Capitulo 7: Arquitecturas de referencia y Deep ROS 2

Nav2, PlanSys2, EasyNav, executors y estrategias de tiempo real

Francisco Martin Rico y Rodrigo Perez Rodriguez

1. Parte I – Arquitecturas de referencia
2. Parte II – Sintesis transversal
3. Parte III – Deep ROS 2

1 Parte I – Arquitecturas de referencia
- Nav2
- PlanSys2
- EasyNav

2 Parte II – Sintesis transversal

3 Parte III – Deep ROS 2

1 Parte I – Arquitecturas de referencia
Nav2
PlanSys2
EasyNav

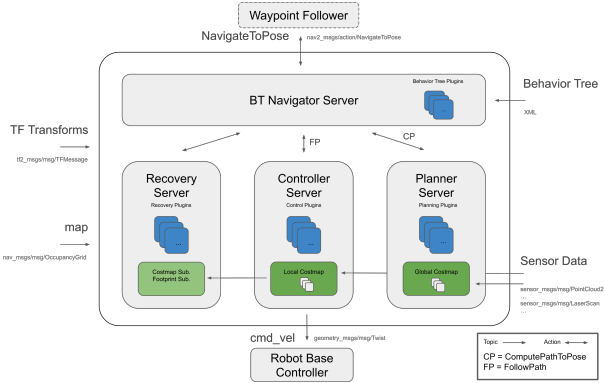
2 Parte II – Sintesis transversal

3 Parte III – Deep ROS 2

Nav2

Referencia de navegacion en ROS 2

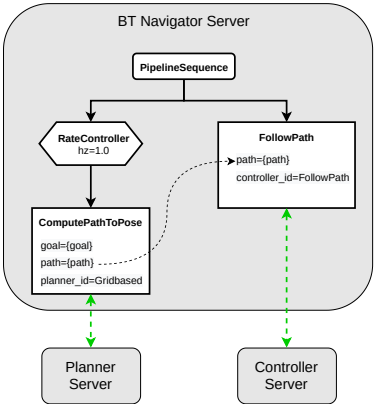
- Se situa en la capa de **capacidad**: resuelve la habilidad de navegar.
- Se organiza como **servidores especializados**: planner, controller, recovery, smoother, costmaps.
- El **BT Navigator** orquesta el flujo de alto nivel y la recuperacion ante fallos.
- Representa un estilo muy alineado con ROS 2: componentes distribuidos, plugins y actions.



Nav2

BT Navigator (Behavior Tree)

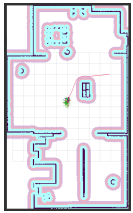
- El **BT Navigator** ejecuta un **Behavior Tree** para orquestar la navegacion.
- El arbol expresa de forma explicita **replanificacion**, **monitorizacion** y **recuperacion**.
- Las hojas suelen delegar en **actions** y servidores (planner/controller/recovery), manteniendo la variabilidad en **plugins**.



Nav2

Decisiones arquitectonicas clave

- **Plugins**: capturan variabilidad sin romper la arquitectura.
- **Costmaps + TF**: contrato espacial compartido para planificar y controlar.
- **Actions**: expresan objetivo, feedback, resultado y preemption.
- **Lifecycle**: mejora despliegue y operabilidad de cada servidor.
- **Trade-off**: mucha modularidad a cambio de mas trafico interno y complejidad de configuracion.

