Real-time programming with ROS 2

Workshop for ROScon 2023
Oct 18 2023

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

Disclaimer

Opinions here are of our own, not representative of employer

Agenda

Morning

- 8:00 - 8:45: Latency, hardware, and OS
- 8:45 - 9:15: Hands-on exercise 1
- 9:15 - 10:15: RT C++ programming
- 10:15 - 10:30: Hands-on exercise 2
- 10:30 - 11:00: Break
- 11:00 - 11:30: Hands-on exercise 2
- 11:30 - 12:00: Panel discussion

Afternoon

- 13:00 - 14:00: ROS 2 execution management
- 14:00 - 14:45: Hands-on exercise 3
- 14:45 - 15:30: Real-time with mainline ROS 2
- 15:30 - 16:00: Break
- 16:00 - 16:45: Hands-on exercise 4
- 16:45 - 17:00: Summary

Introduction and definitions

What is “real-time” programming?

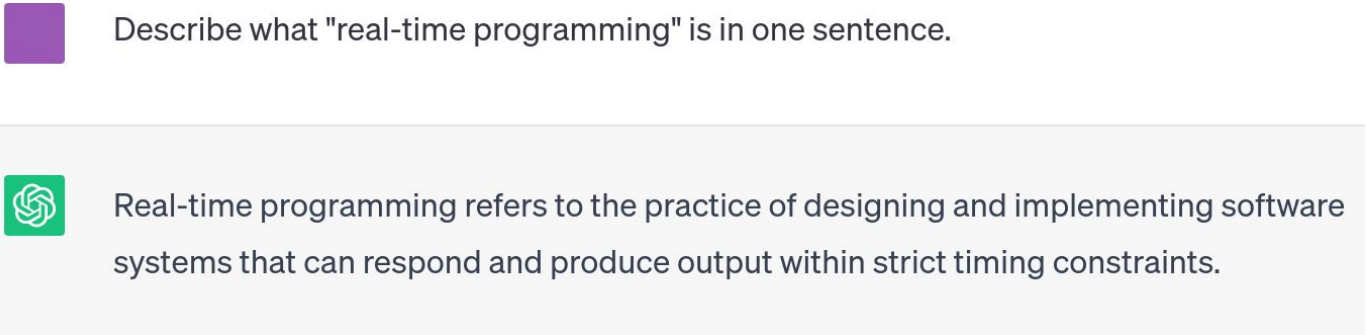
- Real-time: ambiguous term used in multiple domains
  - Real-time delivery tracking
  - Real-time clock
  - Real-time visualizations
  - Real-time operating systems
  - Real-time applications
- Let’s define it for this workshop

What is “real-time” programming?

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.

What is “real-time” programming?



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

What is “real-time” programming?

- Real-time criteria 1: **respond within deadlines**
- ChatGPT neglected, but people often mention “soft”, “hard”, and “firm” real-time.
  - Also ambiguous. So we must define it.

What is “real-time” programming?

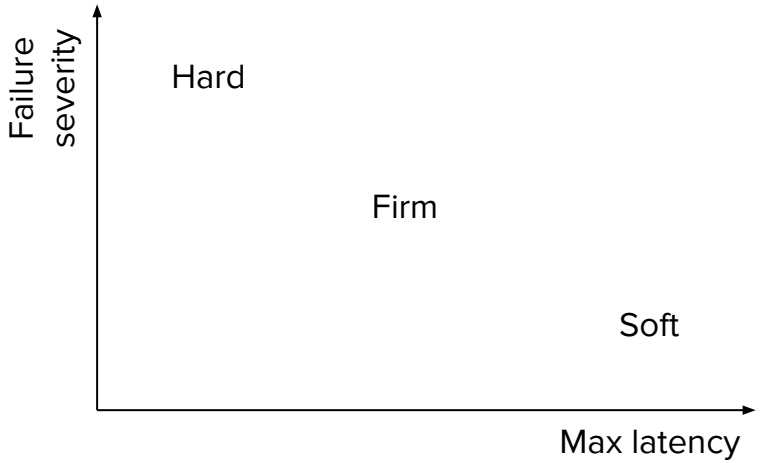
Differentiate between soft and hard real-time in one sentence.

Soft real-time systems have timing constraints that can be occasionally violated without catastrophic consequences, while hard real-time systems have strict timing requirements that must always be met to ensure system functionality.

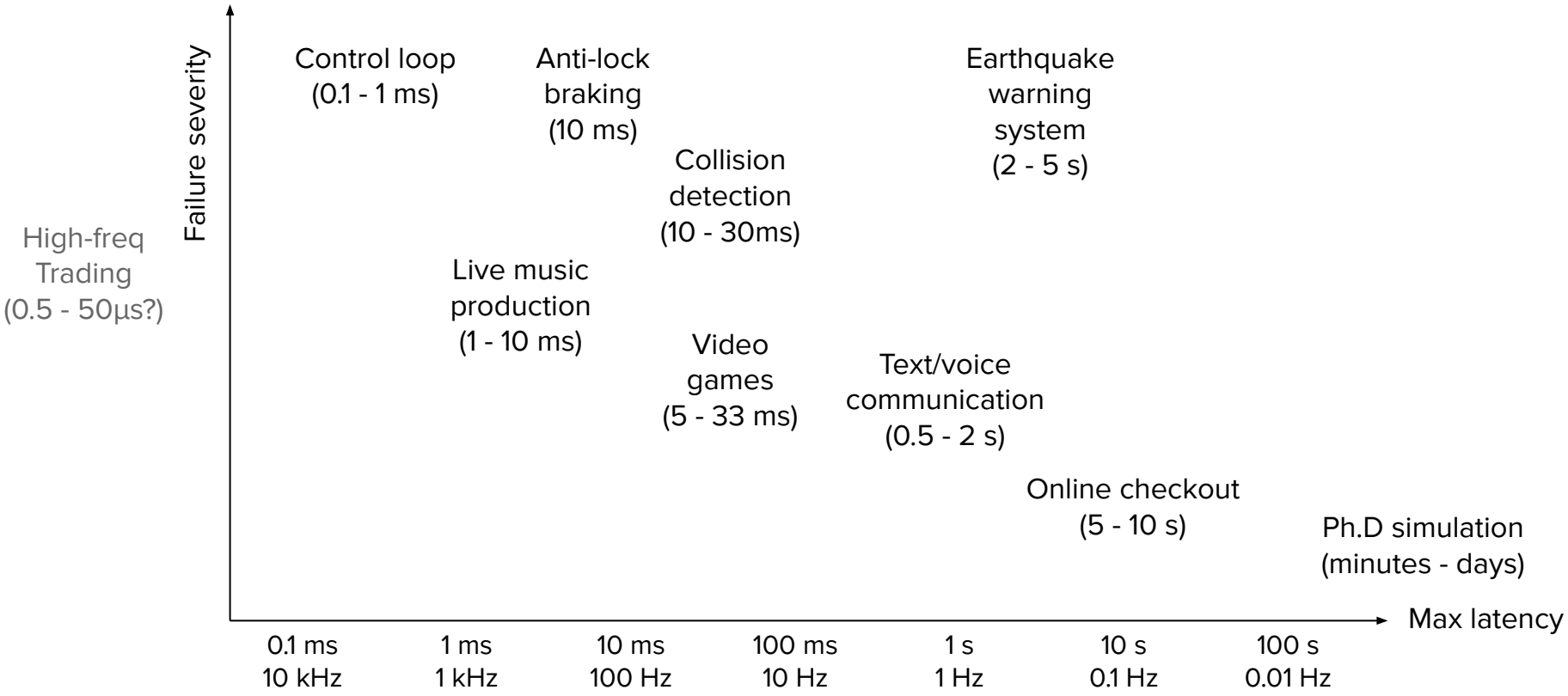
- “Hard” real-time ➜ has catastrophic consequences
  - Some define this as systems that are mathematically proven
- “Soft” real-time ➜ no catastrophic consequences
- “Firm” real-time ➜ somewhere between those two
- These forms a “spectrum” of real-time software

Defining “real-time”

- Real-time criteria 1: **respond within deadlines**
  - Bounded maximum *latency*
- Real-time criteria 2: **consequences of missing deadlines**
- Real-time as a “spectrum”



Examples of real-time systems



Real-time and robotics

- High-frequency control loops (~1000Hz)
  - Failure: unstable controller can cause unsafe situations
- Object detection and collision avoidance (~100Hz)
  - Failure: collision
- Not necessarily “hard” real-time
  - Other safety systems (watchdog, limits, etc.) to ensure safety
  - Counterpoint: some domains requires certification

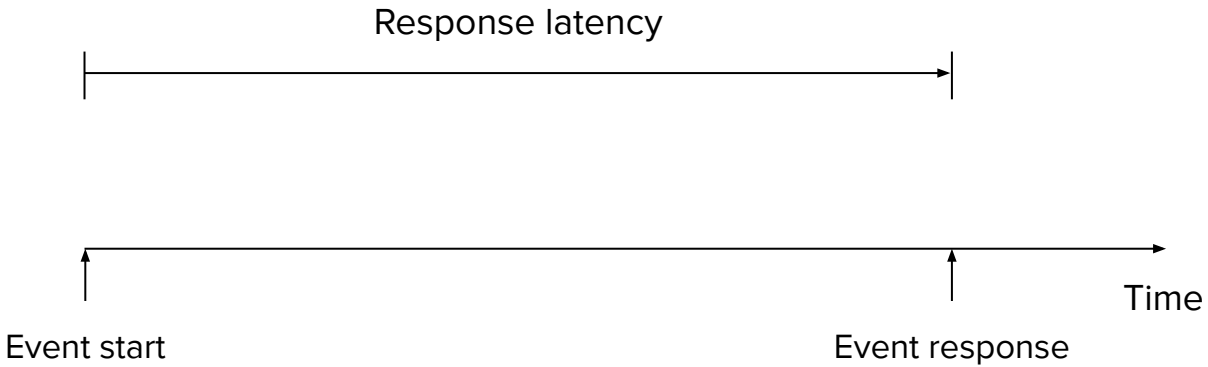
Latency, hardware, and OS

Real-time systems, not real-time software

Sources of latency

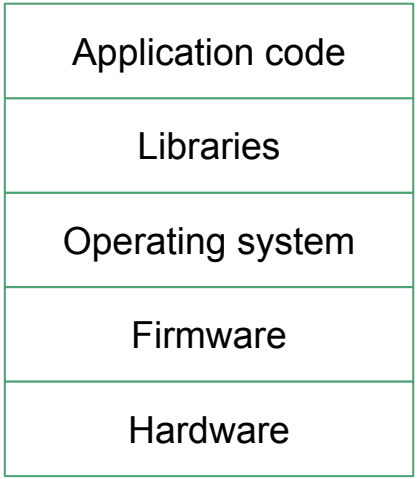
- Response latency:
  - T1 = object appears
  - T2 = robot stops
  - Response latency = T2 - T1

Application code
Libraries
Operating system
Firmware
Hardware



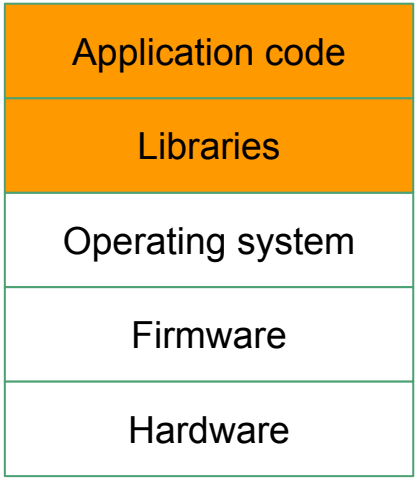
Sources of latency

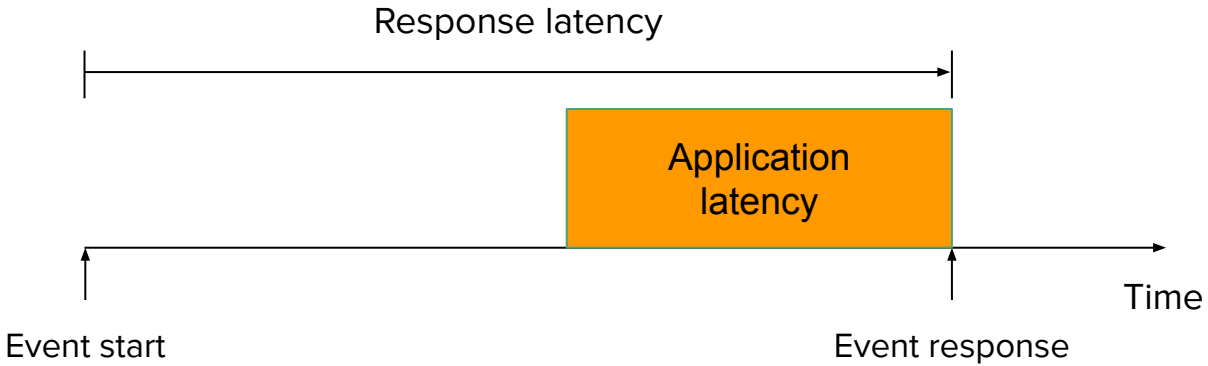
- Modern software stack is very deep
- Each layer can induce latency
- Need to ensure bounded max latency of all layers
  - Difficult task: most software designed for average latency, not max latency



Sources of latency

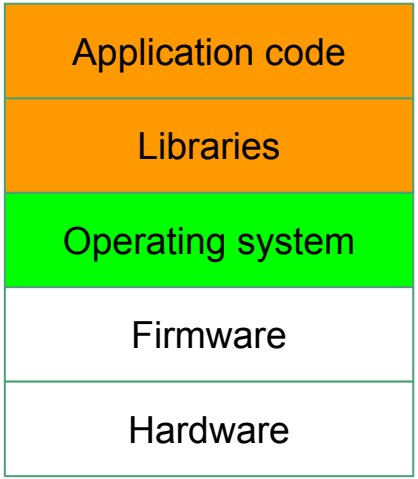
- Application latency
  - Libraries + application code
  - This will be covered more in-depth later

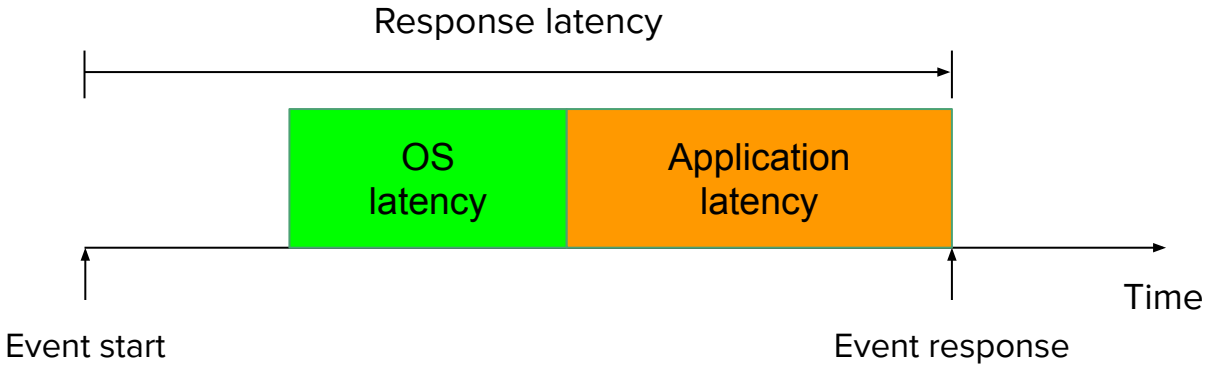




Sources of latency

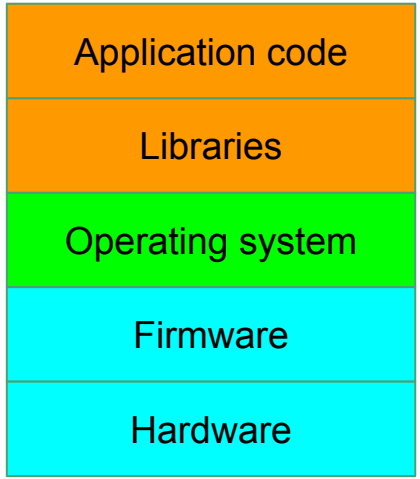
- Application latency
  - Libraries + application code
- OS latency
  - Mostly due to operating system scheduling

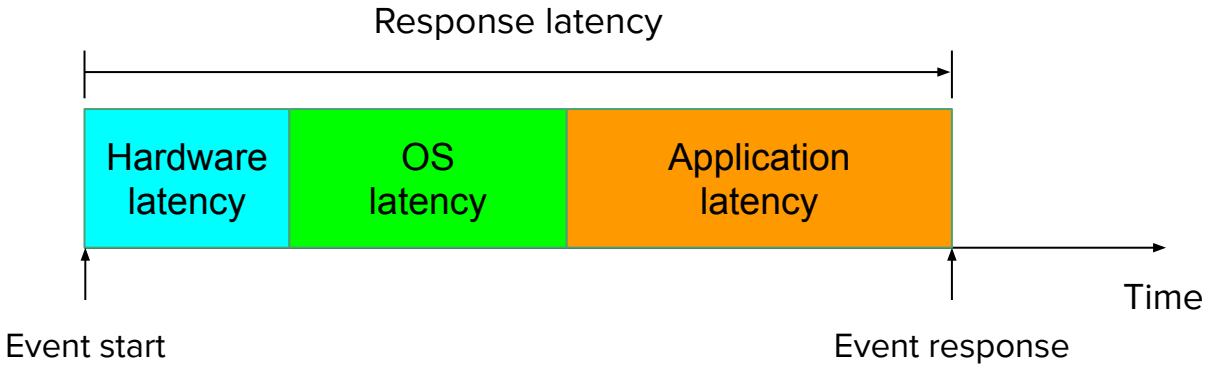




Sources of latency

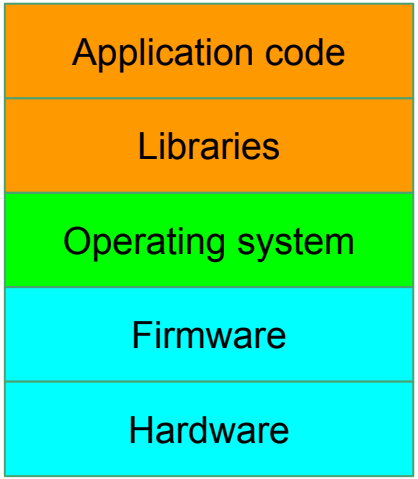
- Application latency
  - Libraries + application code
- OS latency
  - Mostly due to operating system scheduling
- Hardware latency
  - Hardware + firmware

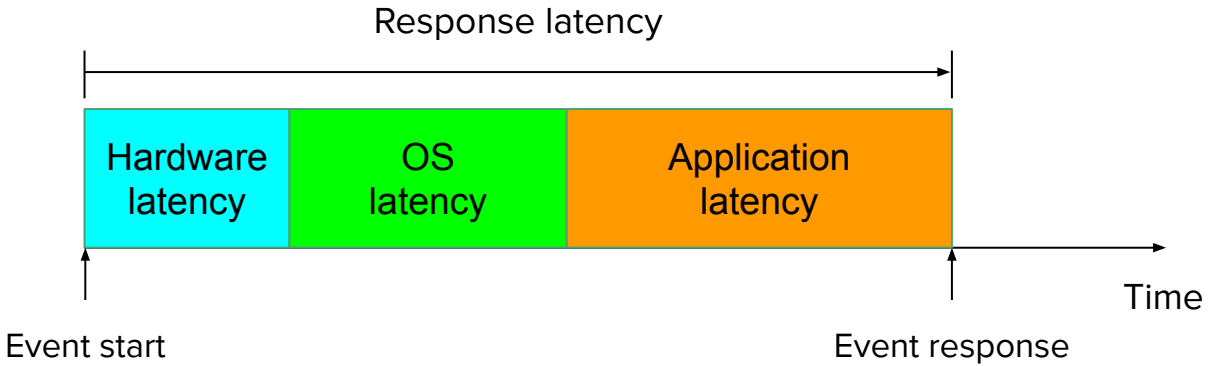




Sources of latency

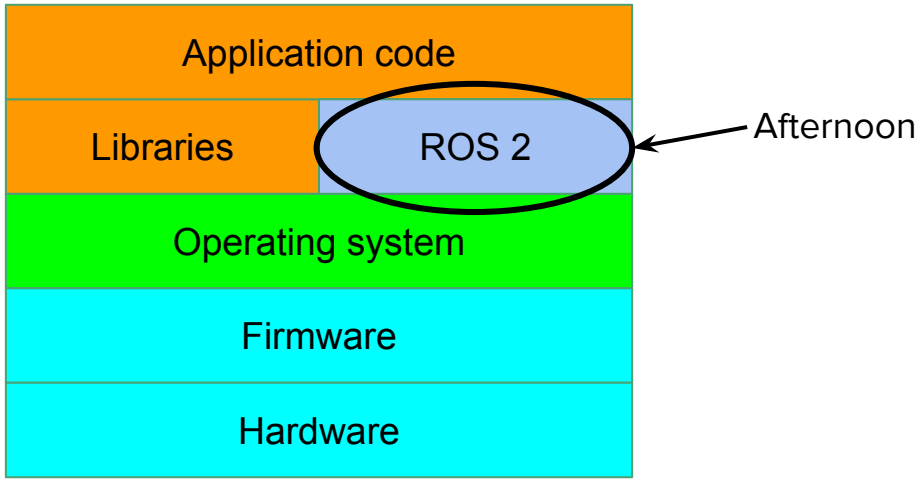
- Worst-case latency of each source can be summed together to infer to worst-case response latency
  - Assume latency caused by different components are independent





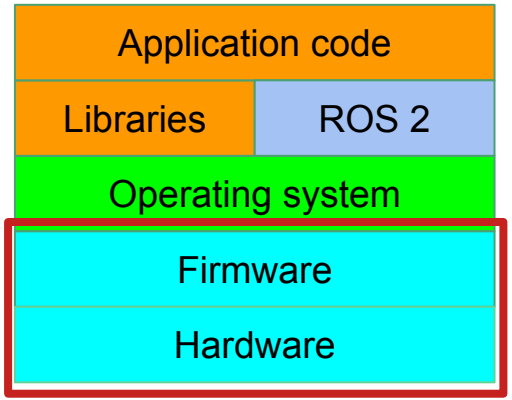
How does ROS fit in?

- ROS is considered as the “application”, specifically the library layer
- More discussions in the afternoon
- Everything else in the morning



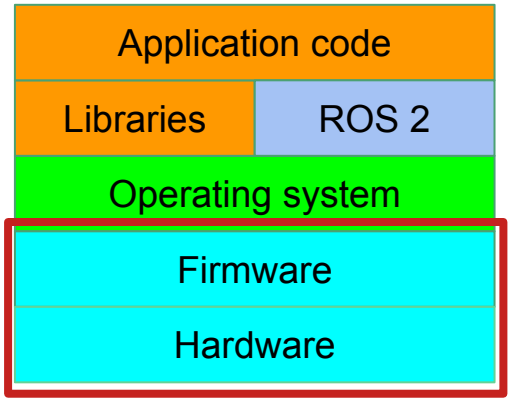
Hardware latency

- Modern hardware is very complex



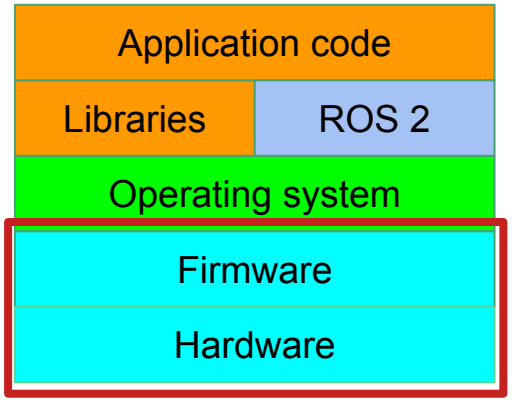
Hardware latency

- Modern hardware is very complex
- OS does not fully control the hardware
  - System management interrupts (SMI)
  - Simultaneous multithreading



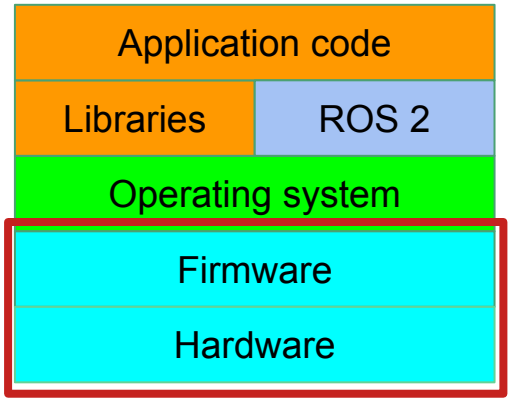
Hardware latency

- Modern hardware is very complex
- OS does not fully control the hardware
  - System management interrupts (SMI)
  - Simultaneous multithreading
- Non-uniform hardware performance over time
  - Dynamic frequency scaling based on power and temperature
  - Cache miss, branch prediction failure, etc



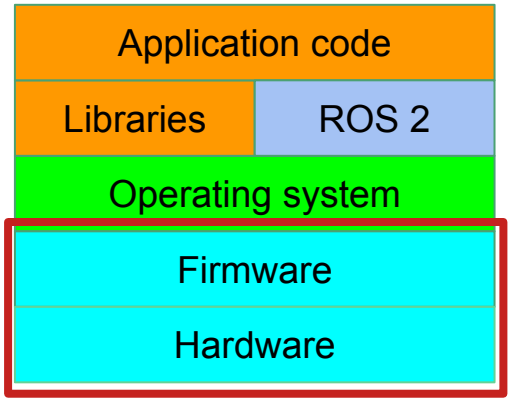
Hardware latency

- Modern hardware is very complex
- OS does not fully control the hardware
  - System management interrupts (SMI)
  - Simultaneous multithreading
- Non-uniform hardware performance over time
  - Dynamic frequency scaling based on power and temperature
  - Cache miss, branch prediction failure, etc
- Latency due to physics
  - Hard disk access latency
  - Memory latency
  - Serial communication latency



Dealing with hardware latency

- Usually the biggest issue is SMI-induced latency
  - Usually only controllable by the OEM
  - Can be tested with **cyclictest** and **hwlatdetect**
- Turn off dynamic frequency scaling and simultaneous multithreading
- Test IO and peripheral performance



Case study on hardware latency

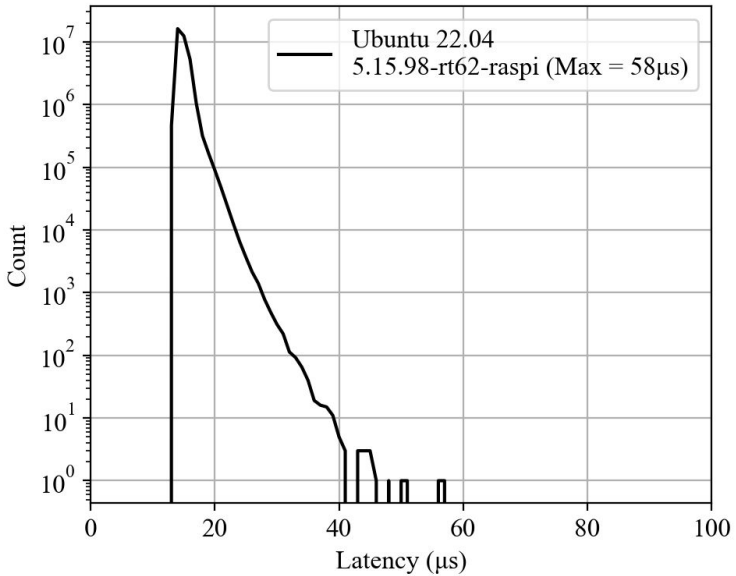
	SMI-induced latency
Hardware	Off-the-shelf workstation
Symptom	Constant 0.4 - 1.2ms stutters
Root cause	System management interrupts (likely)
Diagnosed with	cyclictest, hwlatdetect
Solution	A new computer

Case study on hardware latency

	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

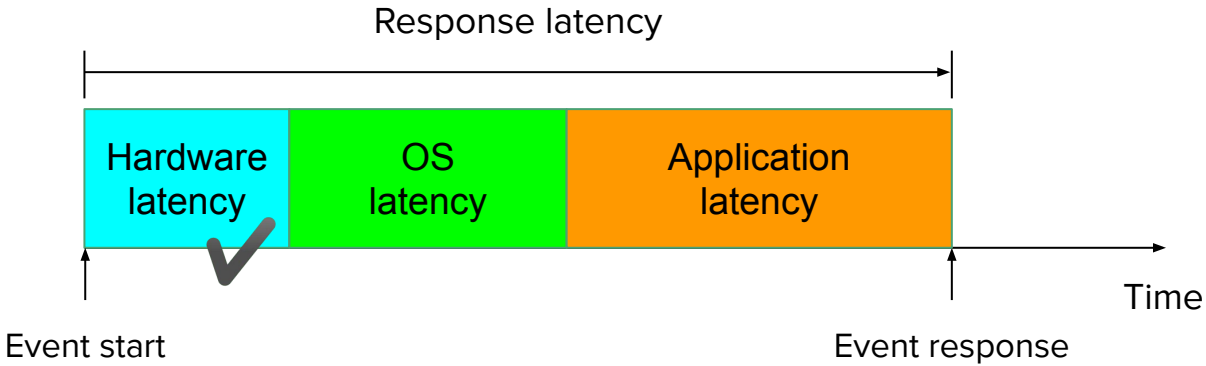
Raspberry Pi 4 as a real-time platform

- Raspberry Pi 4 does not appear to have excessive hardware latency
- max(OS + hardware latency) ≃ 100µs
  - Implies max(hardware latency) < 100µs



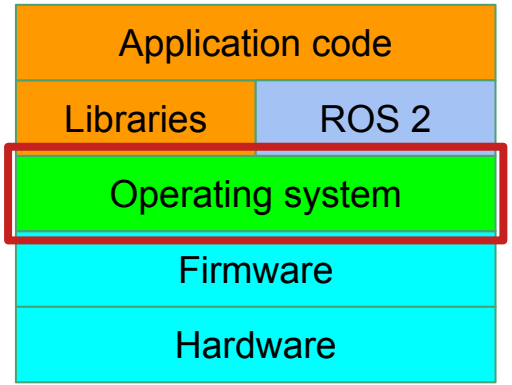
Hardware latency: a summary

- Hardware is complex and can induce latency
- Buy the right hardware
- Test rigorously
- Trust but verify!



Operating system latency

- Modern operating systems are very complex
- Primarily: latency due to scheduling
  - Different OS have different latency profiles
- Kernel/driver code may play a role depending on the OS



What is scheduling?

- A CPU core runs one instruction at a time*

What is scheduling?

- A CPU core runs one instruction at a time*
- Main job of OS:
  - Run multiple tasks “simultaneously” on a single CPU

What is scheduling?

- A CPU core runs one instruction at a time*
- Main job of OS:
  - Run multiple tasks “simultaneously” on a single CPU
- Example:
  - User input while running compute-intensive tasks
  - Pressing E on the keyboard while encoding video

What is scheduling?

- A CPU core runs one instruction at a time*
- Main job of OS:
  - Run multiple tasks “simultaneously” on a single CPU
- Example:
  - User input while running compute-intensive tasks
  - Pressing E on the keyboard while encoding video
- CPU scheduler is responsible for this
- Known as task switching

Scheduler latency (at a high level)

- Task switching (context switching) is expensive
  - Save and restore register values to and from RAM

Scheduler latency (at a high level)

- Task switching (context switching) is expensive
  - Save and restore register values to and from RAM
- Interrupt processing
  - Some interrupts must run or the system will crash
  - This can take time

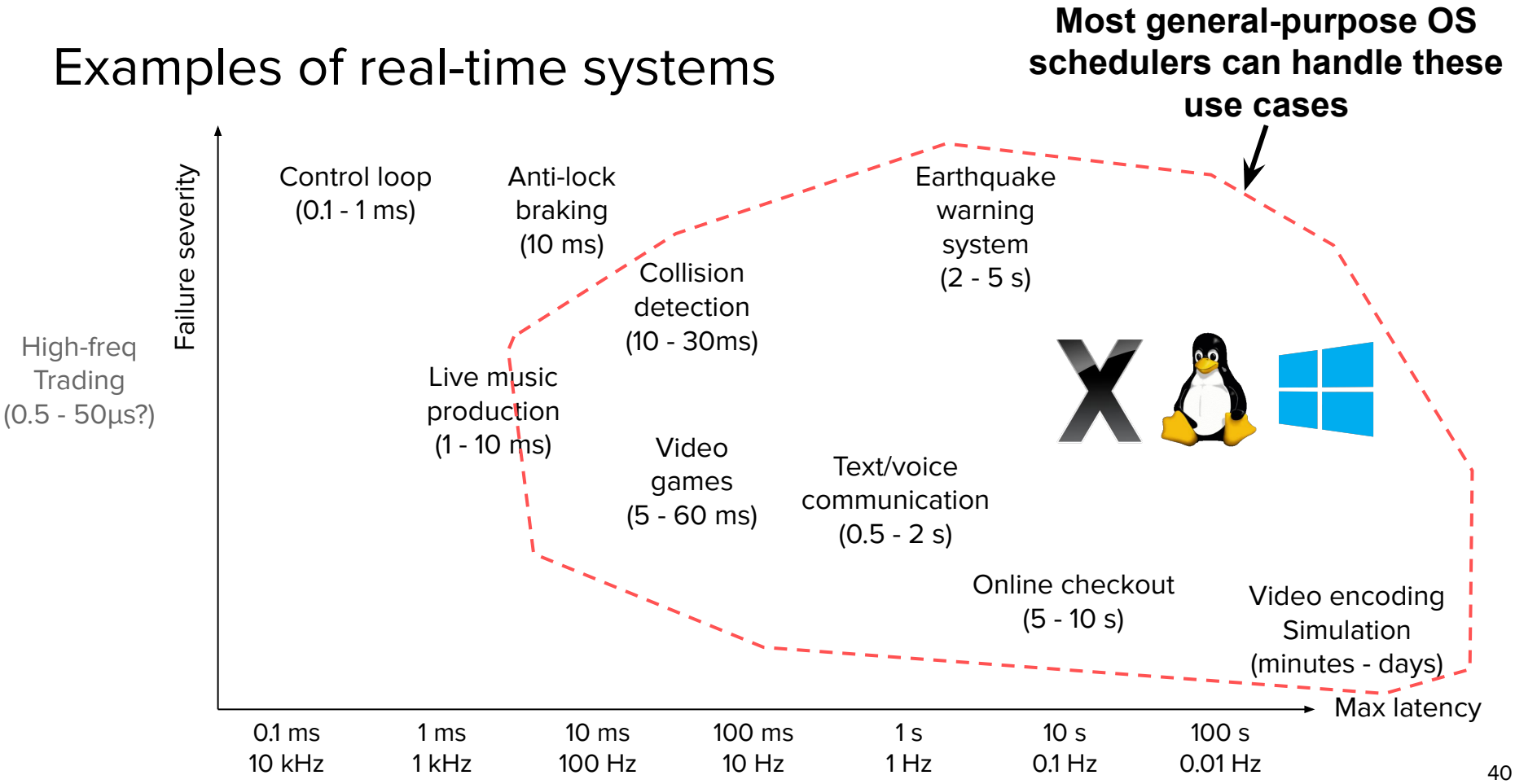
Scheduler latency (at a high level)

- Task switching (context switching) is expensive
  - Save and restore register values to and from RAM
- Interrupt processing
  - Some interrupts must run or the system will crash
  - This can take time
- Some kernel/driver code cannot be interrupted
  - Dependent on the OS and drivers

Scheduler latency (at a high level)

- Task switching (context switching) is expensive
  - Save and restore register values to and from RAM
- Interrupt processing
  - Some interrupts must run or the system will crash
  - This can take time
- Some kernel/driver code cannot be interrupted
  - Dependent on the OS and drivers
- **max(response latency) ≥ max(scheduler latency)**

Examples of real-time systems



Dealing with scheduling latency with different OSes

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”?

Dealing with scheduling latency with different OSes

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”?

