

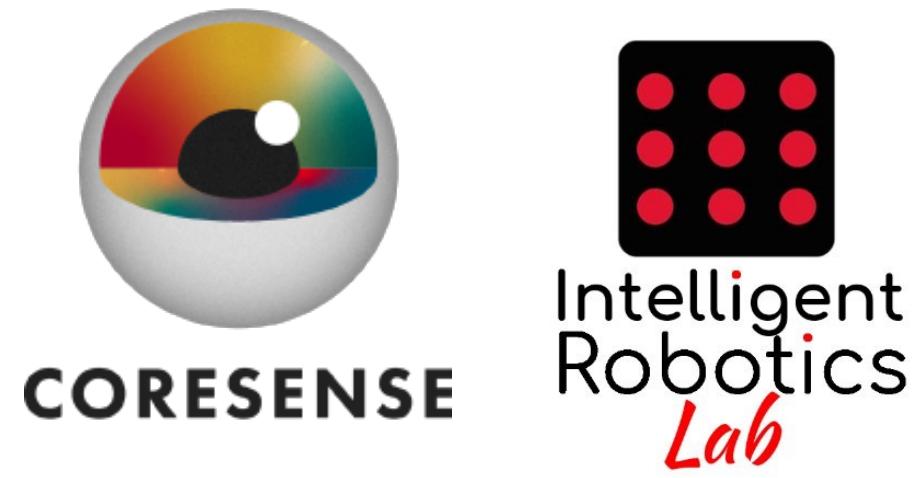
2
S
O
R
• • •

Workshop de Tiempo Real

Prof. Dr. Francisco Martín Rico
19-09-2024



francisco.rico@urjc.es
@fmrico



at

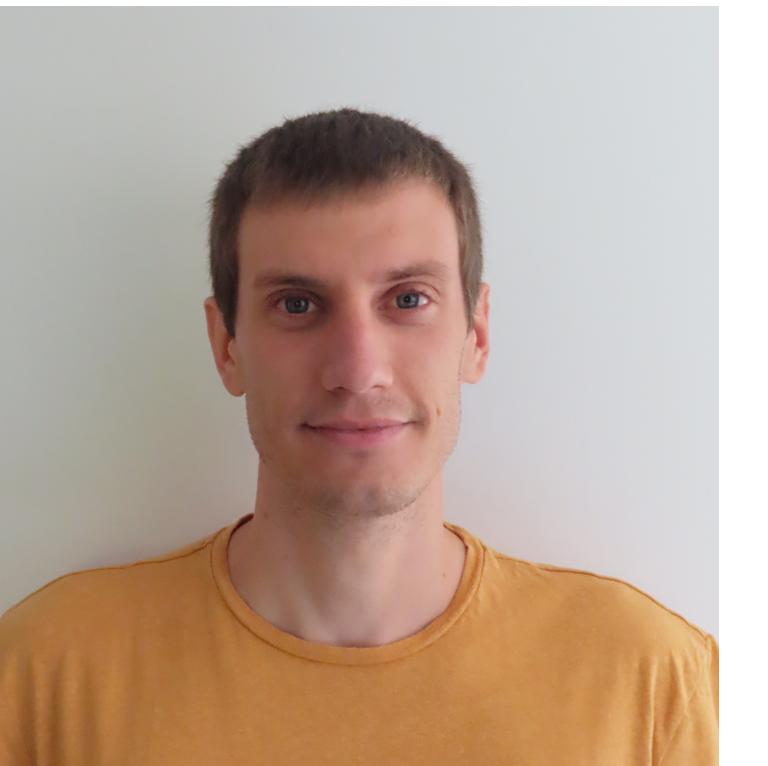
En este Workshop



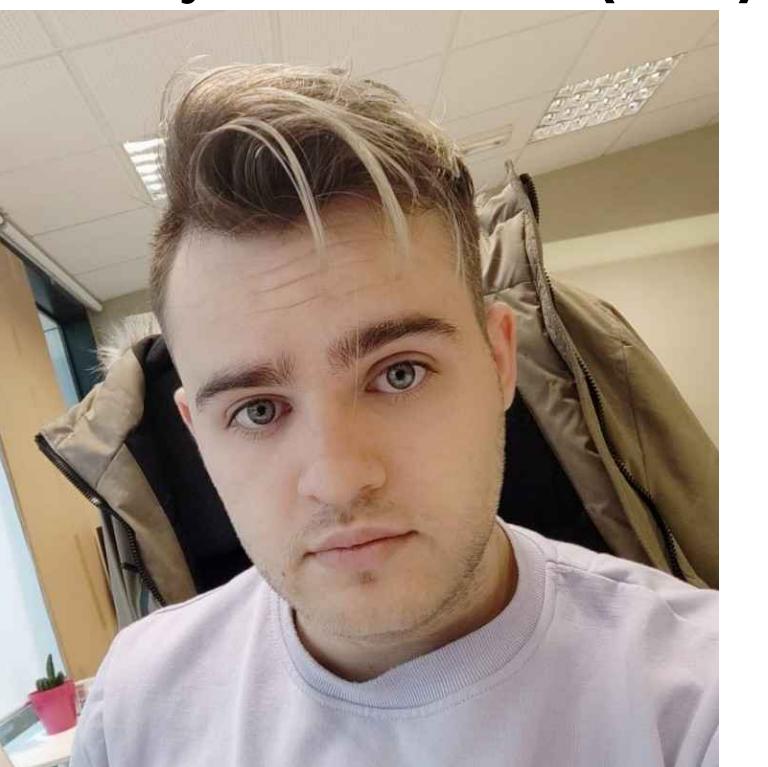
Francisco Martín Rico
Catedrático de Universidad (URJC)
ROS 2 TSC



José Miguel Guerrero
Profesor Contratado Doctor
(URJC)



**Francisco Miguel
Moreno Olivo**
Profesor Ayudante Doctor (URJC)



**Juan Carlos
Manzanares Serrano**
Técnico de Laboratorio (URJC)

Autores originales del Workshop

Este workshop está basado en el Workshop de la ROSCon 2023 de Nueva Orleans, y usa diagramas, texto e imágenes de la [presentación original](#)



Shuhao Wu
Senior software engineer
NVIDIA



Jan Staschulat
Research engineer
Bosch



Stephanie Eng
Senior controls engineer
PickNik Robotics



Oren Bell
PhD Candidate
Washington University in St Louis

Organización de este Workshop

Introducción y Definiciones

Latencia, hardware y OS

Ejercicio Práctico 1

~~Intro Programación RT Linux sin ROS~~

Gestión y Ejecución en ROS 2

Ejercicio Práctico 3

Tiempo Real + ROS 2

Ejercicio Práctico 4

Recapitulación

Introducción y definiciones



¿Qué es la programación en “Tiempo Real”?

- Tiempo Real: término ambiguo utilizado en múltiples dominios
 - Seguimiento de entregas en Tiempo Real
 - Reloj en Tiempo Real
 - Visualizaciones en Tiempo Real
 - Sistemas operativos en Tiempo Real
 - Aplicaciones en Tiempo Real
- Definámoslo para este Workshop



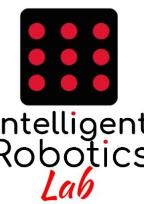
¿Qué es la programación en “Tiempo Real”?



Describe what "real-time programming" is in one sentence.



Real-time programming refers to the practice of designing and implementing software systems that can respond and produce output within strict timing constraints.



¿Qué es la programación en “Tiempo Real”?

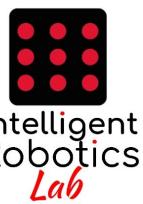


Describe what "real-time programming" is in one sentence.



Real-time programming refers to the practice of designing and implementing software systems that can respond and produce output within strict timing constraints.

- ChatGPT: “Respond [...] with strict timing constraints”



¿Qué es la programación en “Tiempo Real”?

- Criterios en Tiempo Real 1: responder dentro de los plazos establecidos
- ChatGPT no se tiene en cuenta, pero la gente suele mencionar “soft”, “hard” y “firm” en Tiempo Real.
- También es ambiguo, por lo que debemos definirlo.



¿Qué es la programación en “Tiempo Real”?



Describe what "real-time programming" is in one sentence.



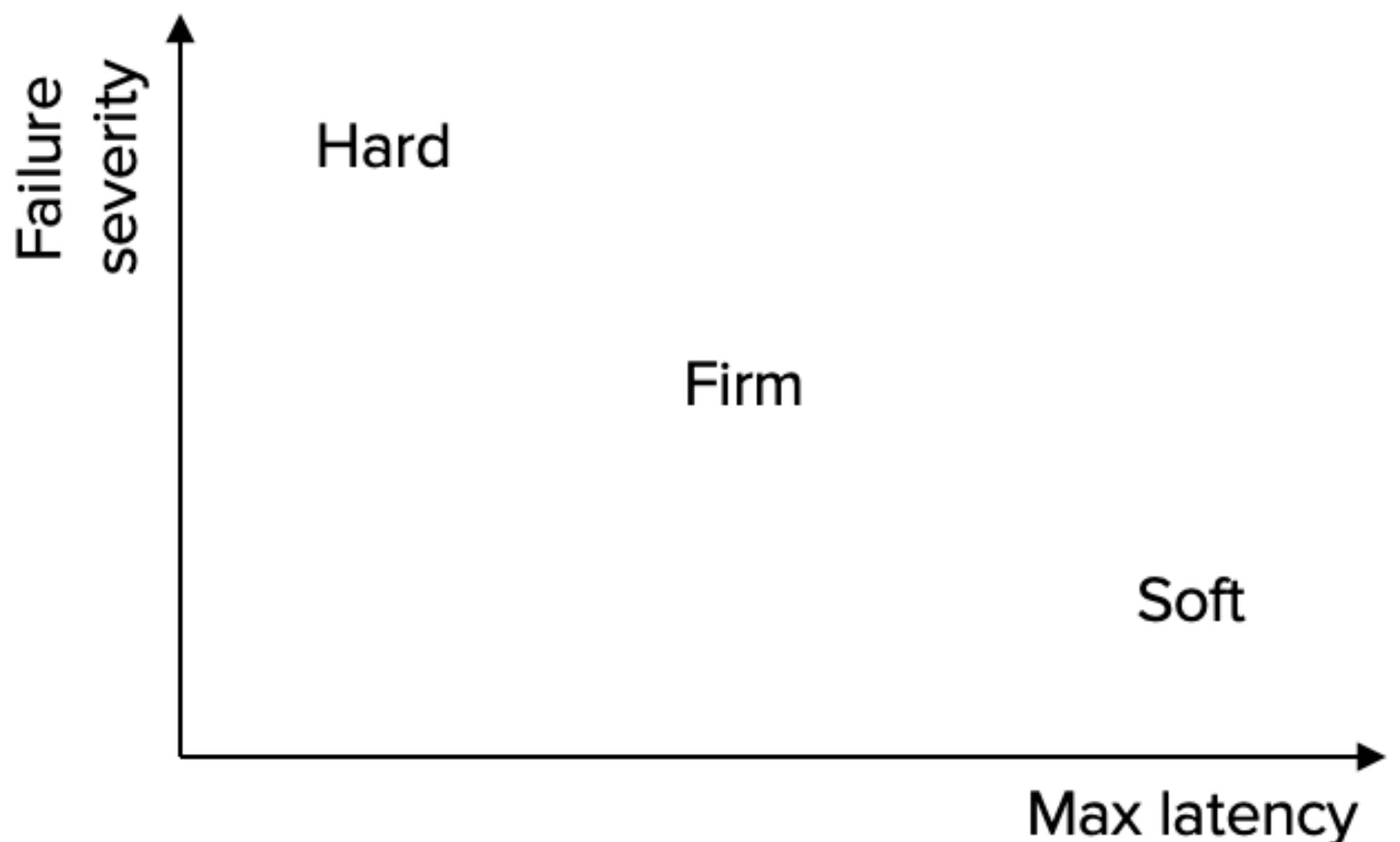
Real-time programming refers to the practice of designing and implementing software systems that can respond and produce output within strict timing constraints.

- Tiempo Real “duro” → tiene consecuencias catastróficas
 - Algunos lo definen como sistemas que están matemáticamente probados
- Tiempo Real “blando” → sin consecuencias catastróficas
- Tiempo Real “firme” → en algún punto intermedio entre estos dos
- Estos forman un “espectro” de software en Tiempo Real

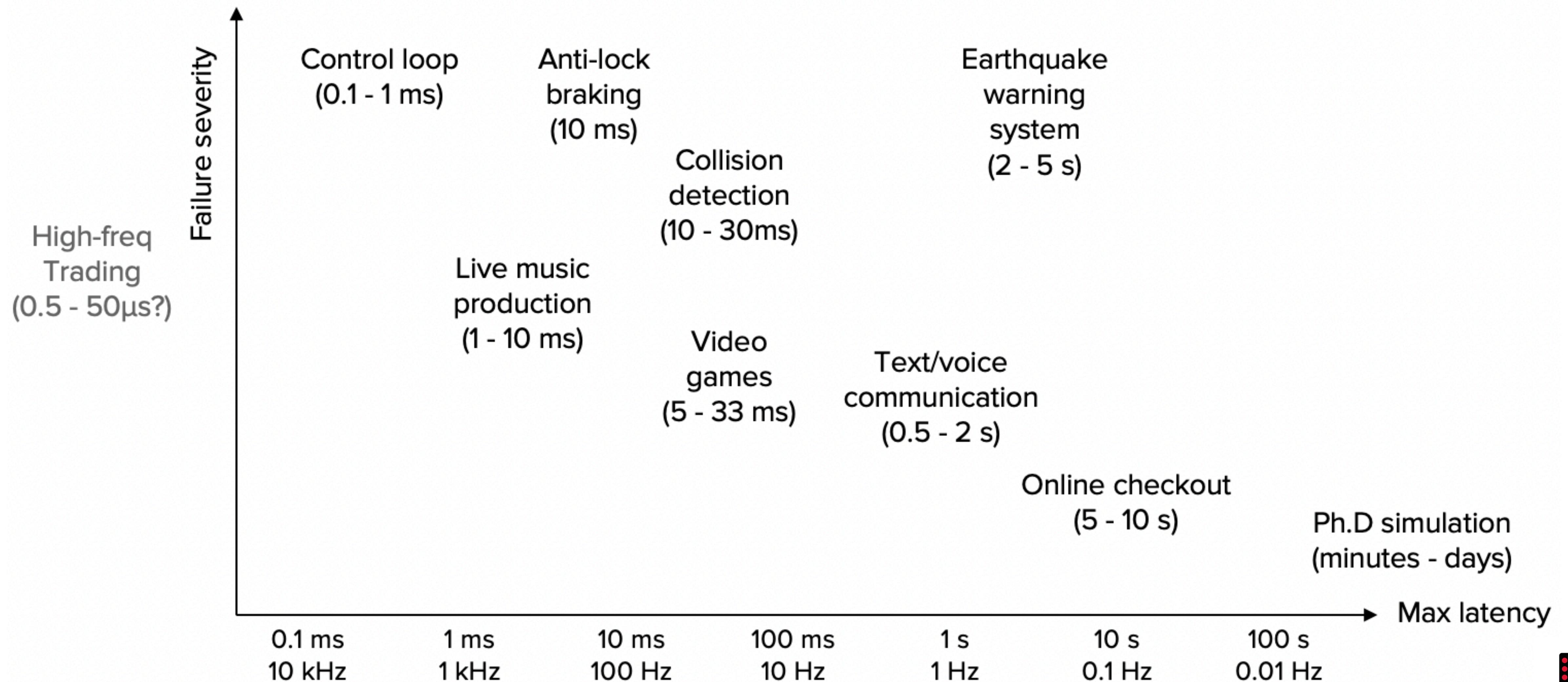


Definiendo “Tiempo Real”

- Criterios de Tiempo Real 1: **responder dentro de los plazos establecidos**
 - Latencia máxima limitada
- Criterios de Tiempo Real 2: **consecuencias de no cumplir los plazos establecidos**
- El Tiempo Real como un “espectro”

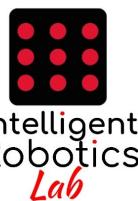


Ejemplos de sistemas de Tiempo Real



Tiempo Real y Robótica

- Bucles de control de alta frecuencia (~1000 Hz)
 - Fallo: un controlador inestable puede provocar situaciones inseguras
- Detección de objetos y prevención de colisiones (~100 Hz)
 - Fallo: colisión
- No necesariamente Tiempo Real “duro”
 - Otros sistemas de seguridad (watchdogs, límites, etc.) para garantizar la seguridad
 - Contrapunto: algunos dominios requieren certificación



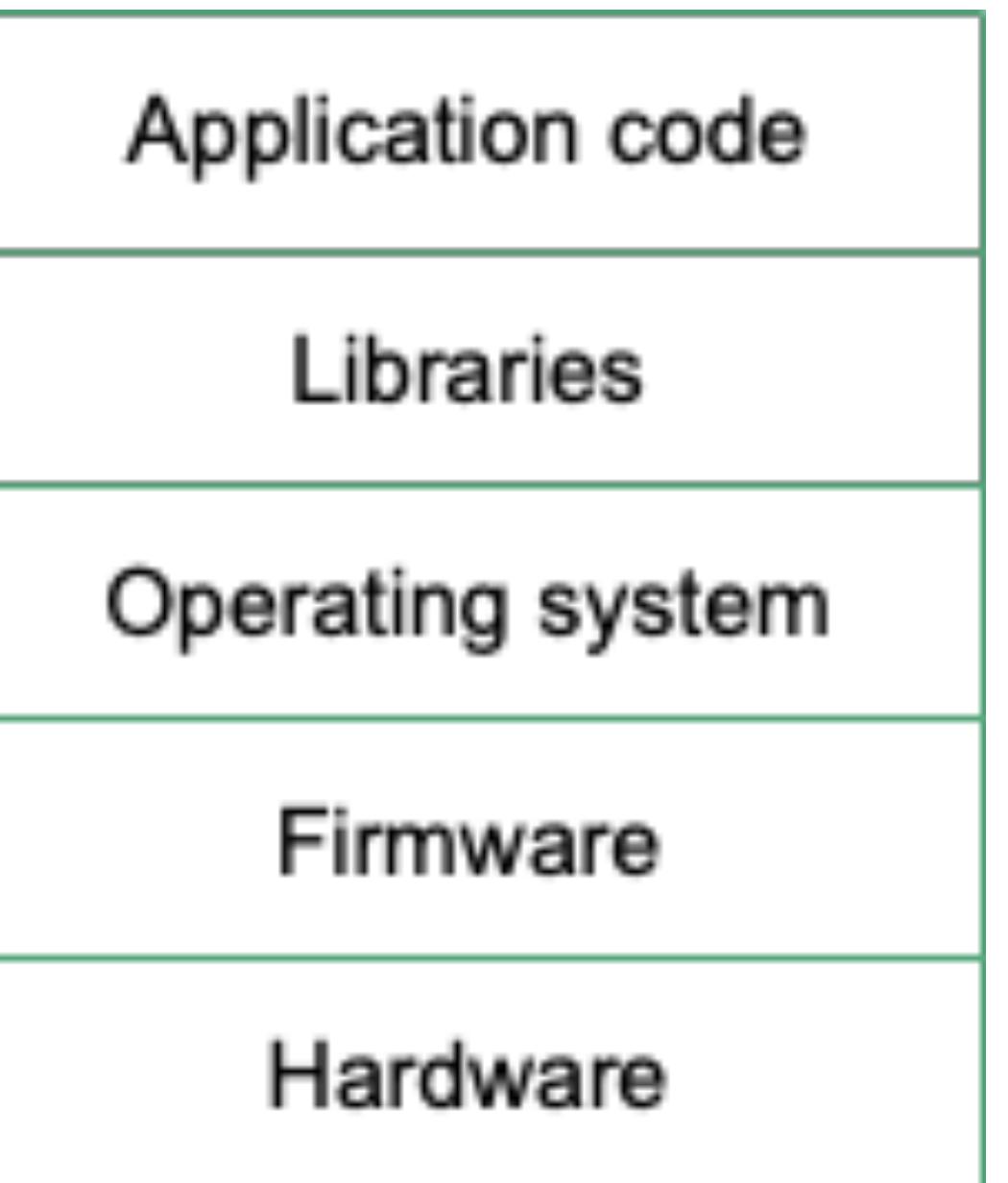
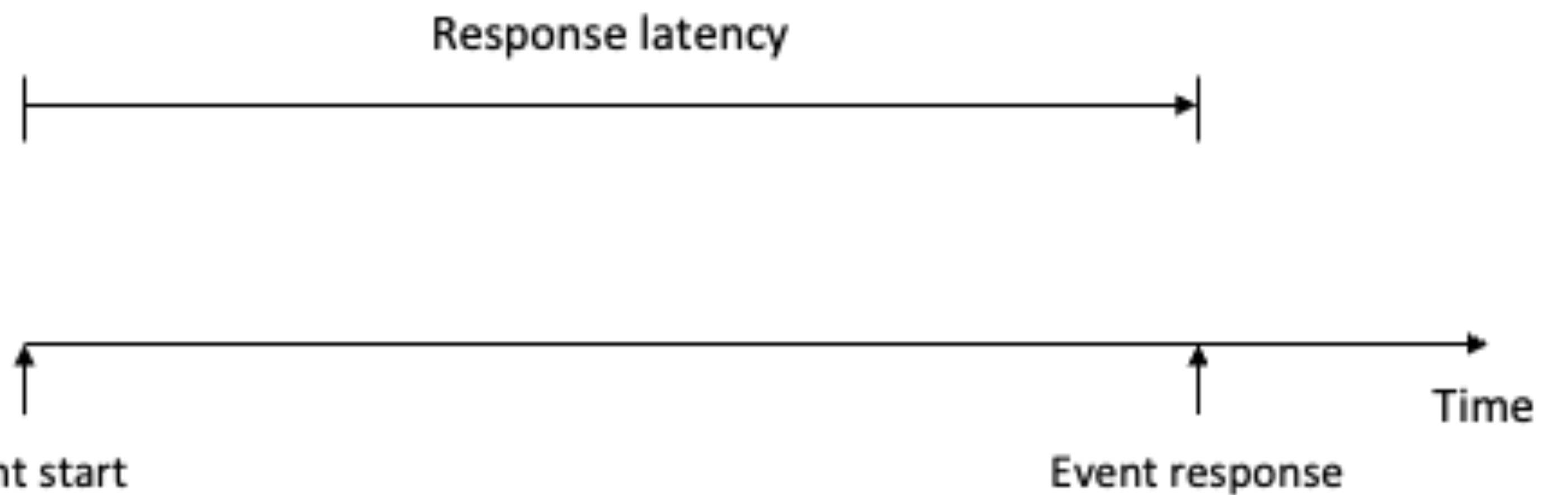
Latencia, hardware y OS

Sistemas en Tiempo Real, no software en Tiempo Real



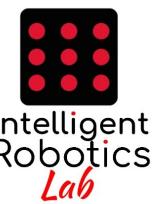
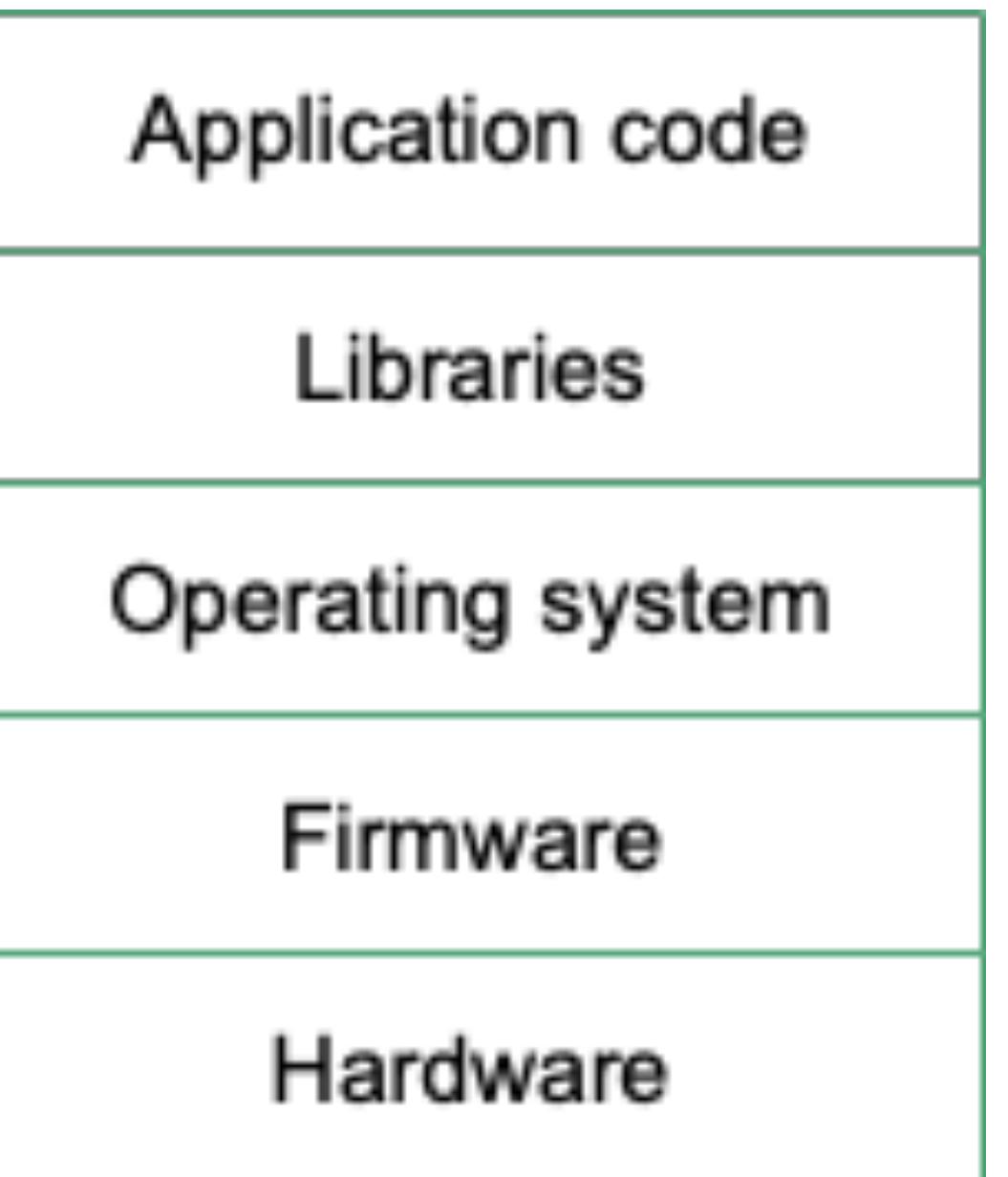
Fuentes de latencia

- Latencia de respuesta:
 - T_1 = aparece el objeto
 - T_2 = el robot se detiene
 - Latencia de respuesta = $T_2 - T_1$



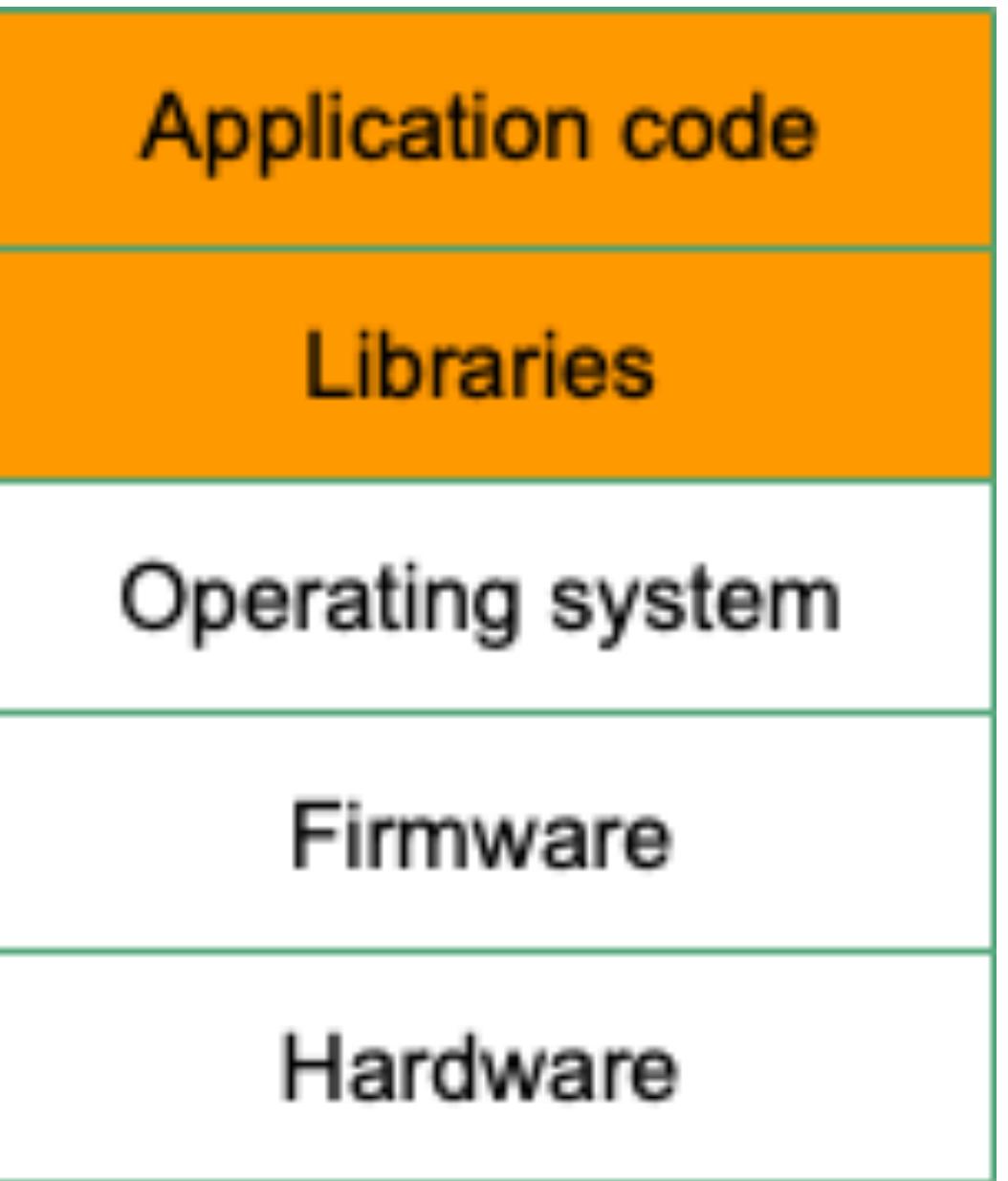
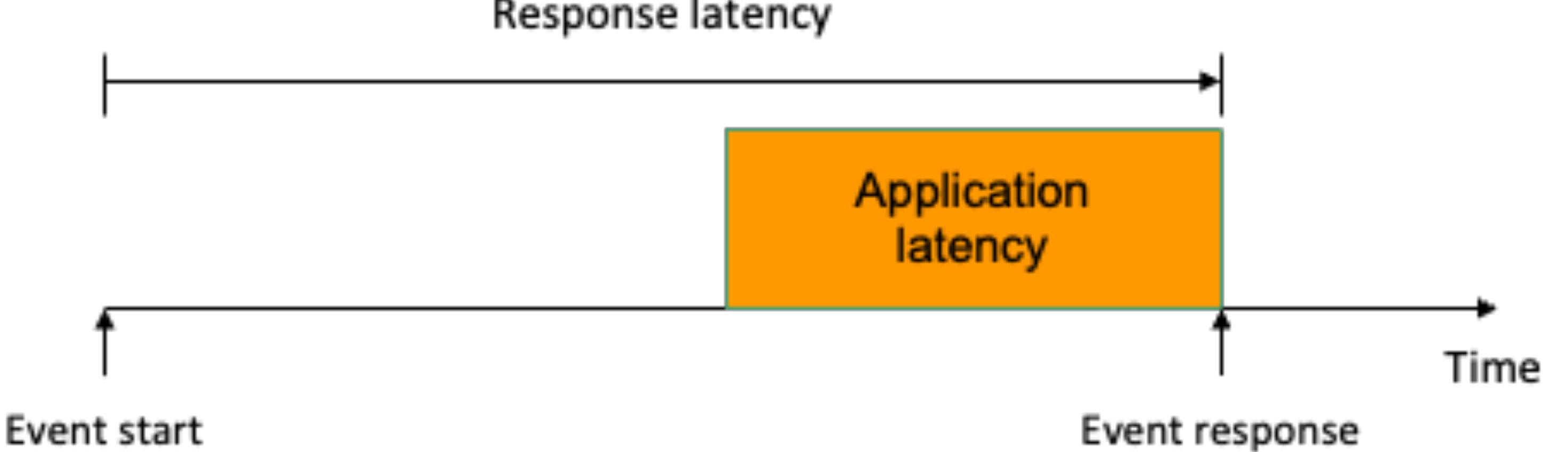
Fuentes de latencia

- La pila de software moderna es muy profunda
- Cada capa puede inducir latencia
- Es necesario garantizar una latencia máxima limitada de todas las capas
 - Tarea difícil: la mayoría del software está diseñado para una latencia promedio, no para una latencia máxima



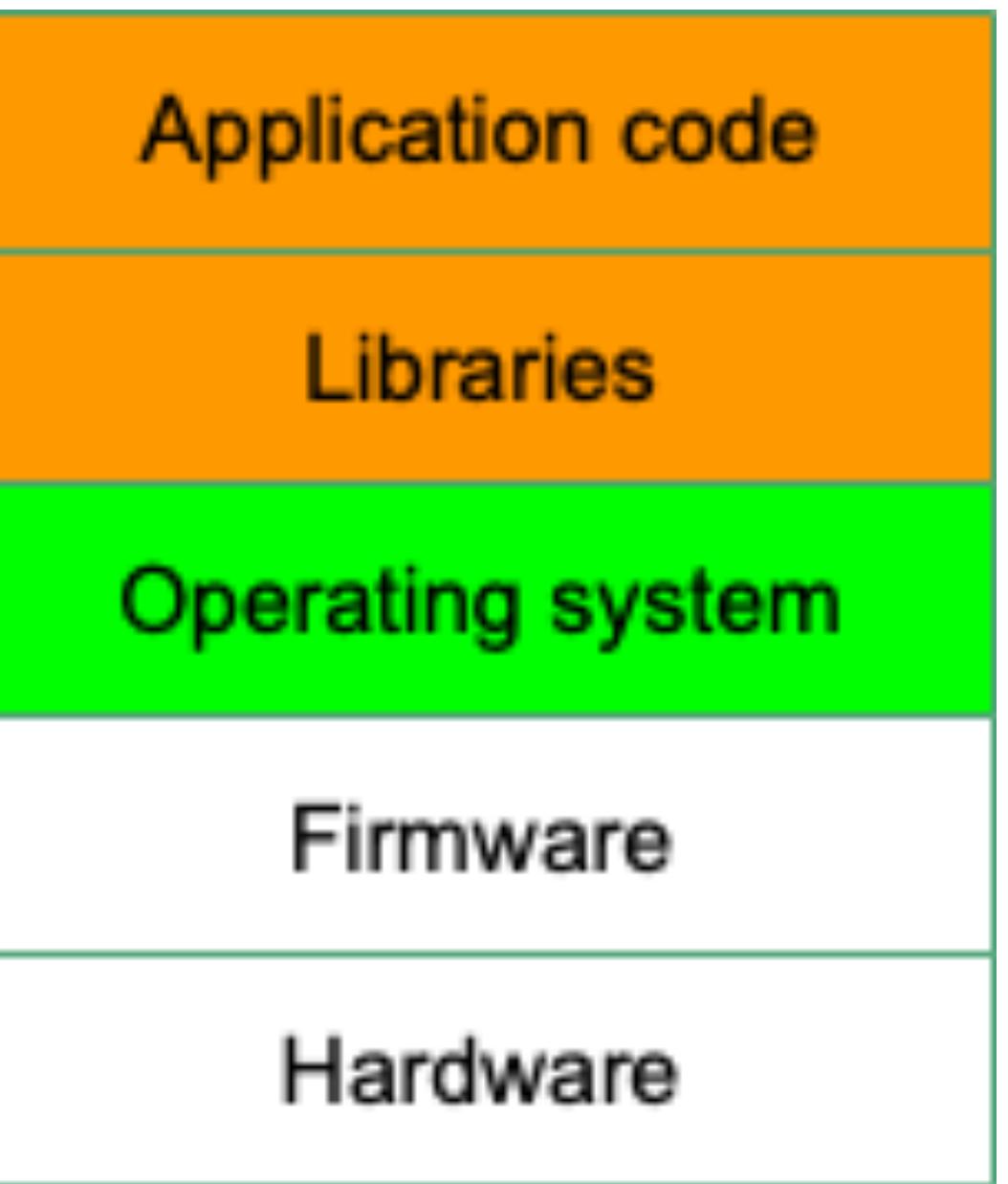
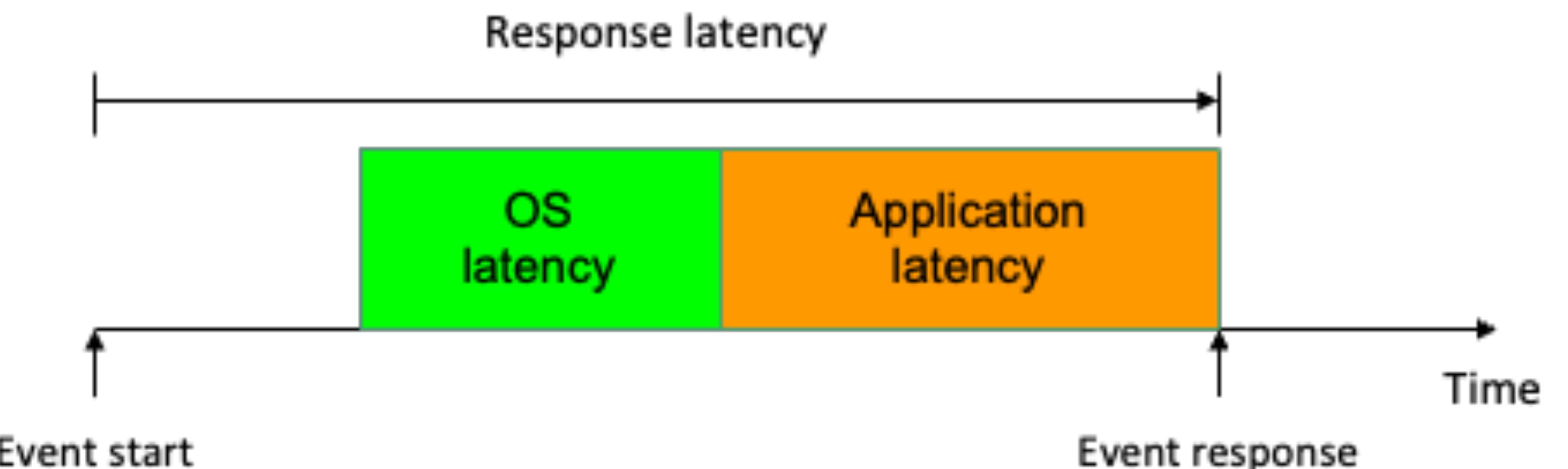
Fuentes de latencia

- Latencia de la aplicación
 - Bibliotecas + código de la aplicación
 - Esto se tratará con más profundidad más adelante



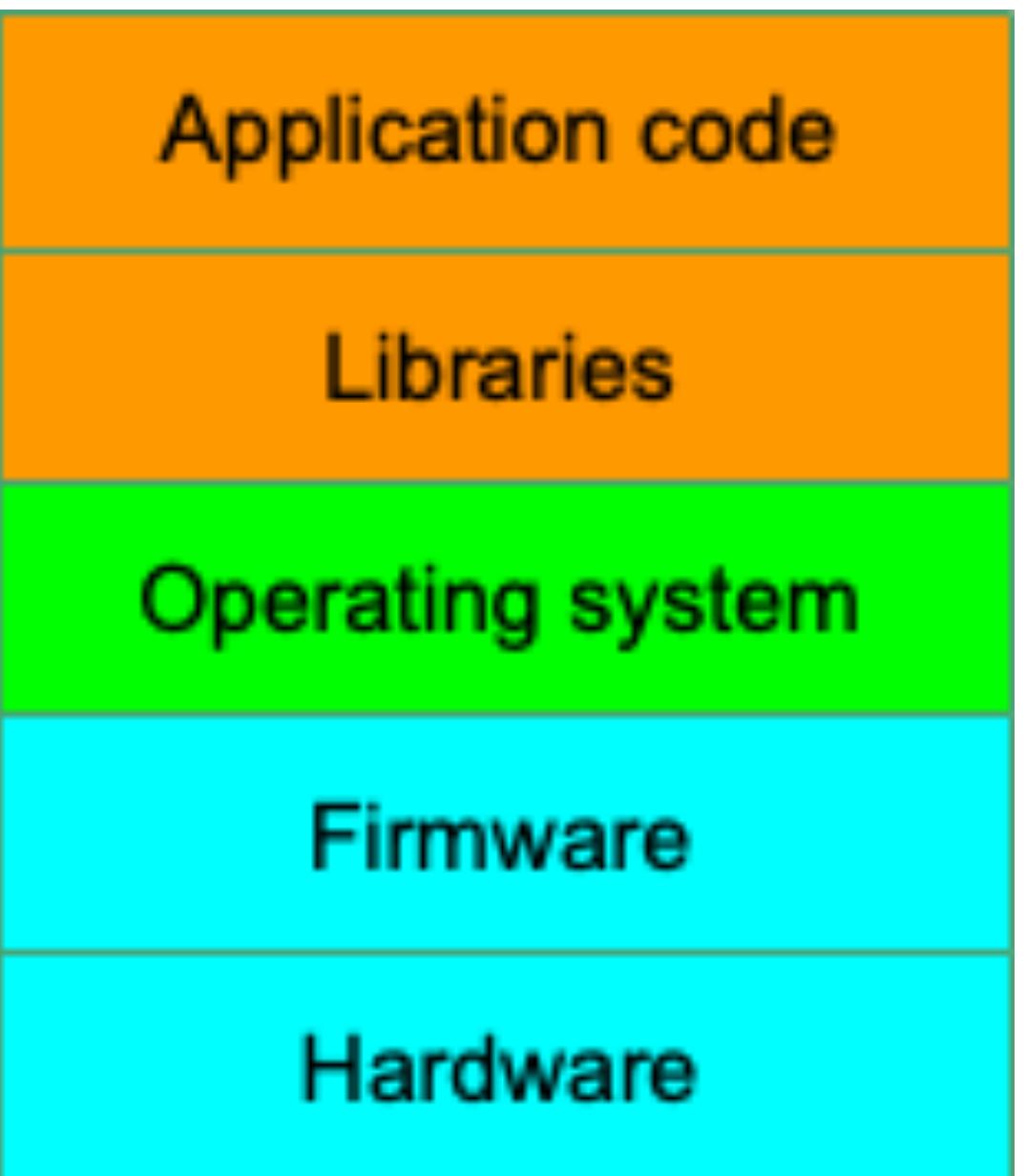
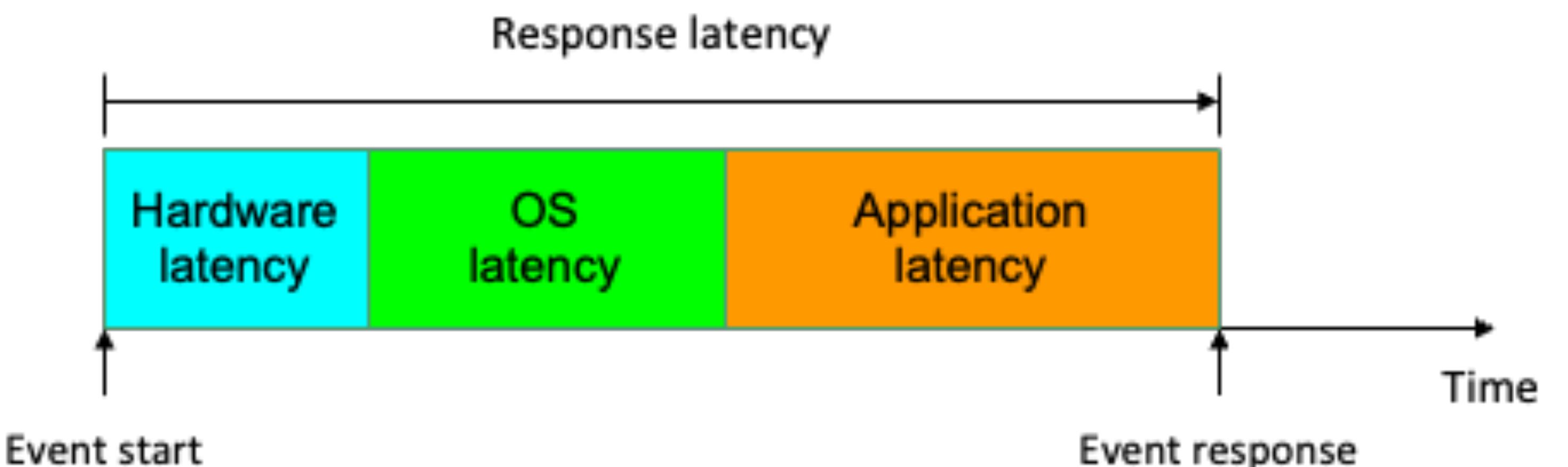
Fuentes de latencia

- Latencia de la aplicación
 - Bibliotecas + código de la aplicación
 - Esto se tratará con más profundidad más adelante
- Latencia del sistema operativo
 - Principalmente debido a la planificación del sistema operativo



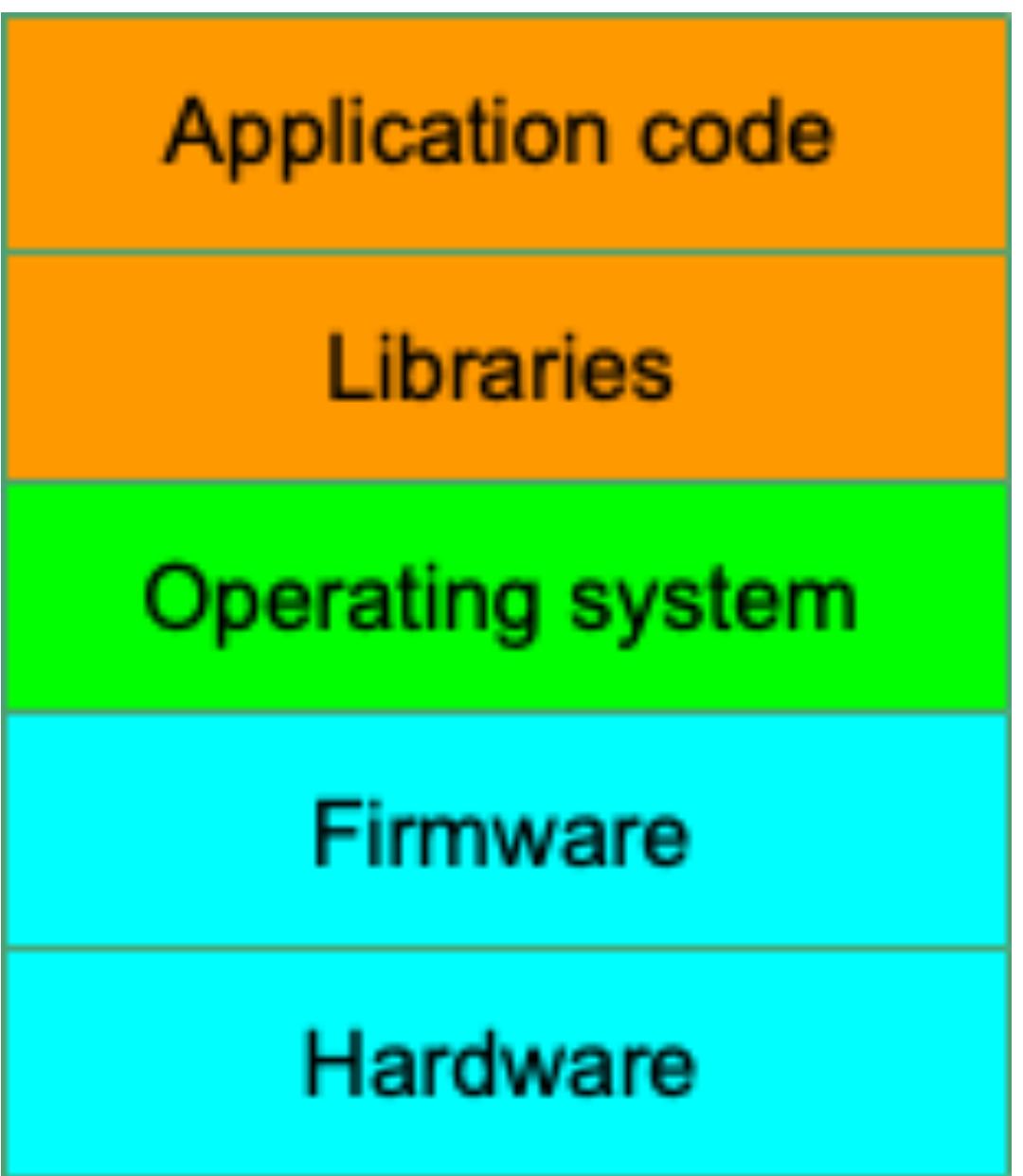
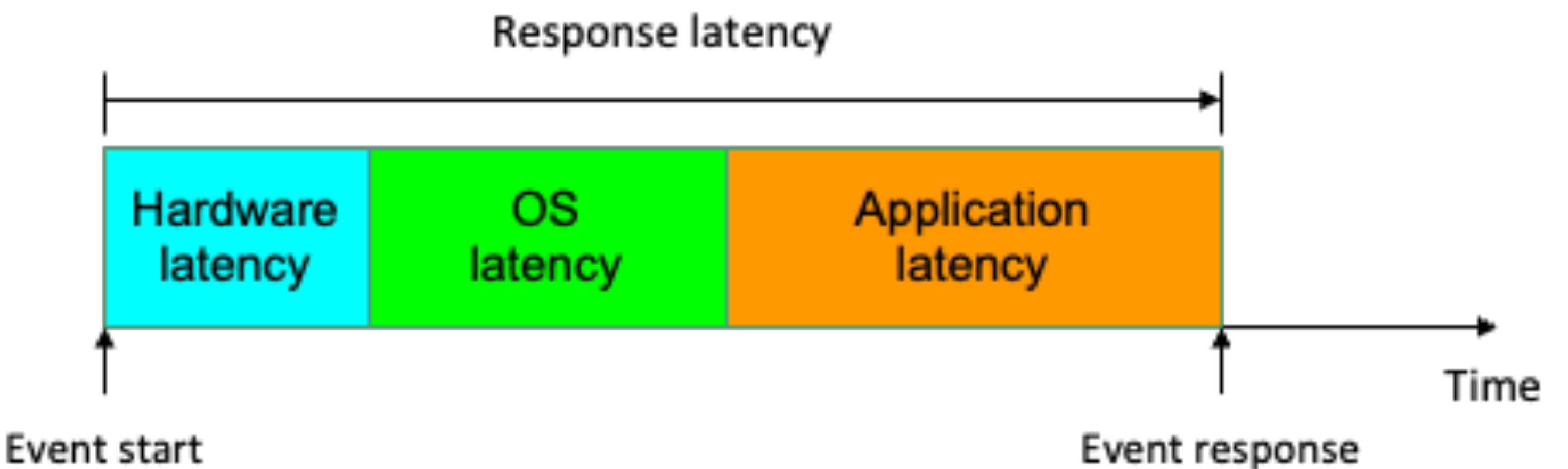
Fuentes de latencia

- Latencia de la aplicación
 - Bibliotecas + código de la aplicación
 - Esto se tratará con más profundidad más adelante
- Latencia del sistema operativo
 - Principalmente debido a la planificación del sistema operativo
- Latencia del hardware
 - Hardware + firmware



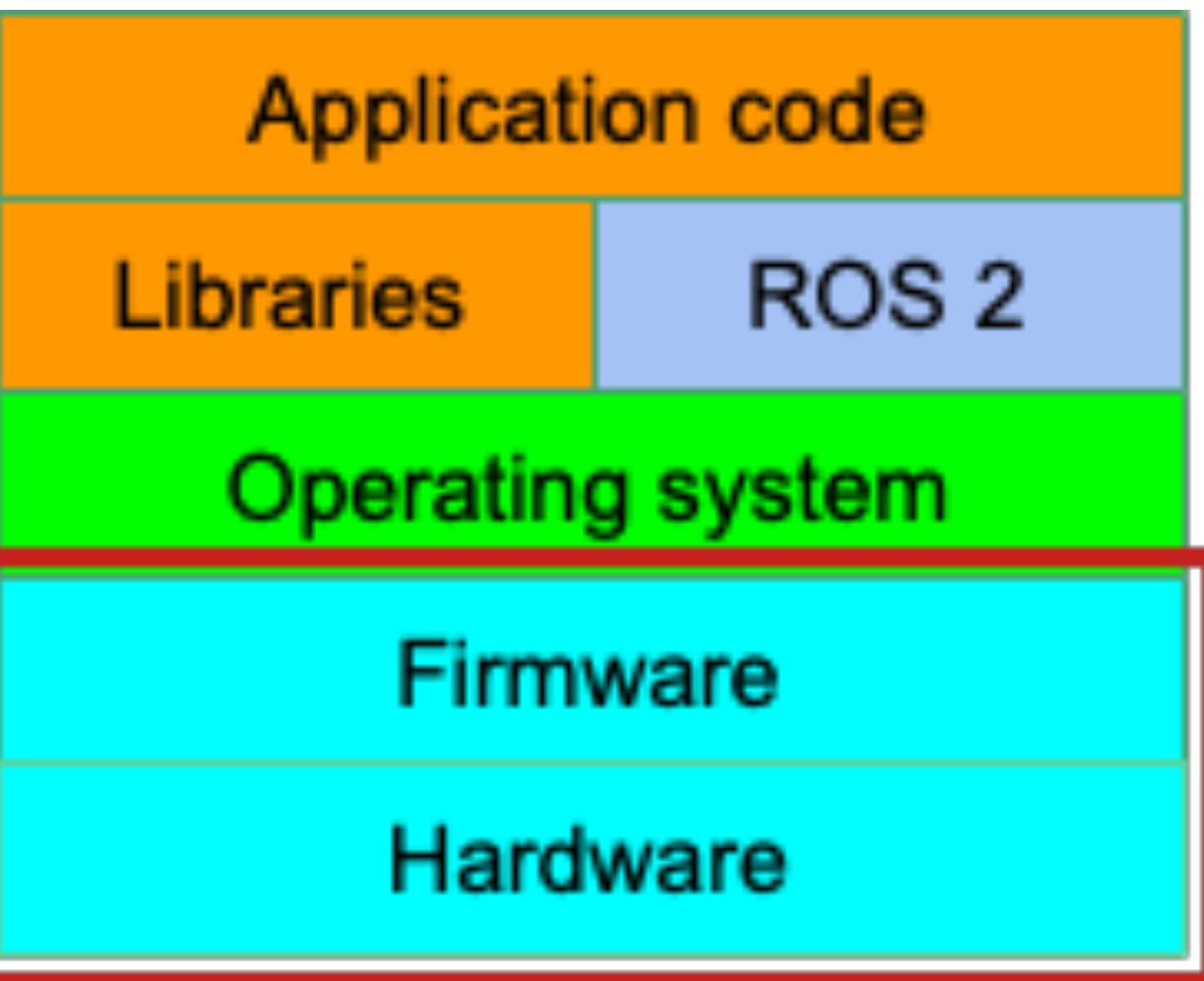
Fuentes de latencia

- La latencia en el peor de los casos de cada fuente se puede sumar para inferir la latencia de respuesta en el peor de los casos.
 - Suponga que la latencia causada por diferentes componentes es independiente



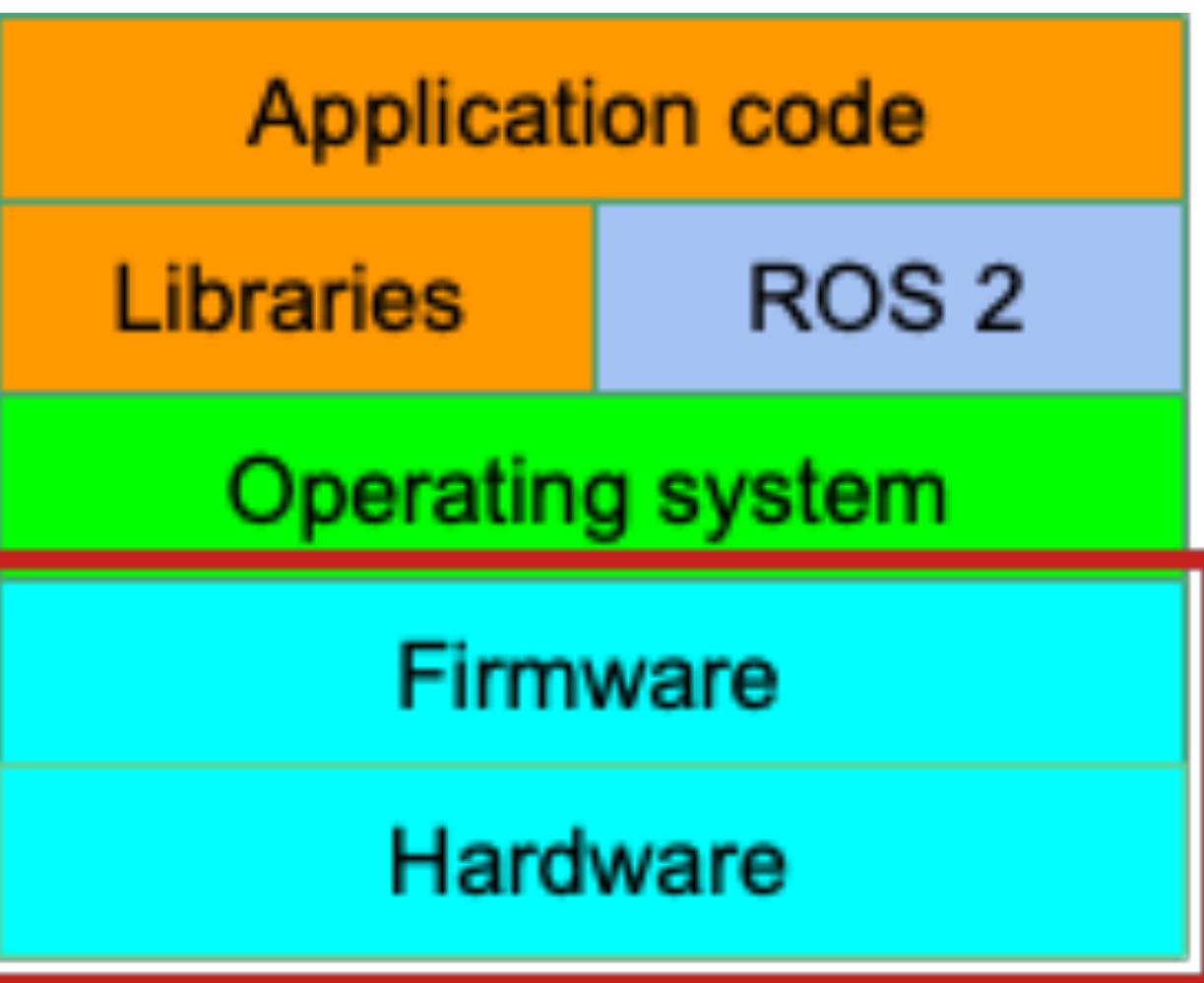
Latencia del hardware

- El hardware moderno es muy complejo
- El sistema operativo no controla completamente el hardware
 - Interrupciones de administración del sistema (SMI)
 - Multiprocesamiento simultáneo
- Rendimiento de hardware no uniforme a lo largo del tiempo
 - Escalado de frecuencia dinámico basado en la potencia y la temperatura
 - Error de caché, error de predicción de bifurcación, etc.
- Latencia debido a la física
 - Latencia de acceso al disco duro
 - Latencia de la memoria
 - Latencia de comunicación serial



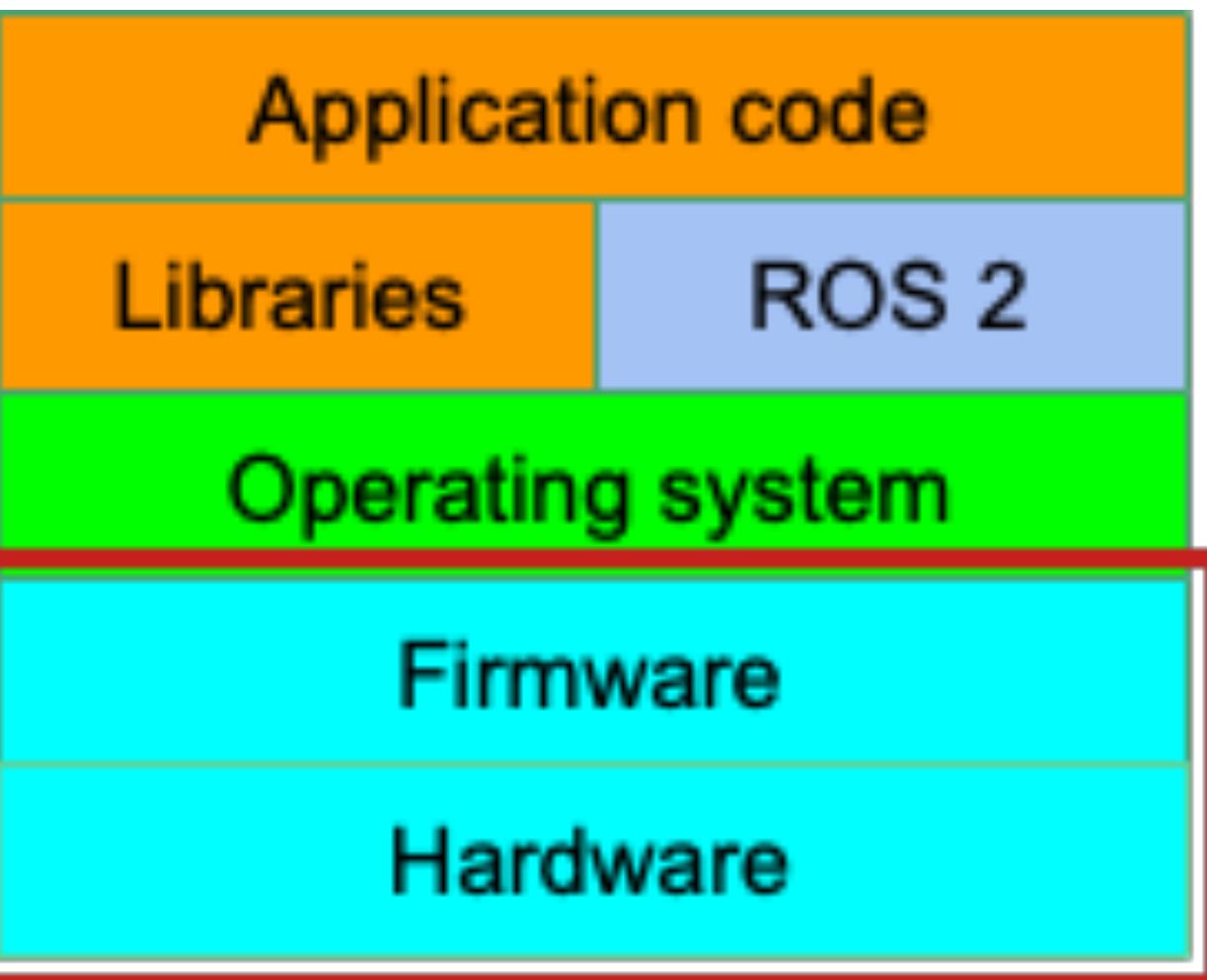
Latencia del hardware

- El hardware moderno es muy complejo
- El sistema operativo no controla completamente el hardware
 - Interrupciones de administración del sistema (SMI)
 - Multiprocesamiento simultáneo
- Rendimiento de hardware no uniforme a lo largo del tiempo
 - Escalado de frecuencia dinámico basado en la potencia y la temperatura
 - Error de caché, error de predicción de bifurcación, etc.
- Latencia debido a la física
 - Latencia de acceso al disco duro
 - Latencia de la memoria
 - Latencia de comunicación serial