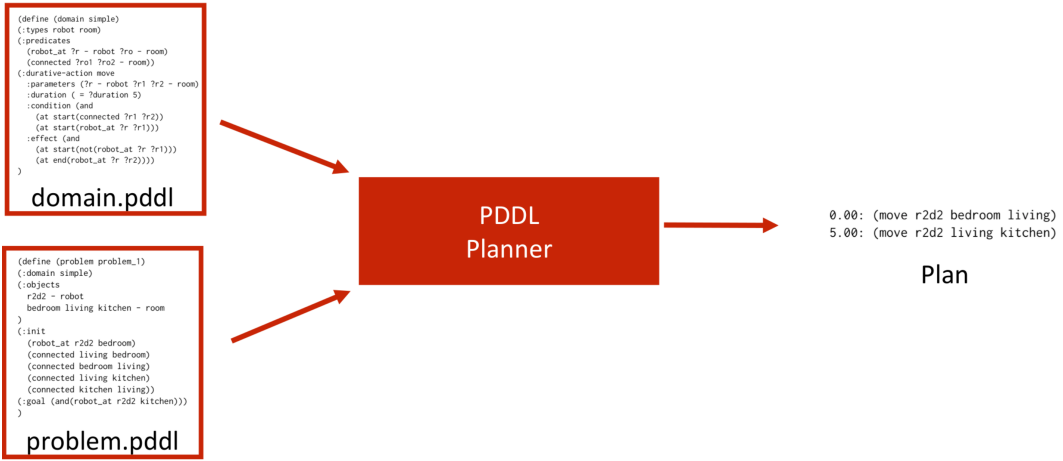


1 Parte I – Arquitecturas de referencia
- Nav2
- PlanSys2
- EasyNav

2 Parte II – Sintesis transversal

3 Parte III – Deep ROS 2

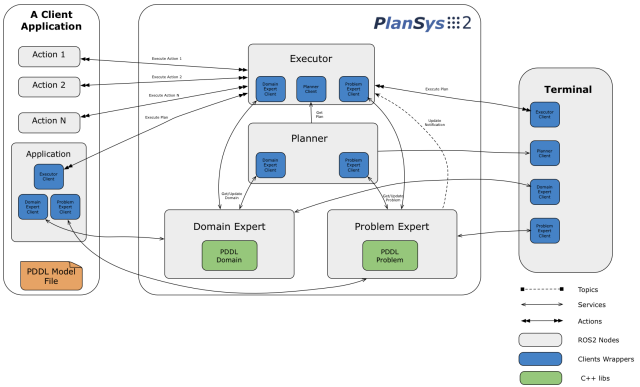
Cómo funciona PDDL



PlanSys2

Deliberacion simbolica en la capa de mision

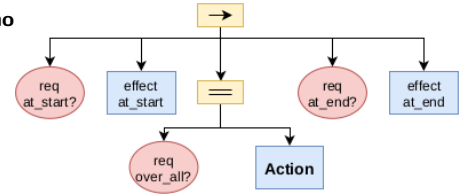
- Lleva PDDL a ROS 2 para razonar sobre **metas, estado y planes**.
- Separa cuatro responsabilidades: **Domain Expert**, **Problem Expert**, **Planner** y **Executor**.
- Encaja de forma natural con una arquitectura **mision–tarea–capacidad**.
- El **Problem Expert** materializa la memoria simbolica del sistema.



PlanSys2

Planificacion, ejecucion y BTs

- El planificador decide **que hacer**; la ejecucion decide **como reaccionar**.
- El Executor usa **Behavior Trees** para supervisar planes, dependencias y recuperaciones.
- Las acciones PDDL se conectan con ROS 2 mediante **performers** y un protocolo propio.
- Patron tipico: una accion simbolica como `move(robot, wp)` delega internamente en Nav2.
- La robustez real exige **monitorizacion y replanning**, no solo planificacion inicial.