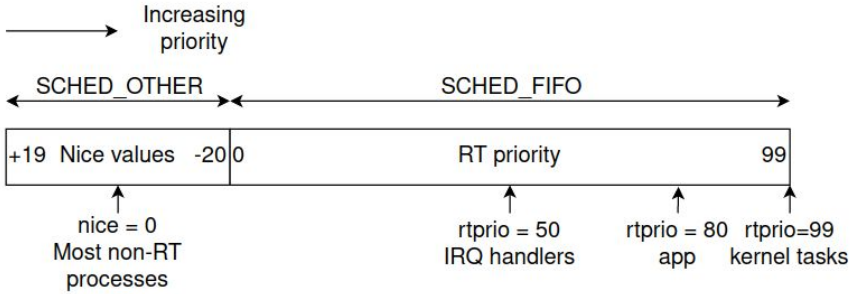
Dealing with OS latency with Linux: scheduler

- Scheduling is an NP-complete problem
- Linux default scheduler: optimizes for most cases, but not real-time
- Linux includes multiple schedulers
- Selectable at run time for each thread

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`	👍

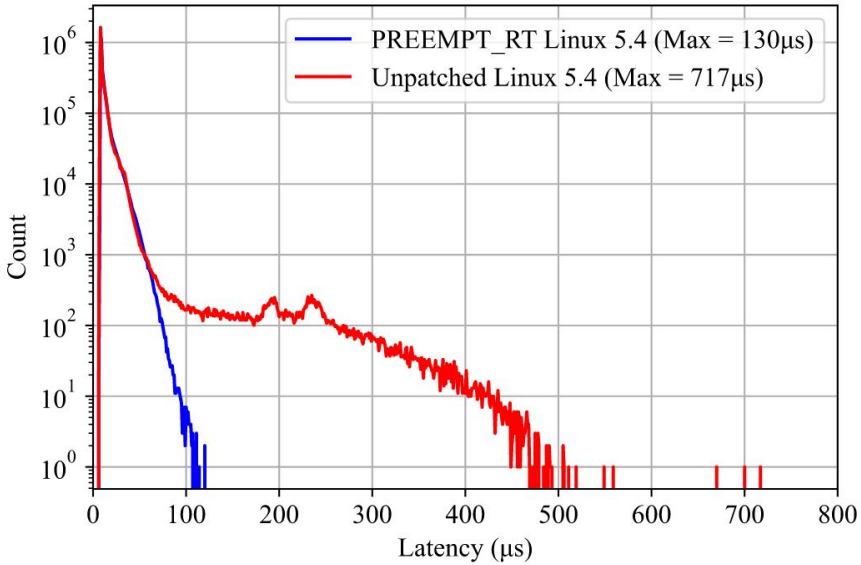
Dealing with OS latency with Linux: scheduler

- Real-time scheduler must be selected and configured
- Requesting for FIFO scheduler:
  - `pthread_attr_setschedpolicy(&attr, SCHED_FIFO);`
- Setting priority:
  - `struct sched_param param; param.sched_priority = 80;`
- All SCHED_FIFO tasks have higher priority than SCHED_OTHER tasks
  - A priority level of 80 is a good first choice



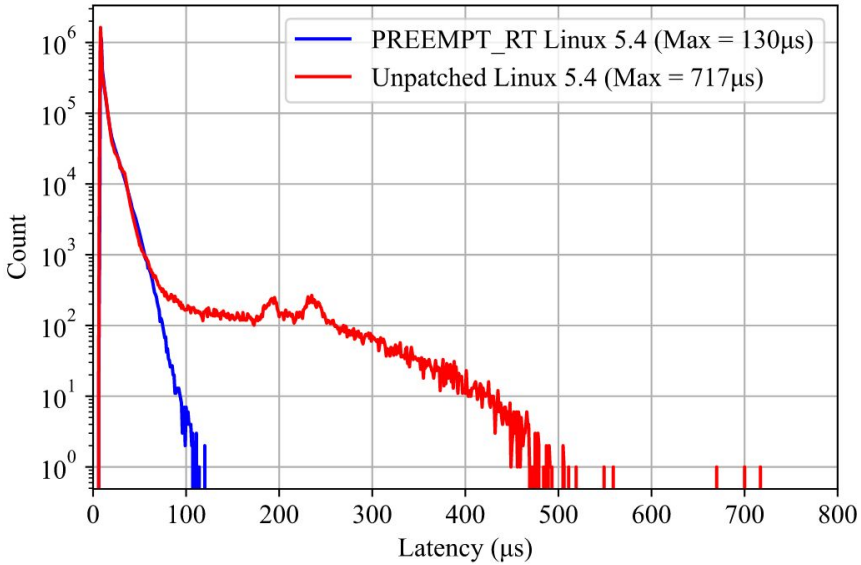
Dealing with OS latency with Linux: PREEMPT_RT

- Some kernel code is not interruptible by default
  - Introduces latency
- PREEMPT_RT addresses most of these
  - Reduces latency
  - Details out of scope for workshop
- PREEMPT_RT is a set of patches for Linux
  - Aiming to merge into mainline eventually
- max(OS latency) ≃ 100 µs
  - Usable for 1-2 kHz real-time systems
- Not proved/certified



Dealing with OS latency with Linux: PREEMPT_RT

- SCHED_FIFO exists in mainline kernel as well
  - Worst-case latency is higher than PREEMPT_RT
- Worst-case latency with SCHED_OTHER even higher
- Getting better as more PREEMPT_RT code is merged to mainline
- Usable for today's exercises



When and when not to use Linux and PREEMPT_RT

- Use Linux + PREEMPT_RT when:
  - ≤ 1000 - 2000 Hz
  - Do not require certification
  - Can tolerate rare/hypothetical deadline miss
- Do not use Linux + PREEMPT_RT when:
  - ≥ 10000 Hz
  - Requires certification for regulatory reasons

Who uses PREEMPT_RT

- National Instruments Linux Real-Time
- SpaceX flight software
- CNC machines
- Autonomous vehicles, robotic arms
- Many others, mostly unpublicized

Dealing with OS latency with Linux: Tuning

- Default kernel/OS configurations not necessarily good for real-time
  - Disable RT throttling
  - Disable unnecessary or latency-harmful services (multipathd on Ubuntu server...)
  - Tune security.conf for privileges
  - isolcpus, irqaffinity, rcu_nocbs
  - nohz, nohzfull

Dealing with OS latency: summary

- Scheduler latency:
  - Request for RT scheduler and set priority
    - `SCHED_FIFO`
    - `sched_priority = 80`
- Linux with PREEMPT_RT
  - Good for 1-2 kHz workload
  - Need patch kernel
  - Requires tuning various kernel and OS configurations

Validation of hardware and OS latency

- How can you determine the maximum latency?
  - Mathematically? Empirically?
  - On kernel, firmware, CPU internal implementations?
  - Where does the validation ends?
- Empirically on Linux
  - hwlatdetect: Measures SMI-induced latency
  - cyclictest: Measures max(hardware + OS latency)
- Run CPU stress test while measuring OS latency
  - `stress-ng -c $(nproc)`
  - We will run this in some of the exercises
- Run the test for hours/days/weeks until desired level of confidence

Summary

- Real-time systems are defined by their max latency and failure severity

Summary

- Real-time systems are defined by their max latency and failure severity
- Latency can be introduced by the hardware, OS, and application

Summary

- Real-time systems are defined by their max latency and failure severity
- Latency can be introduced by the hardware, OS, and application
- Hardware and OS latency can be characterized and minimized (after tuning)

Summary

- Real-time systems are defined by their max latency and failure severity
- Latency can be introduced by the hardware, OS, and application
- Hardware and OS latency can be characterized and minimized (after tuning)
- Linux + PREEMPT_RT is an attractive platform to build real-time applications
  - Up to 1-2 kHz applications without formal verification requirements
  - Many companies deploy PREEMPT_RT into production

Summary

- Real-time systems are defined by their max latency and failure severity
- Latency can be introduced by the hardware, OS, and application
- Hardware and OS latency can be characterized and minimized (after tuning)
- Linux + PREEMPT_RT is an attractive platform to build real-time applications
  - Up to 1-2 kHz applications without formal verification requirements
  - Many companies deploy PREEMPT_RT into production
- Trust, but verify with long duration testing

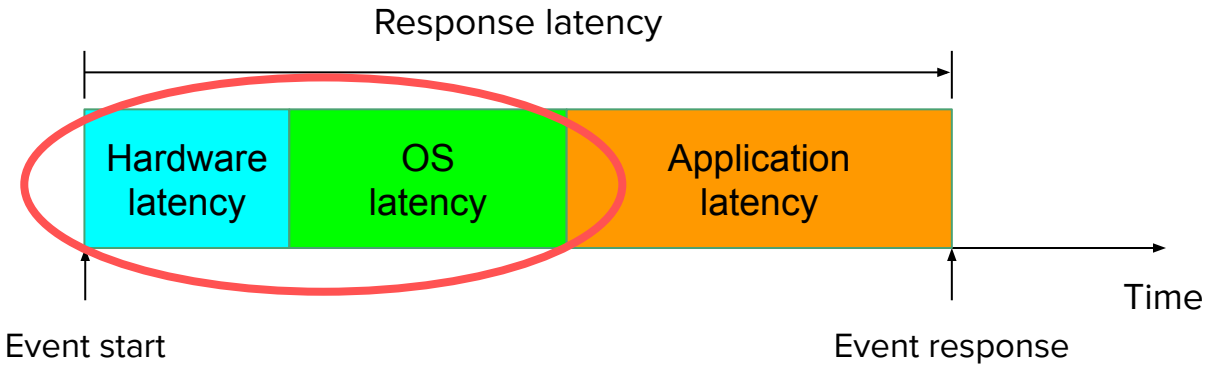
Additional references (for later)

- 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

Hands-on exercise 1:
measuring hardware + OS latency

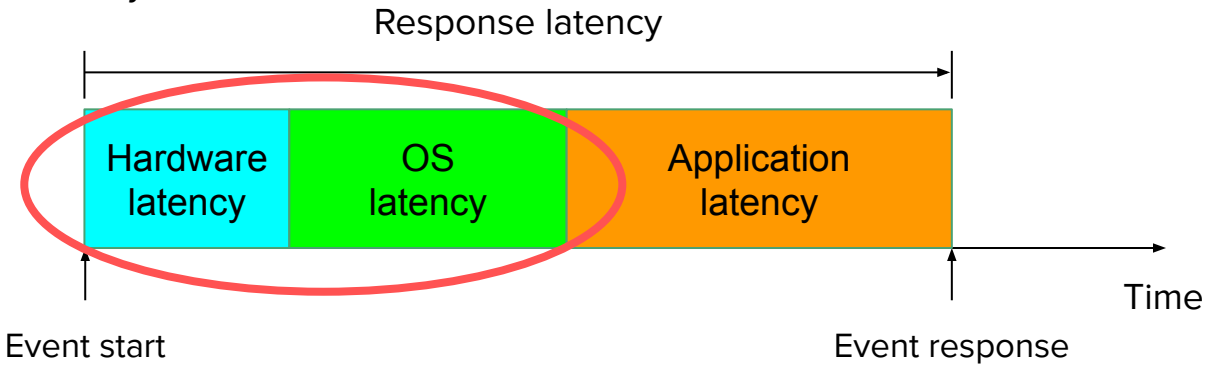
Exercise 1: measuring wakeup latency

- Objective: measuring wakeup latency with and without real-time scheduling
  - Wakeup latency = Hardware latency + OS latency
  - OS latency cannot be measured independently from hardware



Exercise 1: measuring wakeup latency

- Objective: measuring wakeup latency with and without real-time scheduling
  - Wakeup latency = Hardware latency + OS latency
  - OS latency cannot be measured independently from hardware
- How? Repeat the following in a loop:
  - Sleep until time = t1
  - Wake up at time = t2
  - Hardware + OS latency = t2 - t1
  - Do it while system is stressed



Exercise 1: measuring wakeup latency

- Standard tool: cyclictest
  - Measures OS + hardware latency for every core
- Today: dedicated latency tester written with Cactus-RT
- Why?
  - Easier-to-use tracing and visualization system based on Perfetto
  - The rest of the workshop exercises uses the same tracing/visualization setup
  - Gives you a chance to experience how to modify and run code for the rest of the workshop
- Setup:
  - 2 threads pinned on core 0 and core 1, each running at 1000 Hz
  - If wakeup latency > 1000us, deadline is missed

Cactus-RT: a framework for writing real-time C++ apps

- Enforces an application structure
- Takes care of RT-specific concerns for you
  - mlockall
  - Setup real-time schedulers
  - Setup real-time-safe logging
  - Setup real-time-safe tracing
- You can focus on writing the application
  - Request for a loop rate and fill out a `Loop` function
- Designed for 1000Hz real-time applications on Linux
- Open source: https://github.com/cactusdynamics/cactus-rt

Please form into groups of 3-5 with 1 person working on the Pi and rest on laptops!

Laptop workflow (step 1-3 in print out)

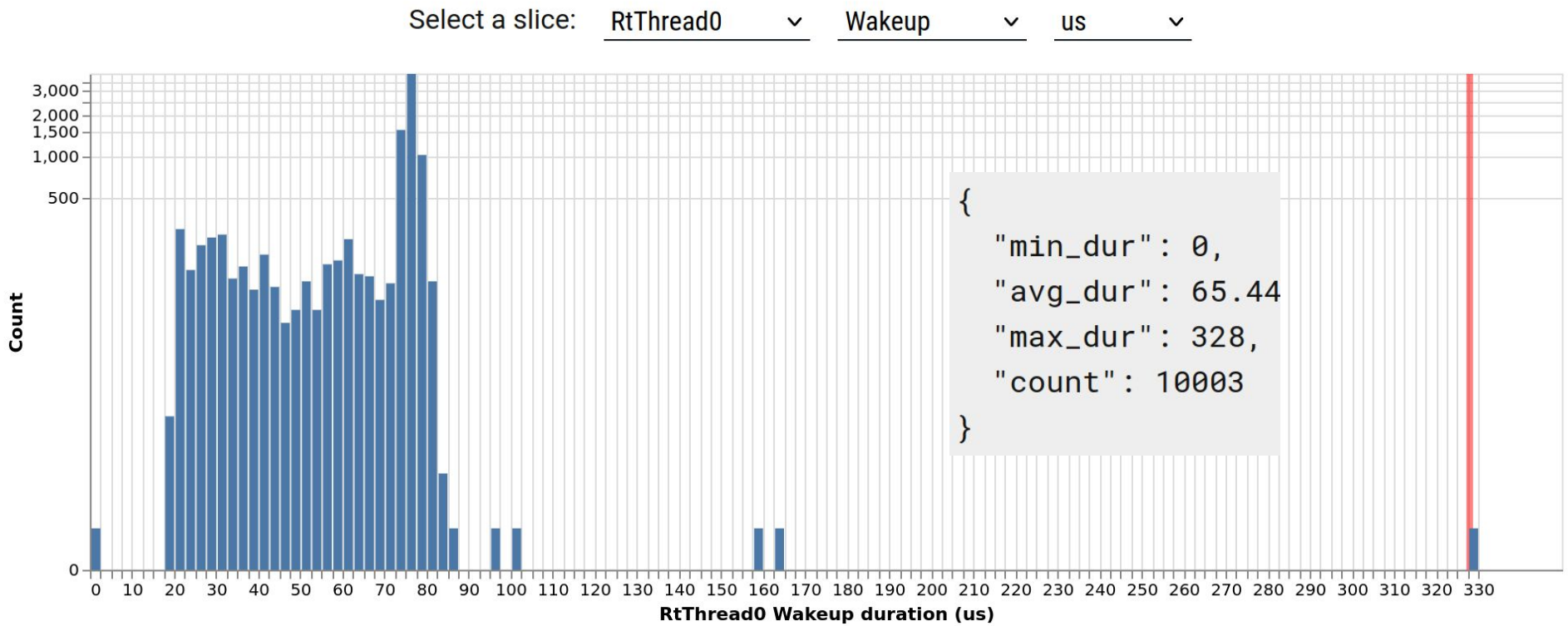
1. Follow print out to start Docker container
2. Terminal 1:
   a. `$ docker/shell`
   b. `$ cd /code/exercise1`
   c. `$ colcon build`
   d. `$ /code/stress.sh`
3. Terminal 2:
   a. `$ docker/shell`
   b. `$ cd /code/exercise1 && ./run.sh`
4. This generates `exercise1.perfetto` in the exercise1 directory
5. Stop the stress test in terminal 1
6. Go to http://localhost:3100, click Open trace file and load `exercise1.perfetto`
7. Click on Latency on the left sidebar
8. On the top of page, select 

9. Note the `max_dur` in the JSON output below the histogram. This is the maximum wakeup latency

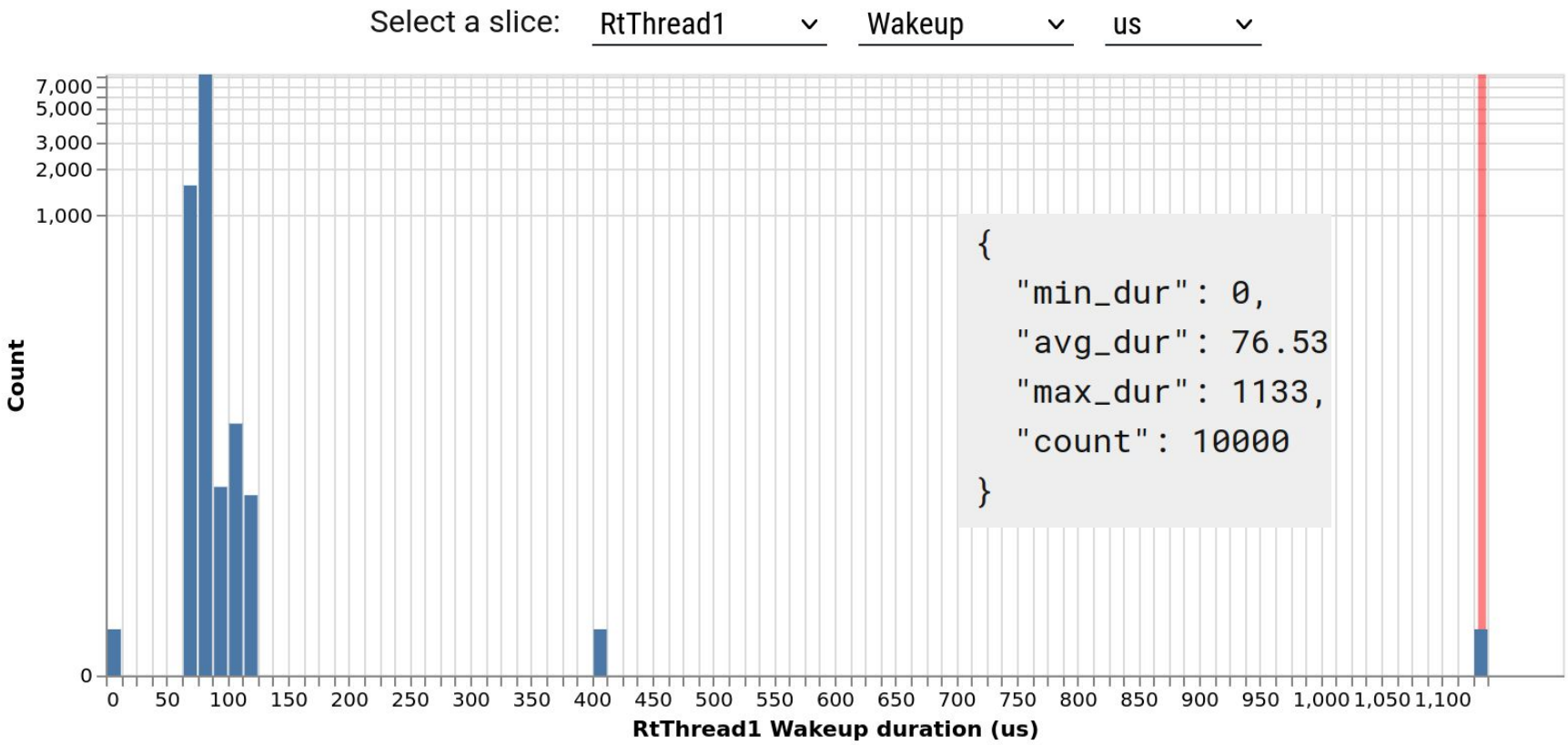
Raspberry Pi workflow

1. Follow print out to start Docker container
2. Connect Raspberry Pi with Ethernet
3. `$ ssh ubuntu@192.168.10.1`
   a. Password: `ubuntu`
4. Continue step 2 → 3 from the left (but without docker/shell and do it in the Raspberry Pi shell)
5. Go to http://192.168.10.1/repo
6. Navigate to `exercise1` and download `exercise1.perfetto`
7. Continue from step 7 on the left.

If you didn't finish, go to http://localhost:3100 and load the file from exercise1/solutions/baseline.perfetto and click the Latency link on the left sidebar



If you didn’t finish, go to http://localhost:3100 and load the file from exercise1/solutions/baseline.perfetto and click the Latency link on the left sidebar



Let's change it to a real-time scheduler!

1. Open exercise1/src/latency_tester/main.cc
2. Change the file in the following way

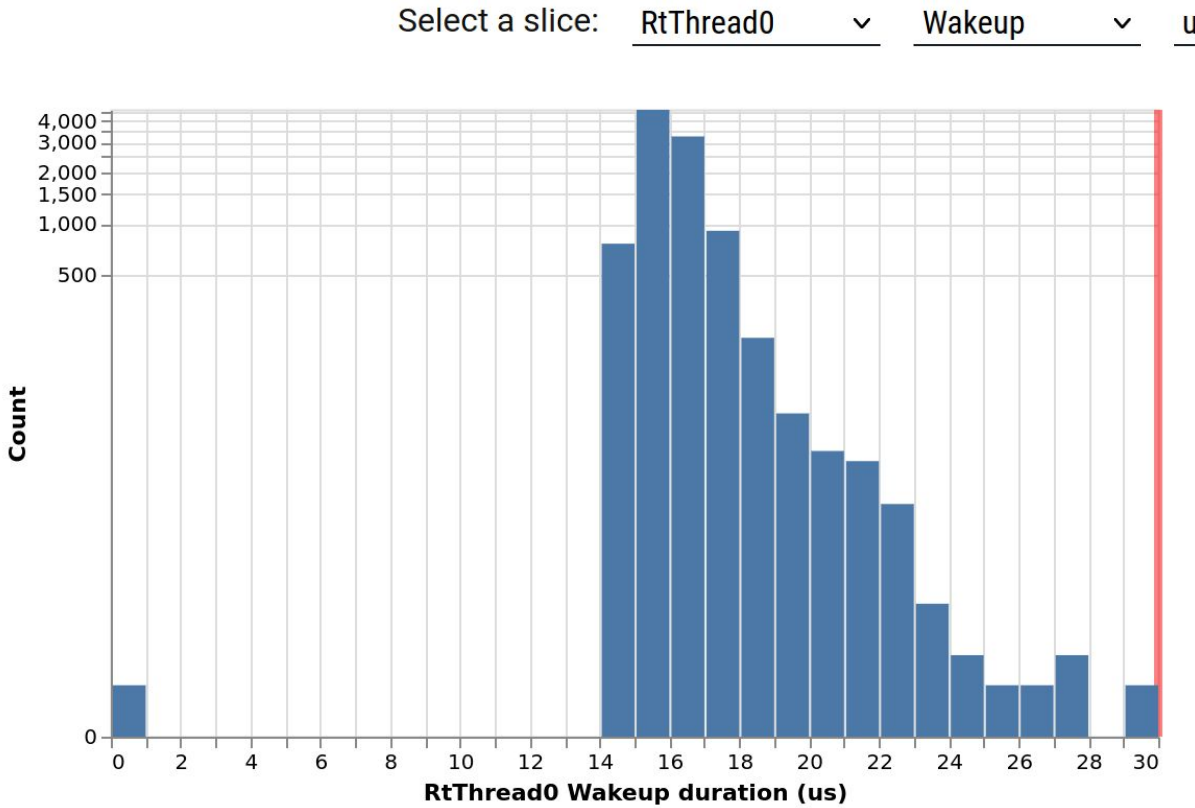
```
 9 cactus_rt::CyclicThreadConfig CreateRtThreadConfig(ui
10   cactus_rt::CyclicThreadConfig config;
11
12   config.SetOtherScheduler();          <- Delete this line
13
14   // Uncomment the following line to use the real-tim
15   // config.SetFifoScheduler(80);      <- Uncomment this line
16
17   config.cpu_affinity = {index};
18
19   config.tracer_config.trace_loop = true;
20   config.tracer_config.trace_overrun = true;
21   config.tracer_config.trace_sleep = true;
22   config.tracer_config.trace_wakeup_latency = true;
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:
   a. `$ docker/shell`
   b. `$ cd /code/exercise1 && ./run.sh`
3. Stop the stress in terminal 1.
4. In Perfetto (http://localhost:3100), click Open trace file and reload the exercise1.perfetto file
5. Load the Latency view from the left side bar

Raspberry Pi users:

1. `$ docker/shell`
2. `$ upload-to-pi`
3. `$ ssh ubuntu@192.168.10.1`
4. Step 1-2 above (without the docker/shell but in the Raspberry Pi shell)
5. Download data from http://192.168.10.1/repo
6. Load download exercise1.perfetto in Perfetto

If you didn’t finish, go to http://localhost:3100 and load the file from exercise1/solutions/solution.perfetto and click the Latency link on the left sidebar




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

Results: wakeup latency with and without RT scheduler

- Wakeup latency = Hardware + OS latency
- Total latency >= Wakeup latency

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

- **Conclusion: real-time scheduler is necessary for real-time applications!**
  - **PREEMPT_RT is also preferred**
  - **Hardware without real-time latency is also needed**
- Use cyclictest for production applications

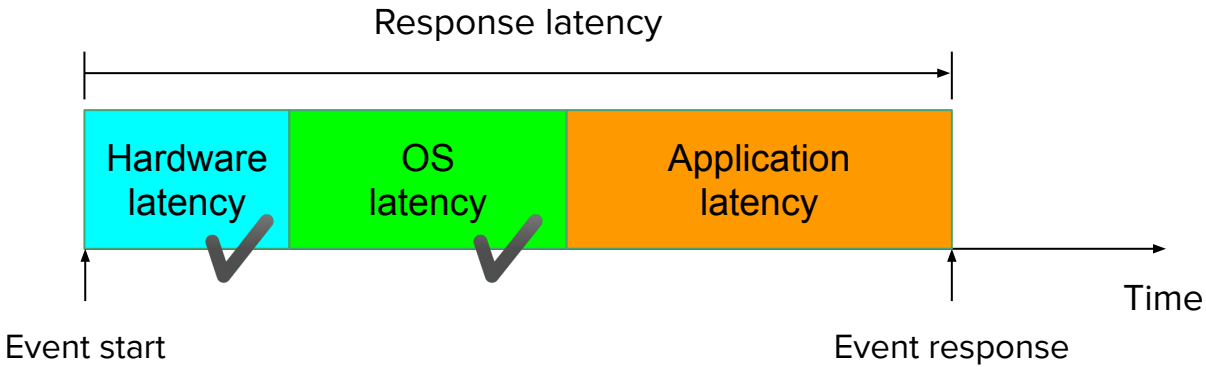
Notes about the rest of the workshop exercises

- All follow a similar workflow
- Refer to cheatsheet on the workflow and usage of various systems (Perfetto, Raspberry Pi, Docker)
- Help others in your group if they are stuck
- Ask questions if you have them

Introduction to RT programming on Linux without ROS

Recap

- Hardware + OS latency: resolved!
  - At least on high level
- Common misconception: PREEMPT_RT will reduce the response latency
  - Response latency = hardware latency + OS latency + application latency
- Application latency frequently the issue
  - Addressed in this session



What's wrong with this code?

```
std::vector<double> logs;


void Loop() {
  auto K = GetPidConstants();
  auto e = ComputeError();
  auto out = ComputePidOutput();

  logs.push_back(out);

  std::cout << “Output: ”
            << out << “\n”;
}
```

```
std::mutex mut;
Pid pid = {1.0, 1e-5, 1e-3};

PidConstant GetPidConstants() {
  std::scoped_lock lock(mut);
  return pid;
}
```

What's wrong with this code?

```
std::vector<double> logs;


void Loop() {
  auto K = GetPidConstants();
  auto e = ComputeError();
  auto out = ComputePidOutput();

  logs.push_back(out);

  std::cout << "Output: "
            << out << "\n";
}
```

Dynamic memory allocation and O(n) copying

```
std::mutex mut;
Pid pid = {1.0, 1e-5, 1e-3};

PidConstant GetPidConstants() {
  std::scoped_lock lock(mut);
  return pid;
}
```

What's wrong with this code?

```
std::vector<double> logs;


void Loop() {
  auto K = GetPidConstants();
  auto e = ComputeError();
  auto out = ComputePidOutput();

  logs.push_back(out);

  std::cout << "Output: "
            << out << "\n";
}
```

Dynamic memory allocation and O(n) copying

Logging

```
std::mutex mut;
Pid pid = {1.0, 1e-5, 1e-3};

PidConstant GetPidConstants() {
  std::scoped_lock lock(mut);
  return pid;
}
```

What's wrong with this code?

```
std::vector<double> logs;


void Loop() {
  auto K = GetPidConstants();
  auto e = ComputeError();
  auto out = ComputePidOutput();

  logs.push_back(out);

  std::cout << "Output: "
            << out << "\n";
}
```

Dynamic memory allocation and O(n) copying

Logging

```
std::mutex mut;
Pid pid = {1.0, 1e-5, 1e-3};

PidConstant GetPidConstants() {
  std::scoped_lock lock(mut);
  return pid;
}
```

Usage of std::mutex

What's wrong with this code?

```
std::vector<double> logs;


void Loop() {
  auto K = GetPidConstants();
  auto e = ComputeError();
  auto out = ComputePidOutput();

  logs.push_back(out);

  std::cout << "Output: "
            << out << "\n";
}
```

Dynamic memory allocation and O(n) copying

Logging

```
std::mutex mut;
Pid pid = {1.0, 1e-5, 1e-3};

PidConstant GetPidConstants() {
  std::scoped_lock lock(mut);
  return pid;
}
```

Possible page fault

Usage of std::mutex

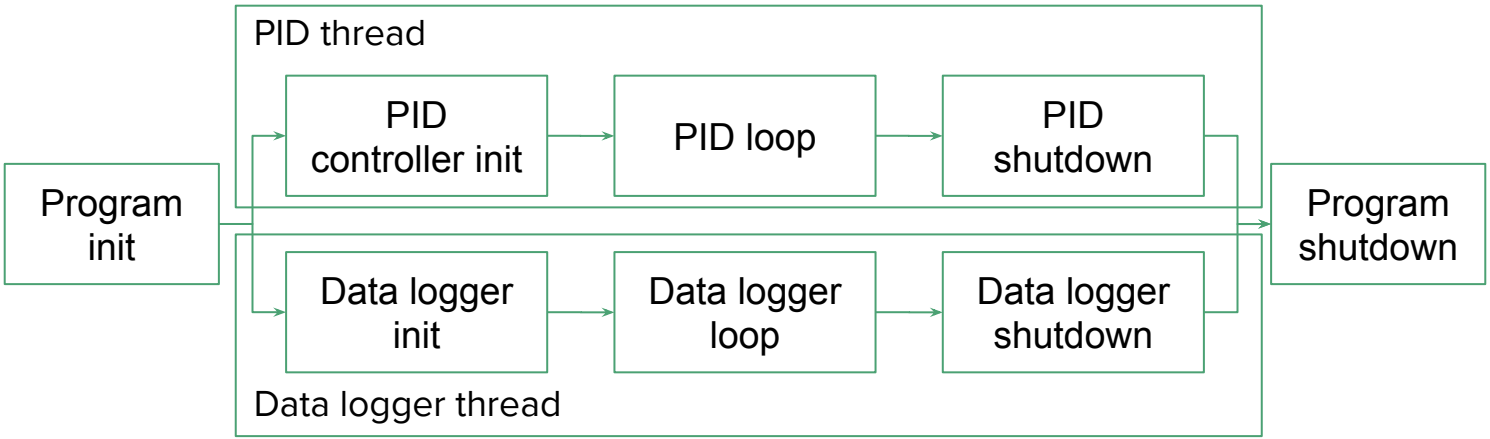
What’s wrong with this code?

- By the end of this session, we will know why these (and more) are wrong
  - Also how we can address them
- Let’s take a step back and define a typical real-time application

`scoped_lock lock(mut)`	Usage of std::mutex
`logs.push_back(out)`	Dynamic memory allocation and O(n) copying
`std::cout << “...”`	Logging
`return pid`	Possible page fault

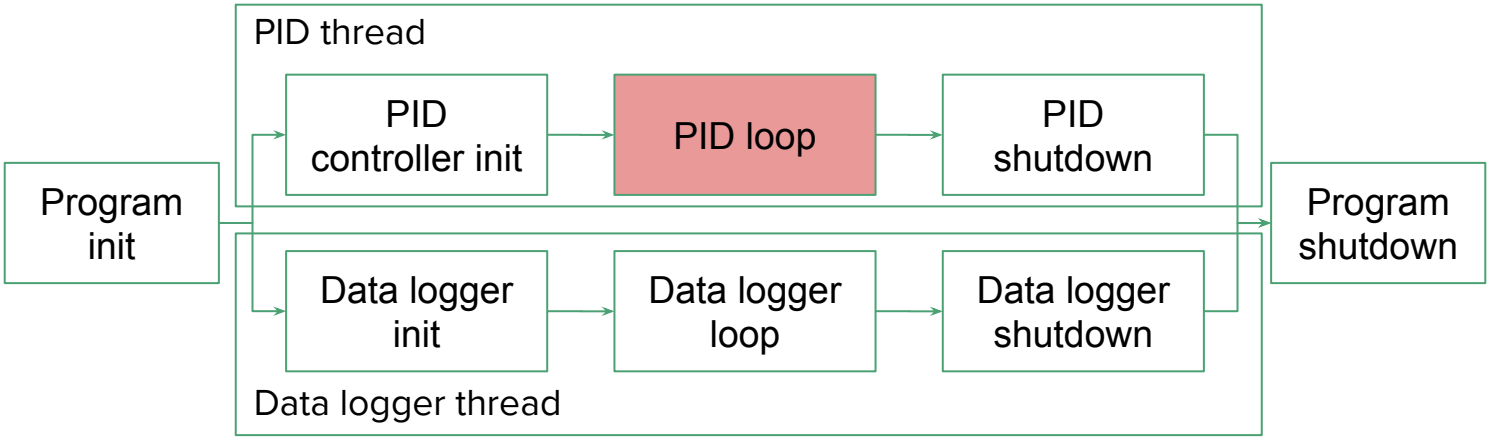
Typical real-time application architecture

- Real-time applications tend to have many threads
  - At least 1 RT thread, typically at least 1 non-RT thread
  - RT and non-RT threads typically need to communicate with each other



Typical real-time application architecture

- Real-time applications tend to have many threads
  - At least one RT thread, typically at least one non-RT thread
  - RT and non-RT threads typically need to communicate with each other
- Within the RT thread, there are sections that are more critical than others
  - Non-critical sections do not need to be “real-time-compliant” (not always true)



Real-time vs non-real-time section

```
std::vector<double> logs;

void NonRT_Setup() {
  logs.push_back(3.14);
  std::cout << “...”;
}
```

```
std::vector<double> logs;

void RT_Loop() {
  logs.push_back(3.14);
  std::cout << “...”;
}
```

Real-time vs non-real-time section

```
std::vector<double> logs;

void NonRT_Setup() {
  logs.push_back(3.14);
  std::cout << “...”;
}
```

OK in non-real-time section ✅

```
std::vector<double> logs;

void RT_Loop() {
  logs.push_back(3.14);
  std::cout << “...”;
}
```

Not OK in real-time section ❌

All techniques only applicable to the real-time sections

Virtual memory and its role in memory latency

- Where does a pointer point to?
  - Address in memory?

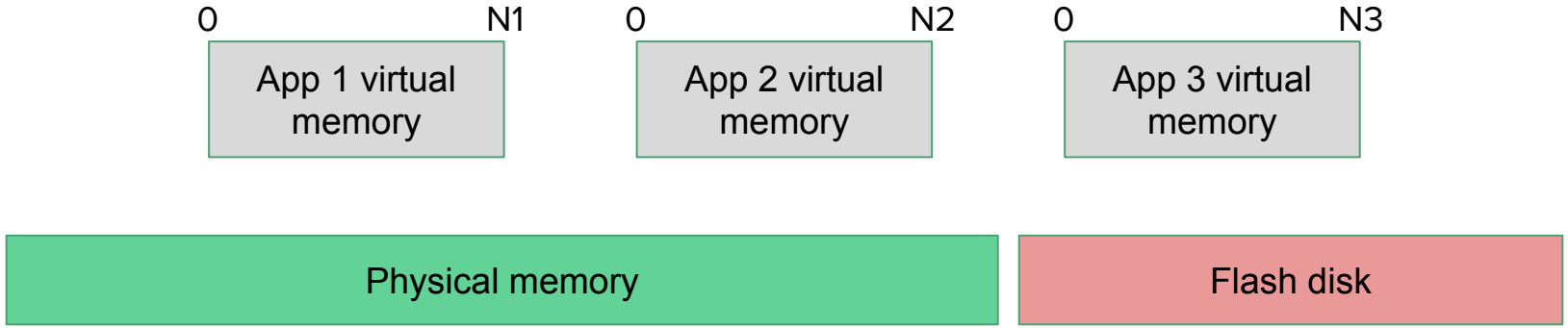
```
return pid
```

Physical memory

Virtual memory and its role in memory latency

```
return pid
```

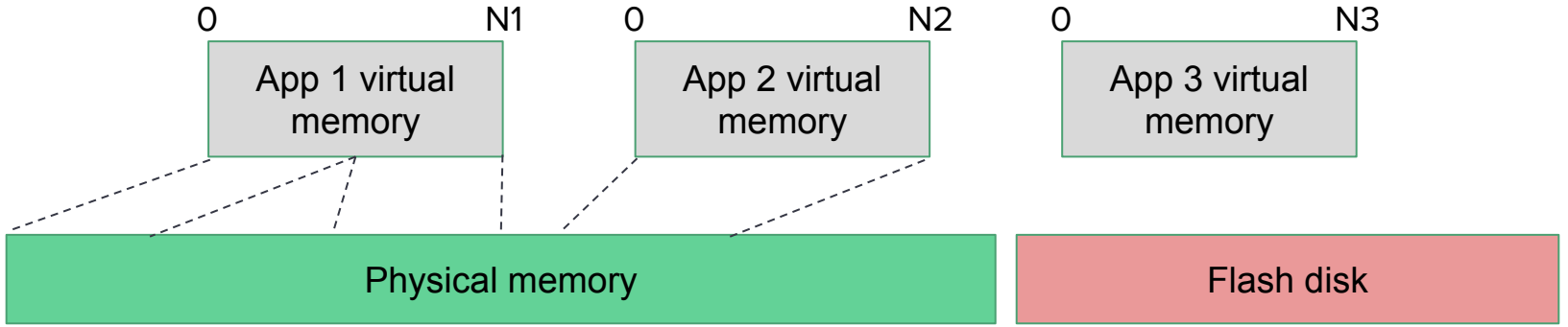
- Where does a pointer point to?
  - Address in **virtual** memory
- Virtual memory: provides private memory address space for each process
  - Two processes cannot collide if they write to the same virtual pointer address



Virtual memory and its role in memory latency

```
return pid
```

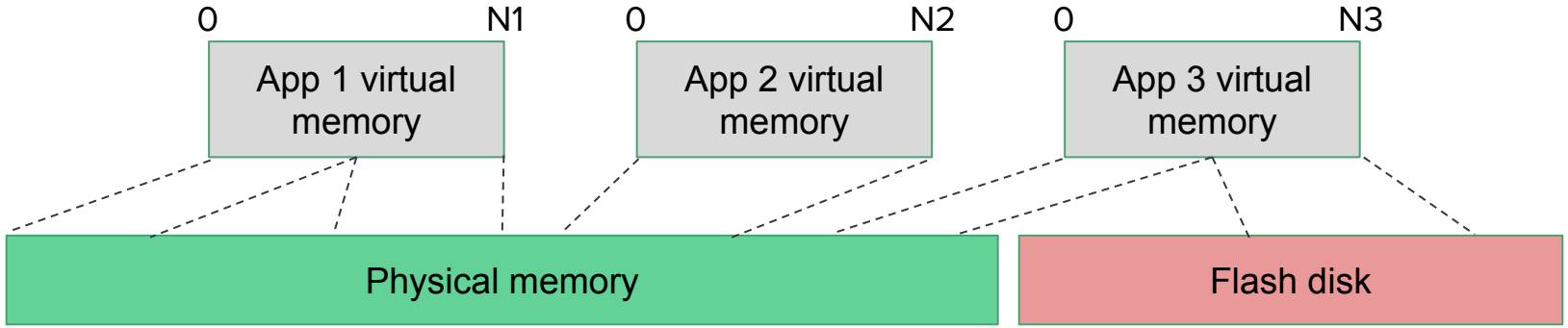
- Where does a pointer point to?
  - Address in **virtual** memory
- Virtual memory: provides private memory address space for each process
  - Two processes cannot collide if they write to the same virtual pointer address
  - Upon dereference, virtual address is translated to physical address



Virtual memory and its role in memory latency