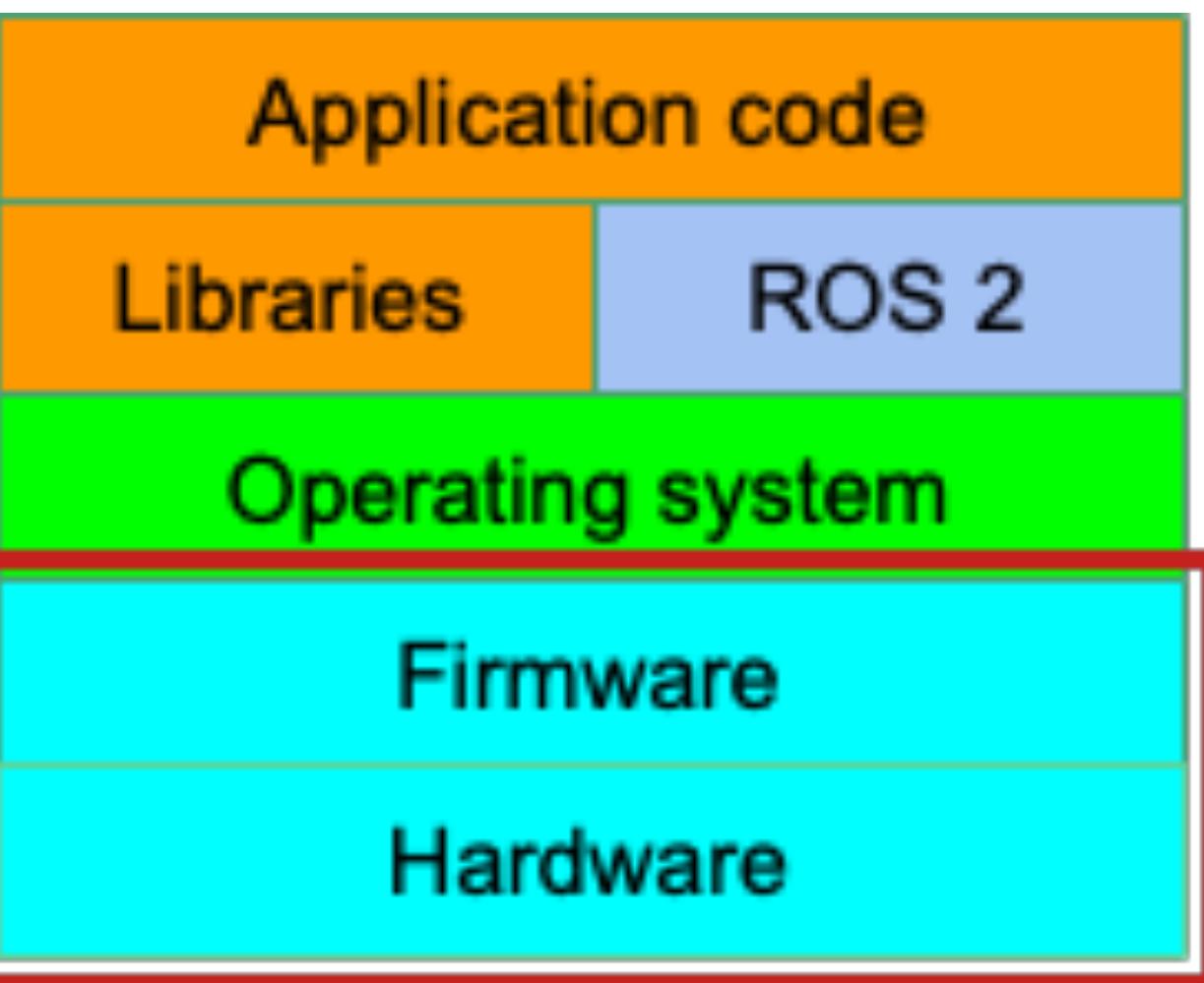
Latencia del hardware

- El hardware moderno es muy complejo
- El sistema operativo no controla completamente el hardware
 - Interrupciones de administración del sistema (SMI)
 - Multiprocesamiento simultáneo
- Rendimiento de hardware no uniforme a lo largo del tiempo
 - Escalado de frecuencia dinámico basado en la potencia y la temperatura
 - Error de caché, error de predicción de bifurcación, etc.
- Latencia debido a la física
 - Latencia de acceso al disco duro
 - Latencia de la memoria
 - Latencia de comunicación serial



Latencia del hardware

- El hardware moderno es muy complejo
- El sistema operativo no controla completamente el hardware
 - Interrupciones de administración del sistema (SMI)
 - Multiprocesamiento simultáneo
- Rendimiento de hardware no uniforme a lo largo del tiempo
 - Escalado de frecuencia dinámico basado en la potencia y la temperatura
 - Error de caché, error de predicción de bifurcación, etc.
- Latencia debido a la física
 - Latencia de acceso al disco duro
 - Latencia de la memoria
 - Latencia de comunicación serial



Caso de estudio sobre latencia de hardware

SMI-induced latency	
Hardware	Off-the-shelf workstation
Symptom	Constant 0.4 - 1.2ms stutters
Root cause	System management interrupts (likely)
Diagnosed with	cyclictest, hwlatdetect
Solution	A new computer

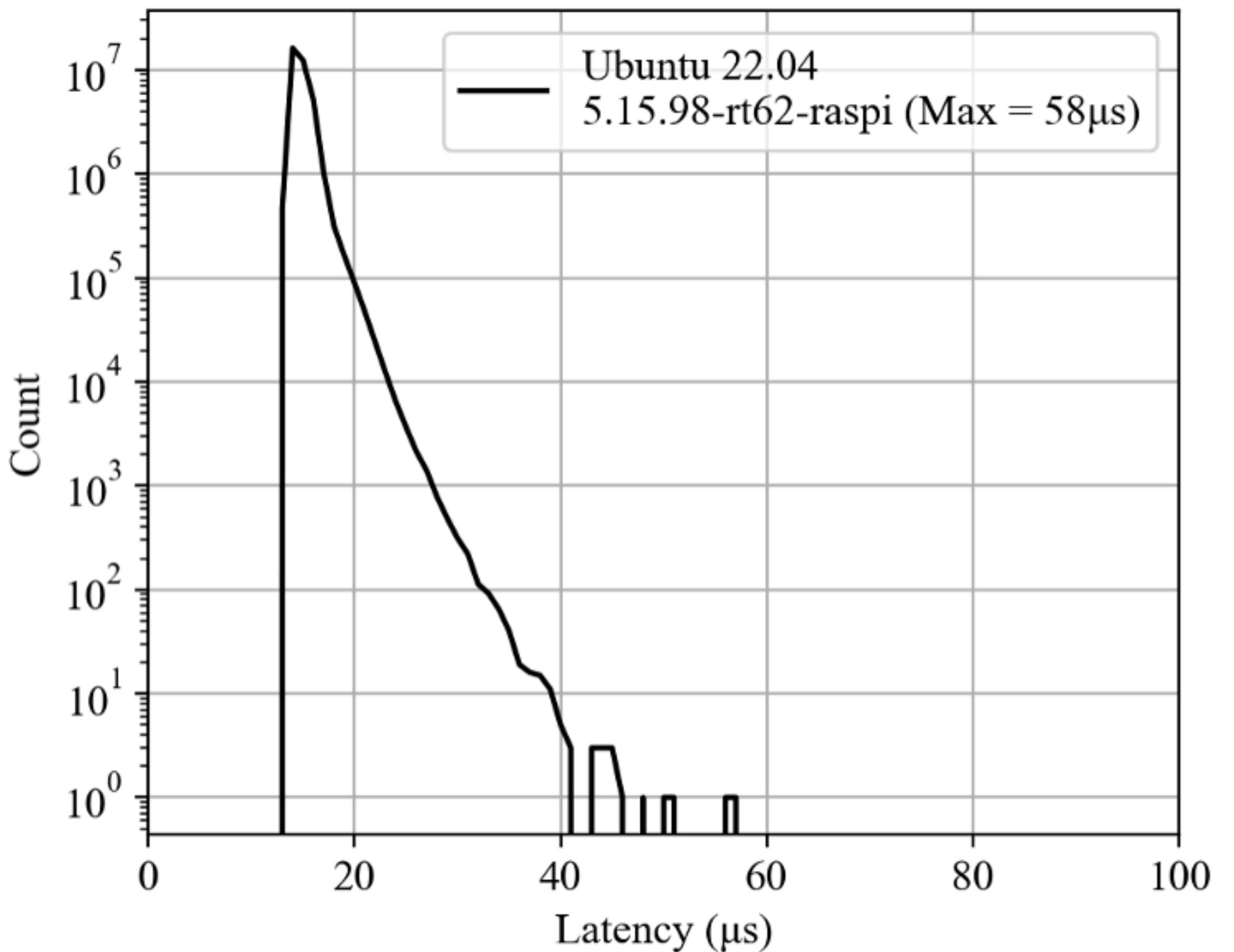
Caso de estudio sobre latencia de hardware

	SMI-induced latency	Peripheral-induced latency
Hardware	Off-the-shelf workstation	Serial-based motor controller
Symptom	Constant 0.4 - 1.2ms stutters	1000 Hz control loop slows down to 100 Hz
Root cause	System management interrupts (likely)	Serial communication and data size limits each command to be ~2ms. Needs 5 commands per loop iteration
Diagnosed with	cyclictest, hwlatdetect	Application tracing
Solution	A new computer	A new motor controller



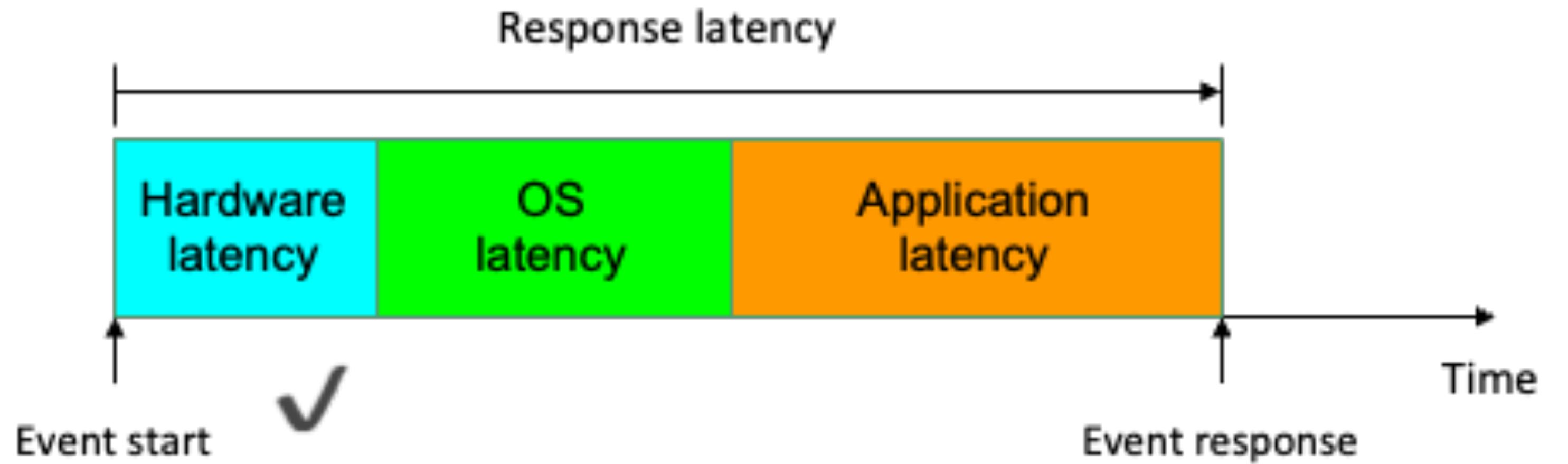
Raspberry Pi 4 como plataforma en Tiempo Real

- Raspberry Pi 4 no parece tener una latencia de hardware excesiva
- máxima (latencia del SO + hardware) $\approx 100 \mu\text{s}$
 - Implica máxima (latencia del hardware) $< 100 \mu\text{s}$



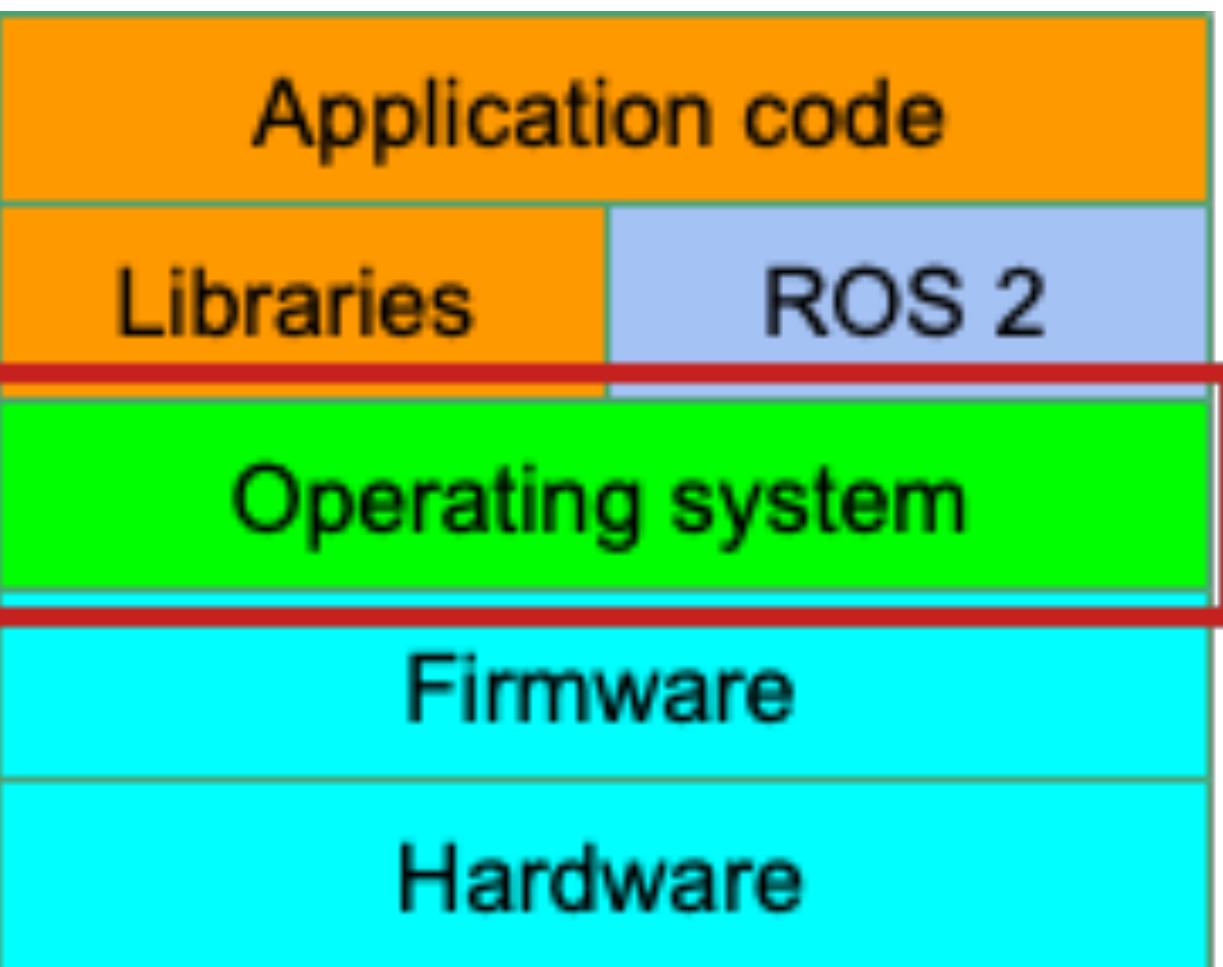
Latencia del hardware: resumen

- El hardware es complejo y puede generar latencia
- Compra el hardware adecuado
- Realiza pruebas rigurosas
- ¡Confía, pero verifique!



Latencia del sistema operativo

- Los sistemas operativos modernos son muy complejos
- Principalmente: latencia debido a la planificación
- Cada sistema operativo tiene diferentes perfiles de latencia
- El código del núcleo o del controlador puede desempeñar un papel según el sistema operativo



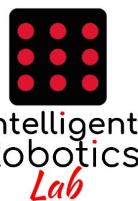
¿Qué es la planificación?

- Un núcleo de CPU ejecuta una instrucción a la vez*
- Tarea principal del sistema operativo:
 - Ejecutar múltiples tareas “simultáneamente” en una sola CPU
- Ejemplo:
 - Entrada del usuario mientras se ejecutan tareas de uso intensivo de recursos computacionales
 - Presionar E en el teclado mientras se codifica un video
- El planificador de la CPU es responsable de esto
- Conocido como conmutación de tareas



¿Qué es la planificación?

- Un núcleo de CPU ejecuta una instrucción a la vez*
- Tarea principal del sistema operativo:
 - Ejecutar múltiples tareas “simultáneamente” en una sola CPU
- Ejemplo:
 - Entrada del usuario mientras se ejecutan tareas de uso intensivo de recursos computacionales
 - Presionar E en el teclado mientras se codifica un video
- El planificador de la CPU es responsable de esto
- Conocido como conmutación de tareas



¿Qué es la planificación?

- Un núcleo de CPU ejecuta una instrucción a la vez*
- Tarea principal del sistema operativo:
 - Ejecutar múltiples tareas “simultáneamente” en una sola CPU
- Ejemplo:
 - Entrada del usuario mientras se ejecutan tareas de uso intensivo de recursos computacionales
 - Presionar E en el teclado mientras se codifica un video
- El planificador de la CPU es responsable de esto
- Conocido como conmutación de tareas

Intelligent
Robotics
Lab

¿Qué es la planificación?

- Un núcleo de CPU ejecuta una instrucción a la vez*
- Tarea principal del sistema operativo:
 - Ejecutar múltiples tareas “simultáneamente” en una sola CPU
- Ejemplo:
 - Entrada del usuario mientras se ejecutan tareas de uso intensivo de recursos computacionales
 - Presionar E en el teclado mientras se codifica un video
 - El planificador de la CPU es responsable de esto
 - Conocido como conmutación de tareas



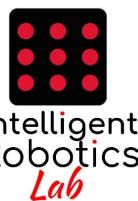
Latencia del planificador (a alto nivel)

- El cambio de tareas (cambio de contexto) es costoso
 - Guardar y restaurar valores de registros en la RAM y desde ella
- Procesamiento de interrupciones
 - Algunas interrupciones deben ejecutarse o el sistema se bloqueará
 - Esto puede llevar tiempo
- Algunos códigos del kernel o del controlador no se pueden interrumpir
 - Depende del sistema operativo y los controladores
- $\max(\text{latencia de respuesta}) \geq \max(\text{latencia del planificador})$



Latencia del planificador (a alto nivel)

- El cambio de tareas (cambio de contexto) es costoso
 - Guardar y restaurar valores de registros en la RAM y desde ella
- Procesamiento de interrupciones
 - Algunas interrupciones deben ejecutarse o el sistema se bloqueará
 - Esto puede llevar tiempo
- Algunos códigos del kernel o del controlador no se pueden interrumpir
 - Depende del sistema operativo y los controladores
- $\max(\text{latencia de respuesta}) \geq \max(\text{latencia del planificador})$



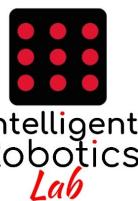
Latencia del planificador (a alto nivel)

- El cambio de tareas (cambio de contexto) es costoso
 - Guardar y restaurar valores de registros en la RAM y desde ella
- Procesamiento de interrupciones
 - Algunas interrupciones deben ejecutarse o el sistema se bloqueará
 - Esto puede llevar tiempo
- Algunos códigos del kernel o del controlador no se pueden interrumpir
 - Depende del sistema operativo y los controladores
- $\max(\text{latencia de respuesta}) \geq \max(\text{latencia del planificador})$

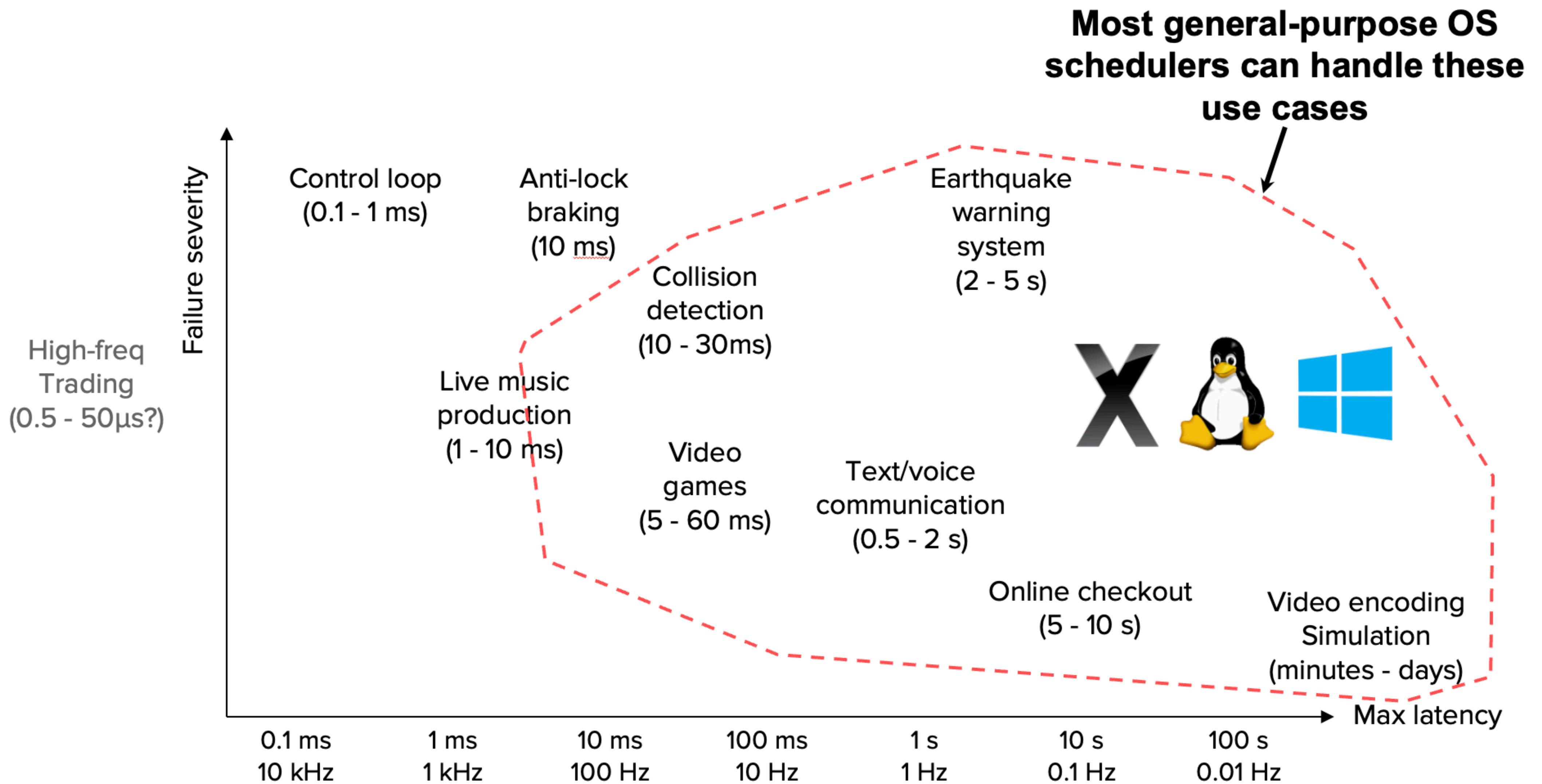


Latencia del planificador (a alto nivel)

- El cambio de tareas (cambio de contexto) es costoso
 - Guardar y restaurar valores de registros en la RAM y desde ella
- Procesamiento de interrupciones
 - Algunas interrupciones deben ejecutarse o el sistema se bloqueará
 - Esto puede llevar tiempo
- Algunos códigos del kernel o del controlador no se pueden interrumpir
 - Depende del sistema operativo y los controladores
- **$\max(\text{latencia de respuesta}) \geq \max(\text{latencia del planificador})$**



Ejemplos de sistemas en Tiempo Real



Cómo lidiar con la latencia de planificación con diferentes sistemas operativos

Multi-CPU:

NonRT Linux +
microcontroller

Hypervisor:

Xenomai
RTAI

Best-effort RT:

Windows
OS X
Linux w/o PREEMPT_RT

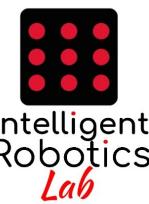
Real-time OS:

NuttX, Zephyr,
QNX, VxWorks,
etc

PREEMPT_RT:

Linux with
PREEMPT_RT
patch

Real-time
“softness”?



Cómo lidiar con la latencia de programación con diferentes sistemas operativos

Multi-CPU:

NonRT Linux +
microcontroller

Hypervisor:

Xenomai
RTAI

Best-effort RT:

Windows
OS X
Linux w/o PREEMPT_RT

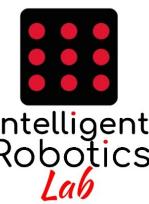
Real-time OS:

NuttX, Zephyr,
QNX, VxWorks,
etc

PREEMPT_RT:

Linux with
PREEMPT_RT
patch

Real-time
“softness”?



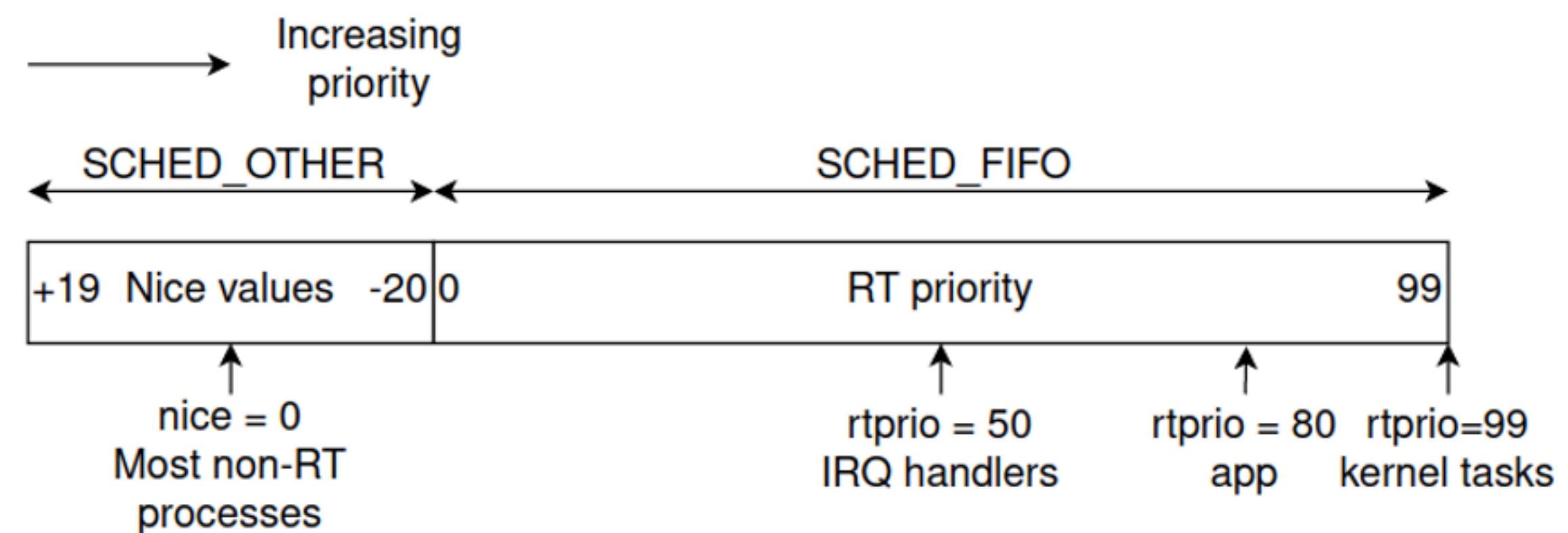
Cómo lidiar con la latencia del sistema operativo con Linux: Scheduler

- La planificación es un problema NP-completo
- Planificador predeterminado de Linux: optimiza para la mayoría de los casos, pero no en Tiempo Real
- Linux incluye varios planificadores
- Seleccionable en tiempo de ejecución para cada hilo

Scheduler	Policy name	Real-time?
Completely-fair scheduler (CFS, default)	SCHED_OTHER	👎
First-in-first-out scheduler	SCHED_FIFO	👍
Earliest-deadline-first (EDF) scheduler	SCHED_DEADLINE	👍

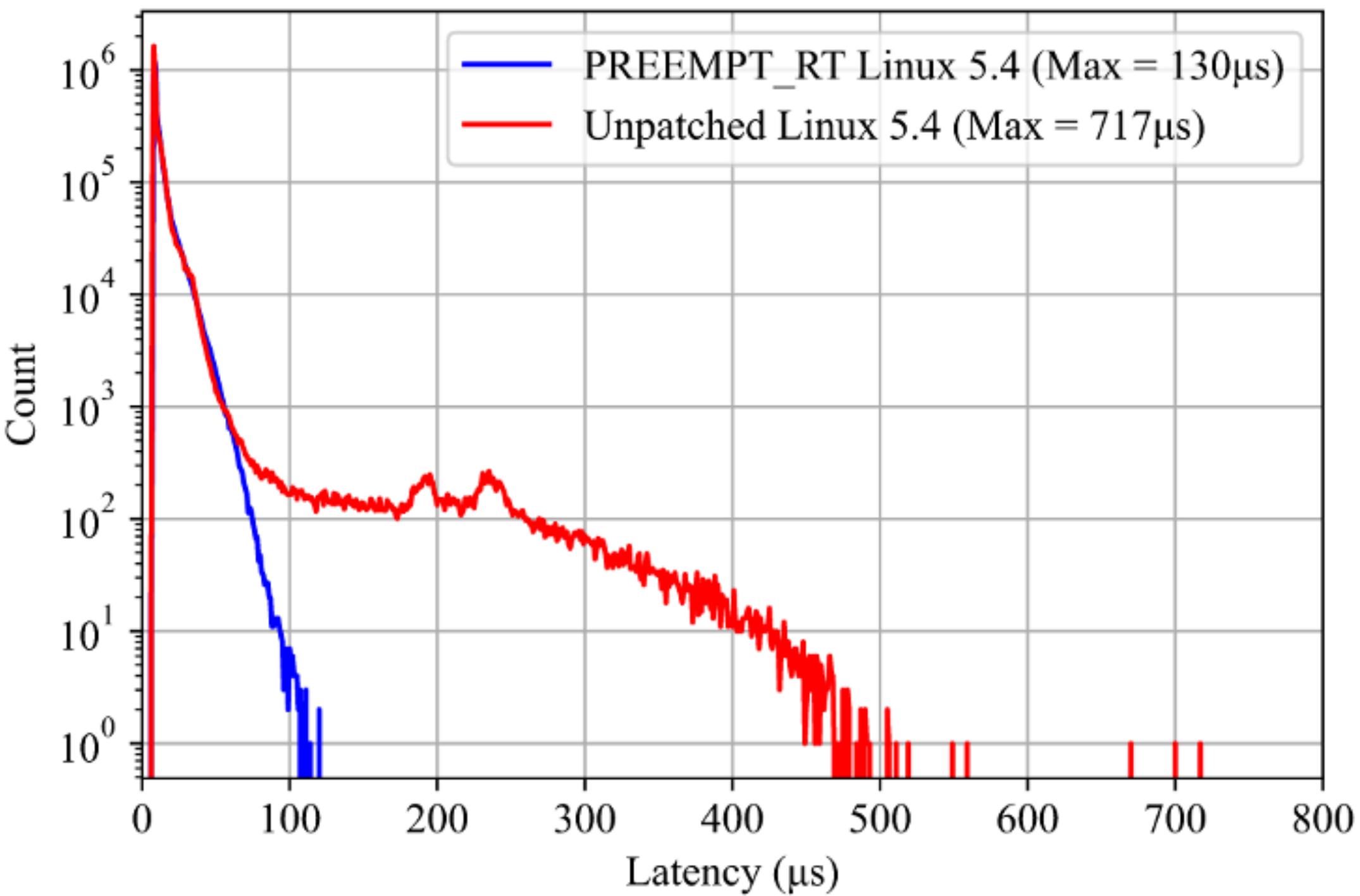
Cómo lidiar con la latencia del sistema operativo con Linux: Scheduler

- Se debe seleccionar y configurar el planificador en Tiempo Real
- Solicitud de planificador FIFO:
 - `pthread_attr_setschedpolicy(&attr, SCHED_FIFO);`
- Establecimiento de prioridad:
 - `struct sched_param param; param.sched_priority = 80;`
- Todas las tareas SCHED_FIFO tienen mayor prioridad que las tareas SCHED_OTHER
- Un nivel de prioridad de 80 es una buena primera opción



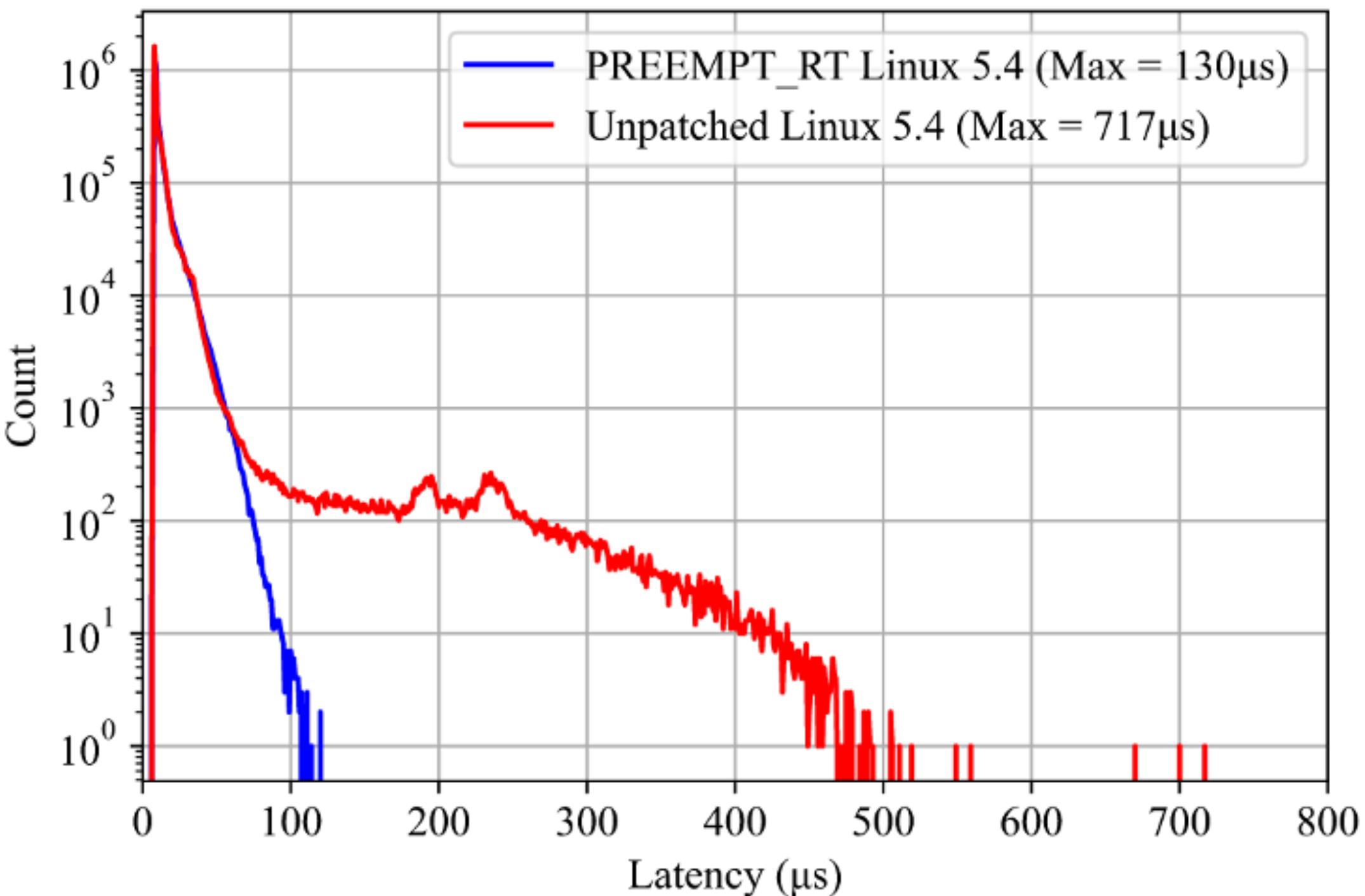
Cómo lidiar con la latencia del sistema operativo con Linux: PREEMPT_RT

- Algunos códigos del kernel no se pueden interrumpir de forma predeterminada
 - Introduce latencia
- PREEMPT_RT soluciona la mayoría de estos problemas
 - Reduce la latencia
 - Detalles fuera del alcance del Workshop
- PREEMPT_RT es un conjunto de parches para Linux
 - Se pretende fusionarlo con la línea principal en el futuro
- Máxima (latencia del SO) $\approx 100 \mu\text{s}$
 - Se puede utilizar para sistemas en Tiempo Real de 1 a 2 kHz
- No probado/certificado



Cómo lidiar con la latencia del sistema operativo con Linux: PREEMPT_RT

- SCHED_FIFO también existe en el kernel normal
 - La latencia en el peor de los casos es mayor que la de PREEMPT_RT
 - La latencia en el peor de los casos con SCHED_OTHER es incluso mayor
 - Está mejorando a medida que se fusiona más código PREEMPT_RT con la línea principal
 - Se puede utilizar para los ejercicios de hoy



Cuándo y cuándo no utilizar Linux y PREEMPT_RT

- Utilice Linux + PREEMPT_RT cuando:
 - \leq 1000 - 2000 Hz
 - No requiera certificación
 - Puede tolerar incumplimientos de plazos poco frecuentes o hipotéticos
- No utilice Linux + PREEMPT_RT cuando:
 - \geq 10000 Hz
 - Requiera certificación por razones regulatorias



¿Quién utiliza PREEMPT_RT?

- [National Instruments Linux Real-Time](#)
- [Software de vuelo de SpaceX](#)
- [Máquinas CNC](#)
- Vehículos autónomos, brazos robóticos
- Muchos otros, en su mayoría no publicitados



Cómo lidiar con la latencia del sistema operativo con Linux: Tuning

- Las configuraciones predeterminadas del kernel y del SO no son necesariamente buenas para el Tiempo Real
 - Deshabilitar la limitación de RT
 - Deshabilitar servicios innecesarios o que dañan la latencia (multipathd en el servidor Ubuntu...)
 - Ajustar security.conf para privilegios
 - isolcpus, irqaffinity, rcu_nocbs
 - nohz, nohzfull



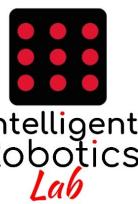
Cómo lidiar con la latencia del sistema operativo: resumen

- Latencia del planificador:
 - Solicitud de planificador RT y establecimiento de prioridad
 - `SCED_FIFO`
 - `sched_priority = 80`
- Linux con `PREEMPT_RT`
 - Adecuado para cargas de trabajo de 1 a 2 kHz
 - Se necesita parchear el kernel
 - Requiere ajustar varias configuraciones del kernel y del SO



Validación de la latencia del hardware y del sistema operativo

- ¿Cómo se puede determinar la latencia máxima?
 - ¿Matemáticamente? ¿Empíricamente?
 - ¿En implementaciones internas de CPU, firmware y kernel?
 - ¿Dónde termina la validación?
- Empíricamente en Linux
 - hwlatdetect: mide la latencia inducida por SMI
 - cyclictest: mide la latencia máxima (hardware + SO)
- Ejecuta una prueba de estrés de la CPU mientras mide la latencia del SO
 - `stress-ng -c $(nproc)`
 - Ejecutaremos esto en algunos de los ejercicios
- Ejecuta la prueba durante horas/días/semanas hasta alcanzar el nivel de confianza deseado



Resumen

- Los sistemas en Tiempo Real se definen por su latencia máxima y la gravedad de las fallas
- La latencia puede ser introducida por el hardware, el sistema operativo y la aplicación
- La latencia del hardware y del sistema operativo se puede caracterizar y minimizar (después del ajuste)
- Linux + PREEMPT_RT es una plataforma atractiva para crear aplicaciones en Tiempo Real
 - Aplicaciones de hasta 1-2 kHz sin requisitos de verificación formal
 - Muchas empresas implementan PREEMPT_RT en producción
- Confía, pero verifique con pruebas de larga duración



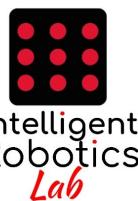
Resumen

- Los sistemas en Tiempo Real se definen por su latencia máxima y la gravedad de las fallas
- La latencia puede ser introducida por el hardware, el sistema operativo y la aplicación
- La latencia del hardware y del sistema operativo se puede caracterizar y minimizar (después del ajuste)
- Linux + PREEMPT_RT es una plataforma atractiva para crear aplicaciones en Tiempo Real
 - Aplicaciones de hasta 1-2 kHz sin requisitos de verificación formal
 - Muchas empresas implementan PREEMPT_RT en producción
 - Confía, pero verifique con pruebas de larga duración



Resumen

- Los sistemas en Tiempo Real se definen por su latencia máxima y la gravedad de las fallas
- La latencia puede ser introducida por el hardware, el sistema operativo y la aplicación
- La latencia del hardware y del sistema operativo se puede caracterizar y minimizar (después del ajuste)
- Linux + PREEMPT_RT es una plataforma atractiva para crear aplicaciones en Tiempo Real
 - Aplicaciones de hasta 1-2 kHz sin requisitos de verificación formal
 - Muchas empresas implementan PREEMPT_RT en producción
 - Confía, pero verifique con pruebas de larga duración



Resumen

- Los sistemas en Tiempo Real se definen por su latencia máxima y la gravedad de las fallas
- La latencia puede ser introducida por el hardware, el sistema operativo y la aplicación
- La latencia del hardware y del sistema operativo se puede caracterizar y minimizar (después del ajuste)
- Linux + PREEMPT_RT es una plataforma atractiva para crear aplicaciones en Tiempo Real
 - Aplicaciones de hasta 1-2 kHz sin requisitos de verificación formal
 - Muchas empresas implementan PREEMPT_RT en producción
- Confía, pero verifique con pruebas de larga duración



Resumen

- Los sistemas en Tiempo Real se definen por su latencia máxima y la gravedad de las fallas
- La latencia puede ser introducida por el hardware, el sistema operativo y la aplicación
- La latencia del hardware y del sistema operativo se puede caracterizar y minimizar (después del ajuste)
- Linux + PREEMPT_RT es una plataforma atractiva para crear aplicaciones en Tiempo Real
 - Aplicaciones de hasta 1-2 kHz sin requisitos de verificación formal
 - Muchas empresas implementan PREEMPT_RT en producción
 - Confía, pero verifique con pruebas de larga duración



Recursos adicionales (para luego)

- [Who needs a Real-Time Operating System \(Not You!\) - Steven Rostedt](#)
- [Understanding a Real-Time System \(more than just a kernel\) - Steven Rostedt](#)
- [Challenges Using Linux as a Real-Time Operating System - Michael M. Madden](#)
- [The Real-Time Linux Kernel: A Survey on PREEMPT_RT - Reghenzani et al.](#)
- [Low Latency Tuning Guide - Erik Rigtorp](#)
- [Tuning a real-time kernel - Edoardo Barbieri](#)
- [Cyclictest documentation](#)
- [Real-time Linux documentation wiki](#)



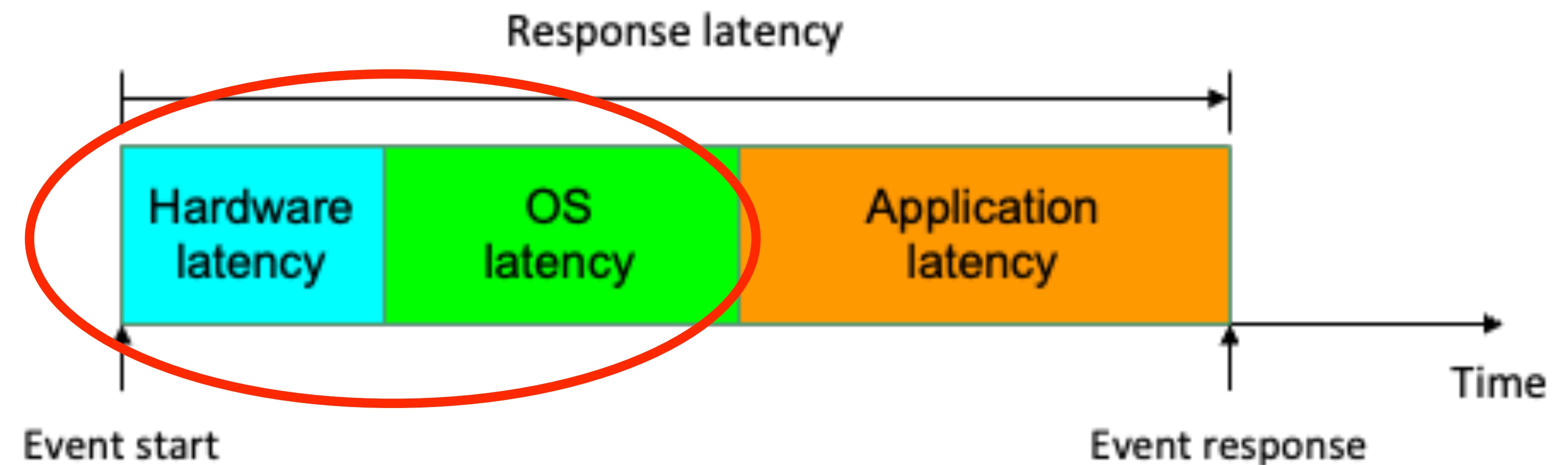
Ejercicio práctico 1:

Medición de la latencia del
hardware y del sistema operativo



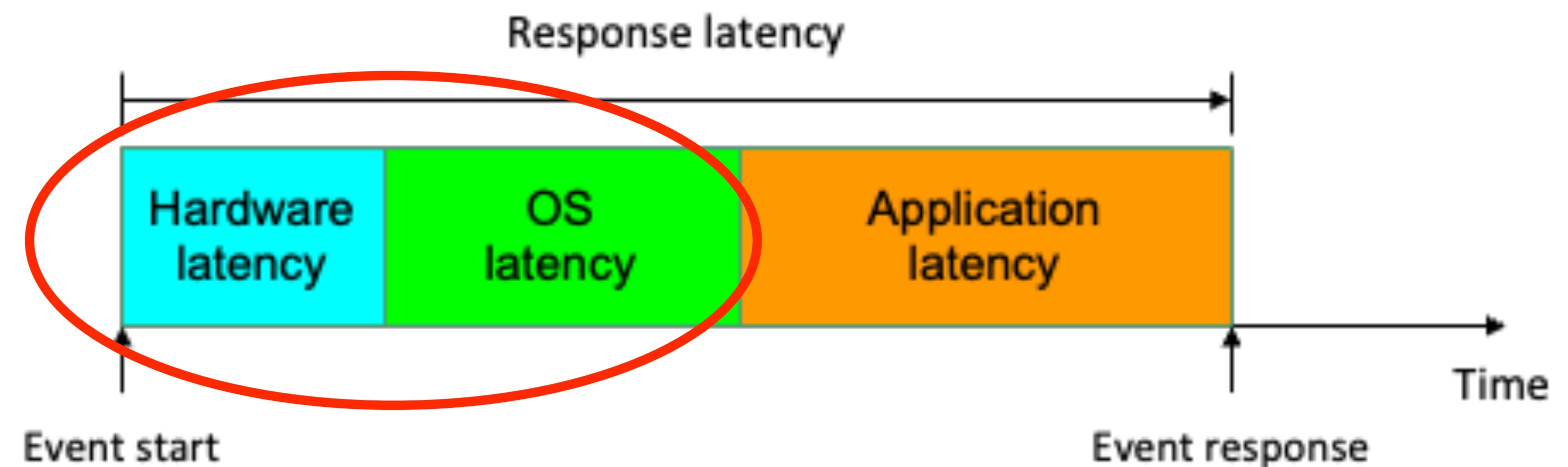
Ejercicio 1: medición de la latencia de activación

- Objetivo: medir la latencia de activación con y sin planificación en Tiempo Real
 - Latencia de activación = latencia del hardware + latencia del SO
 - La latencia del SO no se puede medir independientemente del hardware
- ¿Cómo? Repite lo siguiente en un bucle:
 - Dormir hasta el tiempo = t_1
 - Despertarse en tiempo = t_2
 - Latencia de hardware + SO = $t_2 - t_1$
 - Hazlo mientras el sistema esté bajo estrés



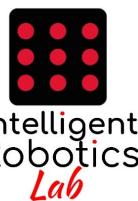
Ejercicio 1: medición de la latencia de activación

- Objetivo: medir la latencia de activación con y sin planificación en Tiempo Real
 - Latencia de activación = latencia del hardware + latencia del SO
 - La latencia del SO no se puede medir independientemente del hardware
- ¿Cómo? Repite lo siguiente en un bucle:
 - Dormir hasta el tiempo = t_1
 - Despertarse en tiempo = t_2
 - Latencia de hardware + SO = $t_2 - t_1$
 - Hazlo mientras el sistema esté bajo estrés



Ejercicio 1: medición de la latencia de activación

- Herramienta estándar: `cyclictest`
 - Mide la latencia del sistema operativo y del hardware para cada núcleo
- Hoy: comprobador de latencia dedicado escrito con [Cactus-RT](#)
- ¿Por qué?
 - Sistema de seguimiento y visualización más fácil de usar basado en [Perfetto](#)
 - El resto de los ejercicios del Workshop utilizan la misma configuración de seguimiento y visualización
 - Te da la oportunidad de experimentar cómo modificar y ejecutar código para el resto del Workshop
 - Instalado dentro del Docker
- Configuración:
 - 2 threads anclados en el núcleo 0 y el núcleo 1, cada uno funcionando a 1000 Hz
 - Si la latencia de activación es > 1000 us, se incumple el deadline



Cactus-RT: un framework para escribir aplicaciones C++ en Tiempo Real

- Fuerza una estructura de aplicación
- Se ocupa de las cuestiones específicas de RT por tí
 - mlockall
 - Configura planificadores en Tiempo Real
 - Configura el registro seguro en Tiempo Real
 - Configura el seguimiento seguro en Tiempo Real
- Puedes centrarse en escribir la aplicación
 - Solicita una tasa de bucle y completa una función **Loop**
- Diseñado para aplicaciones en Tiempo Real de 1000 Hz en Linux
- Código abierto: <https://github.com/cactusdynamics/cactus-rt>



Práctica !!!

<https://github.com/fmrico/roscon-es-2024-realtime-workshop>

Workflow en la computadora portátil (1-3 ya estaba en el README)

1. Sigue el README para iniciar el contenedor Docker
2. Terminal 1:
 - a. \$ docker/shell
 - b. \$ cd /code/exercise1
 - c. \$ colcon build
 - d. \$./code/stress.sh
3. Terminal 2:
 - a. \$ docker/shell
 - b. \$ cd /code/exercise1 && ./run.sh
4. Esto genera `exercise1.perfetto` en el directorio `exercise1`
5. Detén la prueba de estrés en la terminal 1
6. Ve a <http://localhost:3100>, haz clic en Abrir archivo de seguimiento y carga `exercise1.perfetto`
7. Haga clic en Latencia en la barra lateral izquierda
8. En la parte superior de la página, selecciona  Select a slice: RtThread0 ▾ Wakeup ▾ us ▾
9. Observe el `max_dur` en la salida JSON debajo del histograma.

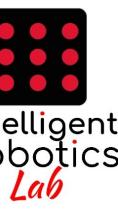
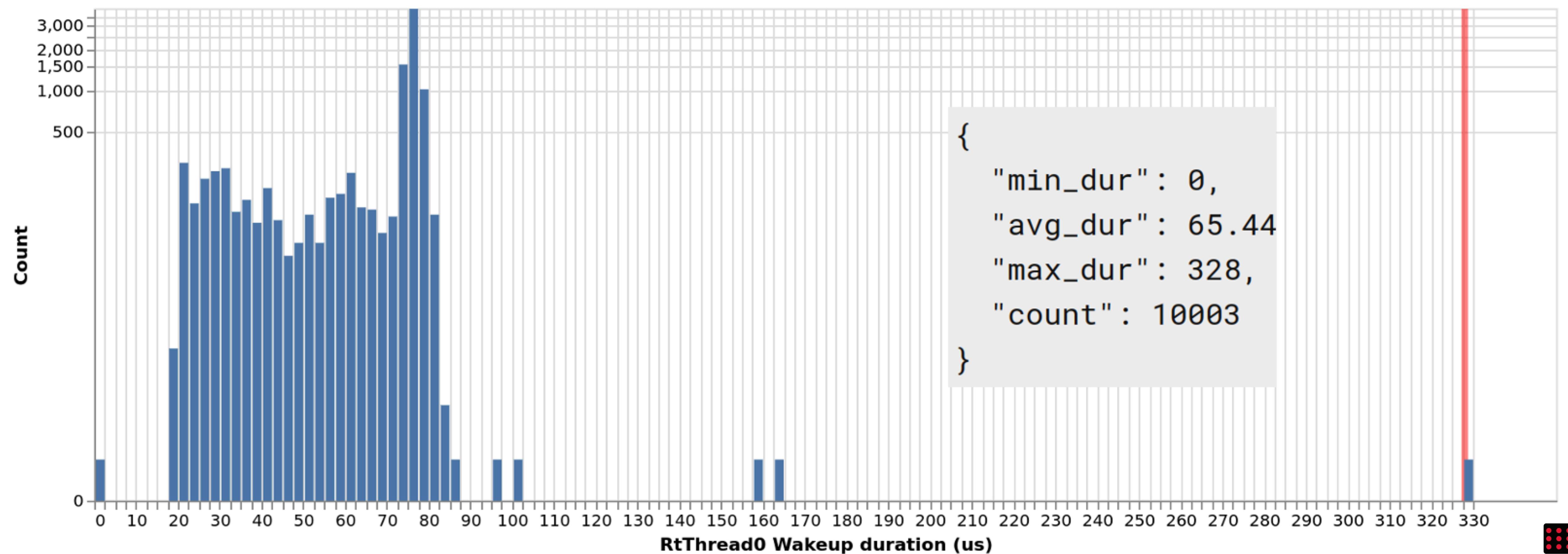
Esta es la latencia de activación máxima

Workflow en la Raspberry Pi

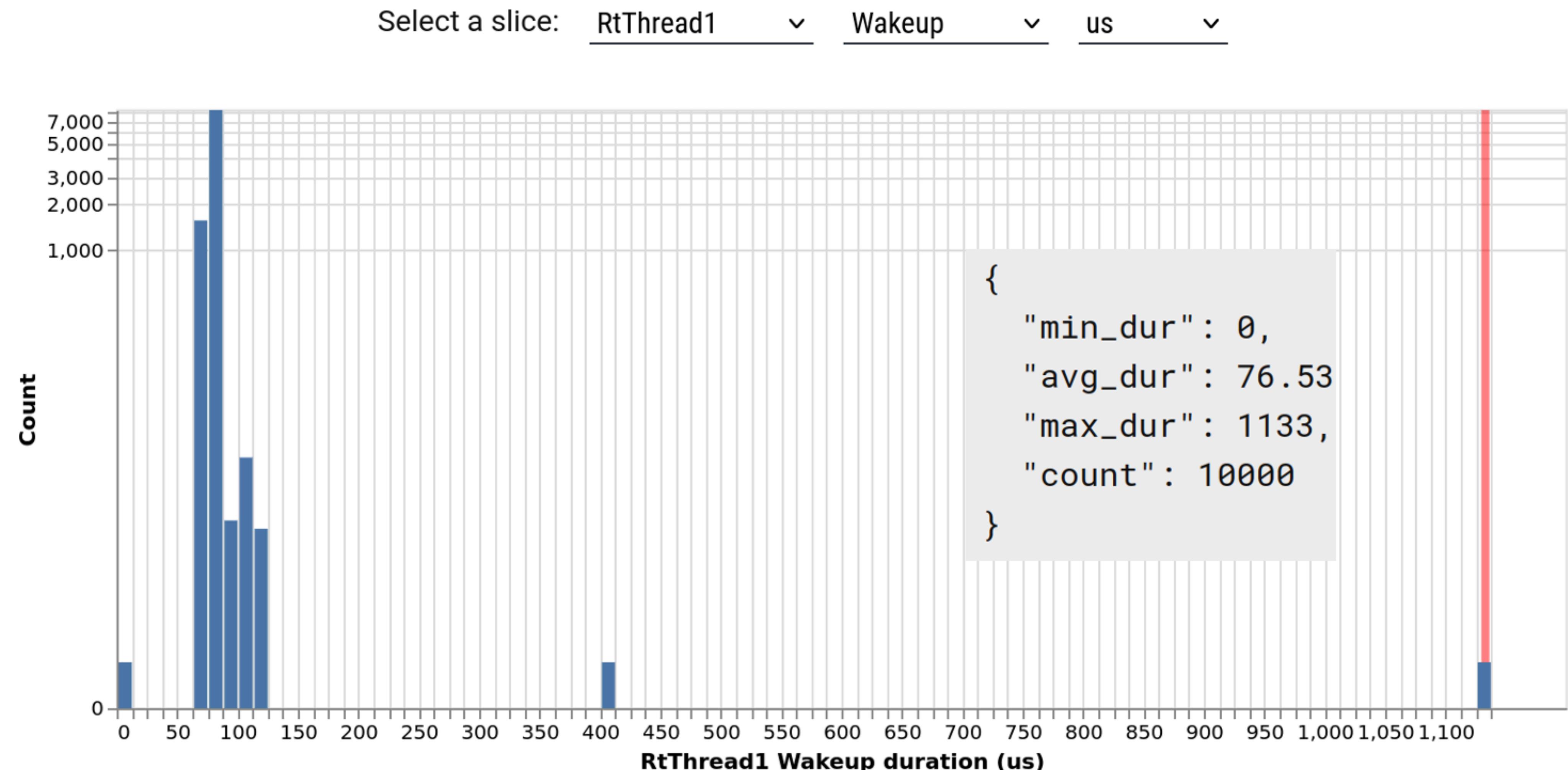
1. Sigue el README para iniciar el contenedor Docker
2. Conecta Raspberry Pi con Ethernet
3. \$ ssh ubuntu@192.168.10.1
 - a. Contraseña: ubuntu
4. Continúa con los pasos 2 → 3 desde la izquierda (pero sin docker/shell y hazlo en el shell de Raspberry Pi)
5. Ve a <http://192.168.10.1/repo>
6. Navega hasta `exercise1` y descarga `exercise1.perfetto`
7. Continúa desde el paso 7 a la izquierda.

Si no terminaste, ve a <http://localhost:3100> y carga el archivo desde exercise1/solutions/baseline.perfetto y haz clic en el enlace Latencia en la barra lateral izquierda

Select a slice: RtThread0 ▾ Wakeup ▾ us ▾



Si no terminaste, ve a <http://localhost:3100> y carga el archivo desde exercise1/solutions/baseline.perfetto y haz clic en el enlace Latencia en la barra lateral izquierda



¡Vamos a cambiarlo a un planificador en Tiempo Real!

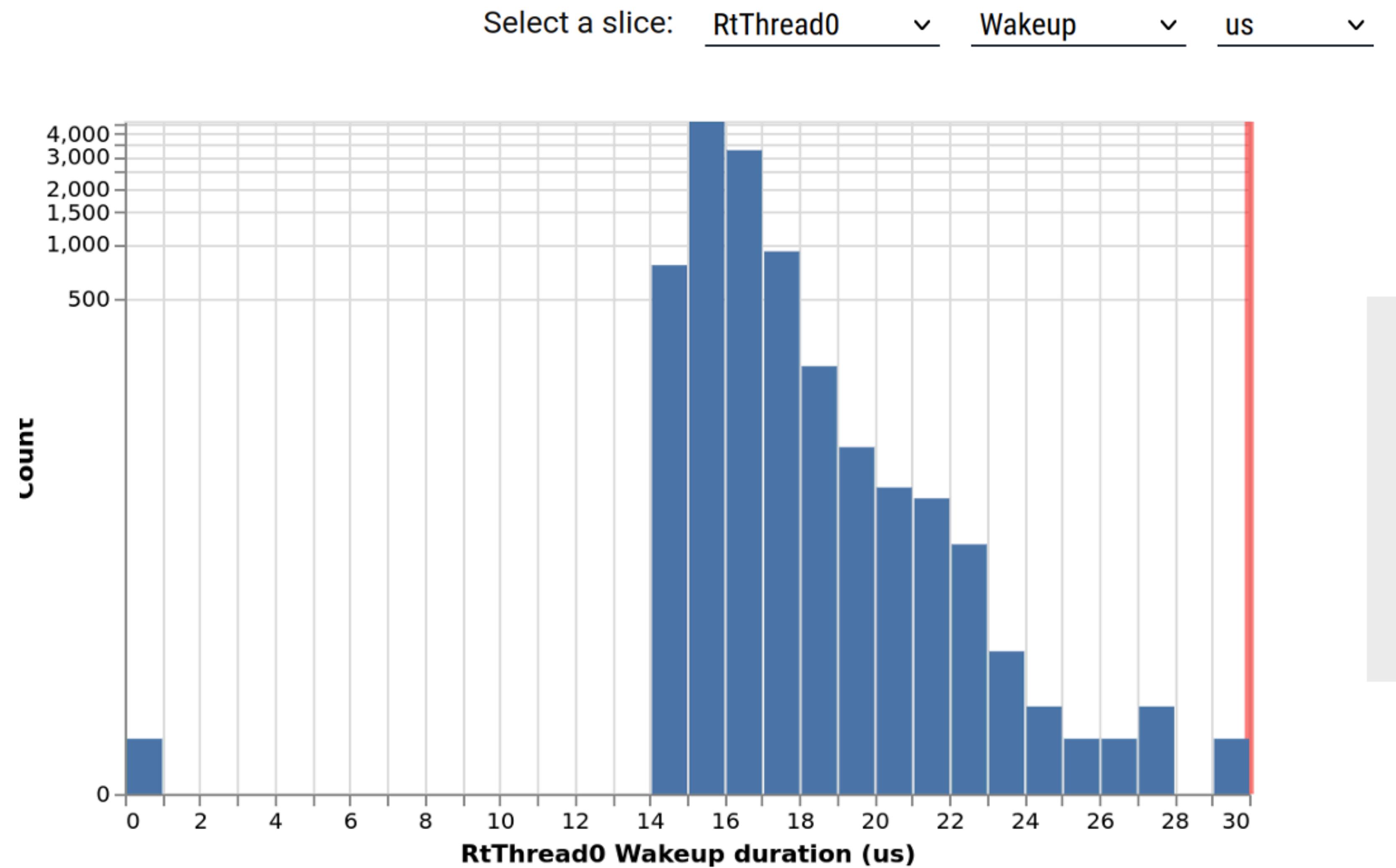
1. Abre el exercise1/src/latency_tester/main.cc
2. Cambia el archivo de la siguiente manera

```

9 cactus_rt::CyclicThreadConfig CreateRtThreadConfig(uti
10   cactus_rt::CyclicThreadConfig config;
11
12 config.setOtherScheduler(); ← Borra esta línea
13
14 // Uncomment the following line to use the real-time
15 // config.SetFifoScheduler(80); ← Descomenta
16 config.cpu_affinity = {index};
17
18 config.tracer_config.trace_loop = true;
19 config.tracer_config.trace_overrun = true;
20 config.tracer_config.trace_sleep = true;
21 config.tracer_config.trace_wakeup_latency = true;
22
23
24 return config;
25 }
```

1. Terminal 1:
 - a. \$ docker/shell
 - b. \$ cd /code/exercise1
 - c. \$ colcon build
 - d. \$./code/stress.sh
 2. Terminal 2:
 1. \$ docker/shell
 2. \$ cd /code/exercise1 && ./run.sh
 3. Deten el estrés en la terminal 1.
 4. En Perfetto (<http://localhost:3100>), haga clic en Open trace file y vuelva a cargar el archivo exercise1.perfetto
 5. Cargue la vista de latencia desde la barra lateral izquierda
- Usuarios de Raspberry Pi:
7. \$ docker/shell
 8. \$ upload-to-pi
 9. \$ ssh ubuntu@192.168.10.1
 10. Pasos 1 y 2 anteriores (sin docker/shell pero en el shell de Raspberry Pi)
 11. Descargue los datos desde <http://192.168.10.1/repo>
 12. Cargue el archivo de descarga exercise1.perfetto en Perfetto

Si no terminaste, ve a <http://localhost:3100> y carga el archivo desde exercise1/solutions/baseline.perfetto y haz clic en el enlace Latencia en la barra lateral izquierda



```
{  
    "min_dur": 0,  
    "avg_dur": 16.10  
    "max_dur": 30,  
    "count": 10014  
}
```

Resultados: latencia de activación con y sin planificador RT

- Latencia de activación = Latencia del hardware + del sistema operativo
- Latencia total >= Latencia de activación

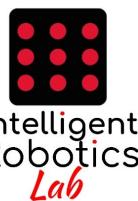
Platform	Without RT scheduler	With RT scheduler
Raspberry Pi with PREEMPT_RT	1100 µs	31 µs
Attendees' laptops	??	??