1 Parte I – Arquitecturas de referencia
Nav2
PlanSys2
EasyNav

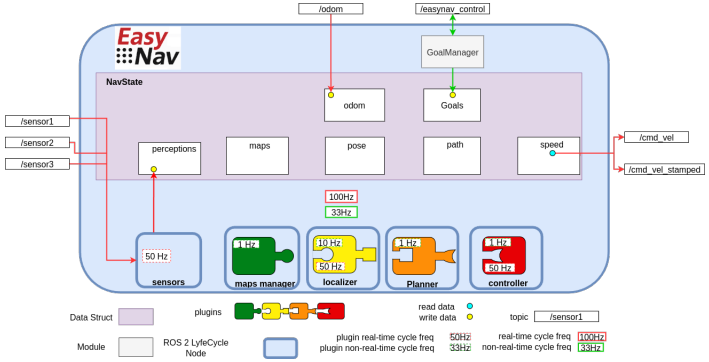
2 Parte II – Sintesis transversal

3 Parte III – Deep ROS 2

EasyNav

Navegacion componible en un unico proceso

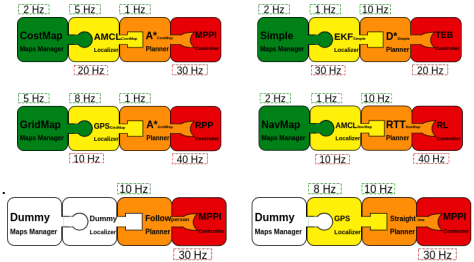
- Alternativa a Nav2 que no fija una sola representacion espacial.
- Compone modulos dentro de un **SystemNode** con lifecycle.
- Explicita **hilos y ciclos** con distintos propositos: tiempo real, no-tiempo-real y TF interno.
- Busca reducir carga sobre ROS 2 reservandolo principalmente para la frontera externa.



EasyNav

Blackboard, plugins y trade-offs

- El **NavState** actua como blackboard compartido y *thread-safe*.
- Los modulos internos cooperan sin topic/service/action para cada intercambio.
- Mantiene **plugins** para planificadores, controladores, localizadores y mapas.
- **Ventaja**: menos latencia y menos ruido en middleware.
- **Coste**: mayor cuidado con contratos internos, concurrencia y aislamiento de fallos.



Arquitecturas de referencia

Lectura comparada

Framework	Nivel	Idea central	Trade-off dominante
Nav2	Capacidad	Servicios especializados + BT + plugins	Modularidad frente a sobrecarga distribuida
PlanSys2	Mision	Estado simbolico + planner + executor	Deliberacion rica frente a sincronizacion con mundo real
EasyNav	Capacidad	Monoproceso compuesto + blackboard	Menos latencia frente a mas acoplamiento interno

- Los tres frameworks concretan decisiones arquitectonicas ya vistas: capas, contratos, memoria, supervision y tiempo.
- La pregunta correcta no es cual es *mejor*, sino **que supuestos de concurrencia, representacion y despliegue hace cada uno**.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Patrones arquitectonicos y antipatrones

- **Patron**: solucion recurrente a un problema estructural recurrente; **antipatron**: atajo que degrada el sistema cuando crece.
- **Patrones clave**: blackboard, pipeline/dataflow, publish–subscribe, componentes, plugins y microservicios frente a monolito compuesto.
- **Antipatrones tipicos**: abuso de topics, acoplamiento oculto, logica dispersa en callbacks y estado distribuido inconsistente.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Como evaluar una arquitectura robotica

- Evaluar arquitectura no es solo ver si *funciona*, sino medir propiedades globales del sistema bajo carga y cambio.
- **Criterios centrales**: latencia extremo a extremo, throughput, determinismo, escalabilidad, acoplamiento, observabilidad y testabilidad.
- Ningun criterio se optimiza gratis: mas modularidad puede implicar mas latencia; mas integracion puede exigir mas disciplina interna.
- La pregunta correcta no es cual es la mejor arquitectura, sino para que contexto y con que restricciones lo es.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Testing, validacion y debugging arquitectonico

- El testing arquitectonico valida interacciones entre modulos, tiempos, contratos y modos de despliegue, no solo unidades aisladas.
- **Tecnicas clave**: integracion, simulacion vs real, reproducibilidad, logging estructurado, tracing e introspeccion del sistema.
- Una arquitectura buena debe permitir explicar que ha pasado cuando el robot falla o se degrada, no solo seguir operando.
- Sin observabilidad y repetibilidad, muchos fallos emergentes quedan reducidos a anecdotas imposibles de analizar.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Gestion de errores y tolerancia a fallos

- Tolerar fallos es decidir como detectar, aislar, degradar y recuperar el sistema cuando algo sale mal.
- **Mecanismos tipicos**: retries, watchdogs, modos degradados, reinicio parcial y estrategias de recovery explicitas.
- No todos los fallos son iguales: transitorios, persistentes, logicos, temporales o de informacion requieren respuestas distintas.
- Robustez no es solo resistir: es tener una politica clara de respuesta ante fallos parciales y totales.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Arquitecturas distribuidas reales