```
return pid
```

- Where does a pointer point to?
  - Address in **virtual** memory
- Virtual memory: provides private memory address space for each process
  - Two processes cannot collide if they write to the same virtual pointer address
  - Upon dereference, virtual address is translated to physical address
- **OS can locate some addresses on disks!**



Memory latency

- CPU is fast; memory access is not
- Reading/writing to different memory can have widely varying latency

Event	Latency
1 CPU cycle (at 3 GHz)	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	3 ns
Memory access	100 ns
Solid state disk access	10 - 100 µs

Source: System Performance 2e by Brendan Gregg, Table 2.2

Memory latency

- CPU is fast; memory access is not
- Reading/writing to different memory can have widely varying latency
- A single memory read/write can be 1-10% of a 1 kHz loop iteration!

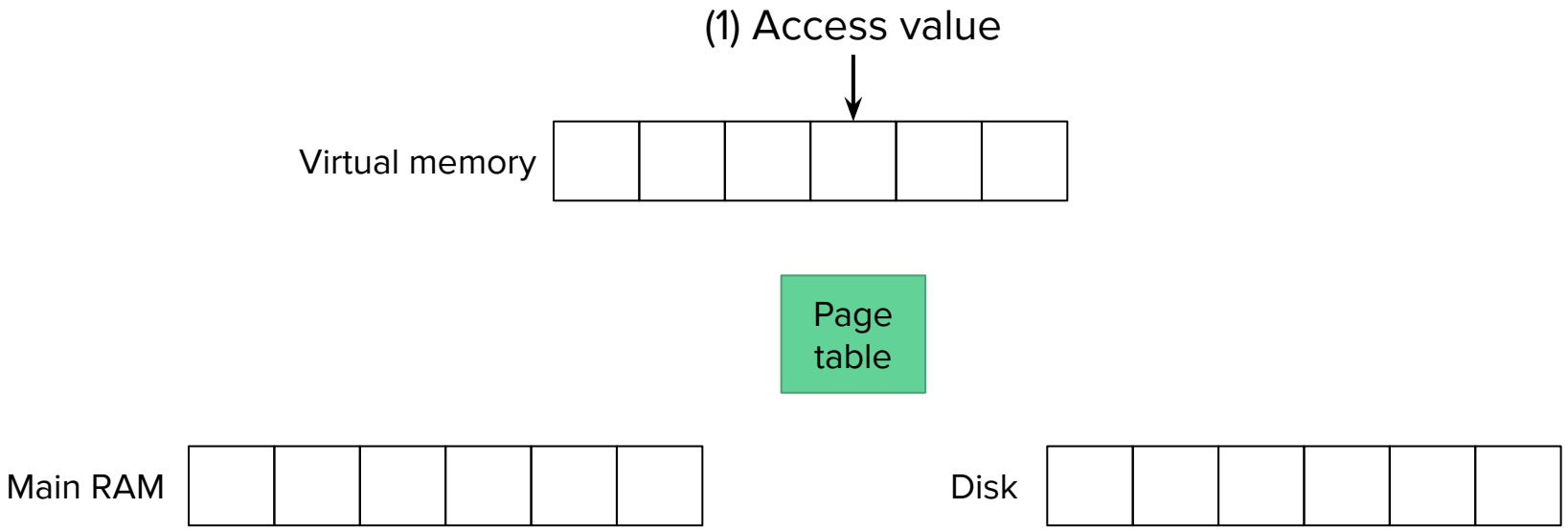
Event	Latency
1 CPU cycle (at 3 GHz)	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	3 ns
Memory access	100 ns
Solid state disk access	10 - 100 µs

Source: System Performance 2e by Brendan Gregg, Table 2.2

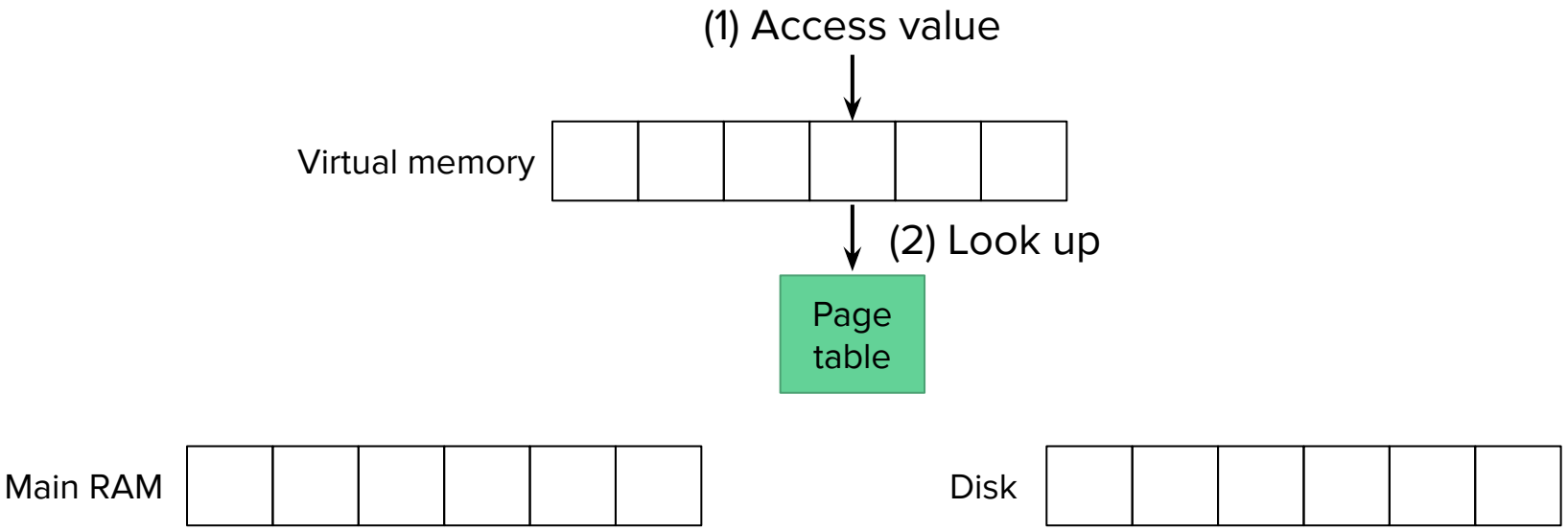
Page fault

- Asking OS to allocate memory doesn’t actually do that
- OS will wait until you access the memory you requested before allocating
- Page fault…

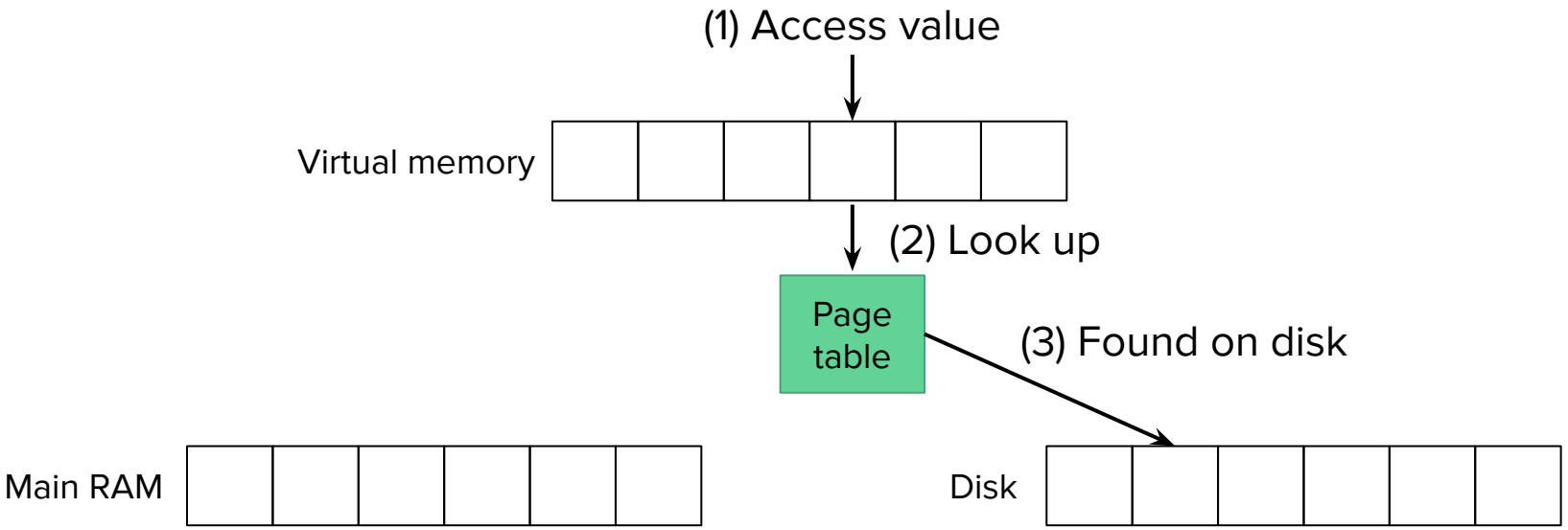
Page fault



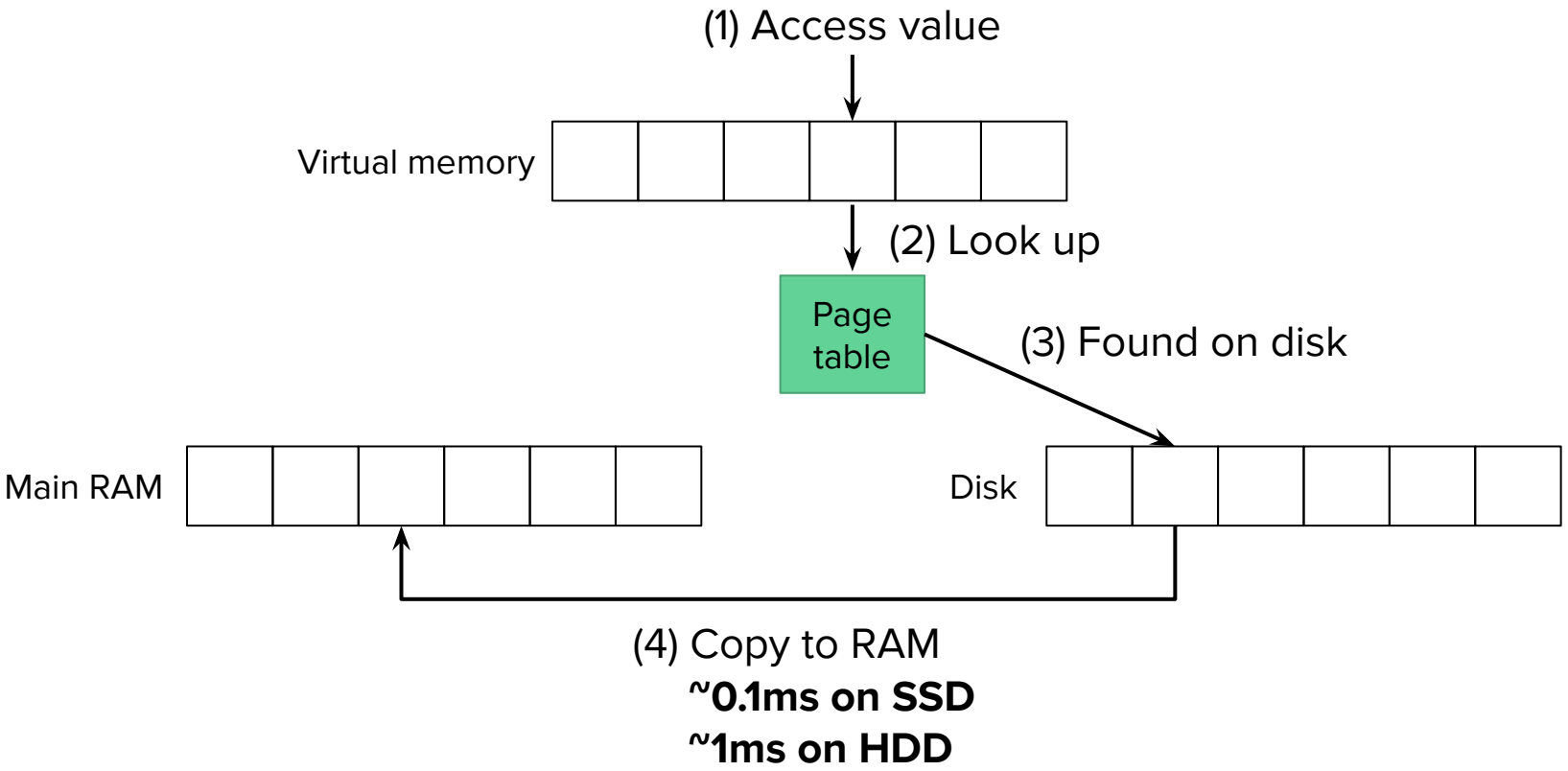
Page fault


(1) Access value
Virtual memory
(2) Look up
Page table
Main RAM
Disk

Page fault


(1) Access value
Virtual memory
(2) Look up
Page table
(3) Found on disk
Main RAM
Disk

Page fault


(1) Access value
Virtual memory
(2) Look up
Page table
(3) Found on disk
Main RAM
Disk
(4) Copy to RAM
~0.1ms on SSD
~1ms on HDD

Swapping

- Demand paging is not the only reason for page faults
- OS may copy unused memory to disk to save RAM
    - Copies it back upon access

Solutions to latency induced by page faults

- The `mlockall` function instructs the kernel to commit and lock all allocated memory into physical RAM.
- Call `mlockall(MCL_CURRENT | MCL_FUTURE)` at application startup
- Turns demand paging and swapping off for this application

~~`return pid` Possible page fault~~

Dynamic memory allocation

```
logs.push_back(out)
```

- Dynamic memory allocation is very common in C++
  - `std::vector::push_back`
  - `std::function` (sometimes)
- Two problems with memory allocation:
  - Memory allocation is usually not O(1)
  - Can trigger page faults even if `mlockall(MCL_CURRENT | MCL_FUTURE)` is called
- Memory allocation is mostly of the time fast, but occasionally very slow
  - Not real-time-safe

Solutions to latency induced by dynamic allocation

- Do not dynamically allocate and statically allocate
  - `vector.reserve(N)` before program enters RT section
  - Also remember to pre-allocate the stack
- Advanced and out of scope: use O(1) allocators
  - TLSF allocator
  - Reserve heap space
  - Prevent OS from reclaiming heap space

~~`logs.push_back(out)` Dynamic memory allocation and O(n) copying~~

Data sharing between threads

- Reading and writing the same variable from 2 threads without synchronization can cause **memory corruption** due to **undefined behavior**!

```
bool running = true;

void thread1() {
  while (running) {
      doStuff();
  }
}

void thread2() {
  sleep(10s);
  running = false;
}
```

Compiler optimization

```
bool running = true;

void thread1() {
  while (true) {
      doStuff();
  }
}

void thread2() {
  sleep(10s);
  running = false;
}
```

Data sharing between threads

- A mutex must be used guard access to shared variables
  - Mutexes cooperate with the scheduler to suspend the thread until mutex is released

```
std::mutex mut;
double pid[] = {1.0, 1e-5, 1e-3};

void high_priority_thread() {
  while (true) {
    {
      std::scoped_lock l(mut);
      controller(pid[0], pid[1], pid[2]);
    }
    sleep(2ms);
  }
}
```

Read operation

```
void low_priority_thread() {
  sleep(10s);
  {
    std::scoped_lock l(mut);
    pid[0] = calc_p();
  }
}
```

Write operation

Data sharing between threads

- A mutex must be used guard access to shared variables
  - Mutexes cooperate with the scheduler to suspend the thread until mutex is released
- Normal mutexes unusable in real-time due to **priority inversion**

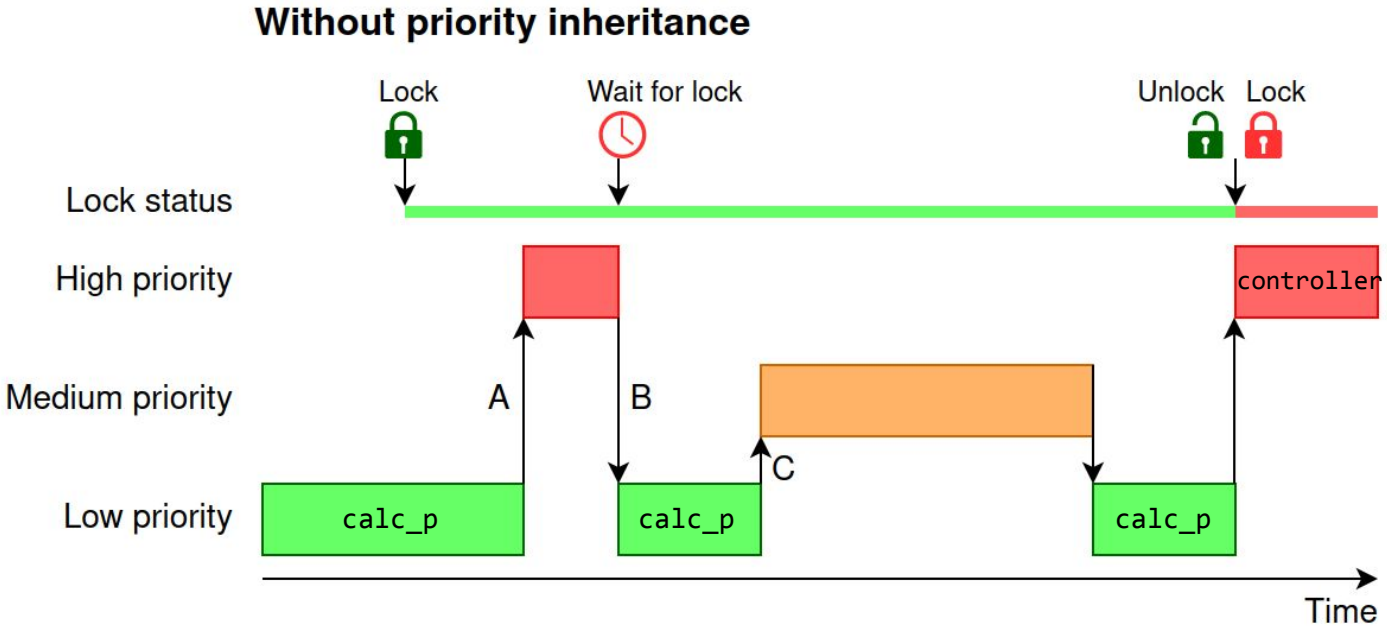
```
std::mutex mut;
double pid[] = {1.0, 1e-5, 1e-3};

void high_priority_thread() {
  while (true) {
    {
      std::scoped_lock l(mut);
      controller(pid[0], pid[1], pid[2]);
    }
    sleep(2ms);
  }
}
```

Read operation

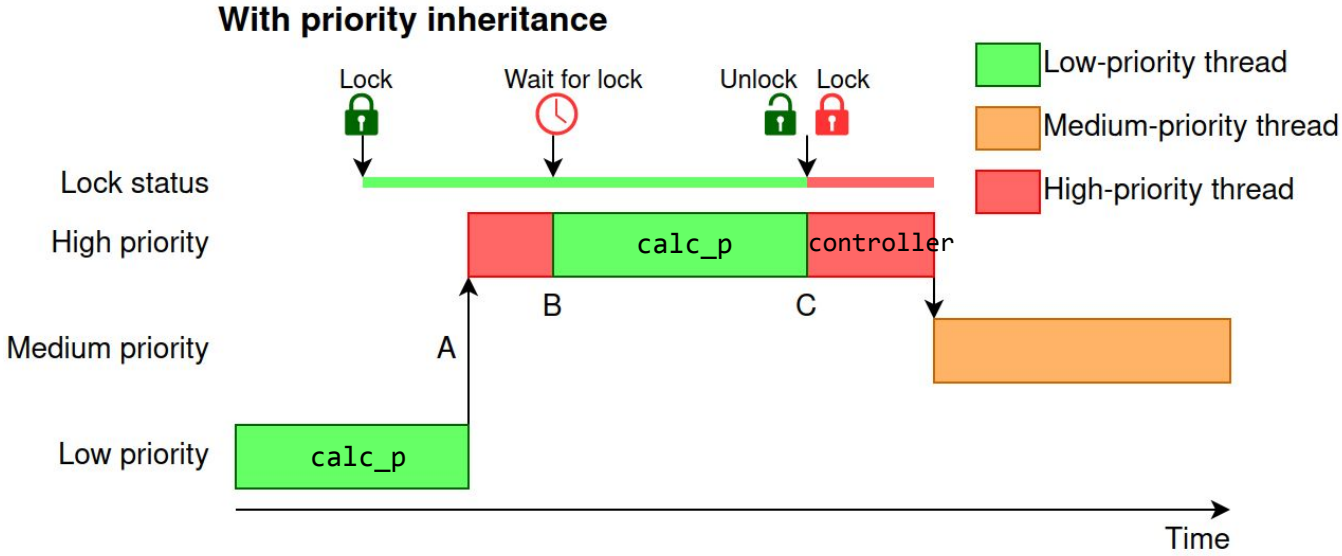
```
void low_priority_thread() {
  sleep(10s);
  {
    std::scoped_lock l(mut);
    pid[0] = calc_p();
  }
}
```

Write operation

std::mutex: priority inversion



Solution to priority inversion: priority inheritance



- `calc_p` must be written to be real-time-safe
- Priority-inheritance mutex reduces latency
- Standard library doesn't support priority inheritance mutex
  - Easily implement your own on Linux (~70 lines)
  - Example implementation: https://github.com/cactusdynamics/cactus-rt/blob/master/include/cactus_rt/mutex.h

Solution to priority inversion: lockless programming

- Sometimes priority-inheritance mutex is still too slow
  - We will see this in the exercises!
- Lockless programming is an advanced technique to safely share data between cores without the usage of locks
- Can be difficult to design, program, and test

Solution to priority inversion: lockless programming

- Different algorithms for different data sizes
- Sharing small amounts of data: atomic variables
- Sharing large amounts of data:
  - Lockless queues (ring buffer)
  - Advanced lockless algorithms
- Lockless programming more common in real-time audio, HFT applications
  - Because Windows doesn’t support priority-inheritance mutex
- If doing lockless: most of the time should use a library
  - Designing lockless algorithms outside the scope of this workshop

Solutions to std::mutex

- Never use std::mutex in real-time sections
- Use priority-inheritance mutex
- Use lockless programming

~~`scoped_lock lock(mut)` Usage of std::mutex~~

Printing/logging

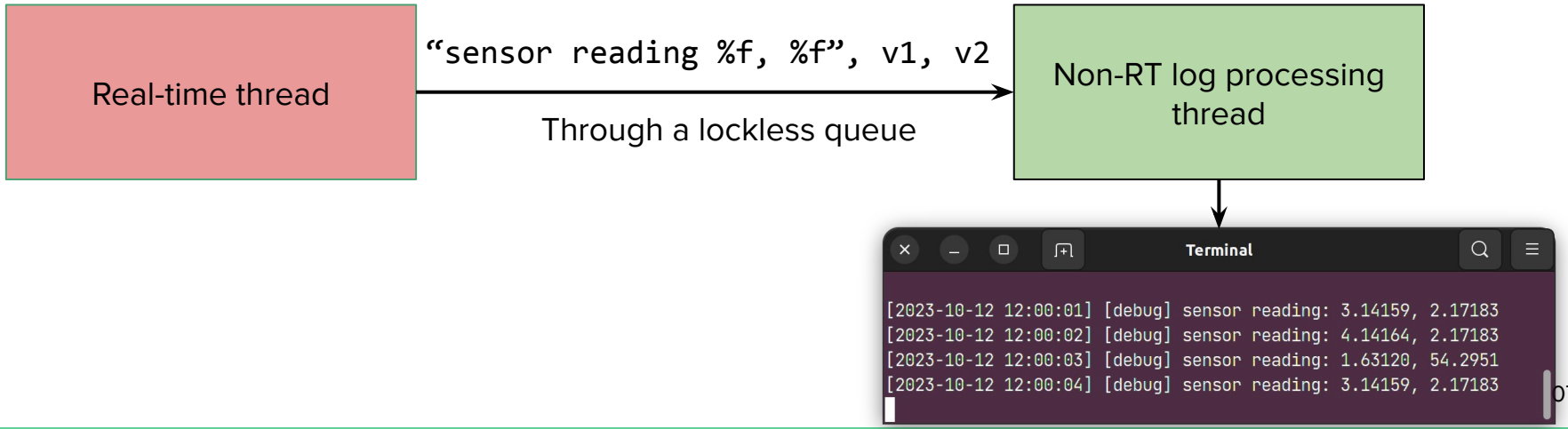
- Printing and logging is generally not real-time-safe
- String formatting usually requires allocation
  - Page fault
- Writing to IO may involve mutexes
  - Priority inversion

Printing/logging

- Normal logging and printing is not real-time-safe
  - Should never call normal print/log functions in real-time sections
- However:
  - Logging is a crucial debugging tool
  - Production applications may require logging for legal reasons

Solutions to printing/logging

- Pass format string pointers, format arguments from RT thread to non-RT thread
  - Through lockless queue
  - Format string and log in non-RT thread
- Known as “asynchronous logging”
- Quill is an existing library that implements this



Data logging and tracing

- Lockless queue can be used to pass other data out of the real-time loop
- Example:
  - Sensor and control data
  - Application timing information

What's wrong with this code?

```
std::vector<double> logs;


void Loop() {
  auto pid = GetPidConstants();
  auto e = ComputeError();
  auto out = ComputePidOutput();

  logs.push_back(out);

  std::cout << "Output: "
            << out << "\n";
}
```

```
std::mutex mut;
Pid pid = {1.0, 1e-5, 1e-3};

PidConstant GetPidConstants() {
  std::scoped_lock lock(mut);
  return pid;
}
```

Dynamic memory allocation and O(n) copying

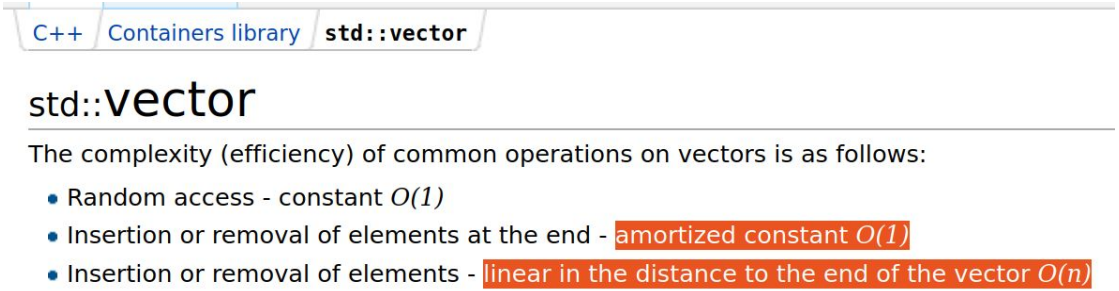
Logging

Possible page fault

Usage of std::mutex

Latency from libraries

- Library calls and system calls can also have high occasional latency
- Do library/system calls:
  - Allocate memory?
  - Use normal mutexes?
  - Uses a non-*O*(1) algorithm that only triggers occasionally?
  - Call to slow IO internally?
  - Call other library/system calls that does the above?



Solutions for latency from libraries

- Not a lot of shortcuts
- Likely requires code review or experimentation
  - Some libraries are known to be allocation-free and lock-free and *O*(1)
- “Proof is left as an exercise for the reader”
  - Example: how do you known that collision detection library has bounded latency?
- Due diligence is required!

Summary: Real-time programming with C++

- Turn off demand paging and swapping
- Do not dynamically allocate memory
- Use priority-inheritance mutex or lockless programming instead of std::mutex
- Use asynchronous logging system to print
  - Use lockless queues to for data logging
- Ensure libraries you call also have bounded latency

Additional resources

- Systems Performance: Enterprise and the Cloud, 2nd Edition - Brendan Gregg
  - Chapter 7: memory; Chapter 3: OS
- Real-time Programming with the C++ Standard Library - Timur Doumler
- Real-time 101: Part 1, Part 2 - Fabian Renn-Giles & Dave Rowland
- A realtime developer's checklist - Marta Rybczyńska
- Sources of latency in real-time applications - Shuhao Wu
- Challenges Using Linux as a Real-Time Operating System - Michael Madden

Hands-on-exercise 2

Inverted Pendulum

- 1000 Hz real-time inverted pendulum controller
- Pendulum controller communicates with ROS
- Start in the Docker container or on the Pi

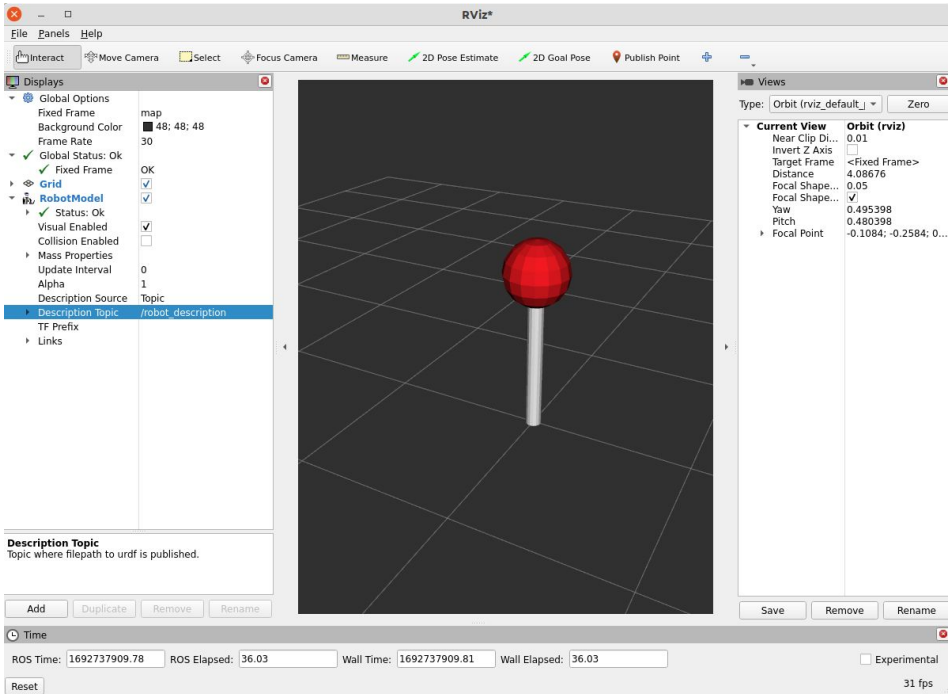
```
$ docker/shell
```

- Change directories to `exercise2-1`:

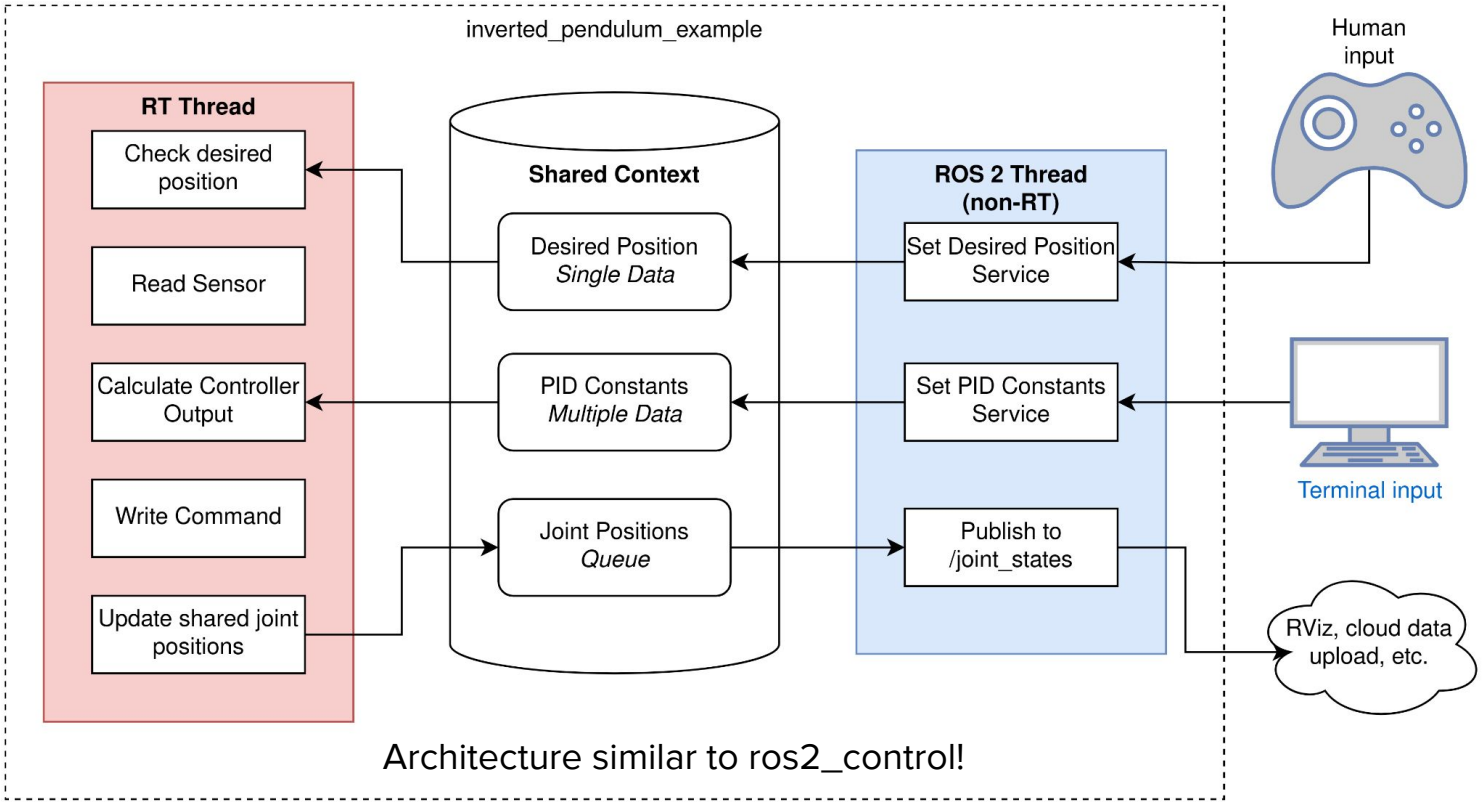
```
$ cd exercise2-1
```

- Build the workspace:

```
$ colcon build
```



Inverted Pendulum Structure



Inverted Pendulum Setup

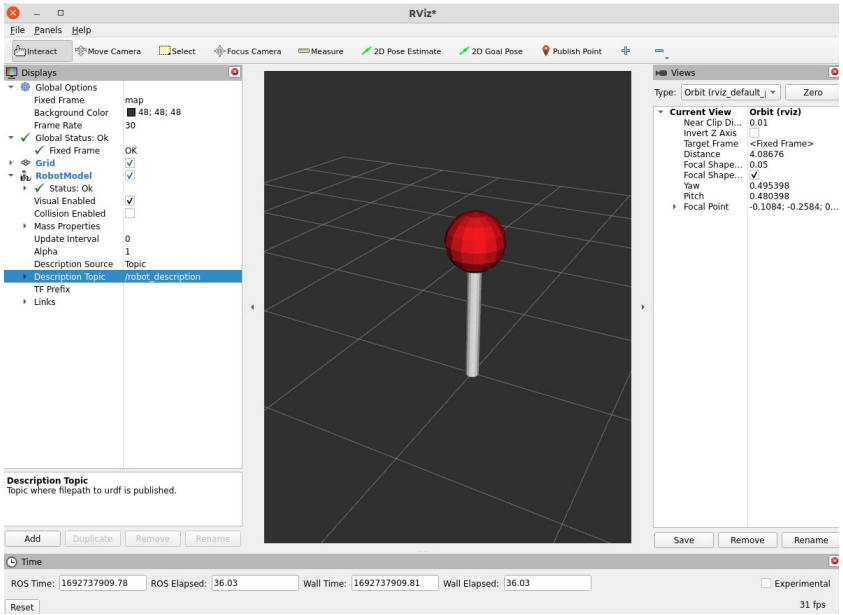
- Stress:

  ```
  $ /code/stress.sh
  ```

- Run the exercise in a different terminal:

  ```
  $ docker/shell
  $ cd exercise2-1 && ./run.sh
  ```

- Stop stress and run after several seconds
  - Generates a file called exercise2-1.perfetto
  - Raspberry Pi users:
    - Connect to the Raspberry Pi via Ethernet
    - Open a browser and go to 192.168.10.1/repo/exercise2-1 to download the file
  - Open this in Perfetto localhost:3100



Interaction

- Reset the pendulum to its initial state:

```
$ ros2 service call /reset_pendulum std_srvs/srv/Empty
```

- Change the pendulum setpoint between (-pi / 2, pi / 2)

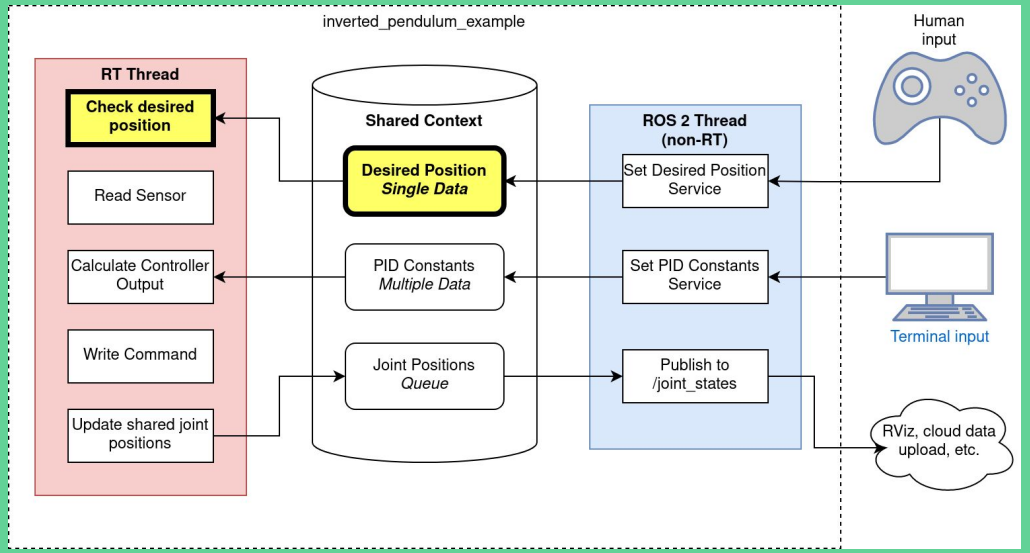
```
$ ros2 service call /set_desired_position
inverted_pendulum_interfaces/srv/SetDesiredPosition "{desired_position:
0.5}"
```

- Change the PID constants