- **Conclusión: ¡el planificador en Tiempo Real es necesario para las aplicaciones en Tiempo Real!**
 - También se prefiere PREEMPT_RT
 - También se necesita hardware sin latencia en Tiempo Real
 - Utilice cyclictest para aplicaciones de producción



Notas sobre el resto de ejercicios del Workshop

- Todos siguen un flujo de trabajo similar
- Consulta la hoja de referencia sobre el flujo de trabajo y el uso de varios sistemas (Perfetto, Raspberry Pi, Docker)
- Ayuda a otros miembros de tu grupo si están estancados
- Haz preguntas si las tienes

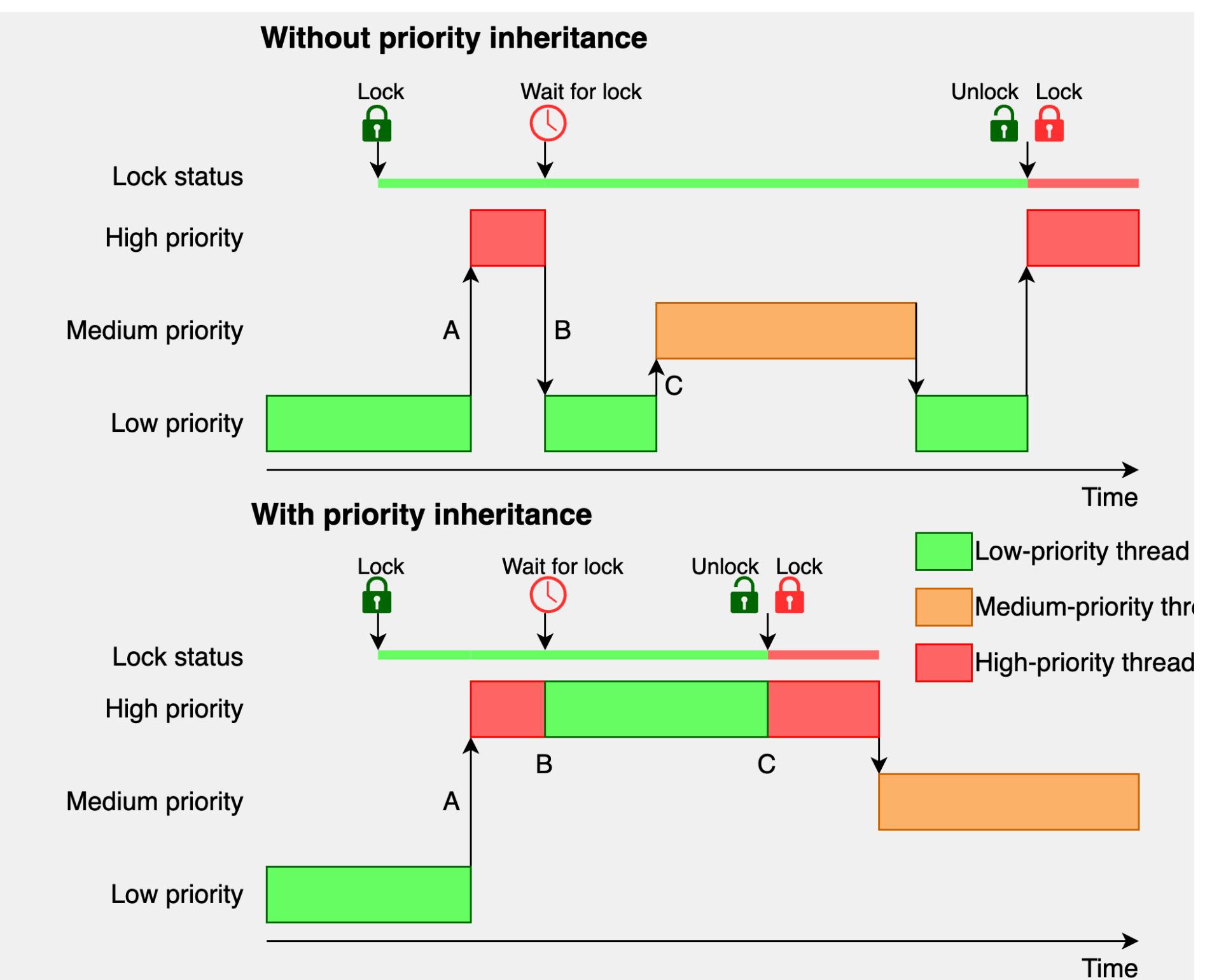


Introducción a la programación RT en Linux sin ROS

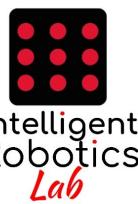


Nos lo saltamos, pero las conclusiones son

- Desactive la paginación y el intercambio a demanda
- No asigne memoria dinámicamente
- Use mutex con herencia de prioridad o planificación sin bloqueo en lugar de `std::mutex`
- Use un sistema de log asíncrono para imprimir
 - Use colas sin bloqueo para el registro de datos
- Asegúrese de que las bibliotecas a las que llama también tengan latencia limitada

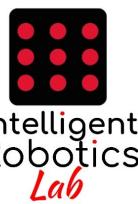


Gestión de ejecución en ROS 2

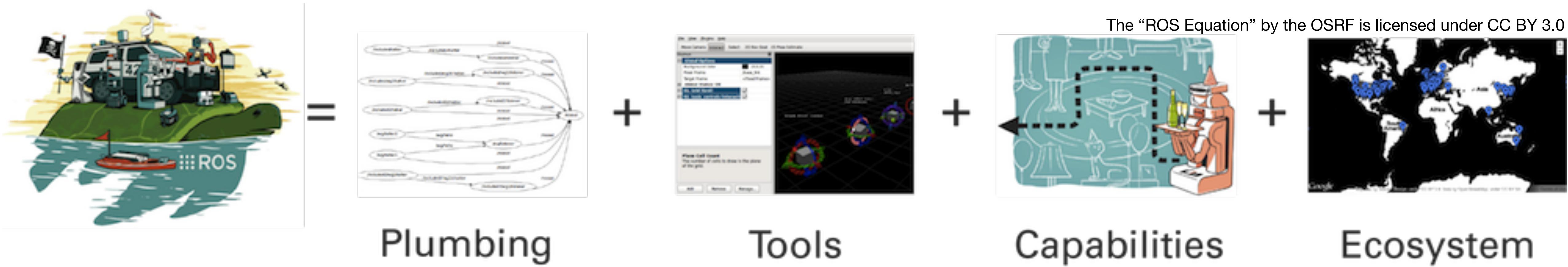


Contenidos: Gestión de la ejecución de ROS 2

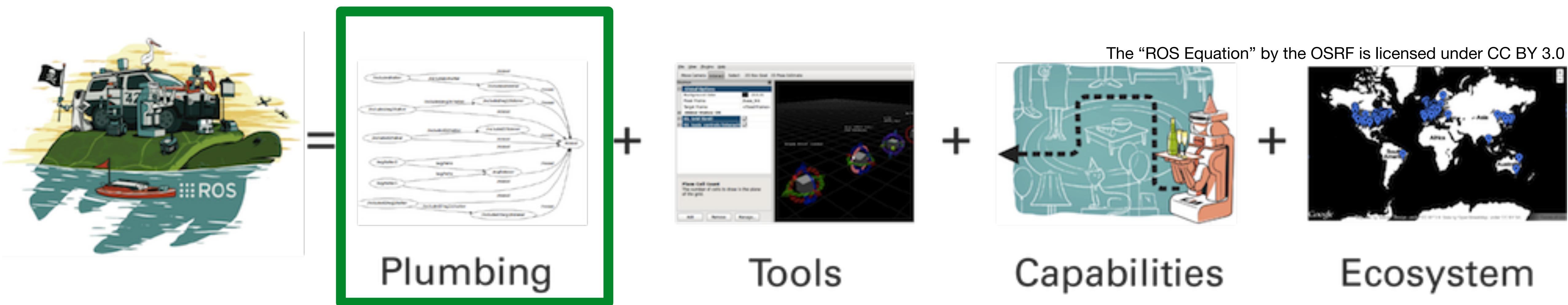
- Gestión de la ejecución en ROS 2
- Impacto en el comportamiento en Tiempo Real y el determinismo
- Más sobre ejecutores y alternativas
- Ejecutor experimental en Tiempo Real



La Ecuación ROS 2



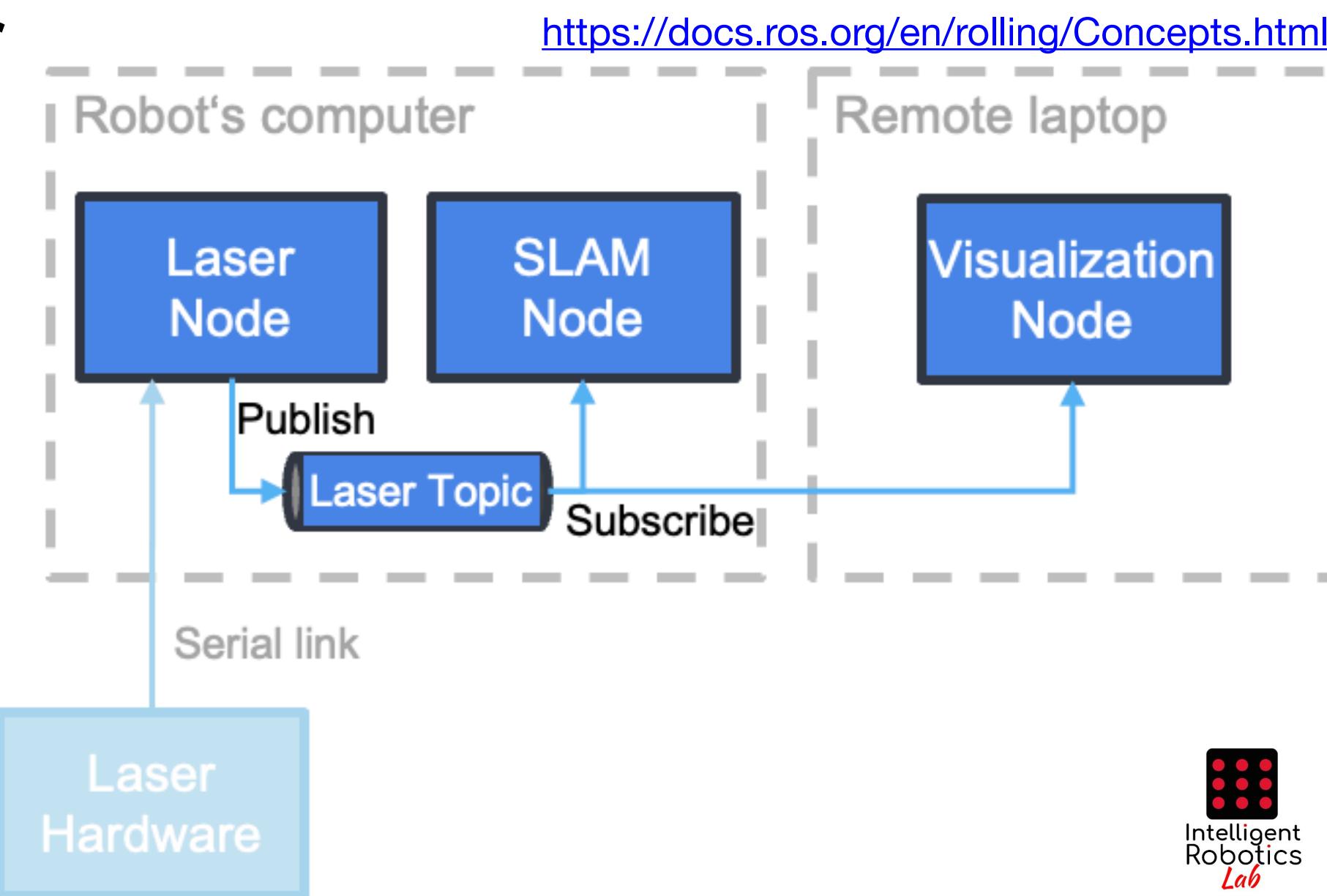
La Ecuación ROS 2



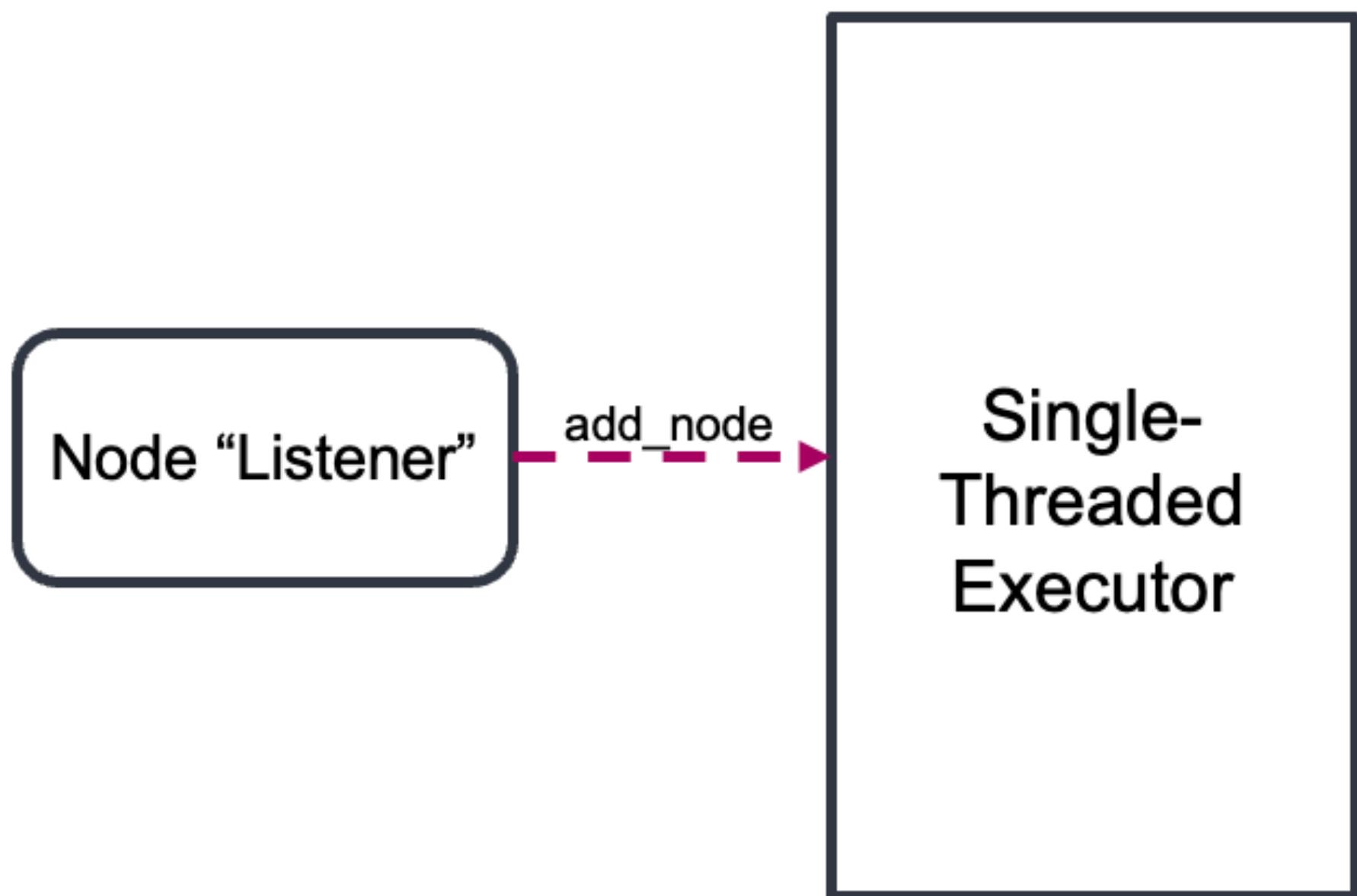
- Gestión de procesos
- Comunicación
- Controladores de dispositivos
- Modelos de datos
- Independencia del lenguaje

Nodos y Comunicación

- Entidad básica: Nodos que intercambian mensajes
- Pueden distribuirse entre máquinas
- Patrones de comunicación estándar
 - Topics: Publicación-suscripción (1 – n, unidireccional, asíncrono)
 - Servicios: Solicitud-respuesta (n – 1, bidireccional, síncrono)
 - Acciones: Solicitud-respuesta avanzada (1-1, multiestado)
- Nodos compuestos por funciones que se activan por datos o tiempo
 - Implementados en C++, Python, ...
 - Ejecución hasta su finalización



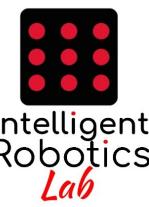
Ejemplo con ejecutor declarado implícitamente



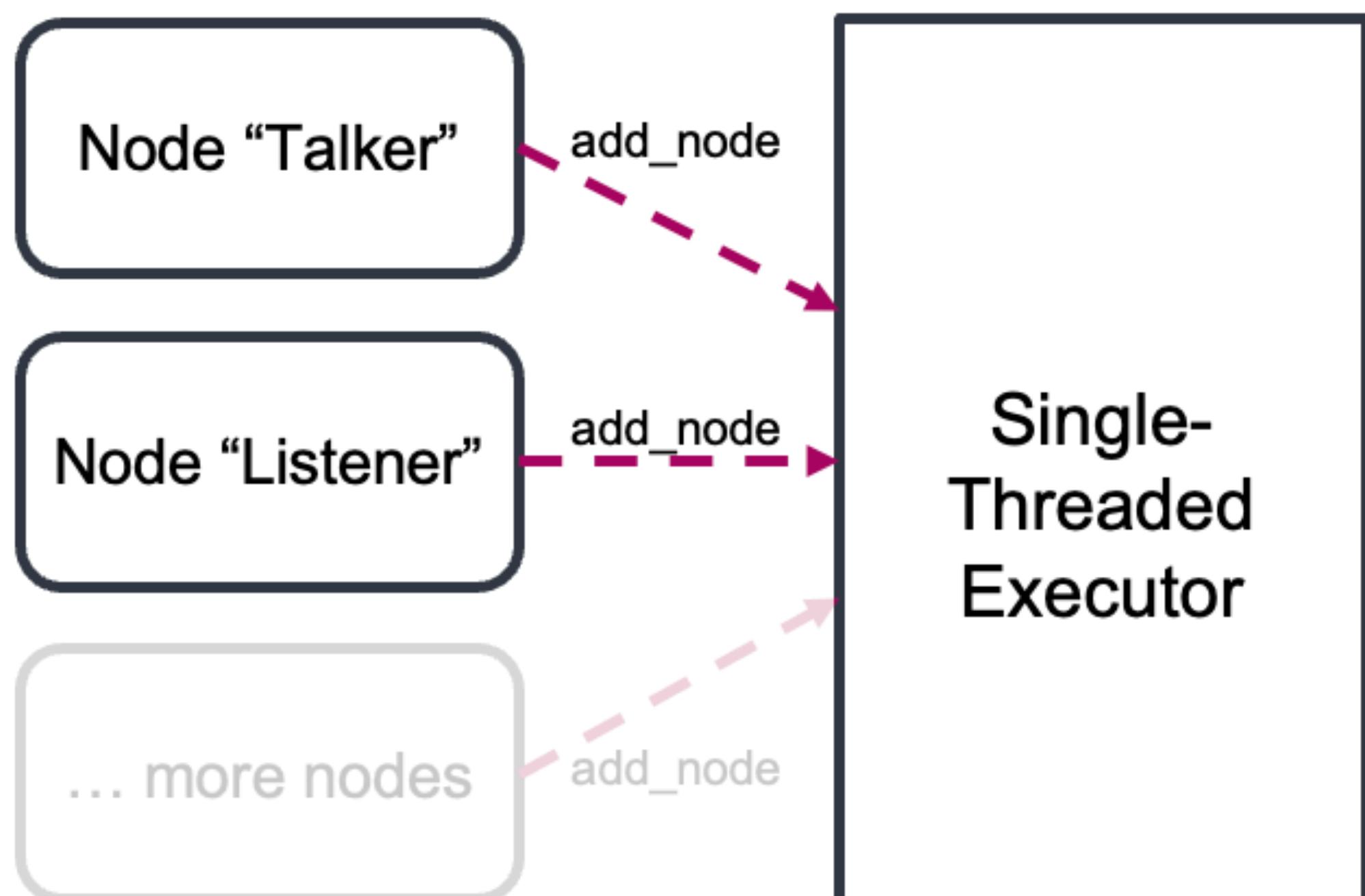
```
int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);

    auto node = rclcpp::Node::make_shared("listener");
    auto sub = node->create_subscription<std_msgs::msg::String>(
        "/chatter", callback);

    rclcpp::spin(node);
    rclcpp::shutdown();
}
```

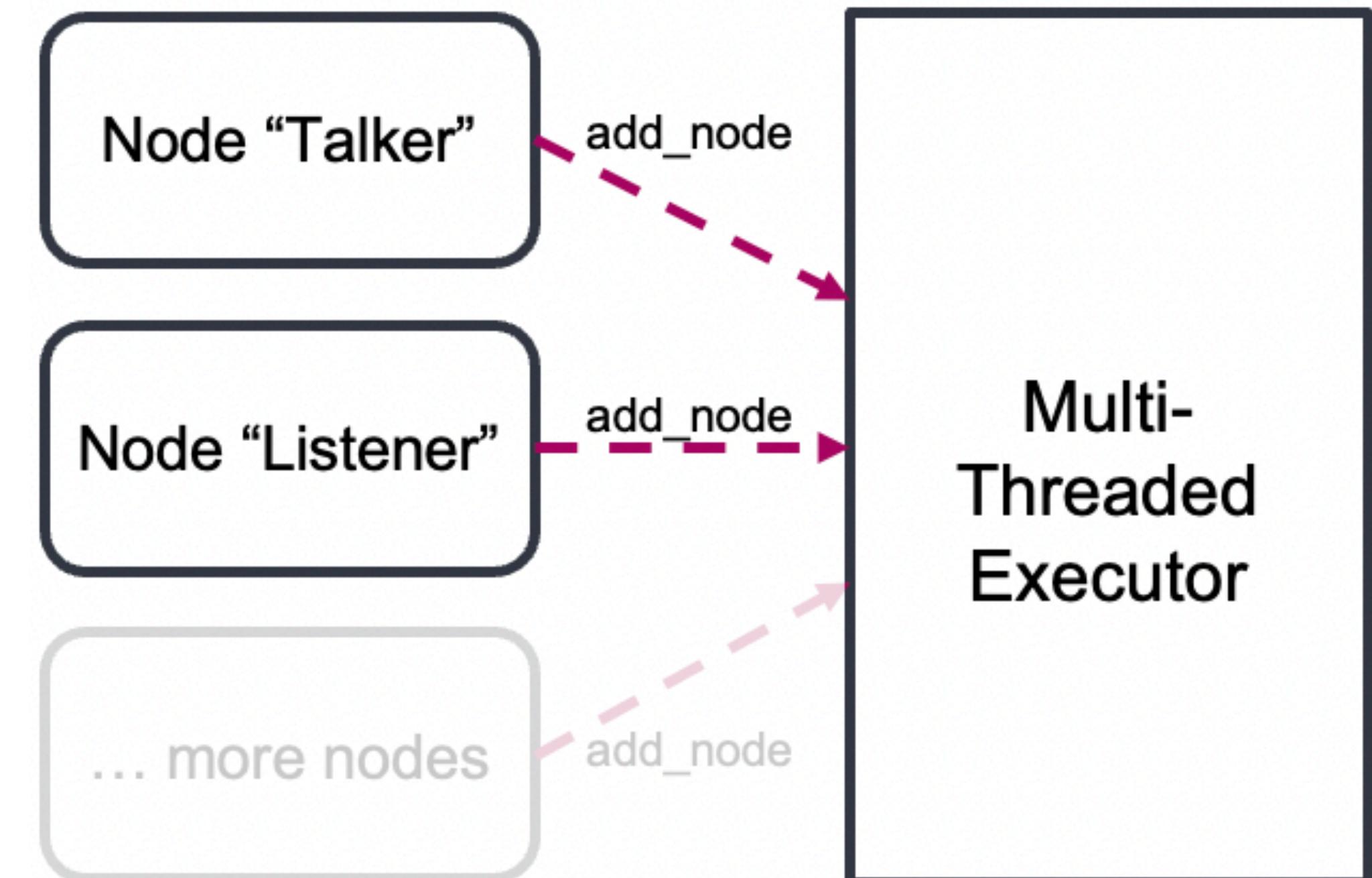


Ejemplo con SingleThreadedExecutor



```
int main(int argc, char * argv[]) {  
    rclcpp::init(argc, argv);  
  
    rclcpp::executors::SingleThreadedExecutor executor;  
  
    auto talker_node = std::make_shared<Talker>();  
    executor.add_node(talker_node);  
  
    auto listener_node = std::make_shared<Listener>();  
    executor.add_node(listener_node);  
  
    executor.spin();  
    rclcpp::shutdown();  
}
```

Ejemplo con MultiThreadedExecutor



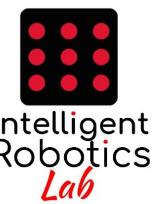
```
int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);

    rclcpp::executors::MultiThreadedExecutor executor;

    auto talker_node = std::make_shared<Talker>();
    executor.add_node(talker_node);

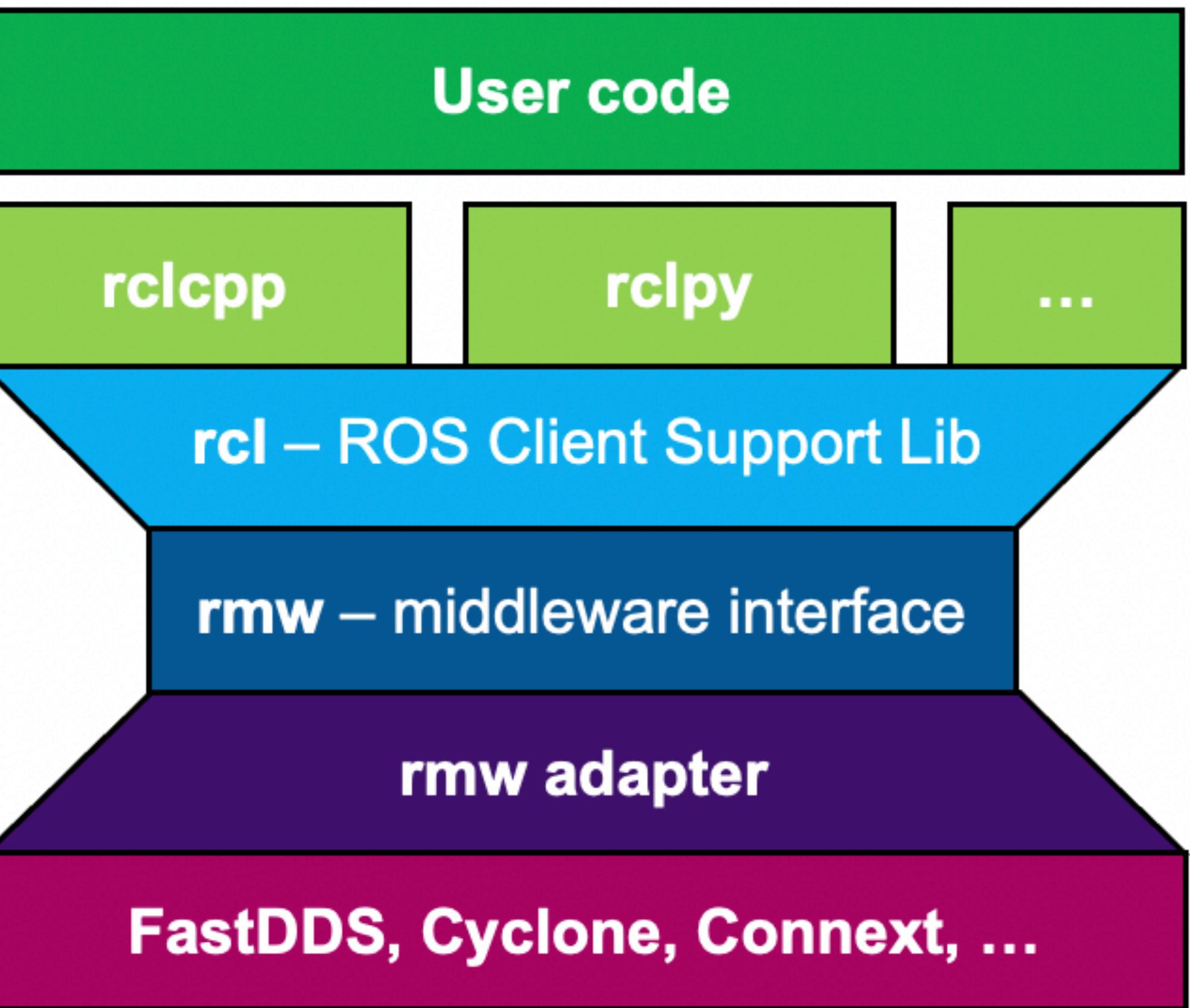
    auto listener_node = std::make_shared<Listener>();
    executor.add_node(listener_node);

    executor.spin();
    rclcpp::shutdown();
}
```



Arquitectura en capas de ROS 2

- **rcl*** – bibliotecas de cliente ROS específicas del lenguaje
- **rcl** – biblioteca C
 - Garantiza los mismos algoritmos básicos en todas las bibliotecas de cliente específicas del lenguaje
- **rmw** – interfaz de middleware ROS
 - Oculta detalles específicos de las implementaciones de DDS
 - Optimiza la configuración de QoS
- **rmw_*** – adaptadores DDS

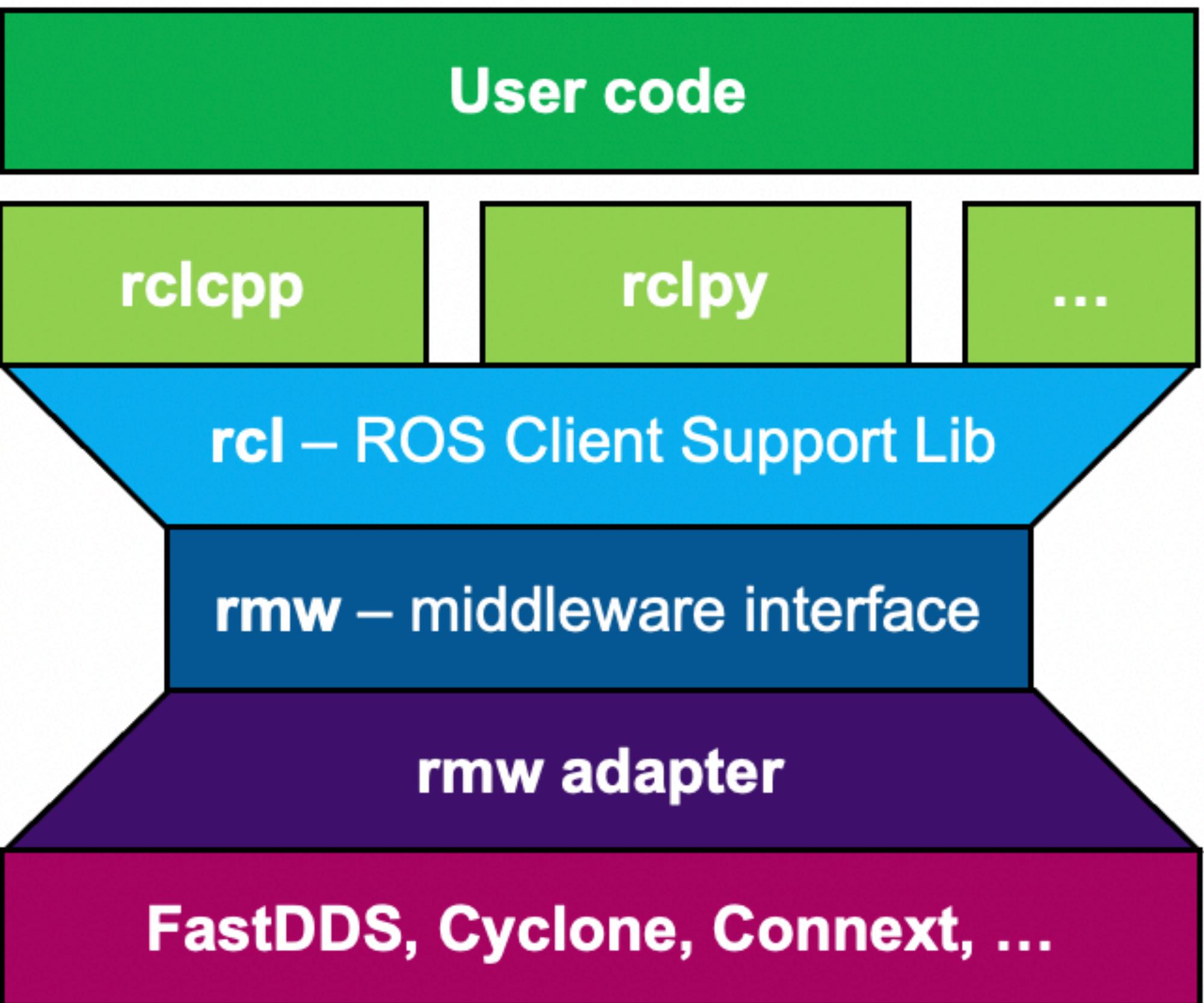


Gestión de la Ejecución

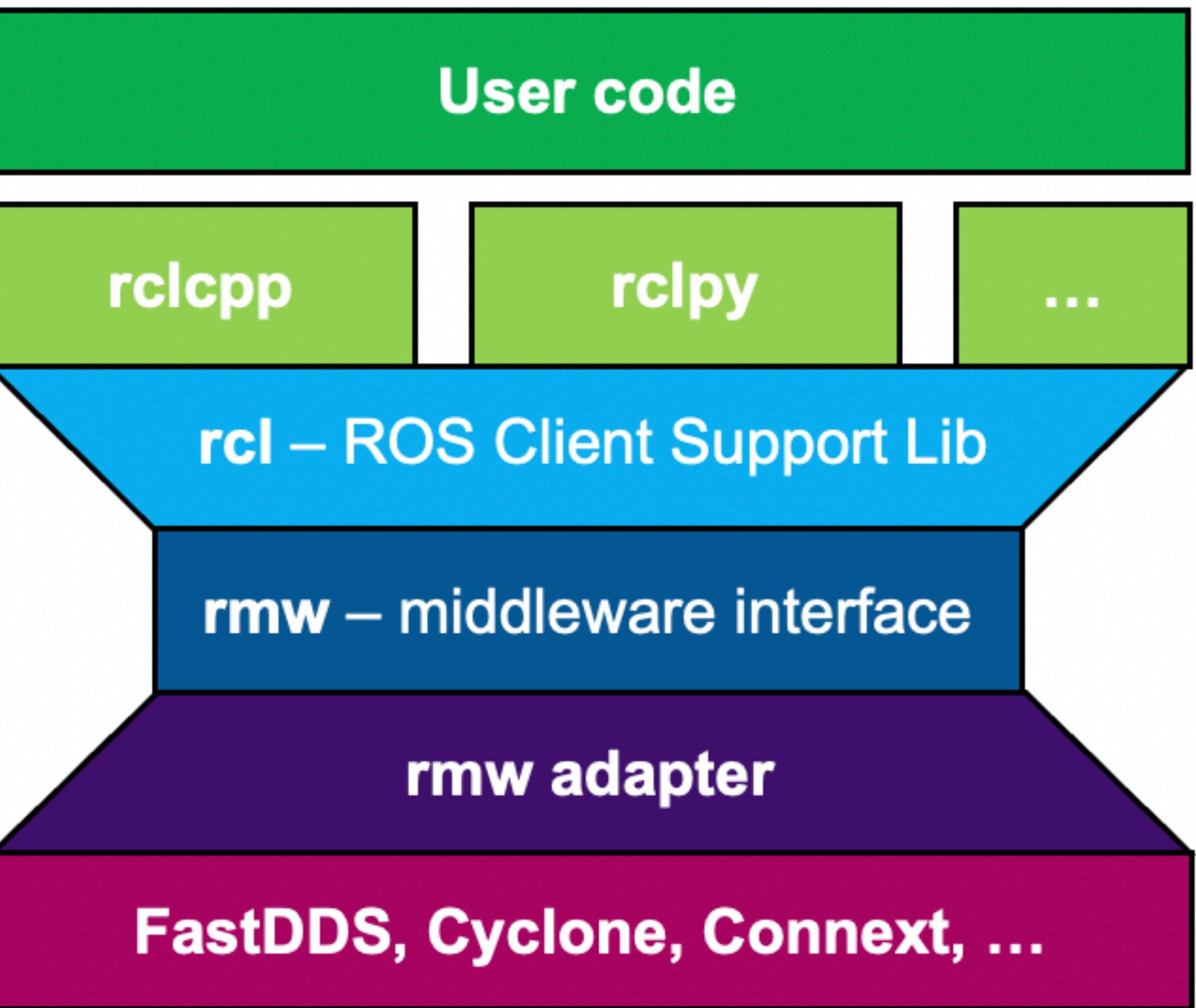
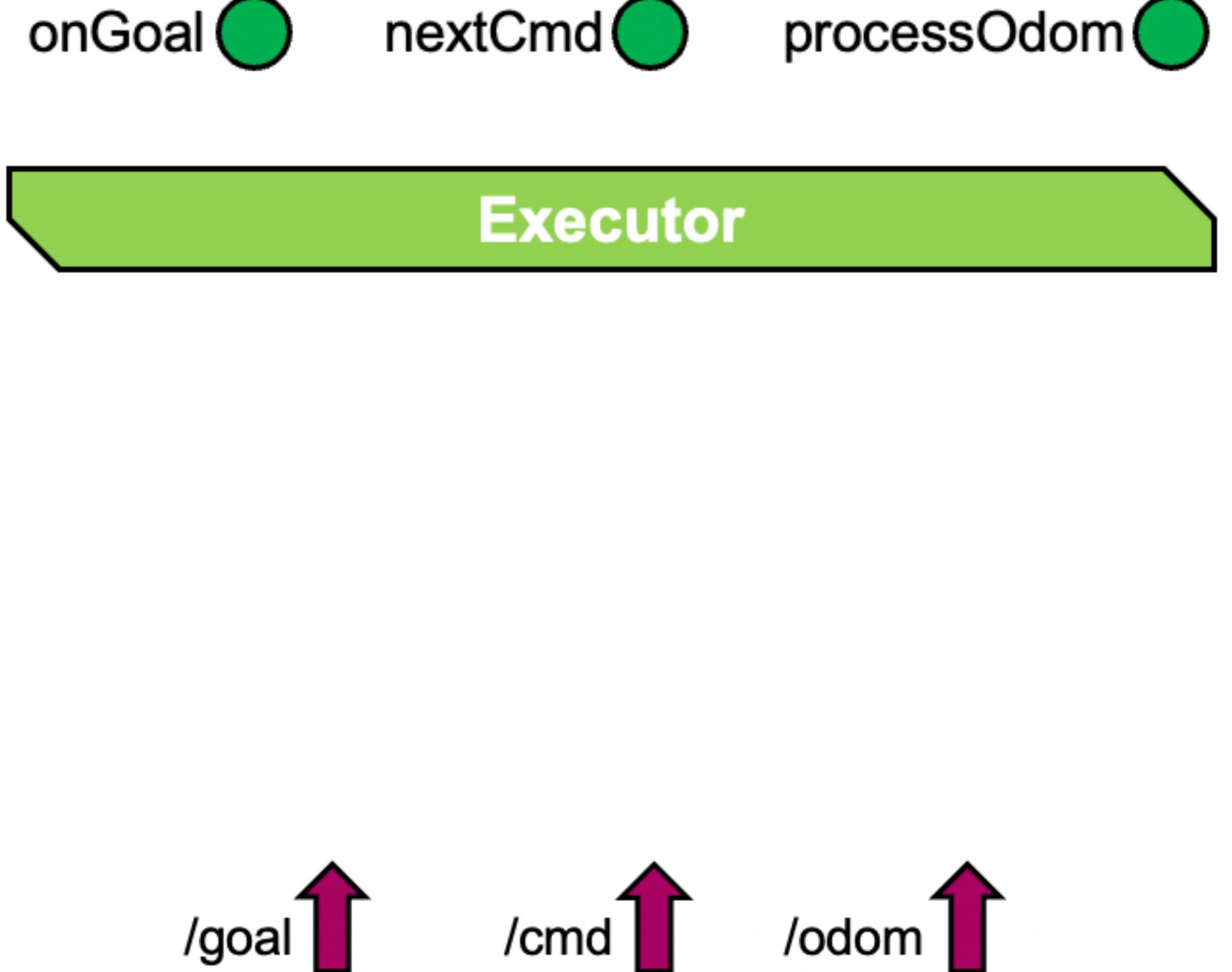
<https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Executors.html>



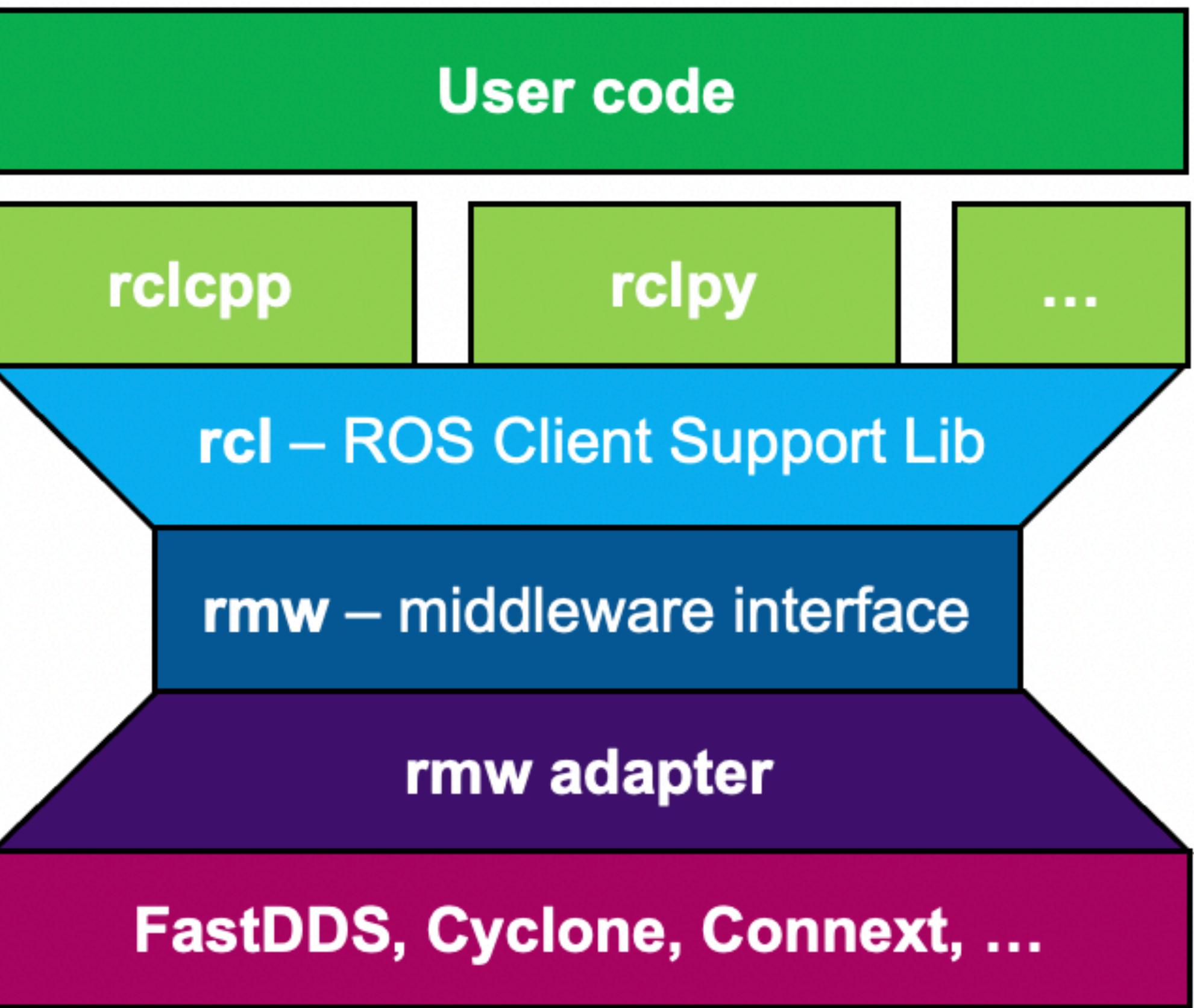
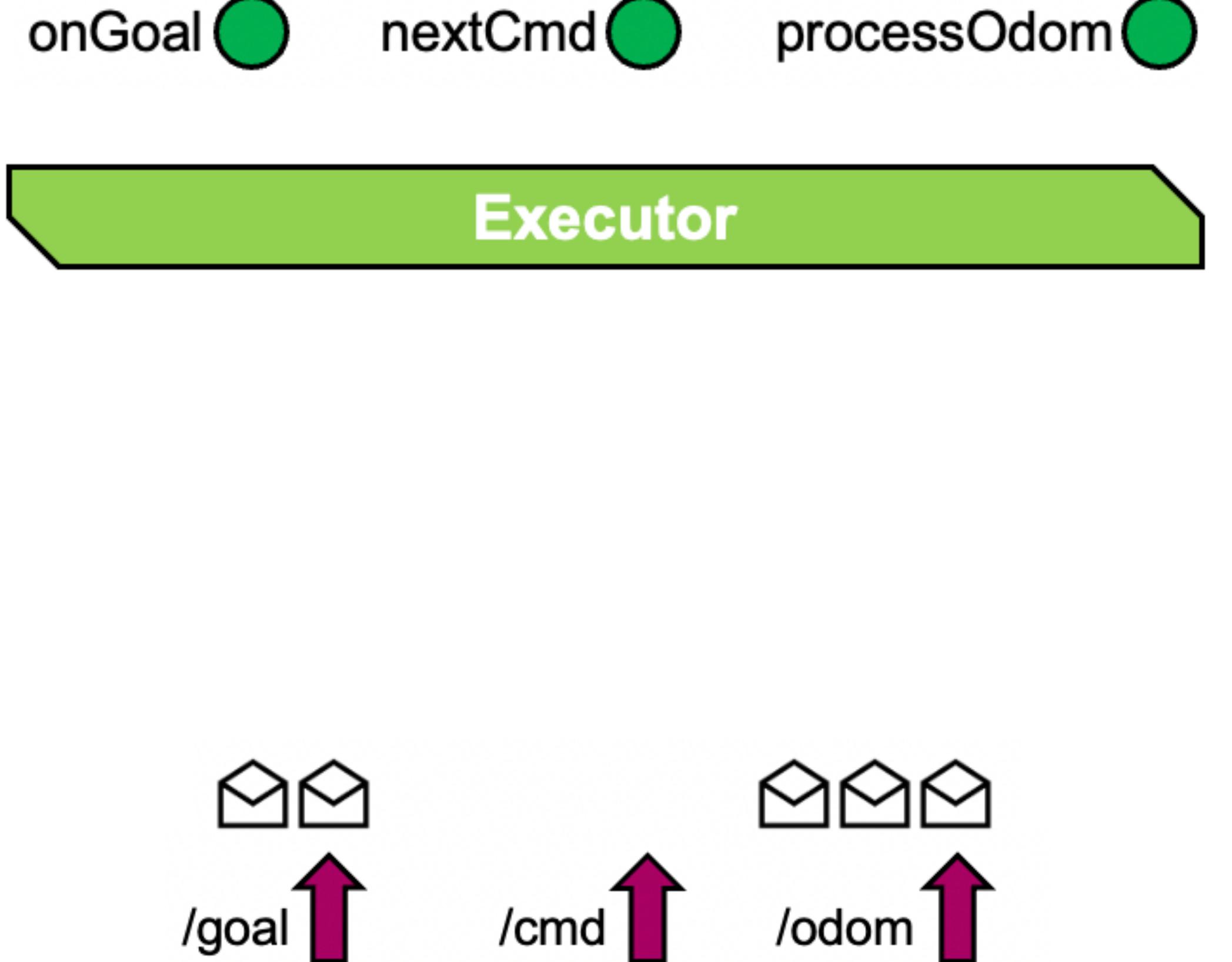
- Invoca callbacks de suscripciones, temporizadores, servicios, etc. en mensajes y eventos entrantes
- Utiliza uno o varios threads



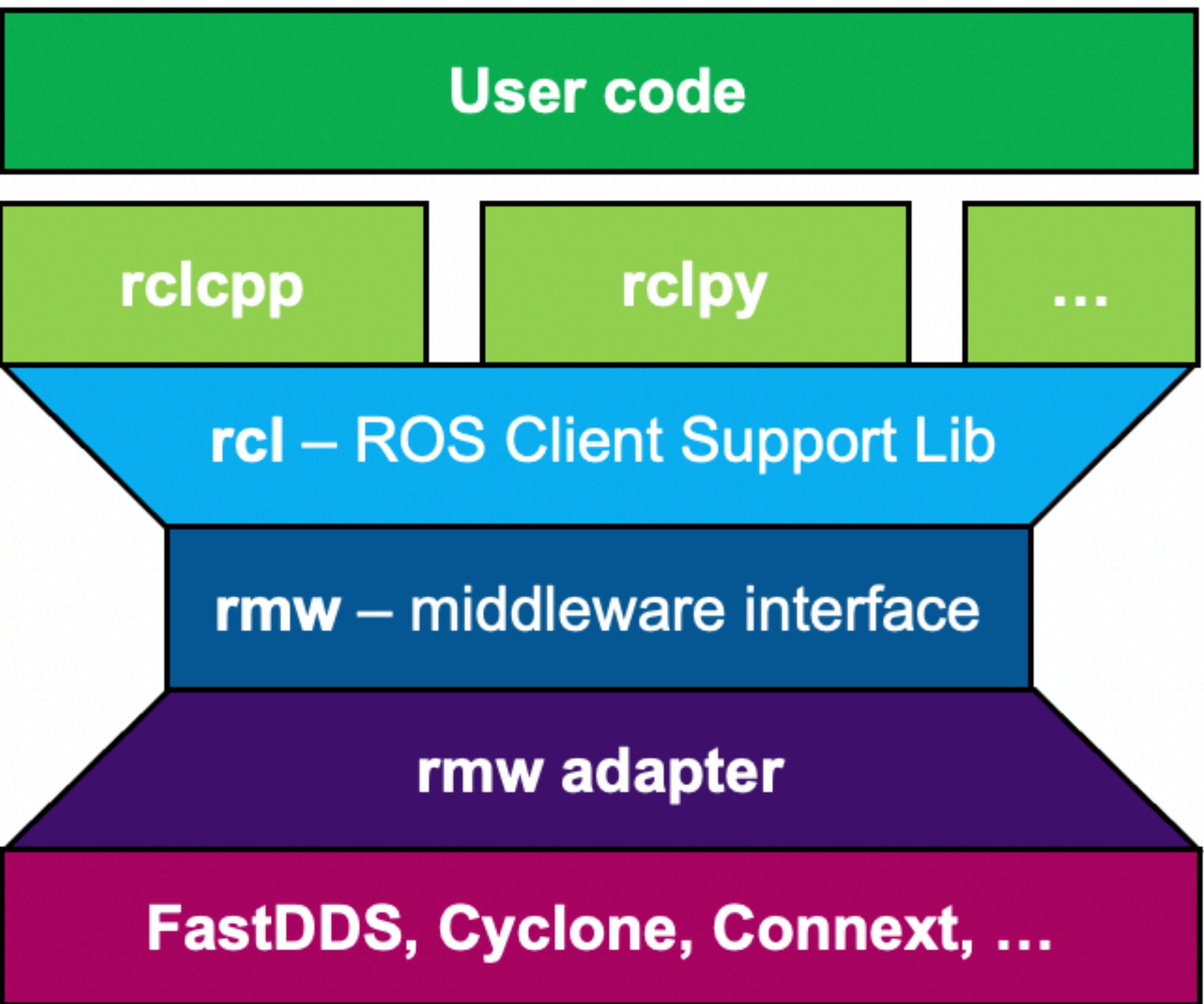
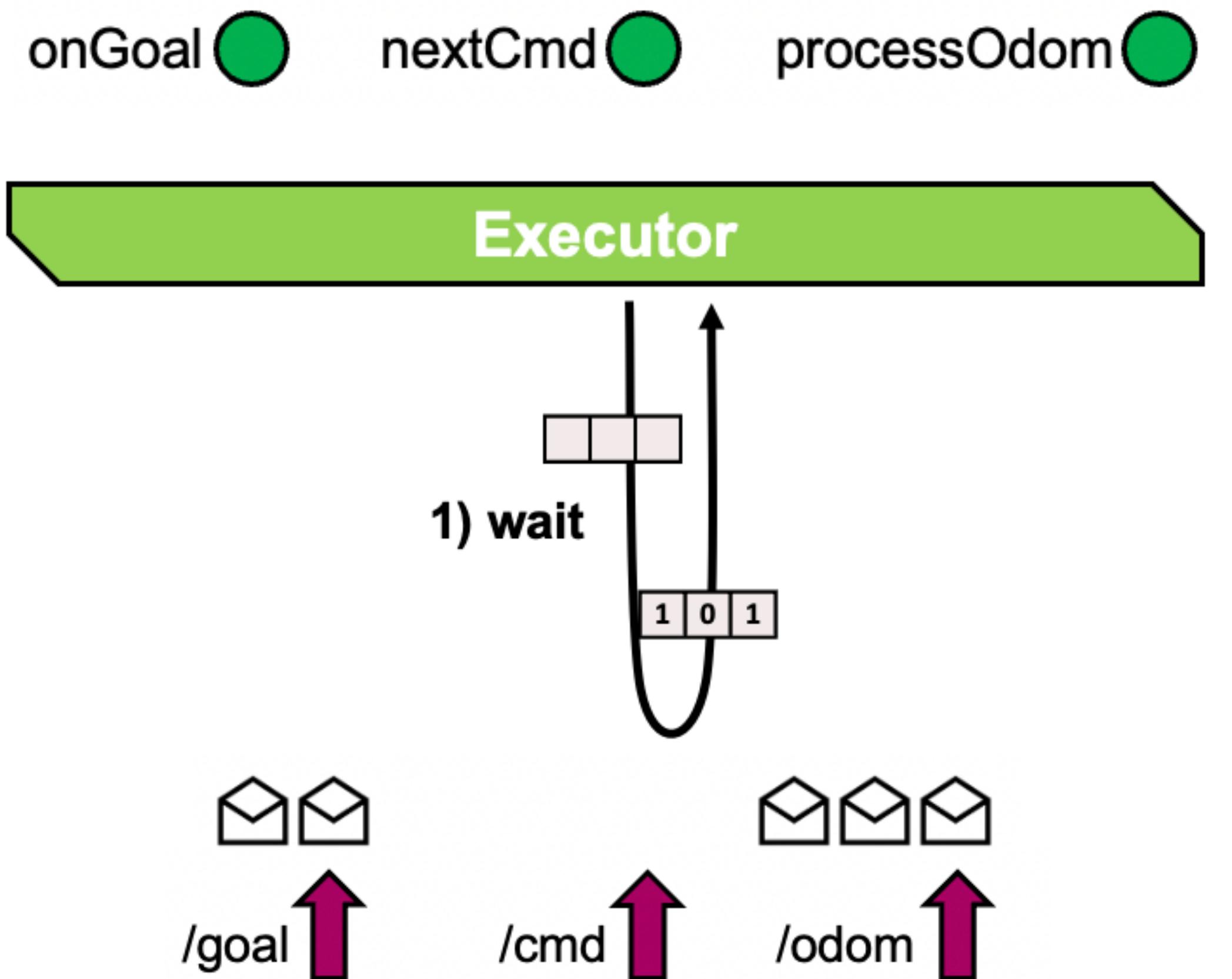
Gestión de la Ejecución



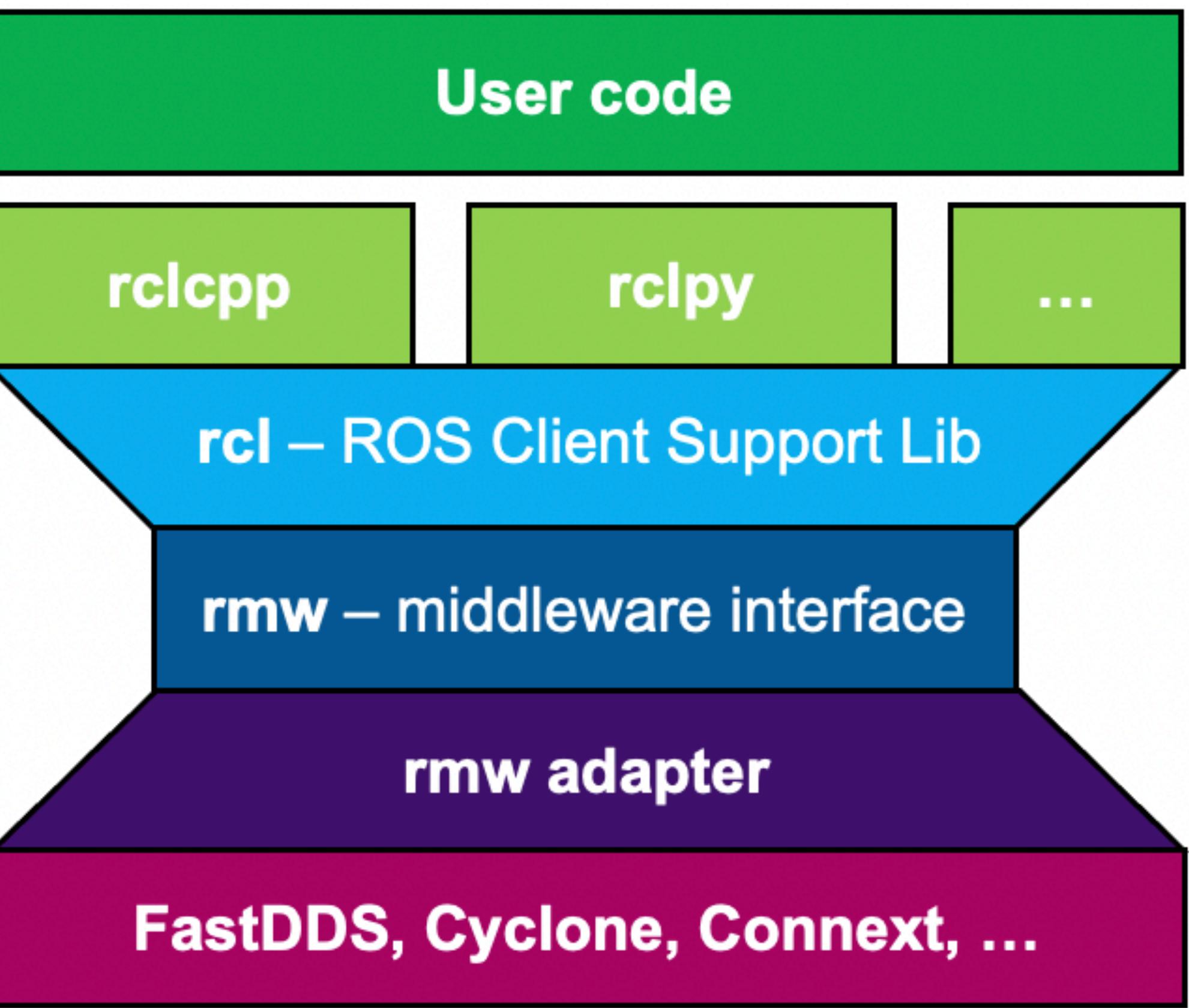
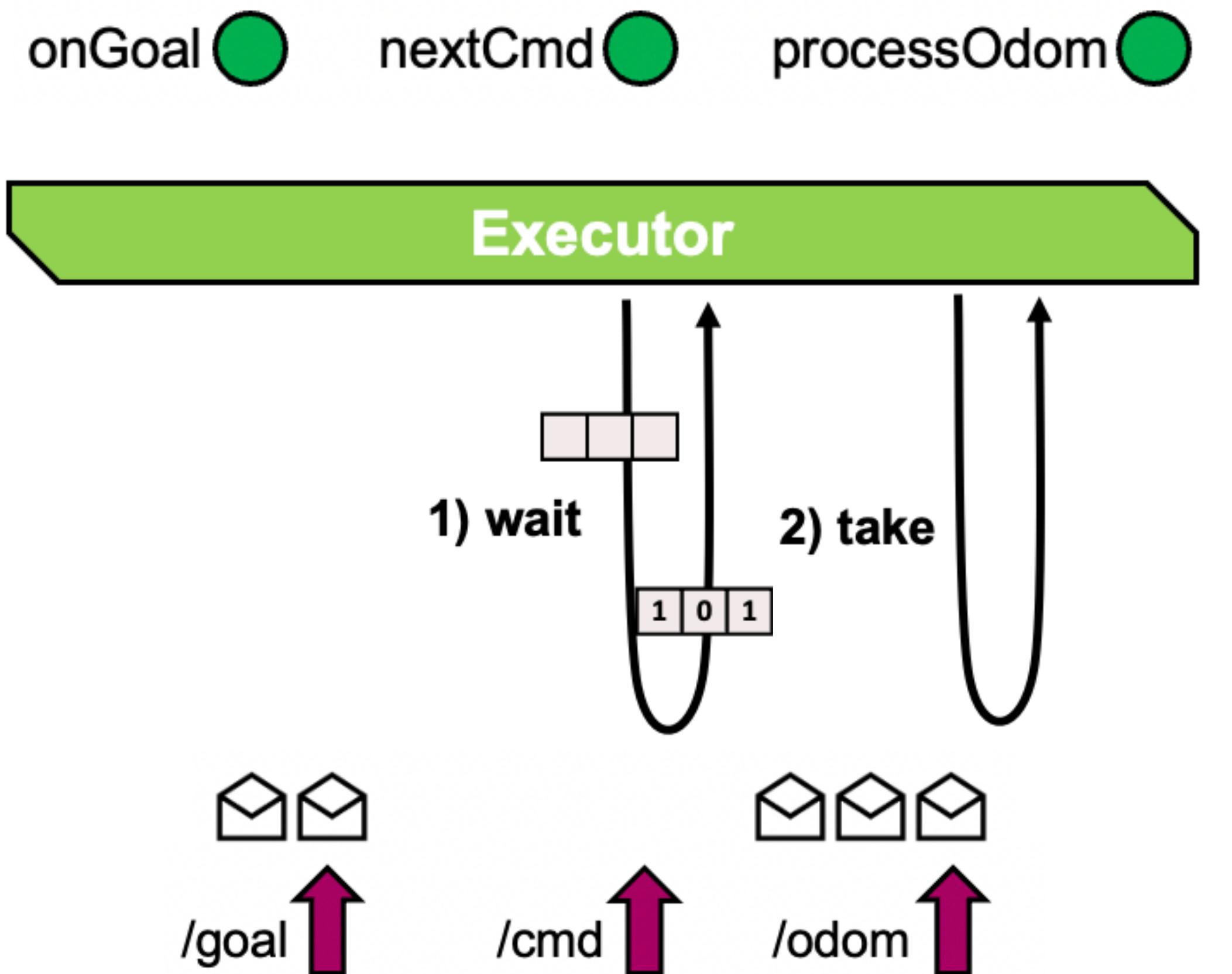
Gestión de la Ejecución