- Distribuir no es solo tener varios nodos: entran en juego reloj, red, replicas del estado y disponibilidad.
- **Problemas clave**: sincronizacion entre maquinas, jitter, perdida, particiones de red y consistencia del estado distribuido.
- La decision **edge vs cloud** redistribuye latencia, capacidad de computo, observabilidad, seguridad y modos de fallo.
- En sistemas reales, la red deja de ser transporte neutro y pasa a formar parte del comportamiento del robot.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Seguridad y *safety*

- *Safety* y seguridad no son anexos finales: condicionan aislamiento, modos seguros y recuperacion del sistema.
- **Safety**: fail-safe, estados seguros por defecto, limitacion del radio de impacto y supervision de recuperaciones.
- **Seguridad**: autenticacion, confidencialidad, control de acceso, segmentacion de red y proteccion de interfaces.
- Ambas dimensiones exigen que el sistema falle de forma acotada y no propague sin control un problema local.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Diseno de interfaces y contratos

- Una interfaz arquitectonica fija que ofrece un modulo, que espera del resto y como puede evolucionar sin romper el sistema.
- Incluye **tipos, semantica temporal, errores, pre/postcondiciones, versionado y compatibilidad**.
- Las buenas interfaces reducen conocimiento innecesario entre modulos y separan dependencia util de acoplamiento accidental.
- Una interfaz pobre obliga a depender de convenciones tacitas y vuelve costosa cualquier evolucion del sistema.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Configuracion y despliegue

- La arquitectura tambien incluye como se configura, se arranca, se reproduce y se opera el sistema real.
- **Temas clave**: parametrizacion sistematica, configuracion reproducible, launch complejo y despliegue robusto en robots reales.
- Una mala configuracion introduce opacidad, errores dependientes del entorno y secuencias de arranque fragiles.
- Si el despliegue no es repetible, la arquitectura tampoco lo es en sentido practico.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Relacion con el sistema operativo

- La arquitectura real depende tambien del scheduler, la memoria, las afinidades y las prioridades del sistema operativo.
- Cuando hay carga y restricciones temporales, Linux deja de ser un detalle y pasa a moldear latencia, interferencia y predictibilidad.
- Pensar solo en nodos y diagramas es insuficiente: importa tambien como viven esos modulos sobre CPU, memoria y politicas temporales.
- La diferencia entre comportamiento ideal y comportamiento real suele aparecer precisamente en este nivel.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal
- Patrones y antipatrones
- Evaluacion arquitectonica
- Testing y depuracion
- Errores y robustez
- Distribucion real
- Seguridad y safety
- Interfaces y contratos
- Configuracion y despliegue
- Arquitectura y sistema operativo
- Ingenieria comparativa

3 Parte III – Deep ROS 2

Sintesis transversal

Ingenieria y evaluacion comparativa

- Ingenierizar una arquitectura es justificar con criterios y metricas por que una organizacion es adecuada para un problema concreto.
- Hay que trabajar con **trade-offs explicitos**: modularidad, determinismo, escalabilidad, observabilidad, mantenibilidad y rendimiento.
- Esto evita dos extremos: la solucion *bonita* sin evidencia y la acumulacion pragmatica de decisiones sin criterio.
- Las metricas son las que convierten intuiciones arquitectonicas en comparaciones tecnicamente defendibles.

1 Parte I – Arquitecturas de referencia

2 Parte II – Sintesis transversal

3 Parte III – Deep ROS 2
- Capas y middleware
- Executors y callback groups
- Tiempo real

1 Parte I – Arquitecturas de referencia

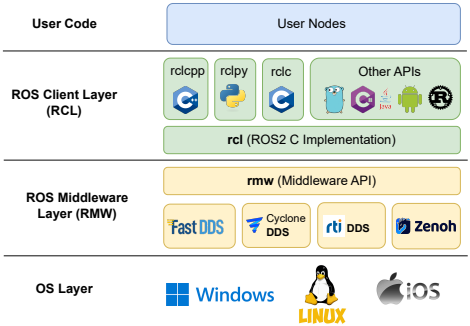
2 Parte II – Sintesis transversal

3 Parte III – Deep ROS 2
- Capas y middleware
- Executors y callback groups
- Tiempo real

Deep ROS 2

Capas de software y middleware

- **rclcpp** y **rclpy** exponen la API al programador.
- Ambas se apoyan en **rcl**, nucleo funcional comun.
- **rmw** desacopla la API del middleware concreto.
- DDS ha sido la base historica; **Zenoh** amplia el espacio de diseño en ROS 2 reciente.
- El rendimiento real depende no solo del codigo de usuario, sino tambien de la implementacion de middleware elegida.



