv::waitKey` y `copyTo` con máscara

El ejemplo muestra en pantalla la imagen filtrada (los píxeles que pasan el filtro HSV) para depuración. El patrón típico (tal cual aparece en el ejemplo) es:

```cpp
cv::waitKey(1);

cv::Mat out_image;
image.copyTo(out_image, image_filtered);
cv::imshow("Filtered Image", out_image);
```

Fragmento C++ 3.12: Visualización de una máscara sobre la imagen original (extracto)

Qué hace cada llamada:

- `cv::waitKey(1)`: procesa eventos de la GUI de OpenCV (p.ej. refresco de ventanas). Sin esta llamada, es habitual que `imshow` no actualice la ventana. El argumento es el tiempo máximo de espera en milisegundos; `1` significa "no bloquear" de forma apreciable.
- `image.copyTo(out_image,image_filtered)`: copia `image` a `out_image`, pero **solo** en los píxeles donde la máscara `image_filtered` es distinta de cero. El segundo argumento es una máscara de 1 canal (en el ejemplo, `cv::Mat1b`) con valores 0/255.

- `cv::imshow("Filtered Image", out_image)`: muestra la imagen en la ventana cuyo nombre es el primer argumento. Si la ventana no existe, OpenCV la crea; en el ejemplo se crea antes con `cv::namedWindow("Filtered Image")`.

Nota para el ejercicio: estas funciones son útiles para depurar, pero no son necesarias para la lógica del robot. En un robot real (sin entorno gráfico) suelen desactivarse o sustituirse por publicación de imágenes/markers para RViz.

Publicación del resultado (detección 2D)

En este ejemplo, el "resultado" no es una imagen filtrada, sino una detección 2D publicada como `vision_msgs/msg/Detection2DArray`. El patrón es: rellenar cabecera (frame + timestamp), bbox y publicar.

```
vision_msgs::msg::Detection2D detection_msg;
detection_msg.header.frame_id = image->header.frame_id;
detection_msg.header.stamp = image->header.stamp;
detection_msg.bbox.center.position.x = point.x + bbx.width / 2;
detection_msg.bbox.center.position.y = point.y + bbx.height / 2;
detection_msg.bbox.size_x = bbx.width;
detection_msg.bbox.size_y = bbx.height;

vision_msgs::msg::Detection2DArray detection_array_msg;
detection_array_msg.header.frame_id = image->header.frame_id;
detection_array_msg.header.stamp = image->header.stamp;
detection_array_msg.detections.push_back(detection_msg);

detection_pub_->publish(detection_array_msg);
```

Fragmento C++ 3.13: Publicar una detección 2D (extracto)

Nota: es importante que `header.frame_id` sea coherente con el sensor/cámara, ya que ese frame se utilizará después para transformar detecciones a otros marcos (TF). El patrón recomendable es reutilizar el `frame_id` del mensaje de imagen de entrada.

Uso de `CameraInfo`: modelo pinhole y ángulos

Para convertir coordenadas de imagen (pixeles) a direcciones 3D, se necesita la calibración intrínseca de la cámara, que llega en `sensor_msgs/msg/CameraInfo`. El ejemplo crea un `image_geometry::PinholeCameraModel` y, con él, obtiene el rayo asociado a un píxel.

```
model_ = std::make_shared<image_geometry::PinholeCameraModel>();
model_->fromCameraInfo(*info);
```

Fragmento C++ 3.14: Construcción del `PinholeCameraModel` a partir de `CameraInfo` (extracto)

Y luego calcula un rayo y dos ángulos (yaw/pitch) hacia el centro detectado:

```
cv::Point3d ray = model->projectPixelTo3dRay(model->rectifyPoint(center));
ray = ray / ray.z;
float yaw = atan2(ray.x, ray.z);
float pitch = atan2(ray.y, ray.z);
```

Fragmento C++ 3.15: Del pixel al rayo 3D y a ángulos (extracto)

Estos ángulos se usan típicamente para orientar el robot/cámara hacia el objetivo (control visual).

Reconstrucción 3D con imagen de profundidad

El siguiente paso natural tras una detección 2D es estimar una posición 3D. Si el sistema dispone de una **imagen de profundidad**, cada píxel aporta una distancia Z al sensor. El nodo `ch3_examples/sensors/camera/src/camera/DetectionTo3DfromDepthNode.cpp` implementa este flujo:

$$\texttt{Detection2DArray} + \texttt{depth Image} + \texttt{CameraInfo} \longrightarrow \texttt{Detection3DArray}$$

Sincronización con `message_filters`

La detección 2D y la imagen de profundidad llegan por topics distintos y con timestamps que no coinciden exactamente. Para evitar emparejar una detección con una profundidad de otro instante, el ejemplo usa `message_filters` con una política **ApproximateTime**.

```
depth_sub_ = std::make_shared<message_filters::Subscriber<sensor_msgs::msg::Image>>(
  this, "input_depth", rclcpp::SensorDataQoS().reliable().get_rmw_qos_profile());
detection_sub_ = std::make_shared<message_filters::Subscriber<vision_msgs::msg::Detection2DArray>>(
  this, "input_detection_2d", rclcpp::SensorDataQoS().reliable().get_rmw_qos_profile());

sync_ = std::make_shared<message_filters::Synchronizer<MySyncPolicy>>(
  MySyncPolicy(10), *depth_sub_, *detection_sub_);
sync_->registerCallback(std::bind(&DetectionTo3DfromDepthNode::callback_sync, this, _1, _2));
```

Fragmento C++ 3.16: Sincronización aproximada: depth + detección 2D (extracto)

Interpretación: el sincronizador mantiene pequeñas colas (aquí, tamaño 10) y dispara `callback_sync` cuando encuentra dos mensajes con timestamps suficientemente próximos.

Leer profundidad (16UC1/32FC1) y proyectar a 3D

La idea de esta parte es convertir una detección 2D (un píxel o una caja) en un punto 3D. El dato clave es la profundidad Z asociada a un píxel (u, v) en la imagen de profundidad. En el ejercicio, el flujo mínimo es:

1. Elegir el píxel (u, v) (por ejemplo, el centro de la `bbox`).
2. Leer su profundidad Z (en metros) desde la imagen de profundidad.
3. Convertir (u, v, Z) a coordenadas 3D (X, Y, Z) usando el modelo pinhole (intrínsecos de `CameraInfo`).

(A) Qué significan `16UC1` y `32FC1` El campo `encoding` del mensaje de profundidad indica cómo están almacenados los valores en el búfer de píxeles:

- `16UC1`: un canal, enteros sin signo de 16 bits (`uint16_t`). Muy habitual en sensores RGB-D: la profundidad viene en **milímetros**. Por eso hay que convertir a metros.
- `32FC1`: un canal, `float` de 32 bits. Normalmente la profundidad ya viene en **metros**.

Nota para el ejercicio: no es raro que haya valores inválidos. En `16UC1` suele aparecer 0 (sin retorno) y en `32FC1` pueden aparecer `NaN` o `Inf`. En un nodo real conviene descartar esos casos (y, si procede, no publicar nada).

(B) Leer Z en el píxel (u, v) El ejemplo usa `cv_bridge` para obtener un `cv::Mat` y luego indexar el píxel. La clave es que el tipo de lectura debe coincidir con el `encoding`:

```
cv_bridge::CvImagePtr cv_depth_ptr = cv_bridge::toCvCopy(*image_msg,
     image_msg->encoding);

float depth = 0.0;
if (image_msg->encoding == "16UC1") {
  depth = depth_image_proc::DepthTraits<uint16_t>::toMeters(
    cv_depth_ptr->image.at<uint16_t>(cv::Point2d(u, v)));
} else {
  depth = cv_depth_ptr->image.at<float>(cv::Point2d(u, v));
}
```

Fragmento C++ 3.17: Lectura de profundidad en el centro de la detección (extracto)

Cosas que suelen confundir:

- u y v son coordenadas de píxel (columna y fila). Asegúrate de que están dentro de la imagen antes de acceder ($0 \le u <$ `width`, $0 \le v <$ `height`).
- Si tomas el centro de la bbox, recuerda que muchas veces es `float`. En código normalmente se redondea o se hace *cast* a entero.
- Si el valor leído no es válido (p.ej. 0 en `16UC1` o `NaN` en `32FC1`), lo razonable es descartar esa detección (o buscar un píxel cercano válido, si se quiere robustez extra).

(C) Proyección pinhole: de (u, v, Z) a (X, Y, Z) Una cámara pinhole viene definida (a nivel mínimo) por cuatro parámetros intrínsecos: f_x, f_y, c_x, c_y (focales y centro óptico), que están en `CameraInfo`. Con ellos, la fórmula clásica es:

$$X = \frac{(u - c_x)\,Z}{f_x}, \qquad Y = \frac{(v - c_y)\,Z}{f_y}, \qquad Z = Z$$

El ejemplo lo hace con `image_geometry` para evitar detalles de distorsión y para reutilizar el modelo ya implementado. Conceptualmente, lo que hace es:

1. Construir un **rayo** (dirección) 3D asociado al píxel (u, v).
2. Normalizar el rayo para que su componente z valga 1 (`ray = ray / ray.z`).
3. Escalar el rayo por Z para obtener el punto: `point = ray * depth`.

Con el `PinholeCameraModel` (creado desde `CameraInfo`) obtiene el rayo y lo escala por la profundidad para reconstruir el punto 3D:

```
cv::Point3d ray = model_->projectPixelTo3dRay(model_->rectifyPoint(cv::
     Point2d(u, v)));
ray = ray / ray.z;
cv::Point3d point = ray * depth;
```

Fragmento C++ 3.18: De pixel + depth a punto 3D (extracto)

Interpretación geométrica: `projectPixelTo3dRay` devuelve un vector que apunta desde el centro óptico hacia el píxel. Tras normalizarlo con `/ ray.z`, ese vector representa dónde estaría el punto a una profundidad $Z = 1$ m. Al multiplicar por `depth`, lo llevas a la profundidad real.

Finalmente publica la detección 3D como `Detection3DArray`, rellenando `bbox.center.position.x/y/z` con el punto estimado. Si la profundidad es `NaN`, el ejemplo descarta esa detección.

Reconstrucción 3D con `PointCloud2`

Otra forma de obtener 3D es usar directamente un **PointCloud** generado por un sensor RGB-D o por un pipeline de percepción. En ese caso, el mensaje `sensor_msgs/msg/PointCloud2` ya contiene puntos 3D y el problema se reduce a: "dado un píxel (u, v) de la detección 2D, tomar el punto 3D correspondiente en la nube".

El nodo `ch3_examples/sensors/camera/src/camera/DetectionTo3DfromPCNode.cpp` sincroniza `PointCloud2` y detecciones 2D con `message_filters` (igual que en el caso depth), convierte a tipos PCL y extrae el punto central.

De `PointCloud2` a PCL

```
pcl::PointCloud<pcl::PointXYZ>::Ptr pc(new pcl::PointCloud<pcl::PointXYZ>)
    ;
pcl::fromROSMsg(*pc_msg, *pc);
```

Fragmento C++ 3.19: Conversión de `PointCloud2` ROS a `pcl::PointCloud<pcl::PointXYZ>` (extracto)

Extraer el punto 3D del centro de la detección

El ejemplo asume una **nube organizada** ("imagen 3D"): se puede indexar como una matriz usando `at(u,v)`.

```
pcl::PointXYZ & center = pc->at(u, v);

detection_3d_msg.bbox.center.position.x = center.x;
detection_3d_msg.bbox.center.position.y = center.y;
detection_3d_msg.bbox.center.position.z = center.z;
```

Fragmento C++ 3.20: Extracción del punto 3D en el centro de la bbox (extracto)

Después filtra puntos no finitos (`NaN`/`Inf`) y publica `Detection3DArray`.

Diferencias respecto al método con profundidad

- **Depth + CameraInfo:** se toma Z de la imagen de profundidad y se usa el modelo pinhole (intrínsecos) para deproyectar (u, v, Z) a (X, Y, Z). Es un método muy general.
- **PointCloud2:** la nube ya contiene (X, Y, Z), así que basta con convertir a PCL y tomar el punto en (u, v) (si la nube es organizada). En ese caso, el cálculo es más directo.
- **Limitación operativa:** si la nube no es organizada, no existe una correspondencia directa píxel→punto con `at(u,v)` y habría que usar otra estrategia (no cubierta en este ejemplo).

4 Coordinación de misiones con FSM

4.1 **Objetivo** **171**

4.2 **Guión de desarrollo** . . . **171**

4.3 **Teoría y herramientas para el ejercicio** **174**

Implementación de FSM en C++ para ROS 2 174

Capacidad vs misión 184

Interfaz de capacidad: patrón general y ejemplo con NavigateToPose 184

Este ejercicio se centra en integrar una capacidad robótica realista y construir encima una tarea de nivel superior (misión secuencial basada en objetivos, por ejemplo patrullaje, inspección o reparto interno) coordinada mediante una máquina de estados finitos.

Desde el punto de vista arquitectónico, en este ejercicio se trabajará en:

- Distinguir explícitamente entre *capacidad* y *misión* (ejecutar una lógica sobre una secuencia de objetivos).
- Diseñar interfaces limpias entre capas usando un mecanismo estándar de interacción (acciones).

Desde el punto de vista técnico, se trabajará en:

- Integrar y configurar una capacidad robótica basada en objetivos.
- Enviar objetivos a la capacidad seleccionada y gestionar sus resultados.
- Implementar una FSM de misión que utilice dicha capacidad.

Es importante destacar que en este ejercicio la misión se mantiene deliberadamente simple, de modo que no requiere orquestación compleja: la máquina de estados finitos invoca directamente la capacidad seleccionada para alcanzar los objetivos definidos. No es necesario diseñar una capa de coordinación adicional entre tareas o capacidades, ya que la lógica de misión se resuelve completamente en la FSM, que gestiona la secuencia de objetivos y las posibles recuperaciones ante fallo. Esta simplicidad permite centrarse en la integración y separación clara entre misión, tarea y capacidad, sin introducir complejidad arquitectónica innecesaria.

Este ejercicio se apoya principalmente en los conceptos introducidos en varios capítulos de la Parte I, en particular:

- Capítulo 1: descomposición en misión, tarea y capacidad, y definición de interfaces entre capas.
- Capítulo 2: ejecución reactiva y comunicación con componentes externos mediante mecanismos estándar de ROS 2.
- Capítulo 5: modelado de comportamiento mediante máquinas de estados finitos.

El objetivo no es introducir una arquitectura completa de misión compleja, sino aplicar de forma controlada la separación entre capacidad y misión para construir una FSM clara, observable y fácil de validar.

4.1. Objetivo

El objetivo general es construir un robot capaz de:

1. Ejecutar una misión basada en objetivos (por ejemplo, patrullaje, inspección o reparto interno) mediante una FSM, gestionando éxitos, fallos y repetición según la política de misión definida para el caso de uso elegido.
2. Integrar la misión con una capacidad robótica adecuada al contexto de aprendizaje o experimentación a través de una interfaz de interacción apropiada (por ejemplo, acciones en ROS 2). En este capítulo se ejemplifica con Nav2.

El resultado debe ser observable en simulación y, si el entorno lo permite, ejecutable en robot real.

4.2. Guión de desarrollo

El ejercicio se desarrolla siguiendo una secuencia incremental de pasos, donde cada paso produce un sistema funcional y verificable antes de proceder con el siguiente. Se debe completar cada paso de forma ordenada y validar su correcto funcionamiento antes de continuar.

Posibles ejercicios

La misma estructura de trabajo propuesta en este capítulo puede aplicarse a distintos ejercicios, siempre que se mantenga la separación misión–tarea–capacidad y la coordinación mediante FSM. A continuación se sugieren opciones que pueden adaptarse según el contexto de aprendizaje autónomo:

- **Control de acceso**: espera solicitud en un punto de entrada, valida una credencial (por ejemplo, QR o RFID), autoriza o deniega el paso y vuelve al estado de espera.
- **Acompañamiento a mantenimiento**: recibe una orden de asistencia, guía al personal hasta un equipo concreto, espera confirmación de intervención y retorna a base.
- **Inventario de estanterías**: recorre una lista de ubicaciones, captura lectura de etiquetas en cada estante y registra incidencias de faltantes antes de cerrar la misión.
- **Supervisión de puntos críticos**: visita estaciones definidas (puerta técnica, sala de servidores, almacén), verifica una condición en cada una y publica un estado global.
- **Recarga autónoma programada**: ejecuta tareas de operación normal hasta alcanzar umbral de batería, navega al punto de carga y reanuda la operación al recuperar nivel mínimo.
- **Atención prioritaria de incidencias**: mantiene un flujo principal de servicio y, ante una alarma de alta prioridad, conmuta a un estado de respuesta inmediata, notifica el resultado y retoma el plan pendiente.

En todos los casos, el patrón de desarrollo es el mismo: preparar entorno y robot, integrar la capacidad seleccionada y, finalmente, implementar la FSM de misión con observabilidad y políticas de recuperación.

Paso 1: preparación del entorno y del robot

En este paso inicial, se debe verificar que el robot está correctamente configurado y que dispone de los sensores, transformadas y recursos necesarios para la capacidad seleccionada.

1. Lanzar el entorno de simulación del robot proporcionado así como la capacidad o capacidades necesarias (en este guión, navegación con Nav2).
2. Abrir RViz2 y verificar la configuración visual del robot.
3. Identificar y listar los topics relevantes para la capacidad seleccionada (en el ejemplo de navegación, `/scan`, `/odom` y `/cmd_vel`):
   - Topic de sensor láser (normalmente `/scan`).
   - Topic de odometría (normalmente `/odom`).
   - Topic de comandos de velocidad (normalmente `/cmd_vel`).
4. Verificar que las transformadas del robot están correctamente publicadas:
   - Ejecutar `ros2 run tf2_tools view_frames` para generar el grafo de transformadas.
   - Confirmar la presencia de los marcos `odom`, `base_link` y `base_scan` (o equivalentes según el robot).

Paso 2: integración de la capacidad

En este paso se configura y valida la capacidad que constituirá la base sobre la que se construirá la tarea de misión. Como ejemplo, el guión muestra el caso de navegación autónoma mediante Nav2.

1. Lanzar la capacidad de referencia (Nav2 en este guión) indicando el mapa sobre el cual navegar:
   - Ejecutar el launch file proporcionado por la capacidad seleccionada, especificando la configuración necesaria (en el ejemplo de Nav2, la ruta al archivo `map.yaml`).
   - Verificar en RViz2 que el mapa se visualiza correctamente.
   - Comprobar que la acción de navegación está disponible ejecutando:
     `ros2 action list` y verificando la presencia de `/navigate_to_pose`.
2. Establecer la pose inicial del robot:
   - Usar la herramienta "2D Pose Estimate" de RViz2.
   - Indicar la posición y orientación aproximada del robot en el mapa.
   - Observar cómo las partículas convergen hacia la pose correcta.
3. Validar el funcionamiento de la capacidad, en este caso, de navegación, mediante objetivos manuales:
   - Usar la herramienta "Nav2 Goal" de RViz2 para enviar objetivos de navegación.
   - Observar el comportamiento ante obstáculos (evitación dinámica).