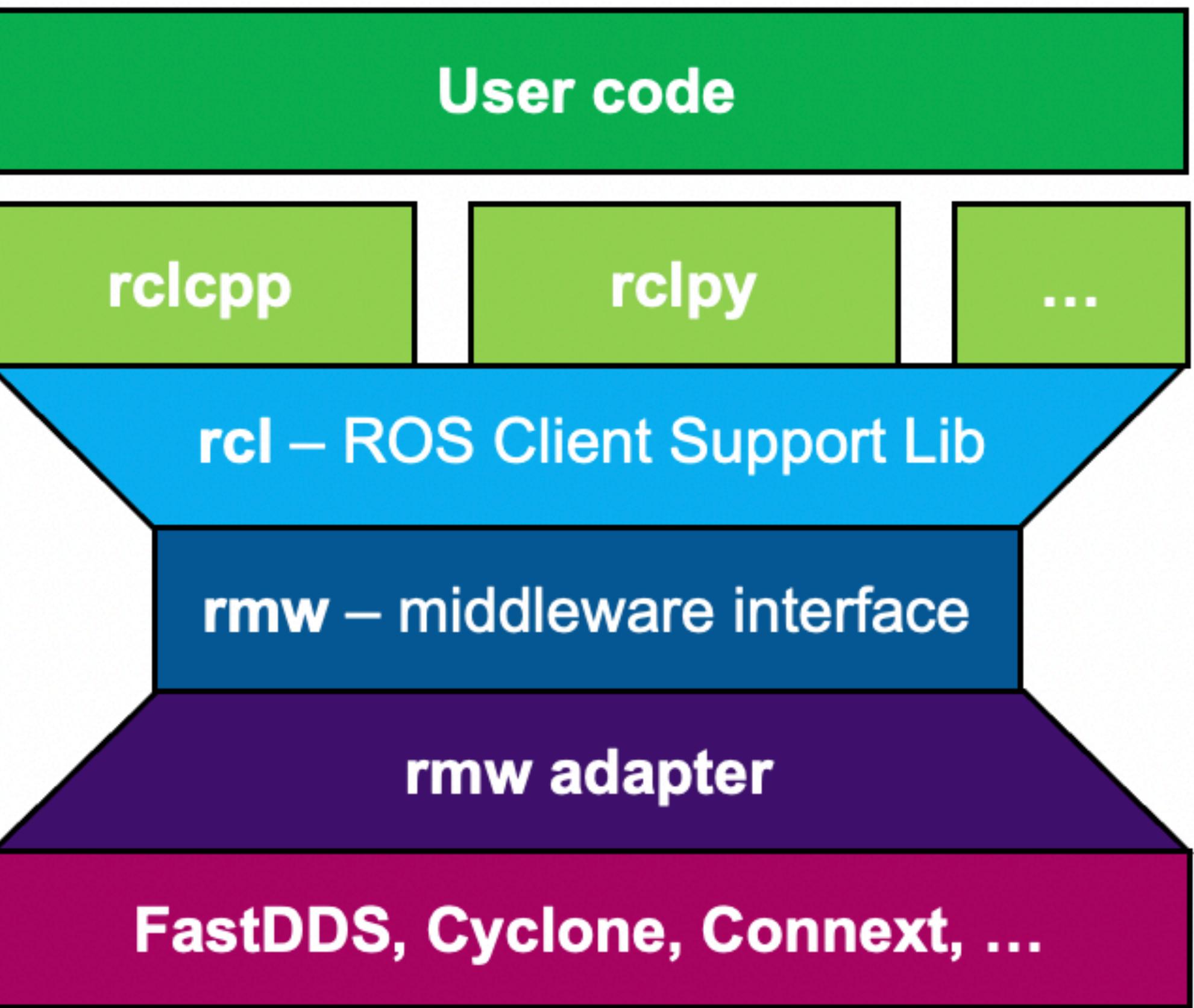
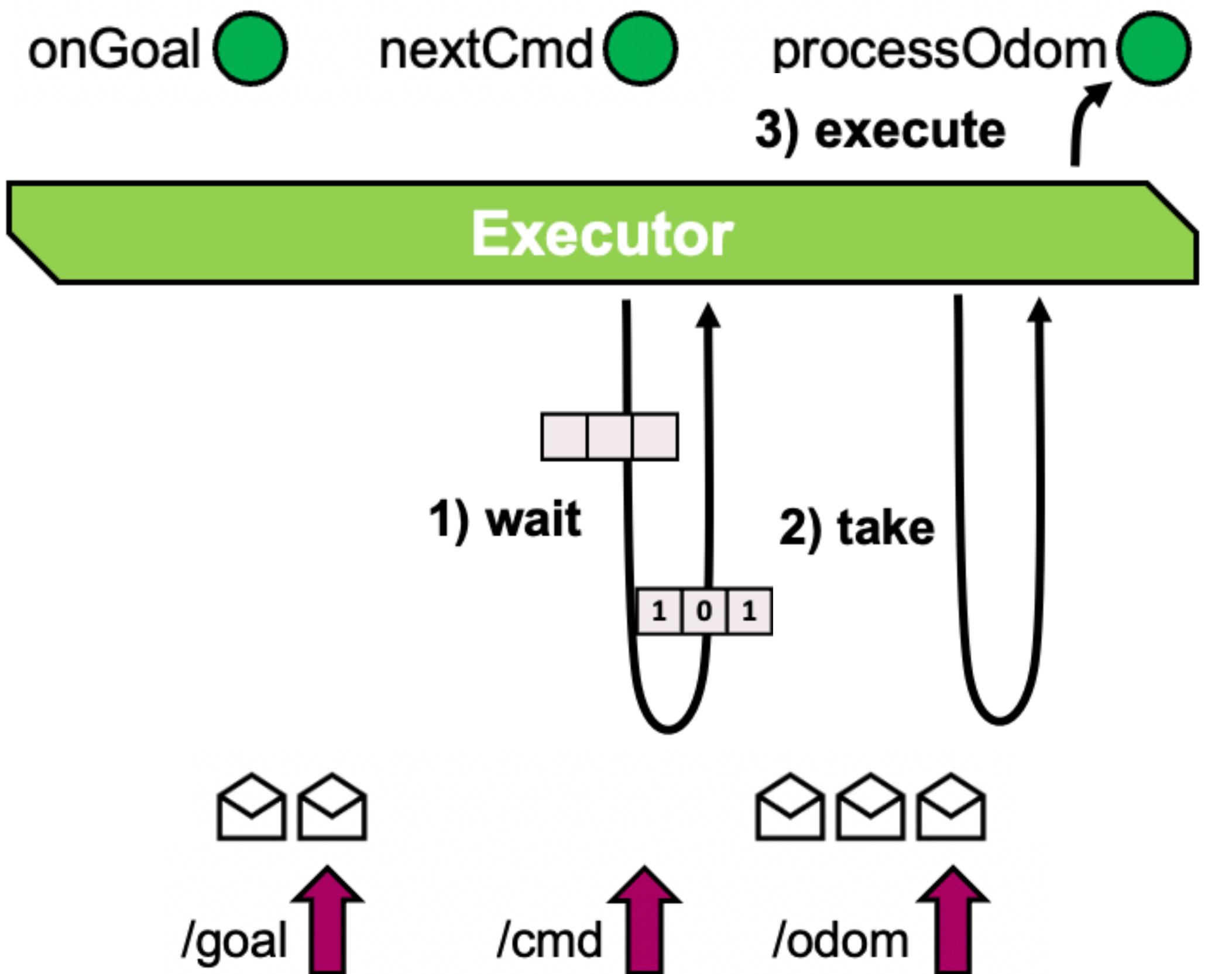
Gestión de la Ejecución



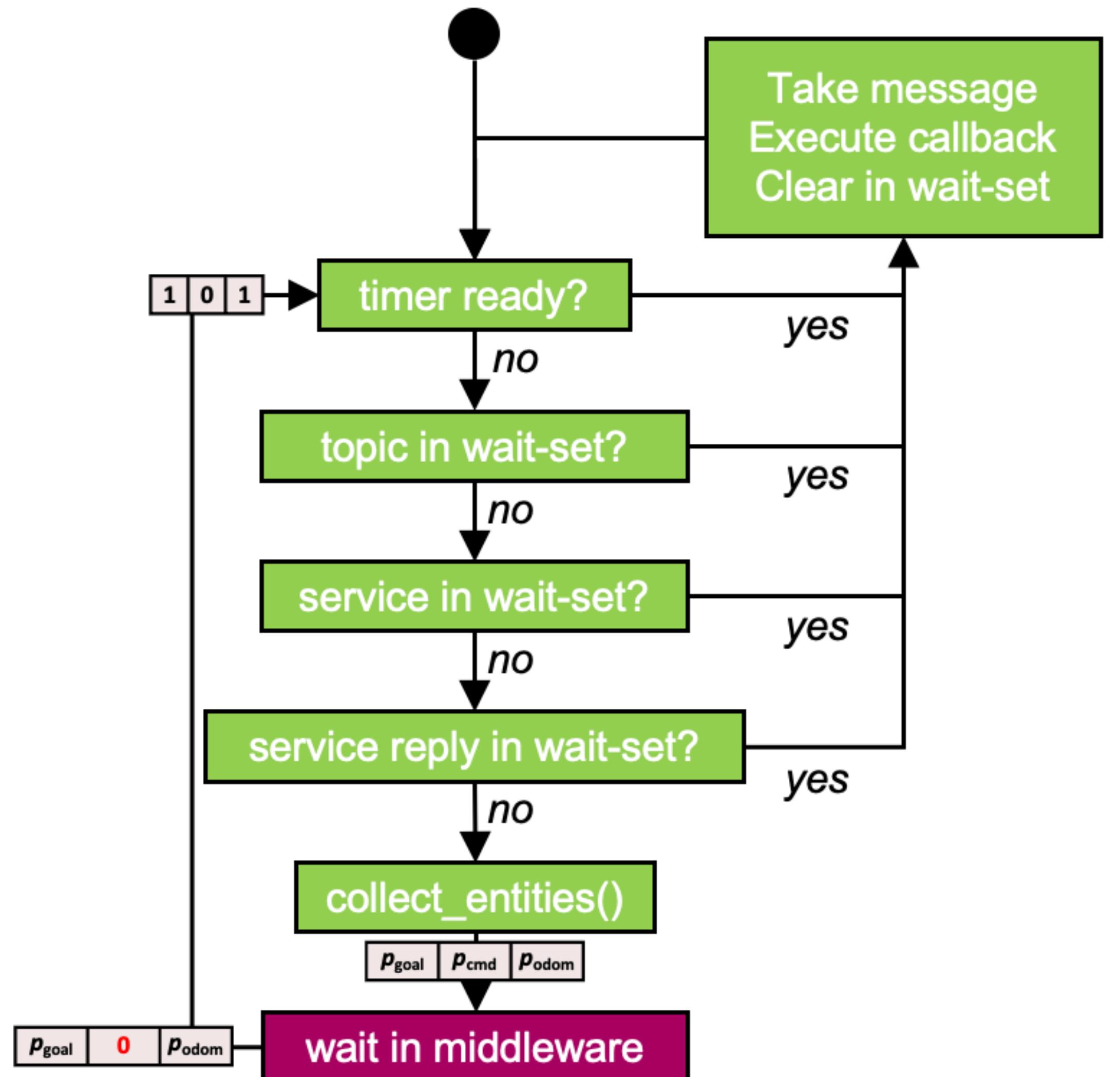
Gestión de la Ejecución



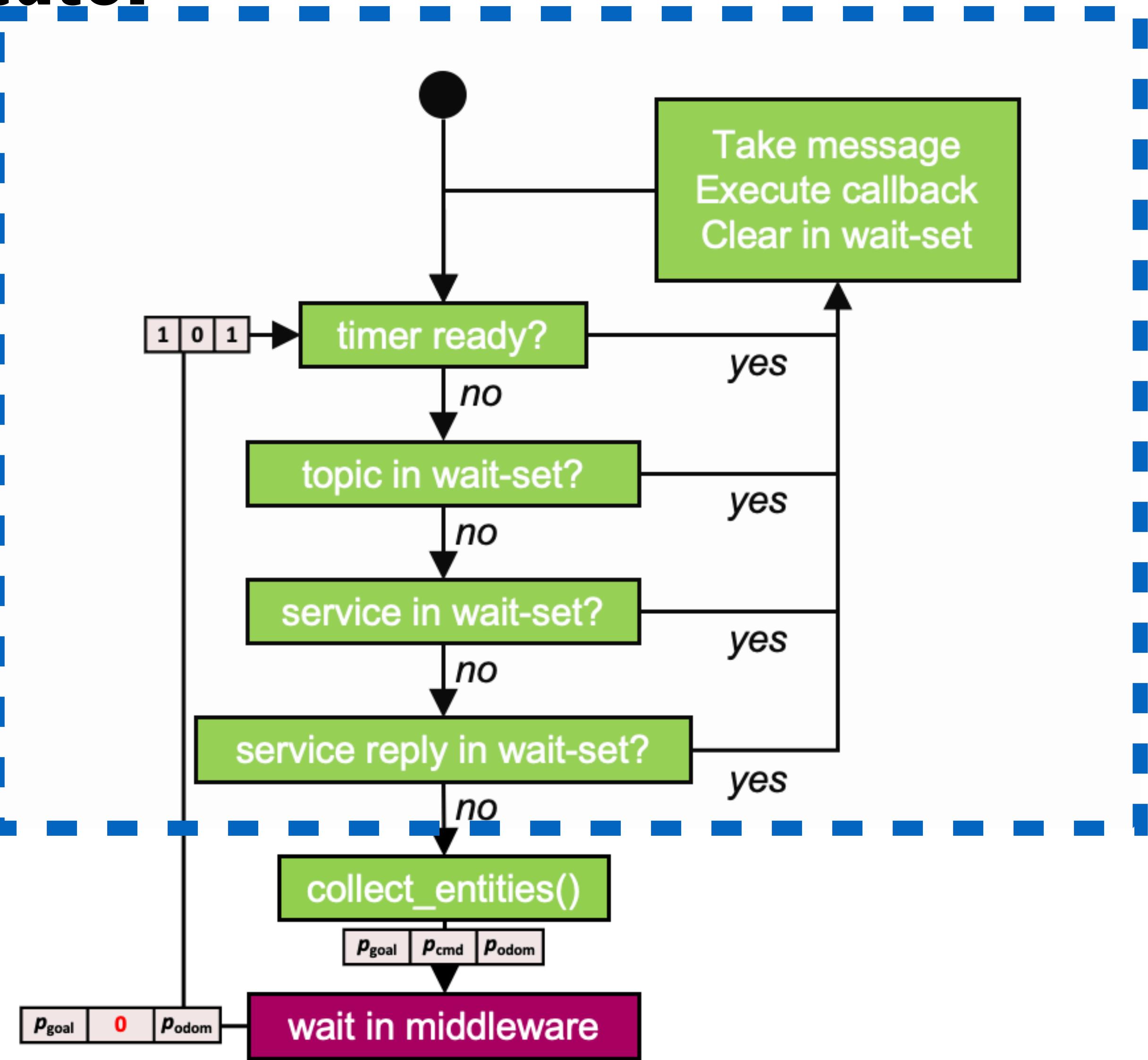
Gestión de la Ejecución



Semántica del ejecutor

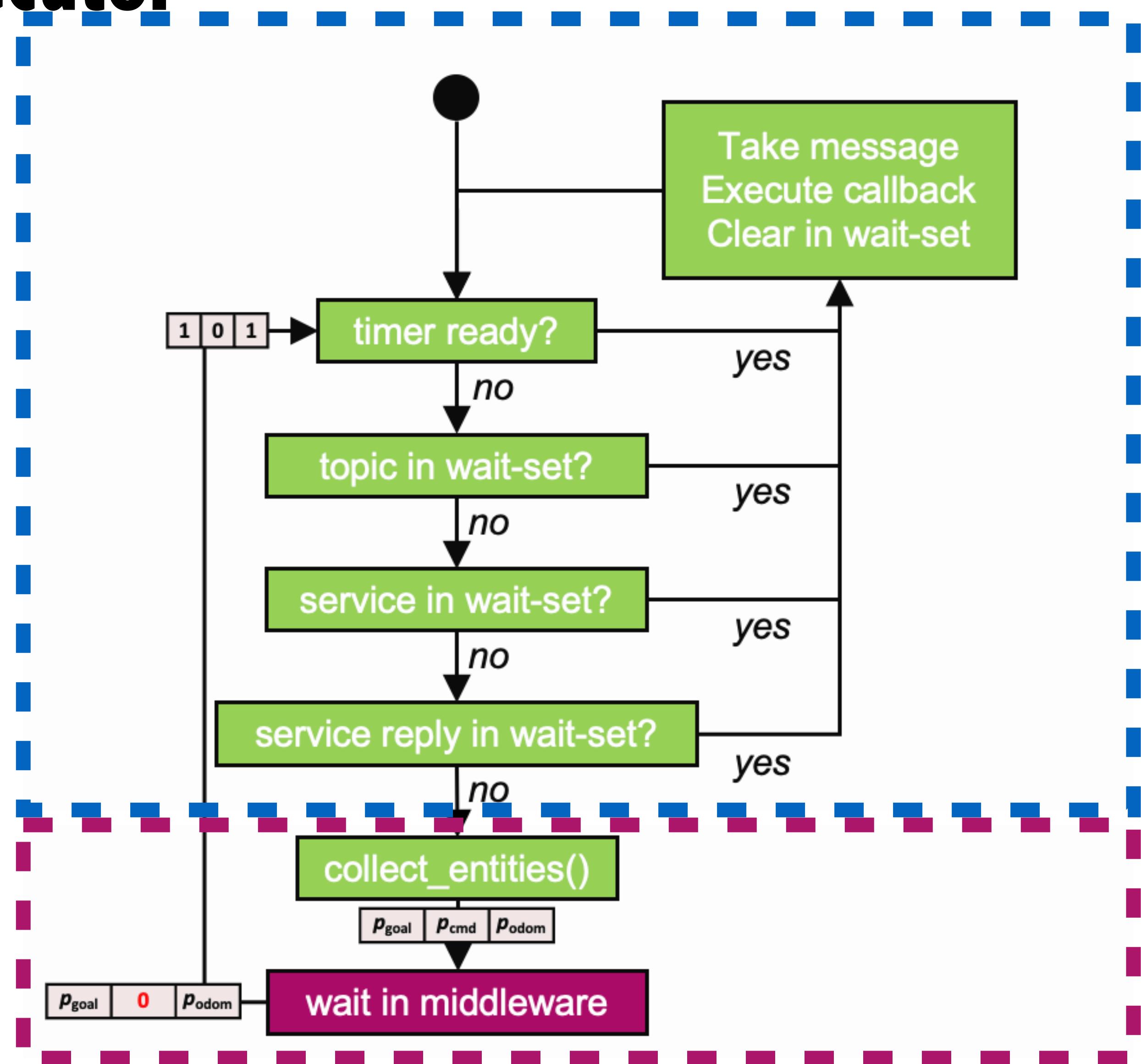


Semántica del ejecutor



Ventana de
Procesamiento

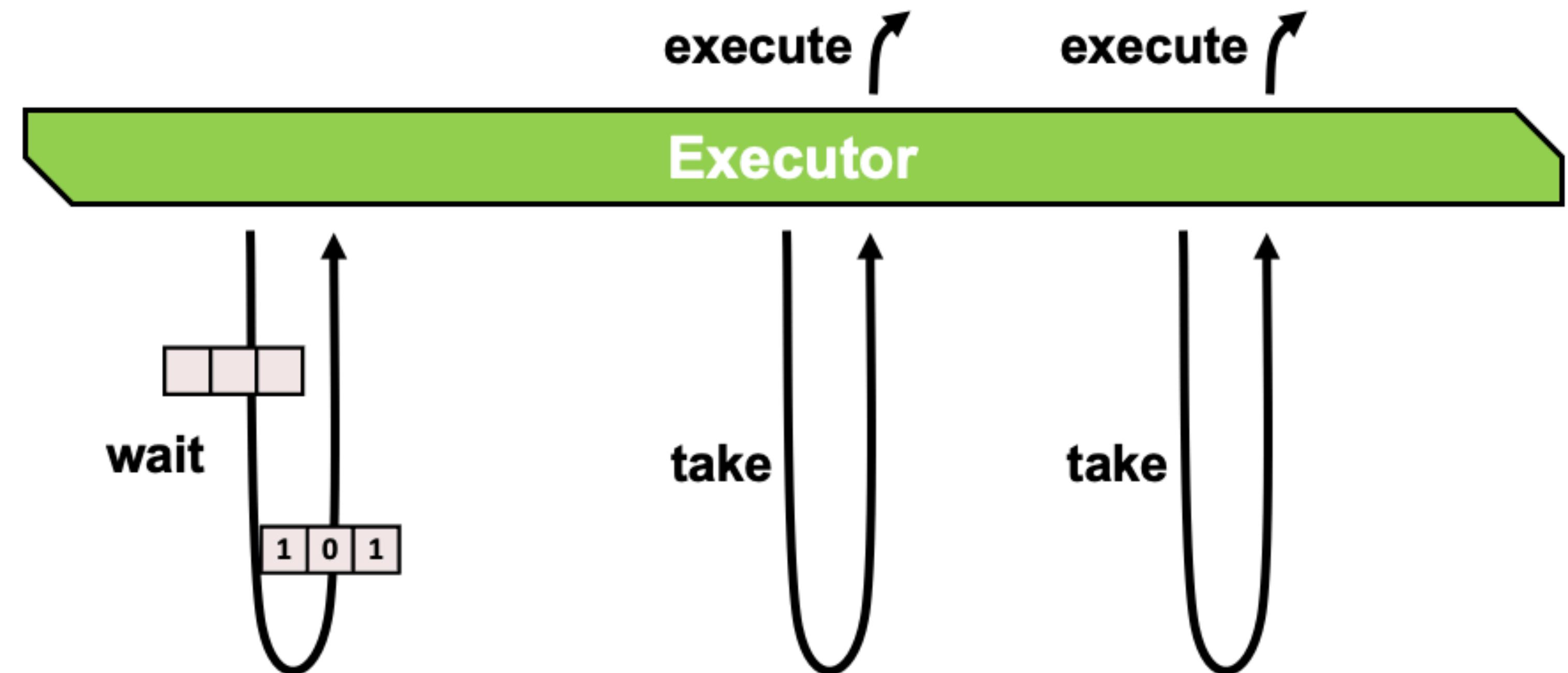
Semántica del ejecutor



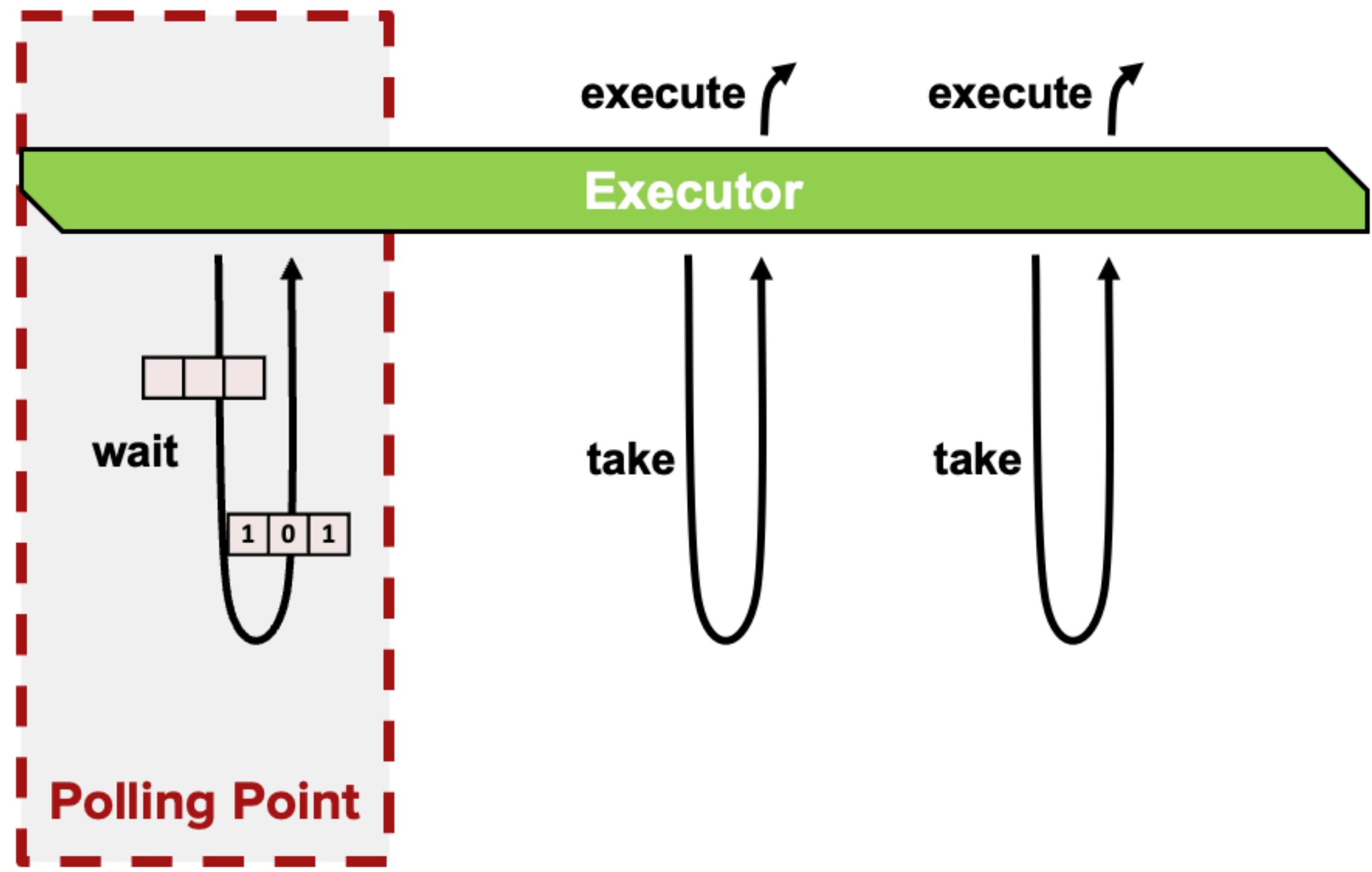
Ventana de
Procesamiento

Punto de Polling

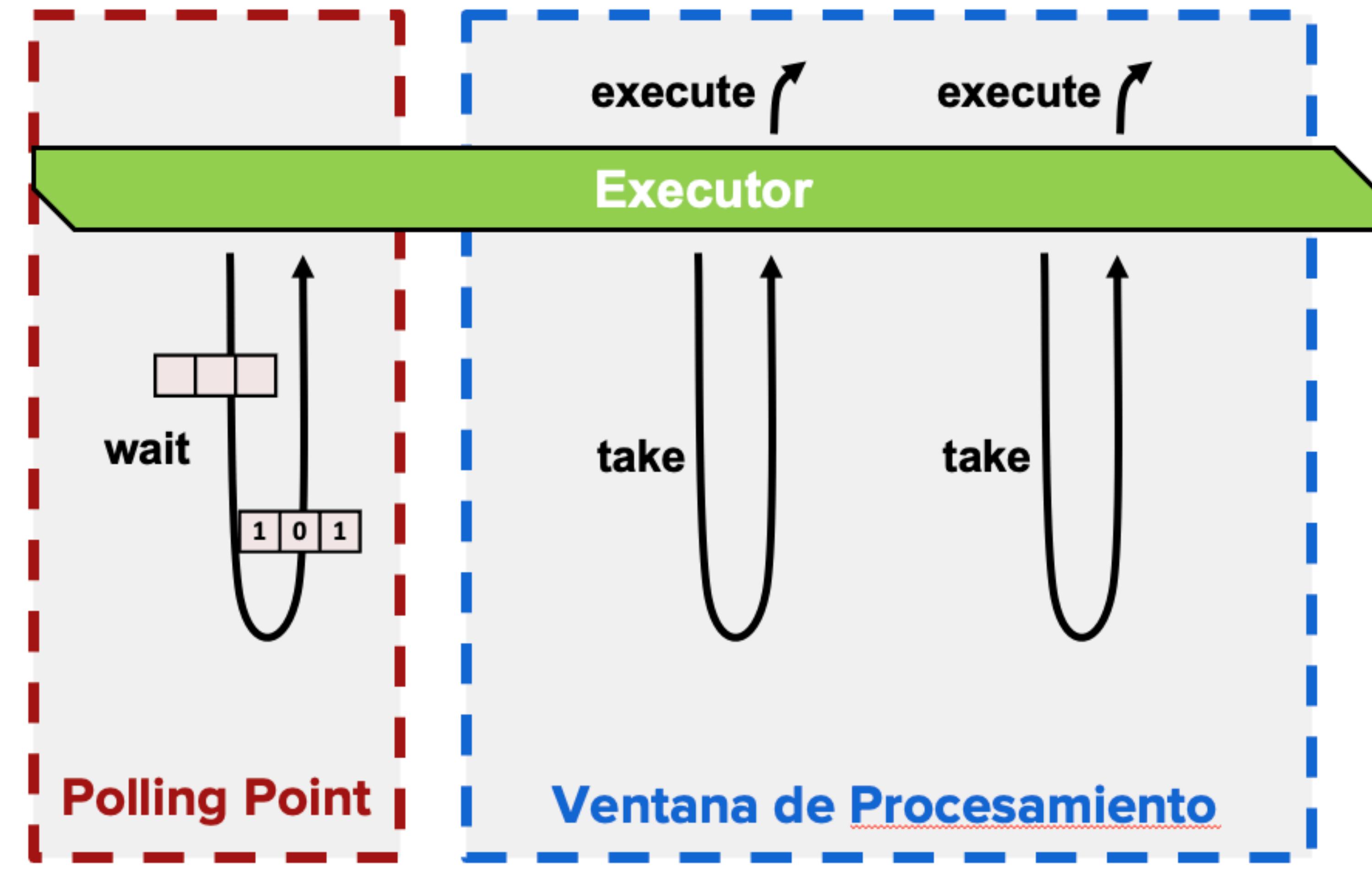
Semántica del Ejecutor en dos pasos



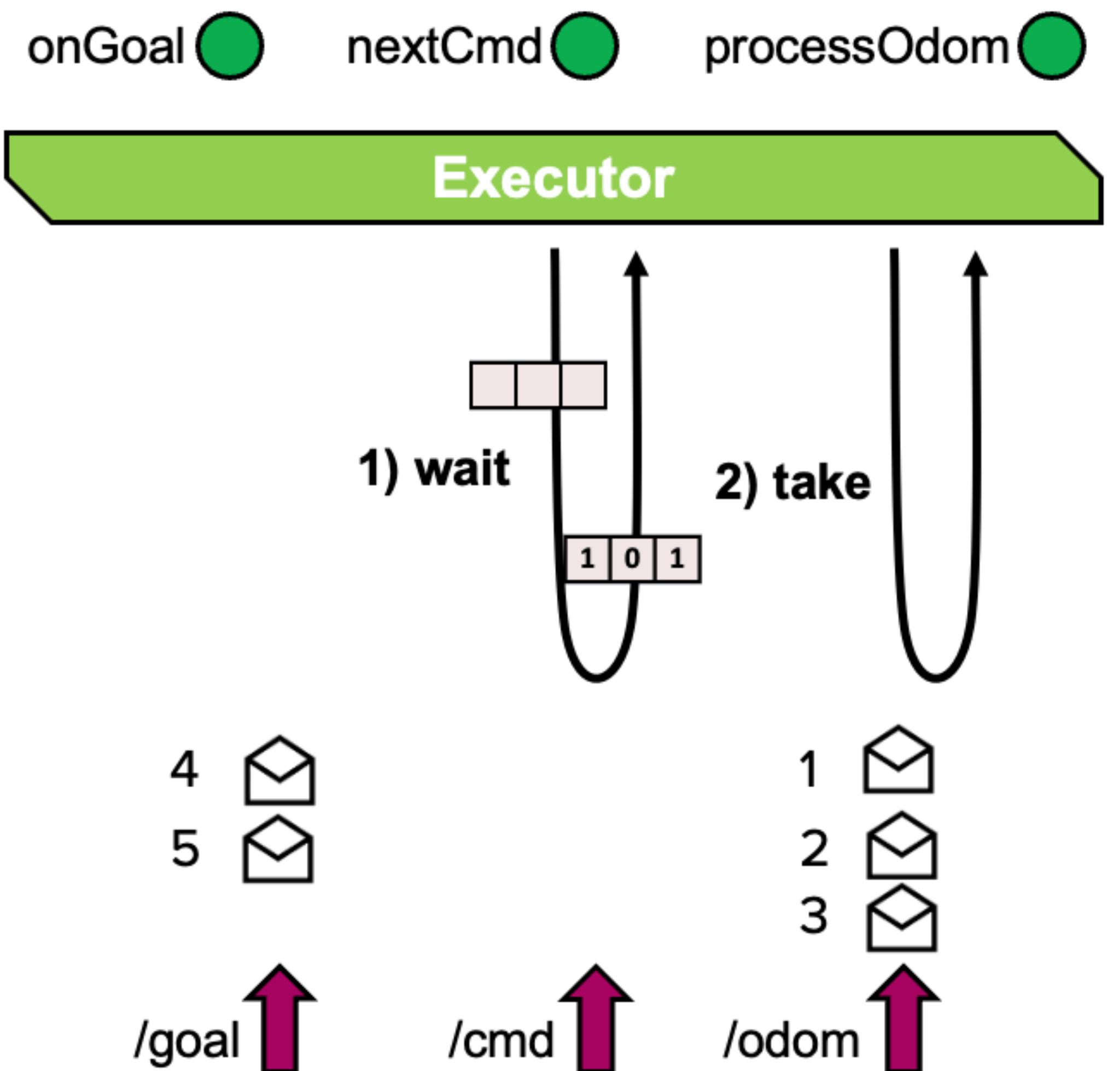
Semántica del Ejecutor en dos pasos



Semántica del Ejecutor en dos pasos

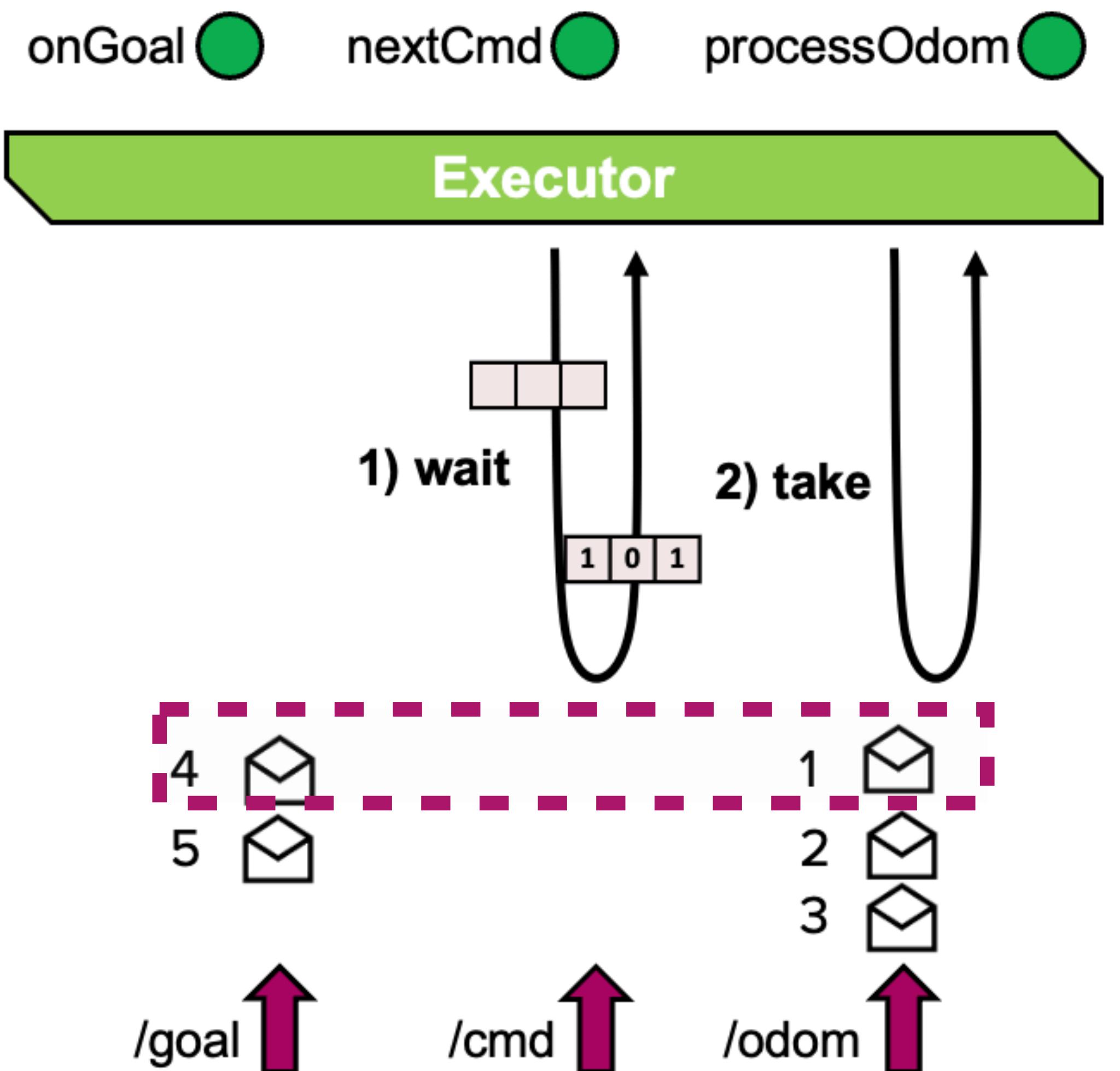


Impacto en el determinismo: procesamiento de mensajes no FIFO



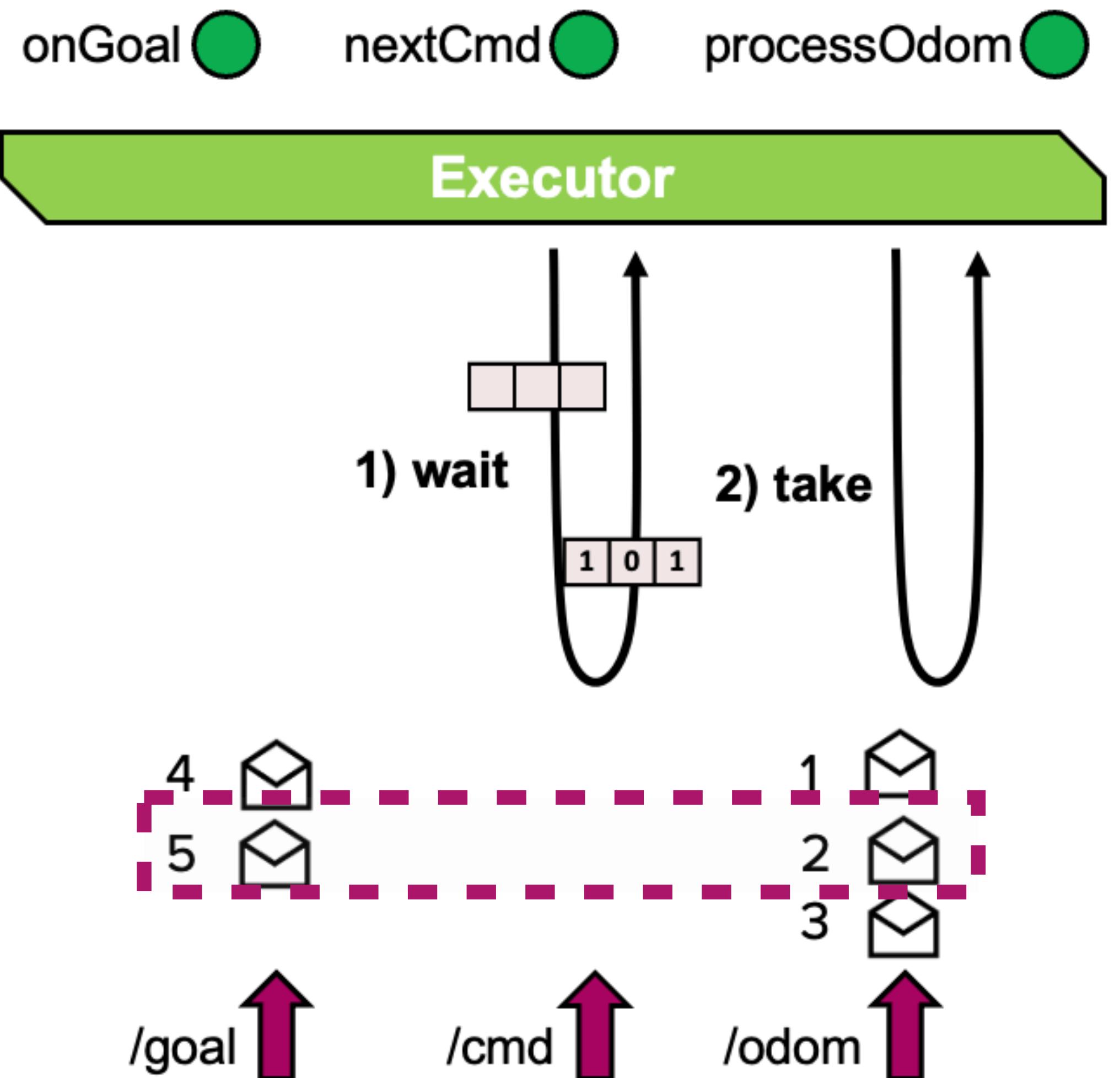
- Todos los mensajes de `/odom` se reciben antes de los mensajes de `/goal`

Impacto en el determinismo: procesamiento de mensajes no FIFO



- PERO: `/odom` (1) y `/goal` (4) se procesan primero
- Toman siempre el “elemento superior” de la cola DDS

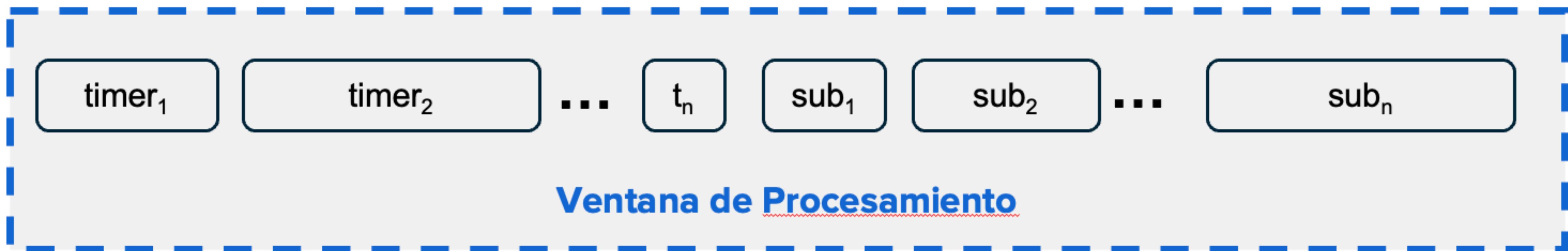
Impacto en el determinismo: procesamiento de mensajes no FIFO



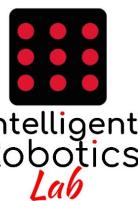
- Después: `/odom` (2) y `/goal` (5)

Impacto en la latencia: orden de procesamiento

Veamos diferentes escenarios...

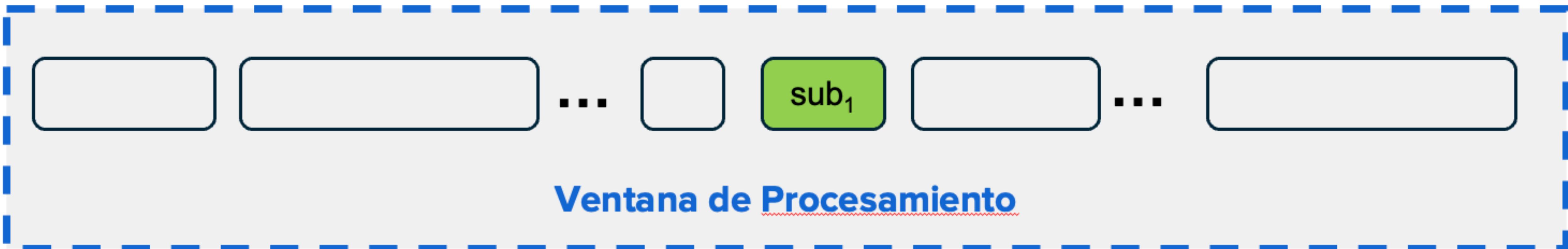


El orden de procesamiento depende ÚNICAMENTE del orden en el que se crearon/agregaron los elementos al Ejecutor.



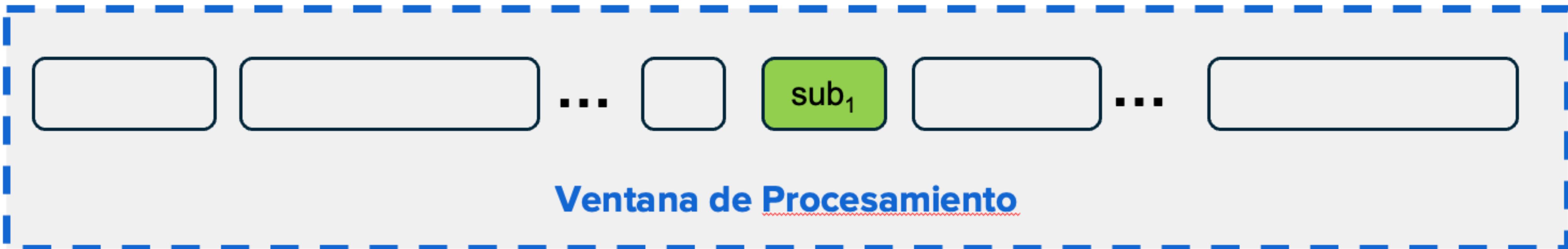
Impacto en la latencia: orden de procesamiento

Una suscripción, sin temporizadores



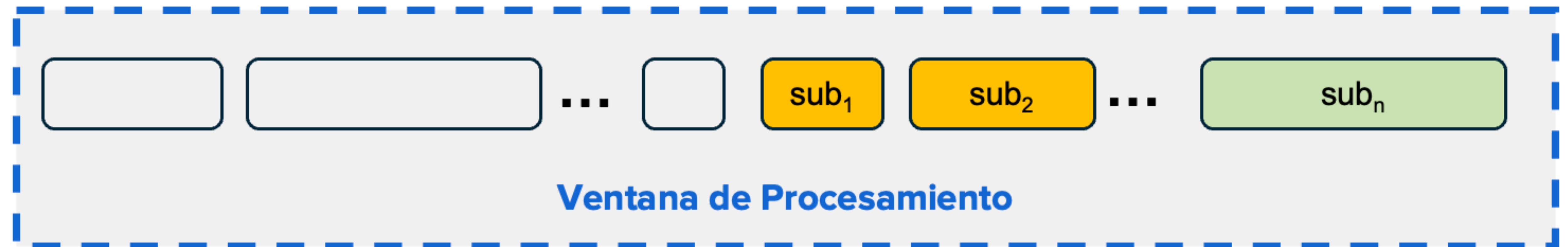
Impacto en la latencia: orden de procesamiento

Una suscripción, sin temporizadores



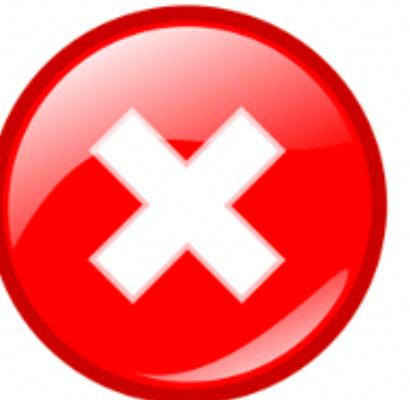
Impacto en la latencia: orden de procesamiento

Sin temporizador, varias otras suscripciones anteriores



Impacto en la latencia: orden de procesamiento

Las suscripciones múltiples también tienen nuevos datos?



Peor caso: todas las demás suscripciones han recibido nuevos datos y se procesan antes => ¡mayor latencia!



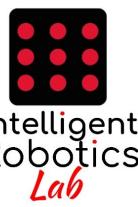
Impacto en la latencia: orden de procesamiento

Temporizador en el pool, sin suscripciones



Impacto en la latencia: orden de procesamiento

Temporizador en el pool, sin suscripciones

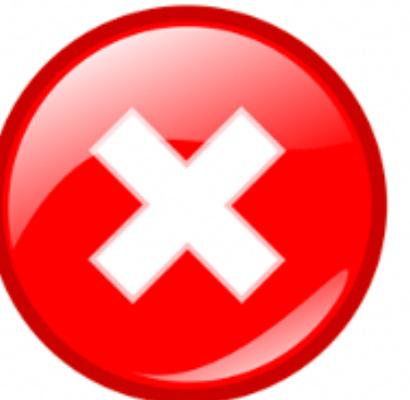


Impacto en la latencia: orden de procesamiento

Temporizador en el pool pero algunas suscripciones más adelante



Impacto en la latencia: orden de procesamiento

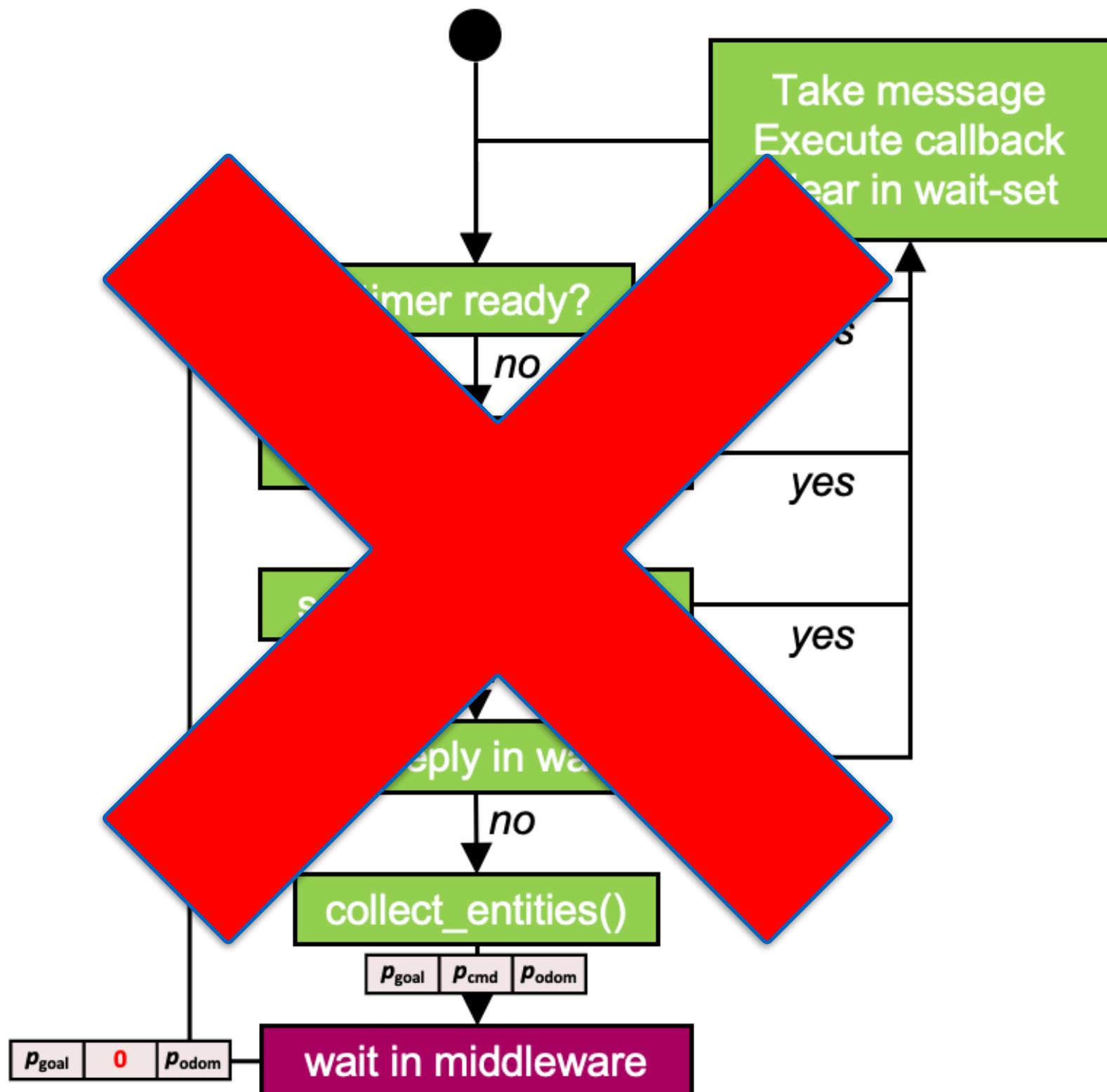


Temporizador en el pool pero algunas suscripciones más adelante



Si el próximo temporizador transcurre durante la ventana de procesamiento =>
No se considera hasta el próximo polling point. => ¡Próximo callback del timer retrasado!

Crea tu propio ejecutor con rclcpp::waitset



```

1 rclcpp::WaitSet wait_set({{{sub1}, {sub2}, {...}}}, {timer1});
2
3 while (rclcpp::ok()) {
4     const auto wait_result = wait_set.wait(2s);
5
6     if (wait_result.kind() == rclcpp::WaitResultKind::Ready) {
7
8         if (wait_result.get_wait_set().get_rcl_wait_set().timers[0U]) {
9             // call timer callback
10        } else if (wait_result.get_wait_set()
11                    .get_rcl_wait_set()
12                    .subscriptions[0U]) {
13            if (sub1->take(msg, msg_info)) {
14                // process message
15            }
16        }
17    }
18 }

```

https://github.com/ros2/examples/tree/rolling/rclcpp/wait_set



Más sobre ejecutores

- **MultiThreadedExecutor**
 - Paraleliza el procesamiento en plataformas multinúcleo
 - Los threads tienen la misma prioridad
 - Mismos problemas de sincronización que SingleThreadedExecutor (basado en el enfoque wait set)
- **StaticSingleThreadedExecutor**
 - Optimiza el coste de tiempo de ejecución para escanear un nodo en términos de suscripciones, temporizadores, servicios...
 - Realiza solo un escaneo cuando se agrega un nodo (otros ejecutores escanean regularmente)
 - Utilízalo solo con nodos que creen todas las suscripciones, temporizadores, etc. durante la inicialización
- **rclc-Executor**
 - C-API diseñada para microcontroladores (micro-ROS)
 - Sin asignación de memoria dinámica en tiempo de ejecución
 - Orden de procesamiento definido por el usuario
 - Otras características deterministas (como condiciones de activación para sincronización)



Más sobre ejecutores

- **MultiThreadedExecutor**
 - Paraleliza el procesamiento en plataformas multinúcleo
 - Los threads tienen la misma prioridad
 - Mismos problemas de sincronización que SingleThreadedExecutor (basado en el enfoque wait set)
- **StaticSingleThreadedExecutor**
 - Optimiza el coste de tiempo de ejecución para escanear un nodo en términos de suscripciones, temporizadores, servicios...
 - Realiza solo un escaneo cuando se agrega un nodo (otros ejecutores escanean regularmente)
 - Utilízalo solo con nodos que creen todas las suscripciones, temporizadores, etc. durante la inicialización
- **rclc-Executor**
 - C-API diseñada para microcontroladores (micro-ROS)
 - Sin asignación de memoria dinámica en tiempo de ejecución
 - Orden de procesamiento definido por el usuario
 - Otras características deterministas (como condiciones de activación para sincronización)



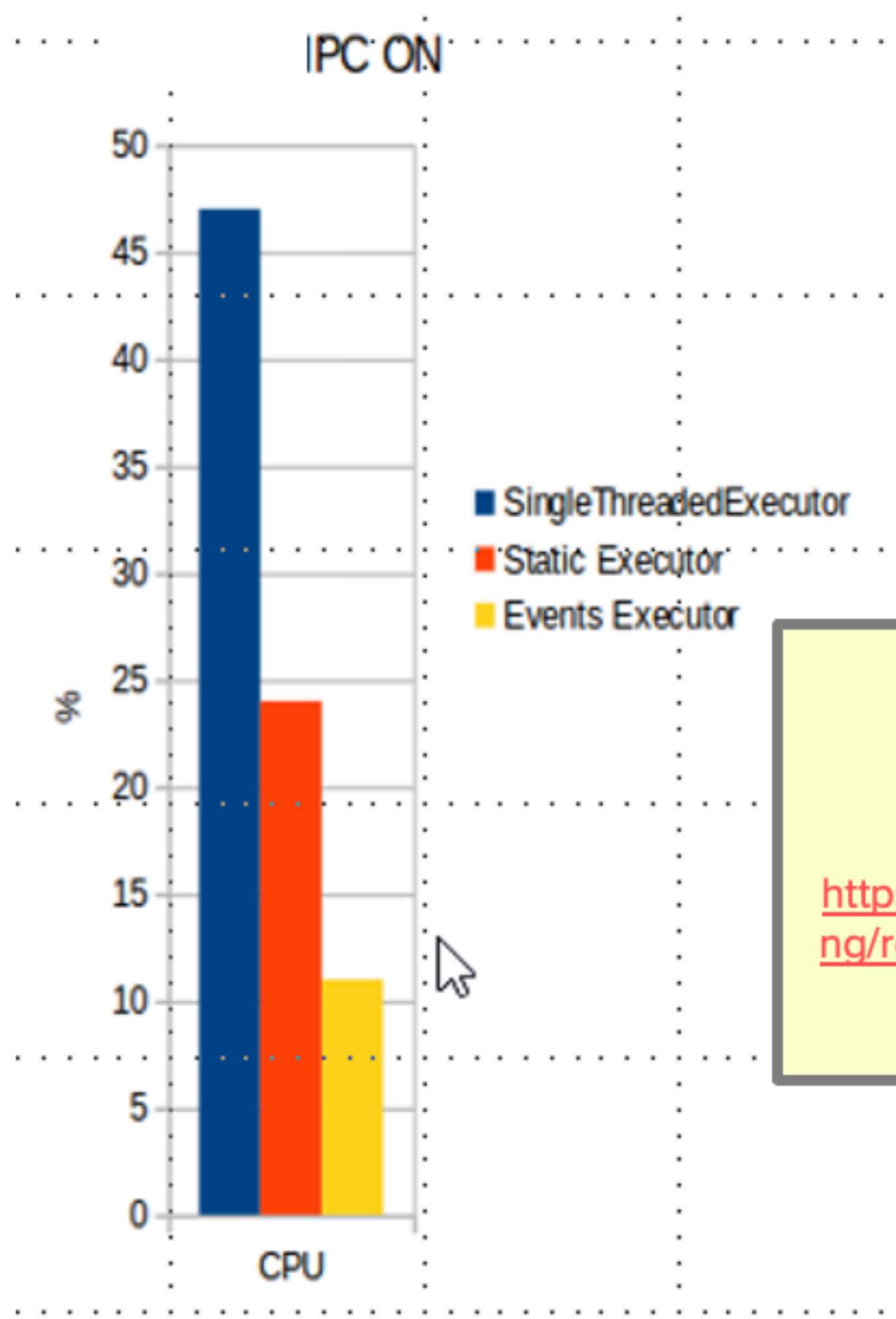
Más sobre ejecutores

- **MultiThreadedExecutor**
 - Paraleliza el procesamiento en plataformas multinúcleo
 - Los threads tienen la misma prioridad
 - Mismos problemas de sincronización que SingleThreadedExecutor (basado en el enfoque wait set)
- **StaticSingleThreadedExecutor**
 - Optimiza el coste de tiempo de ejecución para escanear un nodo en términos de suscripciones, temporizadores, servicios...
 - Realiza solo un escaneo cuando se agrega un nodo (otros ejecutores escanean regularmente)
 - Utilízalo solo con nodos que creen todas las suscripciones, temporizadores, etc. durante la inicialización
- **rclc-Executor**
 - C-API diseñada para microcontroladores (micro-ROS)
 - Sin asignación de memoria dinámica en tiempo de ejecución
 - Orden de procesamiento definido por el usuario
 - Otras características deterministas (como condiciones de activación para sincronización)



Events Executor

- Rendimiento mejorado
 - Sin creación de wait_set
 - API de escucha RMW
- Procesamiento FIFO
 - Cola de eventos
- Administrador de temporizadores
 - Independiente de DDS
- Recopilador de entidades de ejecutor
 - Como StaticSingleThreadedExecutor

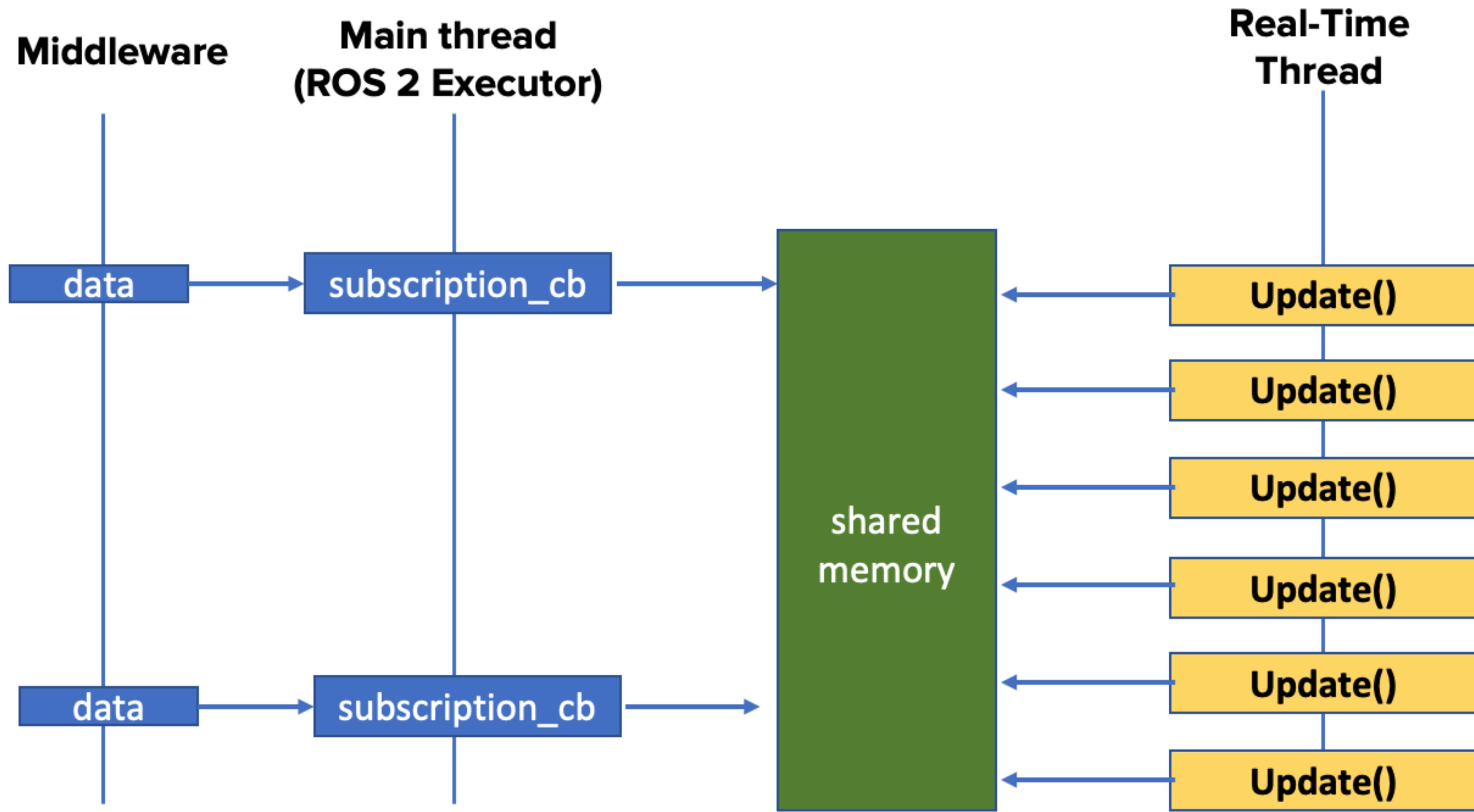


Merged in Rolling
(04/2023)

https://github.com/ros2/rclcpp/tree/rolling/rclcpp/src/rclcpp/experimental/executors/events_executor

Executor Workshop'21: <https://www.apex.ai/roscon-21>

ros2_control framework

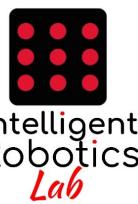


Comparación de ejecutores

Executor	DDS exchange	FIFO processing?	Improved performance?	Real-time?
SingleThreaded Executor	Wait Set			
StaticSingleThreaded Executor	Wait Set			
MultiThreaded Executor	Wait Set			
rclc-Executor	Wait Set			
Events Executor	Listener			

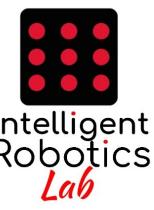
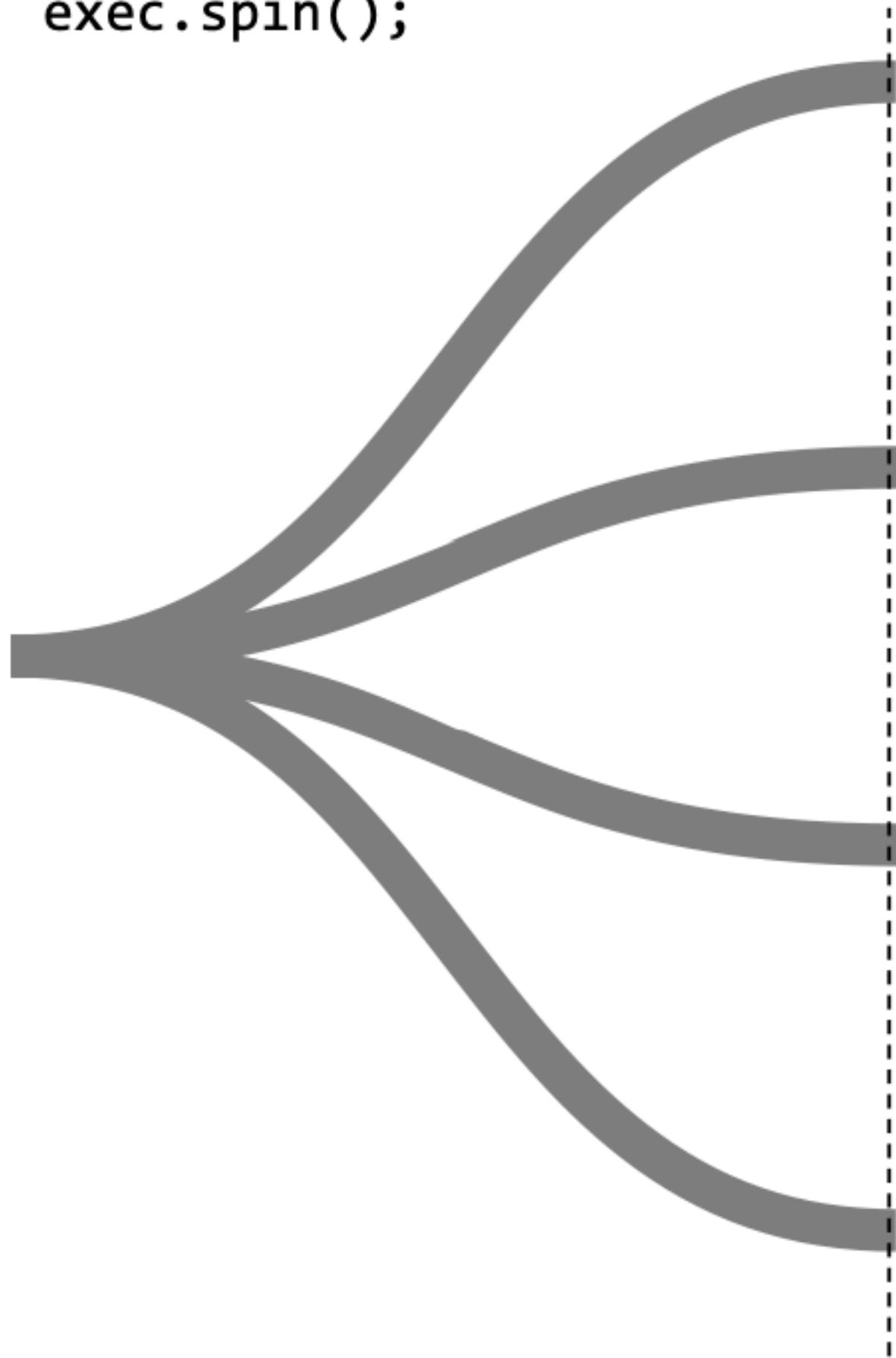
Ejecutor de Workshop en Tiempo Real