```
$ ros2 service call /set_PID_constants
inverted_pendulum_interfaces/srv/SetPIDConstants "{kp: 0, ki: 0, kd: 0}"
```

Exercise 2-1

Single Data



Exercise 2-1: Single Data

- `$ cd /code/exercise2-1`
- Continuously change desired position
  - Sinusoid
  - Example - input from a game controller

```
auto set_desired_positions_thread = std::thread(
  [&]() {
    while (true) {
      const auto now = system_clock::now();
      const auto now_millis = duration_cast<milliseconds>(now.time_since_epoch()).count();
      // ~6 second period
      shared_context->desired_position.Set(sin(static_cast<double>(now_millis) / 1000));
    }
  }
);
```

Exercise 2-1: Single Data

`exercise2-1/src/inverted_pendulum/include/inverted_pendulum/message_passing/single_data.h`

```cpp
#include <mutex>

struct SingleData {
  void Set(const double value) {
    std::scoped_lock lock(mutex_);
    value_ = value;
  }

  double Get() {
    std::scoped_lock lock(mutex_);
    return value_;
  }

 private:
  std::mutex mutex_;
  double value_;
};
```

Called by ROS thread when new desired position is set

Exercise 2-1: Single Data

`exercise2-1/src/inverted_pendulum/include/inverted_pendulum/message_passing/single_data.h`

```cpp
#include <mutex>

struct SingleData {
  void Set(const double value) {
    std::scoped_lock lock(mutex_);
    value_ = value;
  }

  double Get() {
    std::scoped_lock lock(mutex_);
    return value_;
  }

 private:
  std::mutex mutex_;
  double value_;
};
```

Called by ROS thread when new desired position is set

Called by RT thread every iteration!
Has to complete within 1 ms

Exercise 2-1: Single Data

`exercise2-1/src/inverted_pendulum/include/inverted_pendulum/message_passing/single_data.h`

```cpp
#include <mutex>

struct SingleData {
  void Set(const double value) {
    std::scoped_lock lock(mutex_);
    value_ = value;
  }

  double Get() {
    std::scoped_lock lock(mutex_);
    return value_;
  }

 private:
  std::mutex mutex_;
  double value_;
};
```

Called by ROS thread when new desired position is set

Lock contention between these

Called by RT thread every iteration!
Has to complete within 1 ms

Exercise 2-1: Single Data

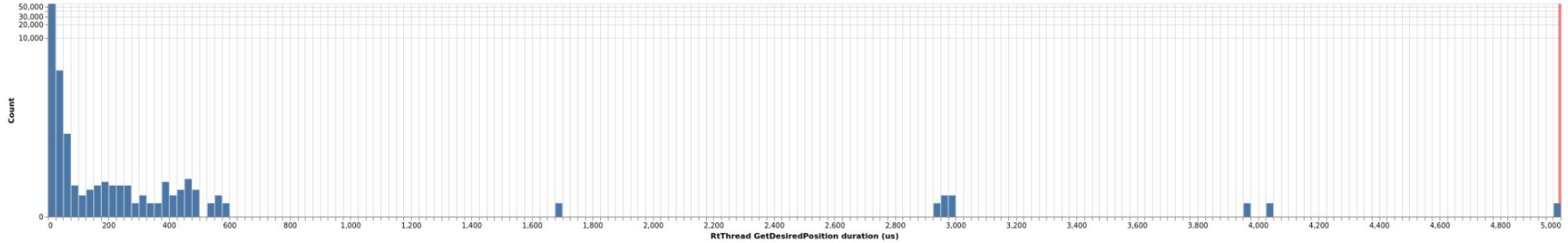
`exercise2-1/src/inverted_pendulum/include/inverted_pendulum/message_passing/single_data.h`

- Build:
  - `$ cd /code/exercise2-1 && colcon build`
- Stress (after building):
  - `$ /code/stress.sh`
- Run the example in a new terminal with `docker/shell`:
  - `$ ./run.sh`
  - Logs will output if there is a loop overrun
- Stop it after several seconds
  - Generates a file called exercise2-1.perfetto
  - Raspberry Pi users:
    - Connect to the Raspberry Pi via Ethernet
    - Open a browser and go to 192.168.10.1/repo/exercise2-1 to download the file
  - Open this in Perfetto localhost:3100
- Interested in GetDesiredPosition slice

Exercise 2-1: Single Data

`exercise2-1/src/inverted_pendulum/include/inverted_pendulum/message_passing/single_data.h`

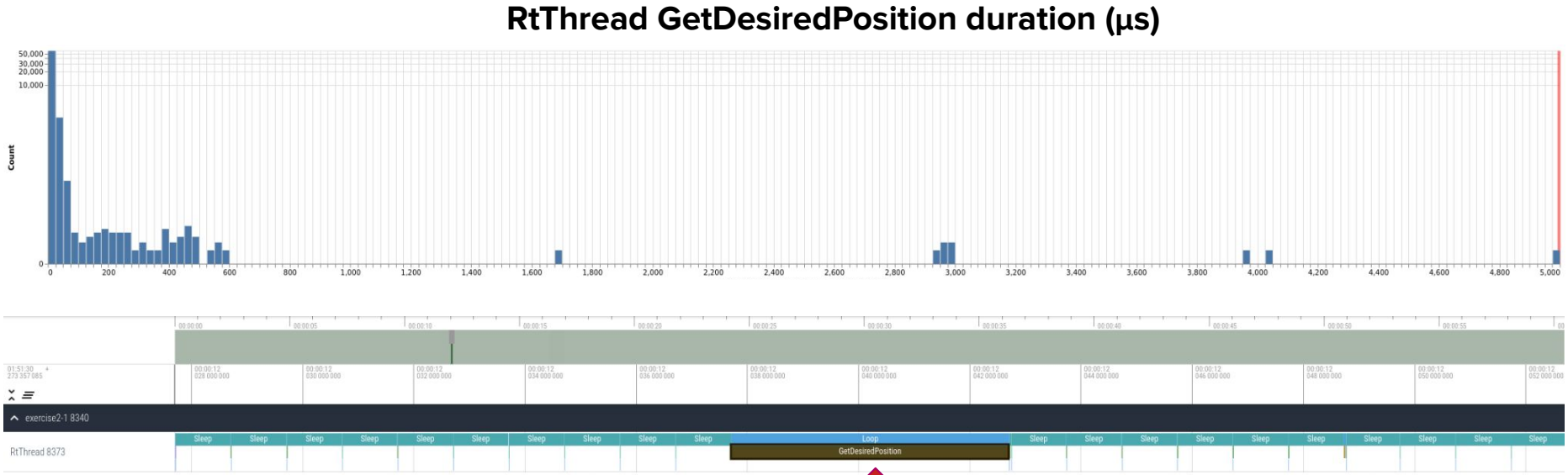
- We have already run baseline case exercise2-1
  - Generated a file called exercise2-1.perfetto
  - Raspberry Pi users:
    - Connect to the Raspberry Pi via Ethernet
    - Open a browser and go to http://192.168.10.1/repo/exercise2-1 to download the file
  - Open this in Perfetto http://localhost:3100
- Interested in **GetDesiredPosition** slice in Latency tab



Exercise 2-1: Single Data

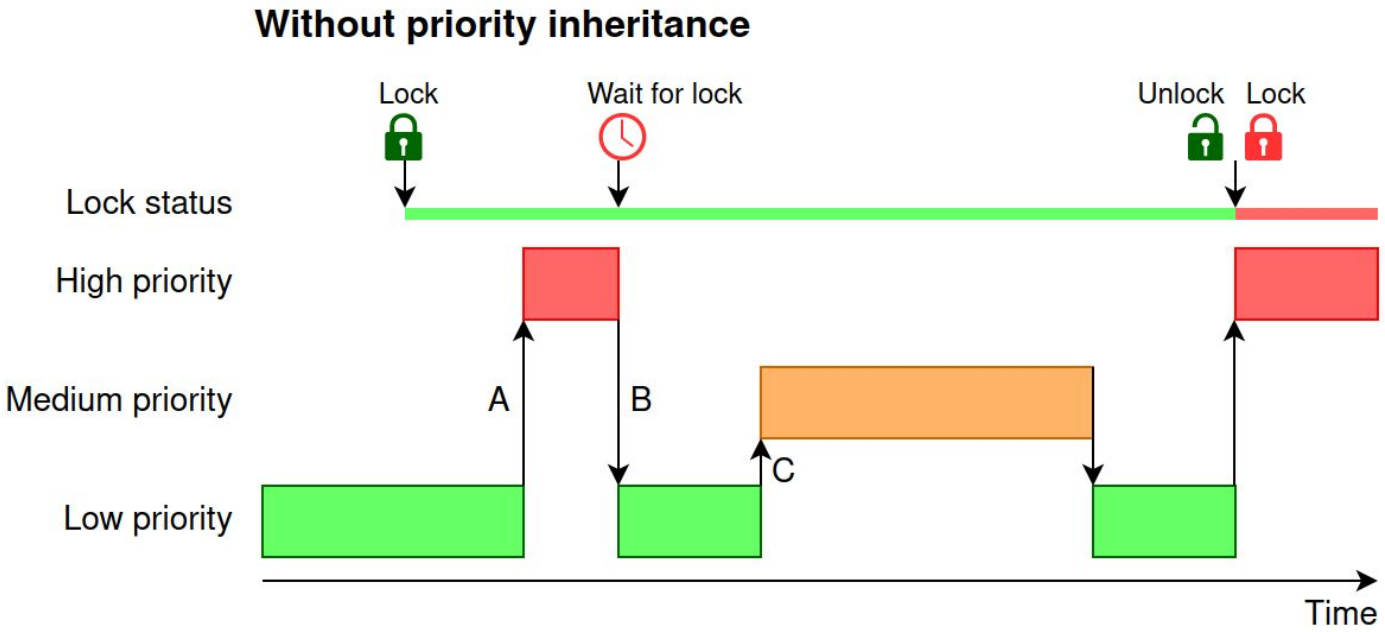
Select a slice: RtThread GetDesiredPosition us

roscon-2023-realtime-workshop/exercise2-1/results/baseline.perfetto



5 000 µs > 1 000 µs

Exercise 2-1: Single Data



Exercise 2-1: Single Data

`exercise2-1/src/inverted_pendulum/include/inverted_pendulum/message_passing/single_data.h`

```
7  struct SingleData {
8  // Remove the lock
9    void Set(const double value) {
10     std::scoped_lock lock(mutex_);
11     value_ = value;
12   }
13
14  // Remove the lock
15   double Get() {
16     std::scoped_lock lock(mutex_);
17     return value_;
18   }
19
20  // Remove the mutex and change to atomic
21  private:
22   std::mutex mutex_;
23   double value_;
24 };
```

Remove the locks

Replace line 23 with:

```
static_assert(std::atomic<double>::is_always_lock_free);
std::atomic<double> value_;
```

Exercise 2-1: Single Data

`exercise2-1/src/inverted_pendulum/include/inverted_pendulum/message_passing/single_data.h`

```
7  struct SingleData {
8  // Remove the lock
9    void Set(const double value) {
10     std::scoped_lock lock(mutex_);
11     value_ = value;
12   }
13
14  // Remove the lock
15   double Get() {
16     std::scoped_lock lock(mutex_);
17     return value_;
18   }
19
20  // Remove the mutex and change to atomic
21  private:
22   std::mutex mutex_;
23   double value_;
24 };
```

Remove the locks

Terminal 1 (docker/shell):

```
$ cd exercise 2-1
$ colcon build
$ /code/stress.sh
```

Terminal 2 (docker/shell):

```
$ cd exercise 2-1
$ ./run.sh
```

- Stop both after around 30 seconds
- Generates exercise2-1.perfetto
- Open file in Perfetto

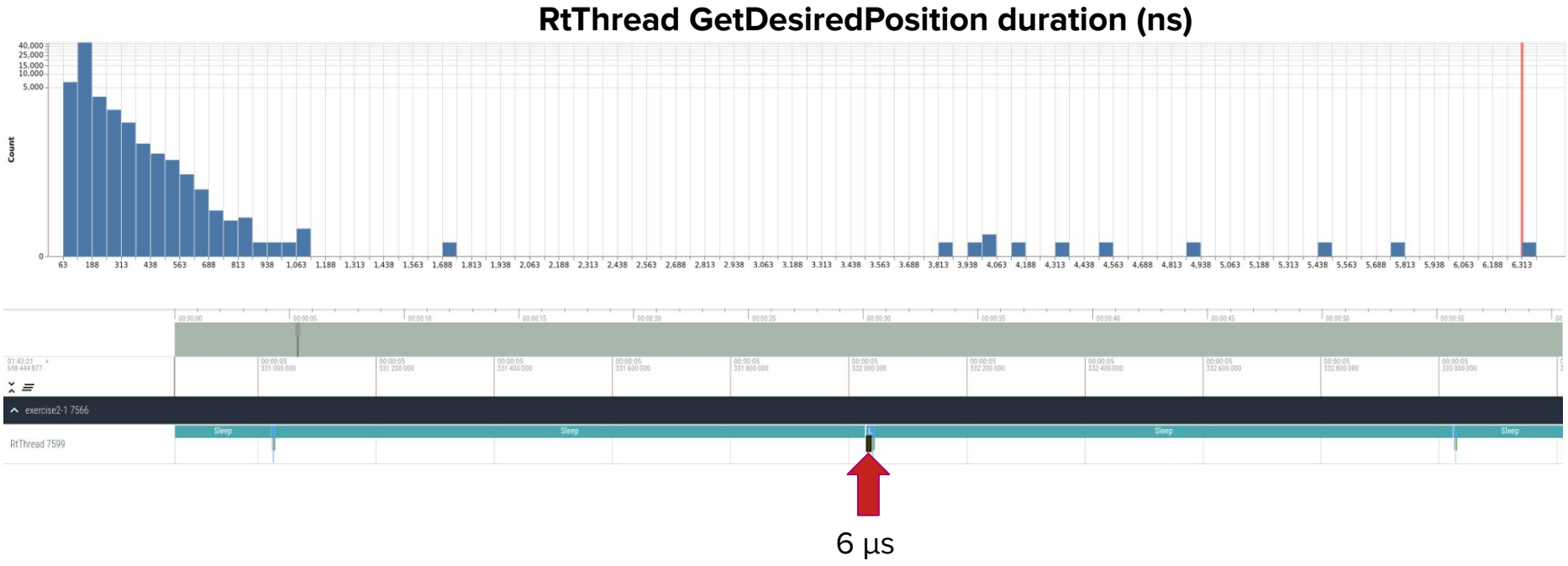
Replace line 23 with:

```
static_assert(std::atomic<double>::is_always_lock_free);
std::atomic<double> value_;
```

Exercise 2-1: Single Data

Select a slice: RtThread GetDesiredPosition ns

roscon-2023-realtime-workshop/exercise2-1/results/solution.perfetto

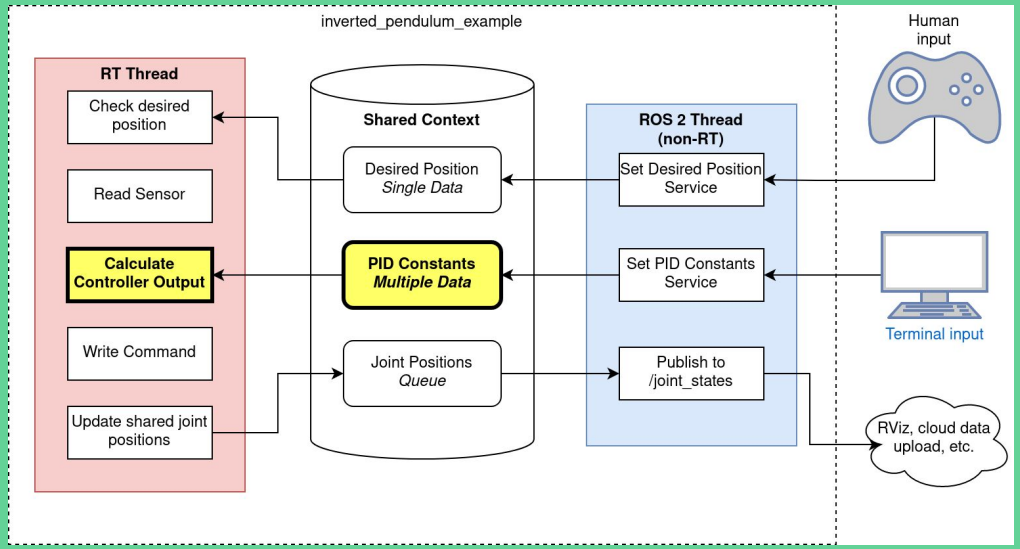


Exercise 2-1: Single Data

	Max duration	**Loop overrun? (> 1000 µs)**
Lock (Baseline)	4997 µs	Yes 👎
Atomic (Solution)	6 µs	No 👍

Exercise 2-2

Multiple Data



Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

- `$ cd /code/exercise2-2`
- We'd like to update the PID constants
  - Three doubles, packaged in a struct
  - Simulate lock contention by repeatedly getting PID constants in a loop
    - Not realistic, but lock contention is rare
    - Real-time reliability needs to be tested over long periods of time

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

```
struct MultipleData {
 void Set(PIDConstants pid_constants) {
  const std::scoped_lock lock(pid_constant_mutex_);
  pid_constants_ = pid_constants;
 }

 PIDConstants Get() {
  const std::scoped_lock lock(pid_constant_mutex_);
  return pid_constants_;
 }

private:
 using mutex = std::mutex;
 mutex pid_constant_mutex_;

 PIDConstants pid_constants_;
};
```

Called by ROS thread when new PID constants are set

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

```
struct MultipleData {
 void Set(PIDConstants pid_constants) {
  const std::scoped_lock lock(pid_constant_mutex_);
  pid_constants_ = pid_constants;
 }

 PIDConstants Get() {
  const std::scoped_lock lock(pid_constant_mutex_);
  return pid_constants_;
 }

private:
 using mutex = std::mutex;
 mutex pid_constant_mutex_;

 PIDConstants pid_constants_;
};
```

Called by ROS thread when new PID constants are set

Called by RT thread every iteration!
Has to complete within 1 ms
Repeatedly called by non-RT thread for latency simulation

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

```
struct MultipleData {
 void Set(PIDConstants pid_constants) {
  const std::scoped_lock lock(pid_constant_mutex_);
  pid_constants_ = pid_constants;
 }

 PIDConstants Get() {
  const std::scoped_lock lock(pid_constant_mutex_);
  return pid_constants_;
 }

private:
 using mutex = std::mutex;
 mutex pid_constant_mutex_;

 PIDConstants pid_constants_;
};
```

Called by ROS thread when new PID constants are set

Lock contention between these

Called by RT thread every iteration!
Has to complete within 1 ms
Repeatedly called by non-RT thread for latency simulation

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

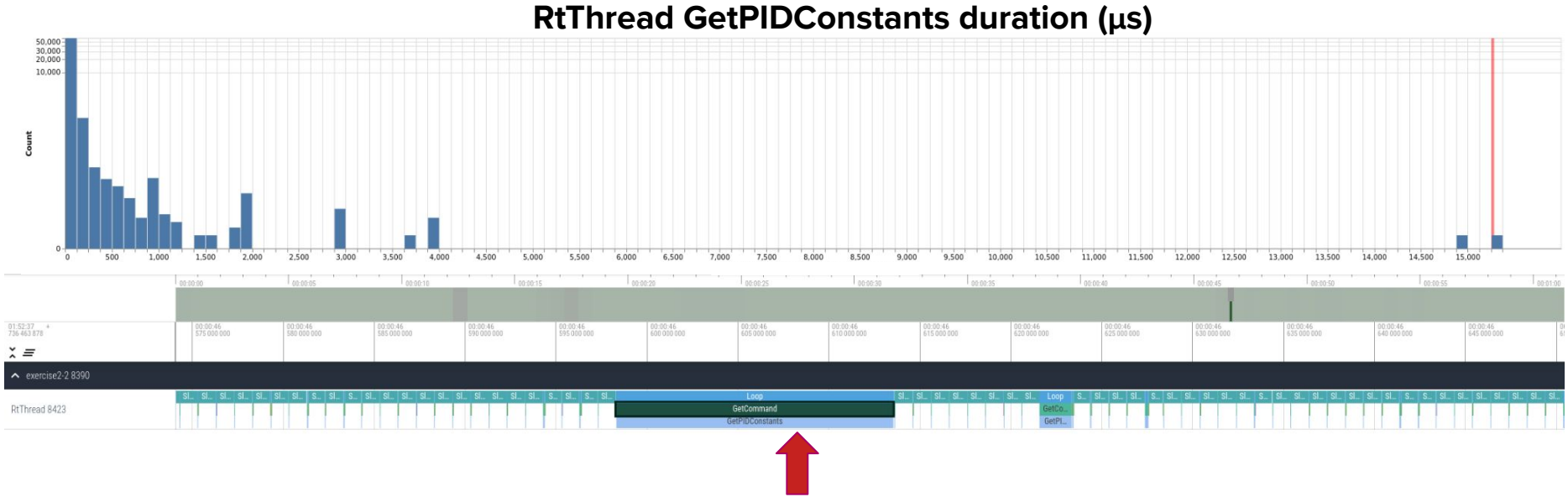
- Build:
  - `$ cd /code/exercise2-2 && colcon build`
- Stress:
  - `$ /code/stress.sh`
- Run the example:
  - `$ ./run.sh`
  - Logs will output if there is a loop overrun
- Stop it after several seconds
  - Generates a file called exercise2-2.perfetto
  - Raspberry Pi users:
    - Connect to the Raspberry Pi via Ethernet
    - Open a browser and go to 192.168.10.1/repo/exercise2-2 to download the file
  - Open this in Perfetto localhost:3100
- Interested in GetPIDConstants slice

Select a slice: RtThread GetPIDConstants us

Exercise 2-2: Multiple Data

roscon-2023-realtime-workshop/exercise2-2/results/baseline.perfetto

Same lock contention problem as exercise 2-1



15 300 µs > 1 000 µs

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

- Atomic vs. lock

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

- Atomic vs. lock
- `static_assert(std::atomic<PIDConstants>::is_always_lock_free);`

  `std::atomic<PIDConstants>`

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

- Atomic vs. lock
- `static_assert(std::atomic<PIDConstants>::is_always_lock_free);`

`std::atomic<PIDConstants>`

Build error!

```
multiple_data.h:14:46: error: static assertion failed
|    static_assert(std::atomic<PIDConstants>::is_always_lock_free);
```

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

```
14 struct MultipleData {
15  void Set(PIDConstants pid_constants) {
16     const std::scoped_lock lock(pid_constant_mutex_);
17     pid_constants_ = pid_constants;
18   }
19
20   PIDConstants Get() {
21     const std::scoped_lock lock(pid_constant_mutex_);
22     return pid_constants_;
23   }
24
25  private:
26   // Change this mutex to a priority inheritance mutex
27   // Swap std::mutex for a cactus_rt::mutex
28   using mutex = std::mutex;
29   mutex pid_constant_mutex_;
30
31   PIDConstants pid_constants_;
32 };
```

Replace line 28 with:

`using mutex = cactus_rt::mutex`

Exercise 2-2: Multiple Data

`exercise2-2/src/inverted_pendulum/include/inverted_pendulum/message_passing/multiple_data.h`

```
14 struct MultipleData {
15  void Set(PIDConstants pid_constants) {
16     const std::scoped_lock lock(pid_constant_mutex_);
17     pid_constants_ = pid_constants;
18   }
19
20   PIDConstants Get() {
21     const std::scoped_lock lock(pid_constant_mutex_);
22     return pid_constants_;
23   }
24
25  private:
26   // Change this mutex to a priority inheritance mutex
27   // Swap std::mutex for a cactus_rt::mutex
28   using mutex = std::mutex;
29   mutex pid_constant_mutex_;
30
31   PIDConstants pid_constants_;
32 };
```

Replace line 28 with:

```
using mutex = cactus_rt::mutex
```

Terminal 1 (docker/shell):

```
$ cd exercise 2-2
$ colcon build
$ /code/stress.sh
```

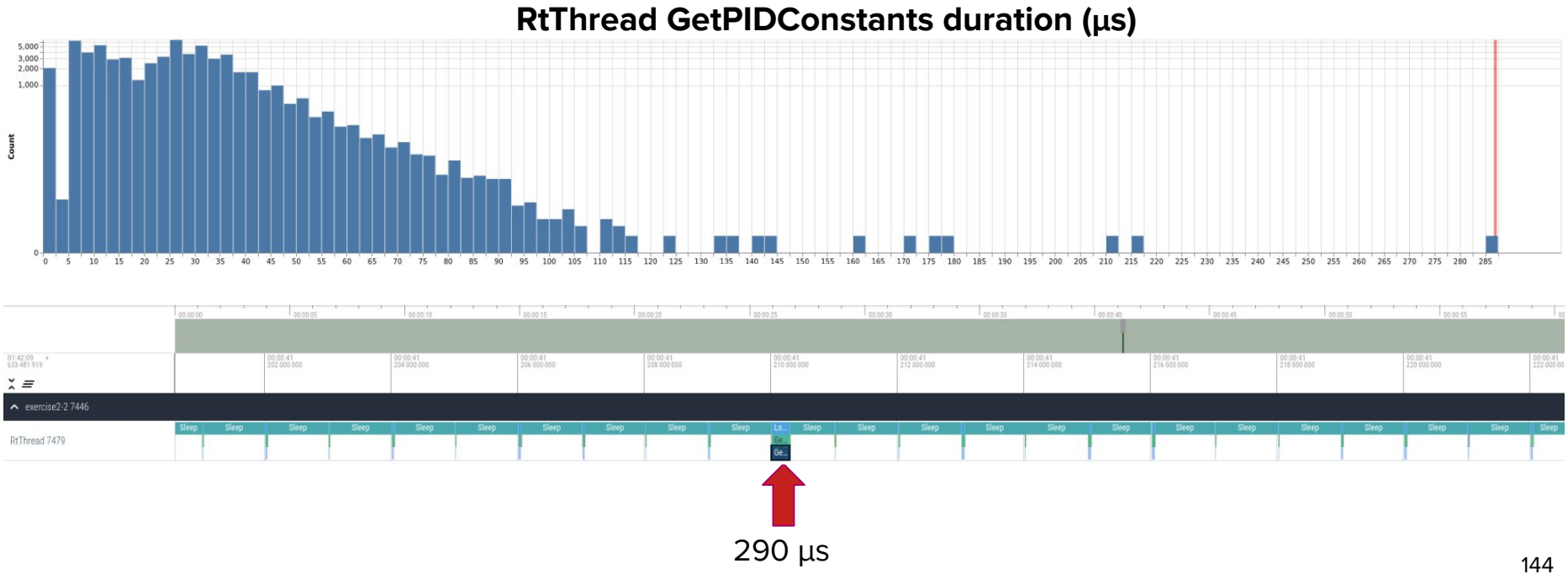
Terminal 2 (docker/shell):

```
$ cd exercise 2-2
$ ./run.sh
```

- Stop both after around 30 seconds
- Generates exercise2-2.perfetto
- Open file in Perfetto

Exercise 2-2: Multiple Data

roscon-2023-realtime-workshop/exercise2-2/results/solution.perfetto



Exercise 2-2: Multiple Data

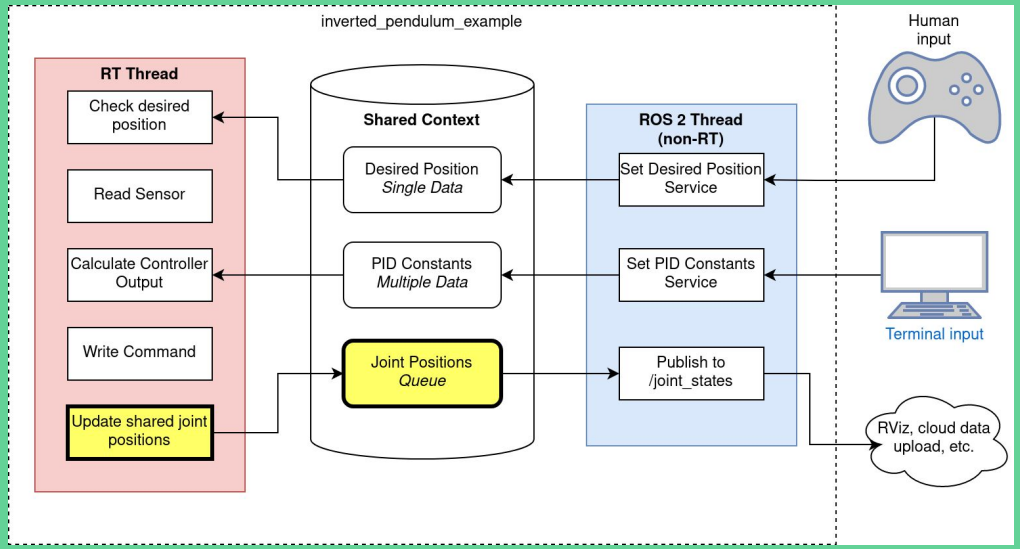
	Max duration	**Loop overrun? (> 1000 µs)**
std::mutex (Baseline)	15 269 µs	Yes 👎
Priority inheritance mutex (Solution)	287 µs	No 👍

Exercise 2-2: Multiple Data

- Other options
  - try_lock - if failure to acquire the shared resource is OK
  - compare and swap exchange loop

Exercise 2-3

Queues



Exercise 2-3: Data Queue

- `$ cd /code/exercise2-3`
- We need to pass the inverted pendulum joint positions from the RT thread to the ROS thread
  - Joint position is a double and a timestamp
  - RT thread will emplace data
  - ROS thread will pop

`exercise2-3/src/inverted_pendulum/include/inverted_pendulum/message_passing/data_queue.h`

```
void RosPendulumNode::TimerCallback() {
  OutputData data;

  // Empty the queue of all data
  while (true) {
    // Check if there is data in the queue
    if (shared_context_->data_queue.PopData(data)) {
      // Construct a joint_state message for the pendulum
      // position

      // ...
```

Exercise 2-3: Data Queue

`exercise2-3/src/inverted_pendulum/include/inverted_pendulum/message_passing/data_queue.h`

- How do we pass the joint states?
  - We need to write this data from the RT thread and read it from the ROS thread
  - Thread-safe queue protected via locks
- What's the issue?
  - The ROS thread (non-RT) is periodically trying to acquire the lock to read new values
  - The RT thread needs to emplace into this queue, so it may need to wait for the lock
    - `std::queue` has no `reserve` method - dynamic memory allocation
  - `WasteTime` - force some lock contention
    - Not realistic - should minimize doing work while holding the lock
    - Treat code in this section of the non-RT thread as real-time

exercise2-3/src/inverted_pendulum/include/inverted_pendulum/message_passing/data_queue.h

```cpp
struct DataQueue {

  bool EmplaceData(struct timespec timestamp,
                   double output_value) noexcept {
    std::scoped_lock lock(mutex_);
    queue_.emplace(timestamp, output_value);
    return true;
  }

  bool PopData(OutputData& data) {
    std::scoped_lock lock(mutex_);
    WasteTime(std::chrono::microseconds(200));
    if (queue_.size() == 0) {
      return false;
    }
    data = queue_.front();
    queue_.pop();
    return true;
  }

  void WasteTime(std::chrono::microseconds duration) {
    // ...
  }

 private:
  std::queue<OutputData> queue_;
  std::mutex           mutex_;
};
```

Simulate latency from other processes

Exercise 2-3: Data Queue

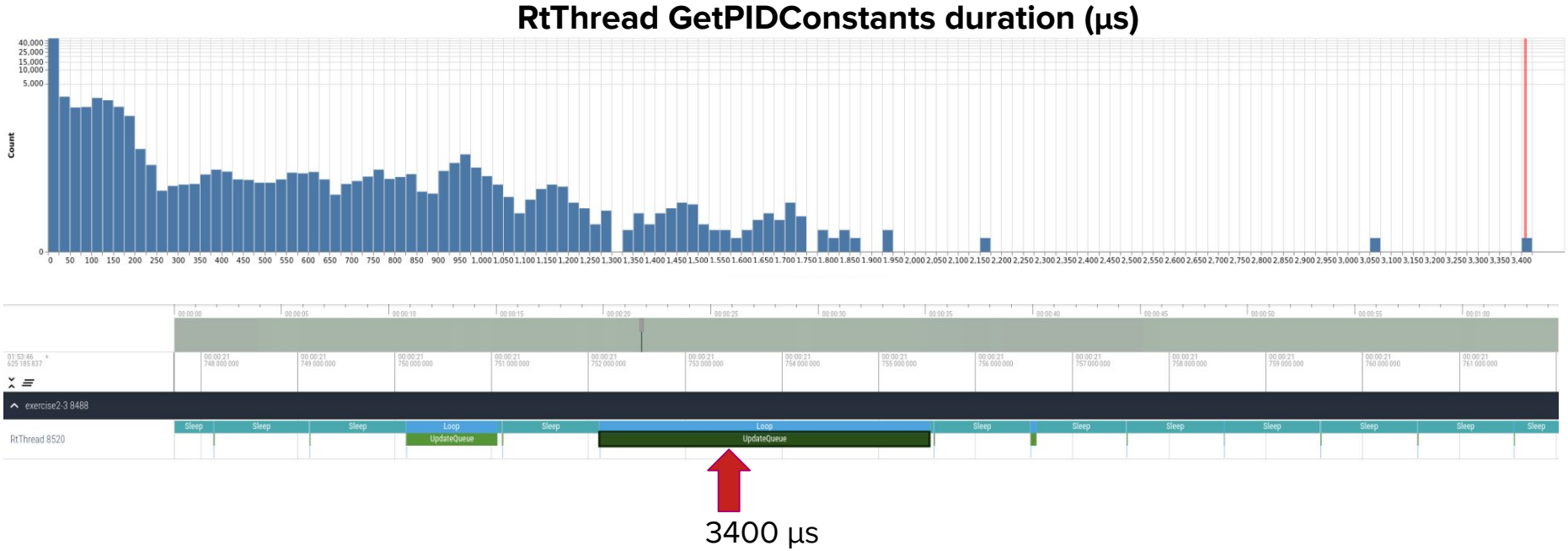
`exercise2-3/src/inverted_pendulum/include/inverted_pendulum/message_passing/data_queue.h`

- Stress:
  - `$ /code/stress.sh`
- Run the example:
  - `$ ./run.sh`
  - Logs will output if there is a loop overrun
- Stop it after several seconds
  - Generates a file called exercise2-3.perfetto
  - Raspberry Pi users:
    - Connect to the Raspberry Pi via Ethernet
    - Open a browser and go to 192.168.10.1/repo/exercise2-3 to download the file
  - Open this in Perfetto
- Interested in the UpdateQueue slice

Exercise 2-3: Data Queue

Select a slice: RtThread UpdateQueue us

roscon-2023-realtime-workshop/exercise2-3/results/baseline.perfetto



Exercise 2-3: Data Queue

`exercise2-3/src/inverted_pendulum/include/inverted_pendulum/message_passing/data_queue.h`

- Use a lockfree queue
- For this exercise, use a queue size of 8192
- We can use this implementation
  - single-producer, single-consumer
  - `try_dequeue` / `try_emplace` guaranteed to never allocate
  - https://github.com/cameron314/readerwriterqueue

```
using moodycamel::ReaderWriterQueue;
```

```
struct DataQueue {

  bool EmplaceData(struct timespec timestamp,
                   double output_value) noexcept {
    std::scoped_lock lock(mutex_);
    queue_.emplace(timestamp, output_value);
    return true;
  }

  bool PopData(OutputData& data) {
    std::scoped_lock lock(mutex_);
    WasteTime(std::chrono::microseconds(200));
    if (queue_.size() == 0) {
      return false;
    }
    data = queue_.front();
    queue_.pop();
    return true;
  }

  void WasteTime(std::chrono::microseconds duration) {
    // ...
  }

 private:
  std::queue<OutputData> queue_;
  std::mutex             mutex_;
};
```

exercise2-3/src/inverted_pendulum/include/inverted_pendulum/message_passing/data_queue.h

Replace line 25 with:

```
return queue_.try_emplace(timestamp, output_value);
```

Replace line 43 with:

```
return queue_.try_dequeue(data);
```

Replace line 43 with:

```
ReaderWriterQueue<OutputData> queue_ =
ReaderWriterQueue<OutputData>(8'192);
```

```
struct DataQueue {

  bool EmplaceData(struct timespec timestamp,
                   double output_value) noexcept {
    std::scoped_lock lock(mutex_);
    queue_.emplace(timestamp, output_value);
    return true;
  }

  bool PopData(OutputData& data) {
    std::scoped_lock lock(mutex_);
    WasteTime(std::chrono::microseconds(200));
    if (queue_.size() == 0) {
      return false;
    }
    data = queue_.front();
    queue_.pop();
    return true;
  }

  void WasteTime(std::chrono::microseconds duration) {
    // ...
  }

 private:
  std::queue<OutputData> queue_;
  std::mutex             mutex_;
};
```

exercise2-3/src/inverted_pendulum/include/inverted_pendulum/message_passing/data_queue.h

Terminal 1 (docker/shell):

```
$ cd exercise 2-3
$ colcon build
$ /code/stress.sh
```

Terminal 2 (docker/shell):