Paso 3: diseño e implementación de la misión con FSM

En este paso final, se implementará una FSM que coordina una tarea de misión basada en objetivos mediante la capacidad validada en el paso anterior.

1. Diseñar la FSM de misión:
   - Definir los estados necesarios.
   - Establecer las transiciones entre estados basadas en eventos de la capacidad seleccionada.
   - Decidir la política de reintentos ante fallos de ejecución de la capacidad.
2. Crear un nuevo paquete ROS 2 para el comportamiento de misión:
   - Usar `ros2 pkg create` con las dependencias necesarias (`rclcpp`, `geometry_msgs` y, si aplica al ejemplo elegido, paquetes específicos de la capacidad; en este guión, `nav2_msgs`).
   - Estructurar el código siguiendo alguno de los patrones de FSM estudiados en la sección de teoría.
3. Implementar el cliente/interfaz de interacción de la capacidad:
   - Crear el cliente correspondiente al contrato de la capacidad (en el ejemplo, un cliente de acción del tipo `NavigateToPose`).
   - Implementar callbacks para goal response, feedback y result.
   - Gestionar adecuadamente los estados de la acción (pending, active, succeeded, aborted, canceled).
4. Definir una secuencia de objetivos/hitos para la misión:
   - Inicialmente, codificar una secuencia estática de objetivos en el código.
   - Definir la representación de cada objetivo según la capacidad elegida (por ejemplo, pose, zona, tarea, etiqueta o comando).
   - Opcionalmente, cargar la secuencia de objetivos desde un archivo de parámetros.
5. Implementar la lógica de la FSM:
   - Al activarse, la FSM debe enviar el primer objetivo de la secuencia a la capacidad seleccionada (en el ejemplo, goal a Nav2).
   - Al recibir confirmación de éxito, avanzar al siguiente objetivo.
   - Al completar todos los objetivos, aplicar la política definida para la misión (por ejemplo, reiniciar ciclo, finalizar ronda o esperar nueva orden).
   - Ante fallos de ejecución, implementar una política de reintentos (por ejemplo, 3 intentos antes de omitir el objetivo actual).
6. Añadir observabilidad al comportamiento:
   - Publicar el estado actual de la FSM en un topic (`std_msgs/String` o mensaje personalizado).
   - Publicar el objetivo/hito actual y el progreso de la misión.
   - Implementar logging adecuado de transiciones de estado y eventos relevantes.
7. Validar el comportamiento completo:
   - Lanzar el sistema completo (simulación + capacidad seleccionada + FSM de misión; en este guión, Nav2).
   - Observar que el robot ejecuta secuencialmente todos los objetivos de la misión.

- Provocar un fallo en la ejecución de la capacidad (por ejemplo, bloqueando el camino en el caso de navegación) y verificar la recuperación.
- Confirmar que la misión se ejecuta de acuerdo con la política definida de forma autónoma.

Al finalizar este paso, se dispondrá de un sistema completo de misión autónoma basada en objetivos que demuestra la separación clara entre capacidad (Nav2 en el ejemplo) y tarea (FSM de misión), consolidando los conceptos arquitectónicos del modelo misión–tarea–capacidad.

4.3. Teoría y herramientas para el ejercicio

Implementación de FSM en C++ para ROS 2

Antes de integrar la capacidad de referencia (Nav2 en este capítulo), es fundamental comprender cómo implementar correctamente una FSM en ROS 2. Esta sección proporciona patrones arquitectónicos que servirán como base para la implementación de la FSM de misión.

Patrón básico: switch-case

Para FSM simples (por ejemplo, de menos de 10 estados), el patrón switch-case es directo y eficiente. El paquete `fsm_examples` incluye un ejemplo completo que ilustra este patrón. El fragmento de código 4.1 muestra la declaración de la clase, que incluye:

- Un `enum class State` con los cuatro estados posibles.
- Variables de estado (`current_state_`, `min_distance_`, etc.).
- Suscriptores, publicadores y un timer para el ciclo de control.

```
class BasicFSMRobot : public rclcpp::Node
{
public:
  enum class State { IDLE, MOVING, OBSTACLE_DETECTED, STOPPED };

  BasicFSMRobot();

private:
  State current_state_;
  double min_distance_ = 10.0;
  const double OBSTACLE_THRESHOLD = 0.5;  // 0.5 metros
  bool start_button_pressed_ = false;
  rclcpp::Time stopped_time_;

  rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr laser_sub_;
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr vel_pub_;
  rclcpp::TimerBase::SharedPtr timer_;

  void laser_callback(const sensor_msgs::msg::LaserScan::SharedPtr msg);
  void control_cycle();
  void publish_velocity(double linear, double angular);

public:
  void simulate_start_button();
};
```

Fragmento C++ 4.1: Declaración de la clase BasicFSMRobot (fsm_basic_example.hpp)

El fragmento de código 4.2 presenta el corazón de la FSM: el método `control_cycle()`, que se ejecuta periódicamente (cada 100ms). Observa cómo:

- Cada `case` ejecuta acciones (`publish_velocity`) en cada ciclo.
- Las transiciones ocurren al cambiar `current_state_`.
- No hay métodos `on_entry` ni `on_exit`, por lo que las acciones se repiten.

Además, el mismo código incluye `laser_callback()`, que ilustra el principio clave: los callbacks de sensores solo actualizan variables (`min_distance_`), sin tomar decisiones.

```
void BasicFSMRobot::control_cycle()
{
  // Este método se llama periódicamente (cada 100ms)
  // AQUÍ es donde la FSM toma decisiones

  switch (current_state_) {
      case State::IDLE:
        // Acción: detener motores (se ejecuta en cada ciclo - no ideal)
        publish_velocity(0.0, 0.0);

        // Transición: Si se presiona start, pasar a MOVING
        if (start_button_pressed_) {
          RCLCPP_INFO(this->get_logger(), "Transición: IDLE -> MOVING");
          current_state_ = State::MOVING;
          start_button_pressed_ = false;
        }
        break;

      case State::MOVING:
        // Acción: avanzar (se ejecuta en cada ciclo)
        publish_velocity(0.3, 0.0);

        // Transición: Si detecta obstáculo, pasar a OBSTACLE_DETECTED
        if (min_distance_ < OBSTACLE_THRESHOLD) {
          RCLCPP_WARN(this->get_logger(),
                      "Obstáculo detectado a %.2f m!",
                      min_distance_);
          current_state_ = State::OBSTACLE_DETECTED;
        }
        break;

      case State::OBSTACLE_DETECTED:
        // Acción: frenar (se ejecuta en cada ciclo - no ideal)
        publish_velocity(0.0, 0.0);

        // Transición inmediata a STOPPED
        RCLCPP_INFO(this->get_logger(), "Transición: OBSTACLE_DETECTED ->
     STOPPED");
        current_state_ = State::STOPPED;
        stopped_time_ = this->now();
        break;

      case State::STOPPED:
        // Acción: mantenerse detenido
        publish_velocity(0.0, 0.0);

        // Transición: Si el obstáculo desaparece después de 2 segundos
        auto elapsed = (this->now() - stopped_time_).seconds();
        if (min_distance_ > OBSTACLE_THRESHOLD && elapsed > 2.0) {
          RCLCPP_INFO(this->get_logger(),
                      "Obstáculo despejado. Transición: STOPPED -> MOVING"
     );
          current_state_ = State::MOVING;
        }
        break;
    }
}

void BasicFSMRobot::laser_callback(const sensor_msgs::msg::LaserScan::
     SharedPtr msg)
{
  // Callback de sensor: SOLO actualiza memoria, NO toma decisiones
```

```
60   if (!msg->ranges.empty()) {
61     min_distance_ = *std::min_element(msg->ranges.begin(), msg->ranges.end
       ());
62   }
63 }
```

Fragmento C++ 4.2: Implementación del ciclo de control (fsm_basic_example.cpp)

Nota importante: Este patrón no implementa el ciclo de vida completo (`on_entry`, `on_do`, `on_exit`), lo que puede provocar que las acciones se ejecuten repetidamente. Es adecuado para prototipos rápidos pero no para sistemas complejos.

Patrón orientado a objetos

Para FSM más complejas, encapsular cada estado en una clase separada permite implementar correctamente el ciclo de vida.

El fragmento de código 4.3 muestra la interfaz base `State`. Todos los estados concretos heredan de esta clase y deben implementar:

- `on_entry()`: se ejecuta *una sola vez* al entrar al estado.
- `on_do()`: se ejecuta en cada ciclo mientras el estado esté activo.
- `on_exit()`: se ejecuta *una sola vez* al salir del estado.
- `check_transitions()`: evalúa condiciones y retorna el siguiente estado o `nullptr`.
- `get_name()`: retorna el nombre del estado para logging.

```
1  class State
2  {
3  public:
4    virtual void on_entry() {}
5    virtual void on_do() = 0;
6    virtual void on_exit() {}
7    virtual State* check_transitions() = 0;
8    virtual ~State() = default;
9
10   virtual std::string get_name() const = 0;
11 };
```

Fragmento C++ 4.3: Interfaz base para estados (fsm_oop_example.hpp)

El fragmento de código 4.4 implementa la clase `StateMachine`, que orquesta los cambios de estado. El método `step()` sigue un patrón claro:

1. Ejecutar `on_do()` del estado actual.
2. Llamar a `check_transitions()` para evaluar si hay transición.
3. Si hay transición: ejecutar `on_exit()`, destruir el estado anterior, crear el nuevo estado y ejecutar su `on_entry()`.

Esta arquitectura garantiza que `on_entry` y `on_exit` se ejecuten exactamente una vez por transición.

```
1  class StateMachine
2  {
3    State* current_state_;
4    rclcpp::Logger logger_;
5
6  public:
7    StateMachine(State* initial_state, rclcpp::Logger logger)
8      : current_state_(initial_state), logger_(logger)
9    {
10     RCLCPP_INFO(logger_, "FSM iniciada en estado: %s",
11                 current_state_->get_name().c_str());
12     current_state_->on_entry();
13   }
14
15   void step() {
```

```cpp
    // 1. Ejecutar lógica del estado actual
    current_state_->on_do();

    // 2. Verificar transiciones
    State* next_state = current_state_->check_transitions();

    // 3. Si hay transición, ejecutar salida, cambiar y ejecutar entrada
    if (next_state != nullptr) {
      RCLCPP_INFO(logger_, "Transición: %s -> %s",
                  current_state_->get_name().c_str(),
                  next_state->get_name().c_str());

      current_state_->on_exit();
      delete current_state_;
      current_state_ = next_state;
      current_state_->on_entry();
    }
  }
};
```

Fragmento C++ 4.4: Implementación de la máquina de estados (fsm_oop_-example.cpp)

Los fragmentos de código 4.5 y 4.6 presentan dos estados concretos como ejemplos.

En `IdleState` (fragmento de código 4.5):

- `on_entry()` detiene los motores *una sola vez* al entrar.
- `on_do()` no hace nada (el robot simplemente espera).
- `check_transitions()` verifica si se ha presionado el botón de inicio y, de ser así, retorna un nuevo `MovingState`.
- `on_exit()` prepara al robot para el siguiente estado.

```cpp
class IdleState : public State
{
  OOPFSMRobot* robot_;
public:
  explicit IdleState(OOPFSMRobot* r) : robot_(r) {}

  void on_entry() override {
    RCLCPP_INFO(robot_->get_logger(),
                "[IDLE] Entrando al estado - Deteniendo motores");
    robot_->publish_velocity(0.0, 0.0);
  }

  void on_do() override {
    // No hace nada, simplemente espera
  }

  State* check_transitions() override {
    if (robot_->is_start_pressed()) {
      robot_->clear_start_button();
      return new MovingState(robot_);
    }
    return nullptr;  // Sin transición
  }

  void on_exit() override {
    RCLCPP_INFO(robot_->get_logger(),
                "[IDLE] Saliendo del estado - Preparando para mover");
  }

  std::string get_name() const override { return "IDLE"; }
};
```

Fragmento C++ 4.5: Ejemplo de estado concreto: IdleState (fsm_oop_-example.cpp)

En `MovingState` (fragmento de código 4.6), la diferencia clave está en:

- `on_do()` publica velocidad en cada ciclo (acción continua).
- `check_transitions()` evalúa la distancia al obstáculo y retorna `StoppedState` si detecta uno cercano.

- `on_exit()` es *crítico*: detiene los motores antes de cambiar de estado, evitando que el robot siga moviéndose durante la transición.

```
class MovingState : public State
{
  OOPFSMRobot* robot_;
public:
  explicit MovingState(OOPFSMRobot* r) : robot_(r) {}

  void on_entry() override {
    RCLCPP_INFO(robot_->get_logger(),
                "[MOVING] Entrando al estado - Iniciando movimiento");
  }

  void on_do() override {
    // Ejecutar acción continua: avanzar
    robot_->publish_velocity(0.3, 0.0);
  }

  State* check_transitions() override {
    if (robot_->get_min_distance() < robot_->get_obstacle_threshold()) {
      RCLCPP_WARN(robot_->get_logger(),
                  "[MOVING] Obstáculo detectado a %.2f m!",
                  robot_->get_min_distance());
      return new StoppedState(robot_);
    }
    return nullptr;
  }

  void on_exit() override {
    RCLCPP_INFO(robot_->get_logger(),
                "[MOVING] Saliendo del estado - Iniciando frenado");
    // CRÍTICO: Detener motores antes de cambiar de estado
    robot_->publish_velocity(0.0, 0.0);
  }

  std::string get_name() const override { return "MOVING"; }
};
```

Fragmento C++ 4.6: Ejemplo de estado MovingState (fsm_oop_example.cpp)

Este patrón garantiza que `on_entry` y `on_exit` se ejecuten exactamente una vez por transición, permitiendo inicialización y limpieza seguras de recursos.

Como se puede observar, cada estado concreto (`IdleState`, `MovingState`, `StoppedState`) mantiene un puntero `OOPFSMRobot* robot_`. Este puntero referencia al **contexto** en el patrón State, término que designa al sistema completo que la FSM controla.

La clase `OOPFSMRobot` (Object-Oriented Programming FSM Robot) representa el robot completo desde el punto de vista del sistema de control. Los estados necesitan acceso a esta clase por tres motivos:

1. **Lectura de información sensorial.** Los métodos `check_transitions()` requieren acceso a datos sensoriales para evaluar condiciones de transición. Por ejemplo, `MovingState` consulta `robot_->get_min_distance()` para detectar obstáculos y decidir si debe cambiar a `StoppedState`.
2. **Ejecución de acciones sobre actuadores.** Las acciones definidas en `on_entry()`, `on_do()` y `on_exit()` deben controlar los actuadores del robot. Por ejemplo, `MovingState::on_do()` invoca `robot_->publish_velocity(0.3, 0.0)` para comandar velocidad lineal. Sin acceso al contexto, los estados carecerían de capacidad de acción.
3. **Acceso a recursos compartidos.** Los estados acceden a funcionalidades del nodo ROS 2 mediante el contexto: logging con `robot_-`

->get_logger(), lectura de parámetros como robot_->get_obstacle_-threshold(), entre otros.

La arquitectura del patrón State mantiene la clase base State genérica y abstracta, sin conocimiento del robot específico. Solo los estados concretos almacenan la referencia al contexto. Cuando un estado debe realizar una transición, crea la nueva instancia de estado pasándole el mismo puntero al contexto, como ilustra el fragmento de código 4.7:

```
State* MovingState::check_transitions() override {
  if (robot_->get_min_distance() < robot_->get_obstacle_threshold()) {
    return new StoppedState(robot_);  // Mismo contexto
  }
  return nullptr;
}
```

Fragmento C++ 4.7: Transmisión del contexto entre estados (fsm_oop_-example.cpp)

De esta forma, todos los estados comparten acceso a la misma instancia de OOPFSMRobot, garantizando consistencia en el acceso a sensores, actuadores y estado del sistema.

La clase OOPFSMRobot cumple cuatro funciones esenciales en esta arquitectura:

- **Contener la máquina de estados:** Mantiene un atributo StateMachine* fsm_ que gestiona el estado actual y las transiciones.
- **Almacenar el estado del sistema:** Variables como min_distance_- (distancia al obstáculo más cercano), start_button_pressed_- (señal de inicio), y constantes como OBSTACLE_THRESHOLD (umbral de detección).
- **Encapsular la comunicación ROS 2:** Gestiona suscriptores (laser_-sub_), publicadores (vel_pub_), y temporizadores (timer_).
- **Proporcionar interfaz controlada:** Métodos públicos como get_-min_distance(), publish_velocity(), get_logger() permiten que los estados accedan a la funcionalidad del robot manteniendo el encapsulamiento.

Un error común es gestionar la FSM dentro de callbacks de sensores. La arquitectura correcta se basa en un **timer periódico**.

El fragmento de código 4.8 muestra la implementación completa de OOPFSMRobot. Los puntos clave son:

- El constructor configura suscripciones (laser_sub_), publicadores (vel_pub_) y un timer a 10Hz.
- laser_callback() solo actualiza min_distance_; no toma decisiones ni cambia estados.
- control_cycle() llama a fsm_->step() en cada ciclo del timer. Aquí es donde la FSM decide.
- La FSM se inicializa con new StateMachine(new IdleState(this), ...), estableciendo el estado inicial.
- Los métodos públicos (get_min_distance(), etc.) permiten que los estados accedan a la información del robot sin romper el encapsulamiento.

```
class OOPFSMRobot : public rclcpp::Node
{
  StateMachine* fsm_;

  // Variables de estado (memoria compartida entre estados)
  double min_distance_ = 10.0;
  const double OBSTACLE_THRESHOLD = 0.5;
  bool start_button_pressed_ = false;

```

```
  rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr laser_sub_;
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr vel_pub_;
  rclcpp::TimerBase::SharedPtr timer_;

public:
  OOPFSMRobot() : Node("oop_fsm_robot")
  {
    // 1. Suscripciones a sensores
    laser_sub_ = this->create_subscription<sensor_msgs::msg::LaserScan>(
      "/scan", 10,
      std::bind(&OOPFSMRobot::laser_callback, this, std::placeholders::_1)
    );

    // 2. Publicadores de comandos
    vel_pub_ = this->create_publisher<geometry_msgs::msg::Twist>("/cmd_vel
    ", 10);

    // 3. Timer para el ciclo de control (10Hz)
    timer_ = this->create_wall_timer(
      std::chrono::milliseconds(100),
      std::bind(&OOPFSMRobot::control_cycle, this));

    // 4. Inicializar FSM en estado IDLE
    fsm_ = new StateMachine(new IdleState(this), this->get_logger());
  }

private:
  void laser_callback(const sensor_msgs::msg::LaserScan::SharedPtr msg) {
    // Callback de sensor: SOLO actualiza memoria, NO toma decisiones
    if (!msg->ranges.empty()) {
      min_distance_ = *std::min_element(msg->ranges.begin(), msg->ranges.
    end());
    }
  }

  void control_cycle() {
    // Callback del timer: AQUÍ la FSM toma decisiones
    fsm_->step();
  }

  // Interfaz pública para que los estados accedan a la información del
     robot
  double get_min_distance() const { return min_distance_; }
  double get_obstacle_threshold() const { return OBSTACLE_THRESHOLD; }
  bool is_start_pressed() const { return start_button_pressed_; }
  void publish_velocity(double linear, double angular);
};
```

Fragmento C++ 4.8: Nodo ROS 2 con FSM orientada a objetos (fsm_oop_example.cpp)

La arquitectura separa claramente responsabilidades entre callbacks. Los callbacks de sensores (como `laser_callback()`) únicamente actualizan variables internas del nodo, es decir, mantienen la memoria del sistema sin ejecutar lógica de decisión. El callback del timer (`control_cycle()`) implementa el ciclo de decisión: ejecuta la FSM que observa el estado actualizado de la memoria y determina acciones y transiciones. Esta separación garantiza que el nodo permanezca reactivo: si la FSM se bloquea, los callbacks de sensores pueden seguir procesándose.