- Diseñado específicamente para este Workshop
- Proporciona una API para establecer la prioridad de las suscripciones
- Construido sobre un ejecutor multiproceso
- Utiliza el planificador en Tiempo Real existente del SO
 - SCHED_FIFO



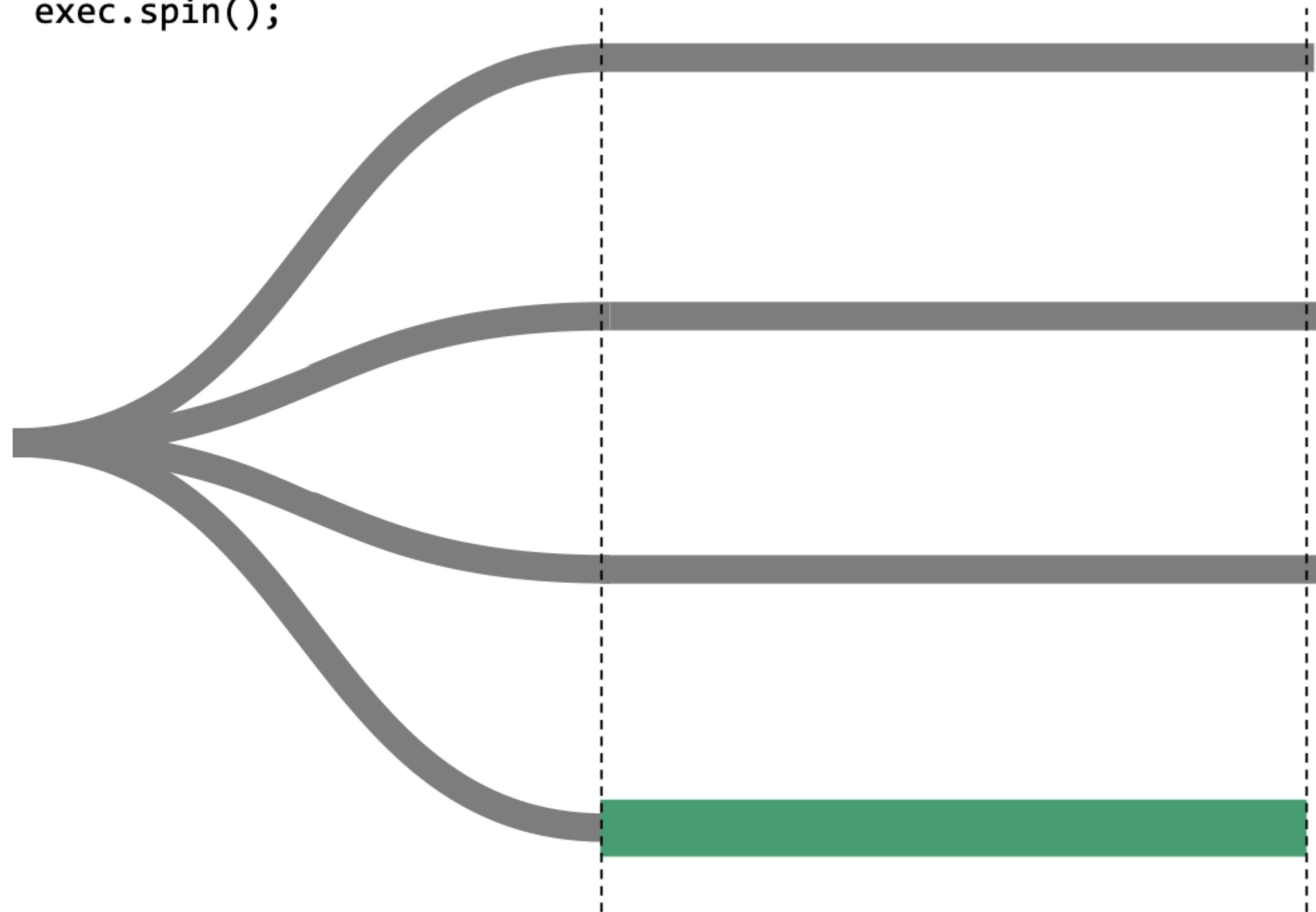
Ejecutor de Workshop en Tiempo Real

```
rclcpp::executor::MultithreadedExecutor exec;  
exec.spin();
```



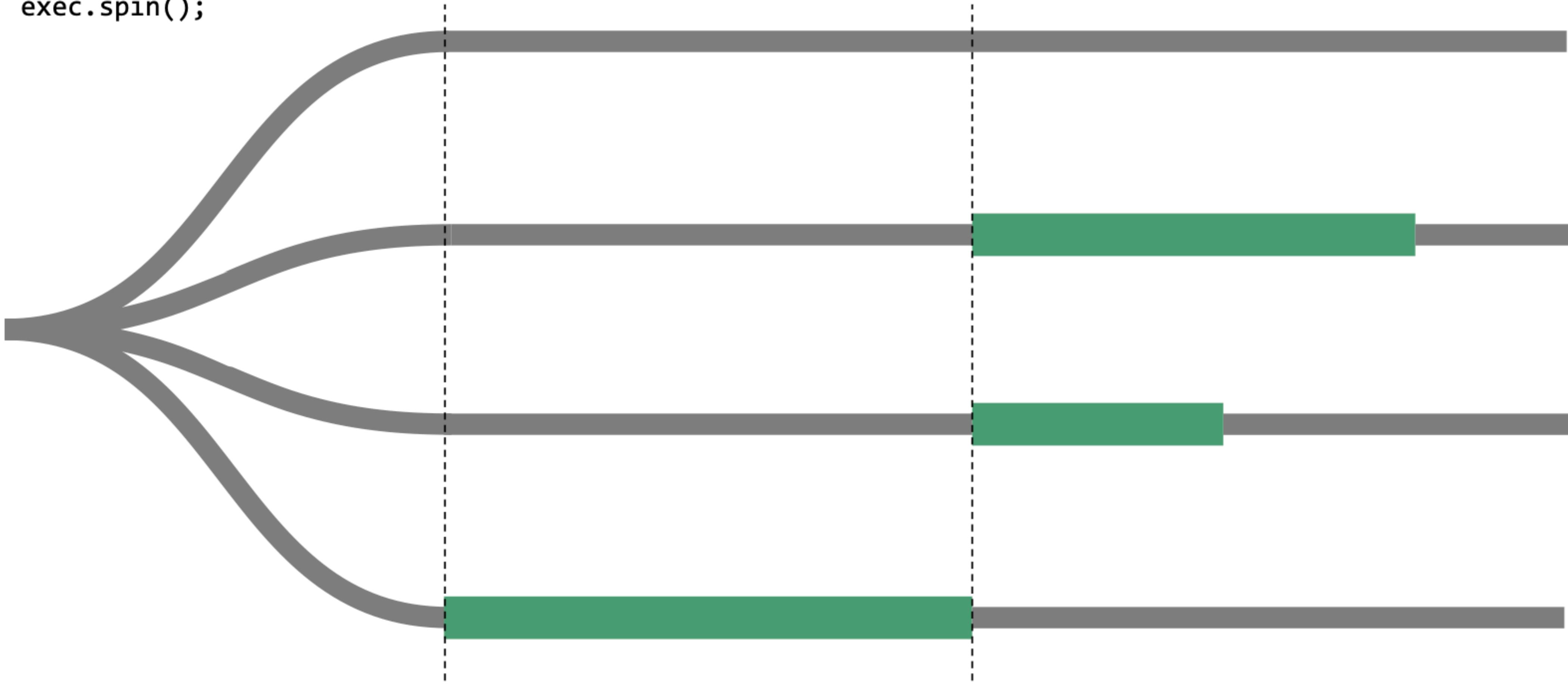
Ejecutor de Workshop en Tiempo Real

```
rclcpp::executor::MultithreadedExecutor exec;  
exec.spin();
```



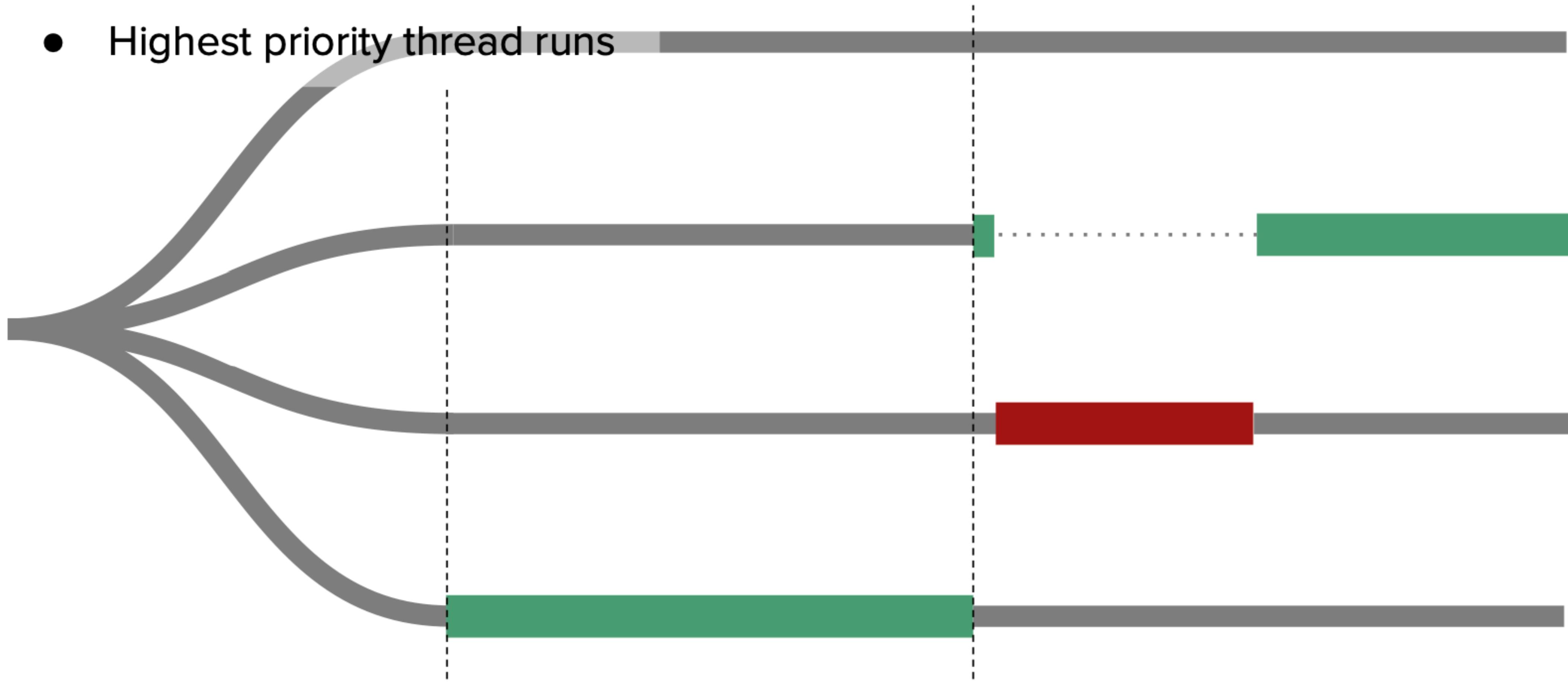
Ejecutor de Workshop en Tiempo Real

```
rclcpp::executor::MultithreadedExecutor exec;  
exec.spin();
```



Ejecutor de Workshop en Tiempo Real

- $\# \text{ threads} > \# \text{ cores}$
- Highest priority thread runs



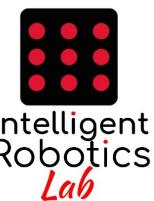
Diseño de ejecutor de Workshop en Tiempo Real

```
class ObjectDetectingNode : rclcpp::Node {
    rclcpp::Publisher<sensor_msgs::msg::BoundingBoxes>::SharedPtr
        image_pub_;
    rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr
        image_sub_;

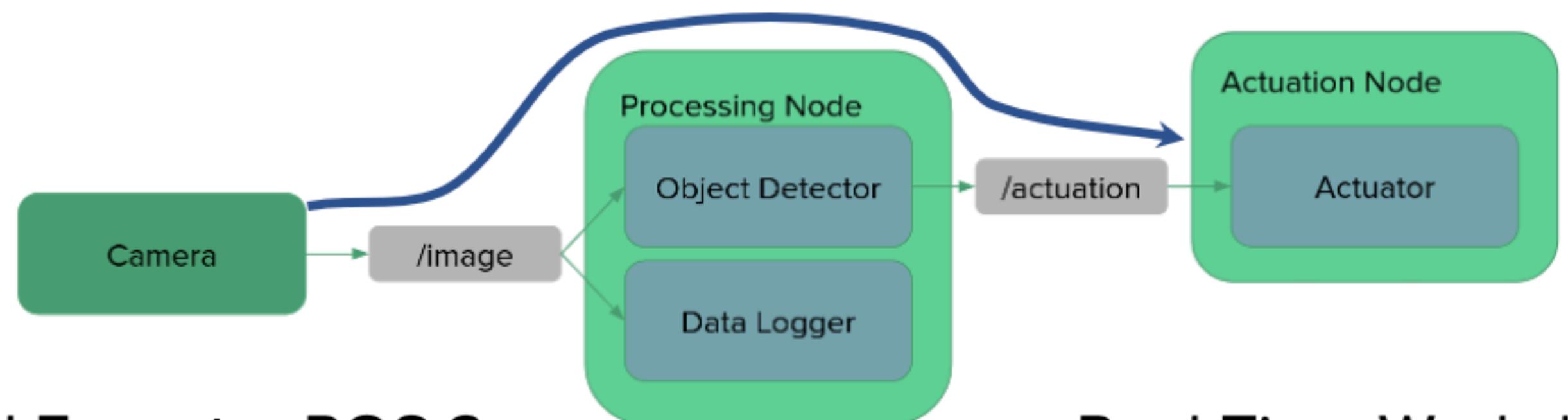
    void image_callback(const sensor_msgs::msg::Image::SharedPtr msg);
};
```

```
image_sub_ = this->create_subscription<sensor_msgs::msg::Image>(
    "image_raw", 10, std::bind(&ObjectDetectingNode::image_callback, this, _1));
```

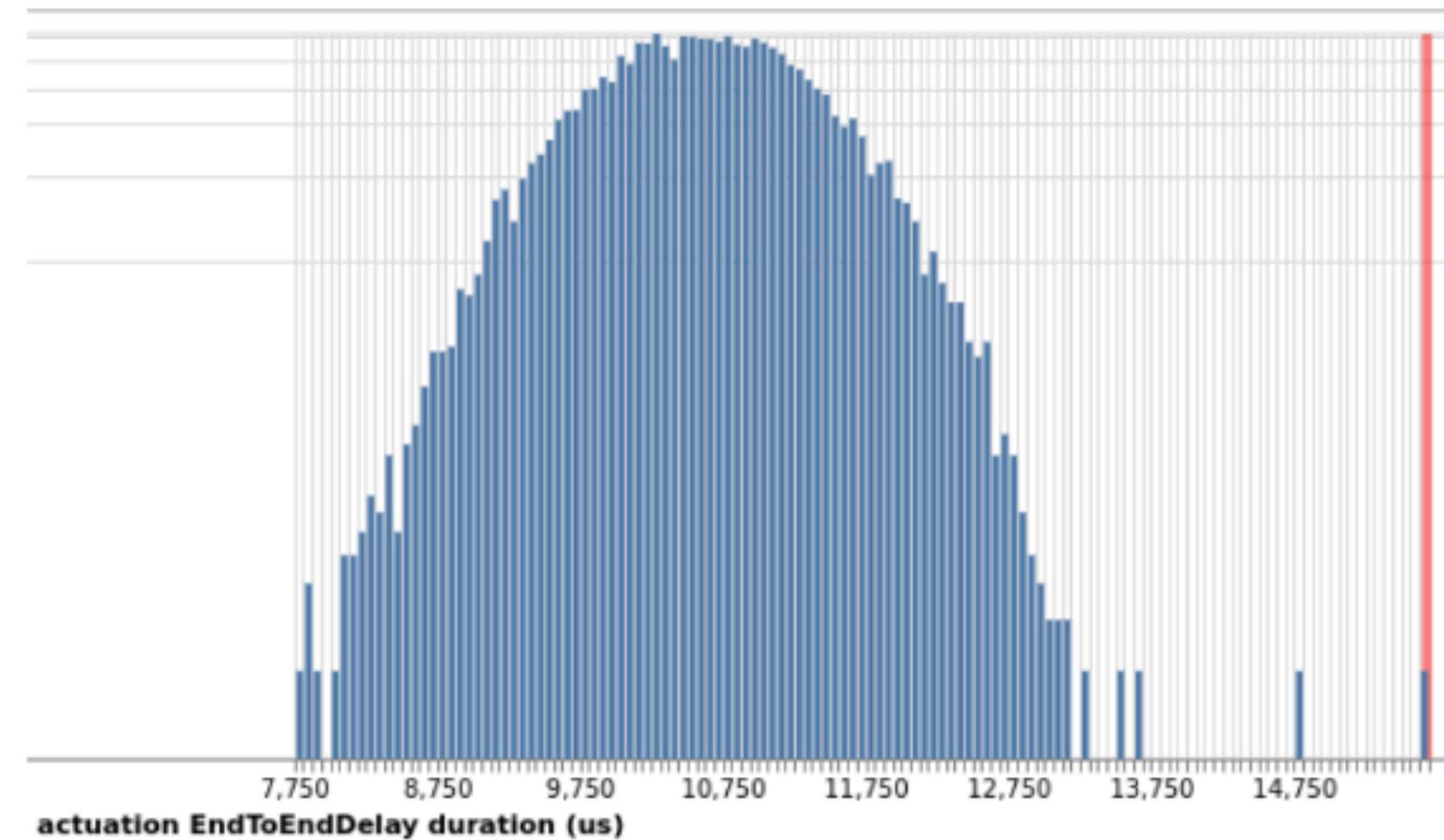
```
sched_param sp;
sp.sched_priority = HIGH;
image_sub_->sched_param(SP);
```



Ejecutor de Workshop en Tiempo Real: resultados de latencia de extremo a extremo

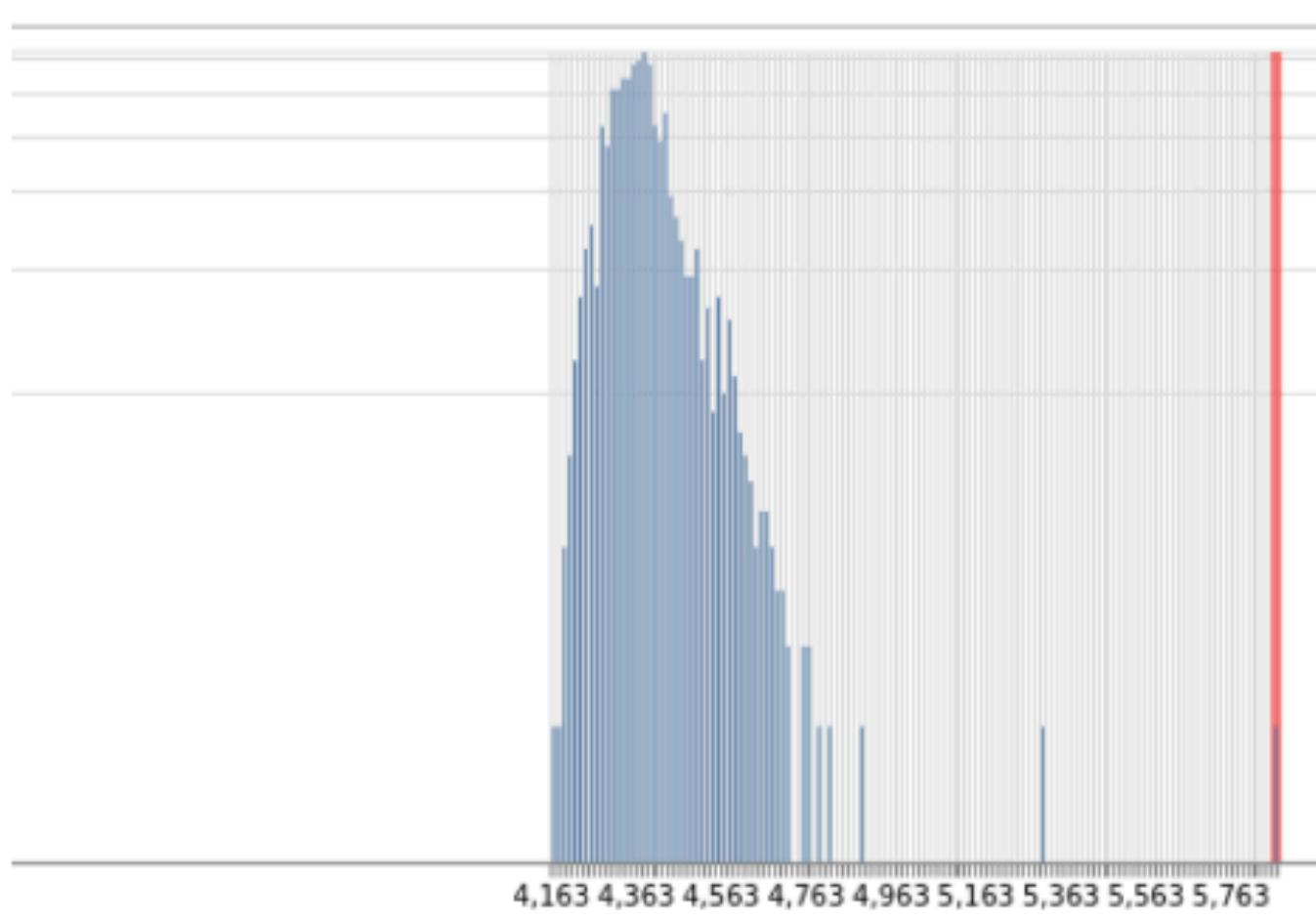


Multi-Threaded Executor ROS 2



High latency

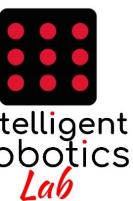
Real-Time Workshop Executor



Low latency

Pros /Cons

- Corrige el retraso ilimitado con el orden first-in-first-out
- El usuario puede configurar las prioridades
- callbacks mutuamente excluyentes
 - Son buenas para evitar condiciones de carrera
 - Un mejor ejecutor tendría una semántica más específica para aplicar secciones críticas del código



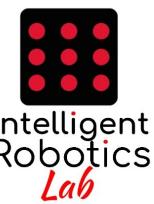
Trabajos relacionados

- Blaß, Tobias, et al. "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance." 2021 IEEE Real-Time Systems Symposium (RTSS). IEEE, 2021.
- Staschulat, Jan, Ralph Lange, and Dakshina Narahari Dasari. "Budget-based real-time executor for micro-ROS." arXiv preprint arXiv:2105.05590 (2021).
- Choi, Hyunjong, Yecheng Xiang, and Hyoseung Kim. "PiCAS: New design of priority-driven chain-aware scheduling for ROS2." 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2021.
- Blass, Tobias, et al. "Automatic latency management for ros 2: Benefits, challenges, and open problems." 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2021.
- reference_system, Executor Workshop, ROS world 2021: <https://www.apex.ai/roscon-21>
- Events Executor: <https://github.com/irobot-ros/events-executor>

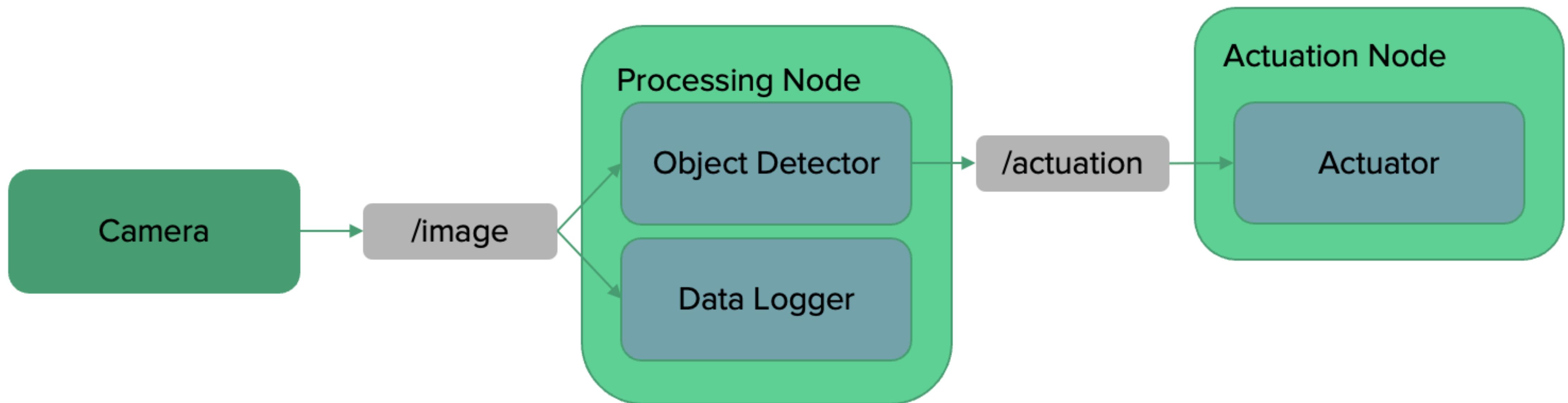


Ejercicio práctico 3:

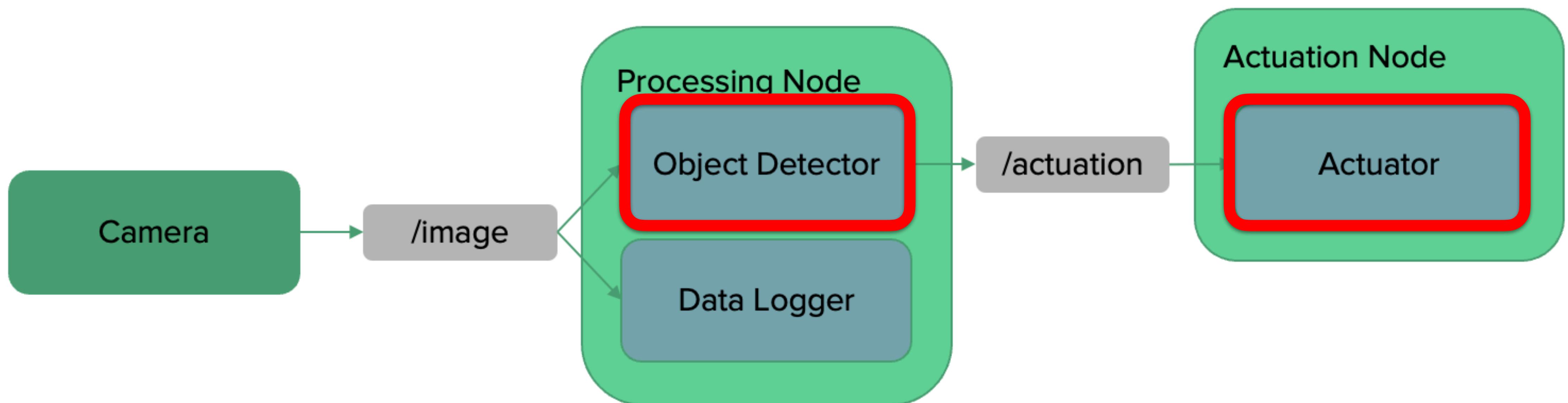
Ejecutor en Tiempo Real



Aplicación de ejemplo

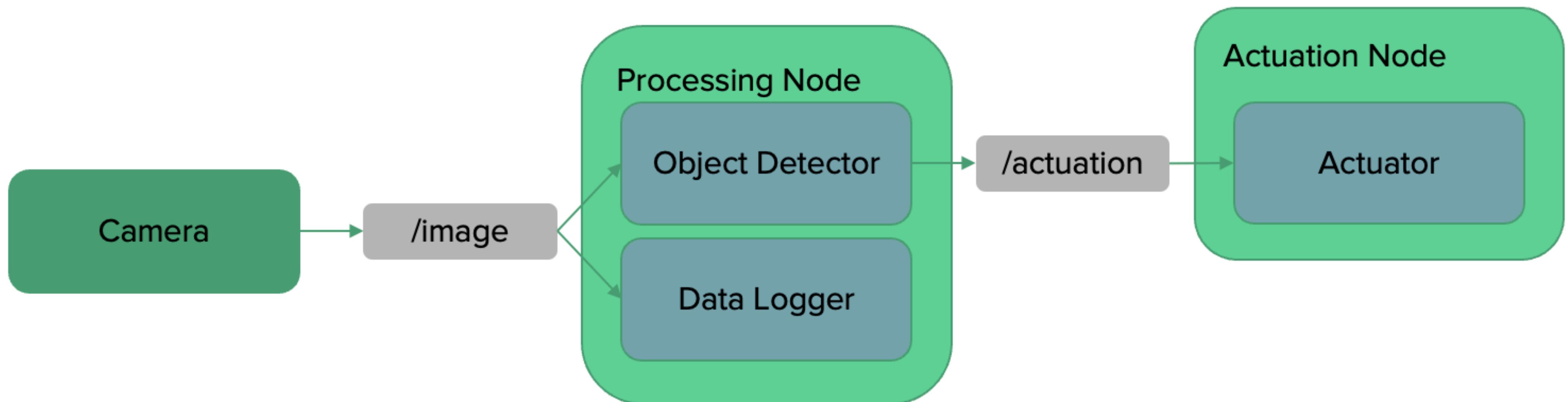


Tiempo Real vs. Best-Effort



Ejercicio 3-1 Objetivos

- Ejecutar la aplicación sin prioridades
- Observar el orden de los callbacks



Ejercicio 3-1

- Abre src/camera_demo/src/main.cc
- Declare el ejecutor y agregue nodos

```
24 // TODO: Create executor, add two nodes to it, and call
25 //           Find example code here: https://docs.ros.org/en/foxy/api/rclcpp/html/executor.html
26
27 rclcpp::executors::SingleThreadedExecutor executor;
28 executor.add_node(camera_processing_node);
29 executor.add_node(actuation_node);
30 executor.spin();
31
32 rclcpp::shutdown();
33 StopTracing();
```

cocker/shell

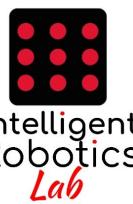
cd /code/exercise3-1

colcon build

Ejercicio 3-1: Compilar y ejecutar

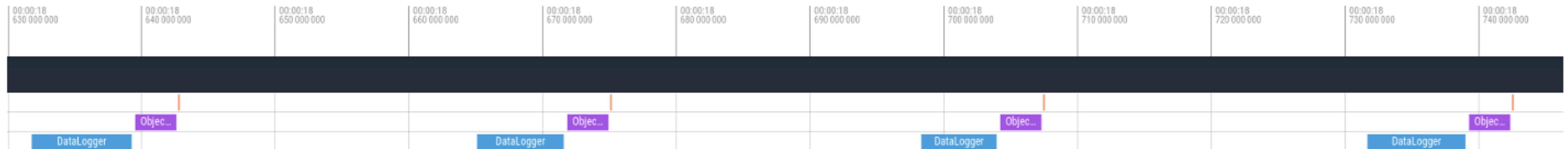
1.(Si aún no lo has hecho) cd /code/exercise 3.1
colcon build

2./run.sh # Ctrl+C after ~20sec



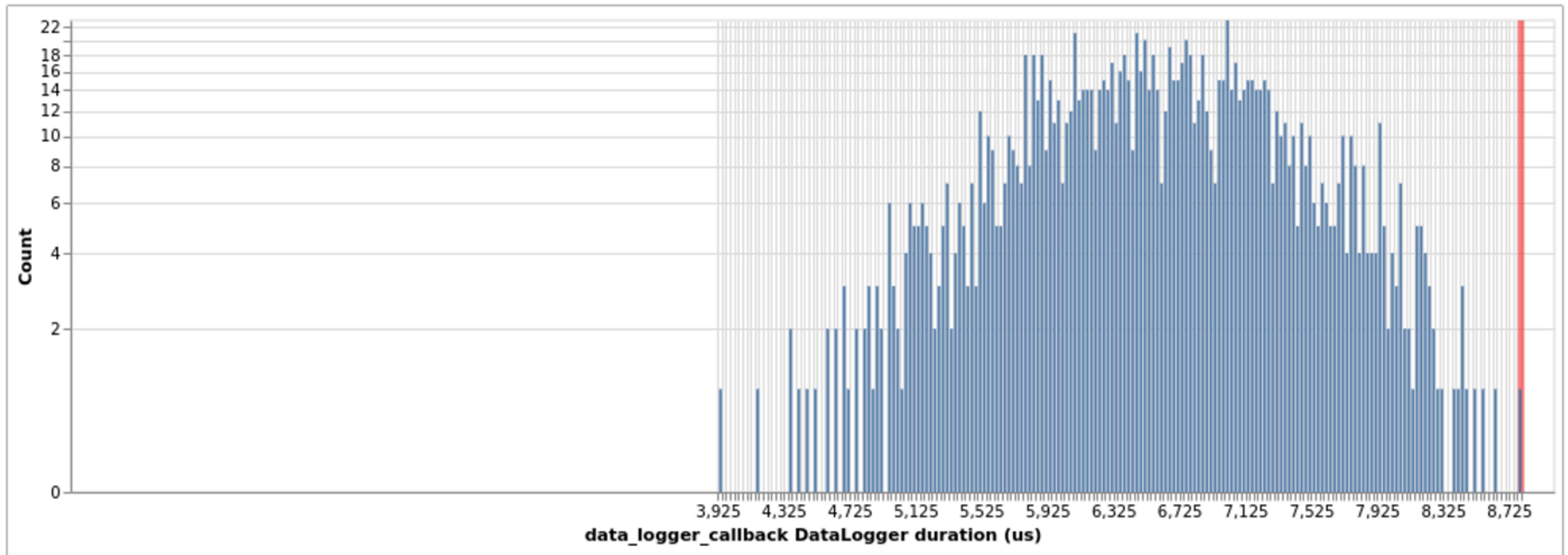
Visualizar traza

- Sube el ejercicio 3-1.perfetto a la herramienta de visualización
 - O utiliza el archivo proporcionado previamente en results/solution.perfetto
- Amplía
- El Data Logger se ejecuta antes que ObjectDetector



Visualizar latencias - DataLogger

Select a slice: data_logger_callback ▾ DataLogger ▾ us ▾

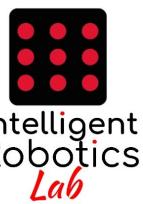
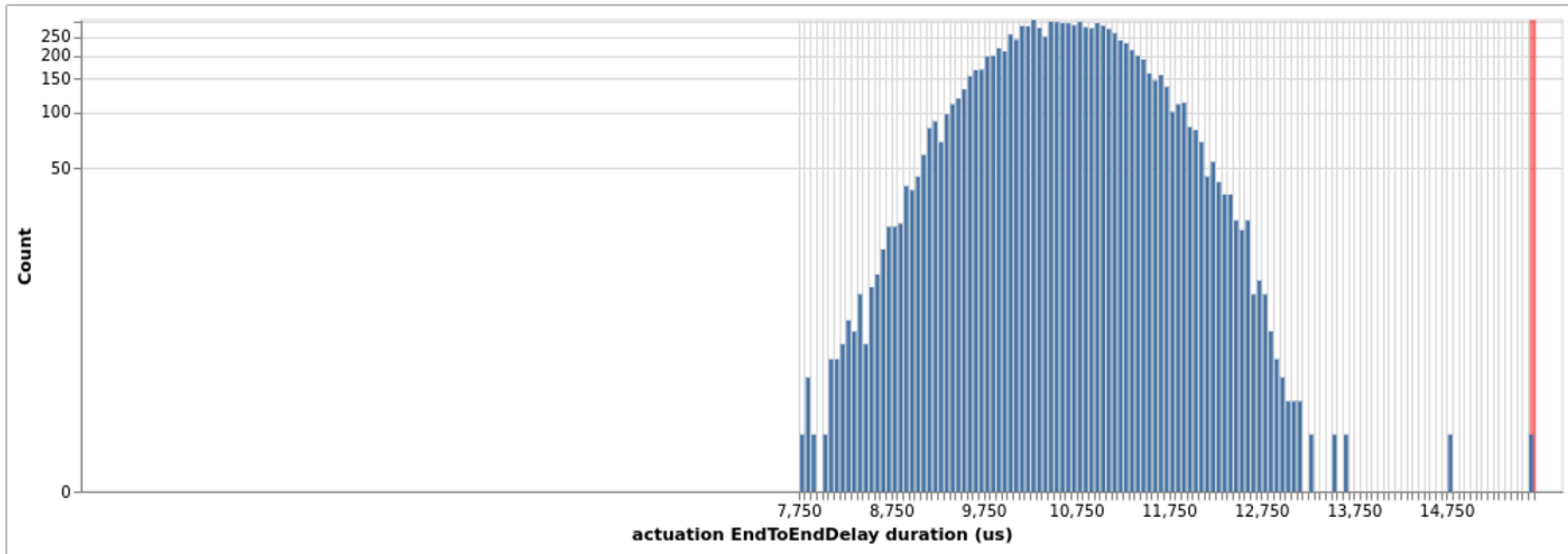


Visualizar latencias - Object Detector



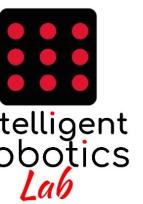
Visualizar latencias - Latencia End to End

Select a slice: actuation ▾ EndToEndDelay ▾ us ▾



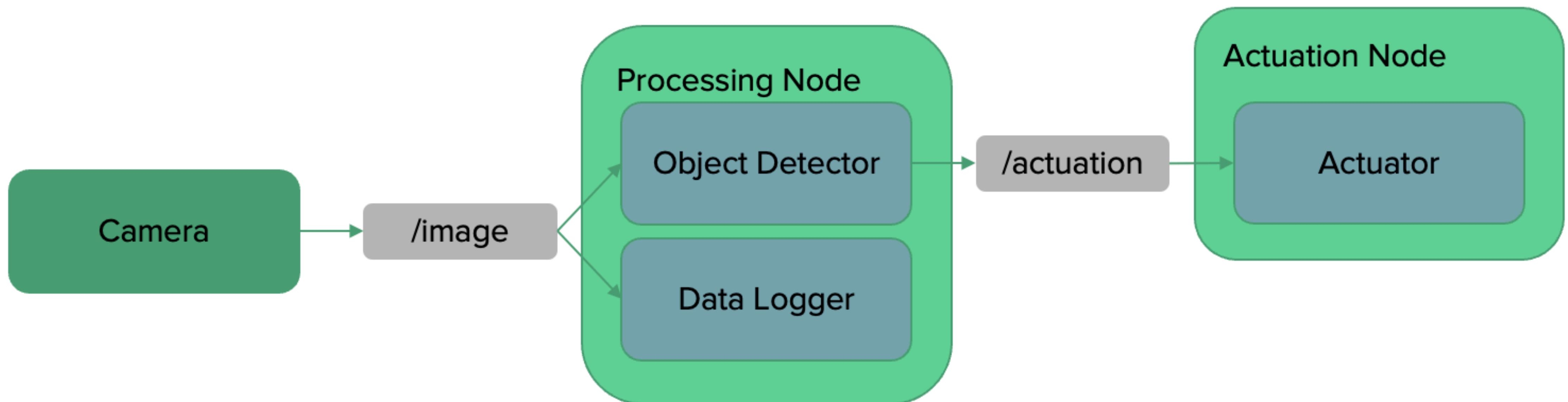
Ejercicio 3-2: Ejecutor en Tiempo Real

- No se puede especificar la prioridad en el ejecutor original
- El ejercicio 3-2 modifica el ejecutor ros2 para permitir establecer prioridades
- El ejecutor rudimentario en Tiempo Real también es robusto frente a un sistema estresado



Ejercicio 3-2 Objetivos

- Ejecutar la aplicación con prioridades
- Observar el orden de los callbacks
- Observar la latencia de la parte en Tiempo Real



- Cerrar todos los archivos fuente del Ejercicio 3-1

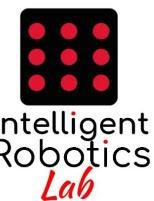
En el workspace del ejercicio 3-2

- # Abre src/camera_demo/src/main.cc
- # Agrega los cambios del ejercicio 3-1
- # Usa un ejecutor multithread
- # Permite que el logger de datos y el detector de objetos actúen de forma independiente
- # Abre src/camera_demo/src/application_nodes.cc
- # Busca el constructor CameraProcessingNode
- # Establece el detector de objetos como prioridad alta
- # El registrador de datos está configurado como prioridad baja de manera predeterminada

```
25 // TODO: Copy your solution from Exercise 3-1, but use rclcpp::executors::MultiThreadedExecutor instead of rclcpp::executor::SingleThreadedExecutor
26
27 rclcpp::executors::MultiThreadedExecutor executor;
28 executor.add_node(camera_processing_node);
29 executor.add_node(actuation_node);
30 executor.spin();
31
32 rclcpp::shutdown();
33 StopTracing();
```

```
39 publisher_ = this->create_publisher<std_msgs::msg::String>("object_detector_publisher", 10);
40
41 // TODO: Set the object detecting subscription to
42 //        // Look at the actuation node for example code
43 //        // sched_param sp;
44 //        sp.sched_priority = HIGH;
45 subscription_object_detector_->sched_param(sp);
46 }
```

```
cd /code/exercise3-2
colcon build #Warnings are expected in rclcpp
```



Ejercicio 3-2: Compilar y ejecutar

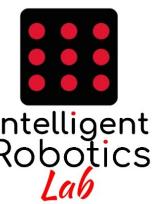
En una terminal

- 1.(Si aún no lo has hecho) cd /code/exercise 3.2
- 2.colcon build

4./run.sh # Ctrl+C after ~20sec

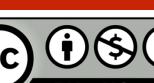
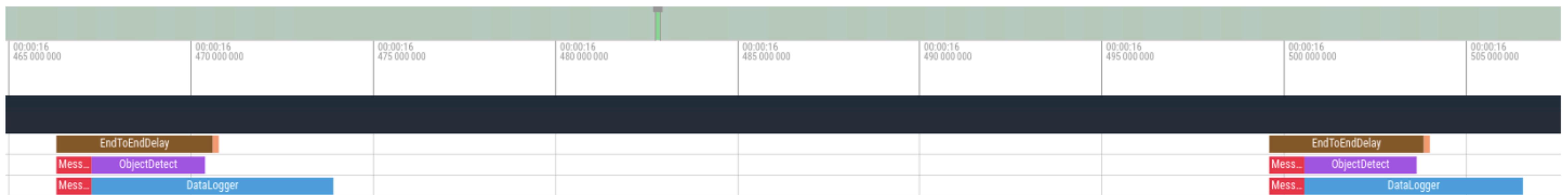
En otra terminal

- 3.docker/shell
/code/stress.sh



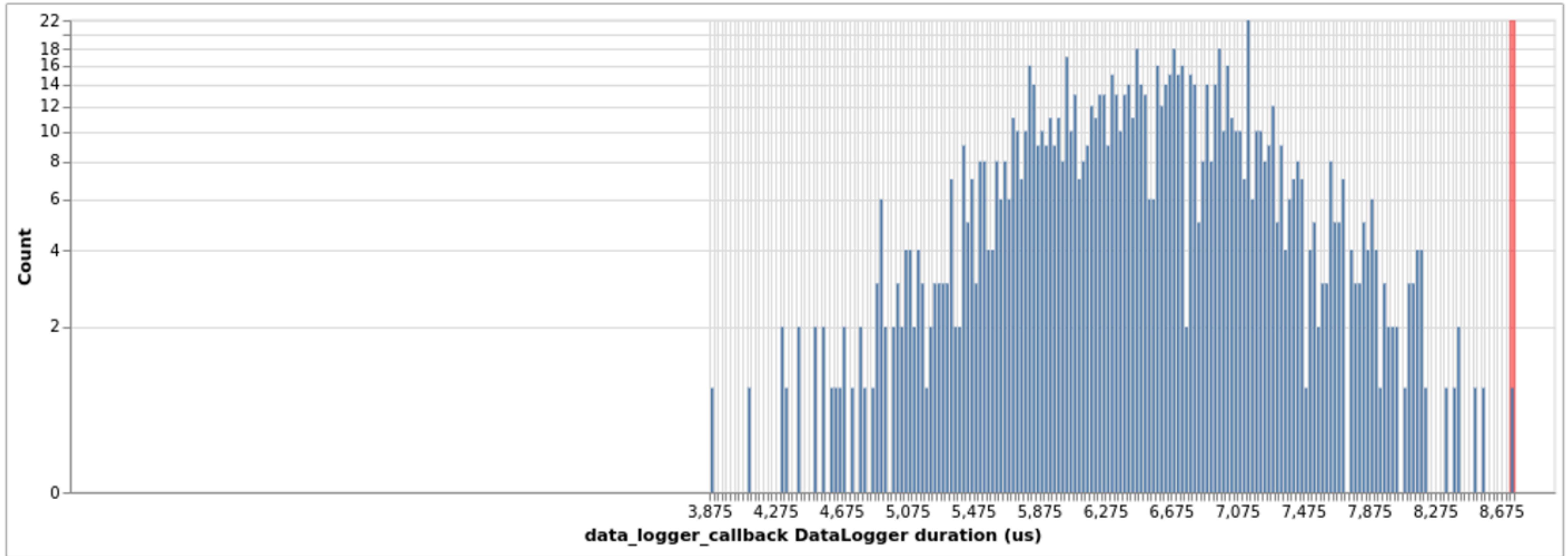
Visualizar traza

- Sube el ejercicio 3-2.perfetto a la herramienta de visualización
 - O utiliza el archivo proporcionado previamente en results/solution.perfetto



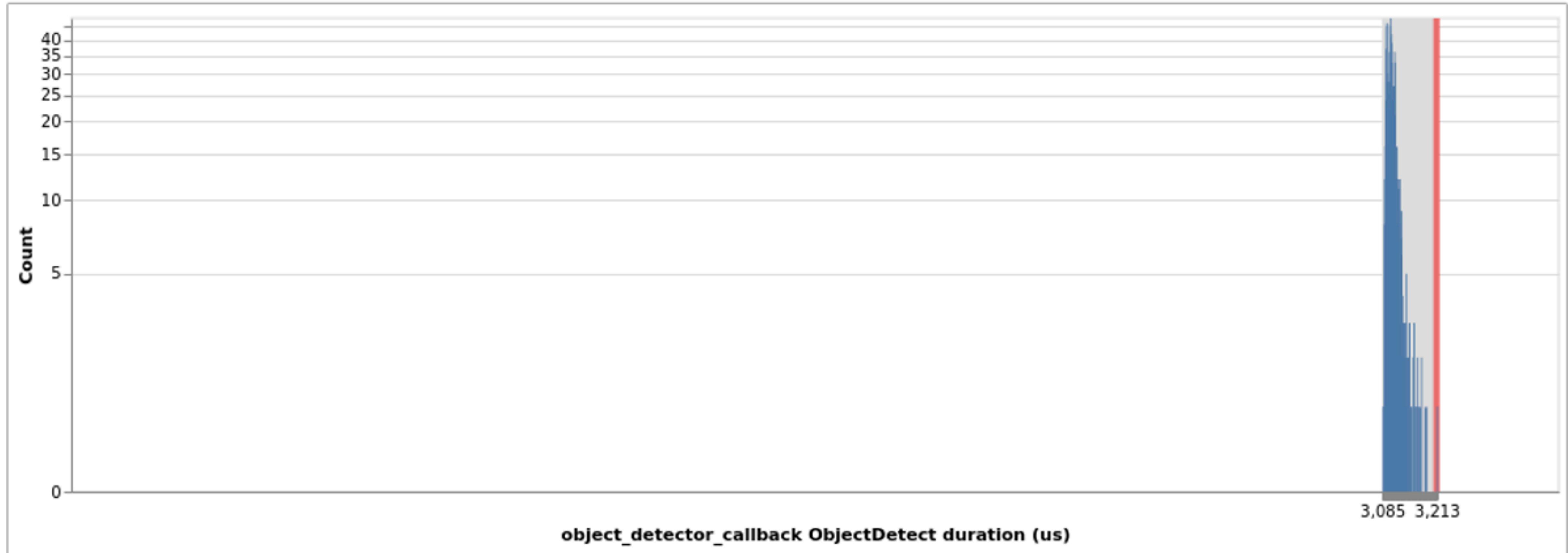
Visualizar latencias - DataLogger

Select a slice: [data_logger_callback](#) ▾ [DataLogger](#) ▾ [us](#) ▾



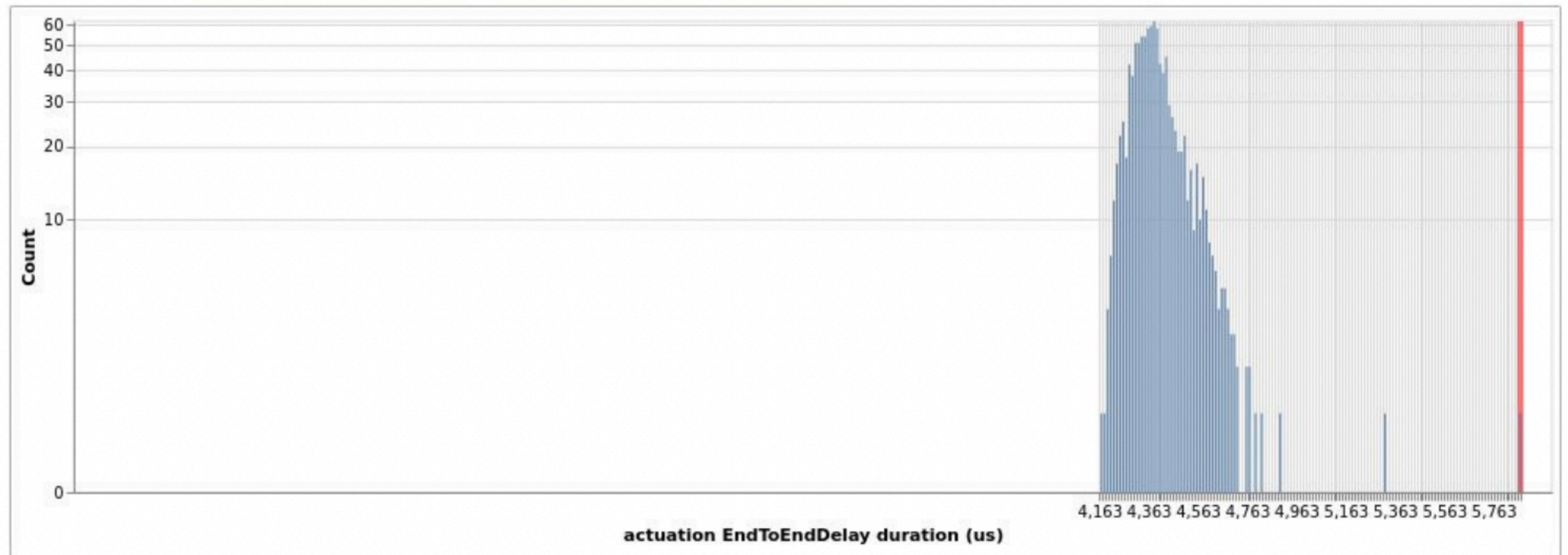
Visualizar latencias - Latencia End to End

Select a slice: object_detector_callback ObjectDetect us



Visualizar latencias - Latencia End to End

Select a slice: actuation ▾ EndToEndDelay ▾ us ▾



Resumen del ejercicio 3

- En 3-1, el data logger retrasa el detector de objetos
- En 3-2, el detector de objetos se ejecuta primero (o en paralelo, si es posible)

Ejecutor ROS 2 predeterminado



El callback RT se retrasa

Ejecutor de Tiempo Real del Workshop



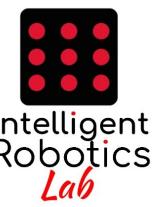
El callback RT no se retrasa

Ejercicio complementario: repetición sin y sin estrés

- ¿Cómo se compara el ejercicio 3-1 (sin Tiempo Real) con y sin estrés?
- ¿Cómo se compara el ejercicio 3-2 (Tiempo Real) con y sin estrés?
- ¿El estrés afecta la latencia del ejecutor predeterminado?
- ¿El estrés afecta la latencia de los callbacks en Tiempo Real?

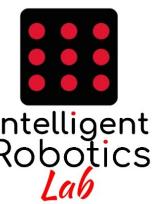


Tiempo Real + ROS 2 con herramientas existentes



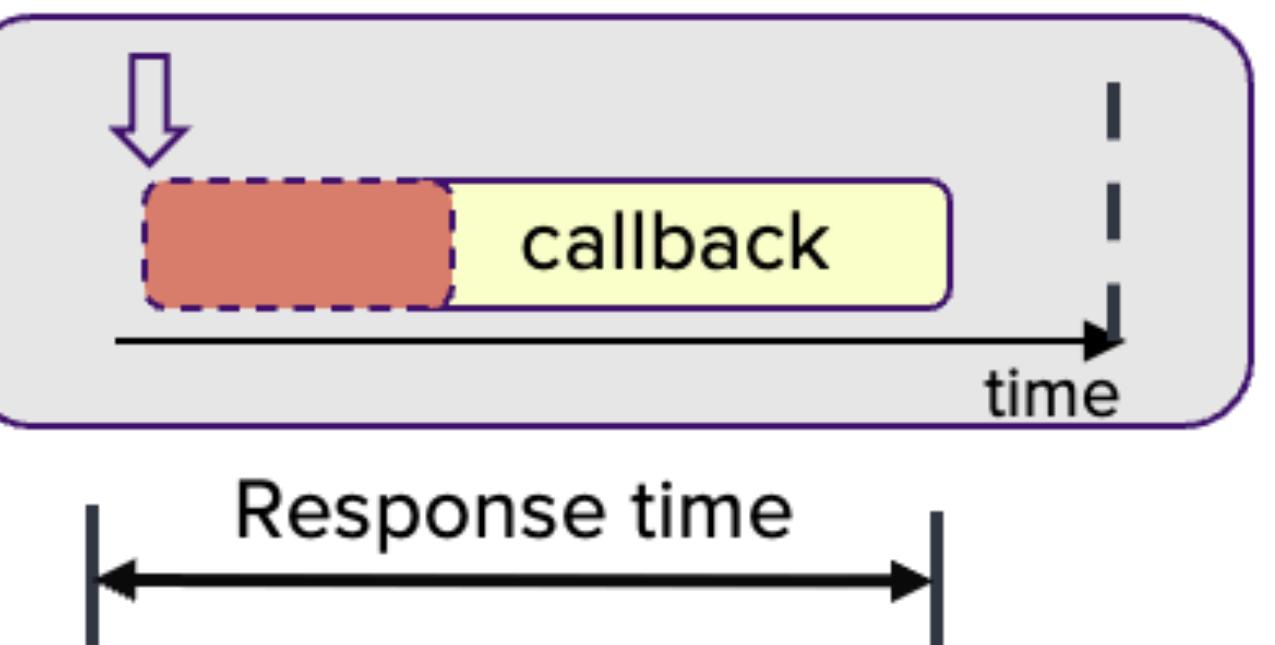
Tiempo Real + ROS 2 con herramientas existentes

- Descripción general de los sistemas en Tiempo Real
- Priorización de callbacks
 - Nodo único ROS 2
 - Callback Groups
 - Nivel de sistema (PICAS)
- Orden de procesamiento de callbacks
- Reducción del tiempo de comunicación (mensajes prestados, copia cero)
- Herramientas y benchmarking



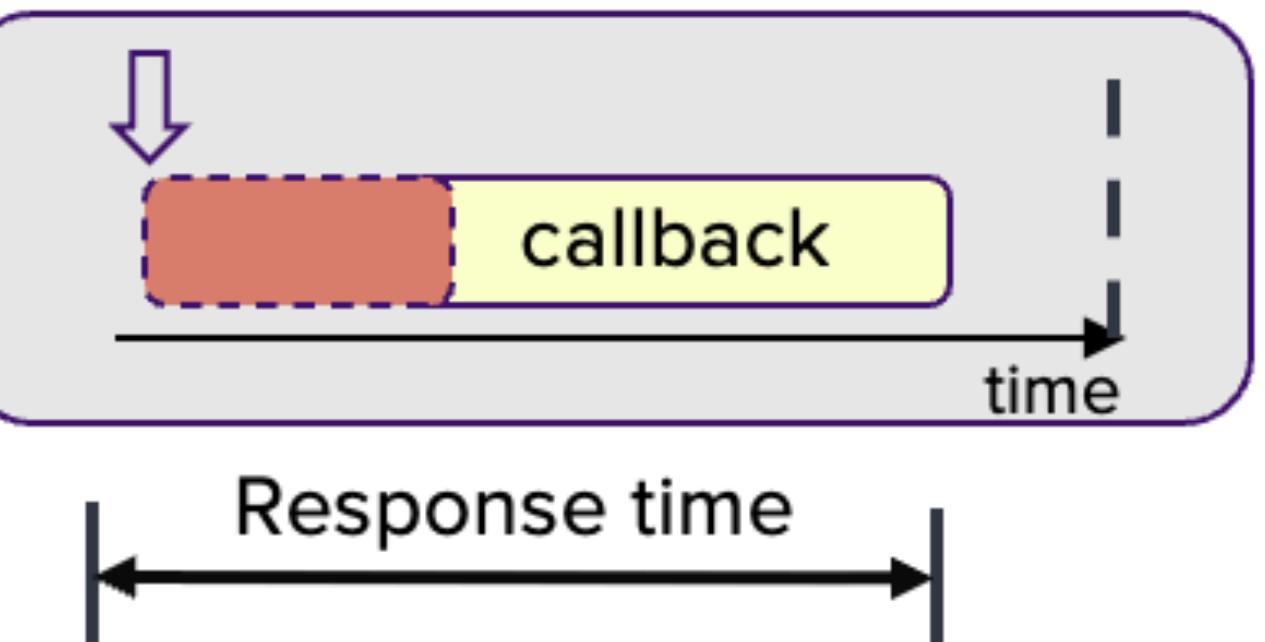
Tiempo Real + ROS 2 con herramientas existentes

- Tiempo de respuesta acotado

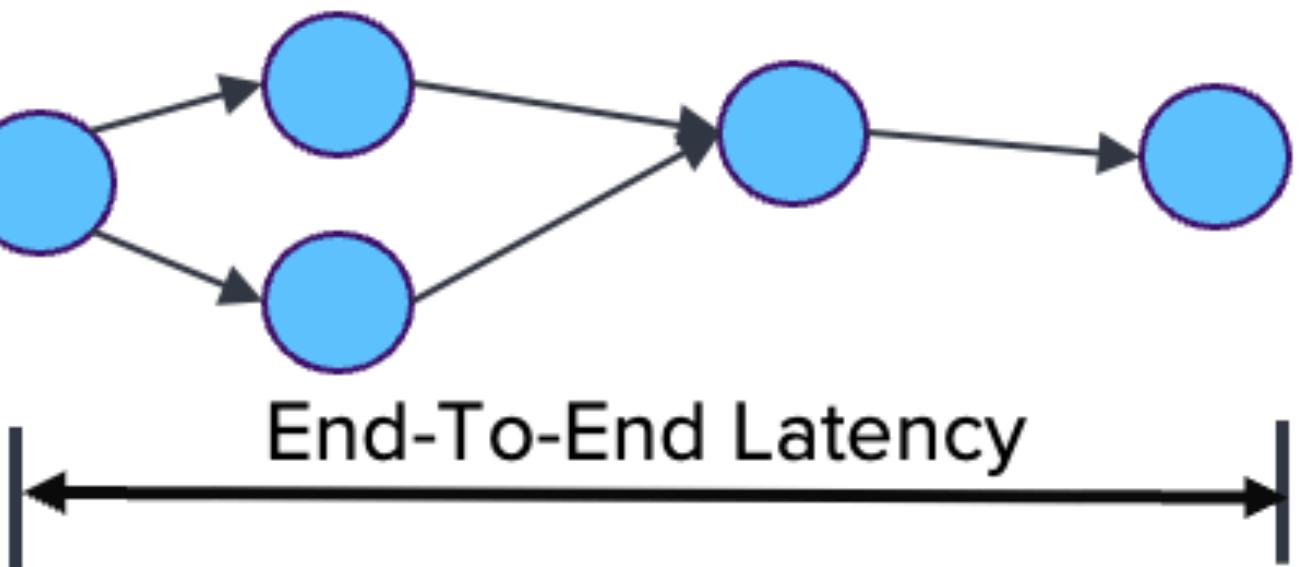


Tiempo Real + ROS 2 con herramientas existentes

- Tiempo de respuesta acotado

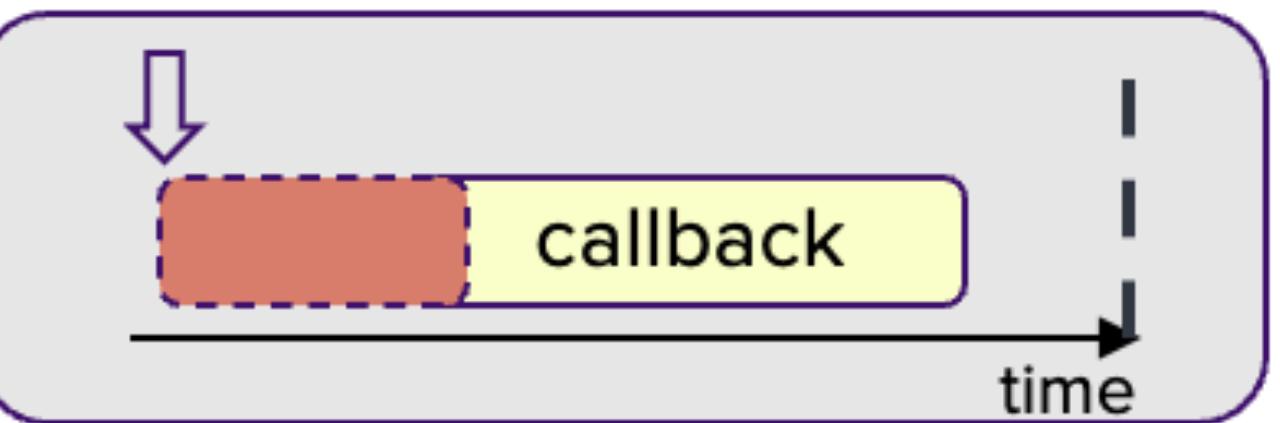


- Latencia End-To-End Garantizada

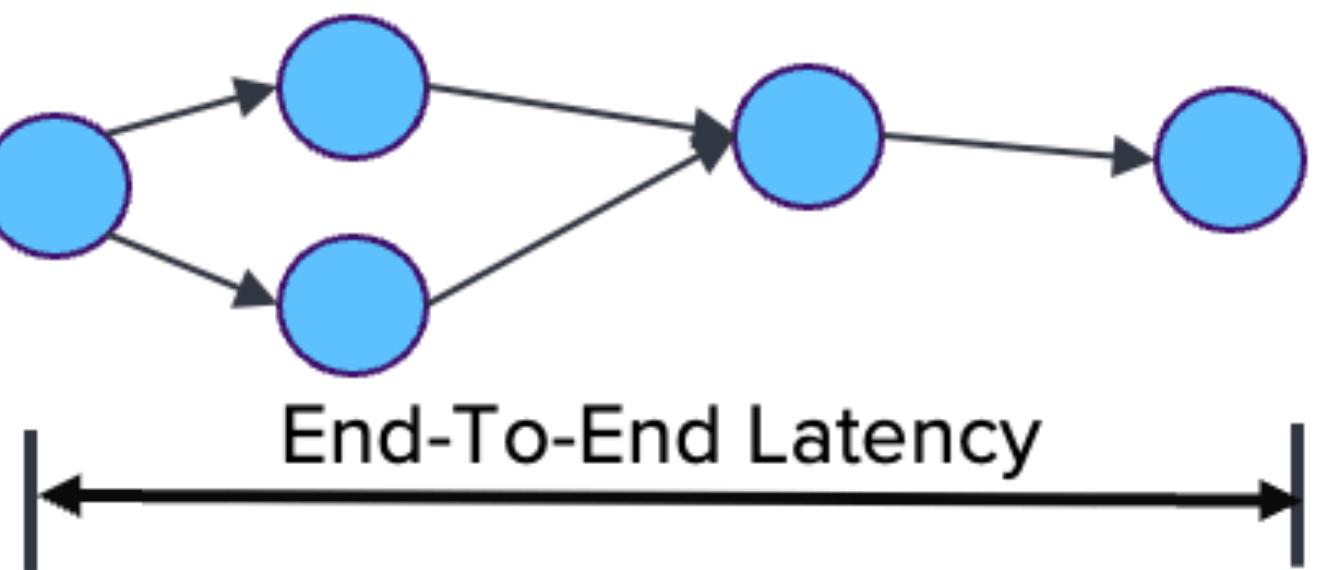


Tiempo Real + ROS 2 con herramientas existentes

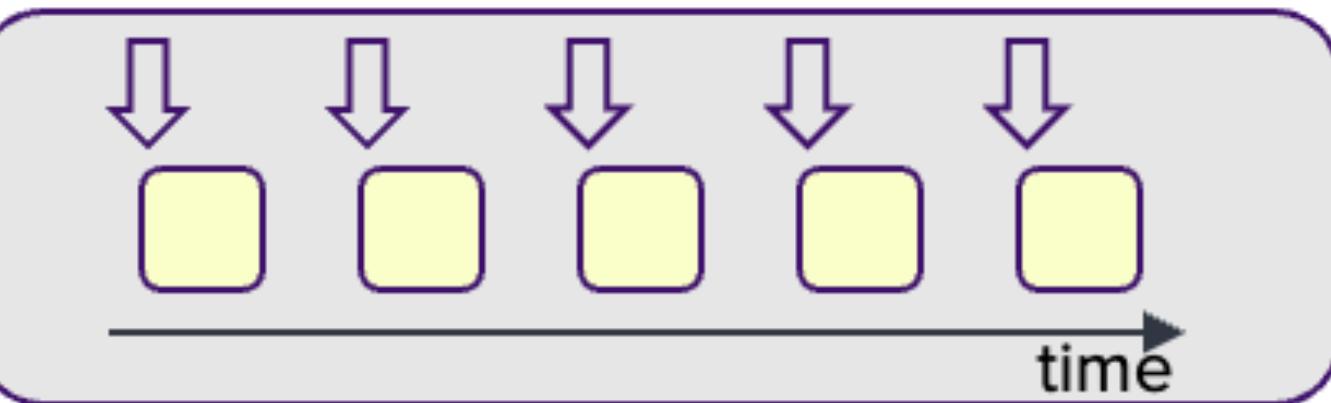
- Tiempo de respuesta acotado



- Latencia End-To-End Garantizada



- Tasas de actualización determinísticas



Descripción general de los sistemas en Tiempo Real

Teoría

- Desde 1970...
- Requisitos del modelo
 - Tiempos de ejecución
 - Activación periódica; invocaciones/intervalo de tiempo
 - Activado por datos, activado por tiempo
- Algoritmos de planificación
 - Rate Monotonic
 - Earliest Deadline First
 - Planificación basada en reservas
 - Probabilística, ..., etc.
- Hardware complejo
 - caché, pipeline, multinúcleo ...
- Análisis de la planificabilidad

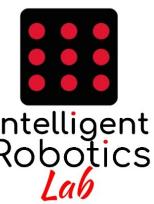
Descripción general de los sistemas en Tiempo Real

Teoría

- Desde 1970...
- Requisitos del modelo
 - Tiempos de ejecución
 - Activación periódica; invocaciones/intervalo de tiempo
 - Activado por datos, activado por tiempo
- Algoritmos de planificación
 - Rate Monotonic
 - Earliest Deadline First
 - Planificación basada en reservas
 - Probabilística, ..., etc.
- Hardware complejo
 - caché, pipeline, multinúcleo ...
- Análisis de la planificabilidad

Práctica

- Sistemas operativos en Tiempo Real
 - QNX
 - FreeRTOS, Zephyr, ...
- Linux en Tiempo Real
 - SCHED_FIFO
 - SCHED_DEADLINE (EDF)
- Automoción: Planificación Rate-monotonic
 - AUTOSAR
 - Períodos fijos
 - LET para intercambio sincronizado de datos



Descripción general de los sistemas en Tiempo Real

Teoría

- Desde 1970...
- Requisitos del modelo
 - Tiempos de ejecución
 - Activación periódica; invocación de tiempo
 - Activado por datos, activación
- Algoritmos de planificación
 - Rate Monotonic
 - Earliest Deadline First
 - Planificación basada en retrasos
 - Probabilística, ..., etc.
- Hardware complejo
 - caché, pipeline, multithreading
- Análisis de la planificabilidad

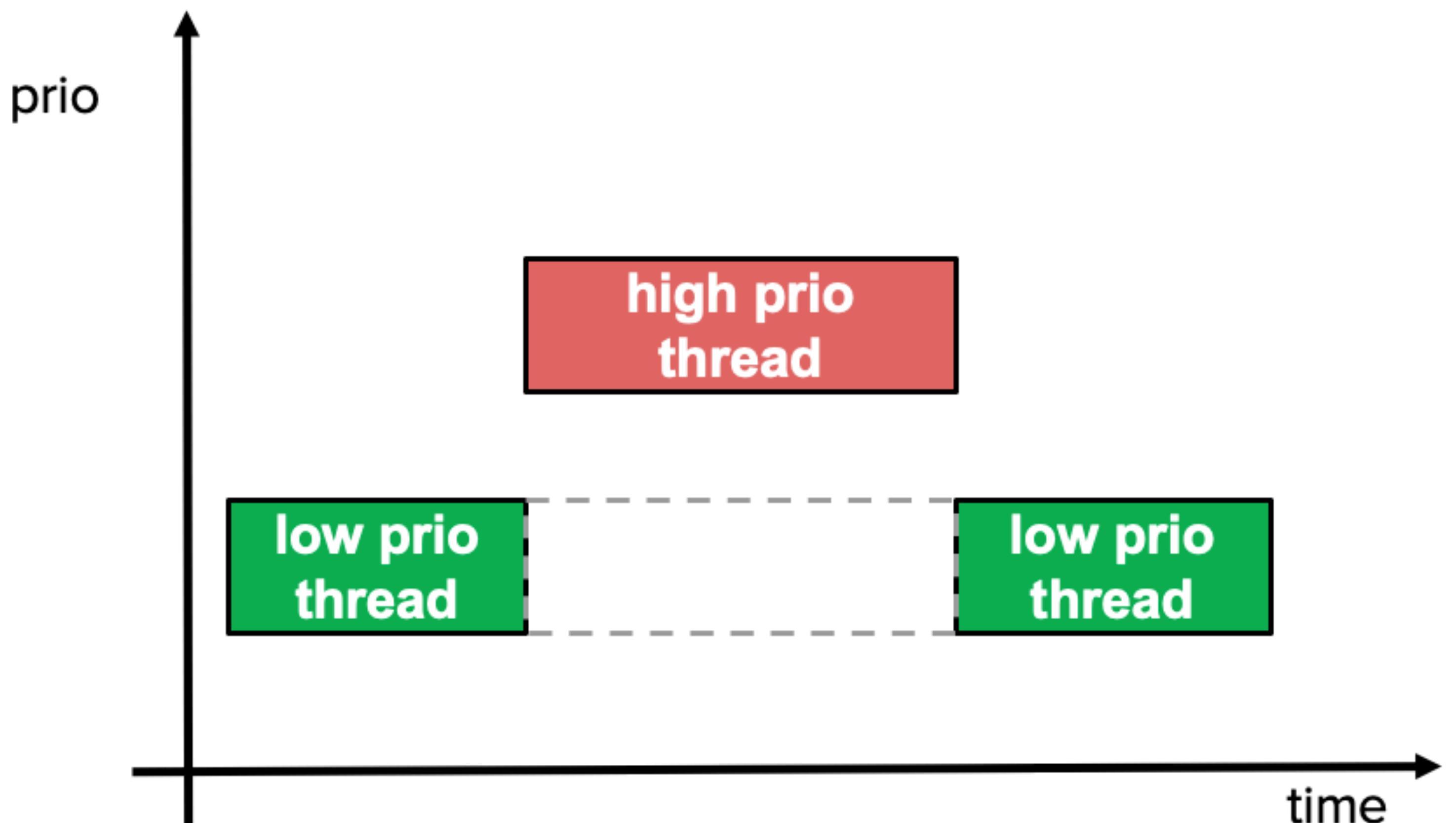
Práctica

Estamos en 2024
ROS 2 no tiene
capacidad de Tiempo
Real

¿Cuál es el problema?