```
$ cd exercise 2-3
$ ./run.sh
```

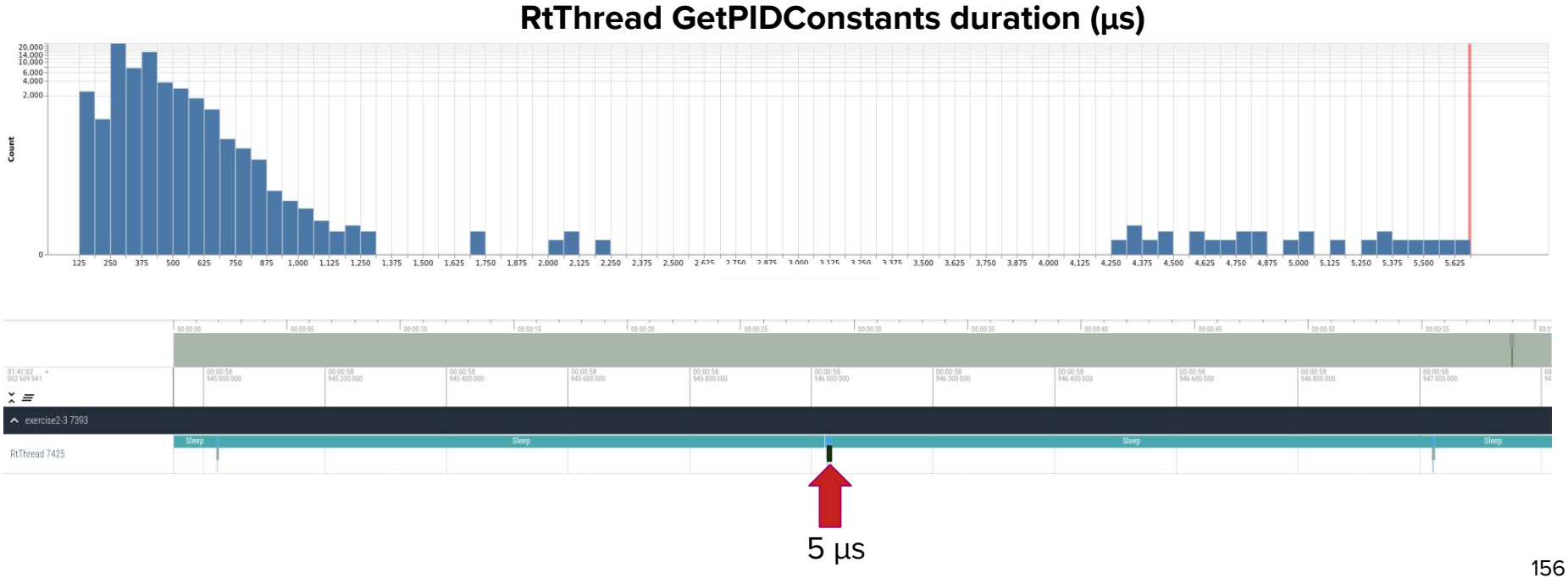
- Stop both after around 30 seconds
- Open file in Perfetto

Exercise 2-3: Data Queue

Select a slice: RtThread UpdateQueue ns

roscon-2023-realtime-workshop/exercise2-3/results/solution.perfetto

Note the unit change!



Exercise 2-3: Data Queue

	Max duration	**Loop overrun? (> 1000 µs)**
std::queue(Baseline)	3410 µs	Yes 👎
Lockfree queue (Solution)	5 µs	No 👍

Summary

- Introduction
  - Inverted pendulum structure and functionality
- Exercise 2-1: Passing small data
  - Constantly change desired position in non-RT thread - this will cause lock contention
    - Trace std::mutex implementation
    - Change to atomic and trace again
- Exercise 2-2: Passing large data
  - Constantly try to get PID constants in non-RT thread - this is to make it faster to find a priority inversion
    - Change std::mutex to atomic and check is_always_lock_free fails
    - Change to PI mutex and trace
- Exercise 2-3: Passing queue
  - Pop from the queue in the non-RT thread and simulate doing some work while holding the lock
    - Change to lockless queue and trace

Morning summary

- Defined “real-time”
- Learned about hardware, OS, and application latency
- Understood the importance of and experimented with the real-time scheduler
- Learned about C++ programming techniques to minimize application latency
- Witnessed and fixed priority inversion problem via a priority-inheritance mutex

Panel discussion

Panel discussion: ROS2 and real-time in production

Agenda

Morning

- 8:00 - 8:45: Latency, hardware, and OS
- 8:45 - 9:15: Hands-on exercise 1
- 9:15 - 10:15: RT C++ programming
- 10:15 - 10:30: Hands-on exercise 2
- 10:30 - 11:00: Break
- 11:00 - 11:30: Hands-on exercise 2
- 11:30 - 12:00: Panel discussion

Afternoon

- 13:00 - 14:00: ROS 2 execution management
- 14:00 - 14:45: Hands-on exercise 3
- 14:45 - 15:30: Real-time with mainline ROS 2
- 15:30 - 16:00: Break
- 16:00 - 16:45: Hands-on exercise 4
- 16:45 - 17:00: Summary

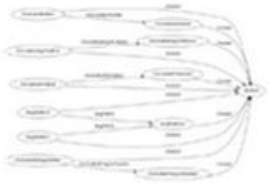
ROS 2 Execution Management

Outline: ROS 2 Execution Management

- Execution Management in ROS 2
- Impact on real-time behavior and determinism
- More on Executors and alternatives
- Experimental real-time Executor

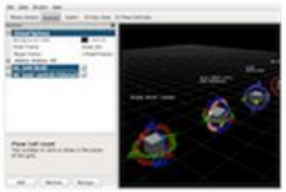
The ROS 2 Equation

=



Plumbing

+



Tools

+

Capabilities

+

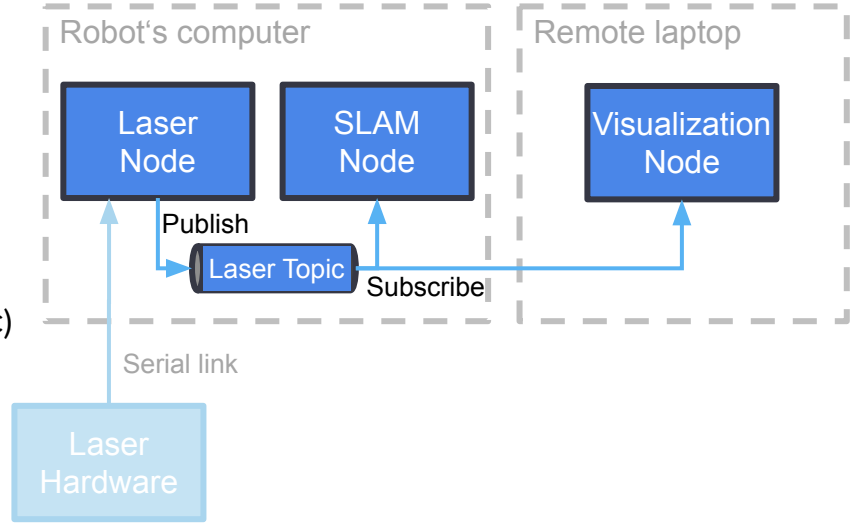
Ecosystem

The ROS 2 Equation

- **Process management**
- **Communication**
- Device drivers
- Data models
- Language independence

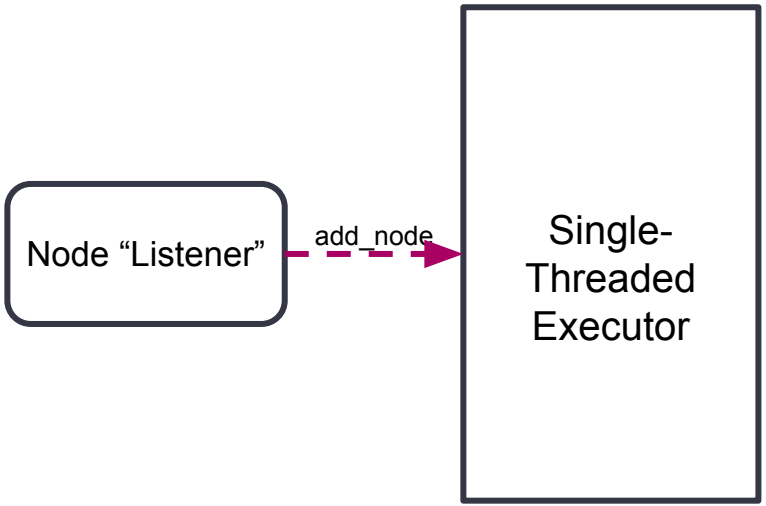
Nodes and Communication

- Basic entity: *Nodes* that exchange messages
- Can be distributed across machines
- Standard communication patterns
  - *Topics*: Publish-Subscribe(1 – n, uni-directional, async)
  - *Services*: Request-Response (n – 1, bi-directional, sync)
  - *Actions:* Advanced Request-Response (1-1, multi-state)
- Nodes comprised of *callables* (functions), which are *data-* or *time-triggered*
  - Implemented in C++, Python, ...
  - Run-to-completion



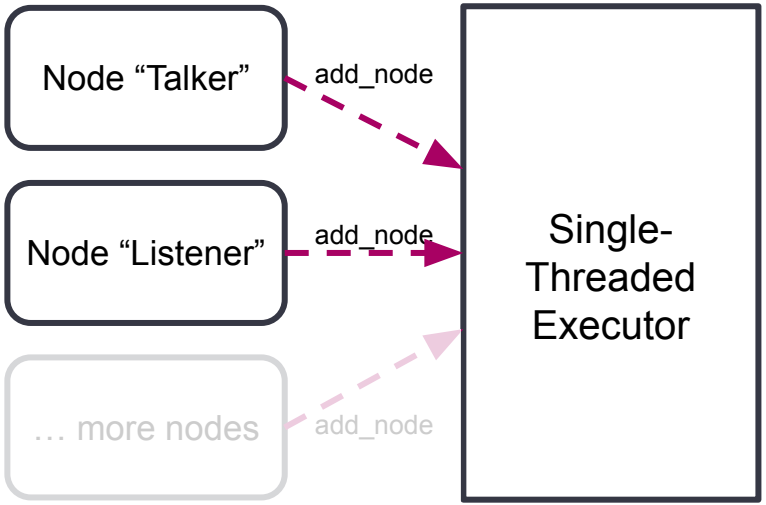
https://index.ros.org/doc/ros2/Concepts/

Example with implicitly declared Executor



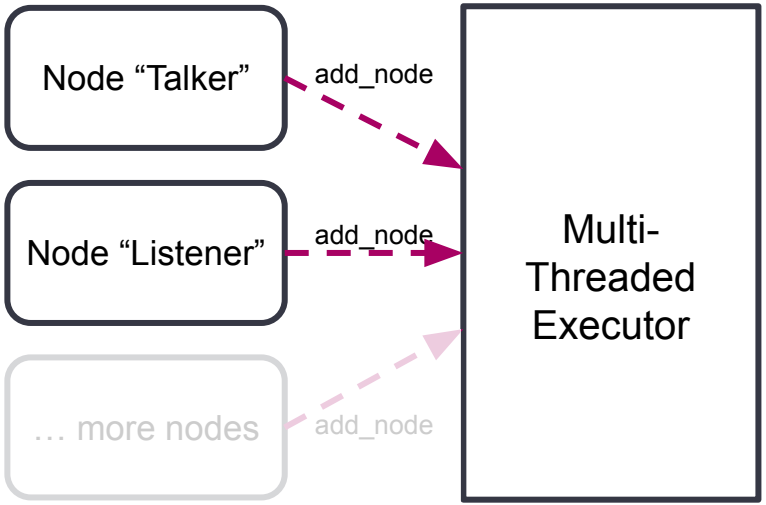

```
int main(int argc, char * argv[]) {
  rclcpp::init(argc, argv);

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

  rclcpp::spin(node);

  rclcpp::shutdown();
}
```

Example with 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();
}
```

Example with 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();
}
```

ROS 2 Layered Architecture

rcl* – language-specific ROS client libraries

rcl – C library

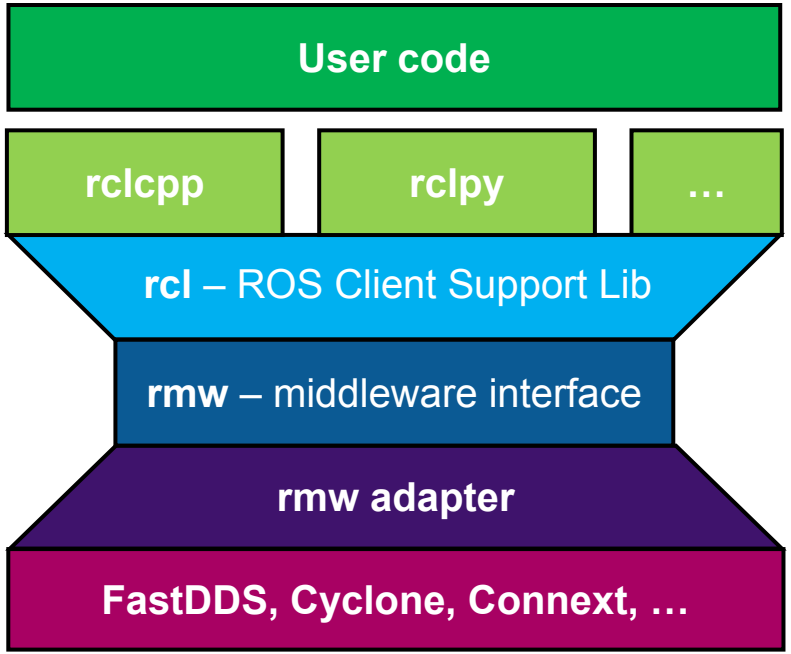
➔ Ensures same core algorithms in all language-specific client libraries

rmw – ROS middleware interface

➔ Hide specifics of DDS implementations

➔ Streamline QoS configuration

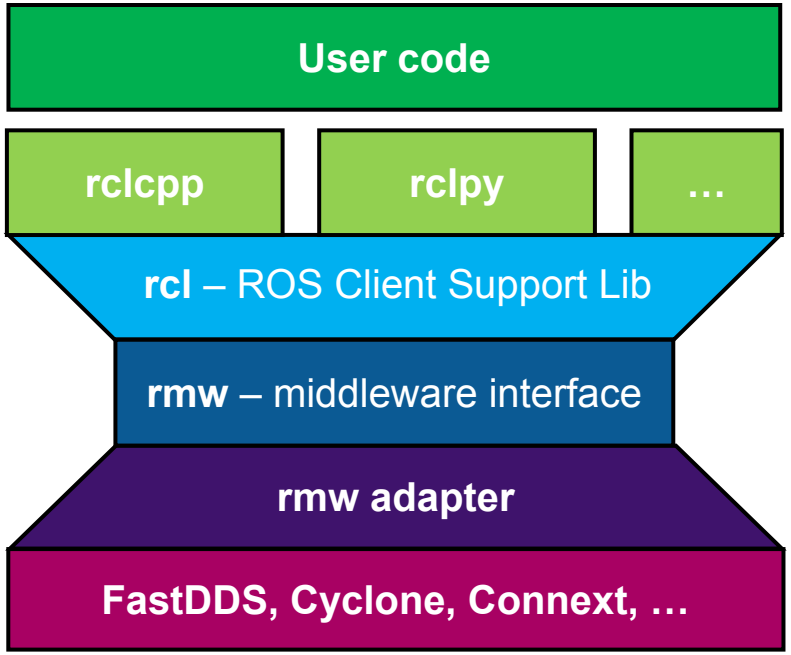
rmw_* – DDS adapters



Execution Management

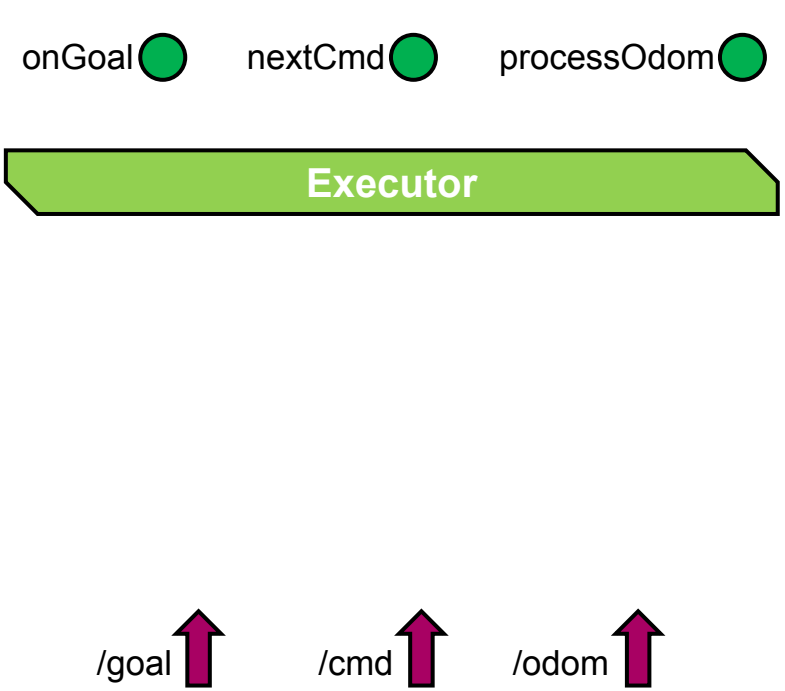
Executor

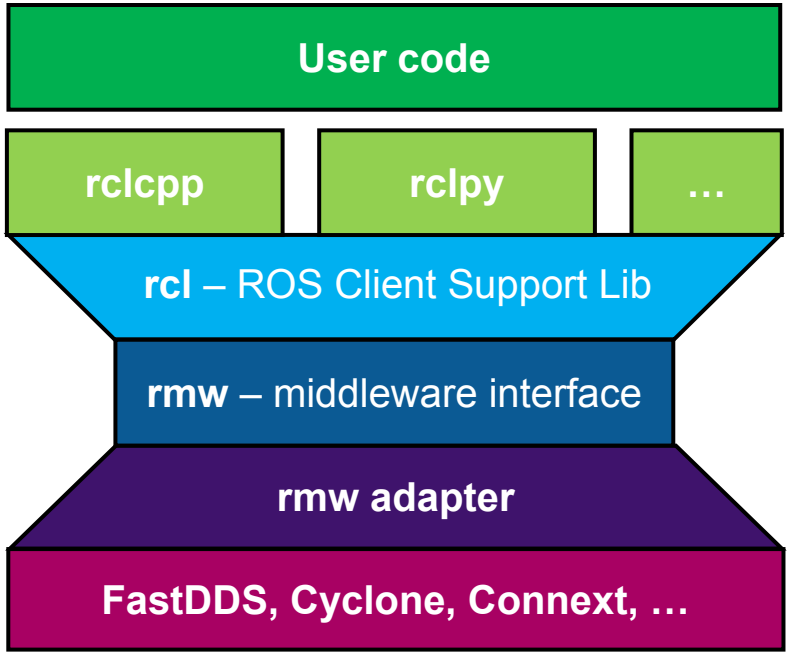
- Invokes callbacks of subscriptions, timers, services, ... on incoming messages and events
- Uses one or multiple threads



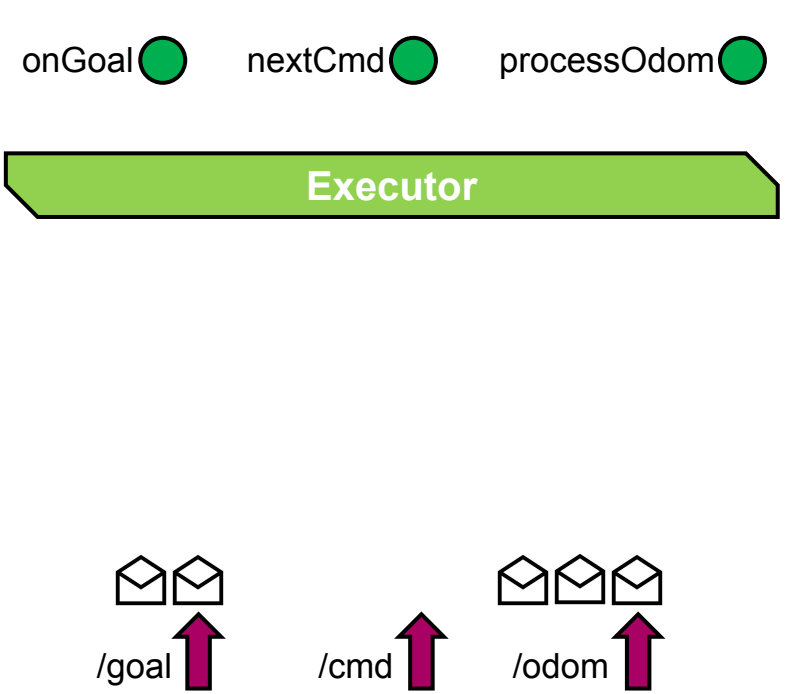
https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Executors.html

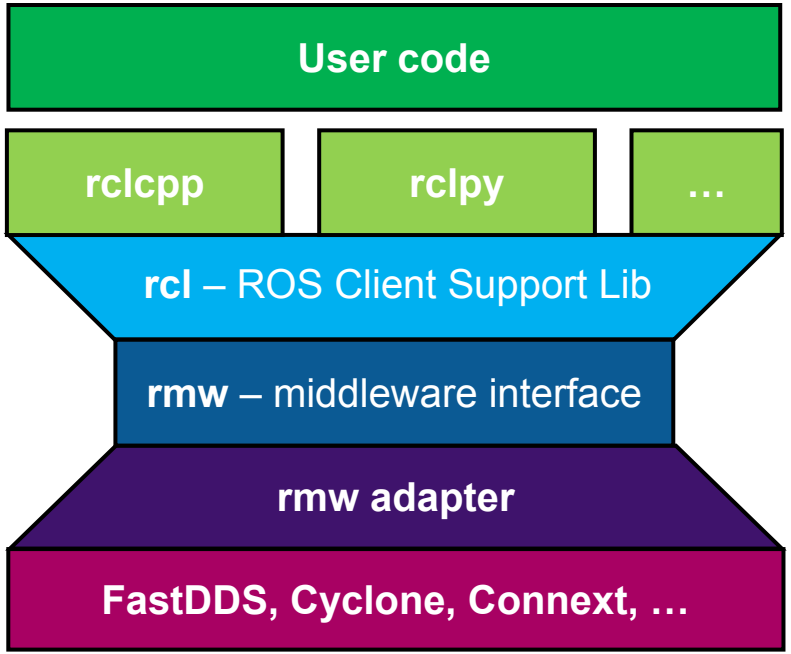
Execution Management



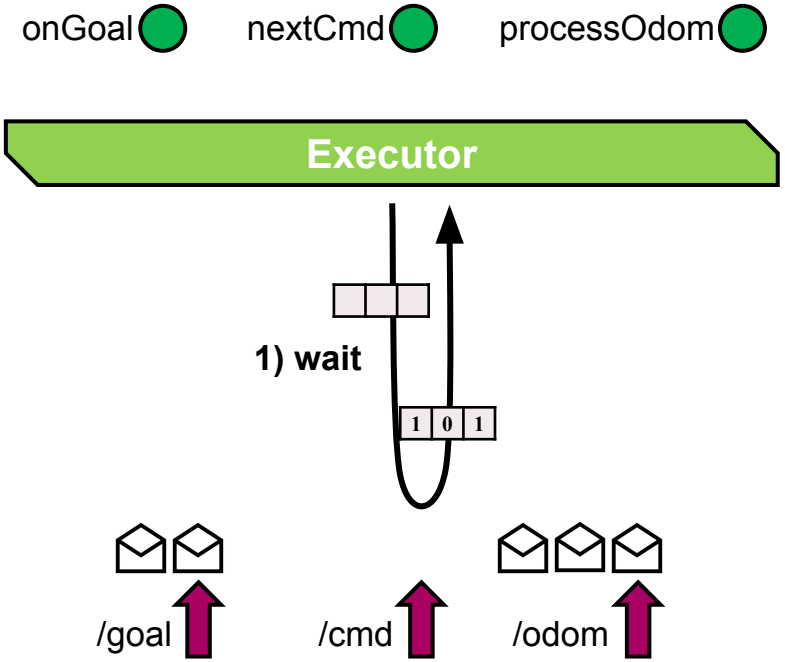


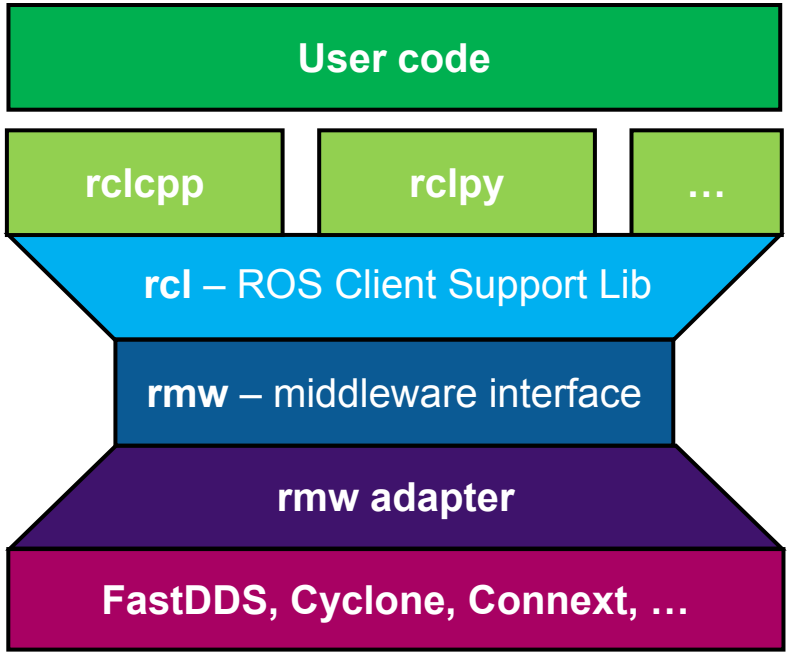
Execution Management



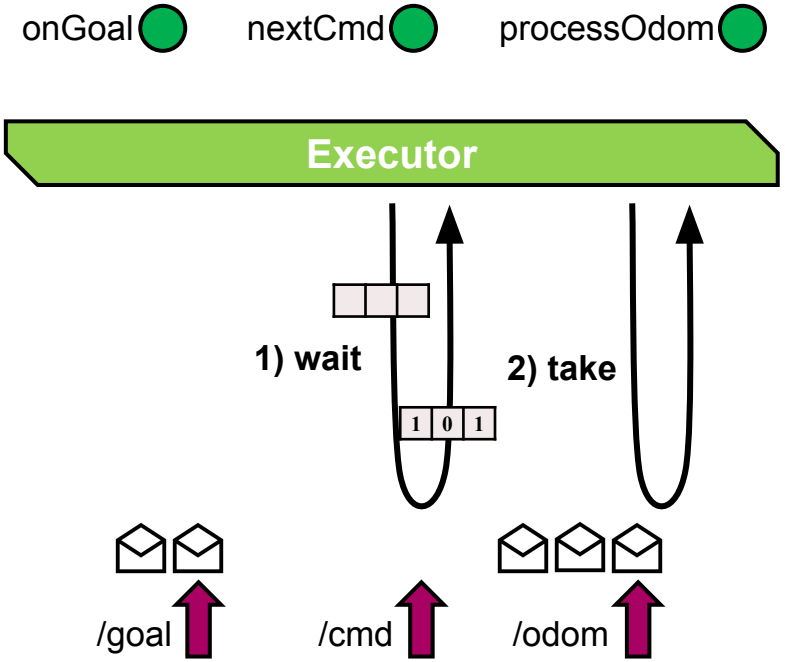


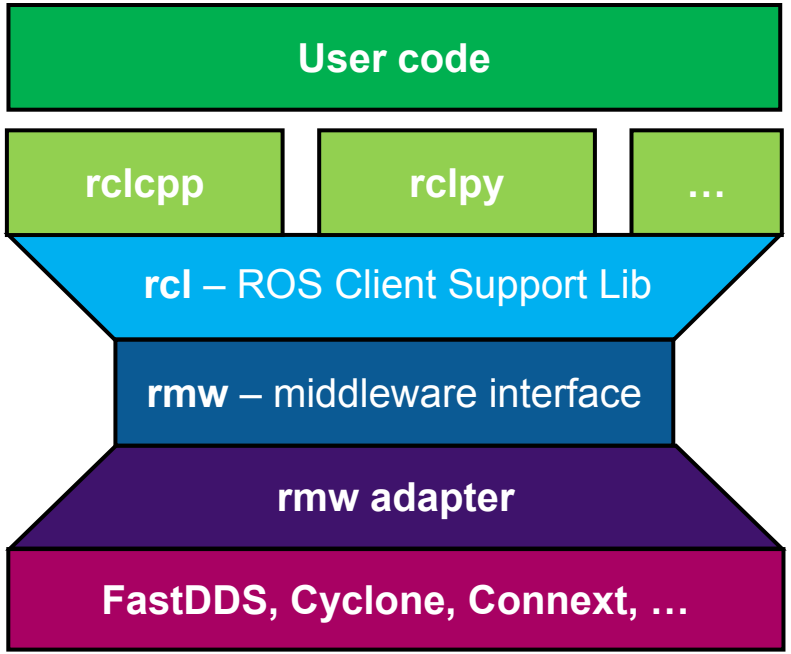
Execution Management



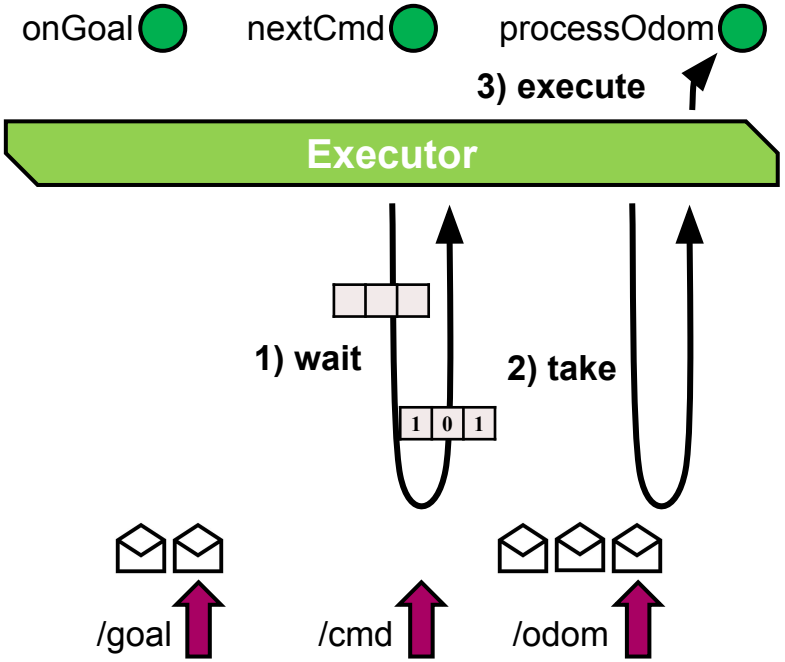


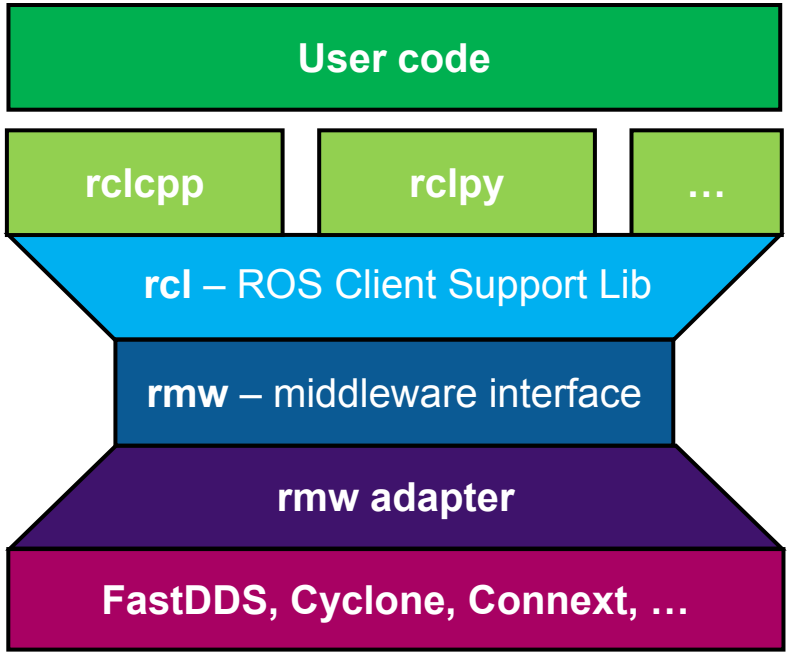
Execution Management



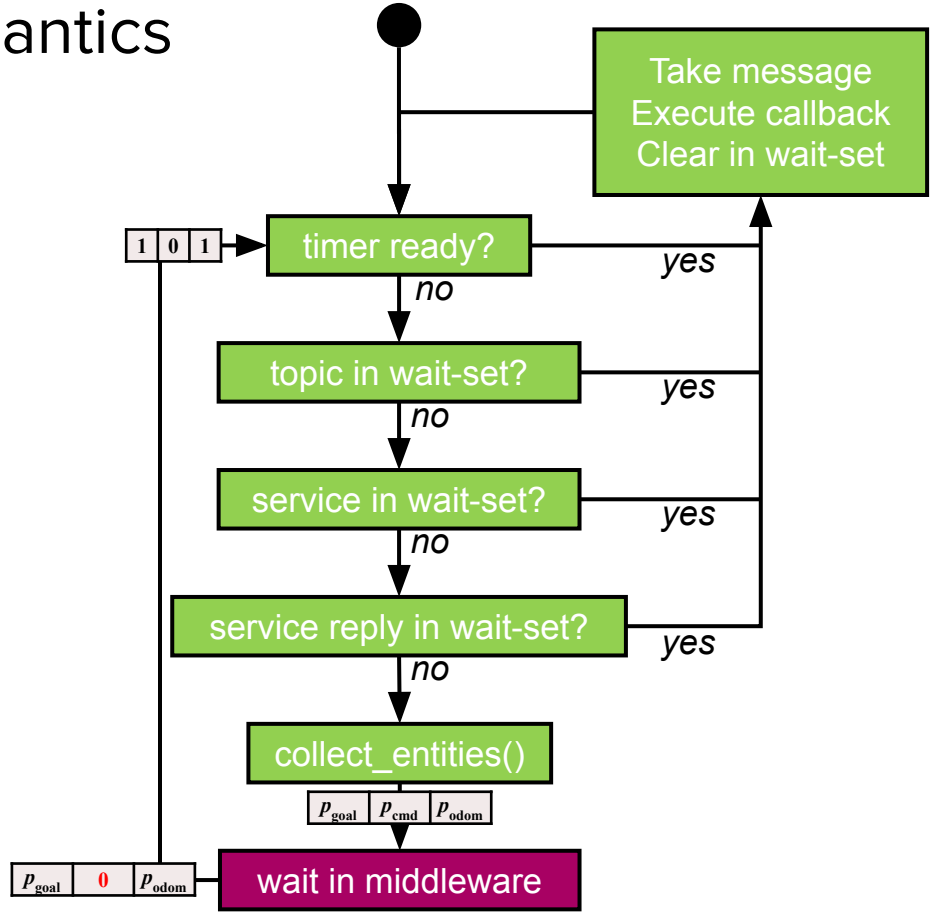


Execution Management

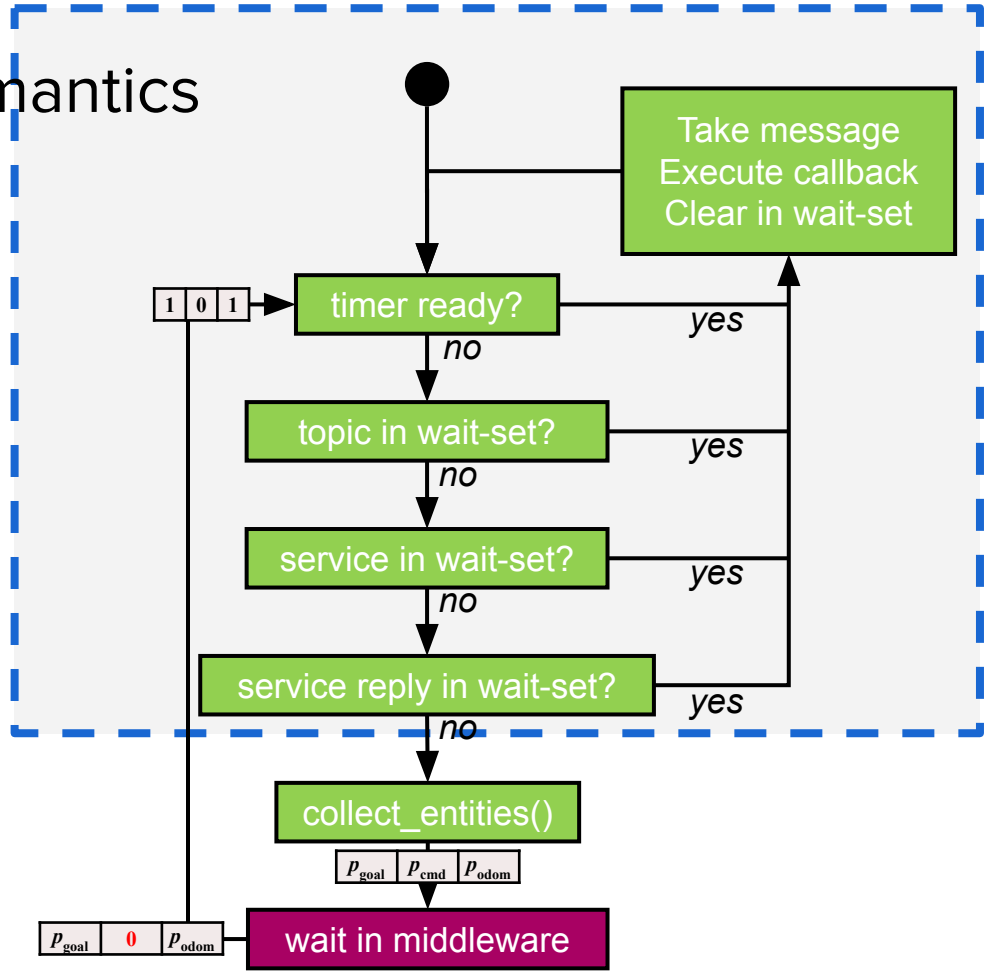




Executor Semantics

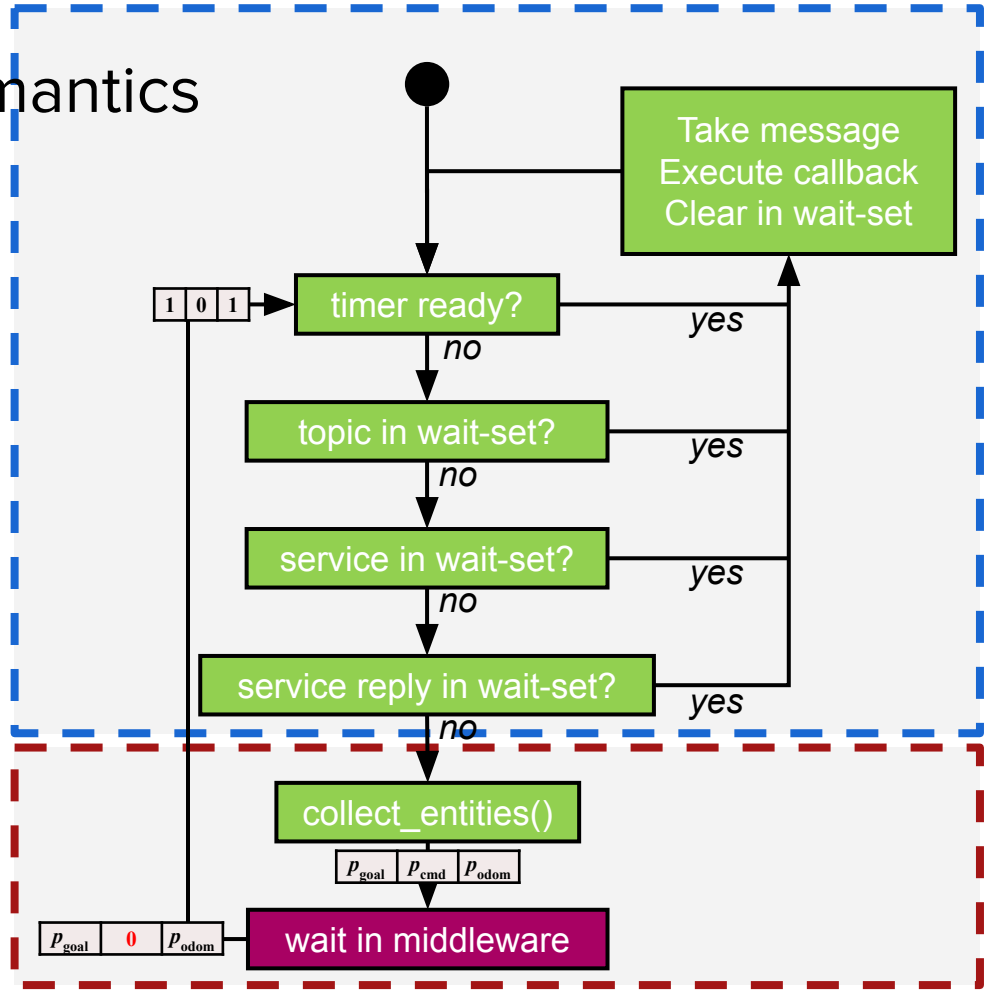


Executor Semantics



Processing Window

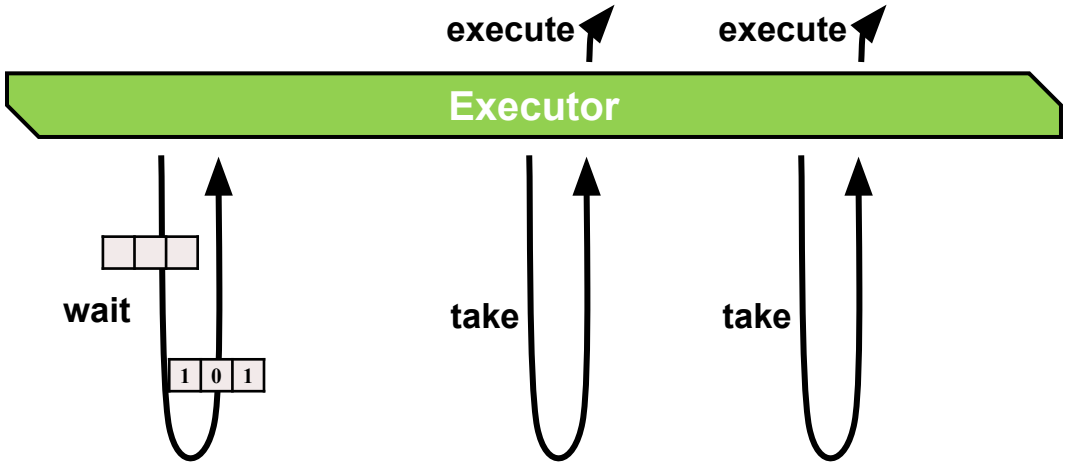
Executor Semantics



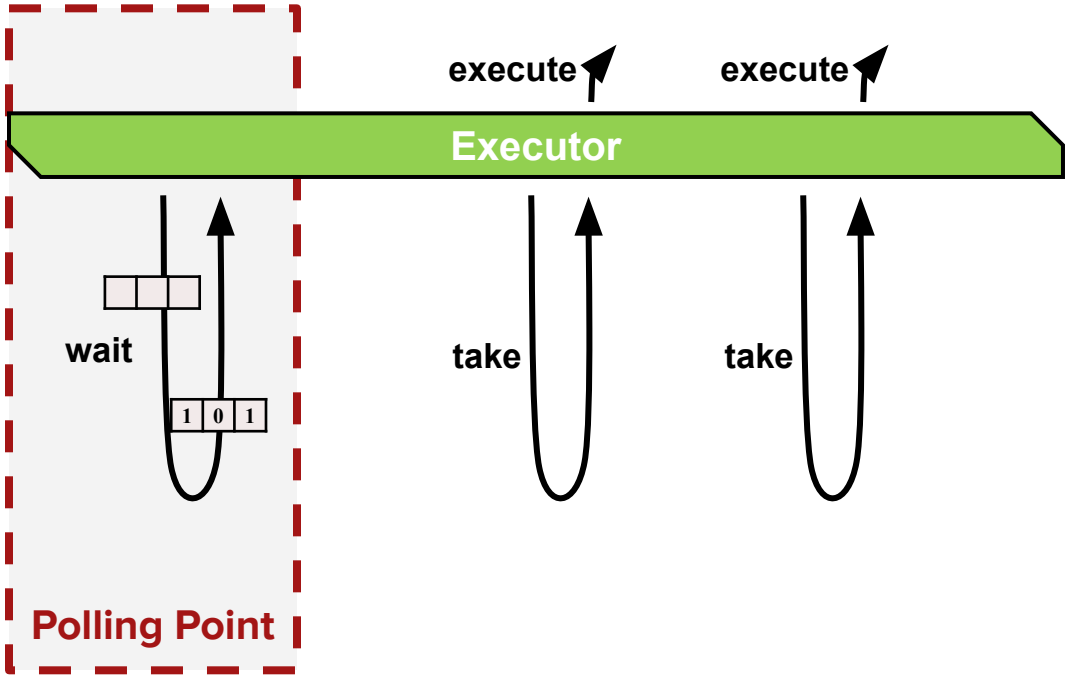
Processing Window

Polling Point

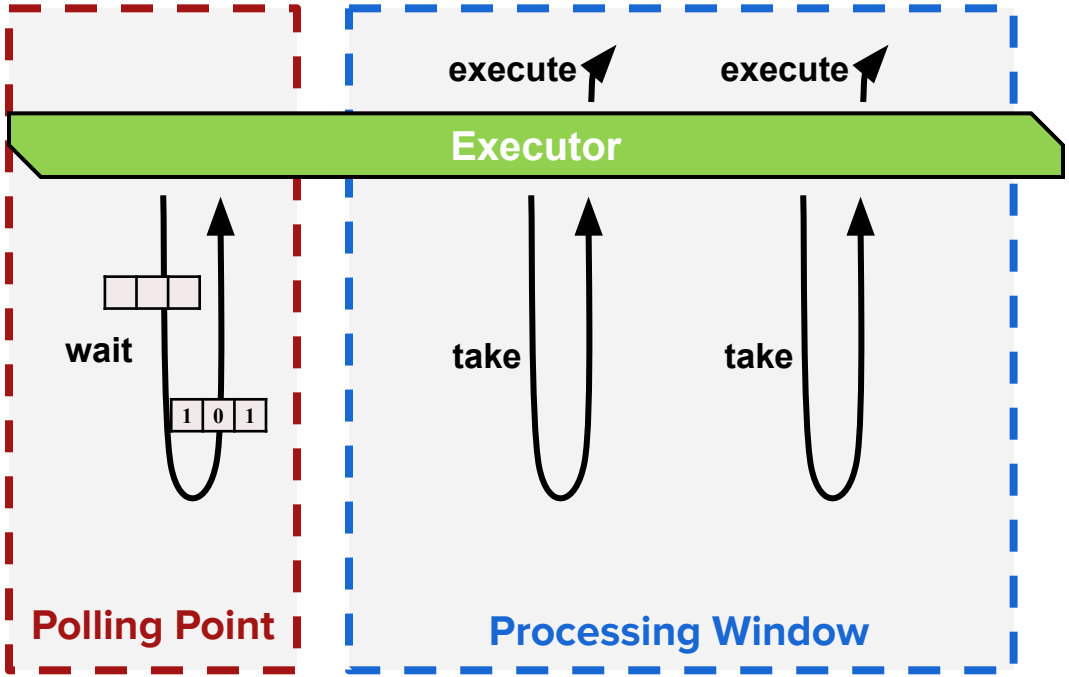
Executor Semantics in two steps



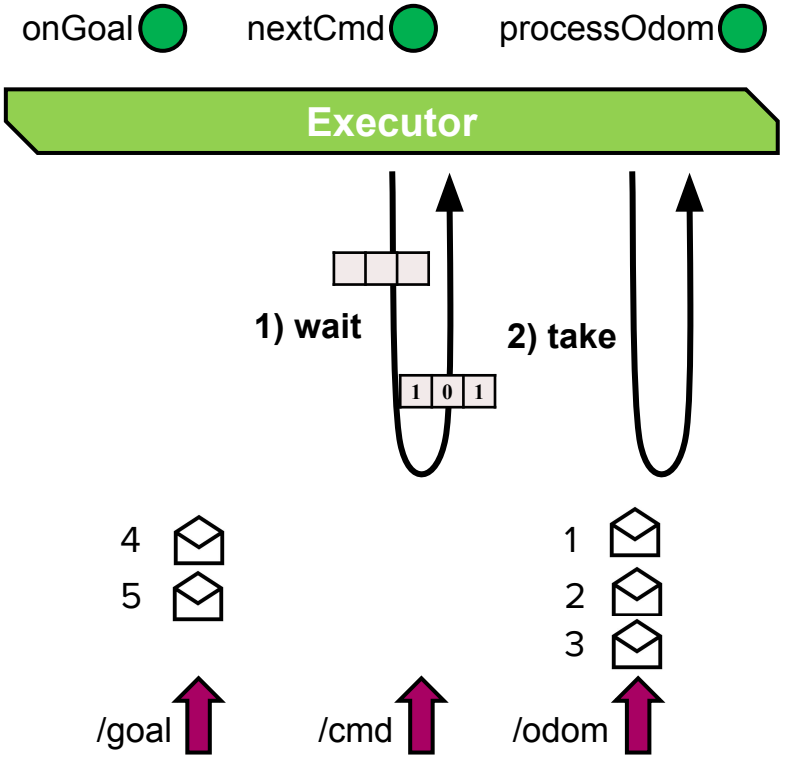
Executor Semantics in two steps



Executor Semantics in two steps

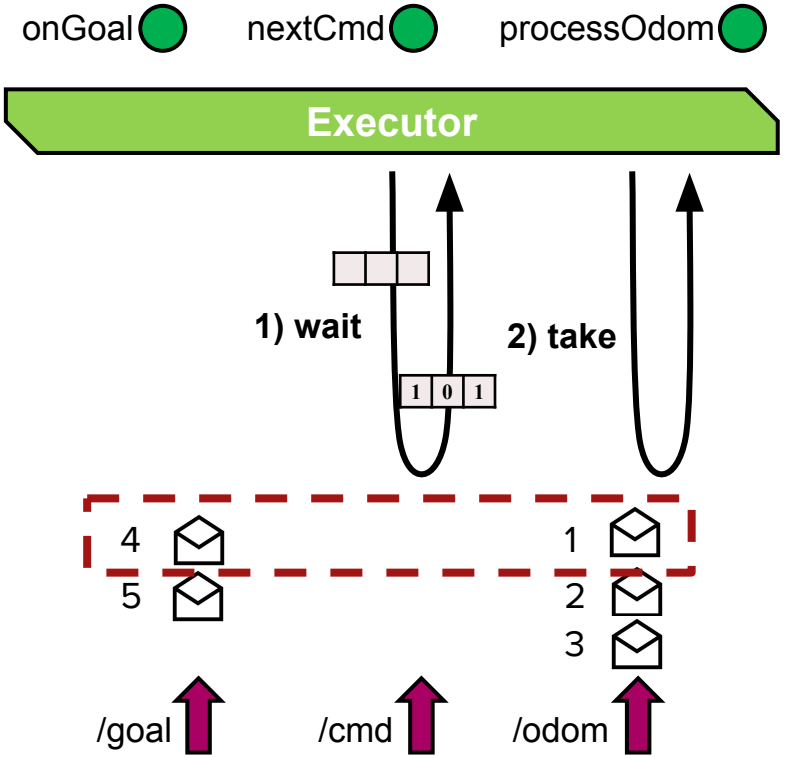


Impact on determinism: non-FIFO message processing



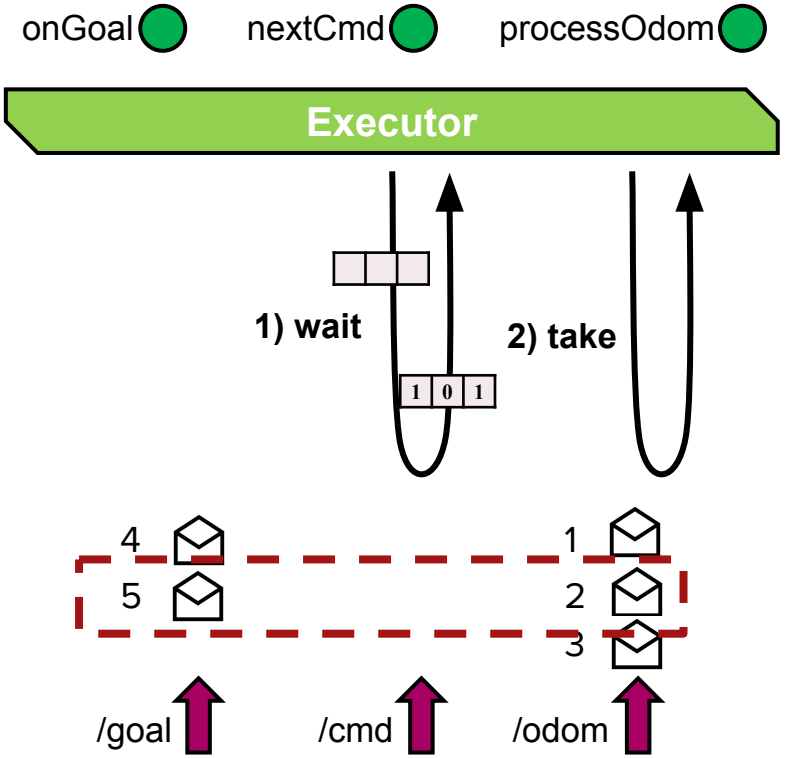
- `/odom` received earlier than `/goal` topics before next polling point

Impact on determinism: non-FIFO message processing



- `/odom` (1) and `/goal`(4) processed first
- Take always “top element” of DDS queue

Impact on determinism: non-FIFO message processing

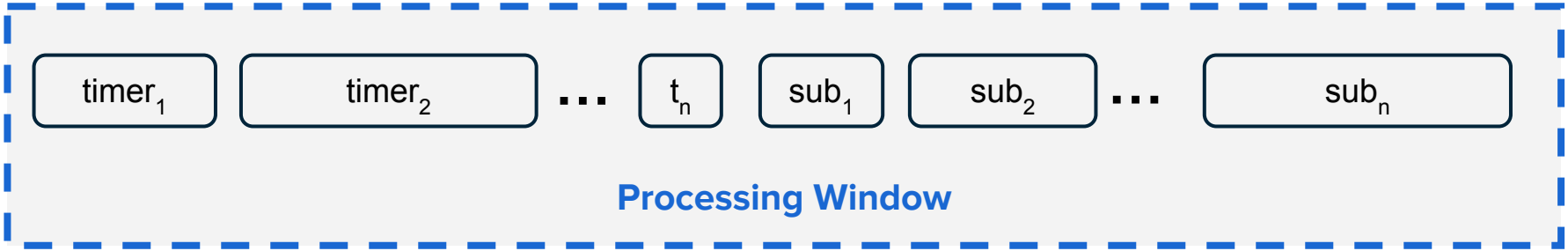


- `/odom` (1) and `/goal`(4) processed first
- Take always “top element” of DDS queue

Not FIFO!

Impact on latency: processing order

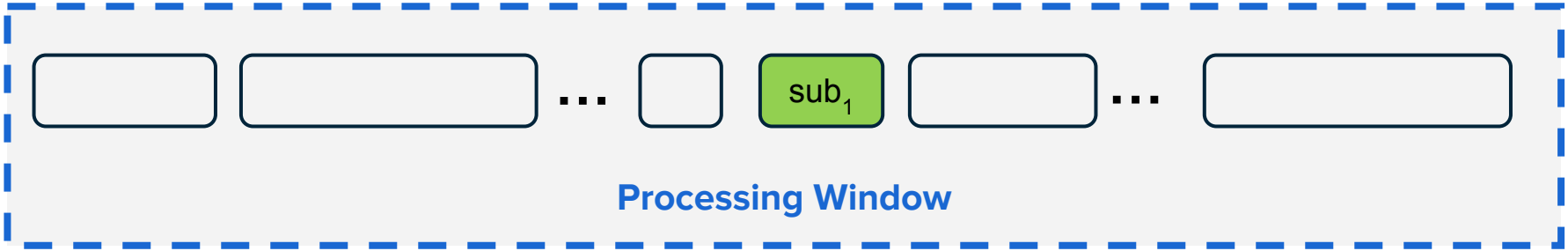
Lets look at different scenarios...



Processing order depends ONLY on the order in which the elements were created/added to the Executor.

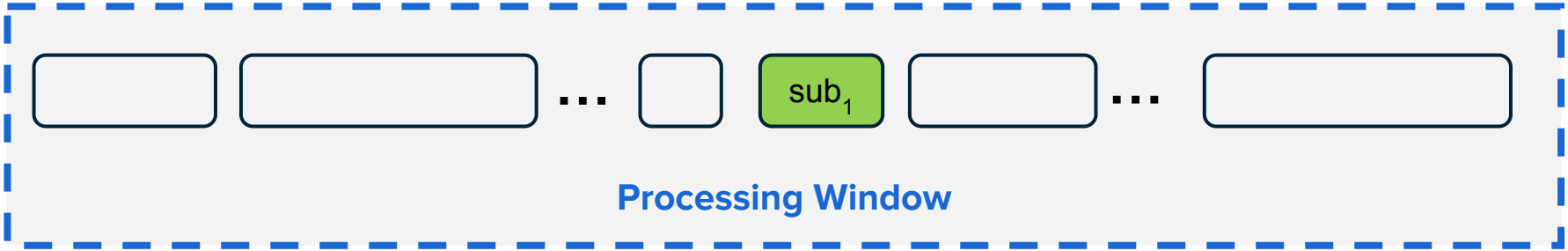
Impact on latency: processing order

One subscription, no timers?



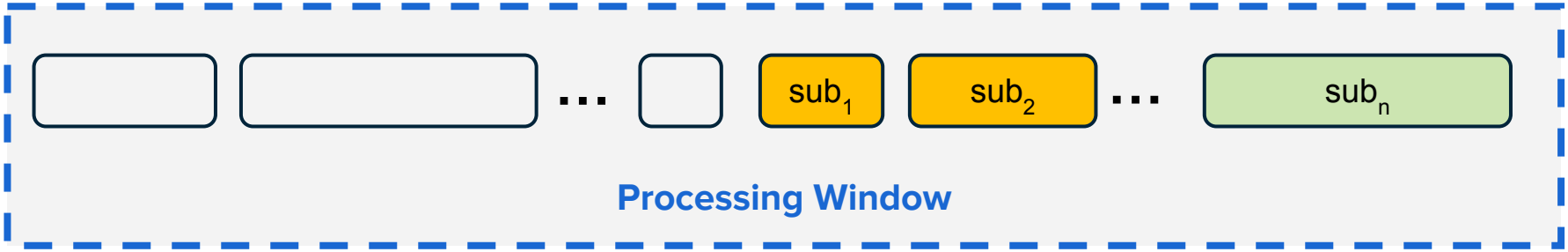
Impact on latency: processing order

One subscription, no timers?



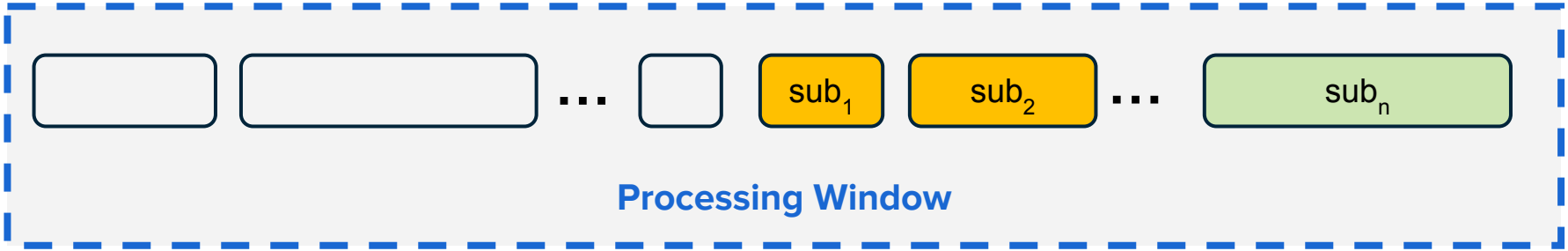
Impact on latency: processing order

No timer, multiple other subscriptions precede?



Impact on latency: processing order

Multiple subscriptions also have new data?

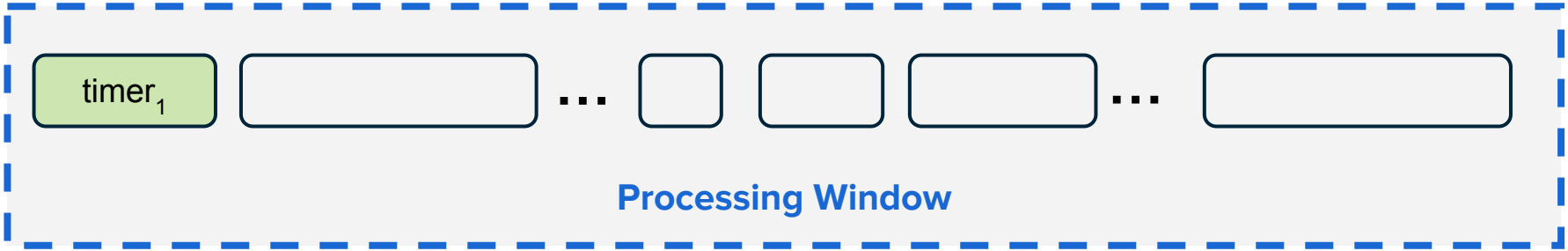


=> see Hands-on-Session 3-1

Worst case: all other subscriptions have received new data and are processed before => **longer latency**!

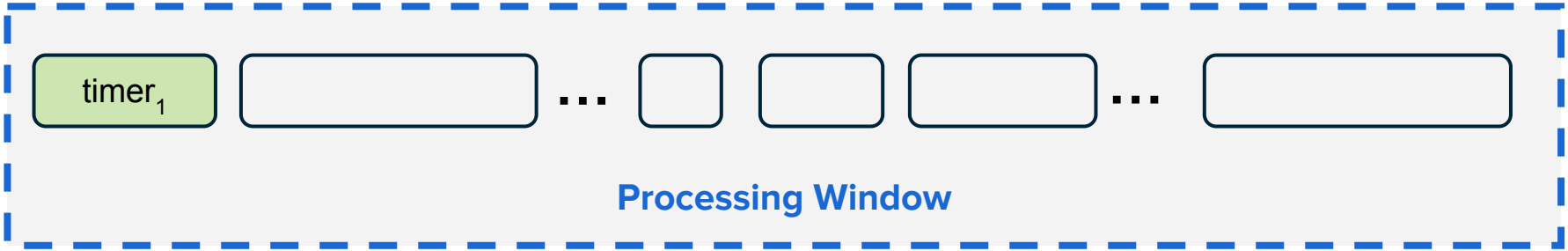
Impact on latency: processing order

Timer at pool position, no subscriptions?



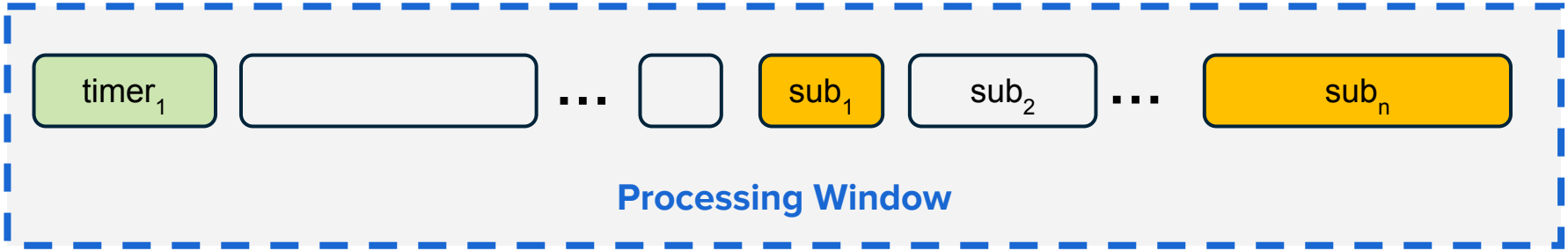
Impact on latency: processing order

Timer at pool position, no subscriptions?



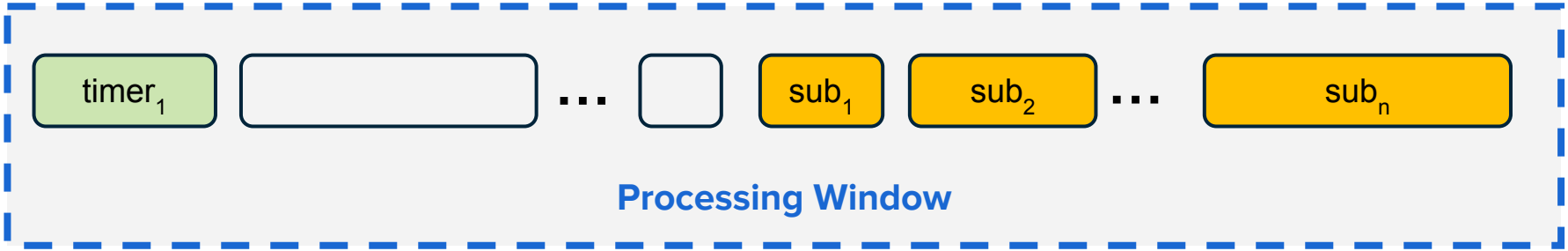
Impact on latency: processing order

Timer at pool position but some subscriptions later on?



Impact on latency: processing order

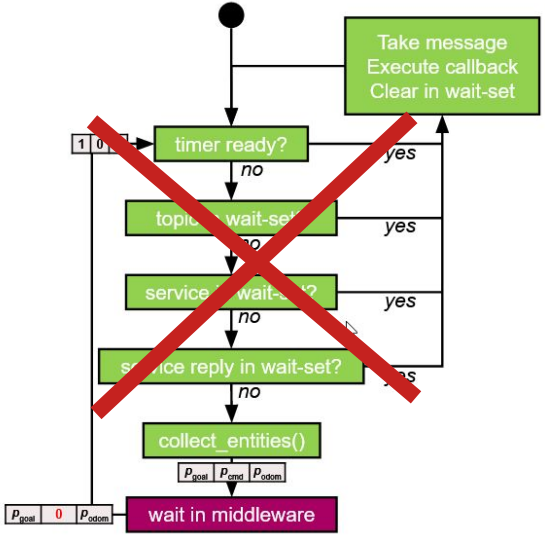
Timer at pool position but some subscriptions later on?



=> see Hands-on-Session 4-1

If next timer elapses during the processing window =>
Not considered until next polling point. => **next timer callback delayed**!

Make your own Executor with rclcpp::waitset