Descomposición ortogonal

Los ejemplos anteriores muestran cómo implementar una FSM que controla un único aspecto del comportamiento del robot (navegación básica con detección de obstáculos). Sin embargo, en sistemas reales, el robot debe gestionar múltiples preocupaciones simultáneas e independientes. Por ejemplo, mientras navega, debe monitorizar continuamente el nivel de batería.

Como se explica en el Capítulo 7, intentar modelar comportamientos ortogonales con una FSM plana provoca una explosión combinatoria de estados. Si la navegación tiene 3 estados (`IDLE`, `MOVING`, `STOPPED`) y la batería tiene 3 estados (`OK`, `LOW`, `CRITICAL`), una FSM plana requeriría $3 \times 3 = 9$ estados combinados: `IDLE_BAT_OK`, `IDLE_BAT_LOW`, `MOVING_BAT_CRITICAL`, etc.

La solución, propuesta por David Harel en su formalismo de *statecharts*, consiste en descomponer el sistema en **regiones ortogonales**: múltiples FSM independientes que se ejecutan concurrentemente. El paquete `fsm_examples` incluye una implementación completa de este patrón.

El fragmento de código 4.9 muestra la implementación de una máquina de estados genérica mediante plantillas C++. Esta clase `GenericStateMachine<T>` puede gestionar cualquier tipo de estado (navegación, batería, luces, etc.), siempre que el tipo `T` proporcione los métodos necesarios: `on_entry()`, `on_do()`, `on_exit()`, `check_transitions()` y `get_name()`. Tanto `NavState` como `BatteryState` implementan estos métodos, lo que permite crear instancias `GenericStateMachine<NavState>` y `GenericStateMachine<BatteryState>`. La clave está en que cada región mantiene su propia máquina de estados independiente.

```cpp
template<typename T>
class GenericStateMachine
{
  T* current_state_;
  rclcpp::Logger logger_;
  std::string region_name_;

public:
  GenericStateMachine(T* initial_state, rclcpp::Logger logger,
                      const std::string& region_name)
    : current_state_(initial_state), logger_(logger), region_name_(
     region_name)
  {
    current_state_->on_entry();
  }

  void step() {
    current_state_->on_do();
    T* next_state = current_state_->check_transitions();

    if (next_state != nullptr) {
      RCLCPP_INFO(logger_, "[%s] Transición: %s -> %s",
                  region_name_.c_str(),
                  current_state_->get_name().c_str(),
                  next_state->get_name().c_str());

      current_state_->on_exit();
      delete current_state_;
      current_state_ = next_state;
      current_state_->on_entry();
    }
  }
};
```

Fragmento C++ 4.9: Máquina de estados genérica con plantillas (fsm_orthogonal_example.hpp)

El fragmento de código 4.10 presenta la interfaz y estados concretos para la región de navegación. Observe cómo esta FSM es idéntica en estructura a la FSM orientada a objetos vista anteriormente, pero ahora está claramente delimitada como una región independiente mediante el prefijo `Nav` en nombres de clases.

```cpp
// Interfaz base para estados de navegación
class NavState
{
public:
  virtual void on_entry() {}
```

```cpp
  virtual void on_do() = 0;
  virtual NavState* check_transitions() = 0;
  virtual void on_exit() {}
  virtual ~NavState() = default;
  virtual std::string get_name() const = 0;
};

// Estados concretos de navegación
class NavIdleState : public NavState { /* ... */ };
class NavMovingState : public NavState { /* ... */ };
class NavStoppedState : public NavState { /* ... */ };
```

Fragmento C++ 4.10: Estados de la región de navegación (fsm_orthogonal_-example.hpp)

El fragmento de código 4.11 muestra la segunda región ortogonal: el monitoreo de batería. Esta FSM opera completamente independiente de la navegación, con su propia jerarquía de estados. La batería puede estar en tres niveles (`OK`, `LOW`, `CRITICAL`), y las transiciones se basan exclusivamente en porcentajes de carga.

```cpp
// Interfaz base para estados de batería
class BatteryState
{
public:
  virtual void on_entry() {}
  virtual void on_do() = 0;
  virtual BatteryState* check_transitions() = 0;
  virtual void on_exit() {}
  virtual ~BatteryState() = default;
  virtual std::string get_name() const = 0;
};

// Estados concretos de batería
class BatteryOKState : public BatteryState { /* ... */ };
class BatteryLowState : public BatteryState { /* ... */ };
class BatteryCriticalState : public BatteryState { /* ... */ };
```

Fragmento C++ 4.11: Estados de la región de batería (fsm_orthogonal_-example.hpp)

El fragmento de código 4.12 muestra cómo el nodo `OrthogonalRobot` integra ambas regiones. Lo fundamental de esta arquitectura es que el método `control_cycle()` ejecuta ambas FSM en cada iteración: primero `nav_fsm_->step()`, luego `battery_fsm_->step()`. Ambas máquinas operan sobre el mismo contexto (el robot), pero gestionan aspectos ortogonales del comportamiento.

```cpp
class OrthogonalRobot : public rclcpp::Node
{
  // Dos FSM independientes (regiones ortogonales)
  GenericStateMachine<NavState>* nav_fsm_;
  GenericStateMachine<BatteryState>* battery_fsm_;

  // Variables compartidas por ambas regiones
  double min_distance_ = 10.0;
  double battery_percentage_ = 100.0;
  bool start_button_pressed_ = false;
  double top_speed_ = 0.3;  // Velocidad máxima permitida (modulada por
     batería)

  const double OBSTACLE_THRESHOLD = 0.5;
  const double BATTERY_LOW_THRESHOLD = 30.0;
  const double BATTERY_CRITICAL_THRESHOLD = 10.0;

public:
  OrthogonalRobot() : Node("orthogonal_robot")
  {
    // Configurar suscripciones y publicadores
    laser_sub_ = this->create_subscription<sensor_msgs::msg::LaserScan
     >(...);
    battery_sub_ = this->create_subscription<sensor_msgs::msg::
     BatteryState>(...);
    vel_pub_ = this->create_publisher<geometry_msgs::msg::Twist>(...);

```

```
    // Inicializar ambas FSM en sus estados iniciales
    nav_fsm_ = new GenericStateMachine<NavState>(
      new NavIdleState(this), this->get_logger(), "NAVEGACIÓN");

    battery_fsm_ = new GenericStateMachine<BatteryState>(
      new BatteryOKState(this), this->get_logger(), "BATERÍA");
  }

private:
  void control_cycle() {
    // CLAVE: Batería se ejecuta primero para actualizar restricciones (
     top_speed_)
    // antes de que navegación genere comandos
    battery_fsm_->step();
    nav_fsm_->step();
  }
};
```

Fragmento C++ 4.12: Nodo con FSM ortogonales (fsm_orthogonal_-example.cpp)

Esta arquitectura resuelve la explosión combinatoria: en lugar de $3 \times 3 = 9$ estados combinados, tenemos $3 + 3 = 6$ estados independientes. Si añadimos una tercera región (por ejemplo, luces del robot con 2 estados), pasamos a $3 + 3 + 2 = 8$ estados en lugar de $3 \times 3 \times 2 = 18$. El crecimiento es lineal en lugar de exponencial.

El fragmento de código 4.13 muestra un detalle arquitectónico importante: cómo las regiones ortogonales pueden coordinarse sin romper la independencia. Las regiones comparten una variable `top_speed_` que actúa como restricción: la región de batería modula esta velocidad máxima (normal=0.3, baja=0.15, crítica=0.0), mientras que la región de navegación la utiliza para publicar comandos. De esta forma, la región de batería impone límites de seguridad sin necesidad de conocer el estado de navegación, y la región de navegación respeta estos límites sin necesidad de consultar el nivel de batería.

```
class BatteryCriticalState : public BatteryState
{
  OrthogonalRobot* robot_;
public:
  explicit BatteryCriticalState(OrthogonalRobot* r) : robot_(r) {}

  void on_entry() override {
    RCLCPP_ERROR(robot_->get_logger(),
                 "[BAT] BATERÍA CRÍTICA - Deteniendo operaciones");
    // Imponer velocidad máxima cero por seguridad
    robot_->set_top_speed(0.0);
  }

  void on_do() override {
    // Mantener restricción de velocidad cero
    robot_->set_top_speed(0.0);
  }

  BatteryState* check_transitions() override {
    if (robot_->get_battery_percentage() >
        robot_->get_battery_critical_threshold() + 5.0) {
      return new BatteryLowState(robot_);
    }
    return nullptr;
  }

  std::string get_name() const override { return "BAT:CRITICAL"; }
};
```

Fragmento C++ 4.13: Interacción entre regiones ortogonales (fsm_orthogonal_-example.cpp)

Esta técnica mantiene la ortogonalidad mediante coordinación indirecta: la región de batería no necesita conocer si el robot está navegando, parado o evitando obstáculos; simplemente establece una restricción (velocidad máxima permitida). La región de navegación, por su parte, no necesita

consultar el nivel de batería; simplemente respeta la velocidad máxima disponible. El patrón es escalable: si se añade una tercera región (por ejemplo, modo de mantenimiento), también podría modular `top_speed_` sin afectar a la lógica de navegación o batería. La clave arquitectónica es que `battery_fsm_->step()` se ejecuta antes que `nav_fsm_->step()` en `control_cycle()`, asegurando que las restricciones se actualicen antes de generar comandos.

Capacidad vs misión

En términos arquitectónicos, la capacidad puede ser cualquier componente con una interfaz clara y una semántica compatible con la misión. La explicación del capítulo se apoya en un ejemplo concreto para ilustrar el patrón, pero el enfoque es aplicable a otras capacidades equivalentes.

En este ejercicio se exige una separación clara:

- **Capacidad**: Por ejemplo, "ir a una pose objetivo en el mapa".
- **Misión**: "Ejecutar una secuencia de objetivos siguiendo una política" (por ejemplo, patrullaje, inspección periódica o reparto interno).

La capacidad se consume mediante una interfaz de alto nivel adecuada a su naturaleza (por ejemplo, acción, servicio, topic o llamada a API). En este capítulo se usa una *acción* en el caso de Nav2. La tarea (FSM) no tiene por qué conocer detalles internos de la capacidad concreta; solo:

- enviar un objetivo,
- recibir estado o feedback si la interfaz lo permite,
- gestionar resultado.

Este patrón fija un contrato arquitectónico estable y reutilizable.

Interfaz de capacidad: patrón general y ejemplo con NavigateToPose

Una capacidad puede exponerse mediante distintos mecanismos de interacción (por ejemplo, *acciones*, servicios, topics o APIs). En este apartado se presenta un ejemplo concreto basado en una *acción* para ilustrar el patrón de integración desde la FSM.

Nav2 proporciona la acción `NavigateToPose` (tipo `nav2_msgs::action::NavigateToPose`) para comandar navegación a una pose objetivo. Esta sección explica cómo crear un cliente de acción, enviar objetivos, procesar feedback y gestionar resultados. El código presentado está diseñado para ser reutilizado directamente en la implementación de una FSM de misión.

Estructura básica del cliente (ejemplo con acción)

El fragmento de código 4.14 muestra la estructura mínima de un nodo que actúa como cliente de la acción de navegación. Los elementos clave son:

- **Alias de tipos**: Se definen alias (`using`) para simplificar la nomenclatura de los tipos relacionados con la acción.
- **Cliente de acción**: Se crea mediante `rclcpp_action::create_client`, especificando el tipo de acción y el nombre del servidor (`navigate_to_pose`).

- **Goal handle**: Almacena la referencia al objetivo enviado, necesaria para consultar estado y cancelar si es necesario.
- **Variables de control**: Flags booleanos que permiten a la FSM conocer el estado de la navegación sin bloquear la ejecución.

```cpp
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "nav2_msgs/action/navigate_to_pose.hpp"
#include "geometry_msgs/msg/pose_stamped.hpp"
#include "tf2/LinearMath/Quaternion.h"

// NavigationClient: Cliente simplificado para interactuar con Nav2
//
// DECISIÓN DE DISEÑO - Composición con puntero raw:
// - Usa puntero raw (rclcpp::Node*) en lugar de shared_ptr por las siguientes razones:
//   1. El cliente siempre será un miembro de un nodo (nunca vive más que el nodo)
//   2. Permite pasar 'this' directamente en el constructor del nodo padre
//   3. Evita problemas con shared_from_this() que no puede llamarse en constructores
//   4. No hay riesgo de dangling pointer porque el ciclo de vida está garantizado
class NavigationClient
{
public:
  // Alias para simplificar tipos
  using NavigateToPose = nav2_msgs::action::NavigateToPose;
  using GoalHandleNav = rclcpp_action::ClientGoalHandle<NavigateToPose>;

  explicit NavigationClient(rclcpp::Node* node)
  : node_(node)
  {
    // Crear el cliente de acción para comunicarse con Nav2
    // Este cliente encapsula toda la complejidad de comunicación así ncrona
    nav_client_ = rclcpp_action::create_client<NavigateToPose>(
      node_, "navigate_to_pose");

    RCLCPP_DEBUG(node_->get_logger(), "Cliente de navegación inicializado");
  }

  // Método para verificar disponibilidad del servidor
  bool wait_for_action_server(std::chrono::seconds timeout = std::chrono::seconds(5))
  {
    // Método de verificación: asegura que Nav2 está operativo
    // FASE 1: Antes de enviar objetivos, verificar disponibilidad de la capacidad
    if (!nav_client_->wait_for_action_server(timeout)) {
      RCLCPP_ERROR(node_->get_logger(),
                   "Servidor de navegación no disponible tras espera");
      return false;
    }
    RCLCPP_DEBUG(node_->get_logger(), "Servidor de navegación disponible");
    return true;
  }

private:
  // Nodo de ROS2 que gestiona la comunicación
  // NOTA: Puntero raw porque el cliente siempre es miembro del nodo (lifetime garantizado)
  rclcpp::Node* node_;

  // Cliente de acción
  rclcpp_action::Client<NavigateToPose>::SharedPtr nav_client_;

  // Goal handle (se actualiza al enviar objetivo)
  std::shared_ptr<GoalHandleNav> goal_handle_;
```

```
  // Variables de control para la FSM
  bool goal_active_ = false;     // Objetivo en progreso
  bool goal_done_ = false;       // Objetivo completado (éxito o fallo)
  bool goal_success_ = false;    // true si terminó con éxito

  // Último feedback recibido
  std::shared_ptr<const NavigateToPose::Feedback> last_feedback_;
};
```

Fragmento C++ 4.14: Estructura básica del cliente de NavigateToPose

Envío de objetivos a la capacidad

El fragmento de código 4.15 implementa el método para enviar un objetivo de navegación. Este método constituye el núcleo de la interacción con Nav2 y requiere atención especial a los callbacks:

- **Construcción del mensaje**: Se crea un `NavigateToPose::Goal` con la pose objetivo en el marco `map`.
- **Opciones de envío**: Mediante `SendGoalOptions` se configuran tres callbacks que Nav2 invocará en distintos momentos.
- **Goal response callback**: Se invoca cuando el servidor acepta o rechaza el objetivo. Un objetivo rechazado impide la navegación.
- **Feedback callback**: Se invoca periódicamente durante la navegación, proporcionando información sobre progreso (distancia restante, tiempo estimado, etc.).
- **Result callback**: Se invoca cuando la navegación termina (éxito, fallo o cancelación), permitiendo actualizar el estado de la FSM.

```
void send_goal(const geometry_msgs::msg::PoseStamped& target_pose)
{
  // Resetear flags de control
  goal_active_ = false;
  goal_done_ = false;
  goal_success_ = false;

  // Construir el mensaje de objetivo
  auto goal_msg = NavigateToPose::Goal();
  goal_msg.pose = target_pose;
  goal_msg.pose.header.stamp = this->now();
  goal_msg.pose.header.frame_id = "map";

  RCLCPP_DEBUG(get_logger(),
               "Enviando objetivo: (%.2f, %.2f)",
               target_pose.pose.position.x,
               target_pose.pose.position.y);

  // Configurar opciones con callbacks
  auto send_goal_options = rclcpp_action::Client<NavigateToPose>::
     SendGoalOptions();

  // Callback 1: Respuesta al envío (objetivo aceptado/rechazado)
  send_goal_options.goal_response_callback =
    std::bind(&NavigationClient::goal_response_callback, this,
              std::placeholders::_1);

  // Callback 2: Feedback periódico durante la navegación
  send_goal_options.feedback_callback =
    std::bind(&NavigationClient::feedback_callback, this,
              std::placeholders::_1, std::placeholders::_2);

  // Callback 3: Resultado final de la navegación
  send_goal_options.result_callback =
    std::bind(&NavigationClient::result_callback, this,
              std::placeholders::_1);

  // Enviar el objetivo de forma asíncrona
```

```
  nav_client_->async_send_goal(goal_msg, send_goal_options);
}
```

Fragmento C++ 4.15: Envío de objetivo con callbacks configurados

Gestión de eventos de ejecución

En una arquitectura general, la capacidad expone eventos de progreso y de finalización mediante el mecanismo de interacción elegido. En este ejemplo (basado en una acción de ROS 2), dichos eventos se materializan como callbacks.

El fragmento de código 4.16 muestra la implementación de tres callbacks, cada uno con una responsabilidad específica:

1. **Goal response callback**: Verifica si Nav2 aceptó el objetivo. Un rechazo puede deberse a que Nav2 no está listo, el mapa no está cargado o la pose objetivo es inválida. Si se acepta, almacena el `goal_handle_` para futuras consultas.
2. **Feedback callback**: Recibe actualizaciones periódicas del progreso y almacena el último feedback recibido en `last_feedback_` para que pueda ser consultado externamente mediante `get_feedback()`. El feedback incluye:
   - `distance_remaining`: Distancia estimada al objetivo.
   - `navigation_time`: Tiempo transcurrido desde el inicio.
   - `estimated_time_remaining`: Tiempo estimado para completar.
   - `current_pose`: Pose actual del robot.