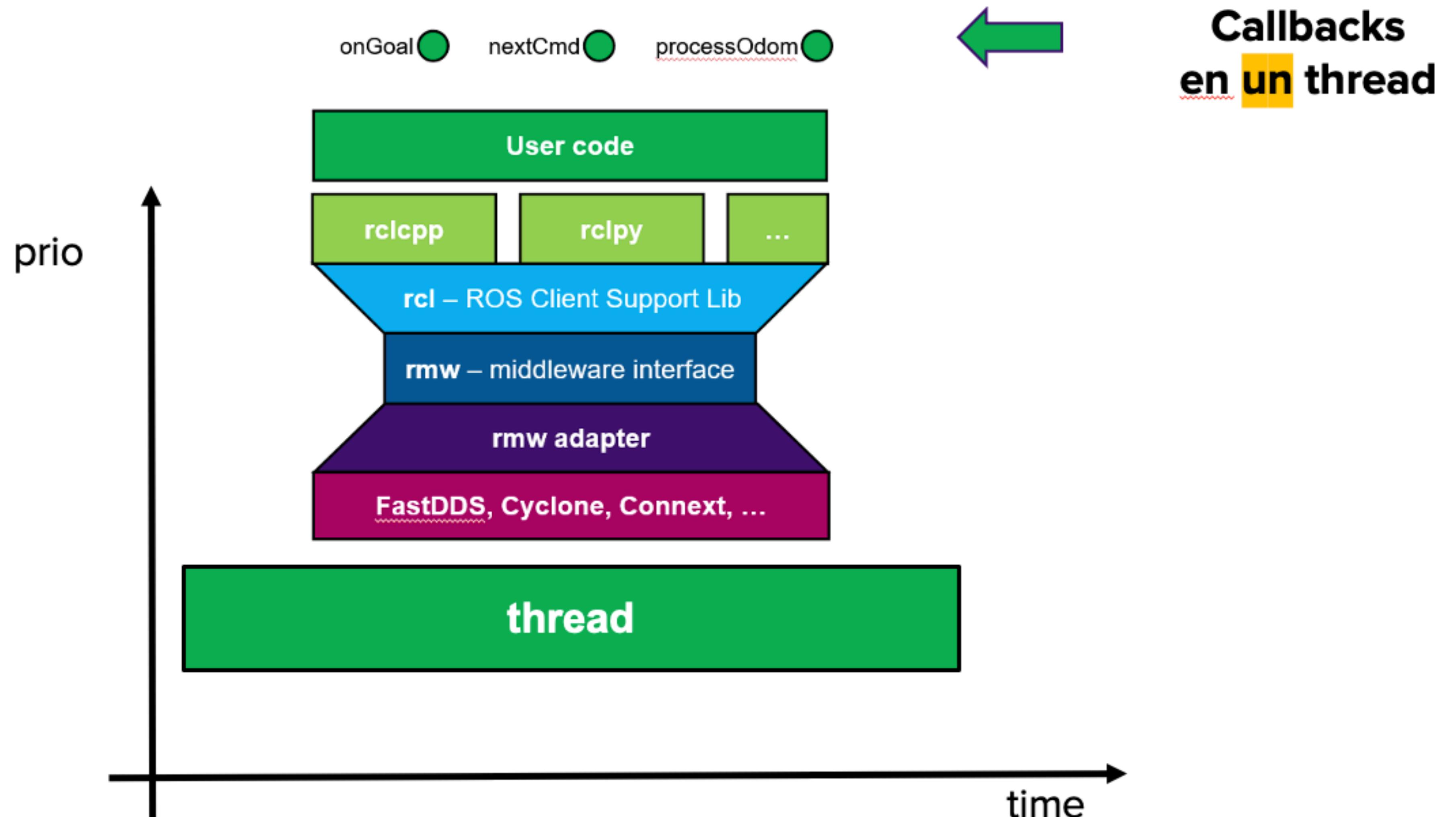
operativos en Tiempo Real
Zephyr, ...
Tiempo Real
FO
EDDF (EDF)
on: Planificación Rate-
os
intercambio sincronizado de datos

Características del Sistemas Operativo



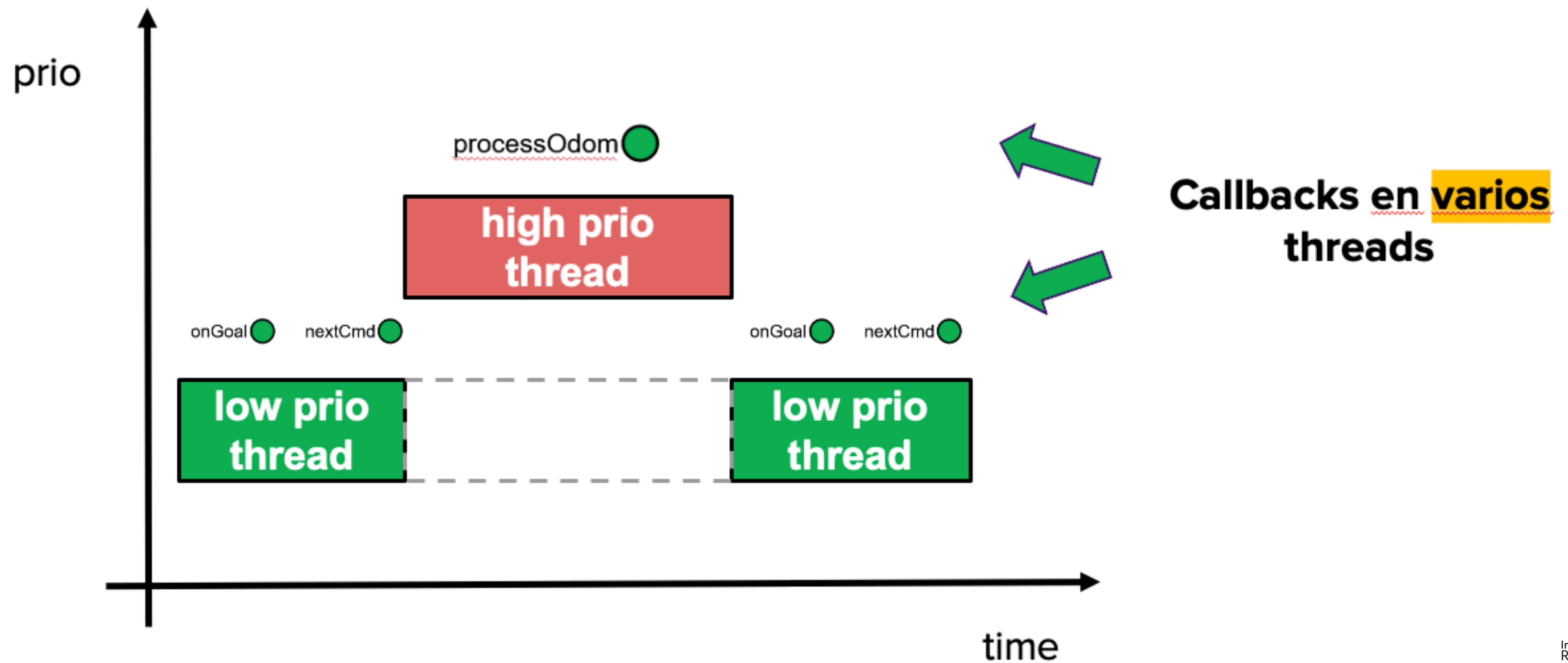
El Sistema Operativo puede planificar threads en Tiempo Real

Características del Stack de ROS 2



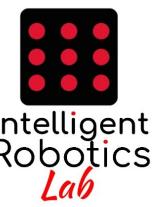
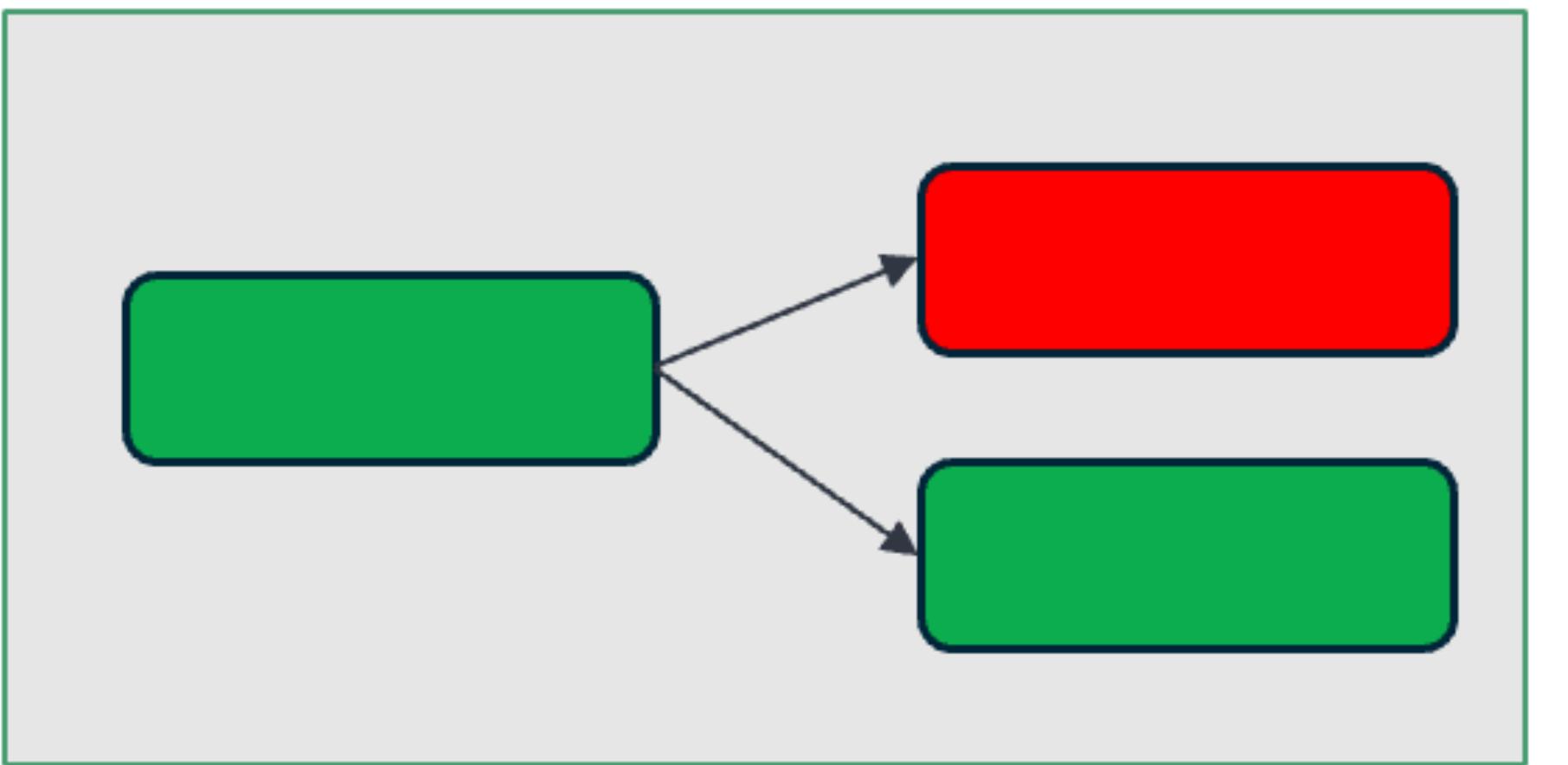
Problema: todas los callbacks en un hilo (o varios hilos con la misma prioridad).

Solución: Distribuir callbacks a diferentes threads priorizados



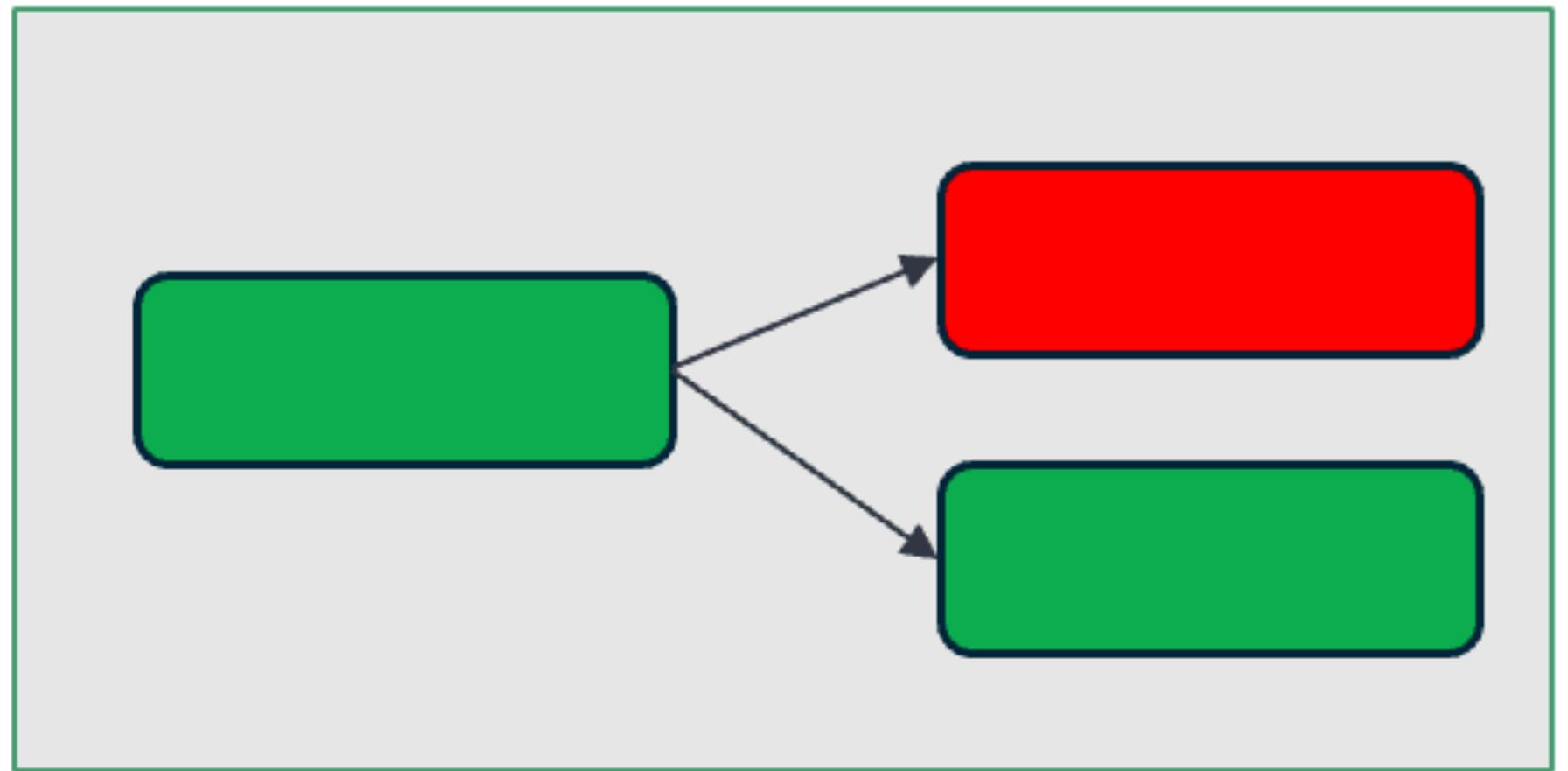
Escenarios de aplicación

Un Nodo Real-Time

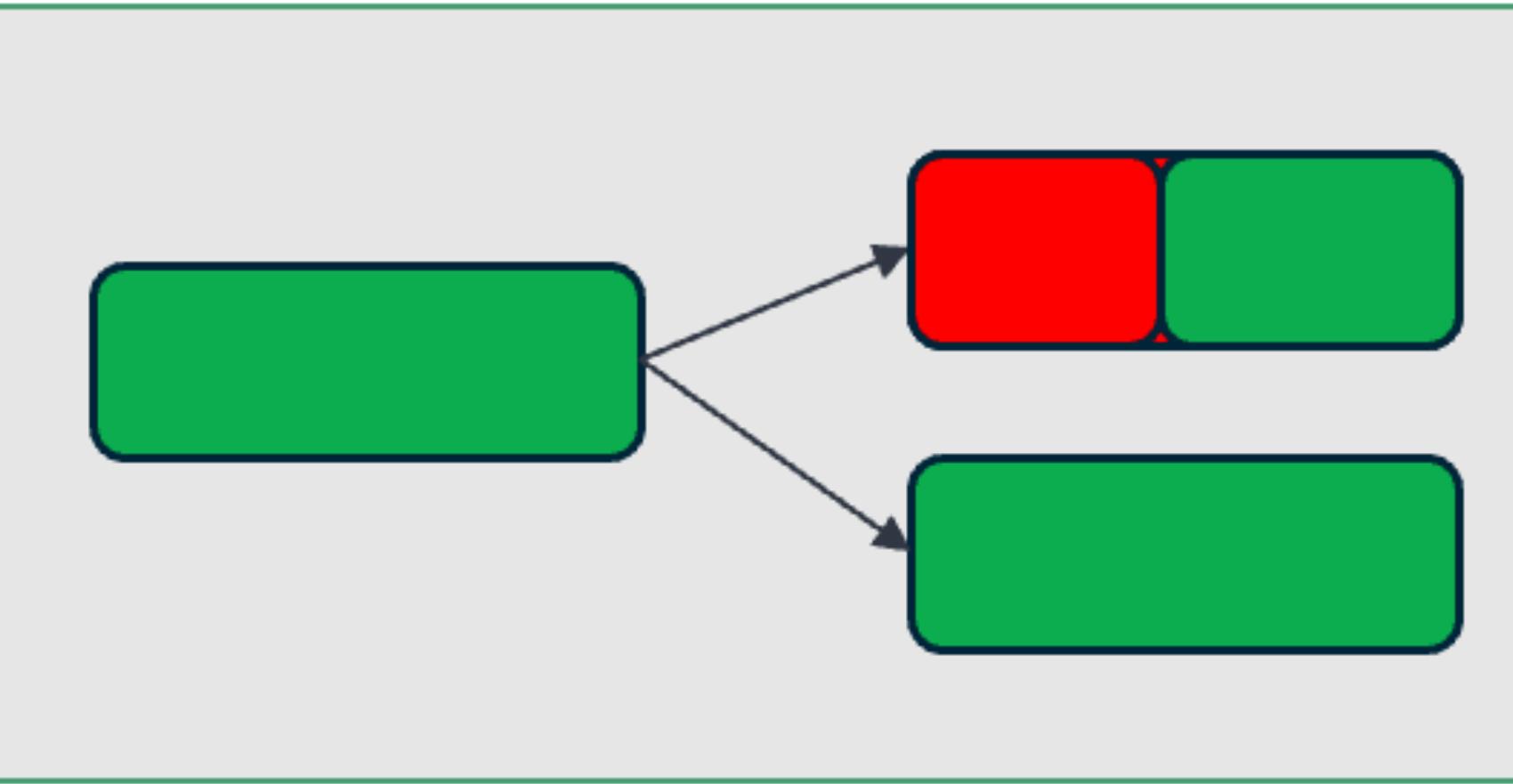


Escenarios de aplicación

Un Nodo Real-Time

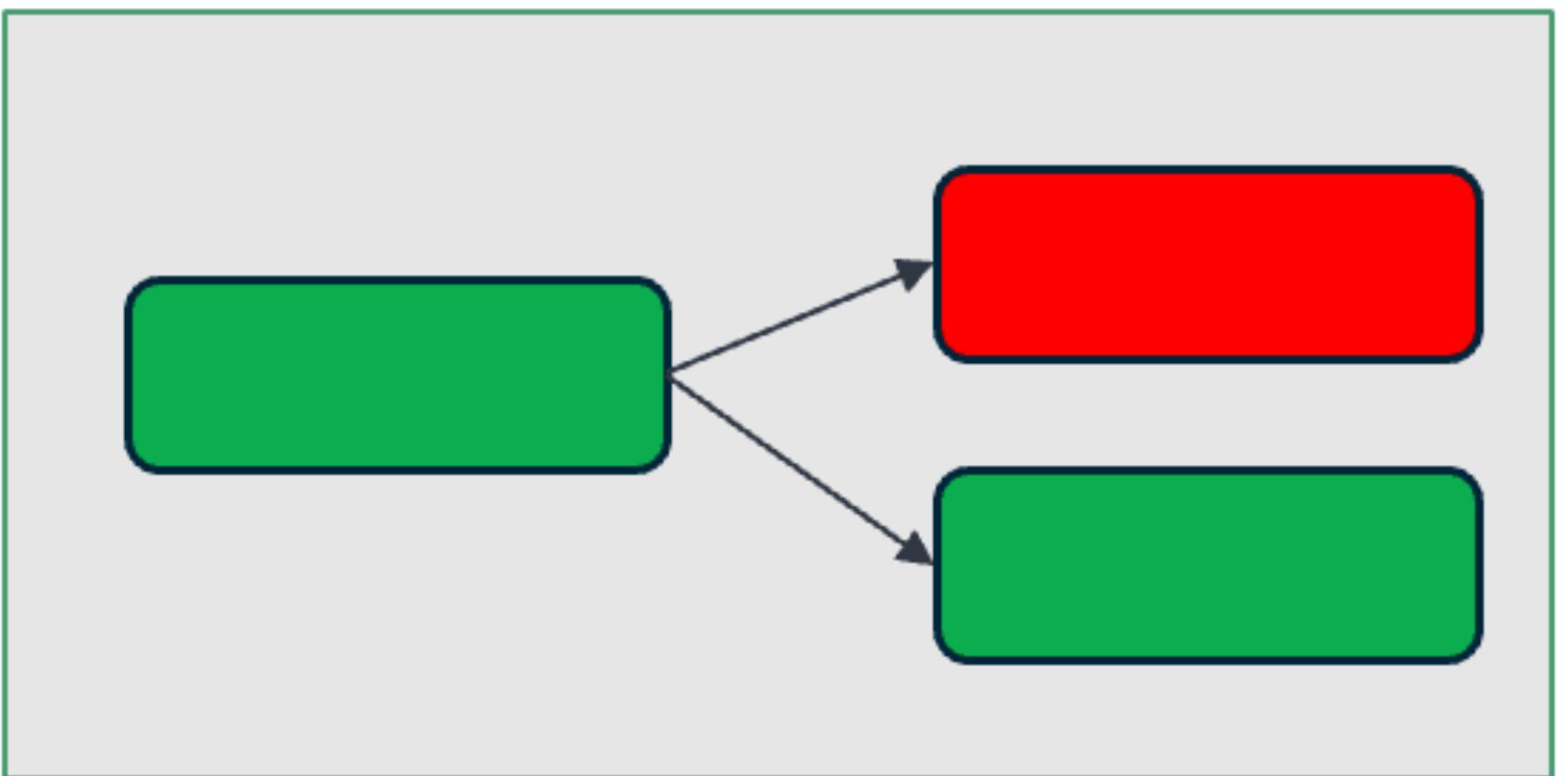


Nodo con Criticidad Mezclada

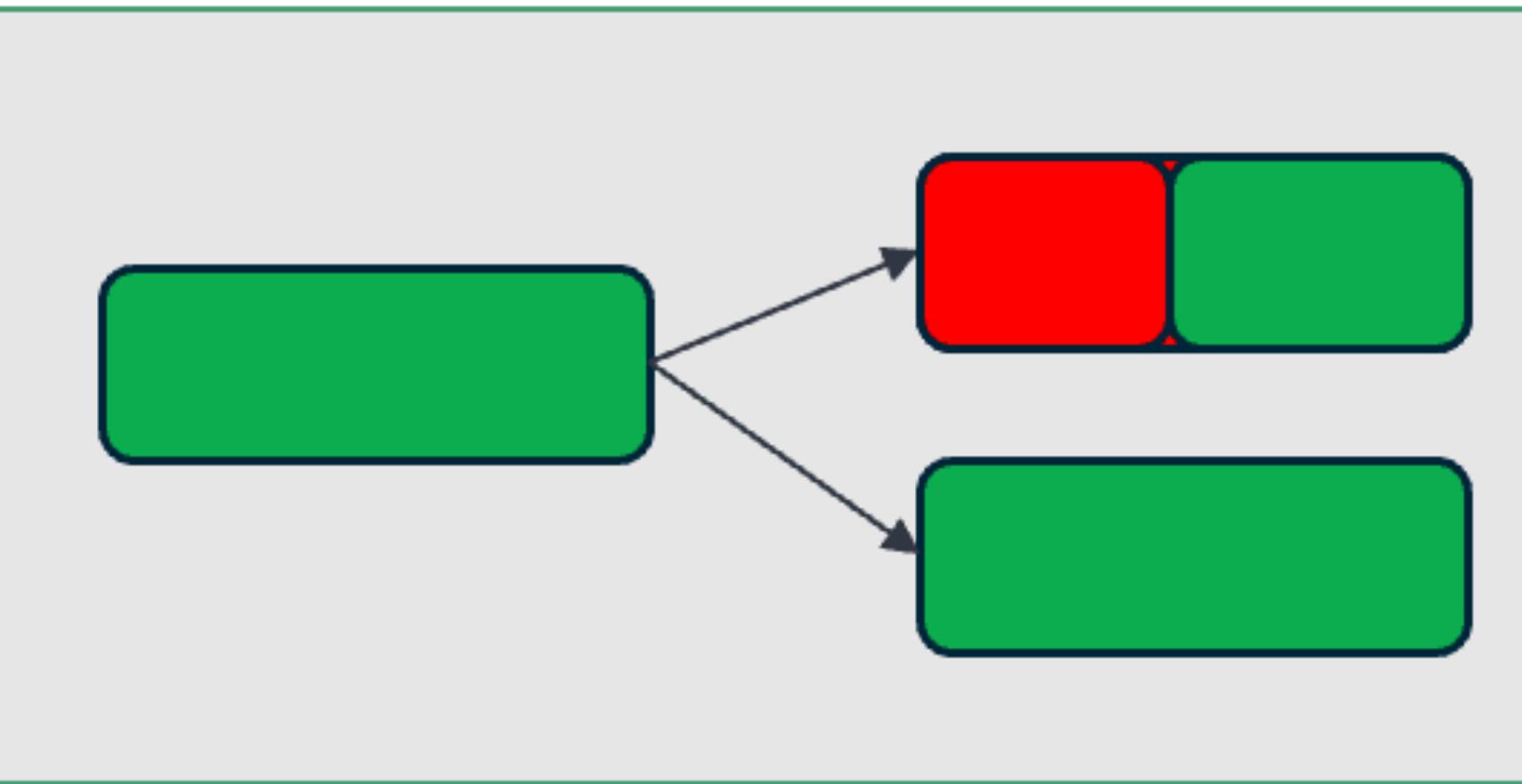


Escenarios de aplicación

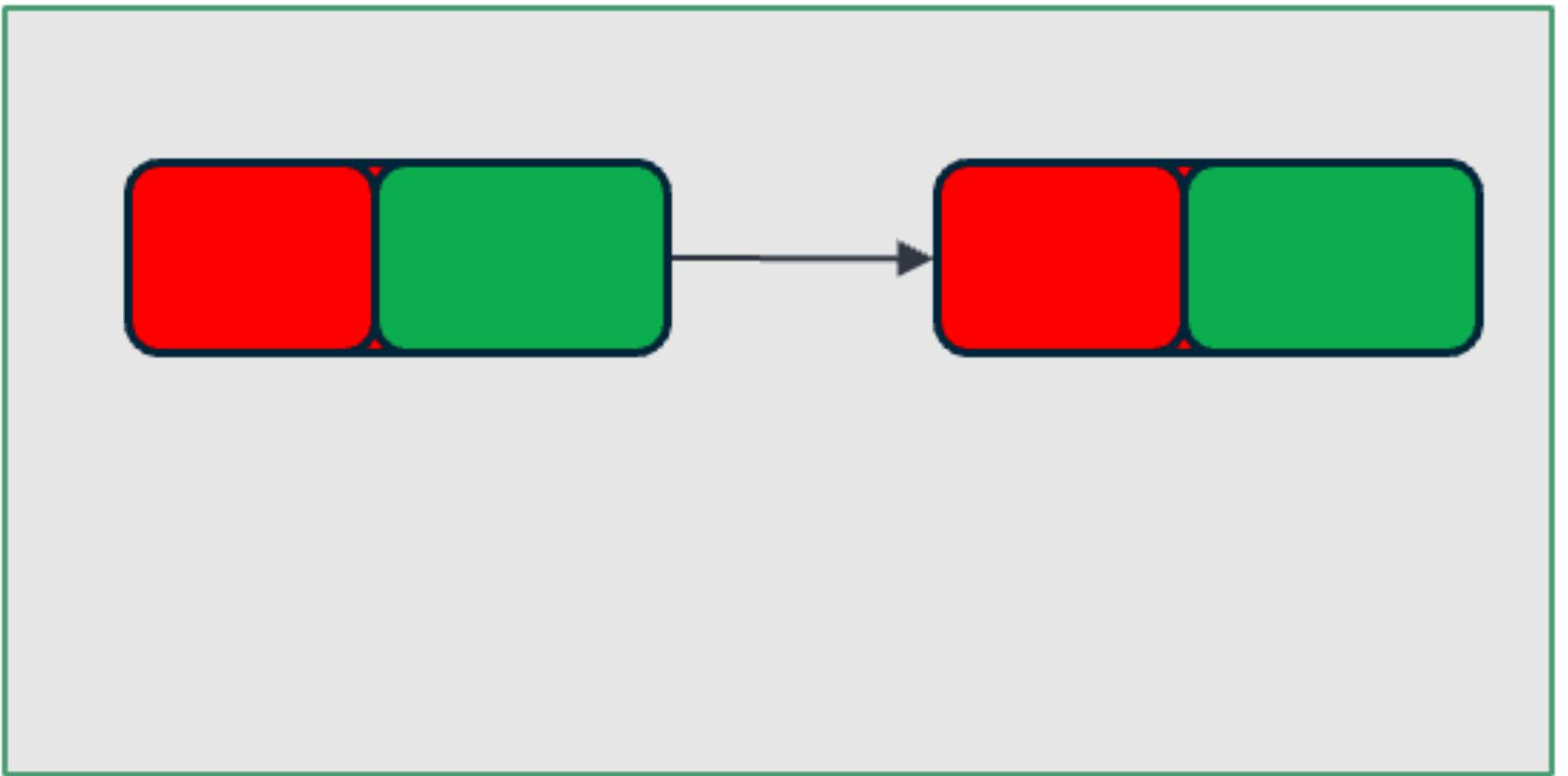
Un Nodo Real-Time



Nodo con Criticidad Mezclada

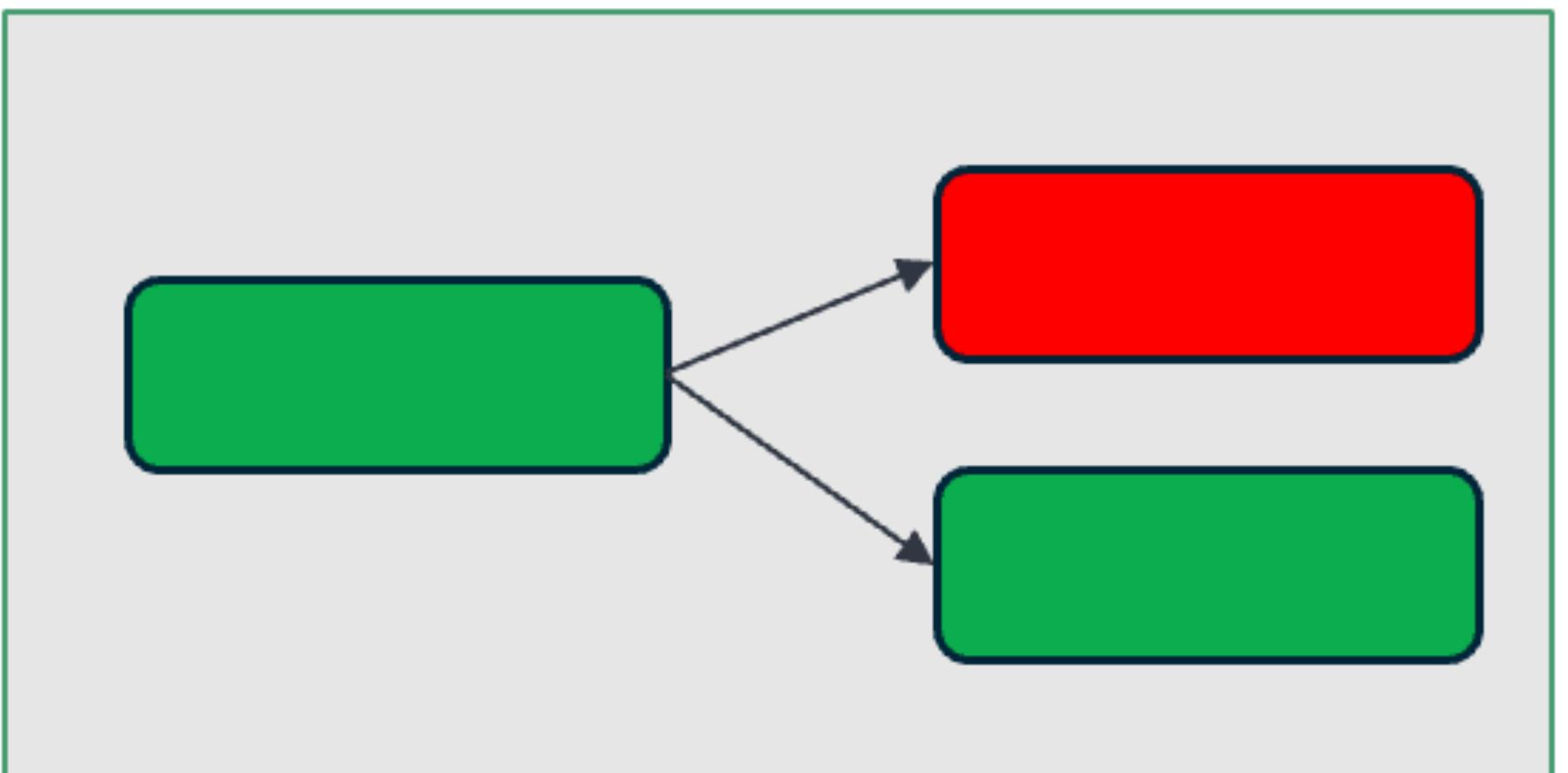


Varios Nodos con Criticidad Mezclada

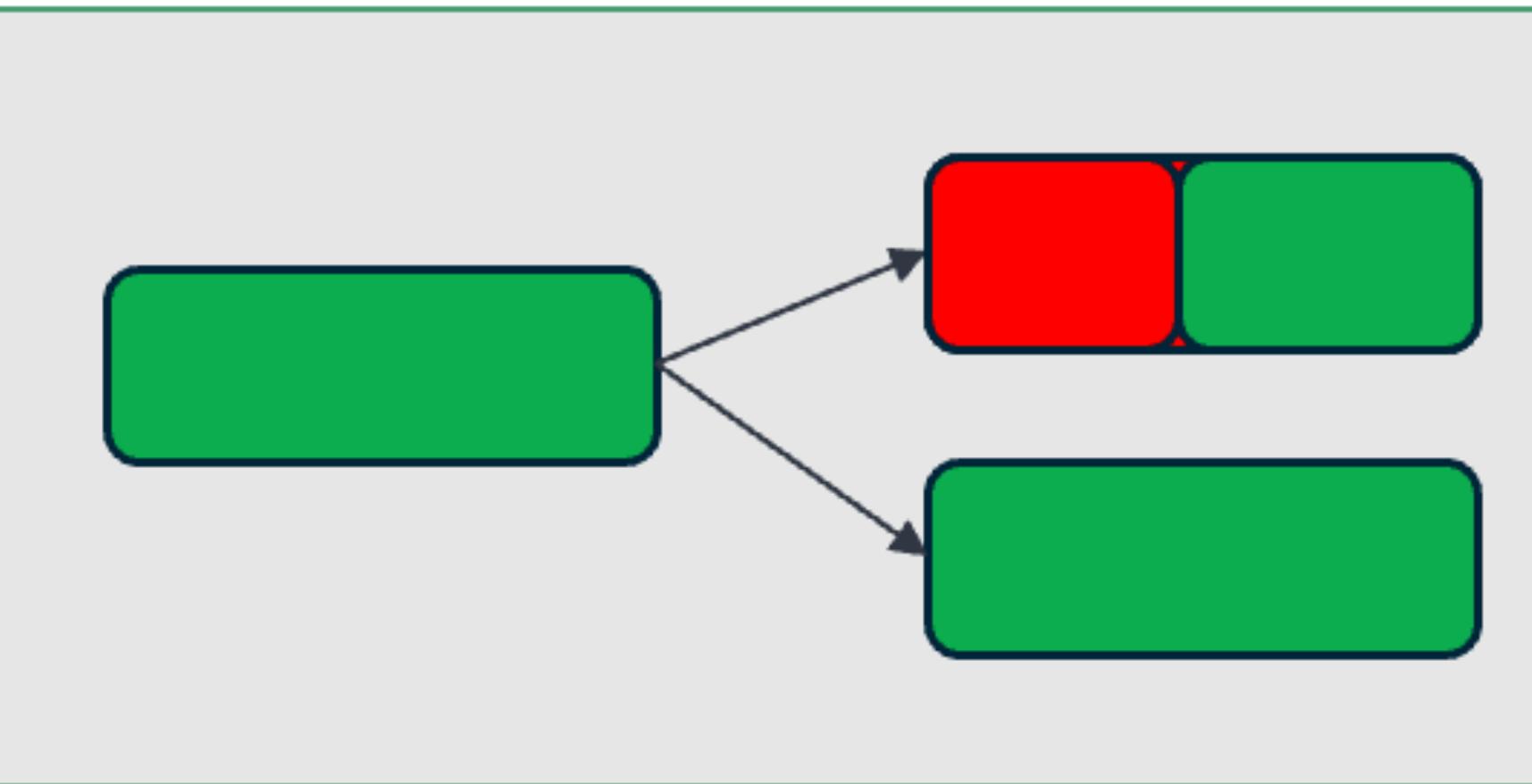


Escenarios de aplicación

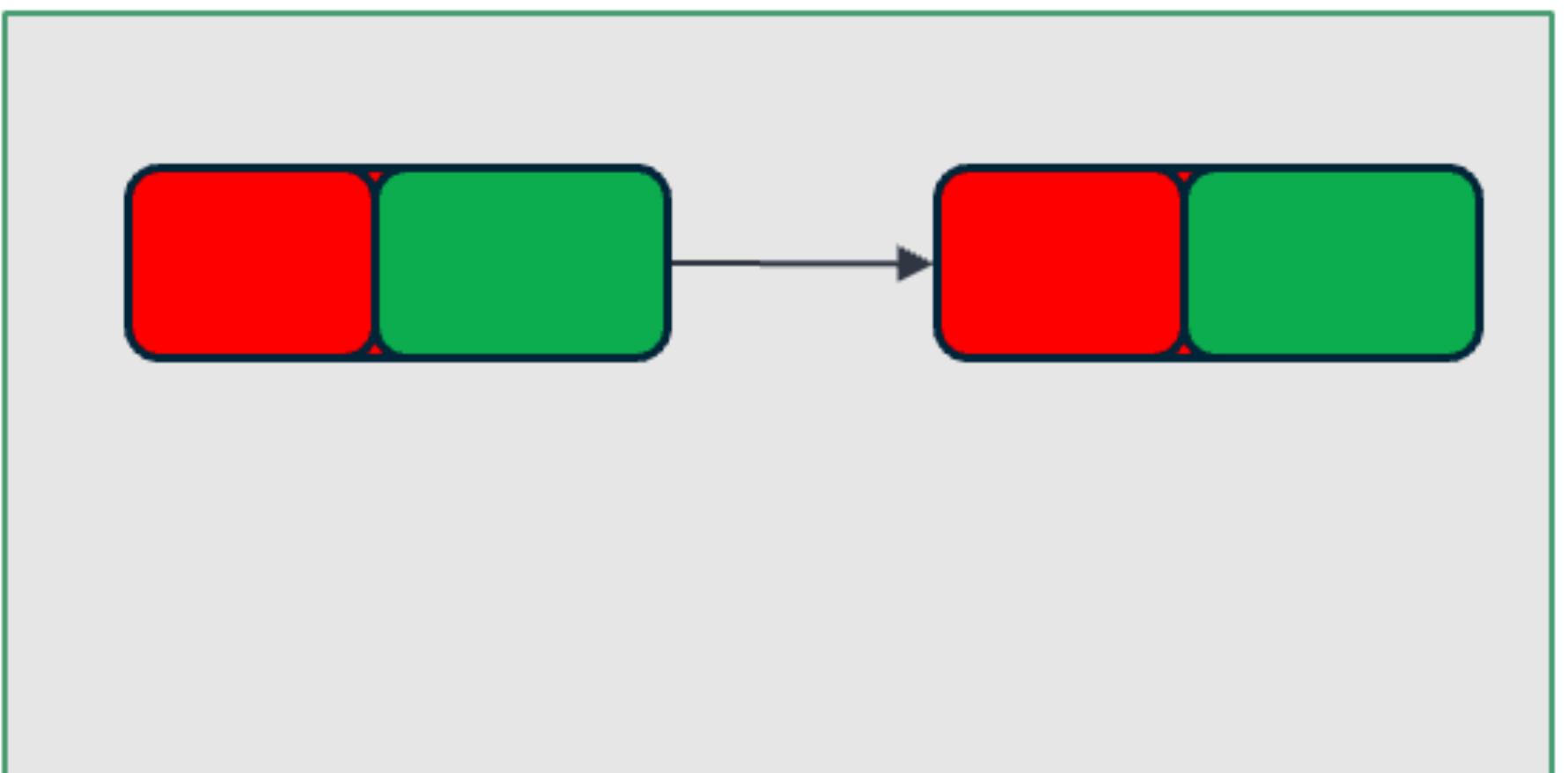
Un Nodo Real-Time



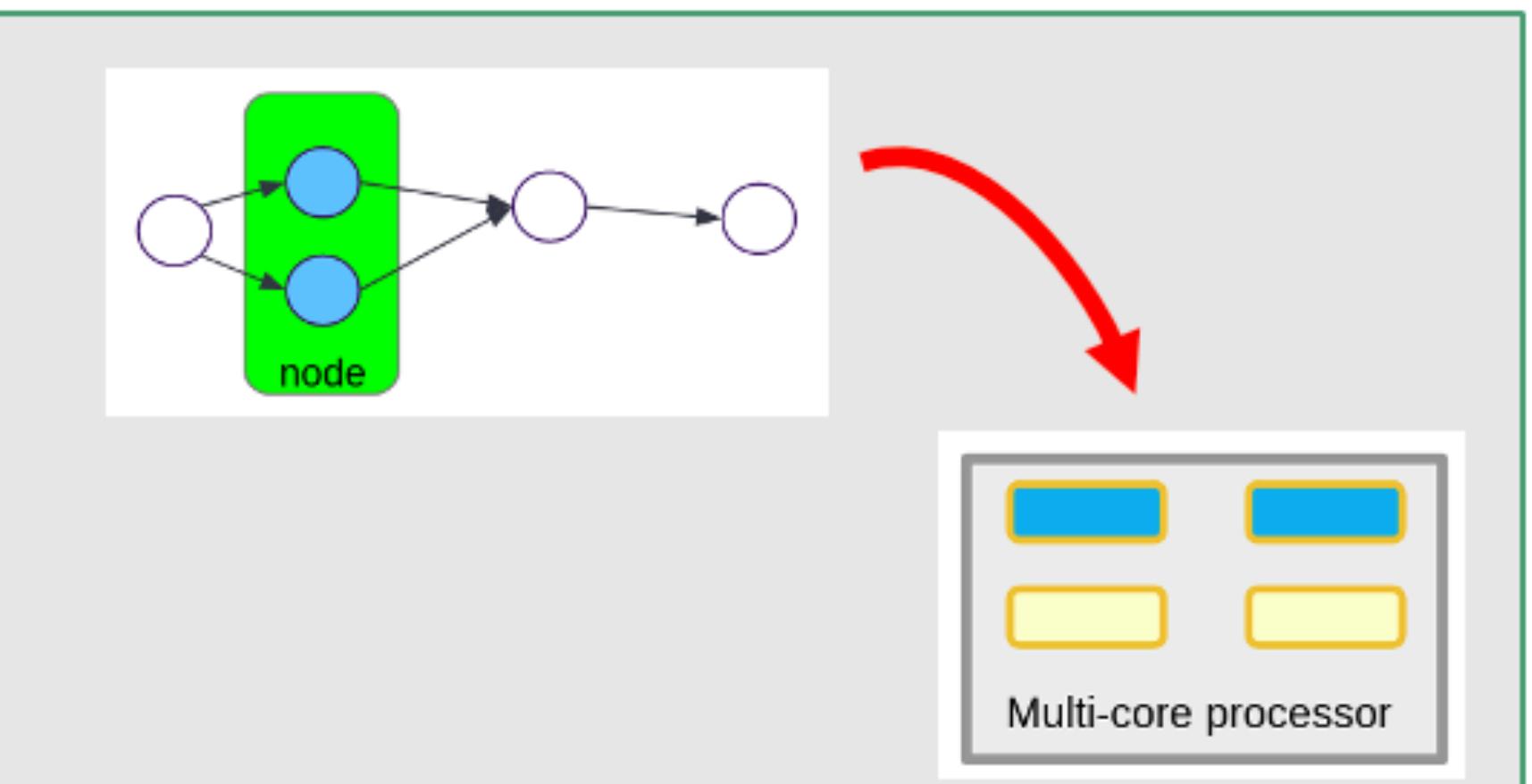
Nodo con Criticidad Mezclada



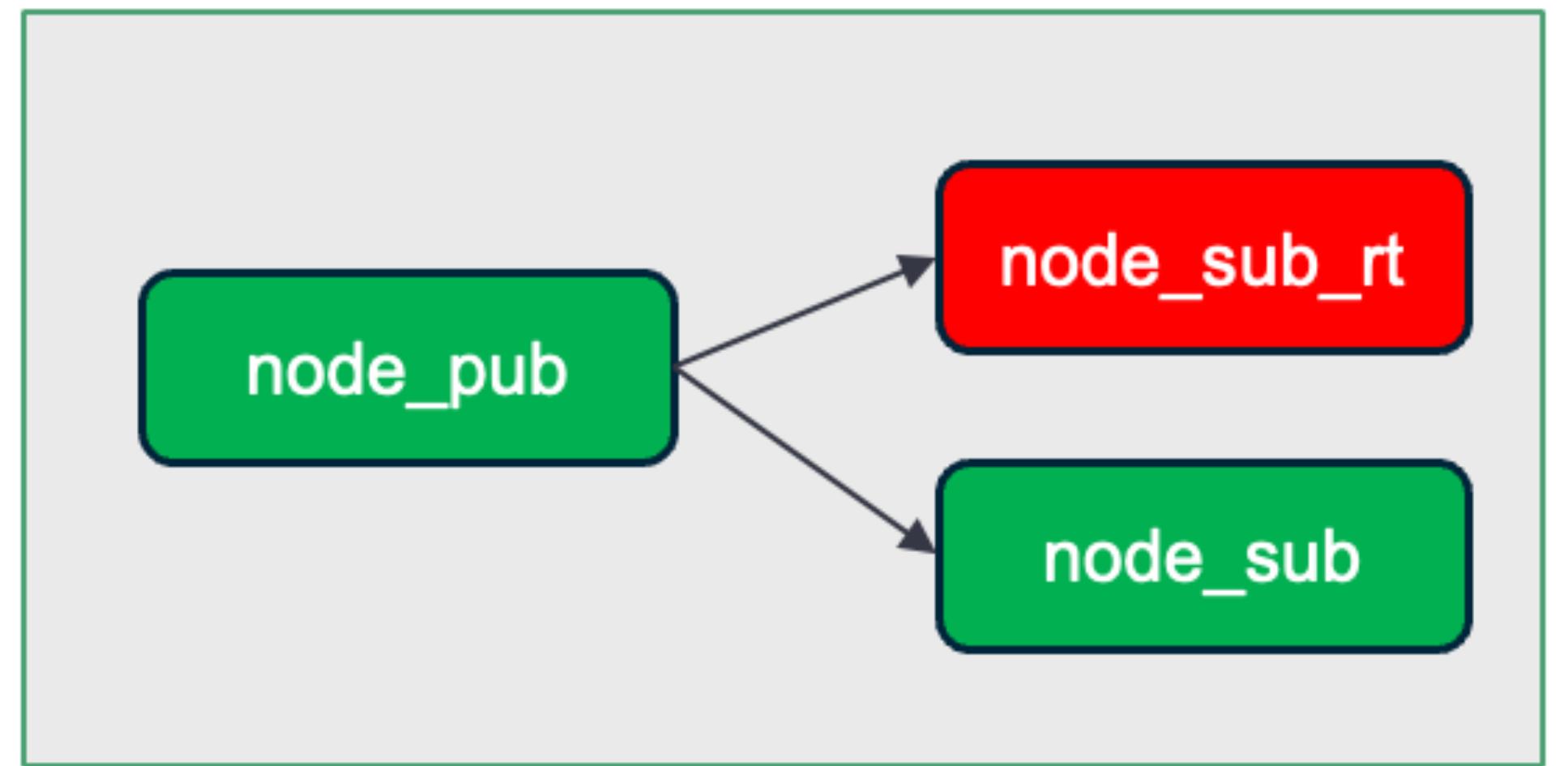
Varios Nodos con Criticidad Mezclada



Paralelización en multi-core

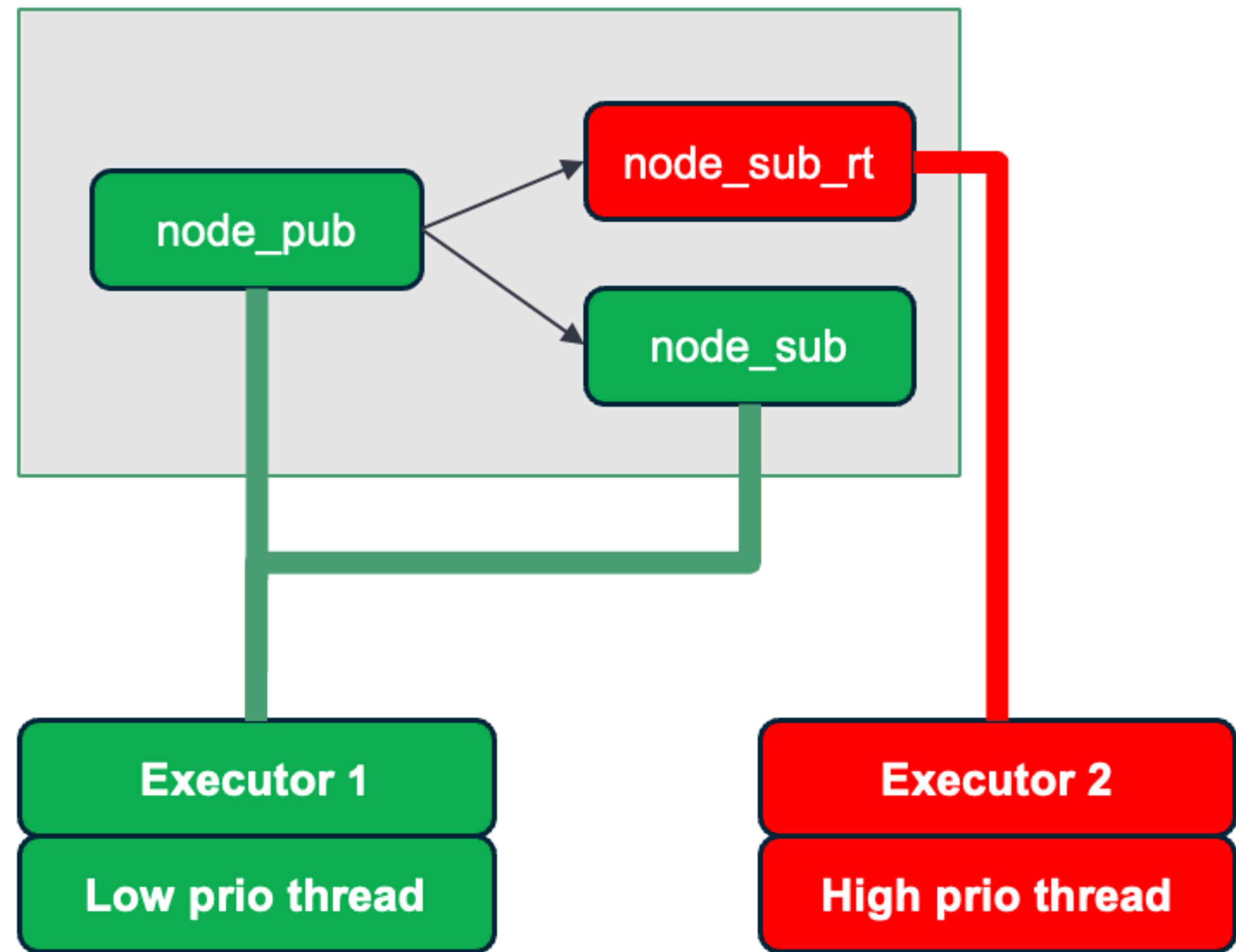


Escenarios de aplicación: Un Nodo Tiempo Real



https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_real_time_executor.cpp

Escenarios de aplicación: Un Nodo Tiempo Real



https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_real_time_executor.cpp

Escenarios de aplicación: Un Nodo Tiempo Real

```
rclcpp::init(argc, argv);

auto node_pub = std::make_shared<MinimalPublisher>();
auto node_sub = std::make_shared<MinimalSubscriber>("minimal_sub1", "topic");
auto node_sub_rt = std::make_shared<MinimalSubscriber>("minimal_sub2", "topic_rt");

rclcpp::executors::StaticSingleThreadedExecutor default_executor;
rclcpp::executors::StaticSingleThreadedExecutor realtime_executor;

// the publisher and non real-time subscriber are processed by default_executor
default_executor.add_node(node_pub);
default_executor.add_node(node_sub);

// real-time subscriber is processed by realtime_executor.
realtime_executor.add_node(node_sub_rt);
```

- Crear dos ejecutores
- Agregar callbacks que no sean en Tiempo Real a un ejecutor
- Agregar callbacks en Tiempo Real a otro ejecutor

https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_real_time_executor.cpp

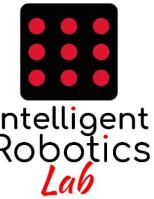
Escenarios de aplicación: Un Nodo Tiempo Real

```
// spin non real-time tasks in a separate thread
auto default_thread = std::thread(
    [&]() {
        default_executor.spin();
    });

// spin real-time tasks in a separate thread
auto realtime_thread = std::thread(
    [&]() {
        realtime_executor.spin();
    });
set_thread_scheduling(realtime_thread.native_handle(),
                      options.policy, options.priority);
default_thread.join();
realtime_thread.join();
rclcpp::shutdown();
```

- Ejecutar el ejecutor predeterminado en un thread normal
- Ejecutar el ejecutor en Tiempo Real en un thread en Tiempo Real

https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_real_time_executor.cpp



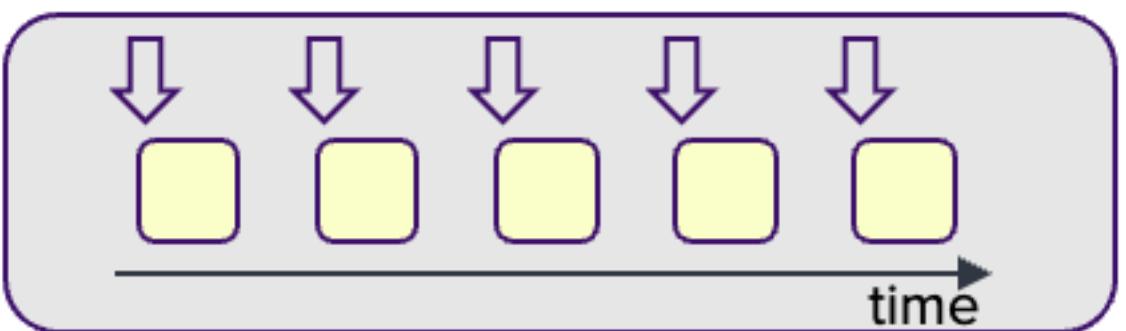
Escenarios de aplicación: Un Nodo Tiempo Real

```
// spin non real-time tasks in a separate thread
auto default_thread = std::thread([&]() {
    default_executor.spin();
});

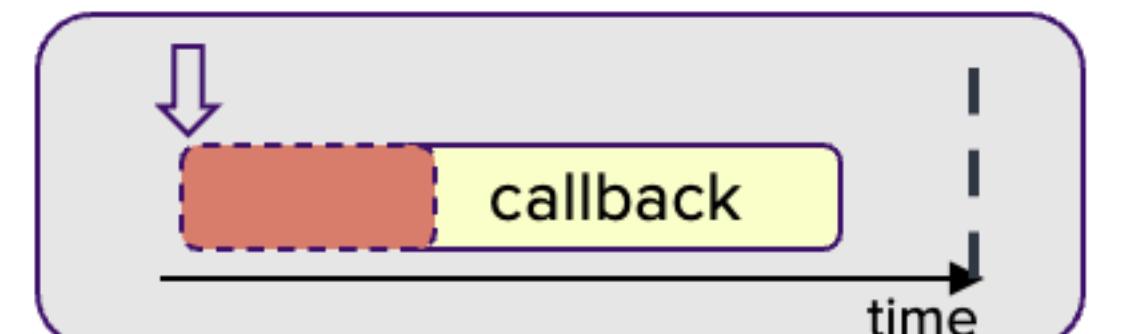
// spin real-time tasks in a separate thread
auto realtime_thread = std::thread([&]() {
    realtime_executor.spin();
});
set_thread_scheduling(realtime_thread, options);
default_thread.join();
realtime_thread.join();
rclcpp::shutdown();
```

Benefits:

- **Improves bounded update rate**



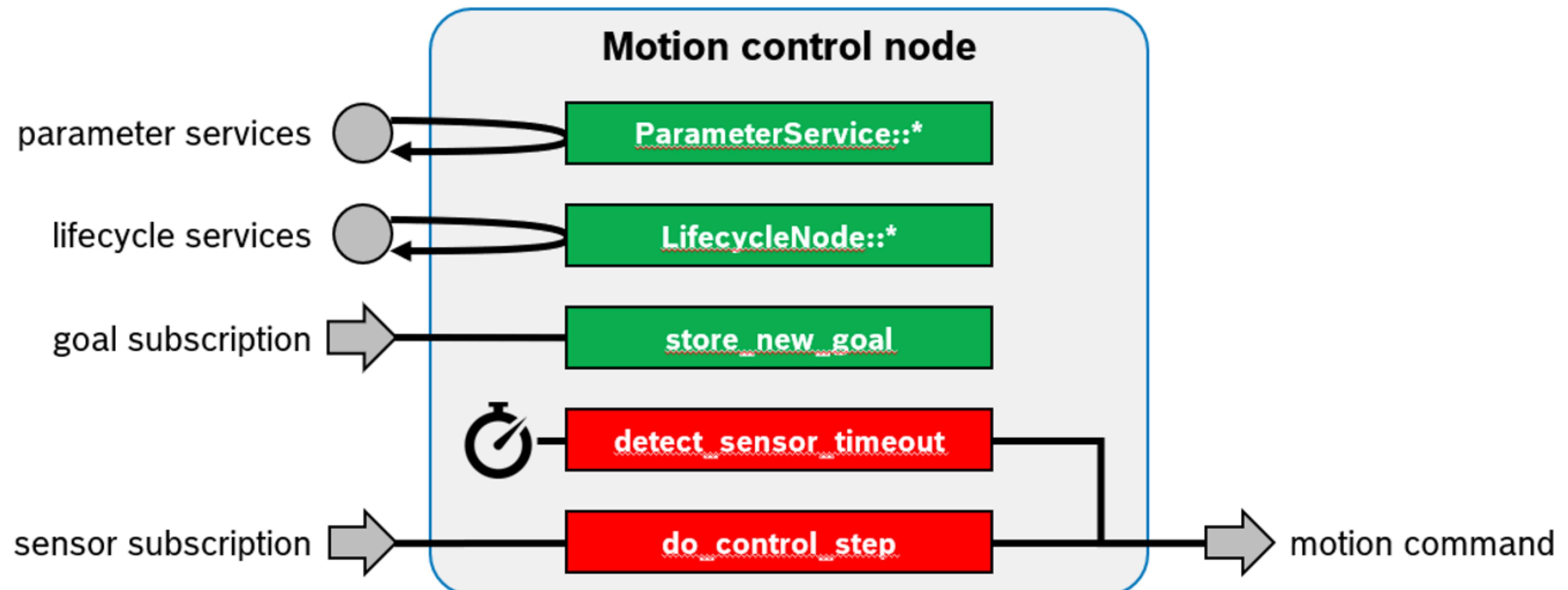
- **Reduces response time**



Ejecutar el ejecutor predeterminado en un thread normal
Ejecutar el ejecutor en Tiempo Real en un thread en Tiempo Real

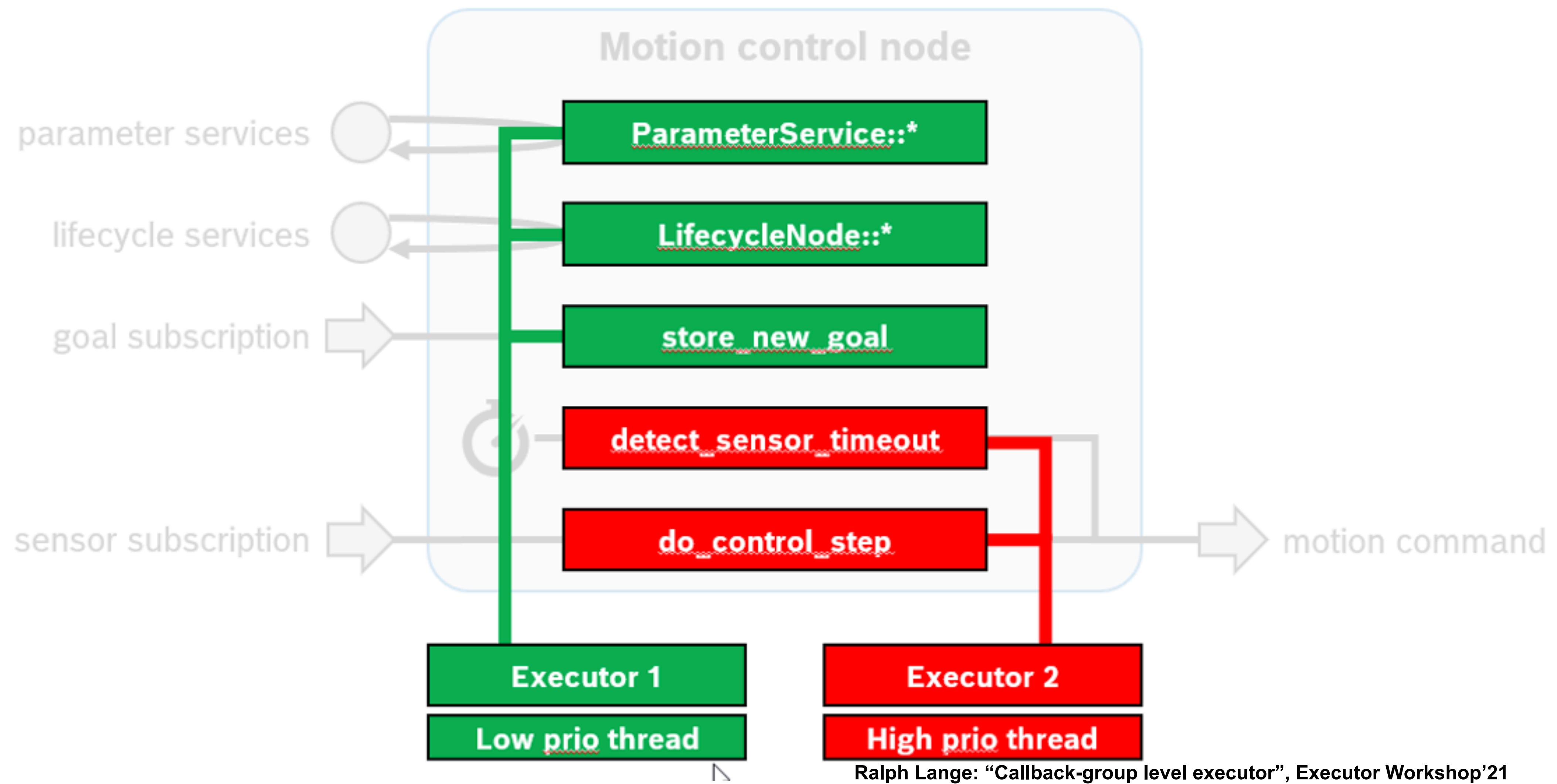
https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_real_time_executor.cpp

Escenarios de aplicación: Nodo con criticidad mezclada



Ralph Lange: "Callback-group level executor", Executor Workshop'21

Escenarios de aplicación: Nodo con criticidad mezclada



Escenarios de aplicación: Nodo con criticidad mezclada

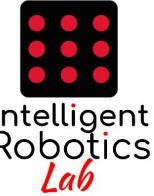
```
class MinimalSubscriber : public rclcpp::Node{
public:
MinimalSubscriber(): Node("minimal_sub"){
subscription1_ = this->create_subscription<>(...)
realtime_callback_group_ = this->create_callback_group(
rclcpp::CallbackGroupType::MutuallyExclusive, false);

rclcpp::SubscriptionOptions subscription_options;
subscription_options.callback_group = realtime_callback_group_;
subscription2_ = this->create_subscription<std_msgs::msg::String>(
"topic_rt", 10, sub_callback_fn,
subscription_options);
}

private:
rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription1_;
rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription2_;
rclcpp::CallbackGroup::SharedPtr realtime_callback_group_;
```

- Añade un callback group
- Añádelo a subscription_options
- Crea la subscripción

https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_callback_group.cpp



Escenarios de aplicación: Nodo con criticidad mezclada

```
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);

    auto node_sub = std::make_shared<MinimalSubscriber>();

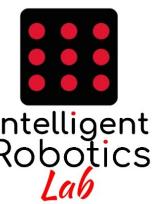
    rclcpp::executors::StaticSingleThreadedExecutor default_executor;
    rclcpp::executors::StaticSingleThreadedExecutor realtime_executor;

    default_executor.add_node(node_sub);

    realtime_executor.add_callback_group(
        node_sub->get_realtime_callback_group(), node_sub->get_node_base_interface()));

    ...
}
```

- Dos ejecutores
- Añade el callback group a un ejecutor



https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_callback_group.cpp

Escenarios de aplicación: Nodo con criticidad mezclada

...