```cpp
rclcpp::WaitSet wait_set({{{sub1}, {sub2}, {...}}}, {timer1});

while (rclcpp::ok())) {
  const auto wait_result = wait_set.wait(2s);

  if (wait_result.kind() == rclcpp::WaitResultKind::Ready) {

    if (wait_result.get_wait_set().get_rcl_wait_set().timers[0U]) {
      // call timer callback
    } else if (wait_result.get_wait_set()
                          .get_rcl_wait_set()
                          .subscriptions[0U]) {
      if (sub1->take(msg, msg_info)) {
        // process message
      }
    }
  }
}
```

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

More on Executors

- MultiThreadedExecutor
  - Parallelizes processing on multi-core platforms
  - Threads have the same priority
  - Same timing issues as SingleThreadedExecutor (based on wait set approach)

More on Executors

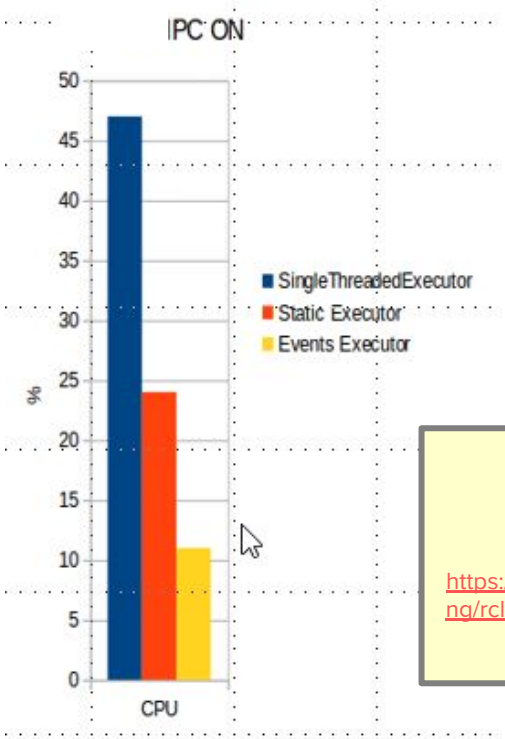
- MultiThreadedExecutor
  - Parallelizes processing on multi-core platforms
  - Threads have the same priority
  - Same timing issues as SingleThreadedExecutor (based on wait set approach)
- StaticSingleThreadedExecutor
  - Optimizes runtime costs for scanning a node in terms of subscriptions, timers, services ...
  - Performs only one scan when node is added (other executors scan regularly)
  - Use only with nodes that create all subscriptions, timers, etc. during initialization

More on Executors

- MultiThreadedExecutor
  - Parallelizes processing on multi-core platforms
  - Threads have the same priority
  - Same timing issues as SingleThreadedExecutor (based on wait set approach)
- StaticSingleThreadedExecutor
  - Optimizes runtime costs for scanning a node in terms of subscriptions, timers, services ...
  - Performs only one scan when node is added (other executors scan regularly)
  - Use only with nodes that create all subscriptions, timers, etc. during initialization
- rclc-Executor
  - C-API designed for micro-controllers (micro-ROS)
  - No dynamic memory allocation at runtime
  - User defined processing order
  - Further deterministic features (like trigger conditions for synchronization)

Events Executor

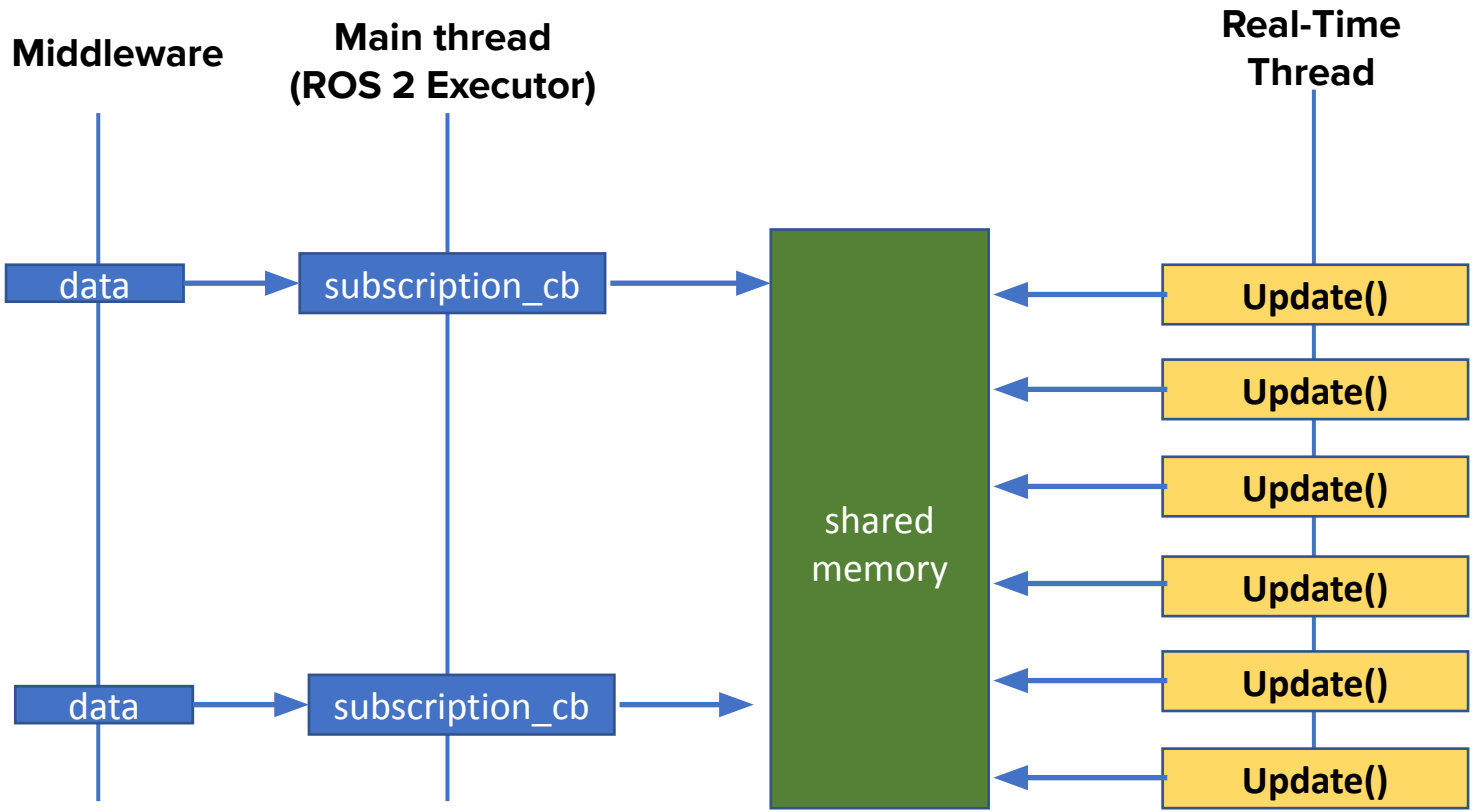
- Improved performance
  - No creation of wait_set
  - RMW Listener API
- FIFO processing
  - Events queue
- Timers manager
  - Independent of DDS
- Executor Entities Collector
  - Like StaticSingleThreadedExecutor



Merged in Rolling (04/2023)

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

ros2_control framework



Comparison Executors

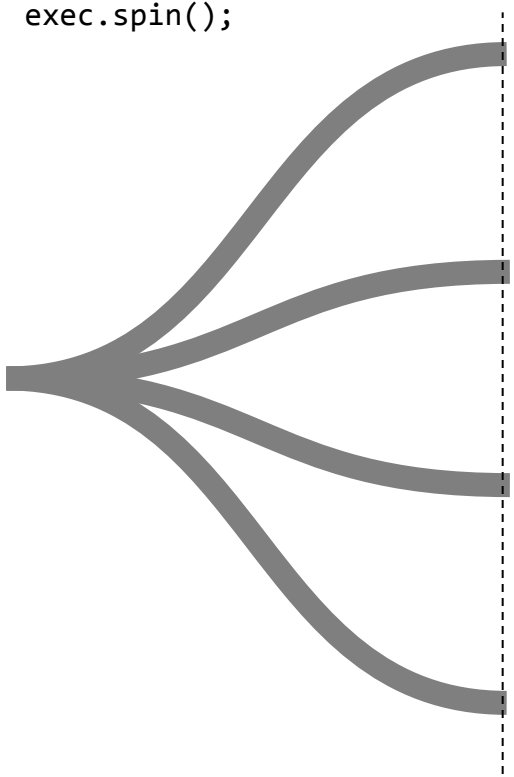
Executor	DDS exchange	FIFO processing?	Improved performance?	Real-time?
SingleThreaded Executor	Wait Set			
StaticSingleThreaded Executor	Wait Set			
MultiThreaded Executor	Wait Set			
rclc-Executor	Wait Set			
Events Executor	Listener			

Real-Time Workshop Executor

- Designed specifically for this workshop
- Provides API to set priority of subscriptions
- Built on Multi-threaded executor
- Uses OS’s existing real-time scheduler
    - SCHED_FIFO

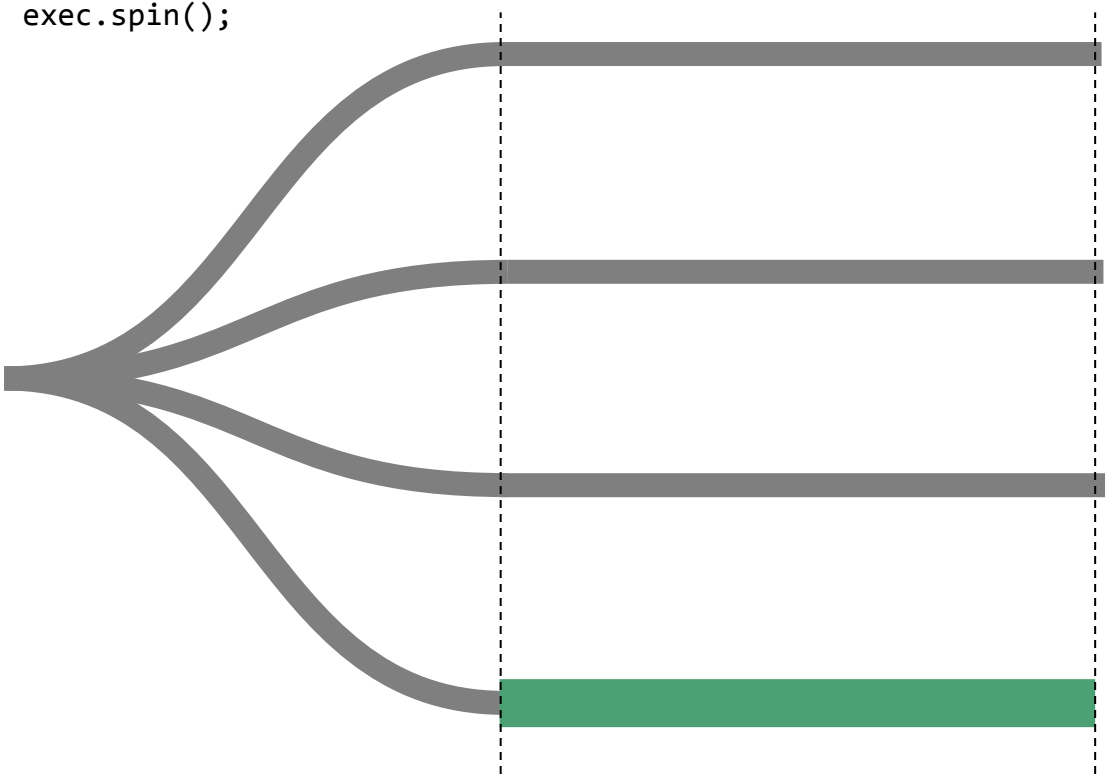
Real-Time Workshop Executor

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



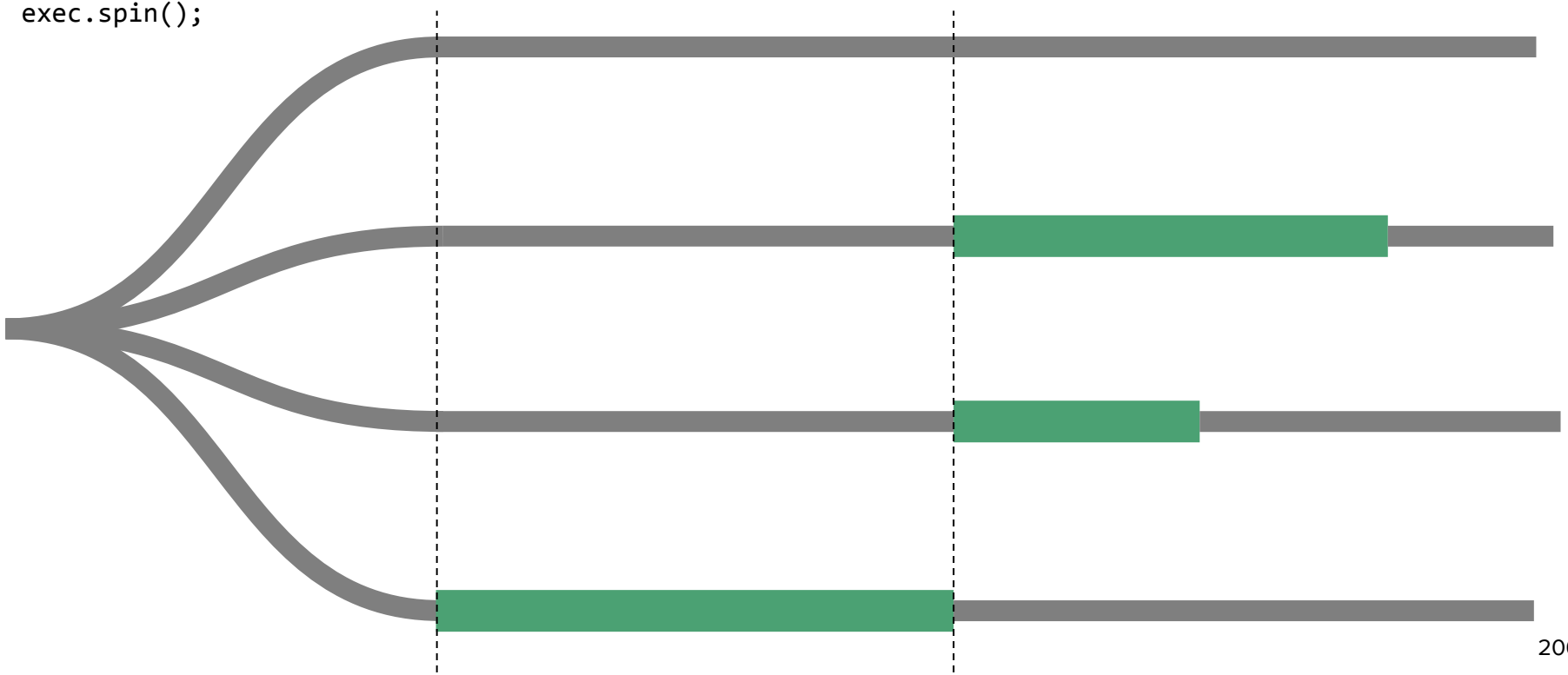
Real-Time Workshop Executor

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



Real-Time Workshop Executor

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



Real-Time Workshop Executor

- # threads > # cores
- Highest priority thread runs

Real-Time Workshop Executor Design

- Priority of thread set by subscription executing on it
- Idle threads are highest priority
    - Doesn’t consume CPU until work is released

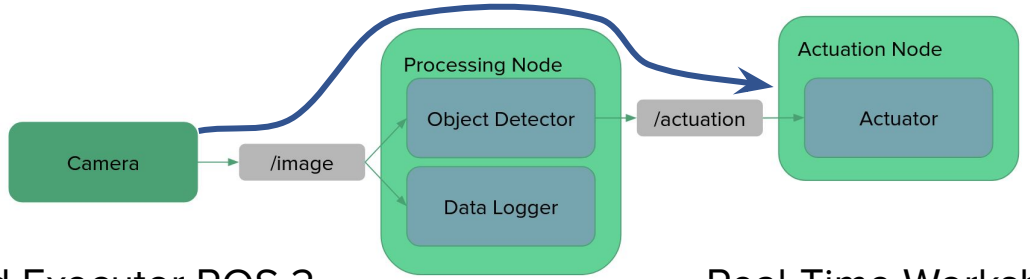
Real-Time Workshop Executor Usage