1 Parte I – Arquitecturas de referencia

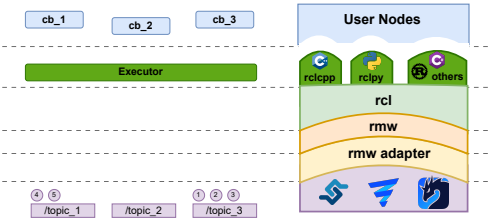
2 Parte II – Sintesis transversal

3 Parte III – Deep ROS 2
- Capas y middleware
- Executors y callback groups
- Tiempo real

Executors en ROS 2

Los callbacks no "corren solos"

- Un nodo ejecuta su logica mediante **callbacks** disparados por eventos.
- El **executor** espera trabajo, extrae eventos y ejecuta callbacks.
- La unidad relevante de planificacion practica no es el nodo, sino el **callback group**.
- Por eso aparecen efectos como *jitter*, colas pendientes o *starvation* aunque la API superficial parezca simple.



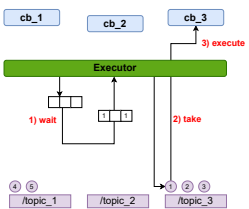

```
rclcpp::executors::SingleThreadedExecutor exec;
exec.add_node(node);
exec.spin();
```

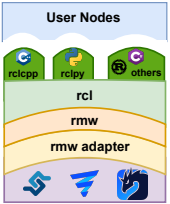
Executors en ROS 2

Wait-set y ventana de procesado

1. El executor espera en un **wait-set** hasta detectar trabajo pendiente.
2. Al activarse, abre una **ventana de procesado**.
3. Extrae mensajes/eventos listos y ejecuta los callbacks asociados.
4. Despues recompone el estado y repite el ciclo.

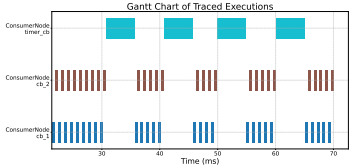
- Consecuencia: una iteracion no vacia necesariamente todas las colas.
- El orden de escaneo y la carga modifican la latencia observable.

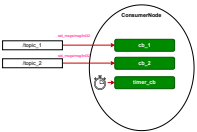


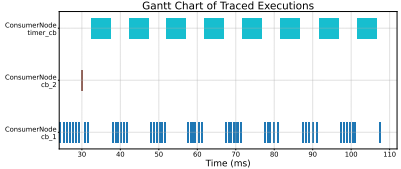


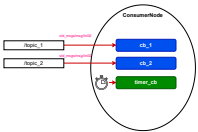
Starvation y callback groups

Mas hilos no implican automaticamente mas concurrencia util









- Con el grupo por defecto, un `MultiThreadedExecutor` puede seguir sin paralelismo util dentro del nodo.
- Separar callbacks en grupos explicitos permite controlar aislamiento, prioridad y concurrencia.

Callback groups

Mutually exclusive frente a reentrant

- **Mutually Exclusive**: dos callbacks del mismo grupo no se ejecutan a la vez.
- **Reentrant**: el executor puede lanzar callbacks del mismo grupo en paralelo.
- Reentrant no resuelve sincronizacion: simplemente **traslada la responsabilidad** al programador.
- La particion en callback groups es una decision de diseño, no un detalle de implementacion.

```
custom_cb_ = create_callback_group(
  rclcpp::CallbackGroupType::MutuallyExclusive);

rclcpp::SubscriptionOptions opts;
opts.callback_group = custom_cb_;

sub_ = create_subscription<Msg>(
  "topic", 100, cb, opts);
```