   Este callback es opcional pero útil para proporcionar información al usuario o detectar situaciones anómalas (robot atascado, progreso insuficiente).
3. **Result callback**: Procesa el resultado final. El atributo `code` del resultado indica el estado final según el estándar de action_msgs:
   - `SUCCEEDED` (4): Navegación completada con éxito.
   - `ABORTED` (5): Navegación fallida (obstáculo insalvable, timeout, etc.).
   - `CANCELED` (6): Navegación cancelada por solicitud externa.

```
void goal_response_callback(
  const GoalHandleNav::SharedPtr & goal_handle)
{
  if (!goal_handle) {
    RCLCPP_ERROR(get_logger(), "Objetivo rechazado por el servidor");
    goal_done_ = true;
    goal_success_ = false;
    return;
  }

  RCLCPP_DEBUG(get_logger(), "Objetivo aceptado, navegación iniciada");
  goal_handle_ = goal_handle;
  goal_active_ = true;
}

void feedback_callback(
  GoalHandleNav::SharedPtr,
  const std::shared_ptr<const NavigateToPose::Feedback> feedback)
{
  // Almacenar el último feedback recibido
  last_feedback_ = feedback;

  // Feedback periódico: distancia restante, tiempo, etc.
  RCLCPP_DEBUG(get_logger(),
               "Distancia restante: %.2f m | Tiempo: %.1f s",
```

```cpp
            feedback->distance_remaining,
            rclcpp::Duration(feedback->navigation_time).seconds());

  // Aquí se puede implementar lógica adicional:
  // - Detectar robot atascado (distancia no disminuye)
  // - Timeout por exceso de tiempo
  // - Actualizar UI con progreso
}

void result_callback(const GoalHandleNav::WrappedResult & result)
{
  goal_active_ = false;
  goal_done_ = true;

  switch (result.code) {
    case rclcpp_action::ResultCode::SUCCEEDED:
      RCLCPP_DEBUG(get_logger(), "Navegación completada con ÉXITO");
      goal_success_ = true;
      break;

    case rclcpp_action::ResultCode::ABORTED:
      RCLCPP_WARN(get_logger(), "Navegación ABORTADA (obstáculo o timeout)
      ");
      goal_success_ = false;
      break;

    case rclcpp_action::ResultCode::CANCELED:
      RCLCPP_WARN(get_logger(), "Navegación CANCELADA");
      goal_success_ = false;
      break;

    default:
      RCLCPP_ERROR(get_logger(), "Estado desconocido: %d",
                   static_cast<int>(result.code));
      goal_success_ = false;
      break;
  }
}
```

Fragmento C++ 4.16: Implementación de callbacks del cliente de acción

Métodos auxiliares de control, estado y observabilidad

El fragmento de código 4.17 presenta métodos auxiliares que facilitan la integración del cliente de acciones con la FSM de misión:

- **Consulta de estado**: Métodos `is_goal_active()`, `is_goal_done()` y `was_goal_successful()` permiten que la FSM consulte el estado de la navegación sin acceso directo al `goal_handle_`.
- **Obtener feedback**: Método `get_feedback()` devuelve el último feedback recibido, permitiendo acceder a información como distancia restante y tiempo de navegación desde fuera del cliente.
- **Cancelación de objetivo**: Método `cancel_goal()` envía una petición de cancelación al servidor. Útil cuando la FSM decide abortar la navegación (emergencia, cambio de misión, timeout).
- **Creación de poses**: Método `create_pose_stamped()` simplifica la construcción de mensajes de pose objetivo, evitando repetición de código.

```cpp
// Consultar estado del objetivo actual
bool is_goal_active() const { return goal_active_; }
bool is_goal_done() const { return goal_done_; }
bool was_goal_successful() const { return goal_success_; }

// Obtener el último feedback recibido
std::shared_ptr<const NavigateToPose::Feedback> get_feedback() const {
     return last_feedback_; }

```

```cpp
// Cancelar el objetivo en progreso
void cancel_goal()
{
  if (goal_handle_ && goal_active_) {
    RCLCPP_DEBUG(get_logger(), "Cancelando objetivo de navegación");
    nav_client_->async_cancel_goal(goal_handle_);
    goal_active_ = false;
  }
}

// Método auxiliar para crear poses
geometry_msgs::msg::PoseStamped create_pose_stamped(
  double x, double y, double yaw)
{
  geometry_msgs::msg::PoseStamped pose;
  pose.header.frame_id = "map";
  pose.header.stamp = this->now();
  pose.pose.position.x = x;
  pose.pose.position.y = y;
  pose.pose.position.z = 0.0;

  // Convertir yaw a quaternion
  tf2::Quaternion q;
  q.setRPY(0.0, 0.0, yaw);
  pose.pose.orientation.x = q.x();
  pose.pose.orientation.y = q.y();
  pose.pose.orientation.z = q.z();
  pose.pose.orientation.w = q.w();

  return pose;
}
```

Fragmento C++ 4.17: Métodos auxiliares para gestión del cliente

Ejemplo de aplicación: consumo de capacidad en una misión

Esta subsección muestra cómo consumir una capacidad desde una aplicación de nivel superior (tarea o misión) mediante un **adaptador de capacidad**. La idea es encapsular la complejidad del mecanismo de interacción elegido (eventos, estados de ejecución, mensajes de progreso y resultado) y exponer una interfaz clara para la lógica de misión.

En este capítulo, dicho patrón se ilustra con `NavigationClient` sobre Nav2.

En el ejemplo de este capítulo, `NavigationClient` proporciona los siguientes métodos para consumir la capacidad:

- **`wait_for_action_server(timeout)`**: Verifica si el servidor de la capacidad está disponible y listo para recibir objetivos. En este ejemplo, se comprueba el servidor de acción de Nav2. Retorna `true` si el servidor responde dentro del timeout especificado.
- **`create_pose_stamped(x, y, yaw)`**: Método auxiliar para construir un objetivo con el formato requerido por la capacidad de ejemplo (Nav2). En otros casos, esta función se sustituiría por la construcción del tipo de objetivo correspondiente.
- **`send_goal(target_pose)`**: Envía un objetivo al servidor de la capacidad. Este método es asíncrono: retorna inmediatamente sin esperar el resultado. En el ejemplo, configura callbacks para procesar la respuesta, el feedback periódico y el resultado final.
- **`is_goal_active()`**: Consulta si hay un objetivo en progreso. Retorna `true` desde que el servidor acepta el objetivo hasta que se completa (éxito o fallo) o se cancela.
- **`is_goal_done()`**: Consulta si el objetivo actual ha finalizado (éxito, fallo o cancelación). Retorna `true` cuando la navegación termina por cualquier motivo.

- **`was_goal_successful()`**: Consulta si el último objetivo completado terminó con éxito. Solo tiene sentido llamar a este método cuando `is_goal_done()` retorna `true`.
- **`get_feedback()`**: Obtiene el último feedback recibido durante la ejecución. En el ejemplo de navegación, incluye distancia restante, tiempo transcurrido y pose actual. Retorna `nullptr` si aún no se ha recibido feedback.
- **`cancel_goal()`**: Solicita la cancelación del objetivo en progreso. Útil cuando la aplicación decide abortar la ejecución de la capacidad (cambio de misión, emergencia, timeout).

Patrón de uso: composición y consulta no bloqueante

El patrón recomendado para usar un adaptador de capacidad sigue estos principios:

- **Composición sobre herencia**: La aplicación (tarea o misión) contiene un `shared_ptr` al adaptador de capacidad (en este ejemplo, `NavigationClient`), no hereda de él. Esto separa la lógica de la capacidad de la lógica de aplicación (secuenciación, recuperación, política).
- **Consulta no bloqueante**: Los métodos de consulta (`is_goal_active()`, `is_goal_done()`, etc.) retornan inmediatamente con el estado actual. El nodo puede seguir procesando otros callbacks mientras consulta periódicamente el progreso.
- **Timer para control de flujo**: Se utiliza un timer periódico para implementar el ciclo de control de la aplicación, que verifica el estado de la capacidad y toma decisiones sin bloquear.
- **Separación de fases**: La aplicación progresa por fases claramente definidas: verificar disponibilidad del servidor → enviar objetivo → monitorizar progreso → procesar resultado. Cada fase utiliza métodos específicos del adaptador.

El fragmento de código 4.18 muestra un ejemplo completo que implementa estos principios. Observe cómo cada fase del `control_cycle()` utiliza métodos distintos del `NavigationClient` para consumir la capacidad de navegación:

```
class SimpleNavigationApp : public rclcpp::Node
{
public:
  SimpleNavigationApp() : Node("simple_navigation_app")
  {
    // 1. Composición: La aplicación TIENE UN NavigationClient (no hereda)
    //    Patrón: Usar capacidades como componentes en lugar de herencias
     múltiples
    //    Uso de 'this': Pasamos el puntero del nodo directamente al
     cliente
    //    - Es seguro porque el cliente es miembro del nodo (mismo ciclo
     de vida)
    //    - No podemos usar shared_from_this() en el constructor
    //    - El cliente usará este nodo para crear clientes, timers,
     logging, etc.
    nav_client_ = std::make_shared<NavigationClient>(this);

    // 2. Usar create_pose_stamped() para construir el objetivo
    //    Método auxiliar: simplifica crear poses sin quaterniones
     manuales
    target_pose_ = nav_client_->create_pose_stamped(6.0, -2.0, 0.0);

    RCLCPP_INFO(get_logger(), "Aplicación de navegación iniciada");
```

```
19
20    // 3. Timer periódico: patron NO BLOQUEANTE
21    //    Permite que el nodo siga procesando callbacks mientras consulta
      estado
22    timer_ = create_wall_timer(
23      std::chrono::milliseconds(500),
24      std::bind(&SimpleNavigationApp::control_cycle, this));
25  }
26
27 private:
28  std::shared_ptr<NavigationClient> nav_client_;  // Adaptador de
      capacidad
29  geometry_msgs::msg::PoseStamped target_pose_;
30  rclcpp::TimerBase::SharedPtr timer_;
31  bool server_ready_ = false;
32  bool goal_sent_ = false;
33
34  void control_cycle()
35  {
36    // FASE 1: Verificar disponibilidad de la CAPACIDAD
37    // Usa: wait_for_action_server(timeout)
38    // Propósito: Asegurar que Nav2 está listo antes de enviar objetivos
39    if (!server_ready_) {
40      if (nav_client_->wait_for_action_server(std::chrono::seconds(1))) {
41        RCLCPP_INFO(get_logger(), "Servidor disponible, preparado para
      navegar");
42        server_ready_ = true;
43      }
44      return;  // Seguir esperando en próxima iteración
45    }
46
47    // FASE 2: Activar la CAPACIDAD (enviar objetivo)
48    // Usa: send_goal(target_pose)
49    // Propósito: Solicitar navegación de forma asíncrona (no bloquea)
50    if (!goal_sent_) {
51      RCLCPP_INFO(get_logger(), "Enviando objetivo de navegación...");
52      nav_client_->send_goal(target_pose_);
53      goal_sent_ = true;
54      return;
55    }
56
57    // FASE 3: Monitorizar PROGRESO de la capacidad
58    // Usa: is_goal_done() para verificar si la capacidad terminó
59    // Usa: get_feedback() para obtener información de progreso
60    if (!nav_client_->is_goal_done()) {
61      // Objetivo en progreso: consultar feedback (opcional pero útil)
62      auto feedback = nav_client_->get_feedback();
63      if (feedback) {
64        RCLCPP_INFO(get_logger(),
65                    "\t-Distancia restante: %.2f m | Tiempo: %.1f s",
66                    feedback->distance_remaining,
67                    rclcpp::Duration(feedback->navigation_time).seconds())
      ;
68      }
69      return;
70    }
71
72    // FASE 4: Procesar RESULTADO de la capacidad
73    // Usa: was_goal_successful() para distinguir éxito/fallo
74    // Propósito: Tomar decisiones según el resultado (continuar,
      reintentar, etc.)
75    if (nav_client_->was_goal_successful()) {
76      RCLCPP_INFO(get_logger(), "Navegación completada con éxito");
77    } else {
78      RCLCPP_WARN(get_logger(), "Navegación fallida");
79    }
80
81    // Detener el timer, tarea completada
82    timer_->cancel();
83    RCLCPP_INFO(get_logger(), "Aplicación finalizada");
84  }
85 };
```

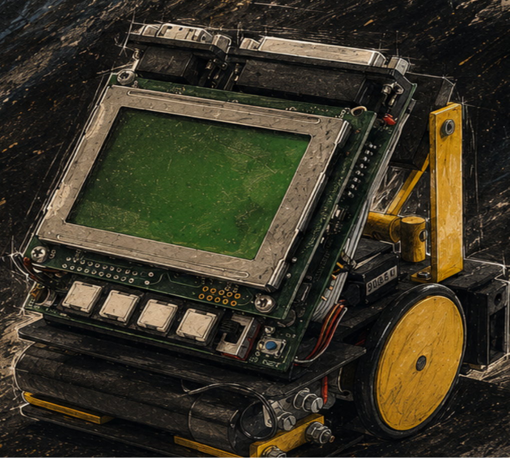
```
86
87 int main(int argc, char** argv)
88 {
89   rclcpp::init(argc, argv);
90   auto app = std::make_shared<SimpleNavigationApp>();
91   rclcpp::spin(app);
92   rclcpp::shutdown();
93   return 0;
94 }
```

Fragmento C++ 4.18: Ejemplo de aplicación que usa el cliente de navegación

Este ejemplo ilustra principios clave para consumir capacidades en arquitecturas robóticas:

1. **NavigationClient como adaptador de capacidad**: El cliente encapsula toda la complejidad de comunicación con Nav2 (gestión de callbacks, estados internos, conversión de formatos). La aplicación solo ve una interfaz simple de métodos de consulta y comando. Este patrón permite:
   - Reutilizar el cliente en múltiples aplicaciones (patrullaje, inspección, reparto, exploración, seguimiento).
   - Testear la lógica de la aplicación sin necesidad de lanzar Nav2 (usando un mock del cliente).
   - Cambiar detalles de implementación del cliente sin afectar a las aplicaciones.
2. **Separación clara entre lógica de capacidad y lógica de aplicación**:
   - **NavigationClient** sabe cómo comunicarse con Nav2 (callbacks, manejo de estados, timeouts).
   - **SimpleNavigationApp** sabe qué hacer con la capacidad (cuándo activarla, cómo interpretar resultados, qué hacer ante fallos).

   Esta separación es fundamental en el modelo misión–tarea–capacidad.
3. **Uso de métodos de consulta no bloqueantes**: Los métodos `is_goal_active()`, `is_goal_done()` y `was_goal_successful()` permiten implementar lógica de alto nivel sin bloquear el nodo. Esto es esencial porque:
   - El nodo puede seguir procesando callbacks de otros sensores o servicios.
   - La aplicación puede implementar timeouts personalizados chequeando el tiempo transcurrido.
   - Se pueden consultar múltiples capacidades en paralelo (navegación + manipulación + percepción).
4. **Progresión por fases sin FSM explícita**: Este ejemplo usa flags booleanos (`server_ready_`, `goal_sent_`) para gestionar el flujo. Para aplicaciones más complejas (misiones con múltiples objetivos, recuperación ante fallos, reintentos), conviene implementar una FSM explícita que use el mismo patrón:
   - Estado `INIT`: Espera servidor usando `wait_for_action_server()`.
   - Estado `SEND_GOAL`: Invoca `send_goal()` con el objetivo actual.
   - Estado `NAVIGATING`: Monitoriza con `is_goal_done()` y `get_feedback()`.
   - Estado `SUCCESS/FAILURE`: Decide siguiente acción según `was_goal_successful()`.



5 Coordinación de misiones con Behavior Trees

Este ejercicio se centra en diseñar e implementar una misión de servicio con Behavior Trees, integrando varias capacidades robóticas mediante interfaces limpias.

Desde el punto de vista arquitectónico, en este ejercicio se trabaja en:

- Distinguir explícitamente entre *misión* (BT global) y *capacidades* (navegación, diálogo, acciones simuladas).
- Diseñar interfaces limpias entre el BT y las capacidades, favoreciendo la reutilización de nodos.
- Construir un sistema observable, donde el robot publica información relevante sobre su estado y acciones.

Desde el punto de vista técnico, se adquirirán competencias en:

- Selección e integración de las capacidades necesarias según los objetivos del ejercicio.
- Construcción y ejecución de Behavior Trees con nodos reutilizables.
- Definición de interfaces claras entre el BT y las capacidades para favorecer modularidad y mantenimiento.
- Simulación, abstracción o encapsulación de pasos complejos de misión mediante nodos parametrizables.

Es importante destacar que en este ejercicio, con el fin de simplificar su diseño, la misión que se implementa no requiere ningún tipo de orquestación compleja mediante FSM: el Behavior Tree modela directamente el flujo completo de la misión invocando las capacidades necesarias. Esta simplicidad permite centrarse en la integración y separación clara entre misión y capacidades, sin introducir complejidad arquitectónica innecesaria.

Este ejercicio se apoya principalmente en los conceptos introducidos en varios capítulos de la Parte I, en particular:

- Capítulo 1: separación entre misión y capacidades, y diseño modular de interfaces.
- Capítulo 2: modelo de ejecución y comunicación necesarios para integrar capacidades en un sistema ROS 2.
- Capítulo 6: modelado de comportamiento mediante Behavior Trees y reutilización de nodos.

5.1 Objetivo **194**
5.2 Guión de desarrollo . . . **194**
5.3 Teoría y herramientas para el ejercicio **196**
Asignación de comportamiento a arquitectura . . . 196
Behavior Trees: secuencias, selectores y nodos hoja . . 196
Nodos reutilizables: patrones mínimos 196
5.4 Ejemplo de referencia: Bump-and-Go con Behavior Trees **197**
Diseño del árbol 197
Comunicación entre nodos mediante puertos y blackboard 198
Organización del paquete de referencia 198
Implementación: Definición XML del árbol 199
Implementación: Nodos personalizados 201
Resumen de conceptos clave 208
5.5 Ejemplo de capacidad de interacción **209**

El objetivo no es combinar varios niveles de coordinación, sino utilizar un único BT de misión para orquestar de forma clara las capacidades seleccionadas y hacer observable el flujo completo del comportamiento.

5.1. Objetivo

El objetivo es construir una misión completa mediante un Behavior Tree, definiendo una secuencia coherente de fases que integre las capacidades seleccionadas para el ejercicio.

De forma general, la misión debe contemplar el siguiente ciclo de alto nivel:

1. Espera o activación por evento de inicio.
2. Obtención de la información necesaria para ejecutar la misión.
3. Ejecución de una o varias fases operativas mediante las capacidades integradas.
4. Verificación de resultado y gestión de posibles incidencias.
5. Cierre de la misión y retorno a estado de reposo o disponibilidad.