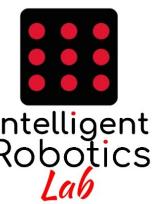
```
// spin real-time tasks in a separate thread
auto realtime_thread = std::thread(
    [&]() {
        realtime_executor.spin();
    });
set_thread_scheduling(realtime_thread.native_handle(), options.policy, options.priority);

default_executor.spin();
realtime_thread.join();

rclcpp::shutdown();
return 0;
```

- Crear un thread de Tiempo Real con un ejecutor de Tiempo Real
- Asigna prioridad de Tiempo Real al thread
- Spin al ejecutor por defecto

https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_callback_group.cpp

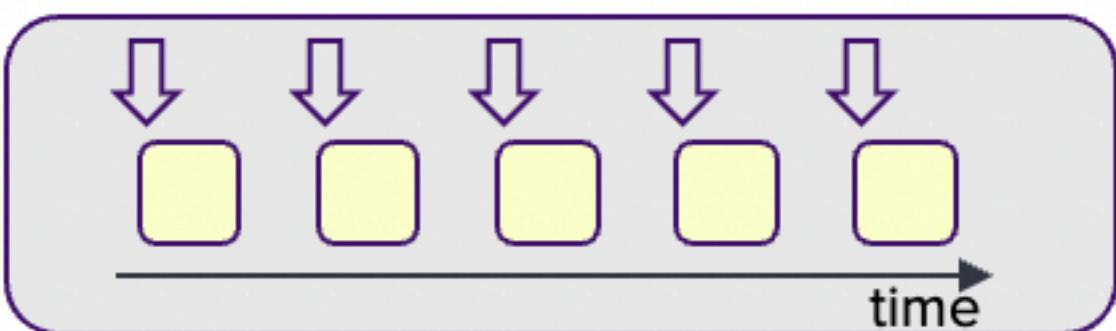


Escenarios de aplicación: Nodo con criticidad mezclada

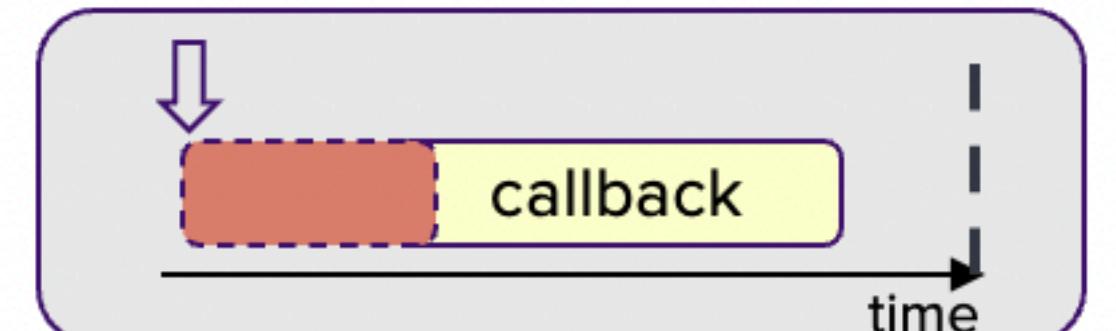
```
....  
// spin real-time tasks in a  
auto realtime_thread = std::  
[&]() {  
    realtime_executor.spin();  
};  
  
set_thread_scheduling(realtime_thread,  
    SCHED_FIFO);  
  
default_executor.spin();  
realtime_thread.join();  
  
rclcpp::shutdown();  
return 0;
```

Benefits:

- Improves bounded update rate



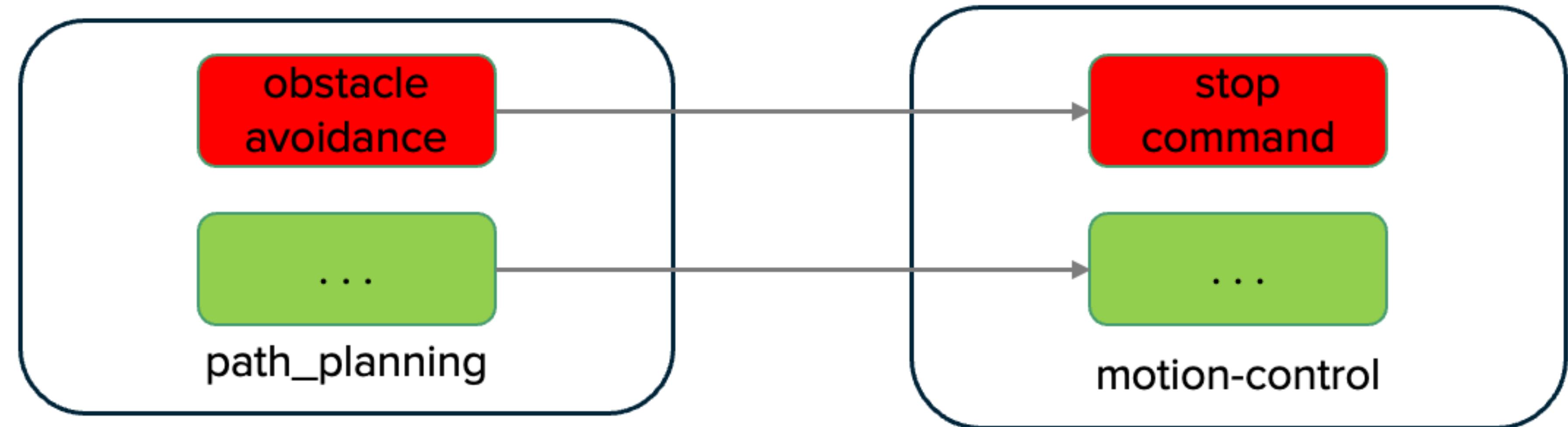
- Reduces response time



ar un thread de Tiempo Real
un el ejecutor de Tiempo Real
na prioridad de Tiempo Real al
ad
al ejecutor por defecto

licy, options.priority);

Escenarios de aplicación: Varios Nodos con criticidad mezclada



- Se pueden agregar grupos de callbacks de múltiples nodos al mismo ejecutor y procesarlos en un hilo en Tiempo Real.
- Vea el ejemplo de código completo en GitHub (enlace a continuación)

https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_callback_group.cpp

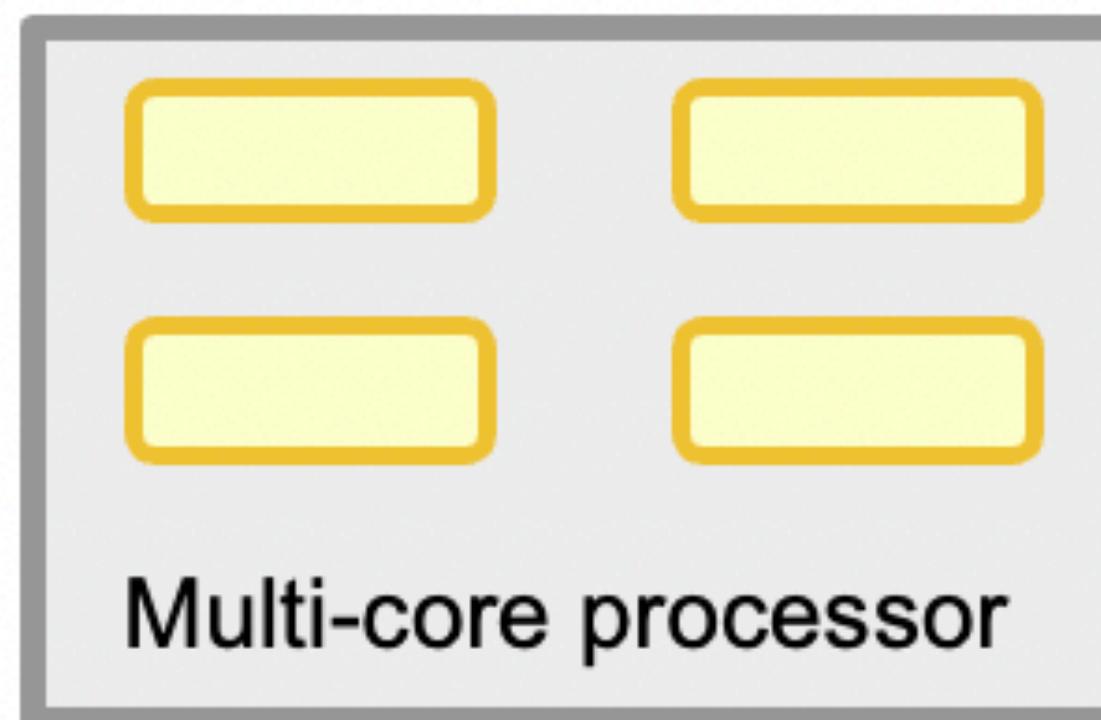
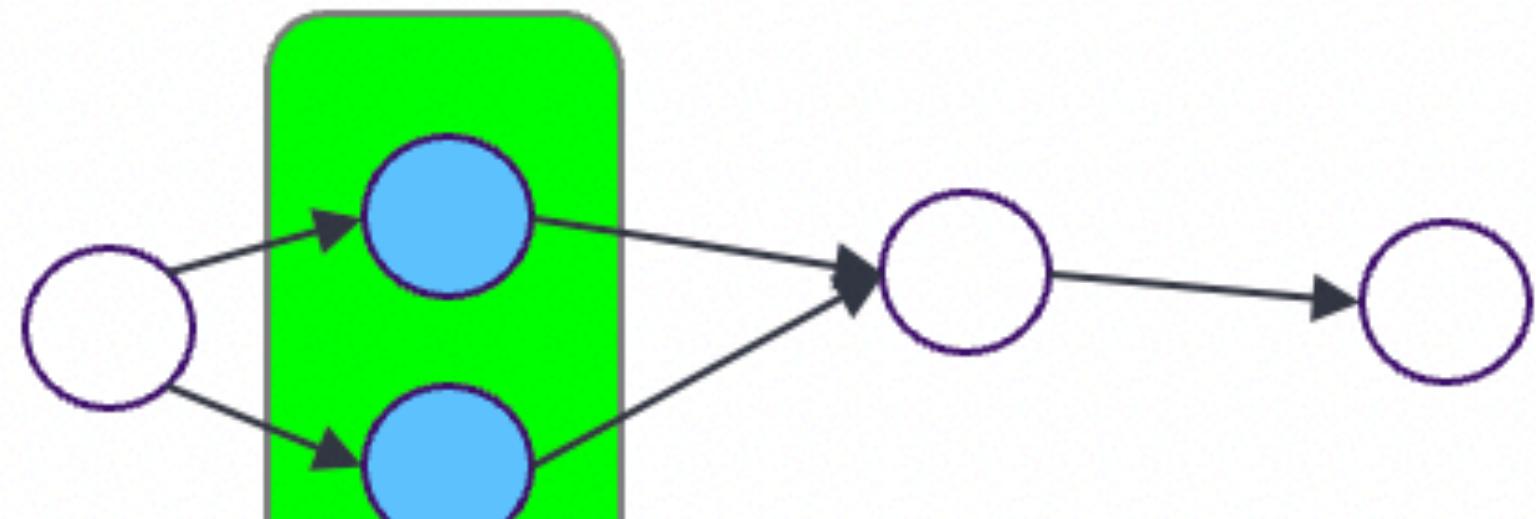
Escenarios de aplicación: Varios Nodos con criticidad mezclada



- Se pueden agregar grupos de ejecutor y procesarlos en un solo en tiempo real.
- Vea el ejemplo de código completo en GitHub (enlace a continuación)

https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_callback_group.cpp

Paralelización en multi-core: un Nodo

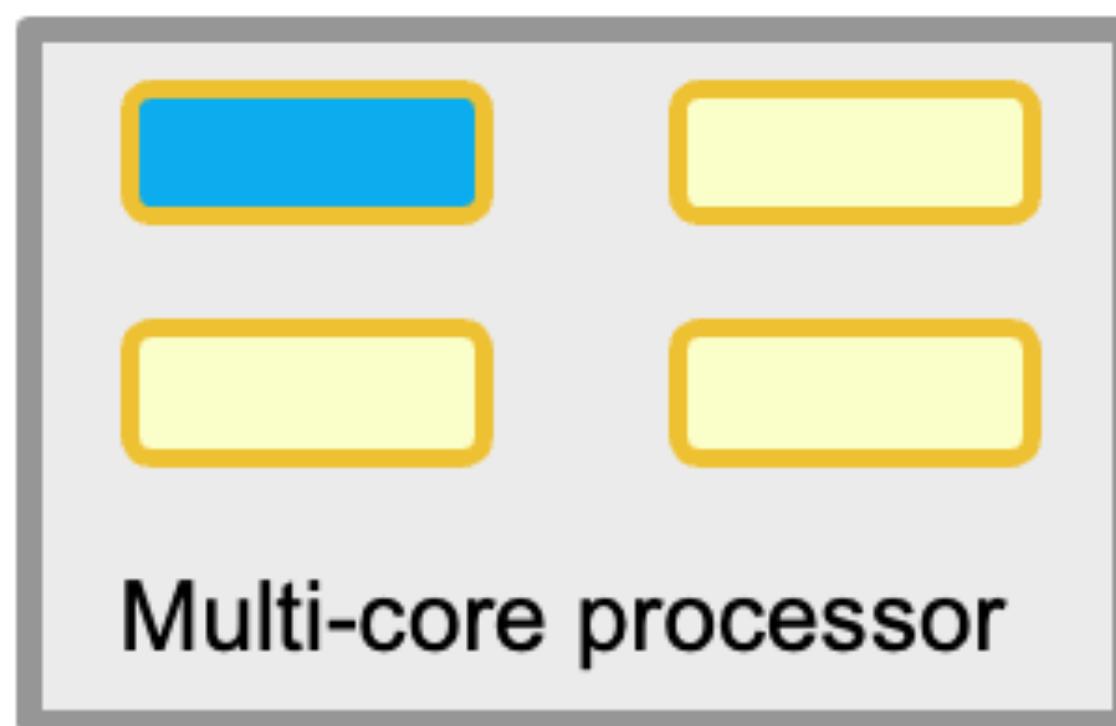
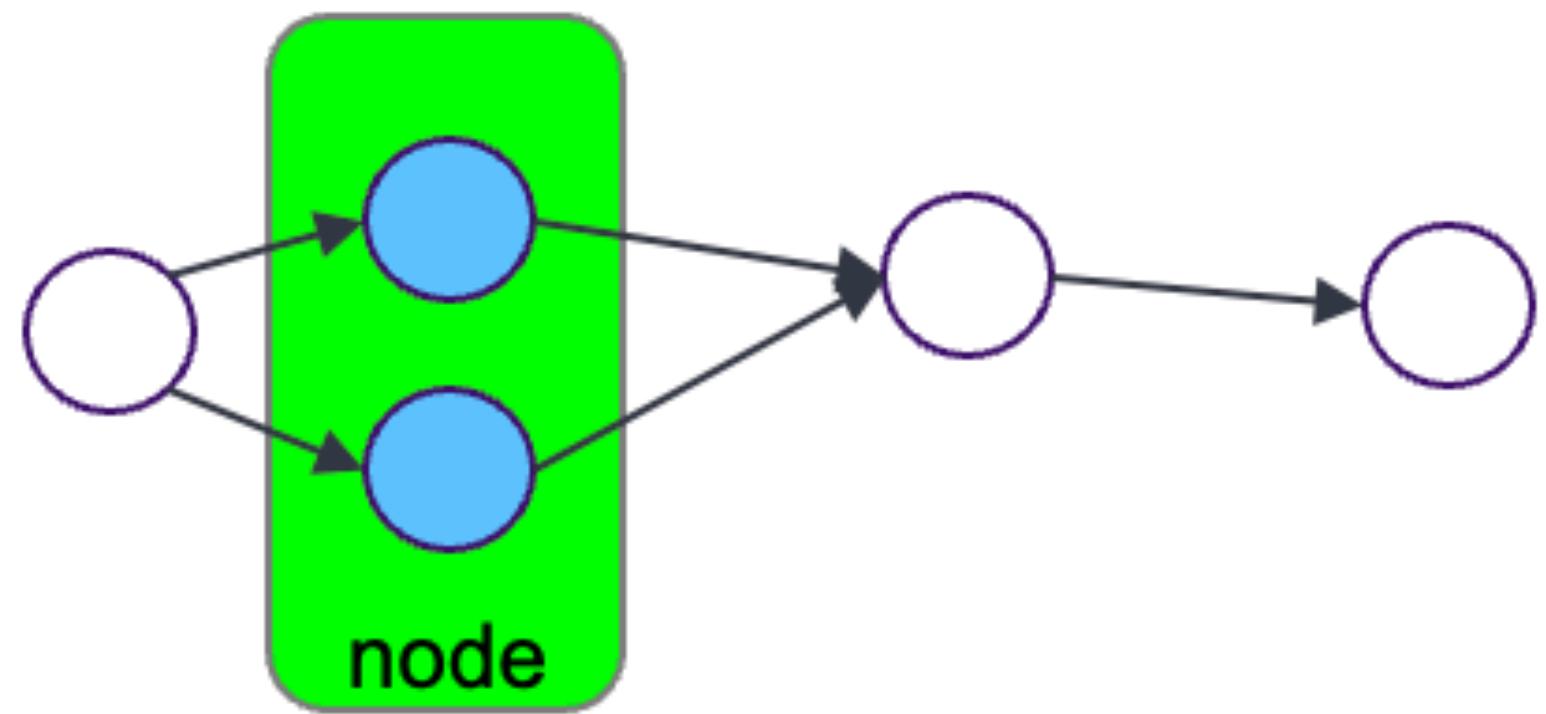


```
int main(int argc, char * argv[])
{
    rclcpp::Node::SharedPtr node = ... (sub_a, sub_b);

    rclcpp::executors::MultiThreadedExecutor executor;
    executor.add_node(node);
    executor.spin();
}
```

<https://docs.ros.org/en/rolling/How-To-Guides/Using-callback-groups.html>

Paralelización en multi-core: dos Nodos



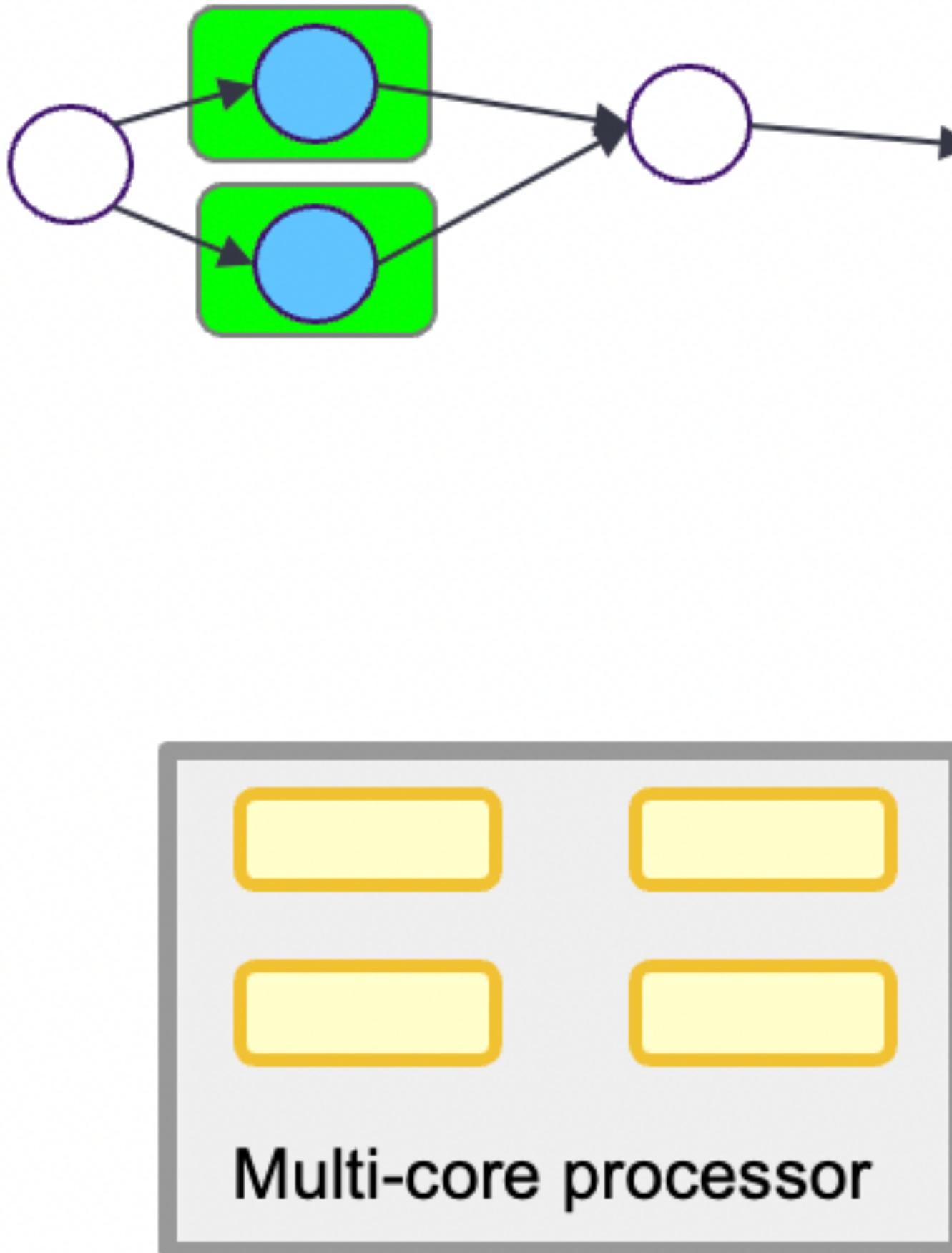
```
int main(int argc, char * argv[])
{
    rclcpp::Node::SharedPtr node = ... (sub_a, sub_b);

    rclcpp::executors::MultiThreadedExecutor executor;
    executor.add_node(node);
    executor.spin();
}
```

¡Error! El callback group implícito es MutuallyExclusive por defecto.

=> todas los callbacks se procesan de manera secuencial

Paralelización en multi-core: dos Nodos

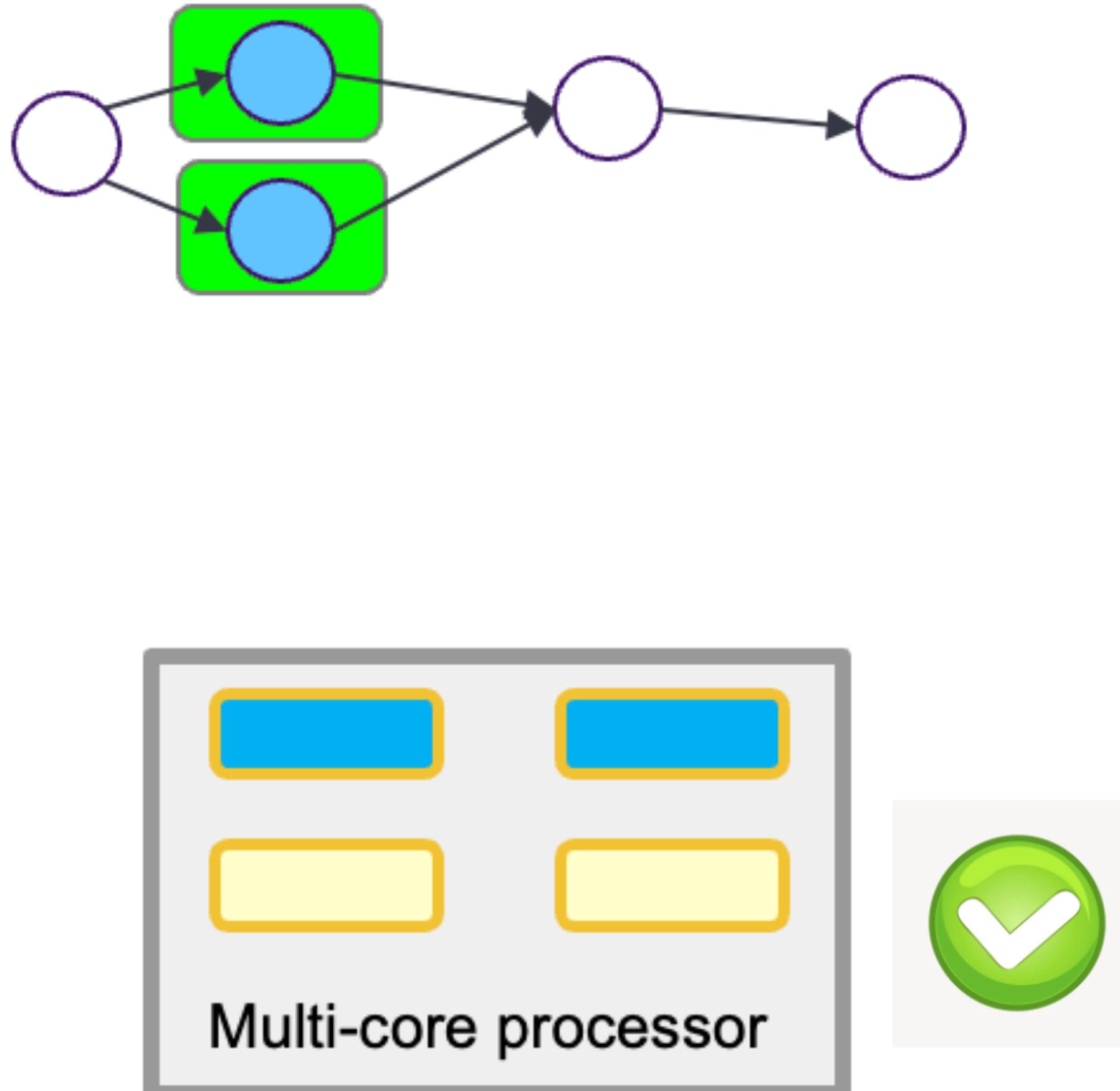


```
int main(int argc, char * argv[])
{
    rclcpp::Node::SharedPtr node1 = ... (sub_a);
    rclcpp::Node::SharedPtr node2 = ... (sub_b);

    rclcpp::executors::MultiThreadedExecutor executor;
    executor.add_node(node1);
    executor.add_node(node2);
    executor.spin();
}
```

<https://docs.ros.org/en/rolling/How-To-Guides/Using-callback-groups.html>

Paralelización en multi-core: dos Nodos



```
int main(int argc, char * argv[])
{
    rclcpp::Node::SharedPtr node1 = ... (sub_a);
    rclcpp::Node::SharedPtr node2 = ... (sub_b);

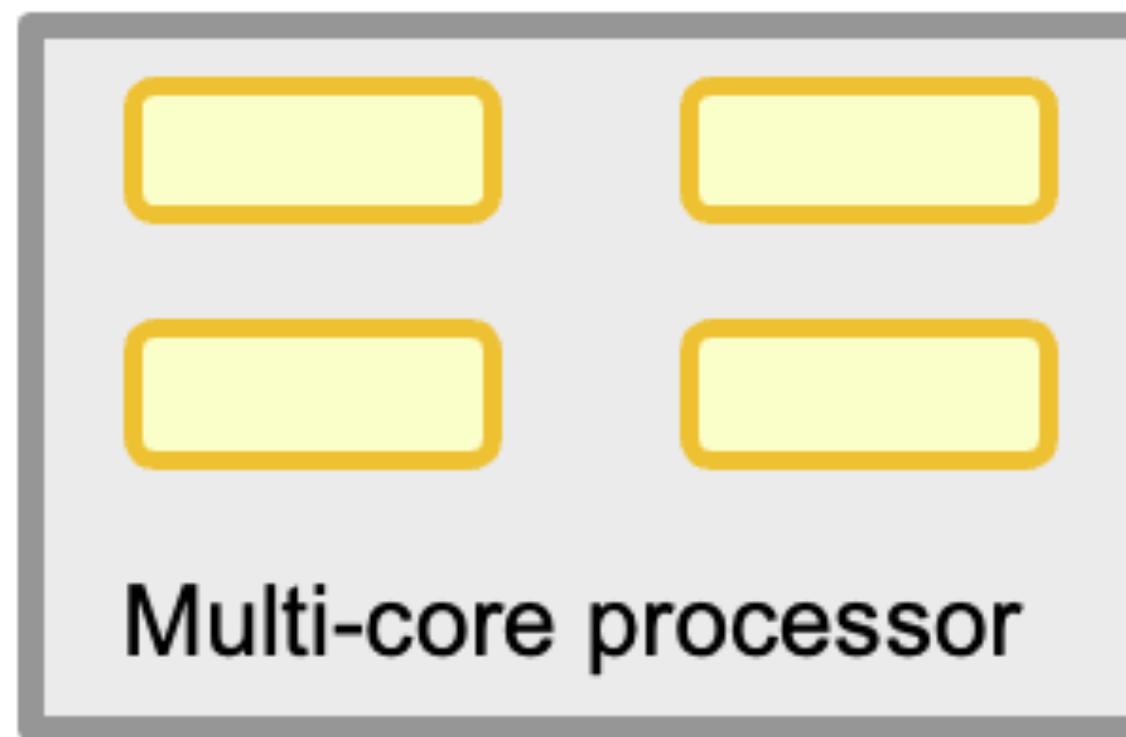
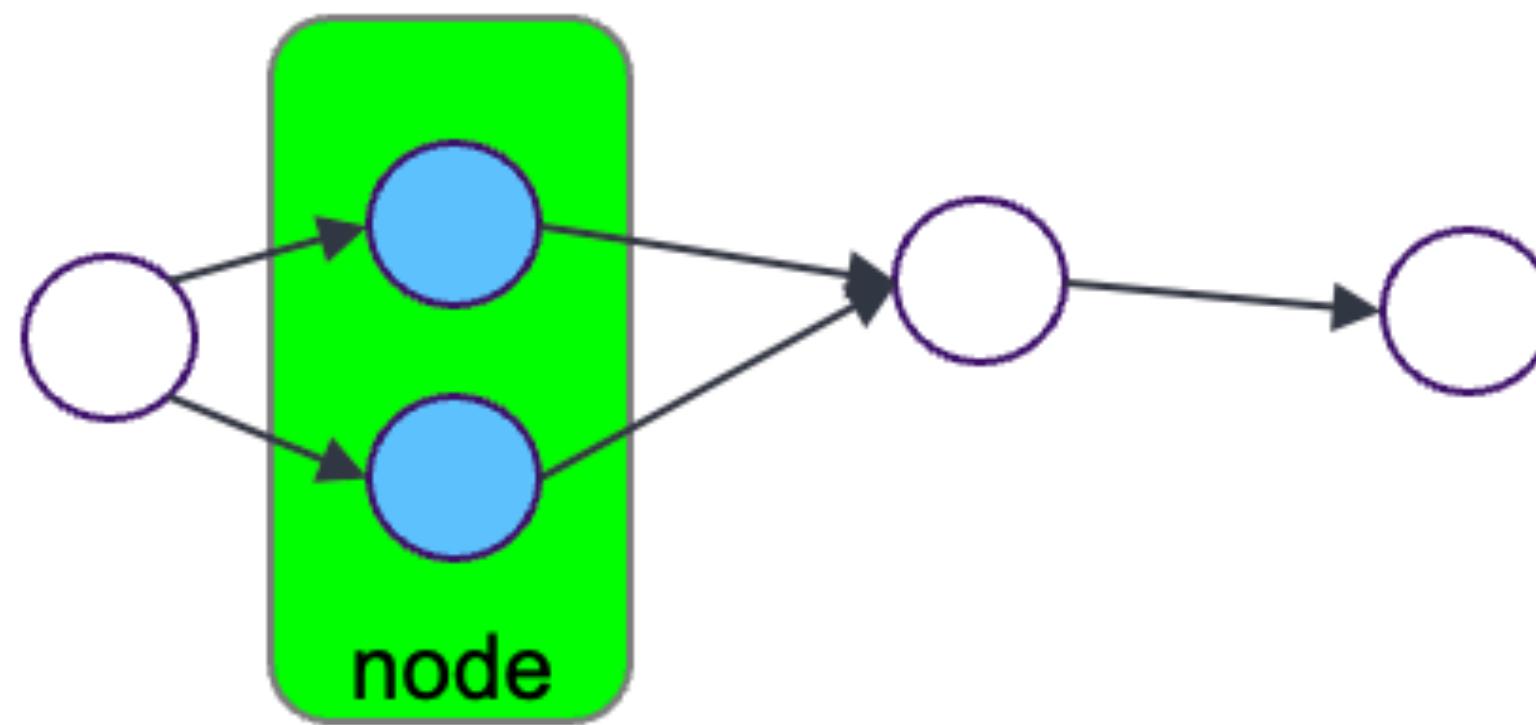
    rclcpp::executors::MultiThreadedExecutor executor;
    executor.add_node(node1);
    executor.add_node(node2);
    executor.spin();
}
```

¡Funciona! Cada nodo tiene un callback group.

=> callbacks paralelizados

<https://docs.ros.org/en/rolling/How-To-Guides/Using-callback-groups.html>

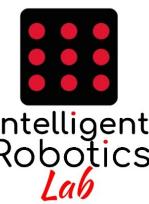
Paralelización en multi-core: un Nodo, dos cb's, dos threads



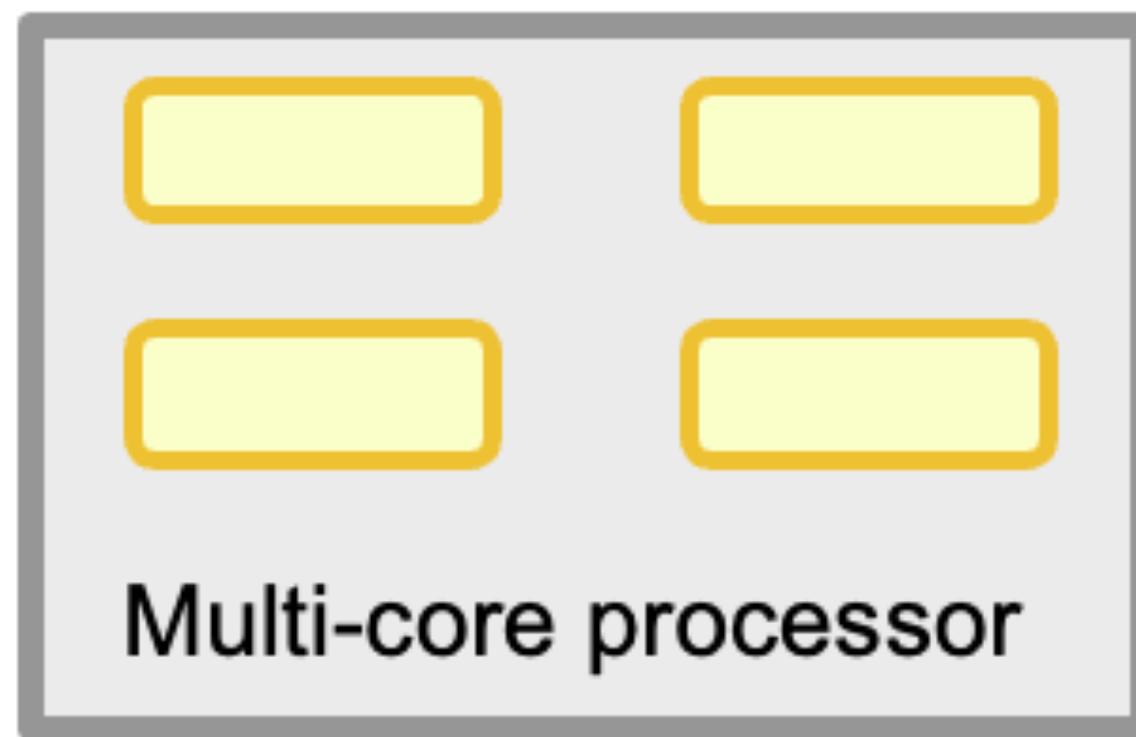
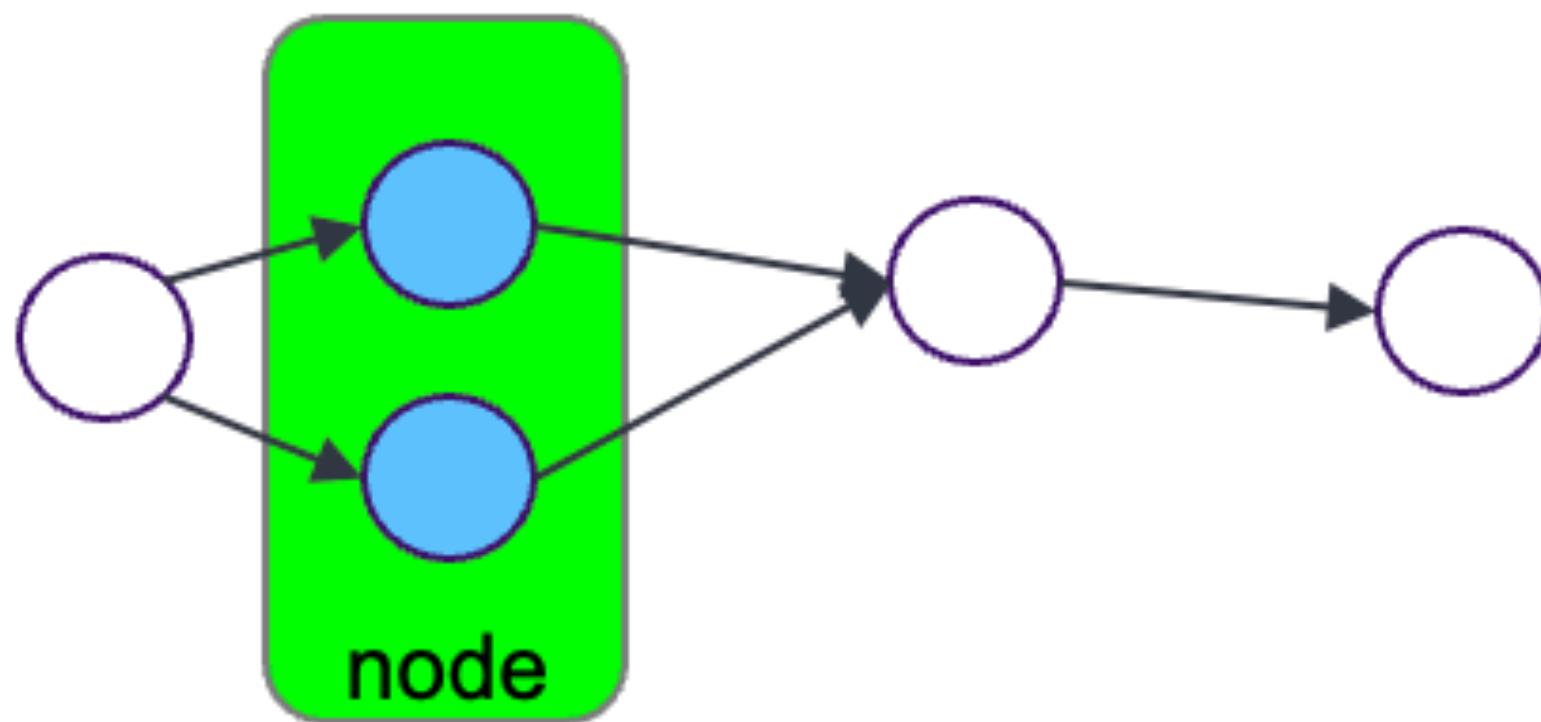
```
class DualThreadedNode : public rclcpp::Node
{
    ...
    callback_group_sub_a = this->create_callback_group(
        rclcpp::CallbackGroupType::MutuallyExclusive);
    callback_group_sub_b = this->create_callback_group(
        rclcpp::CallbackGroupType::MutuallyExclusive);

    auto sub_opt_a = rclcpp::SubscriptionOptions();
    sub_opt_a.callback_group = callback_group_sub_a;
    auto sub_opt_b = rclcpp::SubscriptionOptions();
    sub_opt_b.callback_group = callback_group_sub_b;
    ...
}
```

https://github.com/ros2/examples/tree/humble/rclcpp/executors/multithreaded_executor

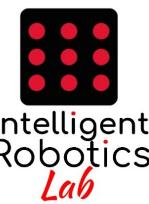


Paralelización en multi-core: un Nodo, dos cb's, dos threads

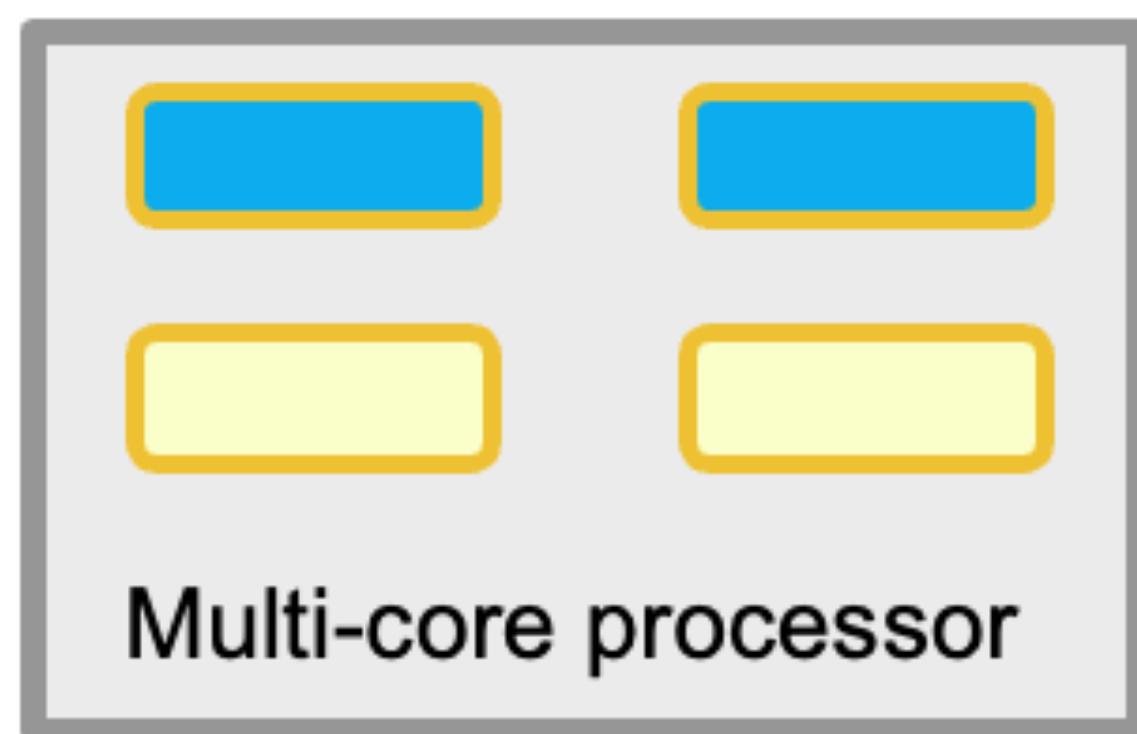
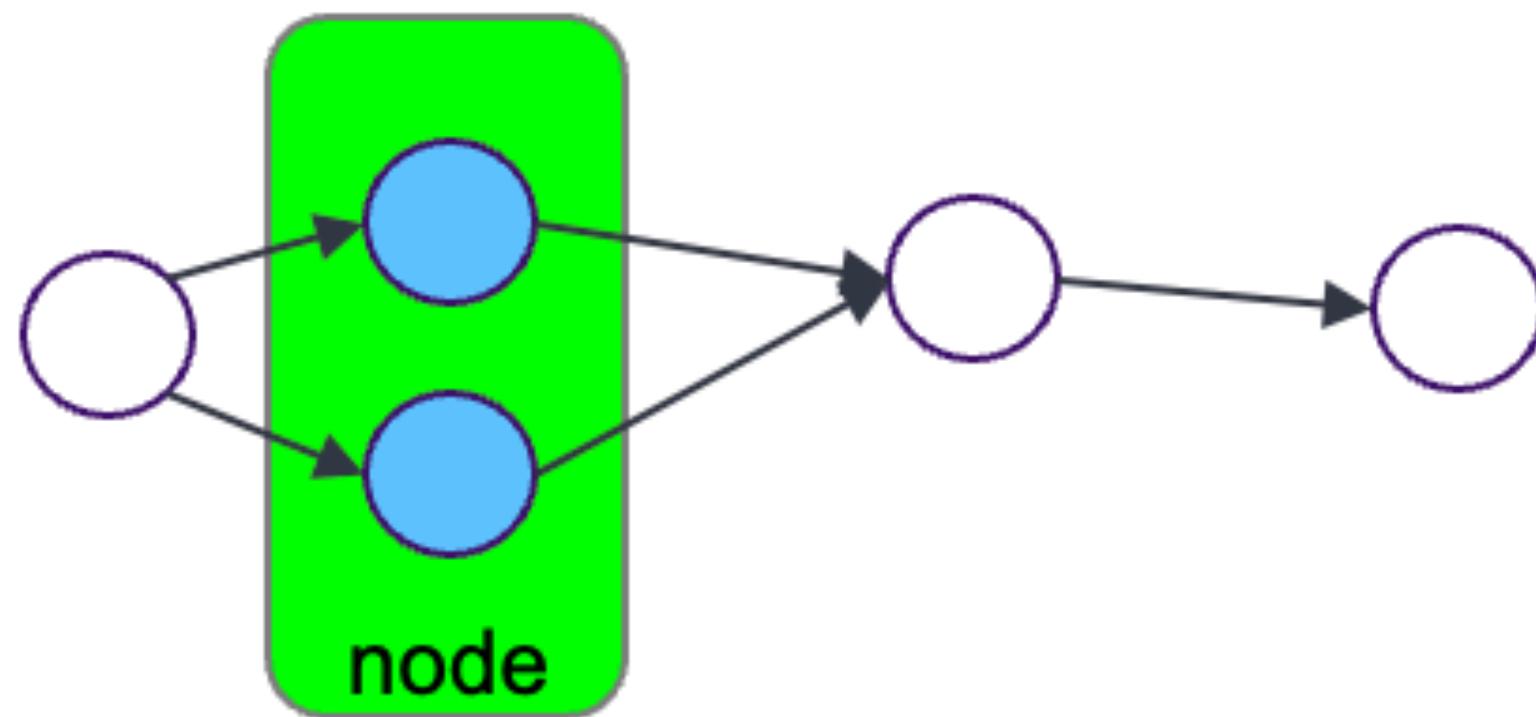


```
class DualThreadedNode : public rclcpp::Node {  
    sub_a = this->create_subscription<std_msgs::msg::String>(  
        "topic_A",..., sub_opt_a);  
    sub_b = this->create_subscription<std_msgs::msg::String>(  
        "topic_B",..., sub_opt_b);  
}  
main(){  
    rclcpp::init(argc, argv);  
    rclcpp::executors::MultiThreadedExecutor executor;  
    auto node1 = std::make_shared<DualThreadedNode>();  
    executor.add_node(node1);  
    executor.spin();  
}
```

https://github.com/ros2/examples/tree/humble/rclcpp/executors/multithreaded_executor



Paralelización en multi-core: un Nodo, dos cb's, dos threads



```
class DualThreadedNode : public rclcpp::Node {  
    sub_a = this->create_subscription<std_msgs::msg::String>(“topic_A”,..., sub_opt_a);  
    sub_b = this->create_subscription<std_msgs::msg::String>(“topic_B”,..., sub_opt_b);  
}  
main(){  
    rclcpp::init(argc, argv);  
    rclcpp::executors::MultiThreadedExecutor executor;  
    auto node1 = std::make_shared<DualThreadedNode>();  
    executor.add_node(node1);  
    executor.spin();
```

¡Funciona! Cada suscripción tiene su callback group.

=> callbacks paralelizados

https://github.com/ros2/examples/tree/humble/rclcpp/executors/multithreaded_executor

Optimizar el orden de procesamiento



```
rclcpp::Node::SharedPtr node1 = ...  
rclcpp::Node::SharedPtr node2 = ...  
rclcpp::Node::SharedPtr node3 = ...  
  
rclcpp::executors::StaticSingleThreadedExecutor  
executor;  
executor.add_node(node1);  
executor.add_node(node2);  
executor.add_node(node3);  
executor.spin();
```

A large green arrow points from the code block to the right side of the slide, indicating a connection between the implementation and the conceptual diagram.