1 Parte I – Arquitecturas de referencia

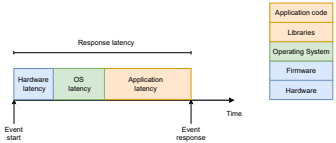
2 Parte II – Sintesis transversal

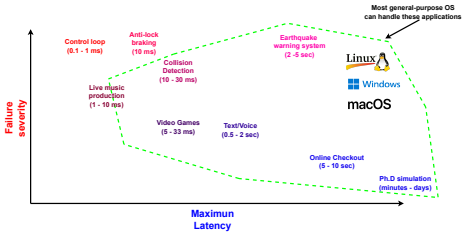
3 Parte III – Deep ROS 2
- Capas y middleware
- Executors y callback groups
- Tiempo real

Tiempo real en ROS 2

El objetivo es acotar la latencia maxima

- Tiempo real no significa solo latencia media baja, sino **latencia maxima acotada**.
- La latencia total mezcla contribuciones de aplicacion, middleware, sistema operativo y hardware.
- ROS 2 por si solo no hace el sistema determinista: hay que diseñar hilos, prioridades y rutas criticas.

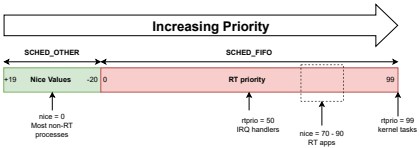




Tiempo real en Linux

SCHED_FIFO y separacion entre hilos criticos y no criticos

- El planificador por defecto optimiza latencia media, no garantias temporales estrictas.
- Una tecnica habitual es lanzar un executor en un hilo con **SCHED_FIFO**.
- La prioridad se expresa en el despliegue: que corre en hilo RT y que queda en hilo normal.

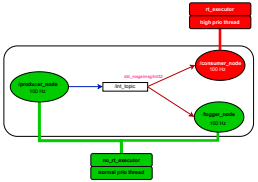


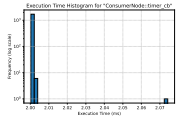

```
auto rt_thread = std::thread([&]() {
  sched_param sch;
  sch.sched_priority = 90;
  sched_setscheduler(0, SCHED_FIFO, &sch);
  rt_executor.spin();
});
```

Estrategia 1

Separar nodos criticos y no criticos en executors distintos

- Solucion mas simple cuando la frontera de criticidad coincide con **nodos completos**.
- Los nodos criticos van a un executor en hilo RT; el resto a uno normal.
- Mejora la cadencia y el tiempo de respuesta del subsistema prioritario.
- Limitacion: demasiado gruesa si un mismo nodo mezcla callbacks criticas y no criticas.

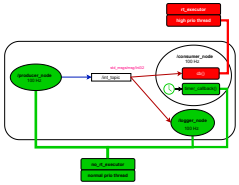


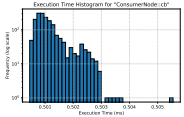


Estrategia 2

Priorizar callback groups dentro del mismo nodo

- Se crea un **callback group dedicado** para la callback prioritaria.
- El nodo entero puede vivir en el executor normal, mientras el grupo critico se añade al executor RT.
- Permite un control fino sin romper artificialmente el diseño logico del nodo.
- Exige mas disciplina con sincronizacion y recursos compartidos.

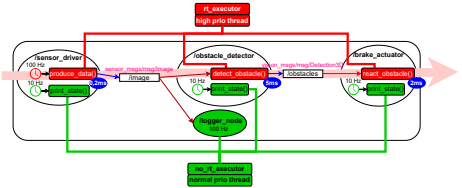


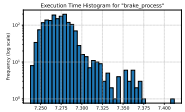


Estrategia 3

Priorizar un camino critico extremo a extremo

- Lo importante puede no ser una callback aislada, sino la **latencia total** desde percepcion hasta actuacion.
- Se identifican los callback groups que forman la cadena critica y se ejecutan en RT.
- La medida correcta es **extremo a extremo**, no solo callback a callback.
- Es la estrategia mas fiel al comportamiento real del robot.





Ideas clave del capitulo

Que conviene retener

- Los frameworks de ROS 2 materializan decisiones arquitectonicas sobre **capas, memoria, contratos y supervision**.
- En ROS 2, la concurrencia real depende de **executors, callback groups e hilos**, no solo del numero de nodos.
- El tiempo real requiere diseño explicito: **que es critico, donde se ejecuta y con que prioridad**.
- La unidad de ingenieria correcta suele ser el **camino critico extremo a extremo**, no un callback aislado.

Preguntas