La misión completa se implementa como un único Behavior Tree que orquesta estos pasos de forma secuencial. Algunos pasos utilizan capacidades reales (en este ejemplo, navegación con Nav2 y diálogo) mientras que otros son simulados (recoger y entregar). El BT invoca directamente las capacidades sin necesidad de una capa de coordinación FSM adicional, ya que la lógica de la misión es lineal y puede expresarse de forma natural con los nodos de control de BehaviorTree.CPP.

5.2. Guión de desarrollo

El ejercicio se desarrolla siguiendo una secuencia incremental de pasos, donde cada paso produce un sistema funcional y verificable antes de proceder con el siguiente. Conviene completar cada paso de forma ordenada y validar su correcto funcionamiento antes de continuar.

Posibles ejercicios

La misma estructura de trabajo propuesta en este capítulo puede aplicarse a distintos ejercicios, siempre que se mantenga la separación misión–capacidad y la coordinación mediante Behavior Trees. A continuación se sugieren opciones que pueden adaptarse según el contexto de aprendizaje autónomo:

- **Robot camarero**: detecta presencia de una persona, toma el pedido por diálogo, navega a una ubicación de recogida, confirma la entrega y vuelve a una posición base.
- **Reparto interno**: recibe una solicitud de transporte entre dos estaciones, navega al punto de origen, confirma recogida, navega al destino y confirma entrega.
- **Inspección guiada**: recorre una secuencia fija de zonas (por ejemplo, zona A, B y C), ejecuta una comprobación en cada una y reporta el resultado antes de continuar.
- **Guía interactiva**: recibe por diálogo un destino de interés, guía a la persona por una ruta predefinida y emite mensajes de estado durante el trayecto.

- **Misión reactiva por eventos**: ejecuta una misión principal y, si llega una alerta externa prioritaria, interrumpe temporalmente el flujo para atenderla y luego retoma la misión donde se dejó.

En todos los casos, el patrón de desarrollo es el mismo: preparar e integrar capacidades, y construir el BT completo de misión.

Paso 1: preparación del entorno e integración de capacidades

Este paso prepara el entorno de ejecución e integra las capacidades sobre las que se construirá la misión. Las capacidades concretas dependerán del ejercicio seleccionado y deben validarse antes de componer el BT completo.

1. Verificar que el entorno de trabajo (simulación o robot real) está operativo y que los recursos necesarios están disponibles.
2. Identificar las interfaces de las capacidades base (acciones, servicios, topics o APIs) y comprobar que son accesibles desde el sistema.
3. Ejecutar pruebas mínimas de validación de las capacidades base:
   - Definir al menos dos casos de prueba representativos.
   - Verificar que la capacidad responde de forma consistente en ambos casos.
   - Registrar resultados y posibles limitaciones observadas.
4. Definir y documentar los hitos de misión que se utilizarán en el BT:
   - Eventos de inicio y fin.
   - Objetivos o etapas intermedias.
   - Criterios básicos de éxito y fallo.
5. Seleccionar capacidades complementarias (por ejemplo, percepción, interacción, manipulación o navegación).
6. Definir el contrato de cada capacidad:
   - Entradas requeridas.
   - Salidas esperadas.
   - Eventos de progreso y finalización.
7. Implementar e integrar nodos hoja del BT para encapsular cada capacidad con interfaces homogéneas y reutilizables.
8. Validar de forma aislada cada nodo integrado, comprobando entradas, salidas y comportamiento ante errores.

Paso 2: diseño e implementación de la misión con Behavior Tree

En este paso se implementa la misión completa como un único Behavior Tree, integrando todas las capacidades validadas anteriormente.

1. Diseñar la estructura del BT de misión (secuencias, selectores, condiciones y acciones) en función de los hitos definidos.
2. Implementar el árbol conectando nodos de capacidad y lógica de control con una política explícita de ejecución.
3. Incorporar gestión de incidencias (reintentos, rutas alternativas o estados de recuperación, según el ejercicio).
4. Añadir observabilidad de ejecución (estado del árbol, nodo activo, eventos de resultado).
5. Validar la misión completa en escenarios nominales y en al menos un escenario de fallo.

5.3. Teoría y herramientas para el ejercicio

Asignación de comportamiento a arquitectura

En este ejercicio se implementa una arquitectura simplificada de dos niveles:

- **BT (misión)**: modela el flujo completo de la misión de alto nivel, orquestando las diferentes capacidades en una secuencia jerárquica (en este capítulo, una misión de servicio es solo uno de los ejercicios posibles).
- **Capacidades**: acciones atómicas accesibles por interfaz (navegar, hablar, tomar pedido, obtener pose).

El objetivo arquitectónico es mantener la separación clara: el BT define qué hacer y en qué orden, mientras que las capacidades definen cómo hacerlo. El BT no contiene lógica interna de navegación o diálogo; simplemente invoca estas capacidades a través de interfaces bien definidas.

Esta aproximación es apropiada cuando la misión es suficientemente simple y no requiere gestión de múltiples modos o estados persistentes que justifiquen coordinación adicional.

Behavior Trees: secuencias, selectores y nodos hoja

Cada tarea se implementa como un Behavior Tree. Se recomienda utilizar nodos de control típicos:

- **Sequence**: ejecutar pasos en orden hasta fallo.
- **Fallback/Selector**: probar alternativas (por ejemplo, recuperación).
- **Condition**: evaluar estado (por ejemplo, "hay presencia").
- **Action**: ejecutar una capacidad (navegar, hablar, esperar).

El énfasis está en diseñar nodos hoja reutilizables, que representen capacidades atómicas o acciones simuladas.

Nodos reutilizables: patrones mínimos

Se recomienda construir una pequeña biblioteca de nodos hoja reutilizables y agnósticos del dominio:

- `Report(message)`: publica un mensaje de estado u observabilidad.
- `ExecuteCapability(name, goal)`: invoca una capacidad externa mediante una interfaz unificada.
- `Wait(seconds)`: introduce una espera controlada en el flujo.
- `CheckCondition(condition)`: evalúa una precondición y devuelve éxito o fallo.
- `SimulateStep(name)`: encapsula una fase simulada de la misión.

En un ejercicio concreto, estos patrones se especializan con nodos de dominio. Por ejemplo, `Report` puede materializarse como `Say(text)` y `ExecuteCapability` como `NavigateTo(pose)` en un escenario con navegación e interacción.

Estos nodos se reutilizarán en múltiples pasos de la misión. Conviene enfatizar que la reutilización se consigue diseñando nodos con responsabilidades atómicas y parámetros explícitos.

5.4. Ejemplo de referencia: Bump-and-Go con Behavior Trees

Para ilustrar la implementación completa de un Behavior Tree, desarrollaremos el comportamiento clásico de *bump-and-go* con recuperación ante obstáculos. Este ejemplo muestra cómo estructurar un árbol que combina navegación con reactividad ante eventos del entorno, demostrando las ventajas arquitectónicas de los BT: expresión jerárquica de prioridades, recuperación automática ante fallos, y composición modular de comportamientos reutilizables.

El robot dispone de un sensor láser que publica medidas de distancia en el topic estándar ROS 2 `/scan` mediante mensajes de tipo `sensor_msgs/msg/LaserScan`. Este mensaje contiene un array `ranges` con las distancias medidas en múltiples direcciones angulares, típicamente cubriendo 360° con resolución de 1°. Los valores del array representan distancias en metros; valores infinitos o NaN indican ausencia de obstáculo en esa dirección.

El objetivo del comportamiento es que el robot avance en línea recta hacia una pose objetivo usando control directo (sin planificación de caminos). Si cualquiera de las medidas del láser indica un obstáculo a menos de un umbral (por defecto, 0.5 metros), el robot debe ejecutar inmediatamente una maniobra de recuperación: retroceder para alejarse y girar para cambiar la orientación, antes de reintentar el avance.

Diseño del árbol

La figura 5.1 muestra la estructura completa del comportamiento.

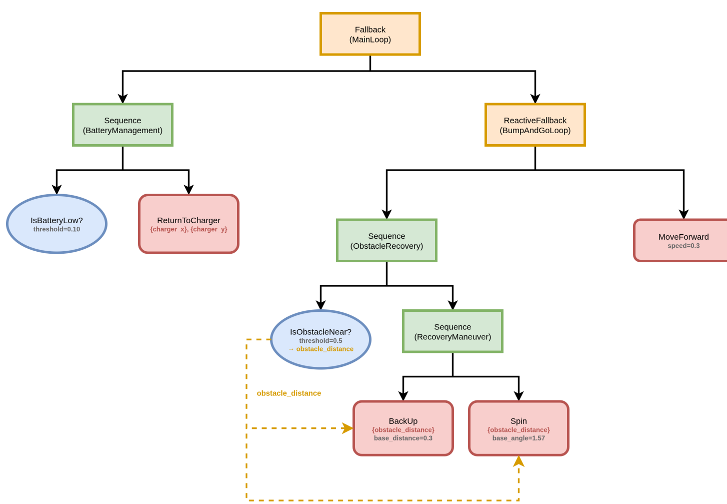


Figura 5.1: Behavior Tree para bump-and-go con control directo y recuperación ante obstáculos

El árbol se estructura en niveles jerárquicos que expresan prioridades y composición. La raíz es un `Fallback` que establece la política de control: intenta primero gestionar obstáculos cercanos (prioridad de seguridad), luego ejecutar la navegación (objetivo principal), y finalmente abortar si todo falla (último recurso). Este patrón expresa explícitamente que la seguridad tiene prioridad sobre el objetivo de navegación.

La gestión de obstáculos se implementa mediante un `Sequence` que verifica si hay un obstáculo cercano y, de ser así, ejecuta la recuperación. Solo retorna `SUCCESS` si detecta el obstáculo *y* la recuperación tiene éxito.

Si no hay obstáculo, retorna `FAILURE` inmediatamente por cortocircuito, permitiendo que el `Fallback` continúe con la navegación. La maniobra de recuperación consiste en la secuencia clásica de bump-and-go: retroceder una distancia fija seguido de un giro, implementada como `BackUp` seguida de `Spin`.

La navegación está protegida con un decorador `Retry(3)` para añadir robustez ante fallos transitorios. Antes de iniciar la navegación, un nodo `HasGoal` verifica que existe un objetivo definido. La propia acción de navegación está envuelta en un `Timeout(120s)` que garantiza que si se atasca, se aborta tras 2 minutos permitiendo reintentar.

Comunicación entre nodos mediante puertos y blackboard

El ejemplo ilustra tres patrones fundamentales de comunicación:

1. **Datos globales de misión**: El programa principal establece `goal_x=5.0`, `goal_y=3.0` y `goal_theta=0.0` en el blackboard global al inicio. El nodo `MoveTowardsGoal` declara puertos de entrada para estos valores y los lee en cada ejecución.
2. **Conexión directa nodo→nodo**: El nodo `IsObstacleNear` detecta la distancia mínima al obstáculo más cercano consultando el láser y escribe este valor en su puerto de salida `obstacle_distance`. Esta información fluye automáticamente hacia `BackUp` y `Spin` mediante sus puertos de entrada. `BackUp` calcula cuánto retroceder sumando 0.2m a la distancia del obstáculo (mínimo 0.3m). `Spin` decide el ángulo de giro: si el obstáculo está muy cerca (<0.3m) gira 180°, si no, solo 90°.
3. **Parámetros configurables con defaults**: Los nodos declaran parámetros como `threshold=0.5`, `base_distance=0.3`, `base_angle=1.57` con valores por defecto razonables, permitiendo ajustar el comportamiento desde el XML sin modificar código.

La elegancia de esta arquitectura radica en que cada nodo tiene una responsabilidad clara y el flujo de control surge naturalmente de la composición. La información fluye desde el sensor (`IsObstacleNear`) hacia las acciones de recuperación (`BackUp`, `Spin`), adaptándose dinámicamente a la situación detectada. Comparemos con una FSM equivalente: necesitaríamos estados para `IDLE`, `NAVIGATING`, `OBSTACLE_DETECTED`, `BACKING_UP`, `SPINNING`, `RETRY_1`, `RETRY_2`, `RETRY_3`, `ABORTING`, transiciones complejas entre todos ellos, *y* variables globales para compartir la distancia del obstáculo. El BT expresa el mismo comportamiento de forma mucho más compacta y legible.

Organización del paquete de referencia

El código completo de este ejemplo se encuentra disponible en el paquete `bt_examples`. Este paquete contiene:

- Definición XML del árbol en `config/bumpandgo_tree.xml`
- Headers de nodos personalizados en `include/bt_examples/`
- Implementaciones (.cpp) en `src/`
- Programa principal en `src/bumpandgo_bt_example.cpp`
- Launch file configurado en `launch/bumpandgo_bt_example.launch.py`
- README con instrucciones de uso y explicación pedagógica
- Archivo `thirdparty.repos` para gestionar dependencias externas

Los detalles de implementación completos se encuentran en las subsecciones siguientes, donde se muestra paso a paso cómo construir este sistema usando BehaviorTree.CPP. El código en el paquete `bt_examples` coincide exactamente con los fragmentos presentados en esta sección.

La separación entre definición estructural (XML) e implementación (C++) permite iterar rápidamente en el diseño del comportamiento sin recompilar código. Es posible modificar la estructura del árbol, añadir decoradores, o reordenar ramas editando únicamente el archivo XML. Solo cuando se necesita un nuevo tipo de nodo se requiere implementación en C++.

Implementación: Definición XML del árbol

Esta sección proporciona ejemplos completos de implementación de nodos BT usando BehaviorTree.CPP, la biblioteca estándar para ROS 2. Se presentan tres componentes fundamentales: definición XML del árbol, implementación de nodos personalizados, y programa principal.

Archivo XML del árbol bump-and-go

El árbol bump-and-go implementa el comportamiento clásico de exploración reactiva: el robot avanza continuamente hasta detectar un obstáculo, momento en el cual retrocede y gira para evitarlo, repitiendo este patrón indefinidamente. Adicionalmente, se integra gestión autónoma de batería: cuando el nivel de carga alcanza un umbral crítico (configurable, por defecto 10 %), el robot interrumpe la exploración y navega autónomamente hacia la base de recarga.

Esta arquitectura ilustra dos conceptos clave: **bucles continuos con ReactiveFallback** (el robot explora indefinidamente mientras la batería esté suficiente) y **interrupciones basadas en prioridad** (la batería baja tiene máxima prioridad, interrumpiendo cualquier actividad en curso). A diferencia de sistemas con navegación a objetivos específicos, este patrón bump-and-go puro enfatiza la reactividad directa a estímulos sensoriales.

```xml
<?xml version="1.0"?>
<root main_tree_to_execute="BumpAndGo">
  <BehaviorTree ID="BumpAndGo">
    <!-- Raíz: Fallback para priorizar batería baja -->
    <Fallback name="MainLoop">

      <!-- Rama 1: Si batería baja, ir a cargar -->
      <Sequence name="BatteryManagement">
        <IsBatteryLow threshold="0.10" />
        <ReturnToCharger
          charger_x="{charger_x}"
          charger_y="{charger_y}" />
      </Sequence>

      <!-- Rama 2: Bucle continuo de bump-and-go mientras batería OK -->
      <ReactiveFallback name="BumpAndGoLoop">

        <!-- Si detecta obstáculo, maniobra de recuperación -->
        <Sequence name="ObstacleRecovery">
          <IsObstacleNear
            threshold="0.5"
            obstacle_distance="{obstacle_dist}" />
          <Sequence name="RecoveryManeuver">
            <BackUp
              obstacle_distance="{obstacle_dist}"
```

```
            base_distance="0.3" />
          <Spin
            obstacle_distance="{obstacle_dist}"
            base_angle="1.57" />
        </Sequence>
      </Sequence>

      <!-- Si no hay obstáculo, simplemente avanzar -->
      <MoveForward speed="0.3" />

    </ReactiveFallback>

  </Fallback>
 </BehaviorTree>
</root>
```

Fragmento C++ 5.1: Definición XML del árbol bump-and-go con gestión de batería

Flujo de ejecución del árbol

El árbol implementa un bucle continuo con interrupciones basadas en prioridad:

1. **Comprobación de batería (máxima prioridad)**: El `Fallback` raíz evalúa primero si `IsBatteryLow` detecta batería crítica. Si es así, interrumpe cualquier actividad y ejecuta `ReturnToCharger`, navegando a la base de recarga especificada en el blackboard (`charger_x`, `charger_y`).
2. **Bucle de exploración**: Si la batería está suficiente, el `ReactiveFallback` interno crea un bucle continuo que nunca completa exitosamente, forzando reevaluación constante. Este patrón es fundamental para comportamientos que deben ejecutarse indefinidamente.
3. **Reacción a obstáculos**: Dentro del bucle, si `IsObstacleNear` detecta un obstáculo cercano (threshold 0.5m), la `Sequence` de recuperación ejecuta `BackUp` y `Spin`, retrocediendo y girando. La distancia del obstáculo fluye mediante el blackboard (`obstacle_dist`), permitiendo adaptar la maniobra.
4. **Avance continuo**: Si no hay obstáculo, `MoveForward` publica comandos de velocidad constante para avanzar. Al retornar SUCCESS, el `ReactiveFallback` fuerza un nuevo tick inmediato, creando el bucle continuo de exploración. Esta es la esencia del comportamiento bump-and-go: avanzar siempre, interrumpirse solo ante obstáculos o batería baja.

La clave del diseño es que `IsBatteryLow` se reevalúa en cada tick del árbol, permitiendo detectar batería crítica inmediatamente incluso durante avance o maniobras de recuperación. El uso de `ReactiveFallback` en lugar de `Fallback` estándar garantiza que todos los hijos se reevalúan constantemente, necesario para detectar cambios en sensores (obstáculos, batería) de forma reactiva.

Observe cómo los valores del blackboard (`charger_x/y`, `obstacle_dist`) establecen comunicación entre nodos y con el programa principal, que inicializa la posición del cargador desde parámetros ROS. A diferencia de sistemas con objetivos de navegación, este diseño no requiere conocer destinos específicos, solo reacciona al entorno.

Implementación: Nodos personalizados

A continuación se muestran las implementaciones de los nodos clave del ejemplo, ilustrando el uso de puertos de entrada y salida. Cada nodo se implementa con separación entre declaración (archivo .hpp) e implementación (archivo .cpp) para favorecer la legibilidad y el reuso.