```
class OjbectDetectingNode : 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);
};
```

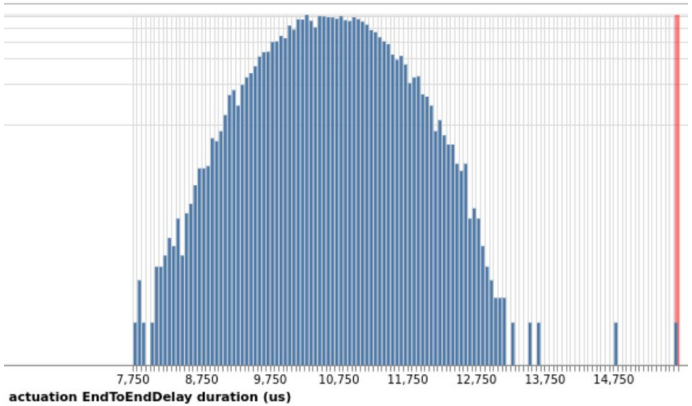
```
image_sub_ = this->create_subscription<sensor_msgs::msg::Image>(
    "image_raw", 10, std::bind(&ObjectDetectingNode::image_callback, this, _1));
```

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

Real-Time Workshop Executor: End-To-End Latency Results

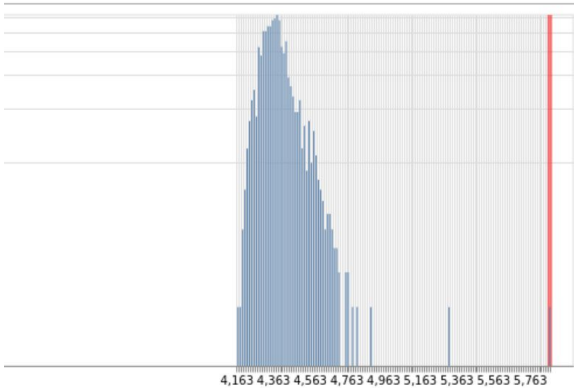


Multi-Threaded Executor ROS 2



High latency

Real-Time Workshop Executor



Low latency

Pros / Cons

- Fixes unbounded delay with first-in-first-out ordering
- Priorities are settable by user

- Mutually exclusive callbacks
  - Good for avoiding race conditions
  - Better executor would have more specific semantics to enforce critical sections of code

Summary

- ROS 2 Execution Management has complex semantics
- Impact on real-time: un-predictable delays, non-FIFO processing
- Several Executor with improvements available (Events-Executor, StaticSingleThreaded); but none are real-time capable.
- Experimental real-time Executor with simple API

Related Work

- 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.

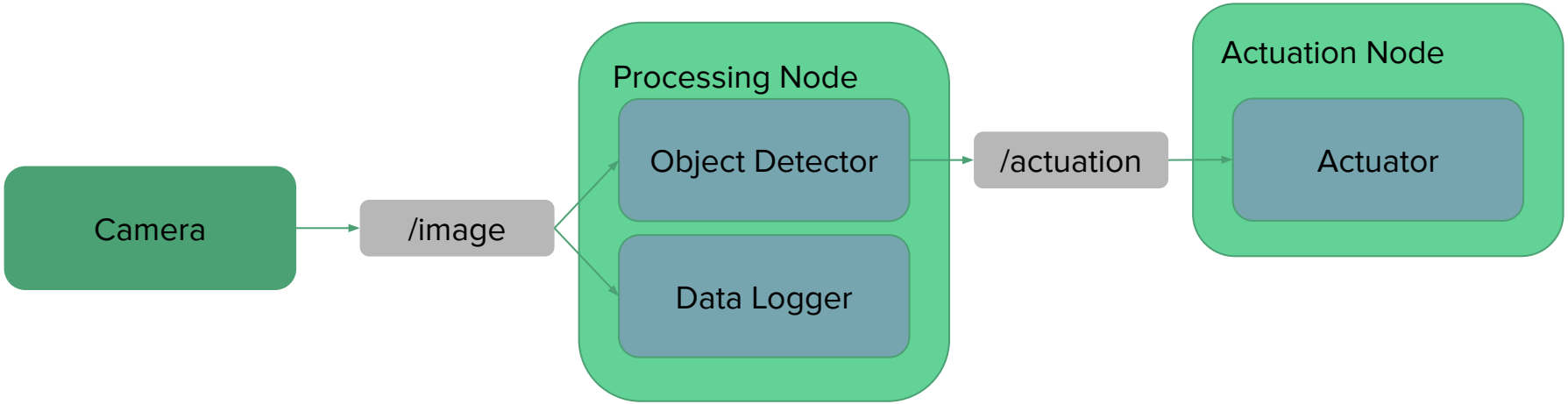
- Blass, Tobias, et al. "Automatic latency management for ros 2: Benefits, challenges, and open problems." 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, 2021.

- reference_system, Executor Workshop, ROS world 2021: https://www.apex.ai/roscon-21

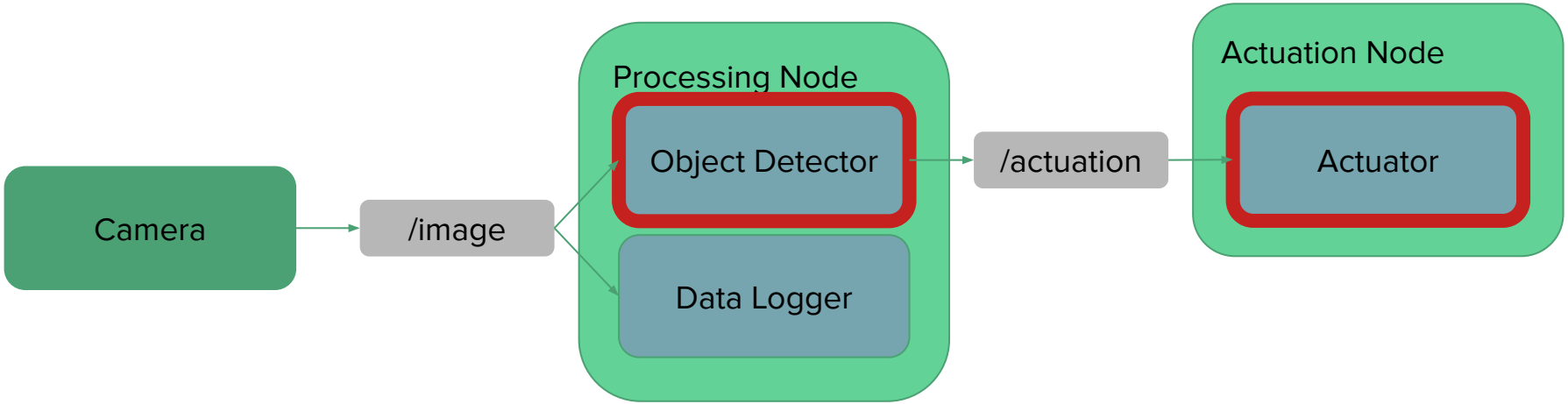
- Events Executor: https://github.com/irobot-ros/events-executor

Hands-on 3

Example Application

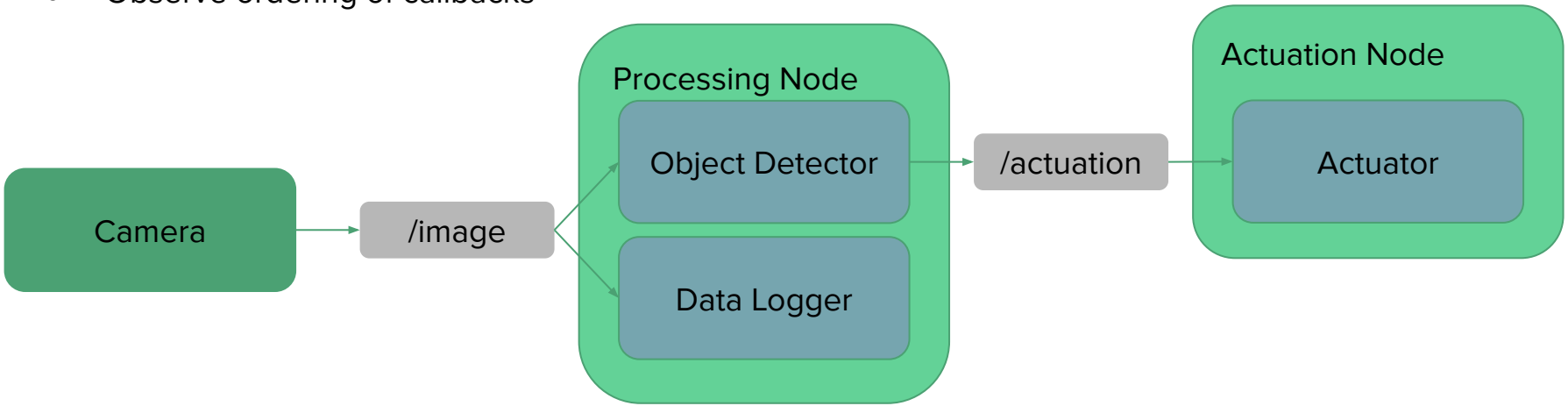


Real-Time vs Best-Effort



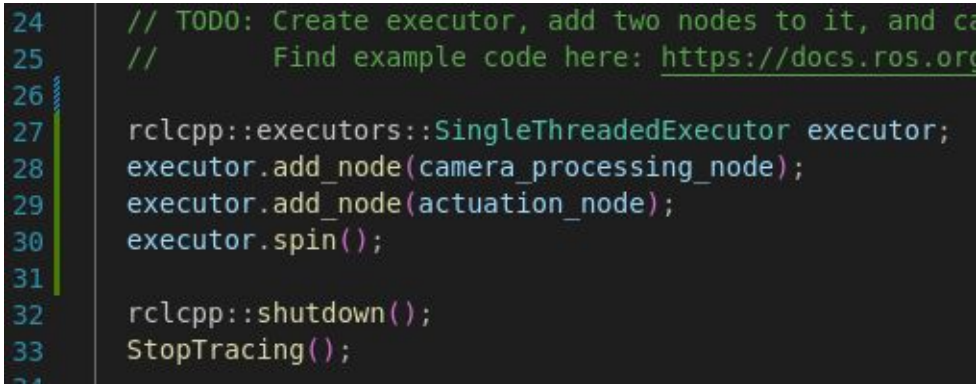
Exercise 3-1 Objectives

- Run application without priorities
- Observe ordering of callbacks



Exercise 3-1

- Open src/camera_demo/src/main.cc
- Declare executor and add nodes




```
cd /code/exercise3-1
colcon build
```

Exercise 3-1: Build and Run

1. (If you haven’t already)

```
cd /code/exercise3-1
colcon build
```

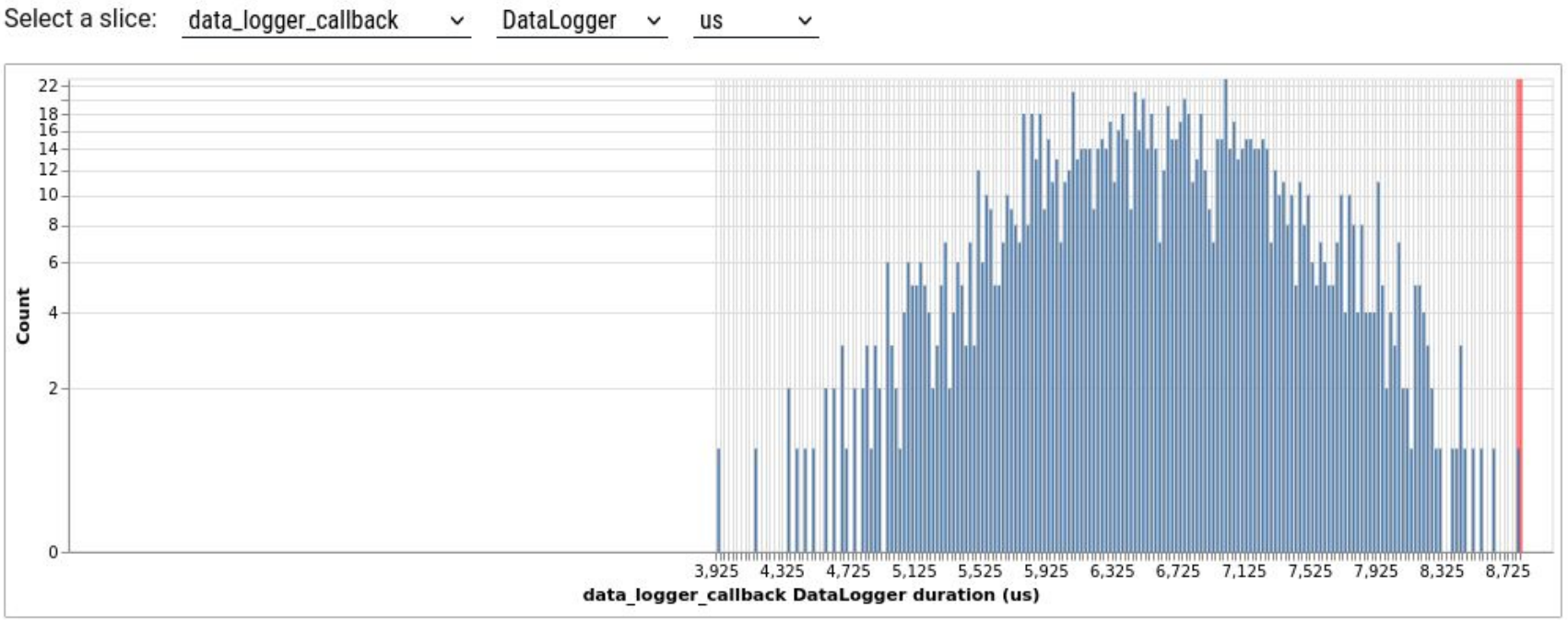
2.

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

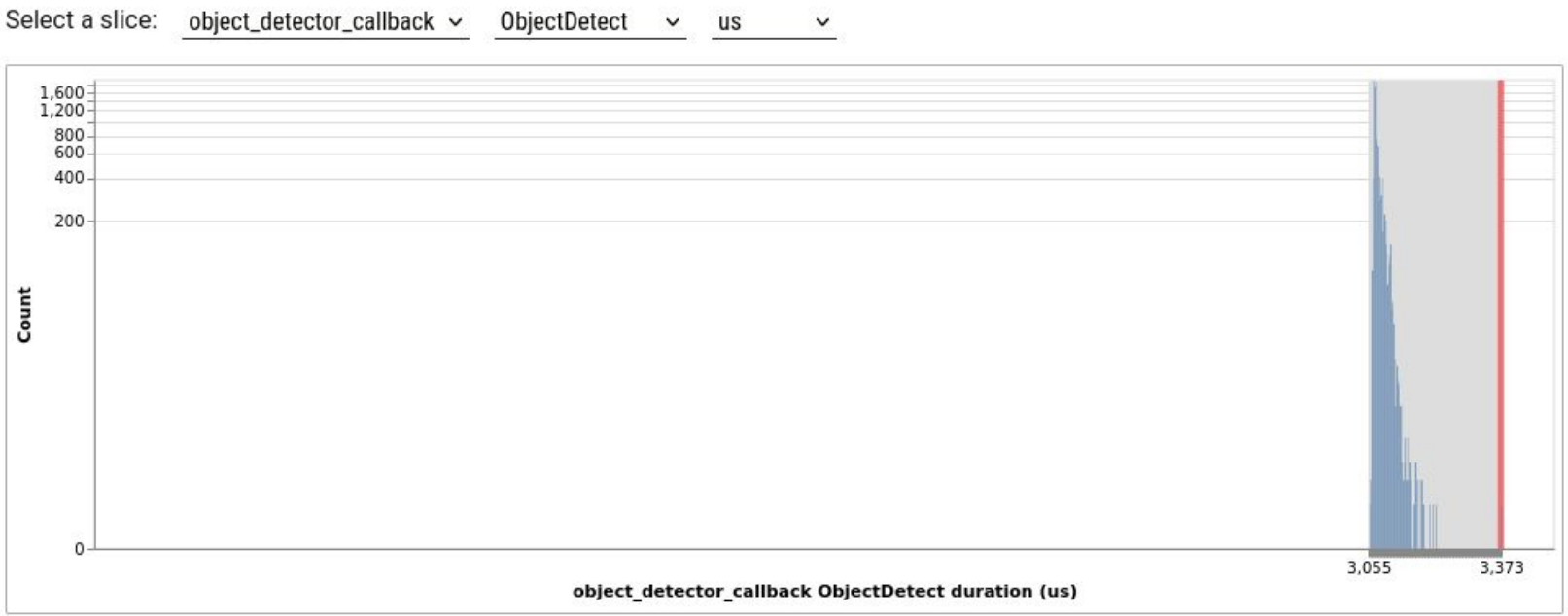
Visualize Trace

- Upload exercise3-1.perfetto to the visualization tool
  - Alternatively use the pre-provided file at results/solution.perfetto
- Zoom in
- Data Logger runs before ObjectDetector

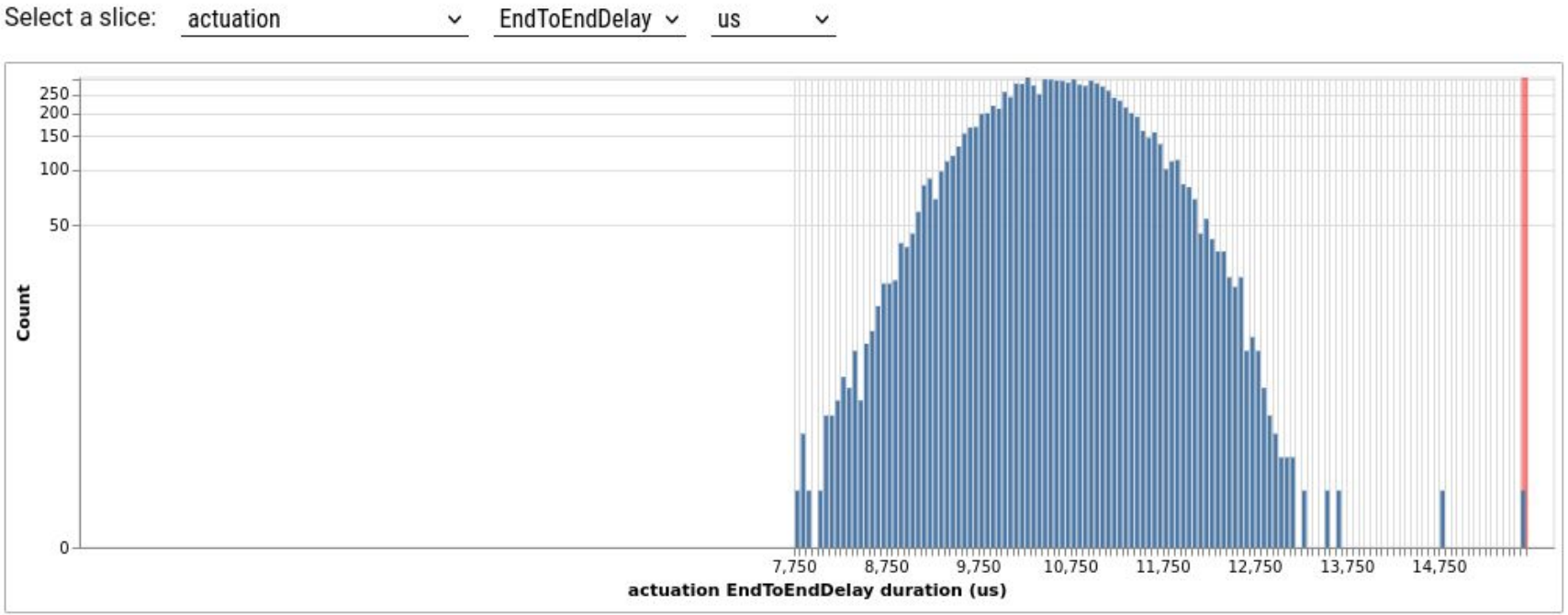
Visualize Latencies - DataLogger



Visualize Latencies - Object Detector



Visualize Latencies - End to End Latency

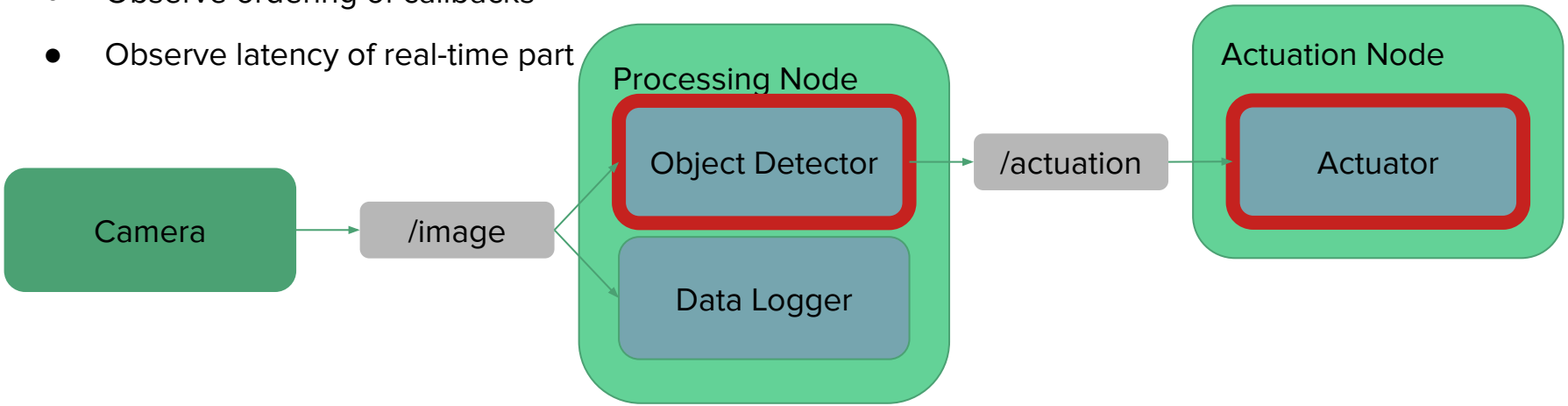


Exercise 3-2: Real-Time Executor

- Can’t specify priority in vanilla executor
- Exercise 3-2 modifies the ros2 executor to allow setting priorities
- Rudimentary realtime executor also robust in face of stressed system

Exercise 3-2 Objectives

- Run application with priorities
- Observe ordering of callbacks
- Observe latency of real-time part



- Close any source files from Exercise 3-1

In exercise3-2 workspace

Open src/camera_demo/src/main.cc
Add your changes from exercise 3-1
Use multi-threaded executor
Allows data logger and object detector to act independently

```
25  // TODO: Copy your solution from Exercise 3-1, but
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();
```

Open src/camera_demo/src/application_nodes.cc
Find the CameraProcessingNode constructor
Set object detector to be high priority
Data logger is set to low priority by default

```
39    publisher_ = this->create_publisher<std_msgs::msg
40
41    // TODO: Set the object detecting subscription to
42    //       Look at the actuation node for example co
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
```

Exercise 3-2: Build and Run

In one terminal

1. (If you haven't already)

```
cd /code/exercise3-2
colcon build
```

3.

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

In another terminal

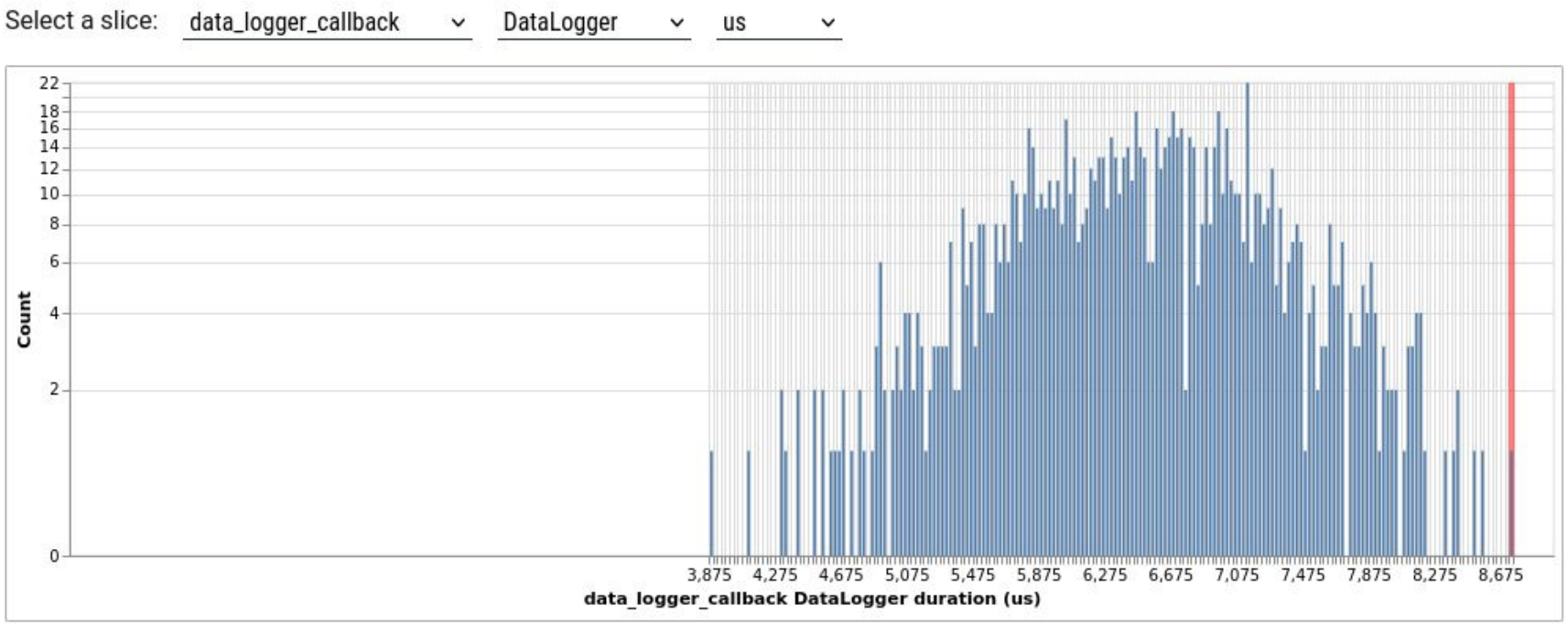
2.