- El orden de procesamiento depende del orden en las estructuras internas del Nodo
 - el tiempo de creación de objetos
 - el orden de creación de suscripciones (dentro de un nodo)
- Pero es difícil de mantener en proyectos más grandes

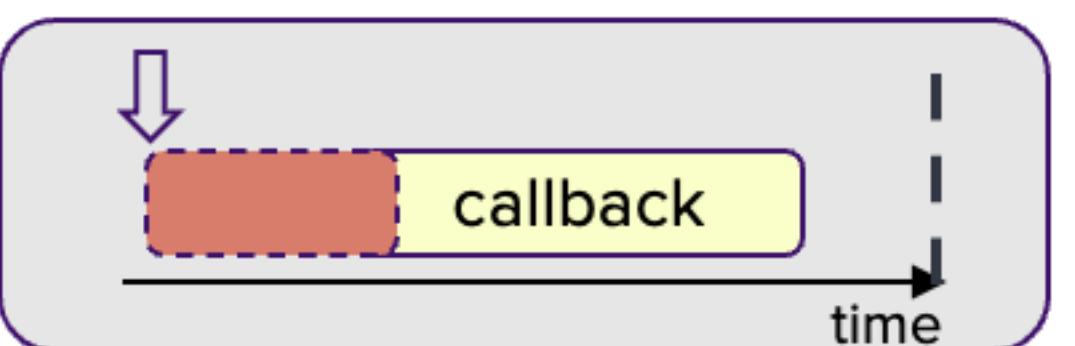
Optimizar el orden de procesamiento



```
rclcpp::Node::SharedPtr node1  
rclcpp::Node::SharedPtr node2  
rclcpp::Node::SharedPtr node3  
  
rclcpp::executors::StaticSing  
executor;  
executor.add_node(node1);  
executor.add_node(node2);  
executor.add_node(node3);  
executor.spin();
```

Benefits:

- Reduces response time



mas grandes

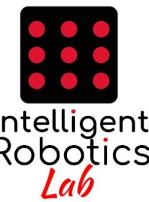
Reducción de la latencia de la comunicación: mensaje prestado

- Mensajes prestados
 - Normalmente, el publisher asigna memoria para el mensaje publicado, que se copia en el middleware
 - Mensaje prestado: el publisher escribe directamente en la estructura de datos propiedad del middleware
 - Beneficio: reduce la latencia de la comunicación
- Ejemplo: publisher que utiliza un mensaje prestado: https://design.ros2.org/articles/zero_copy.html

```
size_t count_ = 1;
pod_pub_ = this->create_publisher<std_msgs::msg::Float64>("chatter_pod",
qos);
```

```
auto pod_loaned_msg = pod_pub_->borrow_loaned_message();
auto pod_msg_data = static_cast<double>(count_);
pod_loaned_msg.get().data = pod_msg_data;
pod_pub_->publish(std::move(pod_loaned_msg));
```

https://github.com/ros2/demos/blob/rolling/demo_nodes_cpp/src/topics/talker_loaned_message.cpp



Reducción de la latencia de la comunicación: middleware de copia cero

- Los sensores de radar, cámara y lidar generan grandes cantidades de datos con alta frecuencia
- La transferencia eficiente de grandes volúmenes de datos es importante, por ejemplo, en la conducción autónoma
- El middleware Zero Copy utiliza memoria compartida para la comunicación
- Compatible con muchas implementaciones de DDS
 - FastDDS
 - Cyclone DDS,
 - ...
- Otros middleware compatibles:
 - Iceoryx: middleware de comunicación entre procesos (IPC), automotriz
 - Zenoh: protocolo de publicación/suscripción/consulta, actualmente compatibilidad con ROS 2 desarrollada como alternativa a DDS

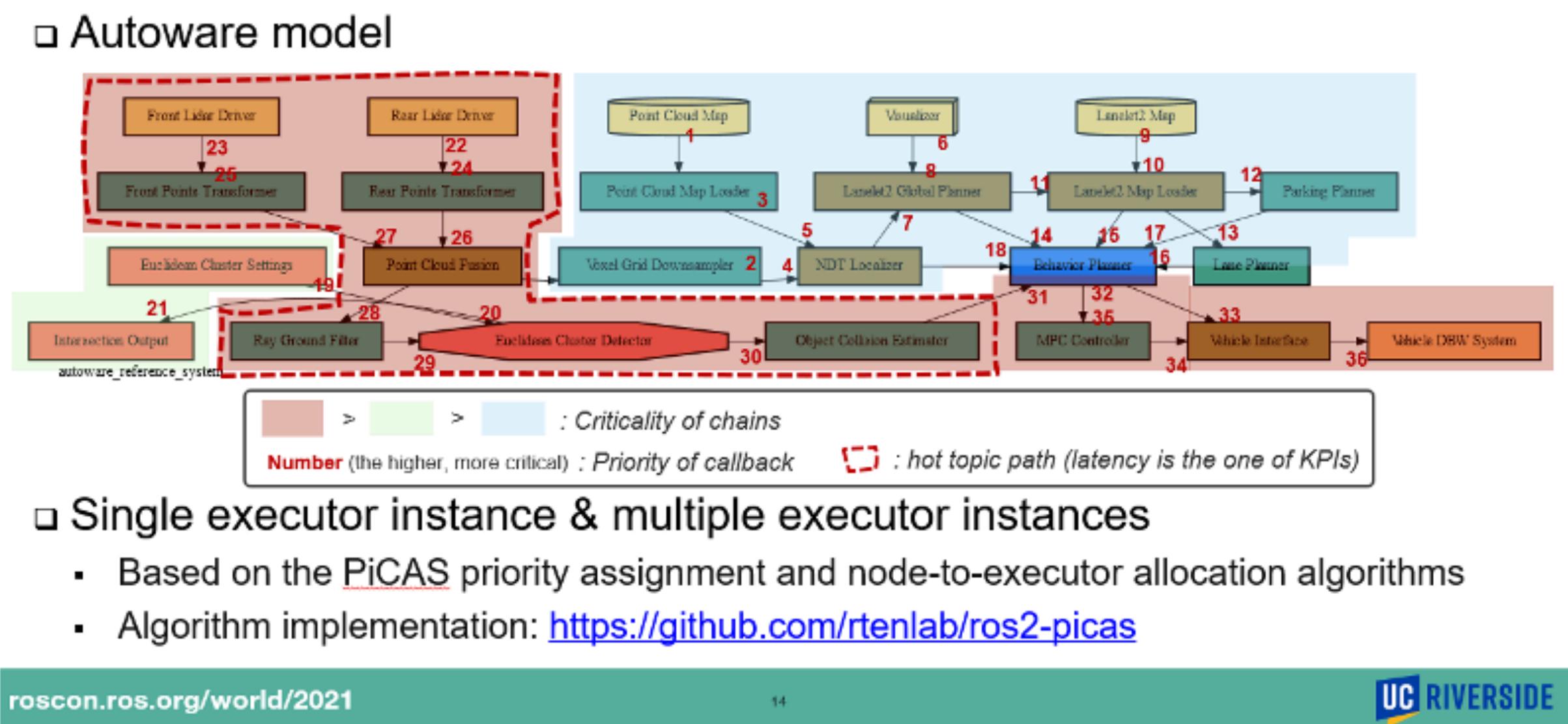


PiCAS: Priority-driven chain-aware scheduling

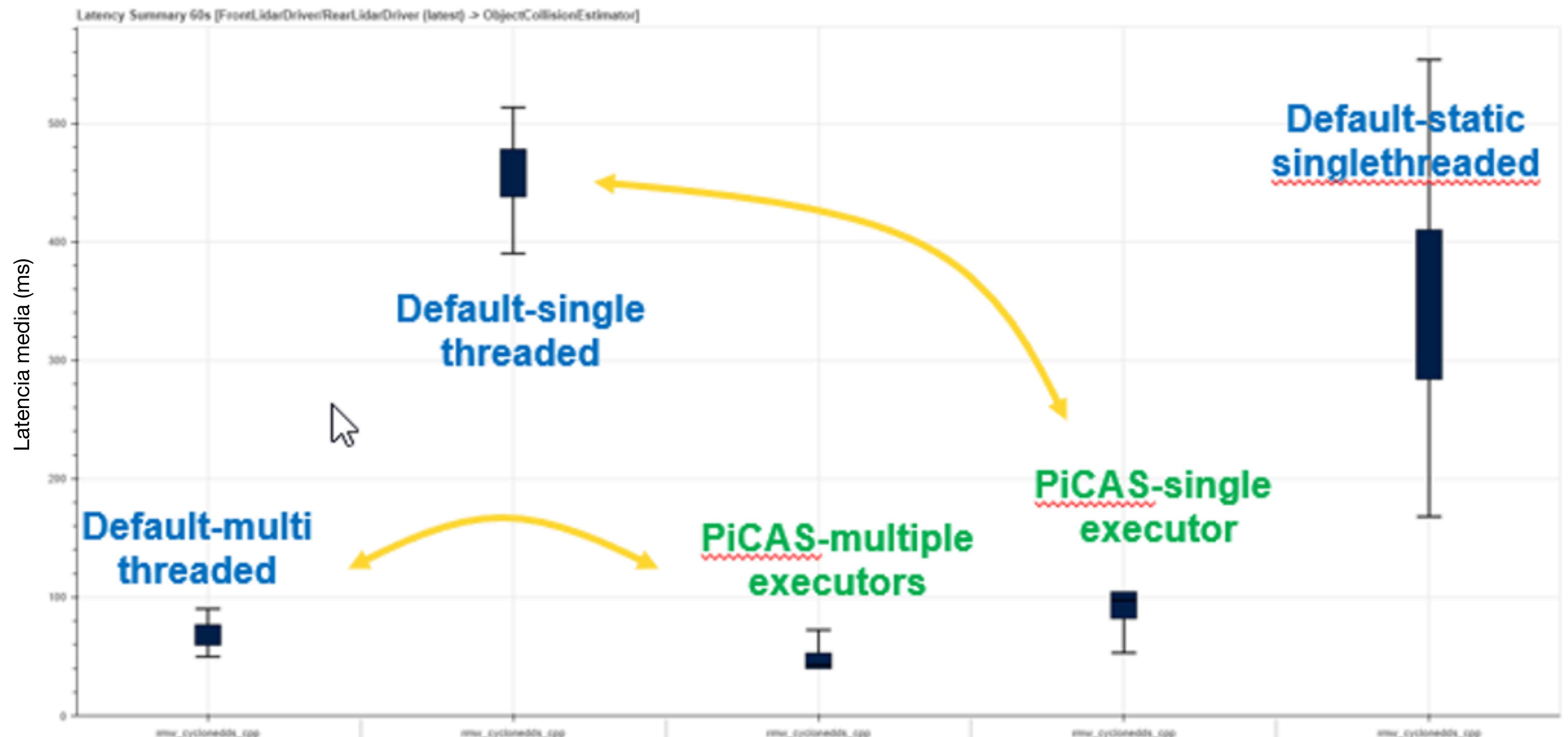
 **PiCAS: Priority-driven Chain-Aware Scheduling framework for ROS2**

- Key idea: enables **prioritization of mission-critical chains** across complex abstraction layers of ROS 2
 - To minimize end-to-end latency
 - To ensure predictability even when the system is overloaded
- PiCAS:** Executor + Resource Allocation Algorithms + Timing Analysis
 - PiCAS executor:** priority-driven callback scheduling
 - Resource allocation algorithms**
 - Callback Priority Assignment
 - Chain-Aware Node-to-Executor Allocation
 - Executor Priority Assignment
 - Backed by **formal end-to-end latency analysis**

roscon.ros.org/world/2021 5 



PiCAS: Priority-driven chain-aware scheduling



<https://www.apex.ai/roscon-21>

Herramientas y benchmarking

- `ros2_tracing`
 - Recopilación de información de ejecución en tiempo de ejecución: duración de callbacks, pila ROS 2, latencia de extremo a extremo
 - Depende de LTTNG (Linux)
 - Sobrecarga muy baja
 - `ros-realtime/reference_system`
 - Evaluación comparativa de rendimiento
 - Compatible con Raspberry Pi, Linux
 - Imagen de Linux en Tiempo Real para Raspberry Pi
 - Se utiliza en este Workshop

https://github.com/ros2/ros2_tracing

<https://github.com/ros-realtime/reference-system>

<https://github.com/ros-realtime/ros-realtime-rpi4-image>

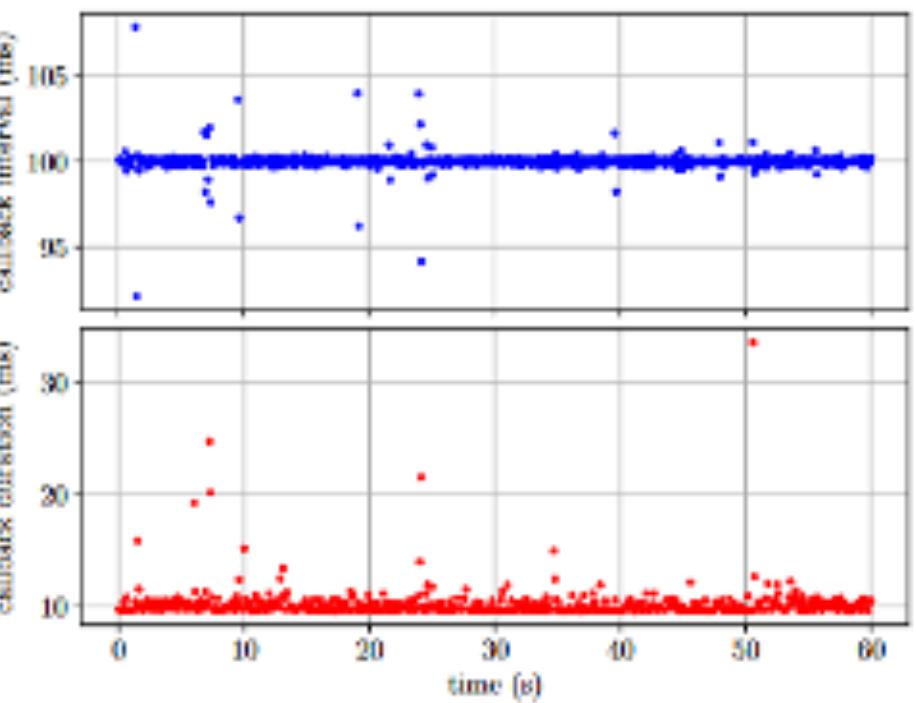
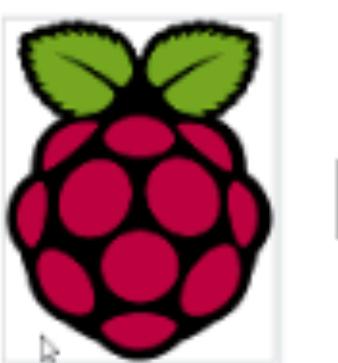
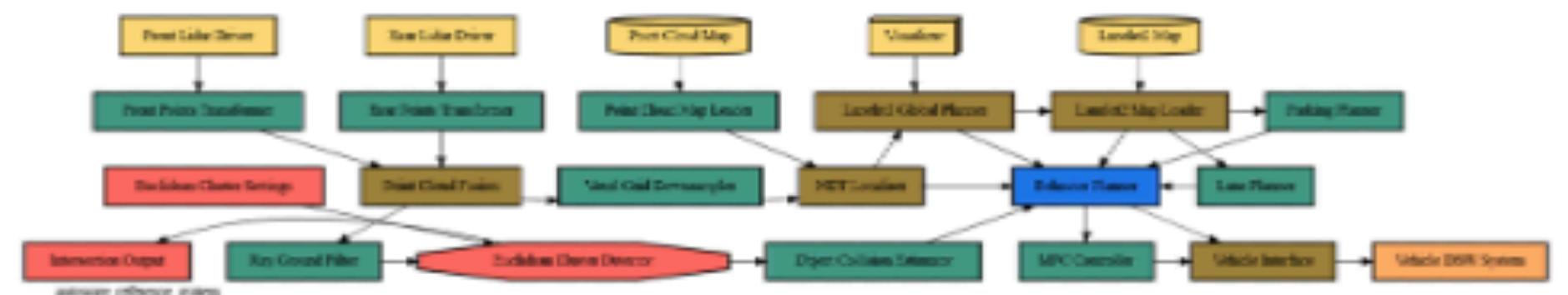
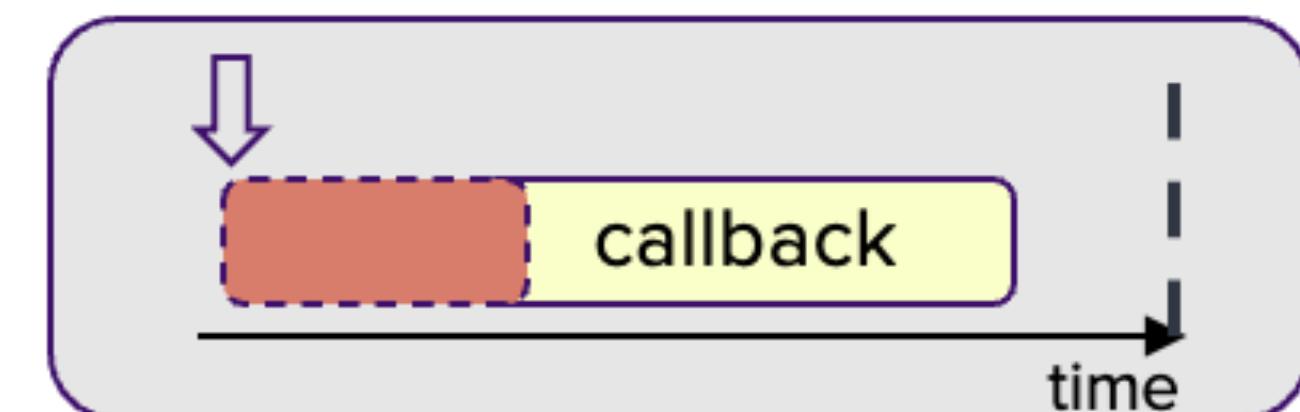


Fig. 4. Example timer callback execution interval (top) and duration (bottom) over time. The callback period is set to 100 ms, while the callback duration depends on the work done. Both contain outliers.



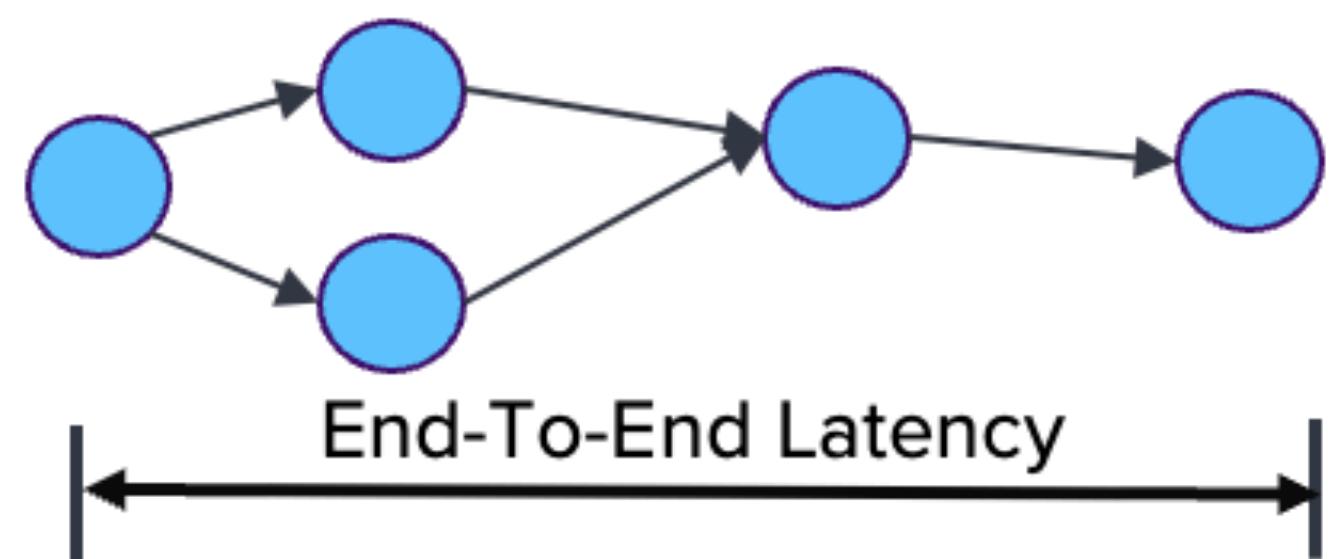
Real-Time Linux

Resumen: Tiempo Real con funciones de ROS 2

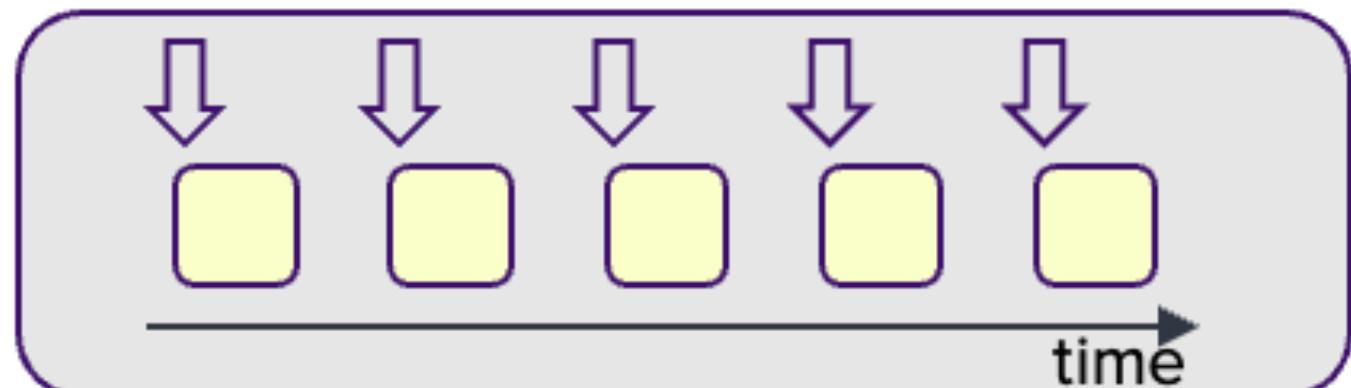


Response Time

- Optimizar el orden de procesamiento de callbacks
- Priorización de nodos
- Priorización con grupos de callbacks



- Priorización de nodos y con grupos de callbacks
- Paralelización con Multi-Threaded Executor
- Mensajes prestados y middleware de copia cero
- PICAS, Events-Executor, ...

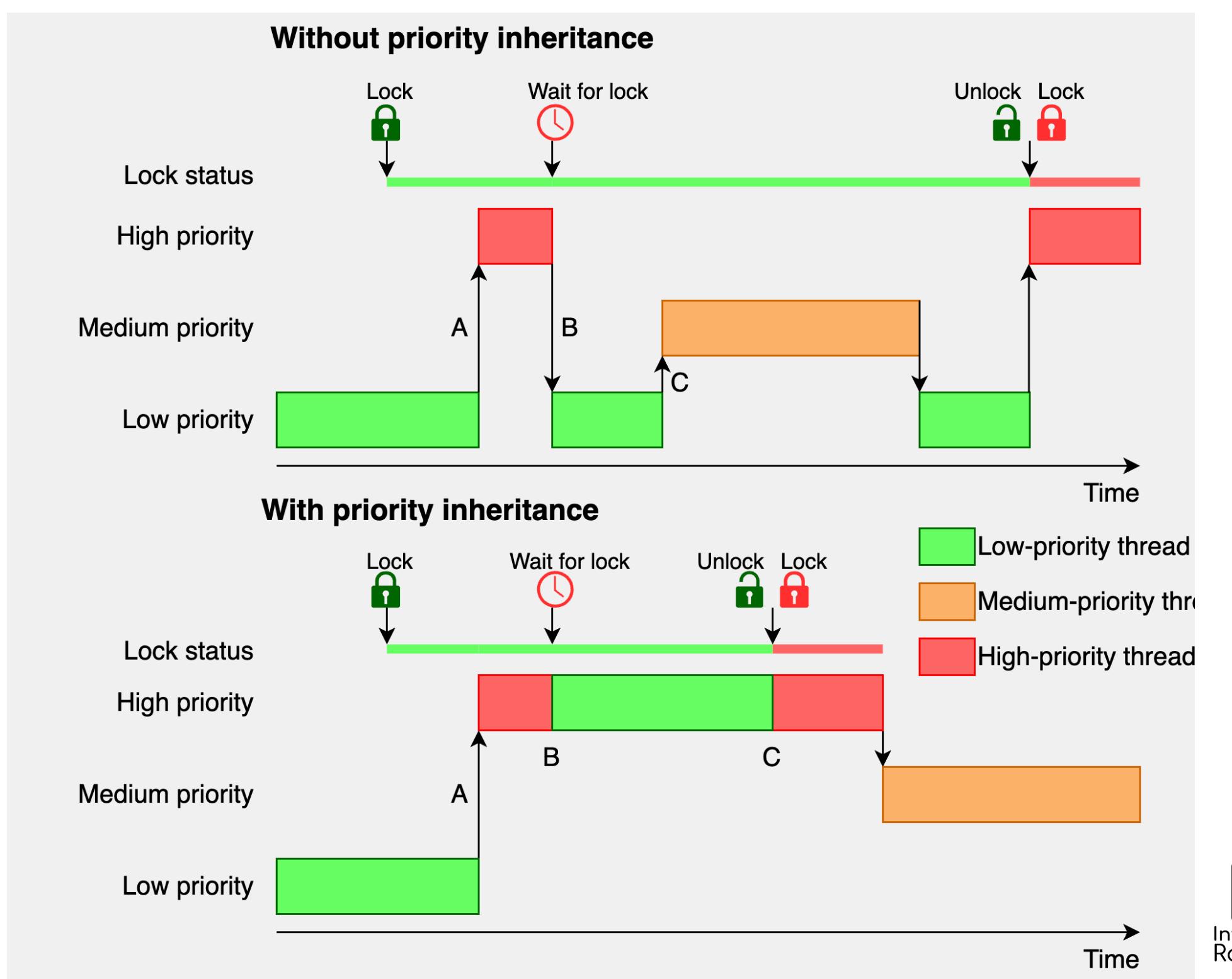


Update Rate

- Priorización de nodos
- Priorización con grupos de callbacks

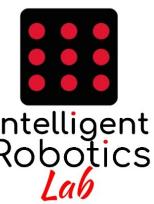
ROS 2 en Tiempo Real: trabajo futuro

- La asignación de memoria dinámica sigue siendo un problema
 - `shared_ptr` en cada callback
 - vectores/tipos de datos STL para todos los mensajes
- Posibles errores de página
- El uso de `std::mutex` en el ejecutor sigue siendo un problema
 - Inversión de prioridad



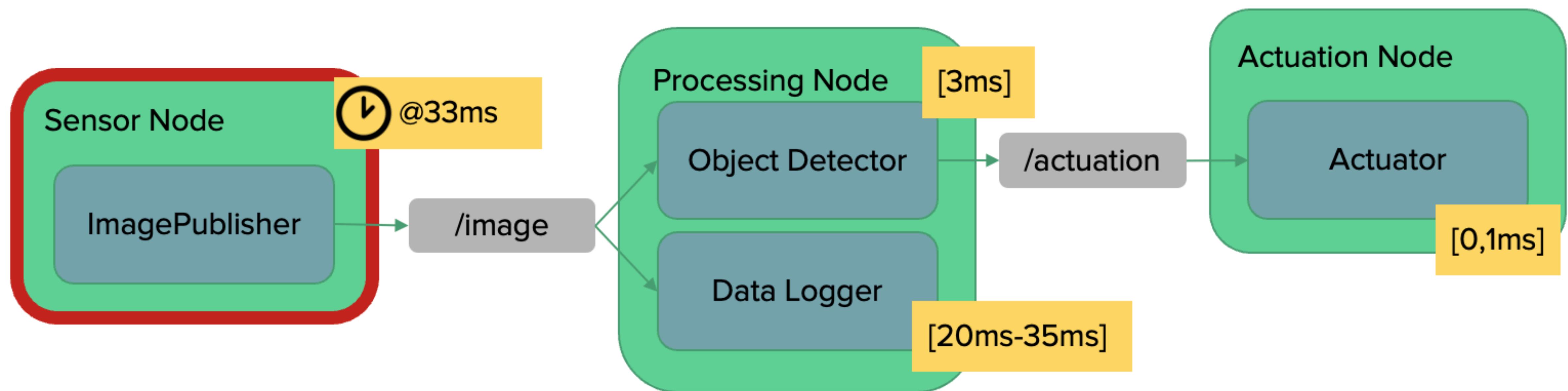
Ejercicio práctico 4:

Nodos ROS 2 en Tiempo Real



Ejercicio 4-1: Nodo ROS 2 en Tiempo Real

- Objetivo: ejecutar el nodo ROS 2 en Tiempo Real utilizando el ejecutor predeterminado en el hilo de Tiempo Real



Ejercicio 4-1: Uso del ejecutor ROS 2 predeterminado

```
int main(int argc, char** argv) {
    rclcpp::init(argc, argv);
    ...
    auto camera_node = std::make_shared<ImagePublisherNode>(camera_tracer, 30.0);
    auto actuation_node = std::make_shared<ActuationNode>(...);
    auto camera_processing_node = std::make_shared<CameraProcessingNode>(...);

    rclcpp::executors::SingleThreadedExecutor executor;

    executor.add_node(camera_node);
    executor.add_node(camera_processing_node);
    executor.add_node(actuation_node);
    executor.spin();
    ...
}
```



Un ejecutor con
tres nodos

exercise4-1/src/camera_demo/src/main.cc

Ejercicio 4-1: Uso del ejecutor ROS 2 predeterminado

```
ImagePublisherNode::ImagePublisherNode(..., double frequency_hz  
) : Node("imagepub"), ... {  
    timer_ = this->create_wall_timer(  
        std::chrono::microseconds(static_cast<int>(1000000 / frequency_hz)),  
        std::bind(&ImagePublisherNode::TimerCallback, this)  
    );  
    publisher_ = this->create_publisher<FakeImage>("/image", 10);  
    ...  
}  
  
void ImagePublisherNode::TimerCallback() {  
    FakeImage img;  
    img.data = {127};  
    ...  
    publisher_->publish(img);  
}
```

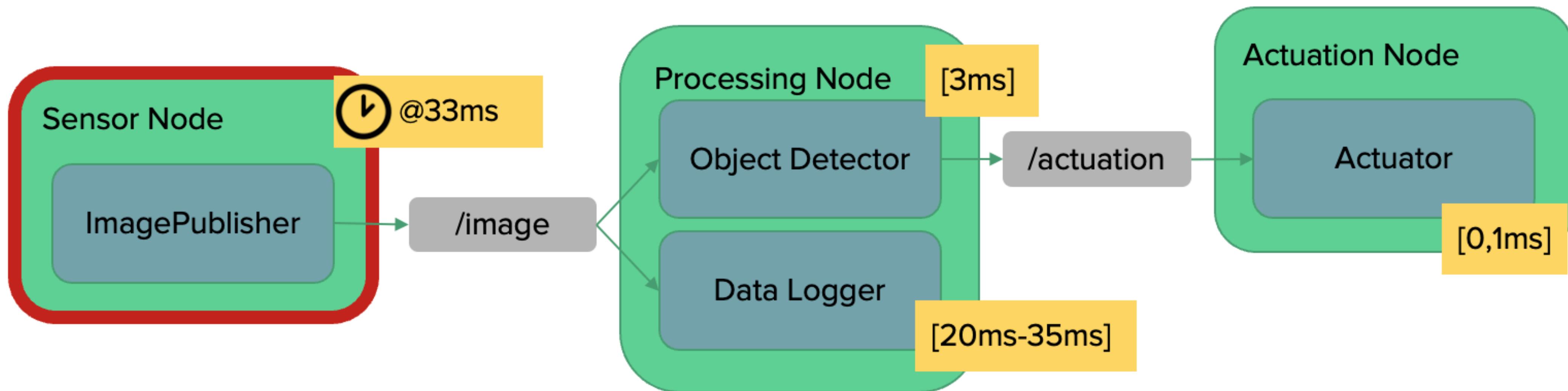
Nodo
Publicador
con un timer

Callback del
timer

exercise4-1/src/camera_demo/src/system_nodes.cc

Ejercicio 4-1: Uso del ejecutor ROS 2 predeterminado

```
void CameraProcessingNode::DataLoggerCallback(const FakeImage::SharedPtr image) {  
    auto span = tracer_data_logger_->WithSpan("DataLogger");  
  
    // variable duration to serialize the data between [20ms,35ms]  
    unsigned int data_logger_latency = 20000 + (rand() % 15001);  
    WasteTime(std::chrono::microseconds(data_logger_latency));  
    . . .  
}
```



exercise4-1/src/camera_demo/src/application_nodes.cc

Ejercicio 4-1: Compilar y ejecutar

En una terminal:

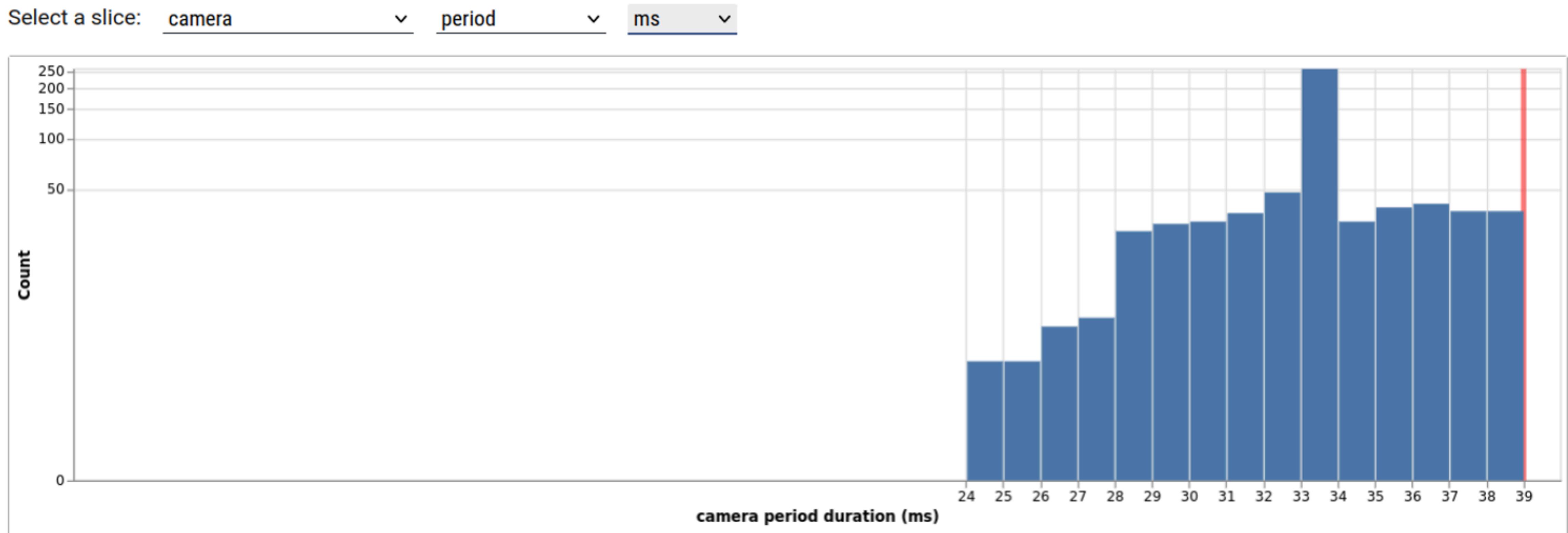
```
docker/shell  
cd exercise4-1  
colcon build  
.run.sh (for 15 sec)
```

Press Ctrl + C

Open perfetto (camera, period, us)



Ejercicio 4-1: Resultado con un Ejecutor de ROS 2 predeterminado



period: [24ms, 39ms]

Ejercicio 4-1: Modifica el código fuente

```

29 // Exercise 4-1 : comment-out next line ←
30 executor.add_node(camera_node);
...
35 // Exercise 4-1 uncomment next block ←
36 rclcpp::executors::SingleThreadedExecutor image_pub_executor;
37 image_pub_executor.add_node(camera_node);
...
41 thr = std::thread([&image_pub_executor] {
42     sched_param sch;
43     sch.sched_priority = 90; ←
44     if (sched_setscheduler(0, SCHED_FIFO, &sch) == -1) {
45         throw std::runtime_error{std::string("failed to set scheduler: ") +
46         std::strerror(errno)};
47     }
48     image_pub_executor.spin();
49 });
54 // Exercise 4-1 (4): uncomment next block ←
55 thr.join();

```

exercise4-1/src/camera_demo/src/main.cc

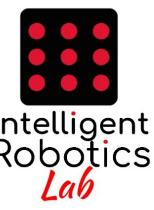
Terminal 1:

```

docker/shell
cd exercise4-1
colcon build
./run.sh (for 15 sec)

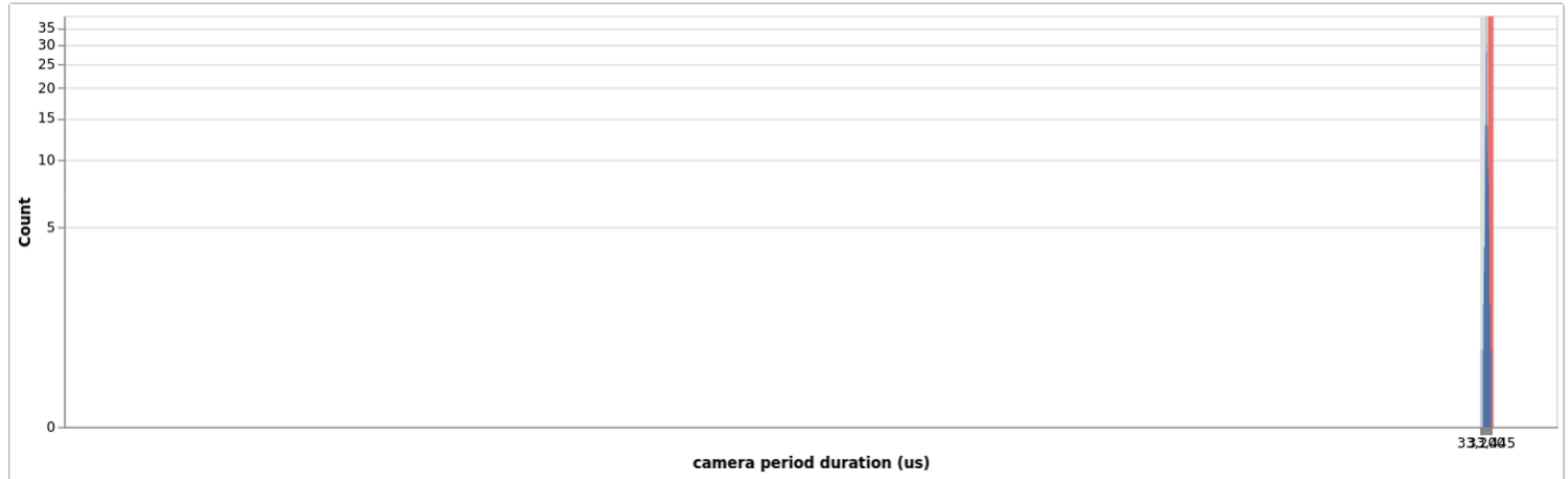
```

Press Ctrl + C
Open perfetto (camera, period, us)



Ejercicio 4-1: Resultado con un Ejecutor en un thread de Tiempo Real

Select a slice: camera ▾ period ▾ us ▾

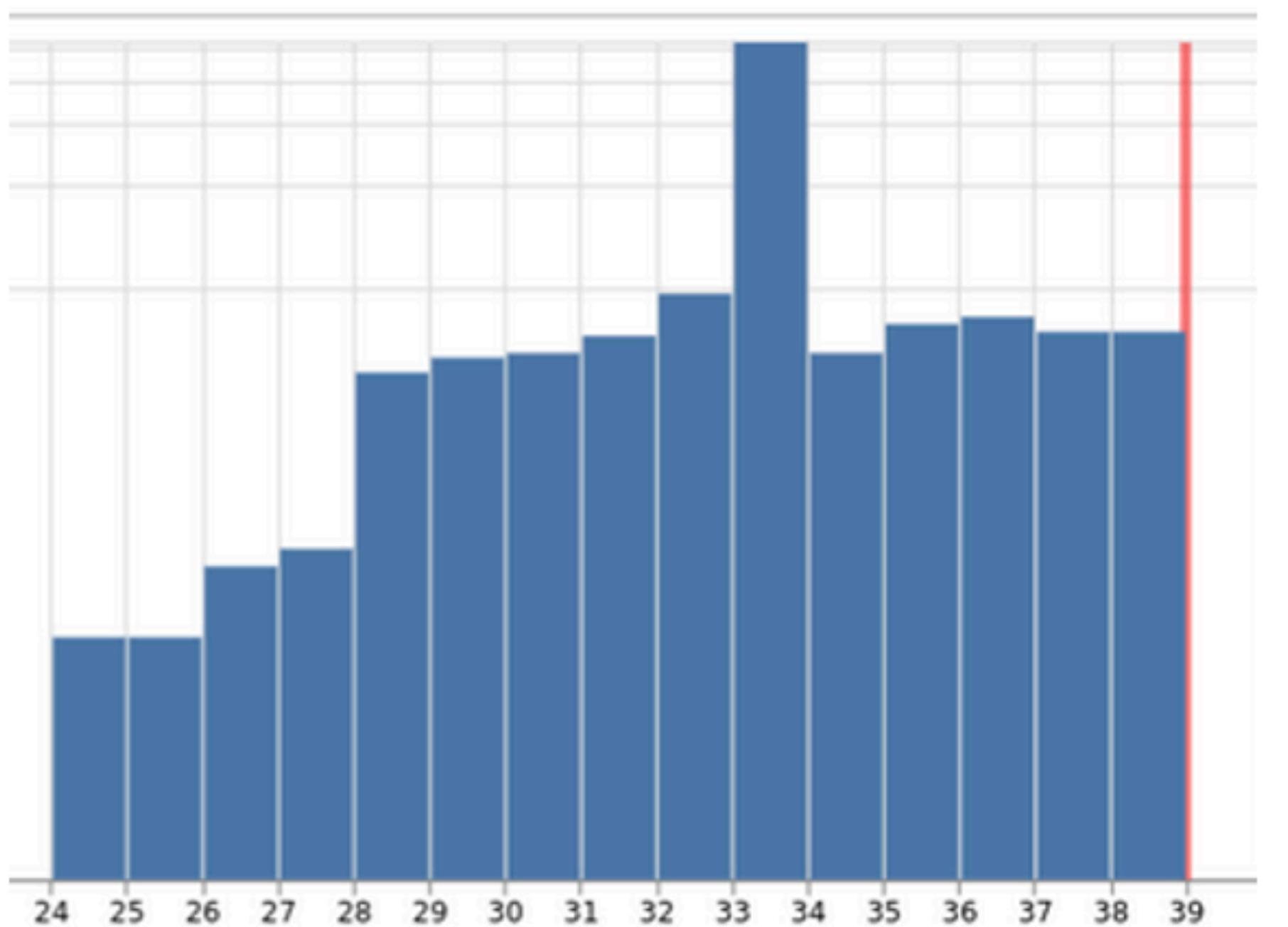


period: [33.2ms, 33.4ms]



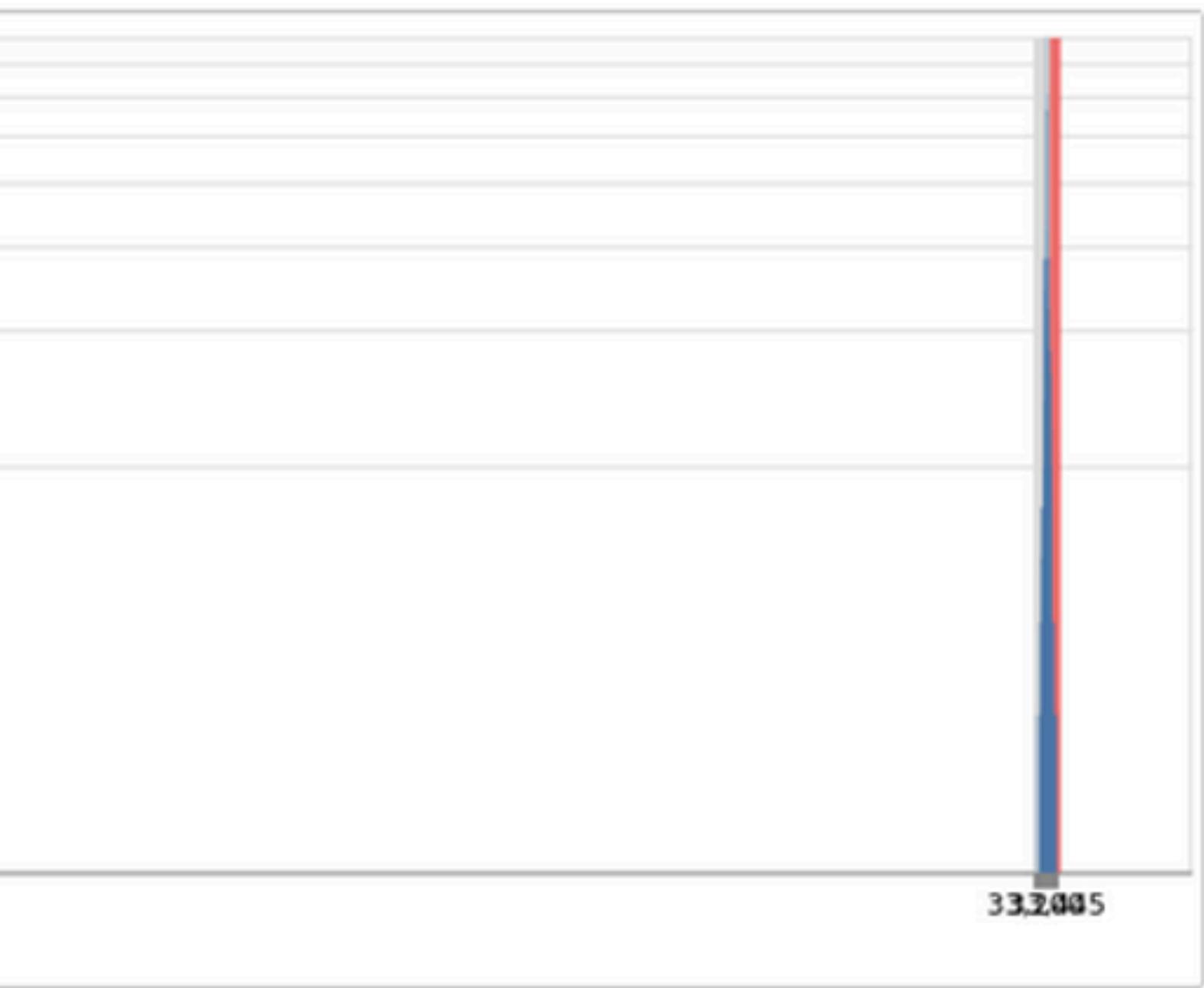
Ejercicio 4-1: Resumen

Publicador ROS 2 predeterminado



Gran variación

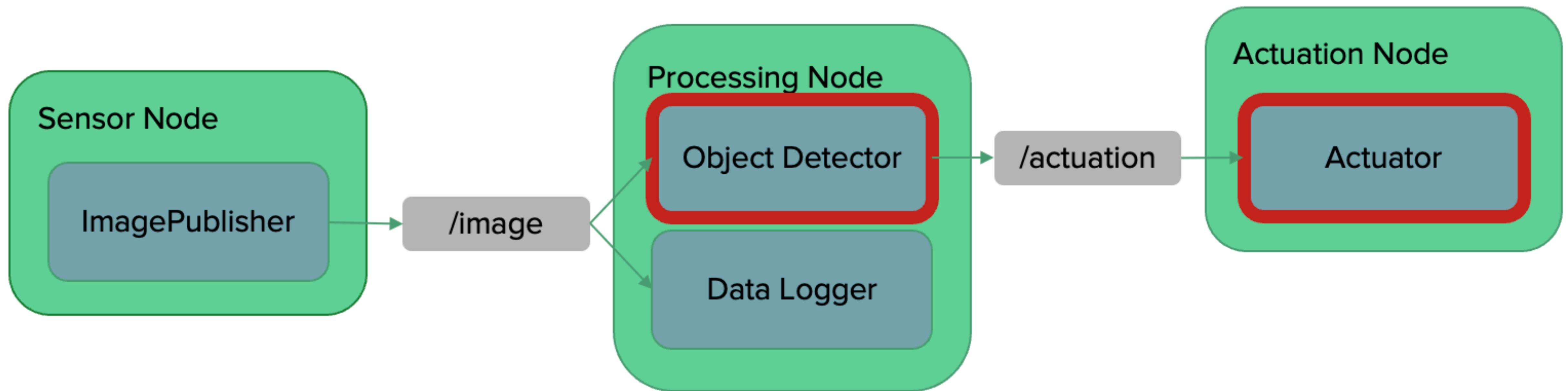
Publicador en thread real-time



Muy preciso

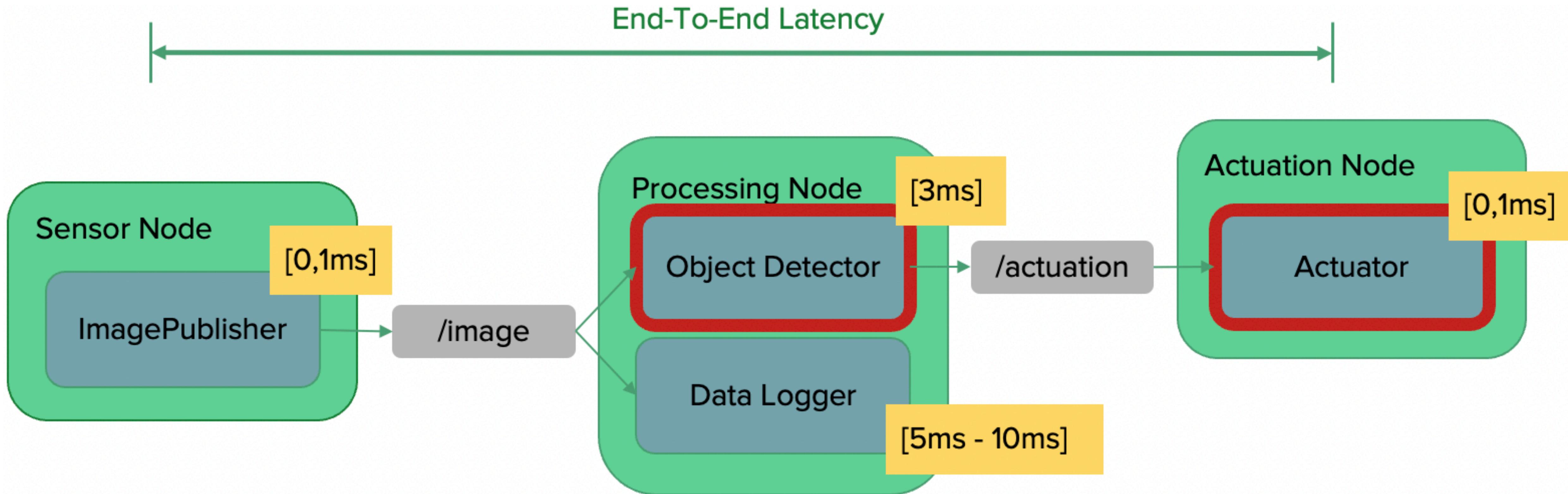
Ejercicio 4-2: Cadena de procesamiento en Tiempo Real

- Objetivo: ejecutar callbacks desde múltiples nodos en Tiempo Real.



Ejercicio 4-2: Cadena de procesamiento en Tiempo Real

- Objetivo: ejecutar callbacks desde múltiples nodos en Tiempo Real.



Ejercicio 4-2: Compilar y ejecutar

En una terminal:

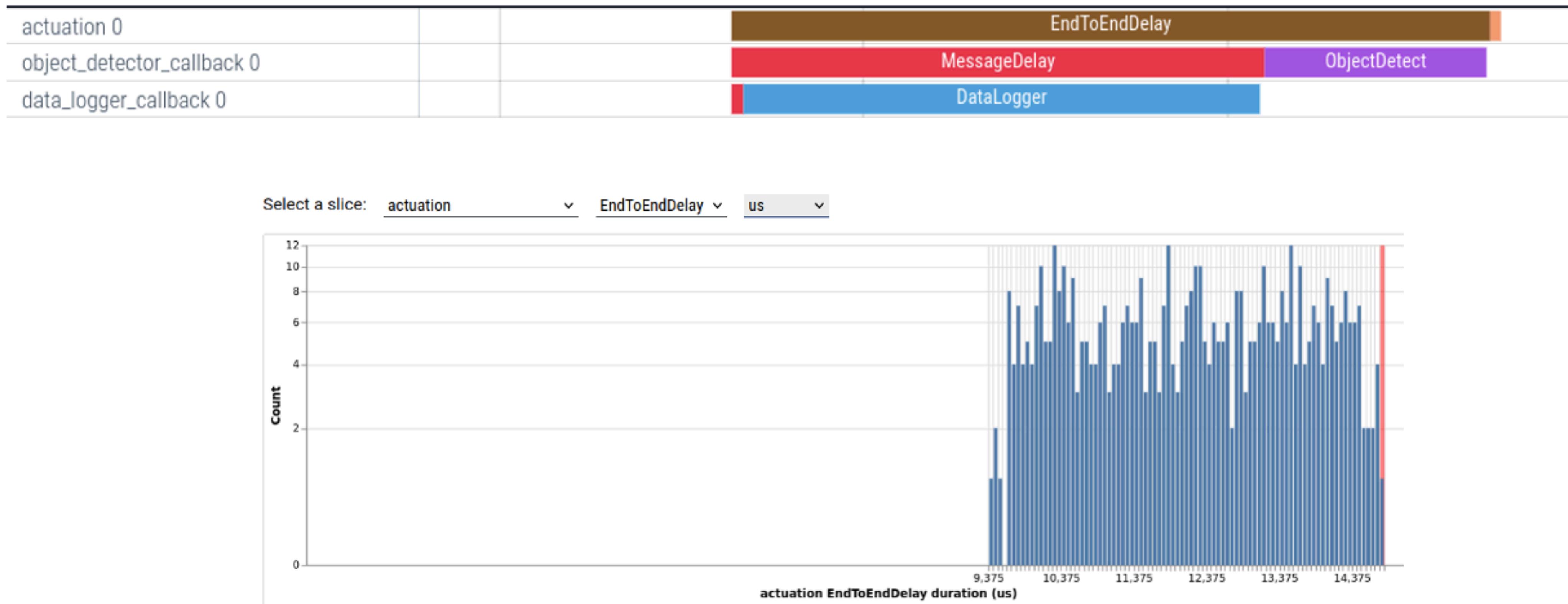
```
docker/shell  
cd exercise4-2  
colcon build  
.run.sh (for 15 sec)
```

Press Ctrl + C

Open perfetto (camera, period, us)



Resultado: retraso de extremo a extremo sin Tiempo Real



End-To-End Delay [9ms, 14ms]

Ejercicio 4-2: Recorrido por el código

```
11 class CameraProcessingNode : public rclcpp::Node {  
...  
15     rclcpp::CallbackGroup::SharedPtr      realtime_group_;  
16     rclcpp::CallbackGroup::SharedPtr      besteffort_group_;  
...  
21     public:  
27         rclcpp::CallbackGroup::SharedPtr get_realtime_cbg();  
28         rclcpp::CallbackGroup::SharedPtr get_besteffort_cbg();  
...  
}  
...
```

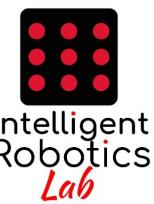
exercise4-1/src/camera_demo/include/application_nodes.h



Ejercicio 4-2: Recorrido por el código

```
rclcpp::CallbackGroup::SharedPtr CameraProcessingNode::get_realtime_cbg() {  
    if (!realtime_group_) {  
        realtime_group_ = this->create_callback_group(  
            rclcpp::CallbackGroupType::MutuallyExclusive  
        );  
    }  
    return realtime_group_;  
}  
  
rclcpp::CallbackGroup::SharedPtr CameraProcessingNode::get_besteffort_cbg() {  
    if (!besteffort_group_) {  
        besteffort_group_ = this->create_callback_group(  
            rclcpp::CallbackGroupType::MutuallyExclusive  
        );  
    }  
    return besteffort_group_;  
}
```

..exercise4-1/src/camera_demo/src/application_nodes.cc



Ejercicio 4-2: Recorrido por el código

```
int main(int argc, char** argv) {
    // initialization of ROS and DDS middleware
    rclcpp::init(argc, argv);

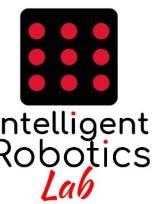
    StartImagePublisherNode();

    auto actuation_tracer = std::make_shared<caactus_rt::tracing::ThreadTracer>("actuation");
    auto actuation_node = std::make_shared<ActuationNode>(actuation_tracer);

    auto camera_processing_node = std::make_shared<CameraProcessingNode>(..., ...);
    auto data_logger_group = camera_processing_node->get_besteffort_cbg();
    auto object_detector_group = camera_processing_node->get_realtime_cbg();

    rclcpp::executors::SingleThreadedExecutor real_time_executor;
    rclcpp::executors::SingleThreadedExecutor best_effort_executor;
```

exercise4-1/src/camera_demo/src/main.cc



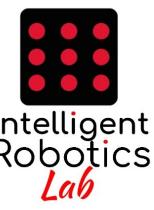
Ejercicio 4-2: Recorrido por el código

```
best_effort_executor.add_callback_group(data_logger_group, ...);  
real_time_executor.add_callback_group(object_detector_group, ...);  
real_time_executor.add_node(actuation_node);
```

```
// Launch real-time Executor in a thread  
std::thread real_time_thread([&real_time_executor]() {  
    sched_param sch;  
    sch.sched_priority = 60;  
    if (sched_setscheduler(0, SCHED_FIFO, &sch) == -1) {  
        perror("sched_setscheduler failed");  
        exit(-1);  
    }  
    real_time_executor.spin();  
});
```

```
best_effort_executor.spin();  
rclcpp::shutdown();
```

exercise4-1/src/camera_demo/src/main.cc



Ejercicio 4-2: Recorrido por el código

```
best_effort_executor.add_callback_group(data_logger_group, ...);
real_time_executor.add_callback_group(object_detector_group, ...);
real_time_executor.add_node(actuation_node);
```

```
// Launch real-time Executor in a thread
std::thread real_time_thread([&real_time_executor]() {
    sched_param sch;
    sch.sched_priority = 60;
    if (sched_setscheduler(0, SCHED_FIFO, &sch) == -1) {
        perror("sched_setscheduler failed");
        exit(-1);
    }
    real_time_executor.spin();
});

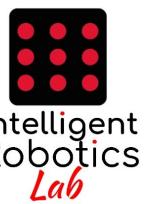
best_effort_executor.spin();
rclcpp::shutdown();
```

exercise4-1/src/camera_demo/src/main.cc

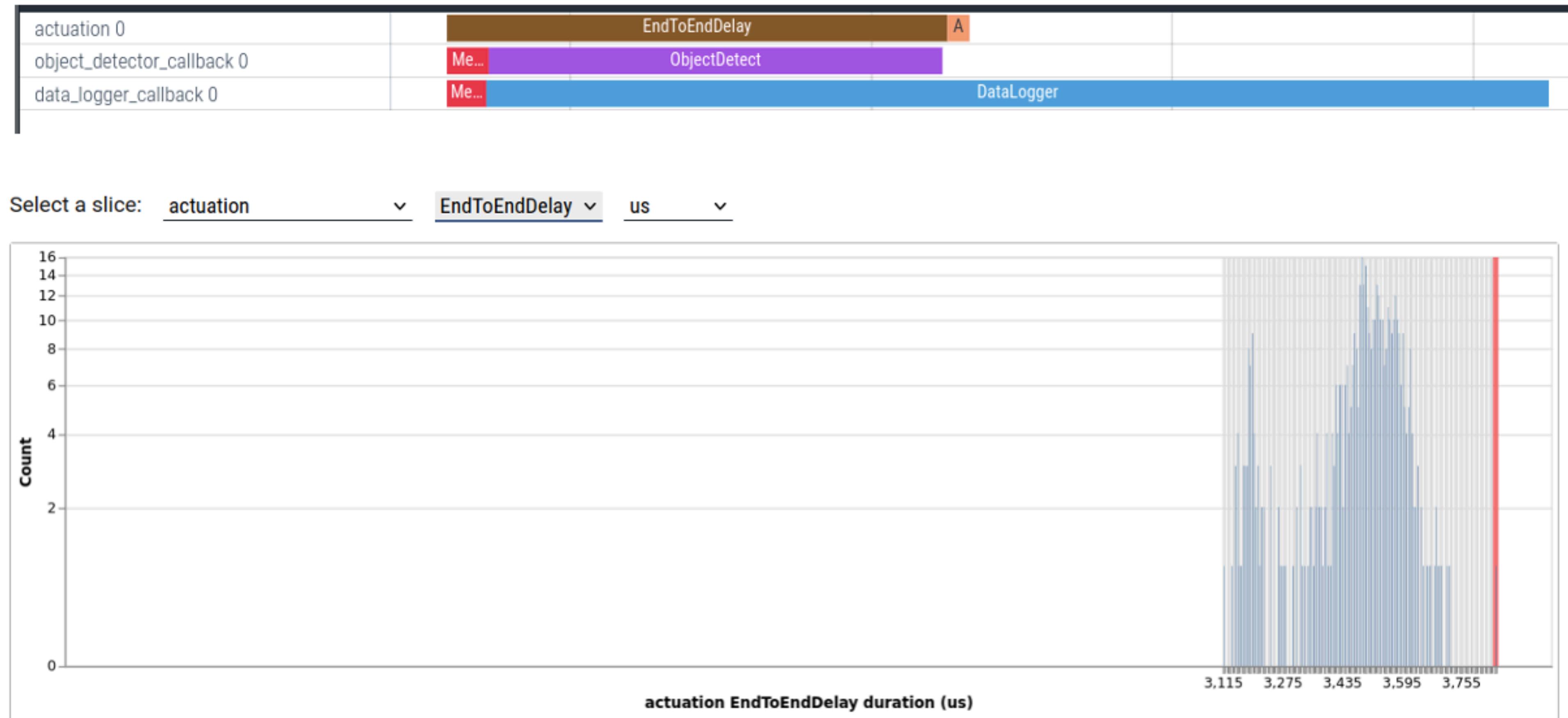
Terminal 1:

```
docker/shell
cd exercise4-2
cp ./solutions/* ./src/camera_demo/src
colcon build
./run.sh (for 15 sec)

Press Ctrl + C
Open perfetto (camera, period, us)
```

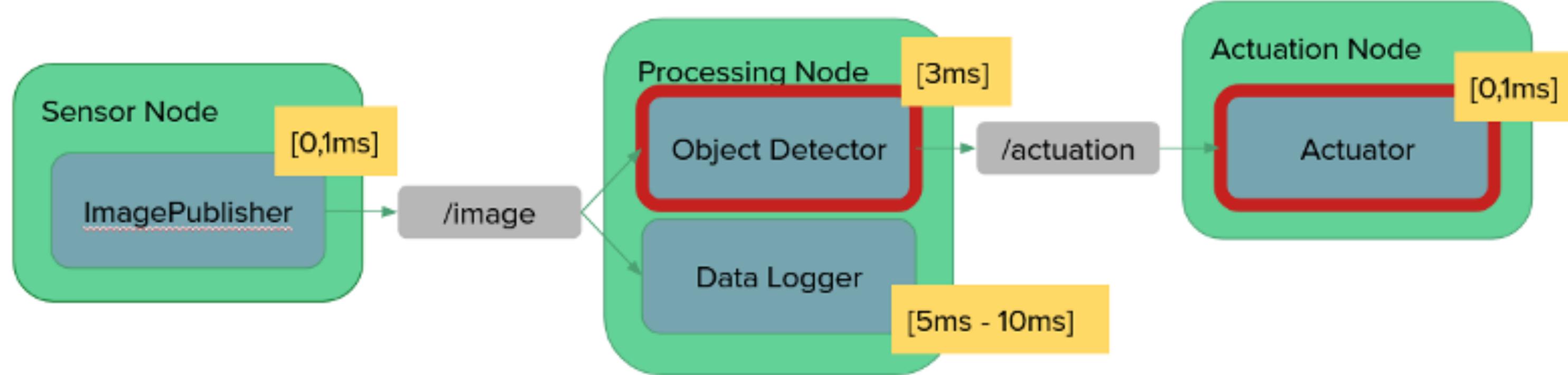


Ejercicio 4-2: Resultado: EndToEndDelay con configuración en Tiempo Real

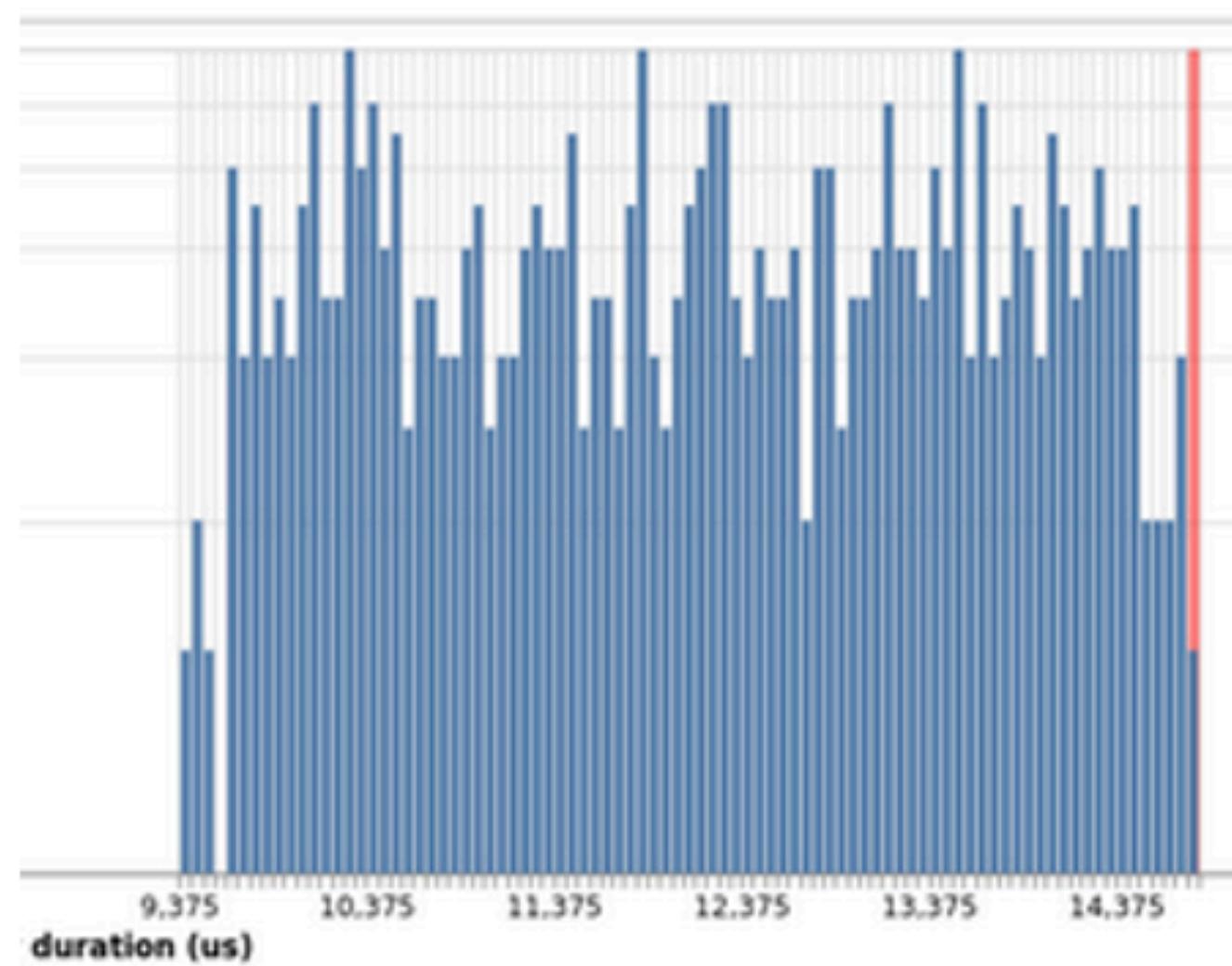


EndToEnd Delay [3.1ms, 3.8ms]

Ejercicio 4-2: Resumen

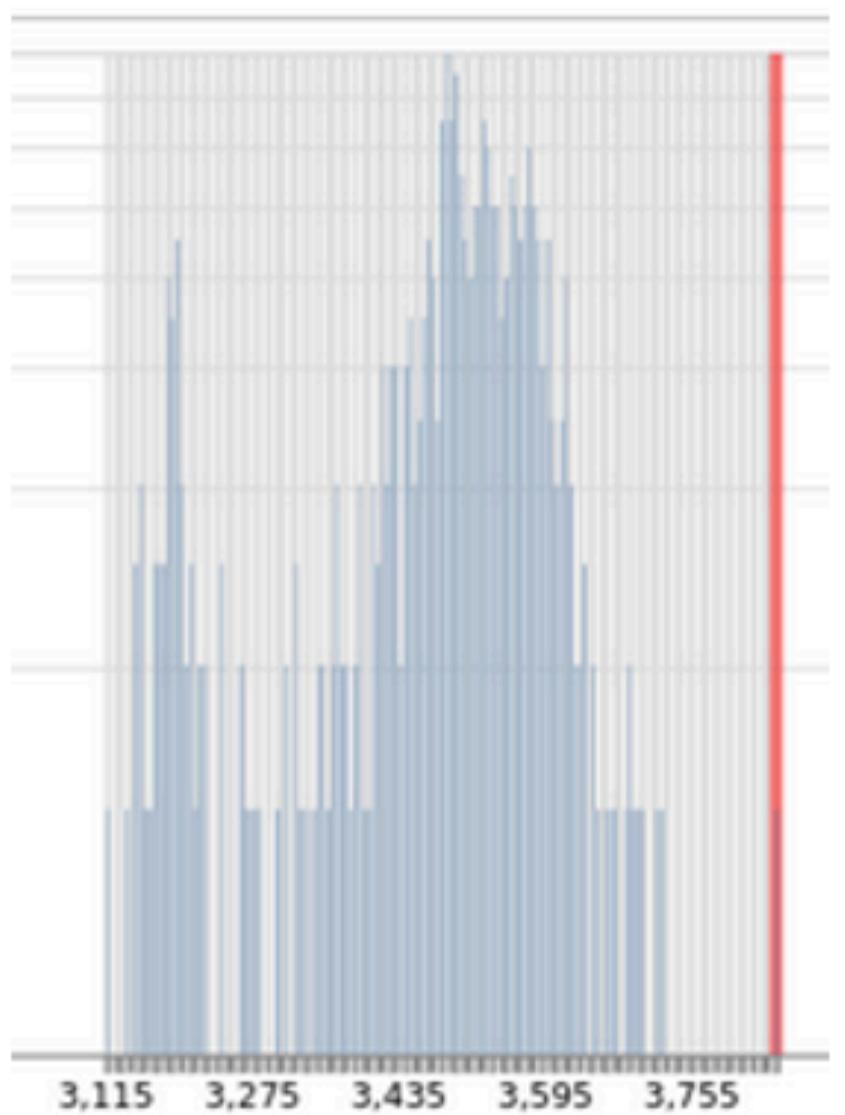


Latencia End-To-End con ROS 2 predeterminado



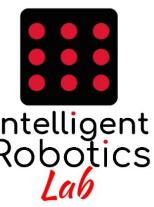
Alta latencia

Latencia End-To-End en un thread real-time



Baja latencia

Resumen del Workshop



¿De qué hemos hablado?

- ¿Qué es “Tiempo Real”?
- Hardware, SO, latencia de aplicaciones
- Planificadores en Tiempo Real y **PREEMPT_RT**
- Arquitectura de software y técnicas de programación en Tiempo Real
- Bloqueos e inversión de prioridades
- Gestión de ejecución de ROS 2
- Impacto en el comportamiento en Tiempo Real
- Descripción general de los diferentes ejecutores
- Ejecutor experimental en Tiempo Real
- Mejores prácticas para el Tiempo Real en ROS 2: priorización de subprocessos, grupos de callbacks
- Mensajes prestados, middleware de copia cero
- Herramientas y puntos de referencia



Nos llevamos...

- Para garantizar un comportamiento en Tiempo Real es necesario garantizar que cada capa de software sea en Tiempo Real.
- La programación en Tiempo Real requiere técnicas especializadas en lo que respecta a la gestión de memoria, la programación concurrente y logging.
- Los threads en Tiempo Real pueden cooperar con threads ROS que no son en Tiempo Real
- La gestión de ejecución de ROS 2 tiene una semántica compleja y no es capaz de ejecutarse en Tiempo Real.
- Se propone un ejecutor experimental en Tiempo Real con una API simple.
- El Tiempo Real en ROS 2 aún se puede lograr mediante el uso de funciones avanzadas, como threads en Tiempo Real, callback groups y mecanismos de memoria de copia cero.



2
S
O
R
• • •

Workshop de Tiempo Real

Gracias por vuestra atención!!

Prof. Dr. Francisco Martín Rico
20-09-2024

francisco.rico@urjc.es
@fmrico



at