Nodo IsObstacleNear: condición con puerto de salida

```cpp
#include "bt_examples/is_obstacle_near_condition.hpp"
#include <limits>

IsObstacleNearCondition::IsObstacleNearCondition(
    const std::string& name,
    const BT::NodeConfiguration& config)
  : BT::ConditionNode(name, config) {
  if (!config.blackboard->get("node", node_)) {
    throw BT::RuntimeError("Missing required node in blackboard");
  }
  laser_sub_ = node_->create_subscription<sensor_msgs::msg::LaserScan>(
    "scan", 10,
    [this](const sensor_msgs::msg::LaserScan::SharedPtr msg) {
      last_scan_ = msg;
    });
}

BT::PortsList IsObstacleNearCondition::providedPorts() {
  return {
    BT::InputPort<double>("threshold", 0.5, "Distance threshold"),
    BT::OutputPort<double>("obstacle_distance", "Min distance to obstacle"
     )
  };
}

BT::NodeStatus IsObstacleNearCondition::tick() {
  double threshold;
  getInput("threshold", threshold);

  if (!last_scan_) {
    return BT::NodeStatus::FAILURE;
  }

  float min_distance = std::numeric_limits<float>::max();
  for (const auto& range : last_scan_->ranges) {
    if (std::isfinite(range) && range < min_distance) {
      min_distance = range;
    }
  }

  // Escribir al blackboard mediante puerto de salida
  setOutput("obstacle_distance", static_cast<double>(min_distance));

  return (min_distance < threshold) ? BT::NodeStatus::SUCCESS
                                    : BT::NodeStatus::FAILURE;
}
```

Fragmento C++ 5.2: Implementación del nodo de detección de obstáculos

Este nodo implementa una condición que detecta obstáculos cercanos. La lógica clave:

- **Constructor**: Obtiene el nodo de ROS 2 del blackboard con `blackboard->get("node", node_)`. Si no está disponible, lanza una excepción.
- **Suscripción continua**: El constructor crea una suscripción al topic `/scan` que almacena el último mensaje láser recibido mediante una lambda.
- **Puerto de entrada**: Lee el umbral de distancia desde el blackboard (valor por defecto: 0.5 metros).

- **Puerto de salida**: Calcula la distancia mínima del láser y la escribe al blackboard mediante `setOutput()`, permitiendo que otros nodos la usen.
- **Retorno**: SUCCESS si hay un obstáculo dentro del umbral, FAILURE en caso contrario.

Nodo BackUp: acción con puerto de entrada

```cpp
#include "bt_examples/backup_action.hpp"
#include <algorithm>

BackUpAction::BackUpAction(const std::string& name,
                           const BT::NodeConfiguration& config)
  : BT::StatefulActionNode(name, config),
    duration_(0, 0) {
  if (!config.blackboard->get("node", node_)) {
    throw BT::RuntimeError("Missing required node in blackboard");
  }
  cmd_vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>(
    "cmd_vel", 10);
}

BT::PortsList BackUpAction::providedPorts() {
  return {
    BT::InputPort<double>("obstacle_distance", "Distance to obstacle"),
    BT::InputPort<double>("base_distance", 0.3, "Base backup distance")
  };
}

BT::NodeStatus BackUpAction::onStart() {
  double obstacle_dist, base_dist;
  getInput("obstacle_distance", obstacle_dist);
  getInput("base_distance", base_dist);

  // Margen de seguridad: retrocede hasta el obstáculo + 20cm extra
  distance_ = std::max(base_dist, obstacle_dist + 0.2);

  start_time_ = node_->now();
  duration_ = rclcpp::Duration::from_seconds(distance_ / 0.2);

  RCLCPP_INFO(node_->get_logger(),
              "Backing up %.2f meters (obstacle at %.2fm)",
              distance_, obstacle_dist);
  return BT::NodeStatus::RUNNING;
}

BT::NodeStatus BackUpAction::onRunning() {
  auto elapsed = node_->now() - start_time_;

  if (elapsed < duration_) {
    geometry_msgs::msg::Twist cmd;
    cmd.linear.x = -0.2;
    cmd_vel_pub_->publish(cmd);
    return BT::NodeStatus::RUNNING;
  }

  geometry_msgs::msg::Twist cmd;
  cmd.linear.x = 0.0;
  cmd_vel_pub_->publish(cmd);

  RCLCPP_INFO(node_->get_logger(), "Back up complete");
  return BT::NodeStatus::SUCCESS;
}

void BackUpAction::onHalted() {
  geometry_msgs::msg::Twist cmd;
  cmd.linear.x = 0.0;
  cmd_vel_pub_->publish(cmd);
}
```

Fragmento C++ 5.3: Implementación del nodo de acción para retroceder

Este nodo retrocede una distancia calculada dinámicamente. La lógica clave:

- **Patrón StatefulActionNode**: Divide la ejecución en `onStart()` (inicialización) y `onRunning()` (bucle de ejecución).
- **Constructor**: Obtiene el nodo de ROS 2 del blackboard con `blackboard->get("node", node_)`. Si no está disponible, lanza una excepción.
- **Puerto de entrada**: Lee `obstacle_distance` del blackboard (escrito por IsObstacleNear) y `base_distance` (configurado en XML).
- **Cálculo dinámico**: La distancia de retroceso es el máximo entre la distancia base y la distancia al obstáculo + 0.2m, asegurando alejarse suficientemente.
- **Control temporal**: Calcula la duración basándose en velocidad constante (-0.2 m/s) y publica comandos `cmd_vel` hasta completar el tiempo.
- **onHalted()**: Detiene el robot si el nodo es interrumpido externamente.

Nodo Spin: acción con lógica condicional basada en puerto

```
#include "bt_examples/spin_action.hpp"

SpinAction::SpinAction(const std::string& name,
                       const BT::NodeConfiguration& config,
                       rclcpp::Node::SharedPtr node)
  : BT::StatefulActionNode(name, config), node_(node),
    duration_(0, 0) {
  cmd_vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>(
    "cmd_vel", 10);
}

BT::PortsList SpinAction::providedPorts() {
  return {
    BT::InputPort<double>("obstacle_distance", "Distance to obstacle"),
    BT::InputPort<double>("base_angle", 1.57, "Base spin angle (radians)")
  };
}

BT::NodeStatus SpinAction::onStart() {
  double obstacle_dist, base_angle;
  getInput("obstacle_distance", obstacle_dist);
  getInput("base_angle", base_angle);

  if (obstacle_dist < 0.3) {
    angle_ = 3.14159;
    RCLCPP_WARN(node_->get_logger(),
                "Obstacle very close (%.2fm), spinning 180 degrees",
                obstacle_dist);
  } else {
    angle_ = base_angle;
    RCLCPP_INFO(node_->get_logger(),
                "Obstacle at %.2fm, spinning %.0f degrees",
                obstacle_dist, angle_ * 180.0 / 3.14159);
  }

  start_time_ = node_->now();
  duration_ = rclcpp::Duration::from_seconds(angle_ / 0.5);

  return BT::NodeStatus::RUNNING;
}

BT::NodeStatus SpinAction::onRunning() {
  auto elapsed = node_->now() - start_time_;
```

```
  if (elapsed < duration_) {
    geometry_msgs::msg::Twist cmd;
    cmd.angular.z = 0.5;
    cmd_vel_pub_->publish(cmd);
    return BT::NodeStatus::RUNNING;
  }

  geometry_msgs::msg::Twist cmd;
  cmd.angular.z = 0.0;
  cmd_vel_pub_->publish(cmd);

  RCLCPP_INFO(node_->get_logger(), "Spin complete");
  return BT::NodeStatus::SUCCESS;
}

void SpinAction::onHalted() {
  geometry_msgs::msg::Twist cmd;
  cmd.angular.z = 0.0;
    cmd_vel_pub_->publish(cmd);
  }

private:
  rclcpp::Node::SharedPtr node_;
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_vel_pub_;
  double angle_;
  rclcpp::Time start_time_;
  rclcpp::Duration duration_;
};
```

Fragmento C++ 5.4: Implementación del nodo de acción para girar

Este nodo gira el robot con lógica adaptativa. La lógica clave:

- **Decisión contextual**: Si el obstáculo está muy cerca (<0.3m), gira 180°; de lo contrario, usa el ángulo base (90° por defecto).
- **Puerto de entrada**: Lee `obstacle_distance` del blackboard para tomar la decisión.
- **Control temporal**: Convierte el ángulo deseado a duración usando velocidad angular constante (0.5 rad/s).
- **Logs informativos**: Registra la distancia del obstáculo y el ángulo de giro calculado para facilitar depuración.

Nodo MoveForward: acción de avance continuo

```
#include "bt_examples/move_forward_action.hpp"

MoveForwardAction::MoveForwardAction(const std::string& name,
                                     const BT::NodeConfiguration& config,
                                     rclcpp::Node::SharedPtr node)
  : BT::SyncActionNode(name, config), node_(node) {
  cmd_vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>(
    "cmd_vel", 10);
}

BT::PortsList MoveForwardAction::providedPorts() {
  return {
    BT::InputPort<double>("speed", 0.3, "Forward speed (m/s)")
  };
}

BT::NodeStatus MoveForwardAction::tick() {
  double speed;
  getInput("speed", speed);

  geometry_msgs::msg::Twist cmd;
  cmd.linear.x = speed;
  cmd.angular.z = 0.0;
  cmd_vel_pub_->publish(cmd);
```

```
25
26    return BT::NodeStatus::SUCCESS;
27  }
```

Este nodo implementa el avance continuo del robot. Al retornar SUCCESS inmediatamente, permite que el `ReactiveFallback` lo reevalúe constantemente, creando un bucle de avance continuo hasta que se detecte un obstáculo o batería baja.

Fragmento C++ 5.5: Implementación del nodo de avance continuo

Nodo IsBatteryLow: condición de batería crítica

```
1  #include "bt_examples/is_battery_low_condition.hpp"
2
3  IsBatteryLowCondition::IsBatteryLowCondition(
4      const std::string& name,
5      const BT::NodeConfiguration& config,
6      rclcpp::Node::SharedPtr node)
7    : BT::ConditionNode(name, config), node_(node) {
8    battery_sub_ = node_->create_subscription<sensor_msgs::msg::BatteryState
       >(
9      "battery_state", 10,
10     [this](const sensor_msgs::msg::BatteryState::SharedPtr msg) {
11       last_battery_ = msg;
12     });
13 }
14
15 BT::PortsList IsBatteryLowCondition::providedPorts() {
16   return {
17     BT::InputPort<double>("threshold", 0.10,
18                           "Battery percentage threshold (0.0-1.0)")
19   };
20 }
21
22 BT::NodeStatus IsBatteryLowCondition::tick() {
23   double threshold;
24   getInput("threshold", threshold);
25
26   if (!last_battery_) {
27     // Si no hay datos de batería, asumimos que está bien
28     return BT::NodeStatus::FAILURE;
29   }
30
31   // BatteryState.percentage va de 0.0 a 1.0 (0% a 100%)
32   bool is_low = last_battery_->percentage < threshold;
33
34   if (is_low) {
35     RCLCPP_WARN(node_->get_logger(),
36                 "Battery low: %.1f%% (threshold: %.1f%%)",
37                 last_battery_->percentage * 100.0,
38                 threshold * 100.0);
39     return BT::NodeStatus::SUCCESS;
40   }
41
42   return BT::NodeStatus::FAILURE;
43 }
```

Este nodo monitoriza continuamente el topic `/battery_state` y retorna SUCCESS cuando la batería cae por debajo del umbral especificado, permitiendo que el árbol interrumpa cualquier actividad y ejecute la secuencia de retorno al cargador.

Fragmento C++ 5.6: Implementación del nodo de detección de batería baja

Nodo ReturnToCharger: integración con Nav2

Los nodos que requieren navegación utilizan la clase `NavigationClient` del paquete `nav2_example`, que encapsula la comunicación con la acción `NavigateToPose` de Nav2.

```cpp
#include <nav2_example/navigation_client.hpp>

ReturnToChargerAction::ReturnToChargerAction(
  const std::string& name,
  const BT::NodeConfiguration& config,
  rclcpp::Node::SharedPtr node)
: BT::StatefulActionNode(name, config), node_(node)
{
  nav_client_ = std::make_shared<NavigationClient>();
}

BT::NodeStatus ReturnToChargerAction::onRunning() {
  if (!nav_client_->wait_for_action_server(std::chrono::seconds(1))) {
    return BT::NodeStatus::RUNNING;
  }

  if (!goal_sent_) {
    auto target_pose = nav_client_->create_pose_stamped(
      charger_x_, charger_y_, 0.0);
    nav_client_->send_goal(target_pose);
    goal_sent_ = true;
    return BT::NodeStatus::RUNNING;
  }

  if (nav_client_->is_goal_done()) {
    return nav_client_->was_goal_successful()
      ? BT::NodeStatus::SUCCESS
      : BT::NodeStatus::FAILURE;
  }

  return BT::NodeStatus::RUNNING;
}

void ReturnToChargerAction::onHalted() {
  if (goal_sent_ && nav_client_->is_goal_active()) {
    nav_client_->cancel_goal();
  }
}
```

Fragmento C++ 5.7: Nodo BT que usa NavigationClient para navegar al cargador

La clase `NavigationClient` proporciona métodos convenientes (`send_goal`, `is_goal_done`, `was_goal_successful`, `cancel_goal`) que abstraen los detalles de la interacción con la acción de Nav2.

Este patrón establece una separación arquitectónica clara: el nodo BT decide *cuándo* invocar Nav2 y *qué hacer* con el resultado, mientras que `NavigationClient` encapsula *cómo* comunicarse con el servidor de acción, y Nav2 decide *cómo* ejecutar la navegación.

La lógica del nodo:

- **Patrón asíncrono**: Primero espera al servidor de acción, luego envía el objetivo (solo una vez marcando `goal_sent_`), finalmente monitoriza que se haya completado.
- **Encapsulación con NavigationClient**: Simplifica el código del nodo BT delegando complejidad de acciones ROS 2 a una clase auxiliar.
- **Cancelación limpia**: `onHalted()` cancela el objetivo activo si el nodo es interrumpido (ej: nueva detección de batería baja).
- **Coordenadas fijas**: Las coordenadas del cargador (`charger_x_`, `charger_y_`) deberían leerse del blackboard o puertos para mayor flexibilidad.

Programa principal del ejemplo bump-and-go

```
#include <rclcpp/rclcpp.hpp>
#include <behaviortree_cpp/bt_factory.h>
#include <behaviortree_cpp/loggers/bt_cout_logger.h>
#include <behaviortree_cpp/decorators/retry_node.h>
#include <behaviortree_cpp/decorators/timeout_node.h>
#include <ament_index_cpp/get_package_share_directory.hpp>

#include "bt_examples/bt_node_registration.hpp"

int main(int argc, char** argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<rclcpp::Node>("bumpandgo_bt");

  // Factory para registrar nodos personalizados
  BT::BehaviorTreeFactory factory;

  // Registrar nodos personalizados
  register_bt_nodes(factory);

  // Leer parámetros ROS
  node->declare_parameter("charger_x", 0.0);
  node->declare_parameter("charger_y", 0.0);

  double charger_x = node->get_parameter("charger_x").as_double();
  double charger_y = node->get_parameter("charger_y").as_double();

  // Crear blackboard y poner recursos ANTES de crear el árbol
  auto blackboard = BT::Blackboard::create();
  blackboard->set("node", node);
  blackboard->set("charger_x", charger_x);
  blackboard->set("charger_y", charger_y);

  // Obtener path al archivo XML
  std::string package_share_dir =
    ament_index_cpp::get_package_share_directory("bt_examples");
  std::string tree_path = package_share_dir + "/config/bumpandgo_tree.xml"
     ;

  // Cargar árbol desde XML con la blackboard que contiene los recursos
  auto tree = factory.createTreeFromFile(tree_path, blackboard);

  // Logger para depuración
  BT::StdCoutLogger logger(tree);

  RCLCPP_INFO(node->get_logger(), "Bump-and-go behavior tree started");
  RCLCPP_INFO(node->get_logger(), "Charger position: (%.2f, %.2f)",
              charger_x, charger_y);

  // Bucle principal: tick del árbol a 10 Hz
  rclcpp::Rate rate(10);
  BT::NodeStatus status = BT::NodeStatus::IDLE;

  while (rclcpp::ok() && status != BT::NodeStatus::FAILURE) {
    RCLCPP_DEBUG(node->get_logger(), "Ticking the behavior tree...");

    // Procesar callbacks de ROS
    rclcpp::spin_some(node);

    // Evaluar árbol
    status = tree.tickOnce();
    rate.sleep();

  }

  if (status == BT::NodeStatus::SUCCESS) {
    RCLCPP_INFO(node->get_logger(), "Mission completed successfully");
  } else {
    RCLCPP_ERROR(node->get_logger(), "Mission failed");
  }
```

```
69
70   rclcpp::shutdown();
71   return 0;
72 }
```

Fragmento C++ 5.8: Programa principal del sistema bump-and-go

El programa principal sigue el patrón estándar de aplicaciones BehaviorTree.CPP en ROS 2. Aspectos clave:

- El **BehaviorTreeFactory** actúa como registro de tipos. Cada nodo personalizado se registra mediante `register_bt_nodes(factory)`, una función centralizada que usa `registerNodeType` para asociar cada clase de nodo con su nombre en XML.
- La **blackboard se crea antes del árbol** usando `BT::Blackboard::create()` y se inicializa con los recursos compartidos: el nodo de ROS (`node`) y los parámetros de configuración (`charger_x`, `charger_y`).
- Los parámetros ROS (`charger_x`, `charger_y`) se declaran con valores por defecto y se leen con `get_parameter()`, permitiendo configuración externa mediante el launch file.
- **Puertos (input/output)**: Los nodos se comunican mediante blackboard usando puertos tipados. `IsObstacleNear` *produce* la distancia del obstáculo (`setOutput`), mientras que `BackUp` y `Spin` la *consumen* (`getInput`). Los puertos se declaran en `providedPorts()` con tipos, nombres, y valores por defecto.
- **Integración con ROS 2**: Cada nodo obtiene el `shared_ptr` al nodo principal de ROS 2 desde el blackboard, permitiendo subscripciones, publicaciones y acceso a capacidades. Por ejemplo, `IsObstacleNear` subscribe a `scan`, `MoveForward /BackUp/Spin` publican en `cmd_vel`, y `ReturnToCharger` utiliza `NavigationClient` para comunicarse con Nav2 mediante la acción `NavigateToPose`.
- La carga del árbol mediante `createTreeFromFile(tree_path, blackboard)` parsea el XML, instancia los nodos según el registro, y les proporciona acceso al blackboard con los recursos. Se usa `ament_index_cpp` para localizar el archivo XML en la instalación del paquete.
- El bucle principal ejecuta `tickOnce()` a 10 Hz, alternando con `rclcpp::spin_some()` para procesar callbacks ROS. Este patrón híbrido permite al árbol reaccionar a cambios sensoriales (LaserScan, BatteryState) mientras mantiene control del flujo de ejecución.

Resumen de conceptos clave

Este ejemplo bump-and-go ilustra los siguientes conceptos arquitectónicos y de implementación:

- **Bump-and-go puro**: Comportamiento reactivo clásico sin objetivos específicos. El robot avanza continuamente y reacciona directamente a estímulos sensoriales (obstáculos, batería), ilustrando la arquitectura reactiva en su forma más simple.
- **Bucles continuos con ReactiveFallback**: Un `ReactiveFallback` que nunca retorna SUCCESS crea un bucle infinito de exploración, reevaluando constantemente condiciones reactivas. Es el patrón fundamental para comportamientos que deben ejecutarse indefinidamente.
- **Interrupciones basadas en prioridad**: La batería baja interrumpe cualquier actividad en curso, demostrando cómo gestionar recursos críticos mediante la estructura del árbol.

- **Gestión de recursos críticos**: La monitorización de batería y navegación autónoma al cargador ilustra cómo los BT pueden gestionar recursos sin intervención externa, mejorando la autonomía del sistema.
- **Comunicación entre nodos mediante blackboard**: El flujo de datos entre nodos (`obstacle_dist`, `charger_x/y`) se implementa de forma explícita y tipada, evitando acoplamiento directo.
- **Separación entre estructura (XML) e implementación (C++)**: Permite iterar rápidamente en el diseño del comportamiento sin recompilar, facilitando experimentación y ajuste.
- **Nodos reutilizables**: `IsObstacleNear`, `BackUp`, `Spin`, `MoveForward`, `IsBatteryLow`, `ReturnToCharger` son genéricos y reutilizables en otros contextos (patrullaje, limpieza, inspección). Los nodos que requieren capacidades complejas de navegación utilizan la clase `NavigationClient` que encapsula la comunicación con Nav2.
- **Acciones stateful**: `StatefulActionNode` permite implementar acciones complejas con ciclo de vida (`onStart`, `onRunning`, `onHalted`), necesario para navegación, manipulación, y otras capacidades de larga duración.
- **Integración natural con ROS 2**: La arquitectura de nodos BT que contienen subscripciones/publicaciones ROS proporciona una forma elegante de conectar percepción y acción con lógica de comportamiento jerárquica.
- **Escalabilidad arquitectónica**: Los BT permiten escalar a sistemas complejos manteniendo modularidad. Para misiones simples, el BT puede invocar directamente capacidades; para sistemas más complejos, se pueden combinar con FSM u otros modelos de coordinación según las necesidades del problema.

5.5. Ejemplo de capacidad de interacción

Los nodos presentados en los ejemplos anteriores (IsObstacleNear, BackUp, Spin, MoveForward, IsBatteryLow, ReturnToCharger) pueden utilizarse como referencia total o parcial para implementar los nodos BT necesarios en distintos ejercicios basadas en interacción. En este apartado se muestra un ejemplo para concretar la integración de capacidades de interacción (diálogo y extracción de información).

Nodo SayTextClient: text-to-speech con HRIClient

Este nodo utiliza la clase `HRIClient` para síntesis y reproducción de voz. HRIClient abstrae la comunicación con los servicios TTS, proporcionando una interfaz asíncrona simple.

```
1  #include "bt_examples/say_text_client_action.hpp"
2  #include "bt_examples/text_utils.hpp"
3
4  SayTextClientAction::SayTextClientAction(
5    const std::string & name,
6    const BT::NodeConfig & config)
7  : BT::StatefulActionNode(name, config)
8  {
9    if (!config.blackboard->get("hri_client", hri_client_)) {
10     throw BT::RuntimeError("Missing required hri_client in blackboard");
11   }
12 }
13
14 std::string SayTextClientAction::formatText(const std::string & text)
```

```
{
  return bt_examples::formatText(text, config().blackboard);
}

BT::NodeStatus SayTextClientAction::onStart()
{
  std::string text;
  if (!getInput("text", text)) {
    RCLCPP_ERROR(hri_client_->get_logger(),
                 "SayTextClientAction: 'text' input port is required");
    return BT::NodeStatus::FAILURE;
  }

  text = formatText(text);  // Expandir variables del blackboard

  RCLCPP_INFO(hri_client_->get_logger(), "Saying: '%s'", text.c_str());

  hri_client_->start_speaking(text);
  start_time_ = std::chrono::steady_clock::now();

  return BT::NodeStatus::RUNNING;
}

BT::NodeStatus SayTextClientAction::onRunning()
{
  auto elapsed = std::chrono::steady_clock::now() - start_time_;
  if (elapsed > std::chrono::seconds(30)) {
    RCLCPP_ERROR(hri_client_->get_logger(), "TTS operation timeout");
    return BT::NodeStatus::FAILURE;
  }

  if (hri_client_->is_speaking_done()) {
    if (hri_client_->get_speaking_result()) {
      RCLCPP_INFO(hri_client_->get_logger(), "TTS completed successfully")
     ;
      return BT::NodeStatus::SUCCESS;
    } else {
      RCLCPP_ERROR(hri_client_->get_logger(), "TTS operation failed");
      return BT::NodeStatus::FAILURE;
    }
  }

  return BT::NodeStatus::RUNNING;
}

void SayTextClientAction::onHalted()
{
  RCLCPP_WARN(hri_client_->get_logger(), "TTS action halted");
}
```

Fragmento C++ 5.9: Implementación del nodo SayTextClient (say_text_client_action.cpp)

La lógica del nodo:

- **Constructor**: Obtiene `HRIClient` del blackboard con `blackboard->get("hri_client", hri_client_)`. Si no está disponible, lanza una excepción.
- **Formateo de texto**: La función `formatText()` expande variables del blackboard (ej: "Hola {name}"se convierte en "Hola Juan"si `name="Juan"` en el blackboard).
- **Patrón asíncrono**: `onStart()` inicia la operación TTS con `start_speaking()`, `onRunning()` monitoriza el progreso sin bloquear.
- **Timeout defensivo**: Implementa un timeout de 30 segundos para evitar bloqueos si el servicio TTS no responde.
- **Verificación de resultado**: Comprueba tanto que la operación terminó (`is_speaking_done()`) como que fue exitosa (`get_speaking_result()`).
- **HRIClient abstrae complejidad**: El nodo no gestiona clientes de servicio ROS directamente, HRIClient encapsula esa lógica.

Nodo ListenTextClient: speech-to-text con HRIClient

Este nodo utiliza la clase `HRIClient` para capturar voz del usuario y convertirla a texto. HRIClient abstrae la comunicación con los servicios de reconocimiento de voz.

```cpp
#include "bt_examples/listen_text_client_action.hpp"

ListenTextClientAction::ListenTextClientAction(
  const std::string & name,
  const BT::NodeConfig & config)
: BT::StatefulActionNode(name, config)
{
  if (!config.blackboard->get("hri_client", hri_client_)) {
    throw BT::RuntimeError("Missing required hri_client in blackboard");
  }
}

BT::NodeStatus ListenTextClientAction::onStart()
{
  RCLCPP_INFO(hri_client_->get_logger(), "Listening for speech...");

  hri_client_->start_listen();
  start_time_ = std::chrono::steady_clock::now();

  return BT::NodeStatus::RUNNING;
}

BT::NodeStatus ListenTextClientAction::onRunning()
{
  auto elapsed = std::chrono::steady_clock::now() - start_time_;
  if (elapsed > std::chrono::seconds(60)) {
    RCLCPP_ERROR(hri_client_->get_logger(), "STT operation timeout");
    return BT::NodeStatus::FAILURE;
  }

  if (hri_client_->is_listen_done()) {
    std::string recognized_text = hri_client_->get_listened_text();

    if (recognized_text.empty()) {
      RCLCPP_ERROR(hri_client_->get_logger(), "STT returned empty text");
      return BT::NodeStatus::FAILURE;
    }

    RCLCPP_INFO(hri_client_->get_logger(), "Recognized: '%s'",
     recognized_text.c_str());
    setOutput("recognized_text", recognized_text);

    return BT::NodeStatus::SUCCESS;
  }

  return BT::NodeStatus::RUNNING;
}

void ListenTextClientAction::onHalted()
{
  RCLCPP_WARN(hri_client_->get_logger(), "STT action halted");
}
```

Fragmento C++ 5.10: Implementación del nodo ListenTextClient (listen_text_client_action.cpp)

La lógica del nodo:

- **Constructor**: Obtiene `HRIClient` del blackboard con `blackboard->get("hri_client", hri_client_)`. Si no está disponible, lanza una excepción.
- **Patrón asíncrono**: `onStart()` inicia la captura de voz con `start_listen()`, `onRunning()` espera sin bloquear.
- **Puerto de salida**: Escribe el texto reconocido al blackboard mediante `setOutput()` para que otros nodos lo procesen.

- **Timeout largo**: 60 segundos, ya que el usuario puede tardar en hablar (a diferencia de TTS que es inmediato).
- **Validación de resultado**: Retorna FAILURE si el texto reconocido está vacío (ej: usuario no habló, ruido ambiental).
- **Integración en flujo**: Este nodo típicamente va seguido de ExtractInfoClient para interpretar el texto capturado.

Nodo ExtractInfoClient: extracción de información con LLM

Este nodo utiliza HRIClient para extraer información específica de un texto mediante un modelo de lenguaje (LLM). Útil para interpretar órdenes del usuario y extraer entidades relevantes (bebida, comida, ubicación, etc.).

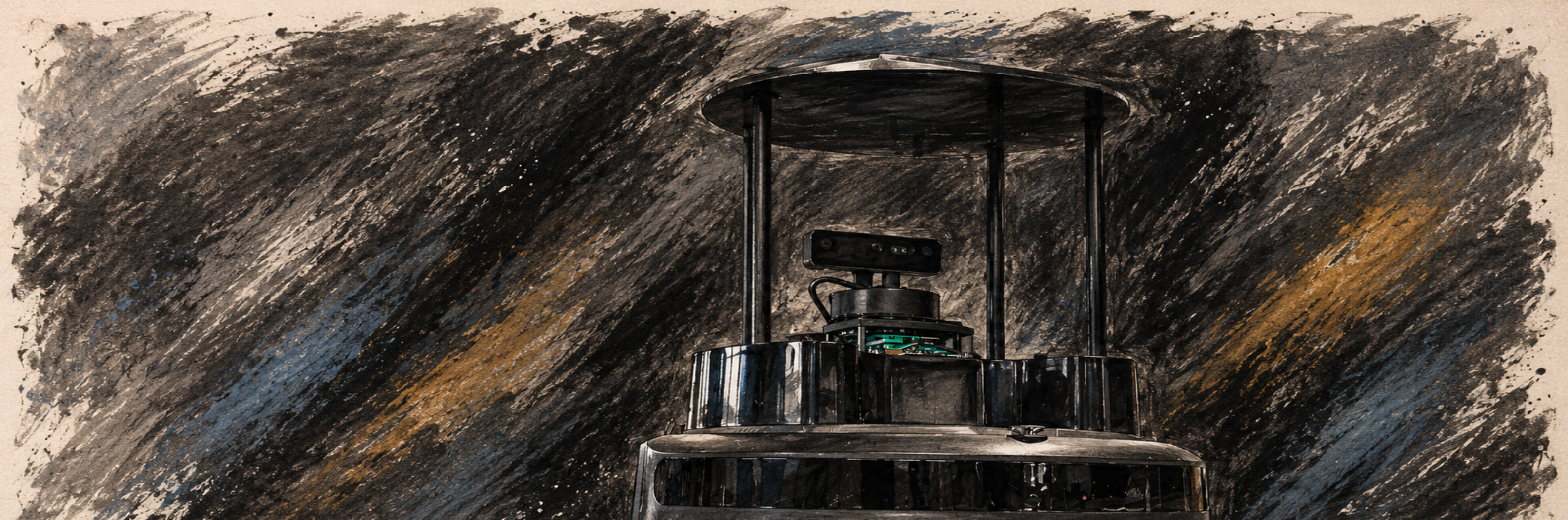
```
#include "bt_examples/extract_info_client_action.hpp"

ExtractInfoClientAction::ExtractInfoClientAction(
  const std::string & name,
  const BT::NodeConfig & config)
: BT::StatefulActionNode(name, config)
{
  if (!config.blackboard->get("hri_client", hri_client_)) {
    throw BT::RuntimeError("Missing required hri_client in blackboard");
  }
}

BT::NodeStatus ExtractInfoClientAction::onStart()
{
  std::string interest;
  if (!getInput("interest", interest)) {
    RCLCPP_ERROR(hri_client_->get_logger(),
                 "ExtractInfoClientAction: 'interest' input port is
     required");
    return BT::NodeStatus::FAILURE;
  }

  std::string full_text;
  if (!getInput("full_text", full_text)) {
    RCLCPP_ERROR(hri_client_->get_logger(),
                 "ExtractInfoClientAction: 'full_text' input port is
     required");
    return BT::NodeStatus::FAILURE;
  }

  RCLCPP_INFO(hri_client_->get_logger(), "Extracting '%s' from: '%s'",
              interest.c_str(), full_text.c_str());

  hri_client_->start_extract(interest, full_text);
  start_time_ = std::chrono::steady_clock::now();

  return BT::NodeStatus::RUNNING;
}

BT::NodeStatus ExtractInfoClientAction::onRunning()
{
  auto elapsed = std::chrono::steady_clock::now() - start_time_;
  if (elapsed > std::chrono::seconds(10)) {
    RCLCPP_ERROR(hri_client_->get_logger(), "Extract operation timeout");
    return BT::NodeStatus::FAILURE;
  }

  if (hri_client_->is_extract_done()) {
    std::string extracted_info = hri_client_->get_extracted_info();

    if (extracted_info.empty()) {
      RCLCPP_WARN(hri_client_->get_logger(), "Extract returned empty
     result");
      return BT::NodeStatus::FAILURE;
```

```
    }

    RCLCPP_INFO(hri_client_->get_logger(), "Extracted: '%s'",
     extracted_info.c_str());
    setOutput("extracted_info", extracted_info);

    return BT::NodeStatus::SUCCESS;
  }

  return BT::NodeStatus::RUNNING;
}

void ExtractInfoClientAction::onHalted()
{
  RCLCPP_WARN(hri_client_->get_logger(), "Extract action halted");
}
```

Fragmento C++ 5.11: Implementación del nodo ExtractInfoClient (extract_info_client_action.cpp)

La lógica del nodo:

- **Constructor**: Obtiene `HRIClient` del blackboard con `blackboard->get("hri_client", hri_client_)`. Si no está disponible, lanza una excepción.
- **Dos puertos de entrada**: `interest` especifica qué extraer (bebida, comida, nombre de la persona...), `full_text` contiene el texto a analizar.
- **Llamada a LLM**: Usa HRIClient para enviar una petición al servicio de extracción de información basado en modelo de lenguaje.
- **Patrón asíncrono**: `start_extract()` inicia la operación, `is_extract_done()` verifica completitud, `get_extracted_info()` obtiene el resultado.
- **Puerto de salida**: Escribe la información extraída al blackboard para que otros nodos la usen (ej: pasar la bebida al nodo de navegación a cocina).
- **Timeout corto**: 10 segundos, asumiendo que el LLM responde rápidamente (puede ajustarse según el servicio usado).
- **Ejemplo de uso**: Si el texto completo es “Quiero un café y una tostada” y el interés es “bebida”, el nodo extraería “café”.



6 Arquitectura Misión–Tarea–Capacidad

6.1 **Objetivo** **215**
6.2 **Guión de desarrollo** . . . **215**
6.3 **Teoría y herramientas para el ejercicio** **216**
Asignación de modelos por nivel 216
Pila de implementación y criterios prácticos 216
6.4 **Ejemplo de referencia: `mission_task_example`** . **217**
Arquitectura del ejemplo de referencia 217
Patrón reutilizable: TaskLifecycleNode 217
Patrón genérico: BTTaskNode 218
Coordinación FSM: MissionExecutor 219
Programa principal: instanciación y ejecución 220
Archivos XML de Behavior Trees 221
Compilación y ejecución del ejemplo 221

Este ejercicio se centra en diseñar e implementar una arquitectura completa siguiendo el patrón **Misión–Tarea–Capacidad**, asignando explícitamente **FSM** al nivel de misión y **Behavior Trees** al nivel de tareas.

Desde el punto de vista arquitectónico, en este ejercicio se trabaja en:

- Separar responsabilidades entre misión, tareas y capacidades, evitando mezclar lógica de coordinación y lógica de ejecución.
- Modelar la coordinación global con FSM mediante estados y transiciones basadas en eventos observables.
- Implementar tareas como BTs modulares y reutilizables, conectadas a capacidades por interfaces claras.
- Integrar lifecycle nodes para control robusto del ciclo de vida de las tareas.

Desde el punto de vista técnico, se adquirirán competencias en:

- Diseño de FSM de misión con criterios de transición y recuperación ante fallo.
- Construcción de tareas BT en XML y su ejecución desde nodos parametrizables.
- Integración de capacidades reales y simuladas con patrones reutilizables.
- Observabilidad del sistema mediante publicación de estado y trazas de ejecución.

Este ejercicio tiene un carácter integrador: combina explícitamente los dos modelos de decisión trabajados previamente y los asigna a niveles distintos de la arquitectura para mantener una separación clara de responsabilidades.

Este ejercicio se apoya principalmente en los conceptos introducidos en varios capítulos de la Parte I, en particular:

- Capítulo 1: modelo Misión–Tarea–Capacidad y separación explícita de niveles arquitectónicos.
- Capítulo 5: uso de FSM para coordinar modos y fases globales de misión.
- Capítulo 6: uso de Behavior Trees para estructurar la ejecución interna de las tareas.

El objetivo es integrar estos conceptos en un único ejercicio, asignando FSM al nivel de misión y BT al nivel de tarea, de modo que la arquitectura resultante sea modular, reutilizable y fácil de validar.

6.1. Objetivo

El objetivo es implementar un ejercicio integrador en el que:

1. La misión global se coordina con una FSM.
2. Cada tarea operativa se implementa como un Behavior Tree independiente.
3. Las capacidades se encapsulan en nodos reutilizables con interfaces homogéneas.
4. La ejecución se gestiona con lifecycle nodes y parámetros externos de configuración.

El ejercicio debe mantener el principio de asignación de modelo por nivel: FSM para decidir *qué fase de misión toca*, BT para decidir *cómo se ejecuta cada tarea*, y capacidades para resolver acciones concretas del robot.

6.2. Guión de desarrollo

El ejercicio se desarrolla de forma incremental, validando cada bloque funcional antes de pasar al siguiente.

Para facilitar la implementación, conviene cerrar cada paso con una evidencia mínima: un esquema de la FSM y el reparto de tareas, una base ejecutable con lifecycle y BT parametrizable, y una integración final con al menos un escenario completo.

Posibles ejercicios

La estructura propuesta puede aplicarse a distintos contextos siempre que se conserve la separación Misión–Tarea–Capacidad con FSM+BT:

- **Asistente de oficina**: FSM para coordinación de fases de atención y BTs para entrega, recordatorio y cierre.
- **Guía de museo**: FSM para selección de modo de visita y BTs para cada recorrido.
- **Inspección de almacén**: FSM para planificación de zonas y BTs de inspección por área.
- **Servicio en restaurante**: FSM para ciclo de atención y BTs para toma de pedido, entrega y recogida.

Paso 1: diseño de misión y tareas

En este primer paso se define el reparto de responsabilidades y se deja cerrada la arquitectura de alto nivel.

1. Definir misión global y estados de la FSM (mínimo 4 estados con transiciones justificadas).
2. Identificar al menos 3 tareas principales y asignarlas a BTs independientes.
3. Especificar capacidades necesarias por tarea y sus interfaces de entrada/salida.

4. Validar la coherencia global del flujo antes de implementar.

El resultado esperado es un diseño suficientemente detallado para empezar a implementar sin tener que reabrir decisiones de arquitectura durante los pasos siguientes.

Paso 2: implementación base con lifecycle y BT

En este paso se construye la base reutilizable sobre la que se apoyará todo el ejercicio.

1. Implementar nodos de tarea lifecycle y un nodo genérico de ejecución BT parametrizable.
2. Definir BTs en XML externos, separados del código C++.
3. Implementar FSM de misión que gestione transiciones lifecycle (`configure/activate/deactivate/cleanup`).
4. Añadir observabilidad de estado de misión y estado de tareas.

La prioridad aquí es dejar una estructura general que permita reutilizar el mismo nodo con distintos XML y distintas misiones sin duplicar lógica.

Paso 3: integración y validación

En el último paso se comprueba que la arquitectura completa funciona de extremo a extremo.

1. Integrar al menos una capacidad real (Nav2, percepción o interacción).
2. Ejecutar escenarios nominales y al menos un escenario de fallo con recuperación.
3. Verificar reutilización de nodos y configuración por parámetros sin recompilar.

Si la integración es correcta, debe quedar claro qué parte resuelve la misión, qué parte ejecuta la tarea y qué capacidades quedan encapsuladas como piezas reutilizables.

6.3. Teoría y herramientas para el ejercicio

Asignación de modelos por nivel

En este ejercicio se adopta la asignación de modelos:

- **Misión (FSM)**: coordinación de fases globales y transiciones de alto nivel.
- **Tareas (BT)**: ejecución estructurada y recuperable de cada tarea operativa.
- **Capacidades**: acciones concretas reutilizables invocadas desde los BTs.

Pila de implementación y criterios prácticos

La implementación debe apoyarse en ROS 2, BehaviorTree.CPP v4.x, lifecycle nodes y un executor multi-hilo, con BTs definidos en XML externos y observabilidad del estado de misión.

6.4. Ejemplo de referencia: `mission_task_example`

El paquete `mission_task_example` del workspace de referencia del libro proporciona una implementación de referencia que demuestra la integración entre misión, tareas y capacidades mediante FSM, lifecycle nodes y Behavior Trees. Quien utilice este libro debería revisarlo antes de abordar el ejercicio, ya que establece las bases arquitectónicas requeridas.

Arquitectura del ejemplo de referencia

El ejemplo implementa cuatro componentes principales que pueden reutilizarse o adaptarse en el ejercicio:

- **TaskLifecycleNode**: Clase base abstracta para nodos de tarea que heredan de `LifecycleNode`. Implementa los callbacks del ciclo de vida (`on_configure`, `on_activate`, `on_deactivate`, `on_cleanup`) y define la interfaz virtual `do_task_work()` que las clases derivadas deben implementar.
- **BTTaskNode**: Clase genérica derivada de `TaskLifecycleNode` que ejecuta un árbol BT. Recibe por parámetro ROS (`bt_xml_file`) la ruta al archivo XML del BT a ejecutar. Esto permite instanciar múltiples nodos del mismo tipo para diferentes tareas sin duplicar código.
- **MissionExecutor**: Clase que implementa la FSM de coordinación. Gestiona estados de misión y coordina las transiciones lifecycle de los nodos de tarea según el flujo de la misión.
- **Main program**: Nodo ROS que instancia múltiples nodos `BTTaskNode` con diferentes parámetros (uno por cada archivo XML de tarea), los pasa al `MissionExecutor`, y ejecuta `update()` periódicamente mientras hace spin de todos los nodos con un `MultiThreadedExecutor`.

Patrón reutilizable: TaskLifecycleNode

La clase base `TaskLifecycleNode` implementa el patrón Template Method para nodos de tarea lifecycle:

```
// include/mission_task_example/task_lifecycle_node.hpp
class TaskLifecycleNode : public rclcpp_lifecycle::LifecycleNode
{
public:
  explicit TaskLifecycleNode(
    const std::string & node_name,
    const rclcpp::NodeOptions & options = rclcpp::NodeOptions());

  // Lifecycle callbacks
  CallbackReturn on_configure(const State & previous_state) override;
  CallbackReturn on_activate(const State & previous_state) override;
  CallbackReturn on_deactivate(const State & previous_state) override;
  CallbackReturn on_cleanup(const State & previous_state) override;
  CallbackReturn on_shutdown(const State & previous_state) override;

  // Task status interface
  bool is_task_complete() const { return task_complete_; }
  bool is_task_success() const { return task_success_; }
  void reset_task();

protected:
  // Hooks for derived classes
  virtual bool do_task_work() = 0;  // Return true when complete
```

```
  virtual void do_task_reset() = 0;
  void set_task_complete(bool success);

  rclcpp::TimerBase::SharedPtr timer_;
  bool task_complete_;
  bool task_success_;
  int work_counter_;
};
```

Fragmento C++ 6.1: Clase base para nodos de tarea lifecycle

Aspectos clave del diseño:

- **Separación configure/activate**: `on_configure()` resetea el estado interno, mientras que `on_activate()` inicia el timer que ejecuta el trabajo. Esto permite reconfigurar sin ejecutar.
- **Timer asíncrono**: El trabajo de la tarea (`do_task_work()`) se ejecuta periódicamente en el timer, permitiendo trabajo incremental sin bloquear.
- **Cleanup vs Deactivate**: `on_deactivate()` detiene el timer pero mantiene el estado, mientras que `on_cleanup()` resetea completamente.
- **Template method pattern**: Las clases derivadas solo implementan `do_task_work()` y `do_task_reset()`, heredando toda la gestión lifecycle.

Patrón genérico: BTTaskNode

En lugar de crear una clase derivada por cada tarea, se implementa un único nodo genérico `BTTaskNode` que ejecuta cualquier árbol BT especificado por parámetro:

```
// include/mission_task_example/bt_task_node.hpp
class BTTaskNode : public TaskLifecycleNode
{
public:
  explicit BTTaskNode(
    const std::string & node_name,
    const rclcpp::NodeOptions & options = rclcpp::NodeOptions());

protected:
  bool do_task_work() override;
  void do_task_reset() override;

private:
  std::string bt_xml_file_;                      // Parámetro: ruta al XML
  BT::BehaviorTreeFactory factory_;              // Factory de BT
  std::unique_ptr<BT::Tree> tree_;               // Instancia del BT
  BT::NodeStatus last_bt_status_{BT::NodeStatus::IDLE};
};
```

Fragmento C++ 6.2: BTTaskNode: nodo genérico que ejecuta BTs configurables

La implementación carga el BT usando lazy loading y ejecuta tick a tick:

```
bool BTTaskNode::do_task_work() {
  // Lazy loading: crear BT en primera ejecución
  if (!tree_) {
    bt_xml_file_ = this->get_parameter("bt_xml_file").as_string();

    if (bt_xml_file_.empty()) {
      RCLCPP_ERROR(get_logger(), "bt_xml_file parameter not set!");
      set_task_complete(false);
      return false;
    }

    tree_ = std::make_unique<BT::Tree>(
      factory_.createTree(bt_xml_file_));
```

```
  }

  // Tick del BT en cada iteración
  last_bt_status_ = tree_->tickRoot();

  // Marcar tarea completa cuando BT termina
  if (last_bt_status_ == BT::NodeStatus::SUCCESS) {
    set_task_complete(true);
    return true;
  } else if (last_bt_status_ == BT::NodeStatus::FAILURE) {
    set_task_complete(false);
    return false;
  }

  return false;  // BT todavía en ejecución (RUNNING)
}
```

Fragmento C++ 6.3: Implementación de BTTaskNode

Ventajas de este patrón:

- **Escalabilidad**: Añadir tareas solo requiere crear archivos XML, no código C++.
- **Configuración dinámica**: Los parámetros se cambian en launch files sin recompilar.
- **Reutilización**: Una sola clase para todas las tareas.
- **Separación de concerns**: Lógica lifecycle vs. lógica de comportamiento.

Coordinación FSM: MissionExecutor

El `MissionExecutor` gestiona transiciones de lifecycle según el estado de la FSM:

```
class MissionExecutor {
public:
  explicit MissionExecutor(
    rclcpp::Node::SharedPtr node,
    std::shared_ptr<rclcpp_lifecycle::LifecycleNode> task_a_node,
    std::shared_ptr<rclcpp_lifecycle::LifecycleNode> task_b_node,
    std::shared_ptr<rclcpp_lifecycle::LifecycleNode> task_c_node);

  void initialize();  // Configura todos los nodos
  void update();      // FSM update loop

private:
  void transition_to(MissionState new_state);
  void handle_task_a_state();
  // ... handlers para otros estados
};
```

Fragmento C++ 6.4: MissionExecutor con lifecycle nodes

La gestión de estados coordina lifecycle transitions:

```
void MissionExecutor::transition_to(MissionState new_state) {
  current_state_ = new_state;

  // Al entrar en estado de tarea, configure+activate el nodo
  if (new_state == MissionState::TASK_A) {
    task_a_node_->configure();
    task_a_node_->activate();
  }

  // Publicar estado para monitorización
  std_msgs::msg::String msg;
  msg.data = stateToString(new_state);
  state_pub_->publish(msg);
}

void MissionExecutor::handle_task_a_state() {
```

```cpp
  auto task_node =
    std::dynamic_pointer_cast<TaskLifecycleNode>(task_a_node_);

  if (task_node && task_node->is_task_complete()) {
    if (task_node->is_task_success()) {
      task_a_node_->deactivate();
      task_a_node_->cleanup();
      transition_to(MissionState::TASK_B);
    } else {
      // Manejo de fallos
      task_a_node_->deactivate();
      task_a_node_->cleanup();
      transition_to(MissionState::RECOVERY);
    }
  }
}
```

Fragmento C++ 6.5: Transiciones FSM con lifecycle

Programa principal: instanciación y ejecución

El main crea múltiples instancias de `BTTaskNode` con diferentes parámetros:

```cpp
int main(int argc, char ** argv) {
  rclcpp::init(argc, argv);

  // Localizar directorio config/ del paquete
  std::string package_share_dir =
    ament_index_cpp::get_package_share_directory("mission_task_example");
  std::string config_dir = package_share_dir + "/config";

  auto node = std::make_shared<rclcpp::Node>("mission_fsm_node");

  // Instanciar BTTaskNode con diferentes parámetros
  rclcpp::NodeOptions options_a;
  options_a.parameter_overrides({
    {"bt_xml_file", config_dir + "/task_a.xml"}
  });
  auto task_a_node = std::make_shared<BTTaskNode>("task_a_node", options_a
     );

  // ... Similar para task_b_node y task_c_node

  // Crear executor de misión
  MissionExecutor executor(node, task_a_node, task_b_node, task_c_node);
  executor.initialize();

  // MultiThreadedExecutor para todos los nodos
  rclcpp::executors::MultiThreadedExecutor executor_mt;
  executor_mt.add_node(node);
  executor_mt.add_node(task_a_node->get_node_base_interface());
  executor_mt.add_node(task_b_node->get_node_base_interface());
  executor_mt.add_node(task_c_node->get_node_base_interface());

  // Timer periódico para FSM update
  auto timer = node->create_wall_timer(
    std::chrono::milliseconds(100),
    [&executor]() { executor.update(); });

  executor_mt.spin();
  rclcpp::shutdown();
  return 0;
}
```

Fragmento C++ 6.6: Main: instanciación de tareas genéricas

Aspectos críticos:

- **Configuración por parámetros**: Cada `BTTaskNode` recibe su XML vía `parameter_overrides`.

- **MultiThreadedExecutor**: Necesario para múltiples nodos lifecycle concurrentes.
- **Timer FSM**: La FSM se actualiza periódicamente a 10 Hz.
- **Localización de recursos**: Uso de `ament_index_cpp` para encontrar archivos de configuración.

Archivos XML de Behavior Trees

Cada tarea se define en un archivo XML separado:

```xml
<root main_tree_to_execute="TaskA">
  <BehaviorTree ID="TaskA">
    <Sequence name="TaskASequence">
      <Action ID="PrintMessage" message="Task A: Starting..." />
      <Action ID="Wait" duration="1.0" />
      <Action ID="PrintMessage" message="Task A: Working..." />
      <Action ID="Wait" duration="1.0" />
      <Action ID="PrintMessage" message="Task A: Completed!" />
    </Sequence>
  </BehaviorTree>
</root>
```

Fragmento C++ 6.7: Ejemplo de task_a.xml

Compilación y ejecución del ejemplo

Para probar el ejemplo de referencia:

```bash
$ cd ~/asr_ws
$ colcon build --packages-select mission_task_example
$ source install/setup.bash
$ ros2 launch mission_task_example mission_task_example.launch.py
```

Monitorización del sistema:

```bash
# Estado FSM
$ ros2 topic echo /mission_state

# Estados lifecycle individuales
$ ros2 lifecycle list

# Transiciones lifecycle
$ ros2 topic echo /task_a_node/transition_event
```

Appendix

Bibliography

Here are the references in citation order.

[1] Francisco M. Rico. *A Concise Introduction to Robot Programming with ROS2*. 1.ª ed. Chapman y Hall/CRC, 2022 (vid. págs. 10, 11).

[2] Francisco M. Rico. *A Concise Introduction to Robot Programming with ROS 2*. 2.ª ed. Chapman y Hall/CRC, 2025 (vid. págs. 10, 11, 98, 115, 118-120, 122-134).

Alphabetical Index

.NET, 116

Abstracción hardware, 5
Abstracción perceptiva, 46
Acciones, 24
Acción
  transición, 67
Adquisición sensorial, 33
Antipatrones, 108
Arbitraje reactivo, 55
Arquitectura de decisión, 65
Arquitectura en capas, 52
Arquitectura software, 2
  definición, 2
  robótica, 2
Arquitecturas de referencia, 97
Arquitecturas deliberativas, 50
Arquitecturas distribuidas, 111
Arquitecturas híbridas, 50
Asincronía, 13
Atributos de calidad, 109

Behavior Trees, 78
BehaviorTree.CPP, 92
Benchmarking arquitectónico, 114
Blackboard, 93
BT
  misión y tarea, 89

C++, 115
Calibración, 27
Callback Group, 122
Callback groups, 122
Callbacks, 13
  diseño, 15
Capas
  mapeo en ROS 2, 57
Ciclo de vida de nodos, 59
Comparación BT vs FSM, 91
Comunicación
  ROS 2, 10
Concurrencia, 3, 13
  FSM, 70
Condiciones de carrera, 70
Configuración, 113
Contratos, 113
Contratos de datos, 22
Contratos de interacción, 52
Costmap, 97
CycloneDDS, 116

Datos compartidos
  BT, 93
DDS, 116
Decoradores, 88
Deep ROS 2, 97
Deliberación, 50
Depuración, 110
Despliegue, 113
Despliegue distribuido, 18
Detección, 46
Diseño arquitectónico, 2
Diseño compositivo, 76
Diseño de ROS 2, 115
Drivers, 27

EasyNav, 105
Ejecución reactiva, 13
Escalabilidad, 72
Estado del mundo, 56
Estados, 65
Estimación de estado, 32
Estrategias de tiempo real, 127
Evaluación arquitectónica, 109
Evaluación recursiva, 81
Evento, 67
Eventos, 13
Executor, 117
executor, 117
Explosión de estados, 72

FAILURE, 80
Fallback, 84
FastDDS, 116
Flujo de datos, 15
ForceFailure, 89
ForceSuccess, 89
Frames, 35
Frameworks
  ROS 2, 97
FSM, 65
  nivel misión, 75
  nivel tarea, 74
Fusión sensorial, 32

Gestión de ejecución, 117
Gestión de errores, 111
Go, 116
Grafo de computación, 57
Guarda, 67

IMU, 39
Interfaces, 113
Interfaces de datos, 21
Inverter, 88

Java, 116
Jerarquía de estados, 73

Latencia, 125
LIDAR, 39
Lifecycle Nodes, 59
Linux scheduler, 126

Marcos de referencia, 30
Matrices homogéneas, 30
Mealy, 68
Mensajes, 22
Middleware robótico, 5
Middlewares de programación de robots, 5
Misión–tarea–capacidad, 52
Modelado de misión, 72
Modelado de tarea, 72
Modelo del mundo, 50
  ROS 2, 61
Modelo reactivo, 13
Modularidad, 79
Moore, 68
mutually exclusive, 123
Máquinas de estados finitos, 65
Máquinas jerárquicas, 73

Nav2, 97
Navigation2, 97
Nodos, 10
  Behavior Tree, 80
  ROS 2, 18
Nodos de acción, 80
Nodos de condición, 80
Nodos de control, 80
NPC, 78

Observabilidad, 9
OMG, 116
on_do, 66
on_entry, 66
on_exit, 66

Parallel, 85

Patrones arquitectónicos, 108
PDDL, 101
Percepción deliberativa, 61
Percepción robótica, 27
Percepción–decisión–actuación, 3
Petición-respuesta, 24
Pipeline perceptivo, 29
PlanSys2, 101
Polling Point, 118
Procesos, 10
Productor-consumidor, 15
Publicación-subscripción, 19
Publishers, 19
Puertos, 93

rcl, 115
rclcpp, 115
rclpy, 115
Reactividad, 79
Real-Time, 125
Reducción de dimensionalidad, 29
reentrant, 124
Regiones ortogonales, 70
Replanificación, 50
Representación del entorno, 56
Representación espacial, 27
Retornos SUCCESS/FAILURE/RUNNING, 80
Retry, 88
Reutilización, 90
RGB-D, 39
RMW, 115
rmw, 117
RMW_IMPLEMENTATION, 117
Robótica software, 2
ROS 2
  arquitectura, 9
  Behavior Trees, 92
RTI Connext, 116
RUNNING, 80
Rust, 116

Safety, 112
SCHED_DEADLINE, 126
SCHED_FIFO, 126
Seguridad, 112
Selector, 84
Sense-Think-Act, 3
Sensores
  ROS 2, 33
Sensores crudos, 27
Sequence, 82
Servicios, 24
SingleThreadedExecutor, 117
Sistema operativo, 114
Starvation, 119
Statecharts, 70
StaticSingleThreadedExecutor, 119
Subscribers, 19
Subsumption, 55
SUCCESS, 80
Supervisión, 50

Testing, 110
TF, 35
Tick, 81
Tiempo real, 125
Timeout, 89
Tolerancia a fallos, 111
Topics, 21
Topologías de FSM, 68
Transform tree, 35
Transformaciones rígidas, 30
Transformación de datos, 29
Transiciones, 65
Trazabilidad, 9

UDP, 116

Validación, 110
Videojuegos, 78

wait-set, 118

Árboles de comportamiento, 78