```
docker/shell
/code/stress.sh
```

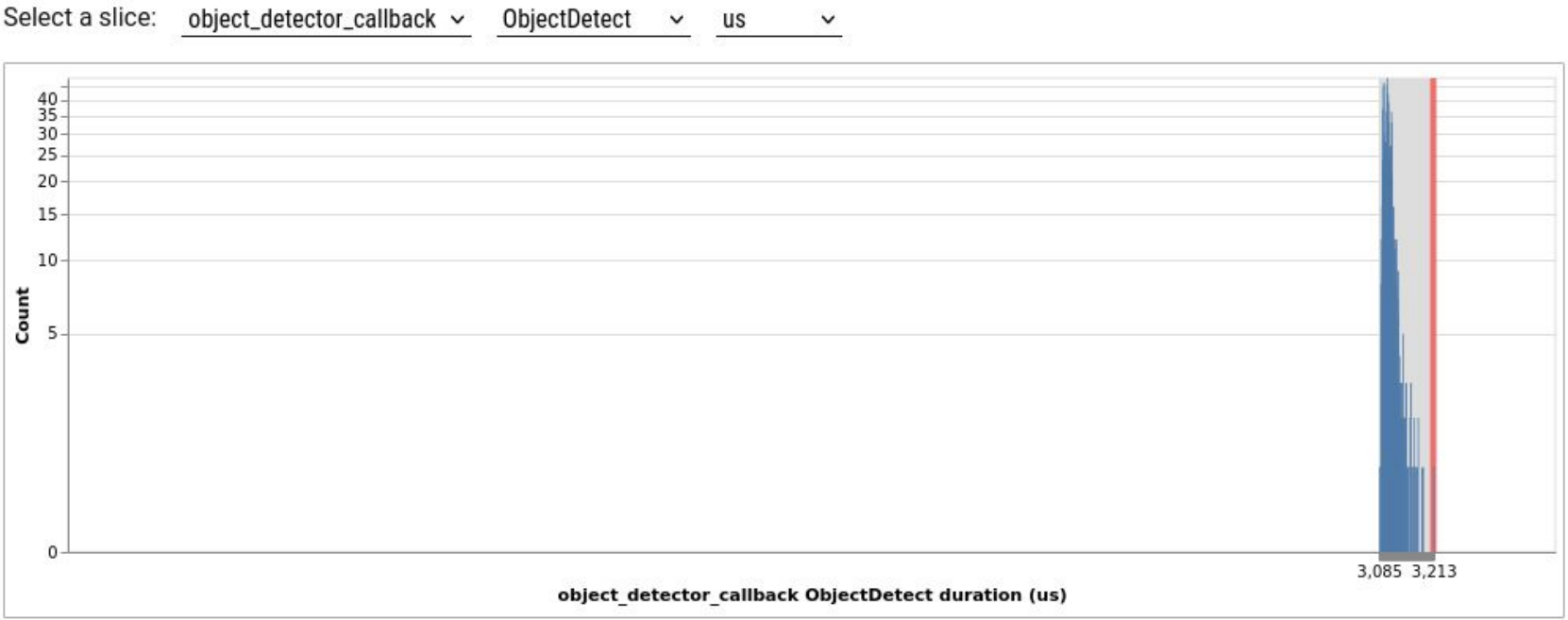
Visualize Trace

- Upload exercise3-2.perfetto to the visualization tool
  - Alternatively use the pre-provided file at results/solution.perfetto

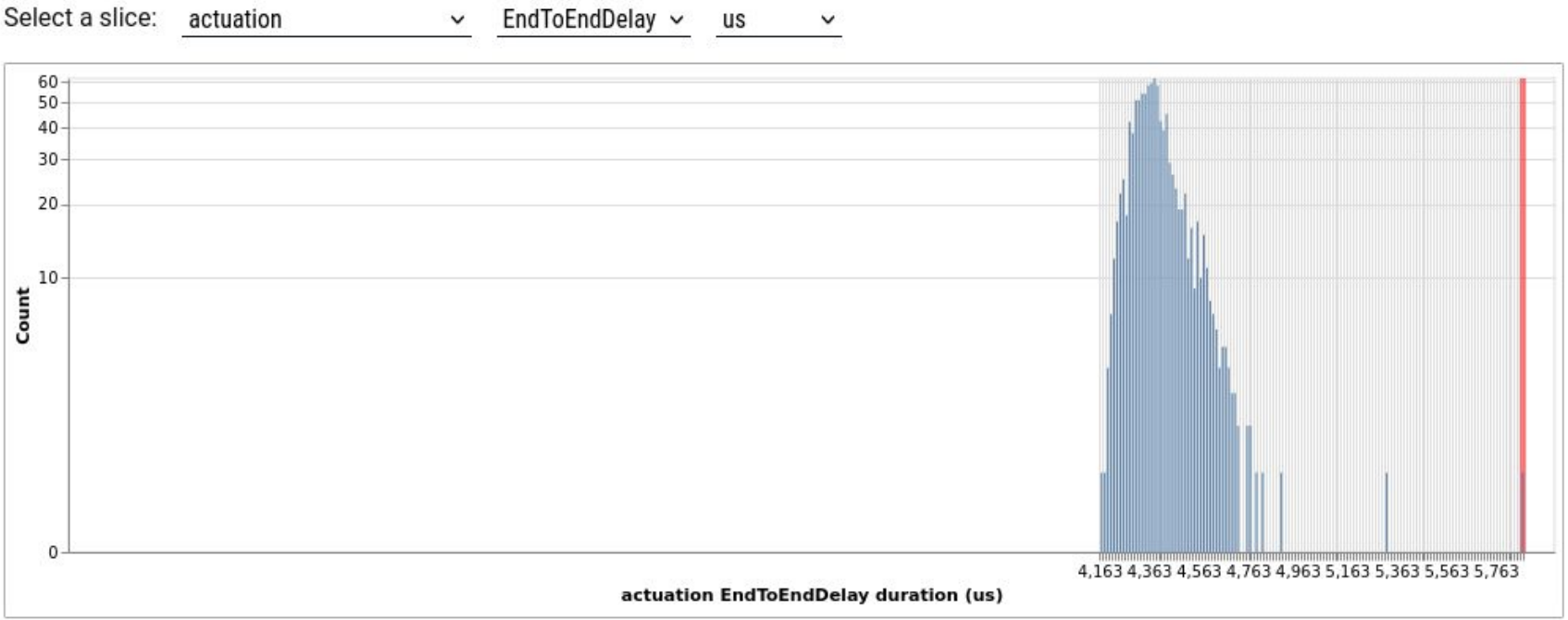
Visualize Latencies - DataLogger



Visualize Latencies - DataLogger



Visualize Latencies - End to End Latency



Exercise 3 Summary

- In 3-1, data logger delays object detector
- In 3-2, object detector runs first (or in parallel, if possible)

Supplementary exercise: rerun without and without stress

- How does exercise 3-1 (no realtime) compare with and without stress?
- How does exercise 3-2 (realtime) compare with and without stress?
- Does stress affect latency of default executor?
- Does stress latency of real-time callbacks?

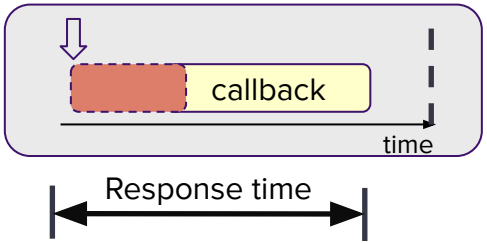
Real-Time + ROS 2 with existing tools

Real-Time + ROS 2 with existing tools

- Overview of Real-Time Systems
- Prioritization of callbacks
  - Single ROS 2 node
  - Callback groups
  - System level (PICAS)
- Callback processing order
- Reducing communication time (loaned messages, zero-copy)
- Tooling and benchmarking

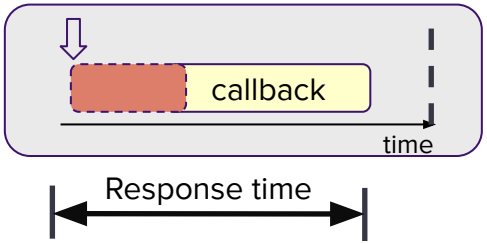
Typical Real-Time Requirements

- Bounded response time

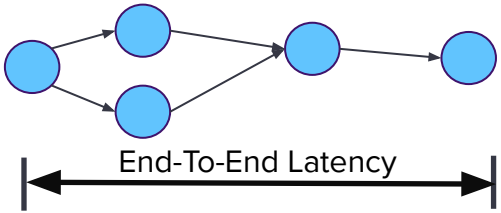


Typical Real-Time Requirements

- Bounded response time

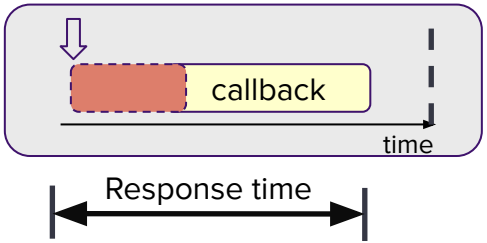


- Guaranteed End-To-End Latency

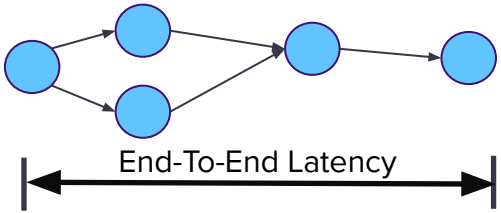


Typical Real-Time Requirements

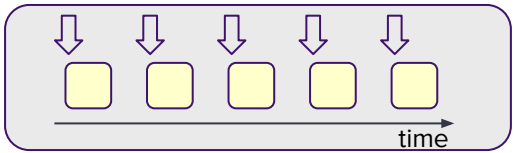
- Bounded response time



- Guaranteed End-To-End Latency



- Deterministic update-rates



Overview of Real-Time Systems

THEORY

- Since 1970 ...
- Model requirements
  - execution times
  - Periodic activation; invocations/time interval
  - Data-triggered, time-triggered
- Scheduling algorithms
  - Rate monotonic
  - Earliest Deadline First
  - Reservation-based scheduling
  - Probabilistic, ..., etc.
- Complex hardware
  - cache, pipeline, multi-core ...
- Schedulability Analysis

Overview of Real-Time Systems

THEORY

- Since 1970 ...
- Model requirements
  - execution times
  - Periodic activation; invocations/time interval
  - Data-triggered, time-triggered
- Scheduling algorithms
  - Rate monotonic
  - Earliest Deadline First
  - Reservation-based scheduling
  - Probabilistic, ..., etc.
- Complex hardware
  - cache, pipeline, multi-core ...
- Schedulability Analysis

PRACTICE

- Real-Time Operating Systems
  - QNX
  - FreeRTOS, Zephyr, ...
- Real-Time Linux
  - SCHED_FIFO
  - SCHED_DEADLINE (EDF)
- Automotive: rate-monotonic scheduling
  - AUTOSAR
  - Fixed periods
  - LET for synchronized data exchange

Overview of Real-Time Systems

THEORY

- Since 1970 …
- Model require
  - execution ti
  - Periodic act
    interval
  - Data-trigger
- Scheduling al
  - Rate monot
  - Earliest Dea
  - Reservation
  - Probabilistic
- Complex hard
  - cache, pipel
- Schedulability Analysis

PRACTICE

- Real-Time Operating Systems

S, Zephyr, …
inux
IFO
EADLINE (EDF)
rate-monotonic
R
iods
nchronized data
exchange

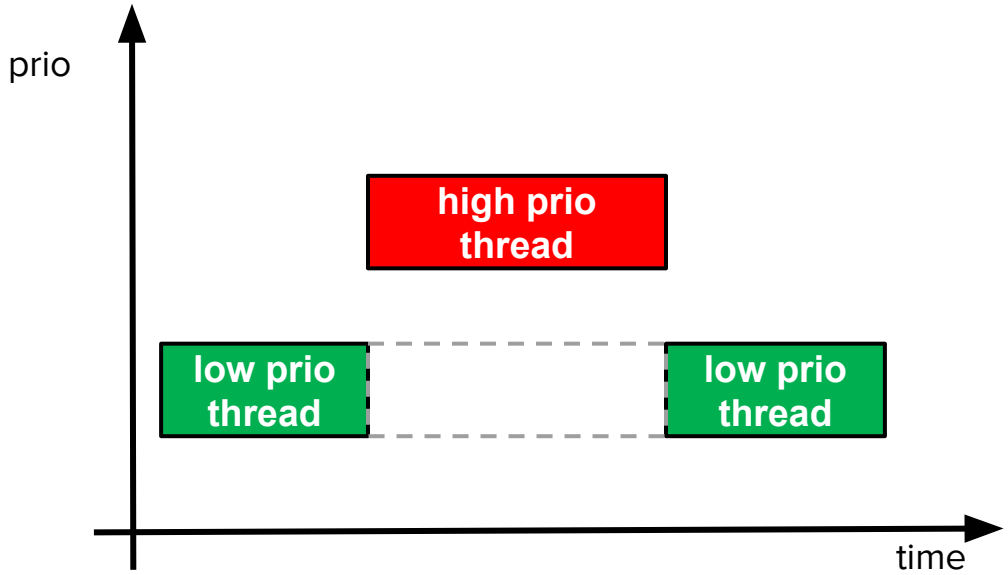
We write the year 2023

ROS 2 is not

real-time capable.

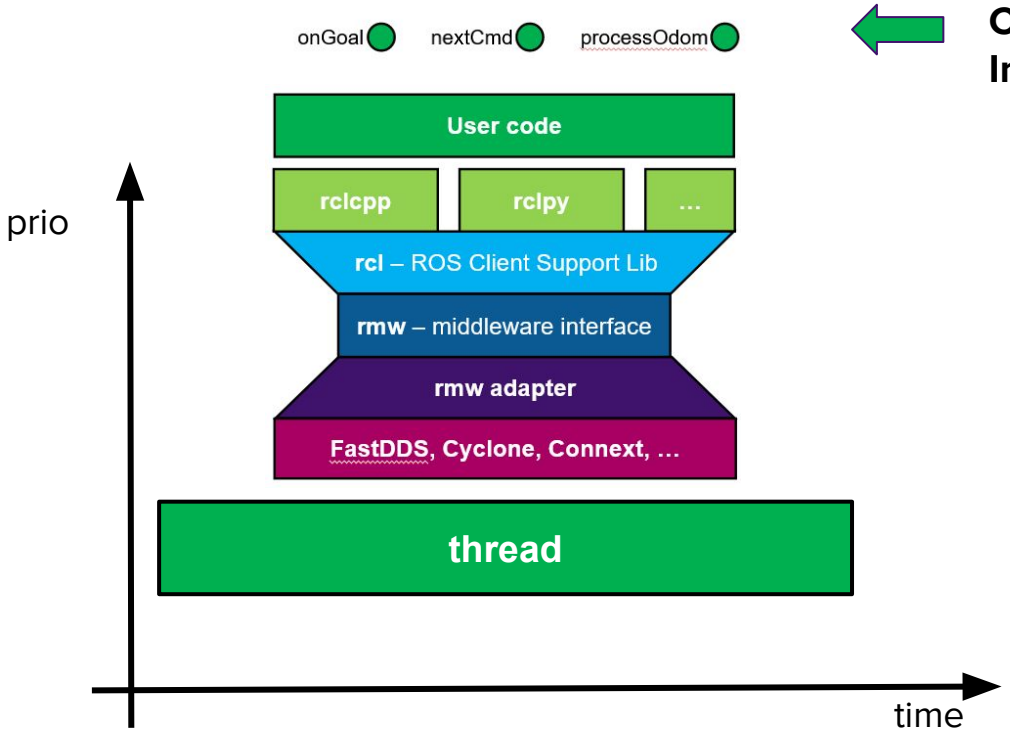
What is the issue?

ROS 2 and the Operating System



The operating system can schedule threads in real-time

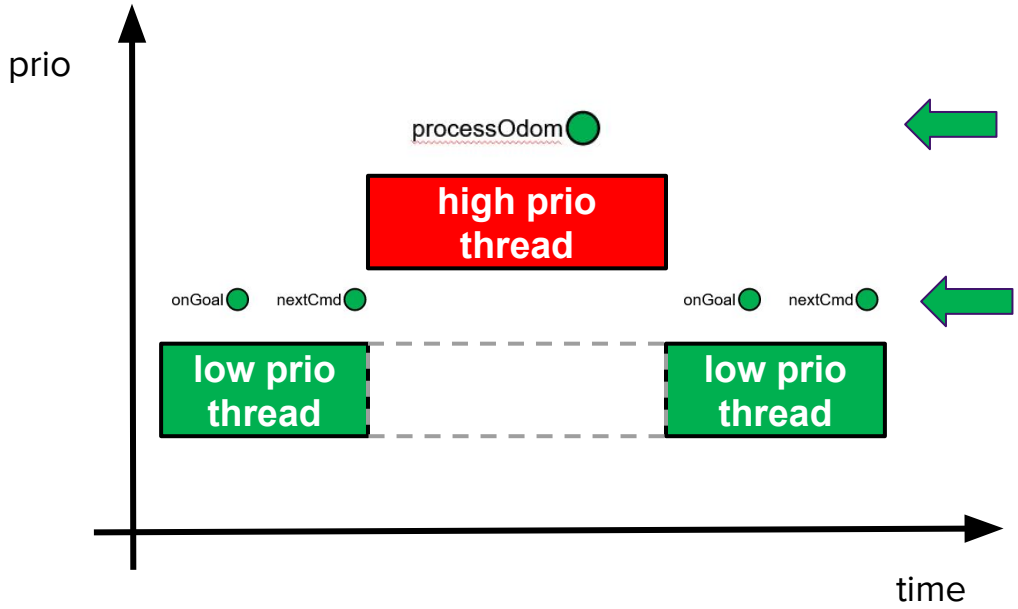
ROS 2 and the Operating System



Callback functions
In one thread

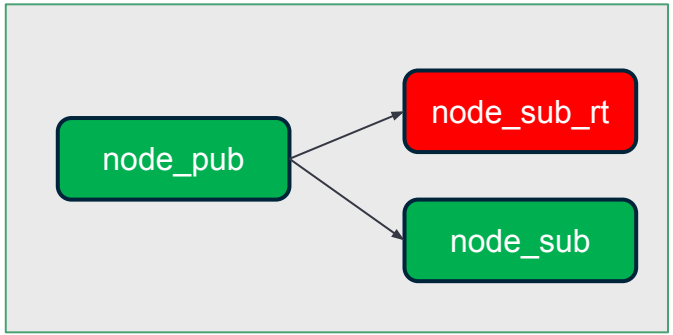
But the ROS 2 stack (with it's Executor) is just **one** thread

Distribute callbacks to multiple threads



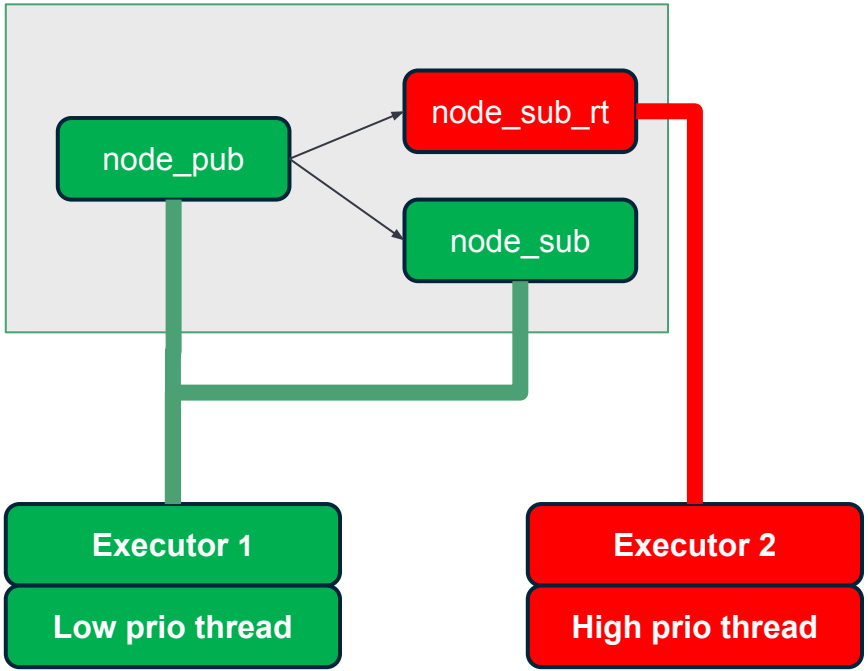
Callback functions in different threads

Application with one real-time node



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

Real-time node is scheduled in high prio thread



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

Example for node prioritization

```
  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);
```

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

Example for node prioritization

```
// 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();
```

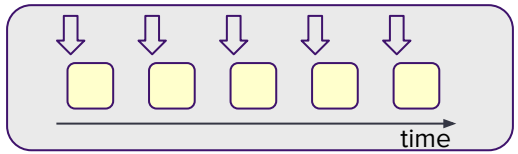
https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_real_time_executor.cpp

Example for node prioritization

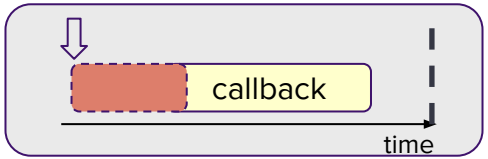
```
// spin non real-time tasks in a se
auto default_thread = std::thread(
   [&]() {
   default_executor.spin();
   });

// spin real-time tasks in a separa
auto realtime_thread = std::thread(
   [&]() {
   realtime_executor.spin();
   });

set_thread_scheduling(realtime_thre
                      options.polic
default_thread.join();
realtime_thread.join();

rclcpp::shutdown();
```

Benefits:

- **Improves bounded update rate**

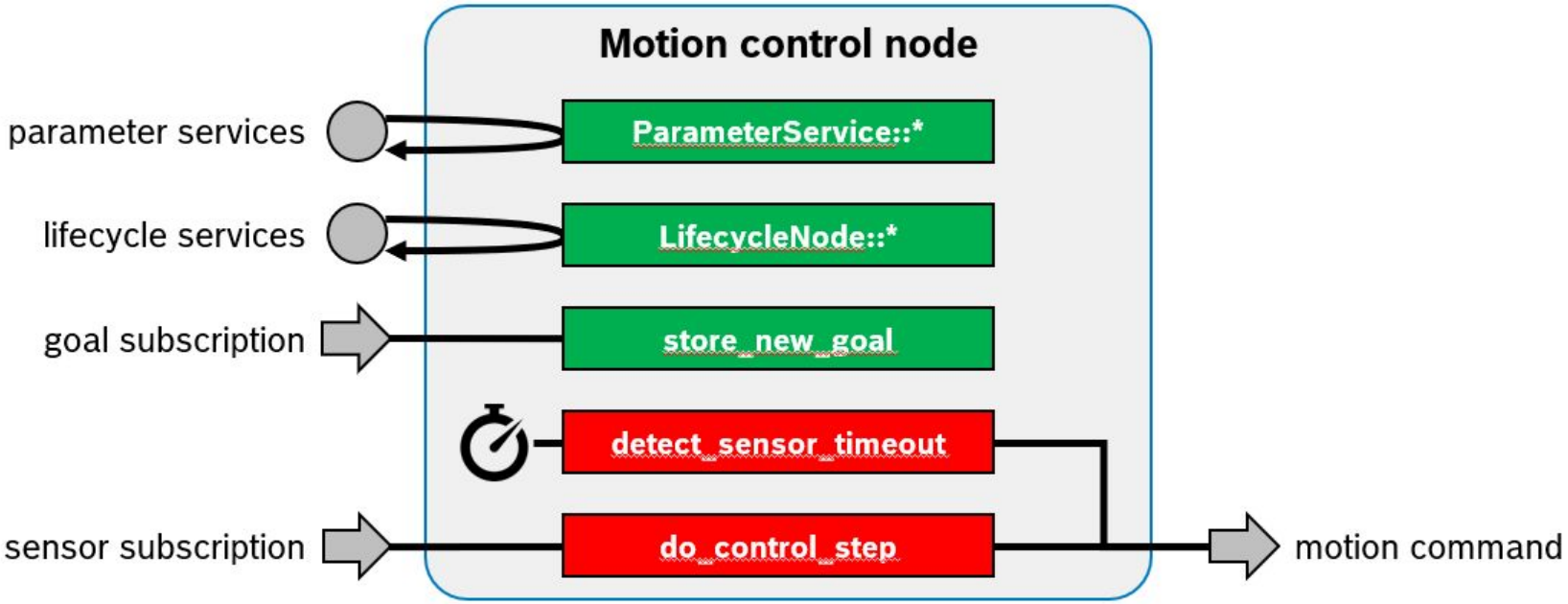


- **Reduces response time**

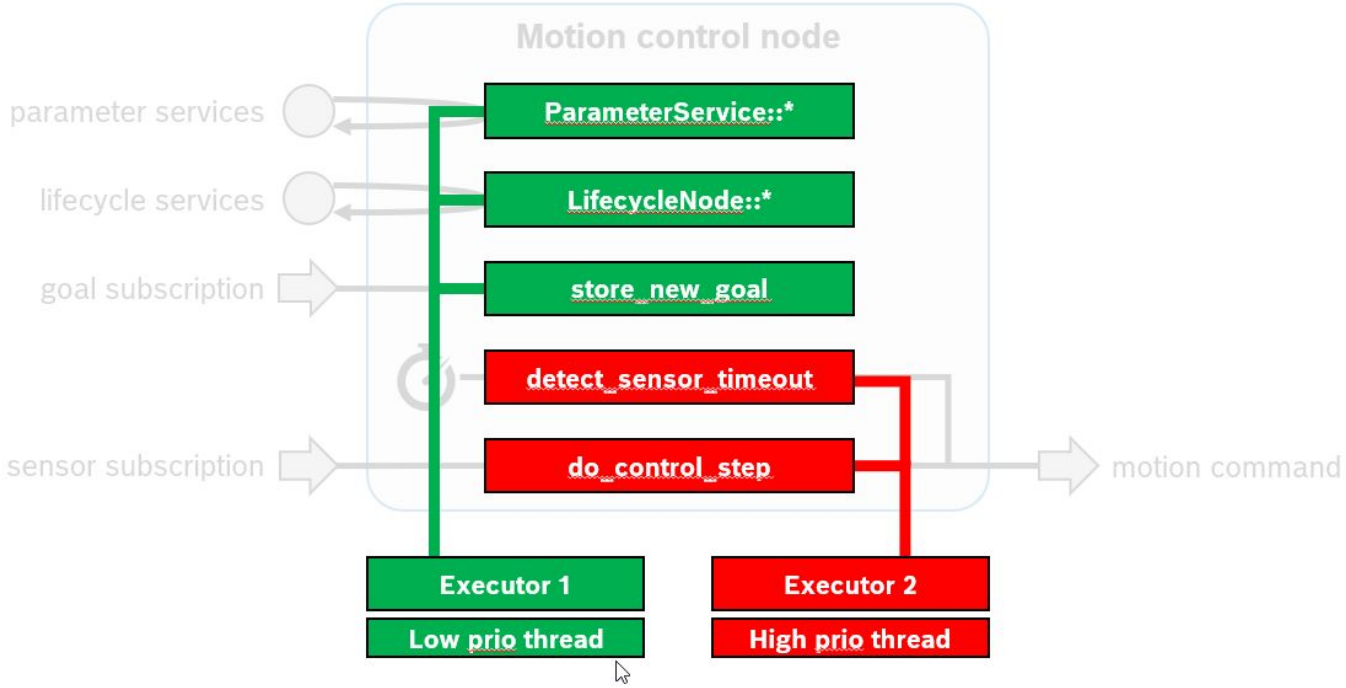


https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_real_time_executor.cpp

Mixed-criticality node



Mixed-criticality node



Ralph Lange: “Callback-group level executor”, Executor Workshop’21

Step 1 - create callback group, assign to subscription

```
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_;
```

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

Step 2: add callback group to an executor

```
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());

...
```

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

Step 3: run executor in real-time thread

```
...
// 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;
}
```

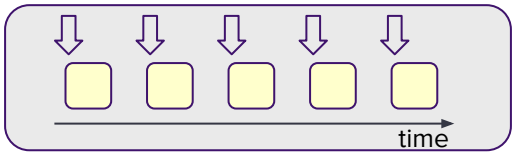
https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_callback_group.cpp

Step 3: run executor in real-time thread

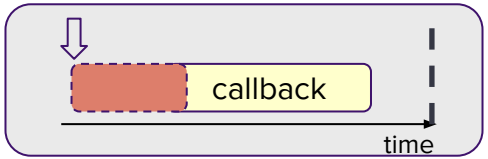
```
...
// spin real-time tasks in a separate th
  auto realtime_thread = std::thread(
     [&]() {
     realtime_executor.spin();
          });

  set_thread_scheduling(realtime_thre
options.priority);

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

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

Benefits:

- **Improves bounded update rate**

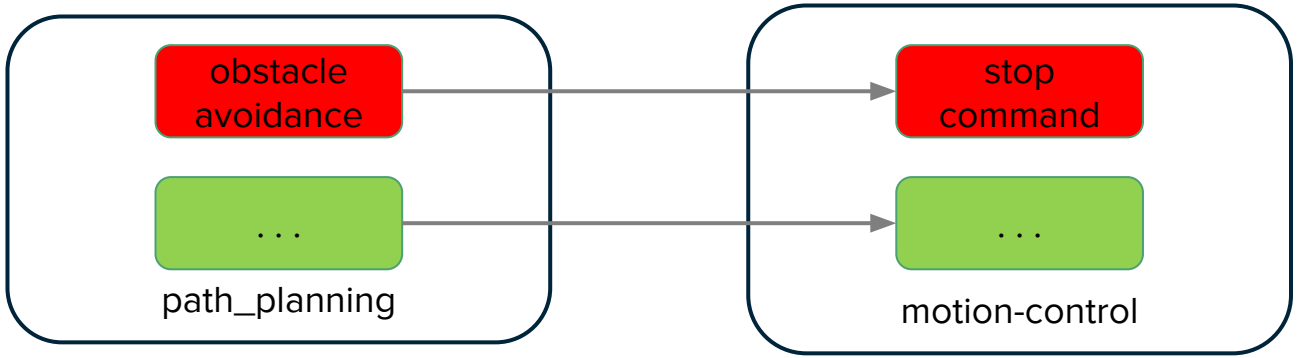


- **Reduces response time**



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

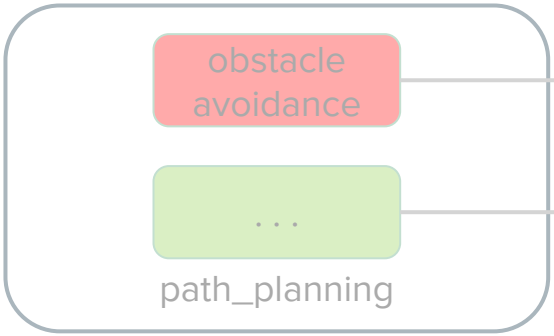
Use-case 2: Multiple nodes with real-time callbacks



- Callback groups from multiple nodes can be added to the same executor and processed in real-time thread
- See complete example on github (link below)

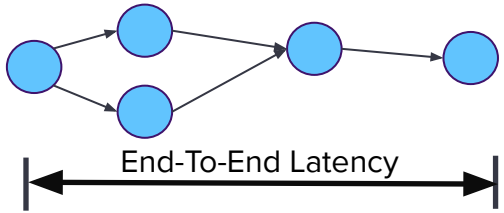
https://github.com/ros-realtime/ros2-realtime-examples/blob/rolling/minimal_scheduling/minimal_scheduling_callback_group.cpp

Multiple nodes with real-time callbacks



Benefits:

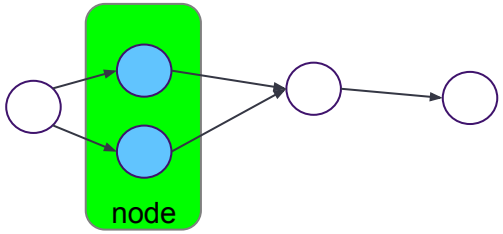
- **Improves End-To-End Latencies**



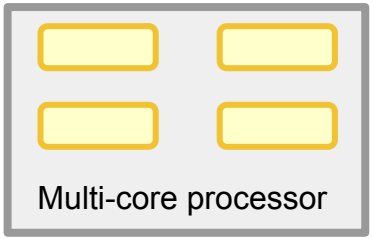
- Callback groups from m
  added to the same executor
- See complete example (below)

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

Parallelization with MultiThreaded Executor: one node

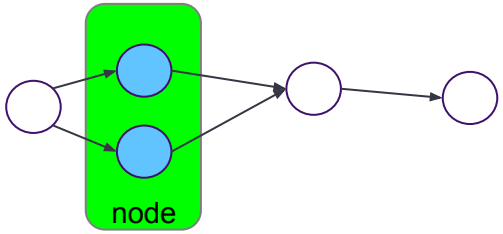



```
int main(int argc, char * argv[])
{
  rclcpp::Node::SharedPtr node = … (sub_a, sub_b);

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

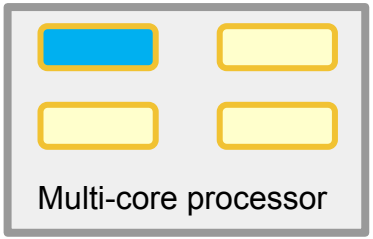


https://docs.ros.org/en/humble/How-To-Guides/Using-callback-groups.html

Parallelization with MultiThreaded Executor: one node




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

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

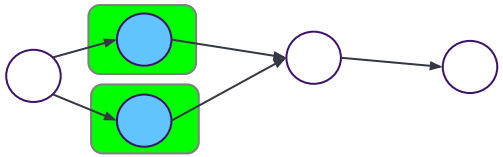


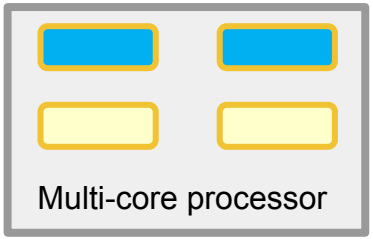
Fails! Implicit callback group is `MutuallyExclusive` by default.

=> all callbacks are processed sequentially

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

Parallelization with MultiThreaded Executor: two nodes





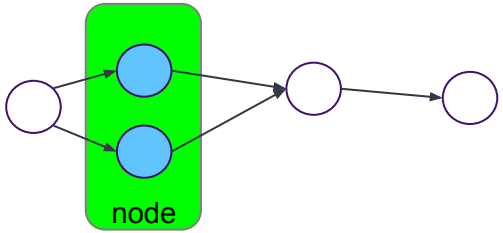

```
int main(int argc, char * argv[])
{
  rclcpp::Node::SharedPtr node1 = … (sub_a);
  rclcpp::Node::SharedPtr node2 = … (sub_b);

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

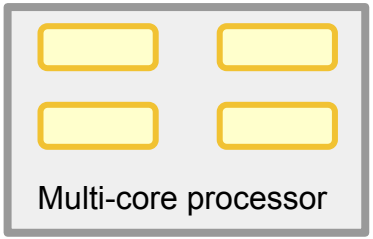
Works! Every node has a callback group.

=> callbacks parallelized

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

One Node, two callbacks, two threads

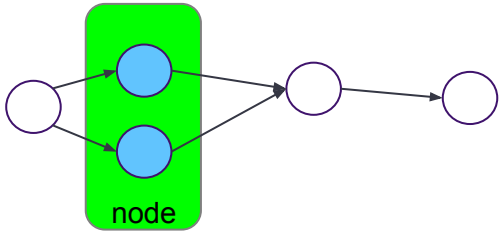


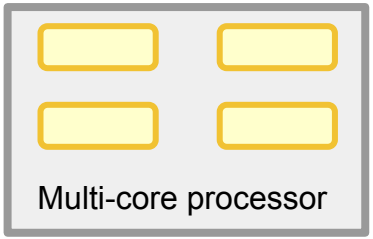



```
class DualThreadedNode : public rclcpp::Node
{ …
  callback_group_sub_a = this->create_callback_group(
rclcpp::CallbackGroupType::MutuallyExclusive);
  callback_group_sub_b = this->create_callback_group(
rclcpp::CallbackGroupType::MutuallyExclusive);


  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

One Node, two callbacks, two threads

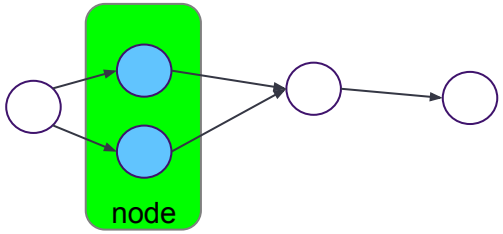




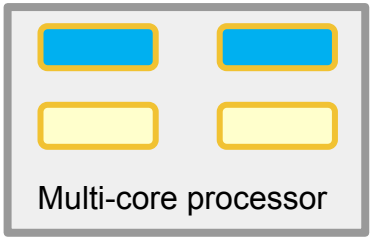

```
class DualThreadedNode : public rclcpp::Node {
  sub_a = this->create_subscription<std_msgs::msg::String>(
"topic_A",…, sub_opt_a);
  sub_b = this->create_subscription<std_msgs::msg::String>(
"topic_B",…, sub_opt_b);
}
main(){
  rclcpp::init(argc, argv);
  rclcpp::executors::MultiThreadedExecutor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

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

One Node, two callbacks, two 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::e
  aut
  exe
  executor.
}
```



Works! Every subscription has its callback group.

=> callbacks parallelized

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

Get precedence for important callbacks


```
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();
```

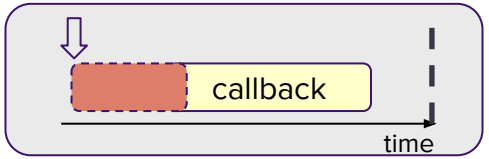
- Processing order depends on the order in internal Node structures
- Depend on
  - object creation time
  - order of subscription creation (inside a node)
- But difficult to maintain in larger projects

Get precedence for important callbacks


```
rclcpp::Node::SharedPtr
rclcpp::Node::SharedPtr
rclcpp::Node::SharedPtr

rclcpp::executors::Stat
executor;
executor.add_node(node1
executor.add_node(node2
executor.add_node(node3
executor.spin();
```

Benefits:

- **Reduces response time**



order depends on the
rnal Node structures

creation time
f subscription creation
a node)
to maintain in larger

Reducing communication latency: loaned message

- Loaned messages
  - Typically publisher allocates memory for the published message, which is copied in middleware
  - Loaned message: publisher writes directly in middleware owned data structure
  - Benefit: reduces communication latency

https://design.ros2.org/articles/zero_copy.html

- Code Example:

```
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

Reducing communication latency: loaned message

- Loaned message
  - Typically publisher allocates memory for the published message, which is copied in middleware
  - Loaned message: publisher writes directly in middleware owned data structure
  - Benefit: reduces communication latency

https://design.ros2.org/articles/zero_copy.html

- Code Example:

```
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

Reducing communication latency: zero-copy middleware

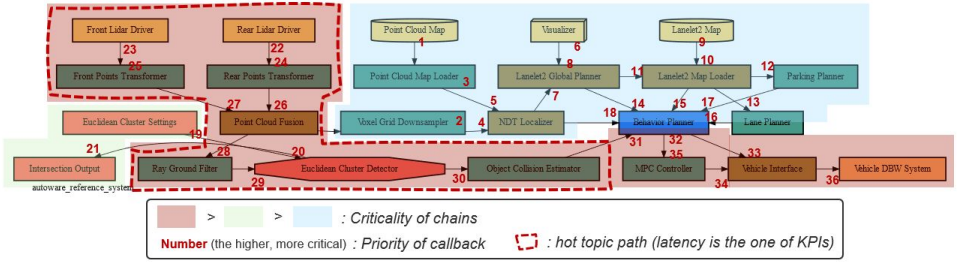
- LiDAR, camera, radar sensors generate large amounts of data with high frequency
- Transferring large volumes of data efficiently is important e.g. autonomous driving
- Zero Copy Middleware uses shared memory for communication
- Supported by many DDS implementations
  - FastDDS
  - CycloneDDS,
  - ...
- Further middleware support:
  - Iceoryx: inter-process-communication (IPC) middleware, automotive
  - Zenoh: pub/sub/query protocol, currently ROS 2 support developed as DDS alternative

PICAS: Priority-driven chain-aware scheduling

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

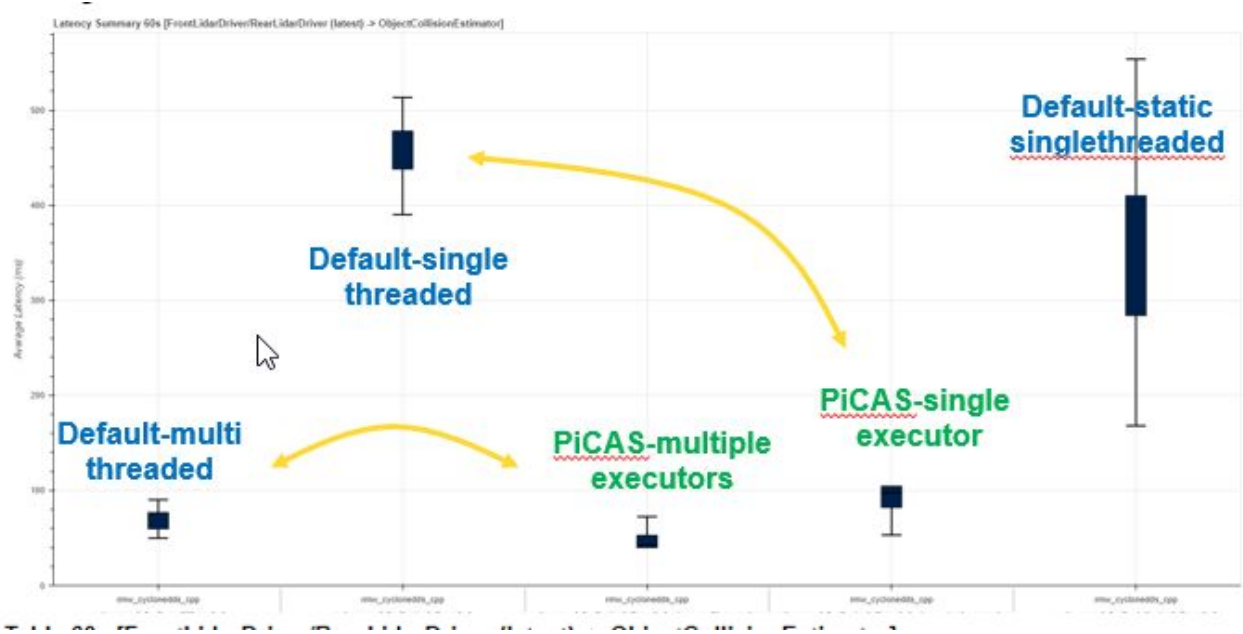
- **Key idea**: enables ***prioritization of mission-critical chains*** across complex abstraction layers of ROS 2
  - To minimize end-to-end latency
  - To ensure predictability even when the system is overloaded
- **PiCAS**: Executor + Resource Allocation Algorithms + Timing Analysis
  - **PiCAS executor:** 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**

- Autoware model



- Single executor instance & multiple executor instances
  - Based on the PiCAS priority assignment and node-to-executor allocation algorithms
  - Algorithm implementation: https://github.com/rtenlab/ros2-picas

PICAS: Priority-driven chain-aware scheduling



Tooling and benchmarking

- ros2_tracing
  - Gather runtime execution information: callback duration, ROS 2 stack, End-To-End latency
  - Depends on LTTNG (Linux)
  - Very low overhead
- ros-realtime/reference_system
  - Performance benchmarking
  - Supports Raspberry Pi, Linux
- Real-time Linux Image for Raspberry Pi
  - Used in this workshop

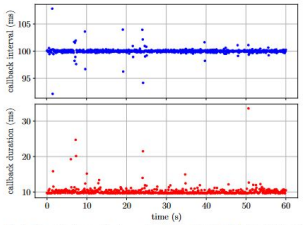
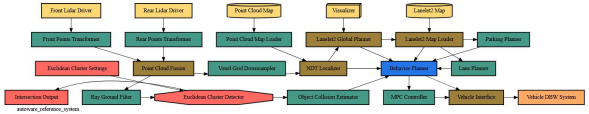
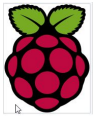


Fig. 4. Example timer callback execution interval (top) and duration (bottom) over time. The callback period is set to 100 ms, while the callback duration depends on the work done. Both contain outliers.



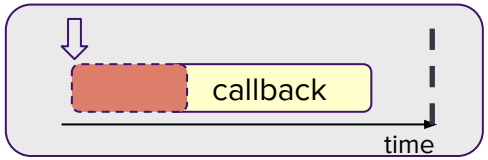

Real-Time Linux


https://github.com/ros2/ros2_tracing


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

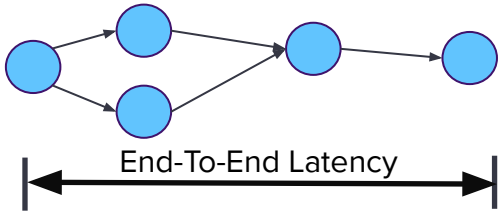

https://github.com/ros-realtime/ros-realtime-rpi4-image

Summary: real-time with ROS 2 features

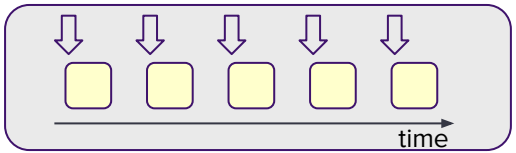


Bounded Response Time

- ❏ Control callback processing order
- ❏ Prioritization of nodes
- ❏ Prioritization with callback groups



- ❏ Prioritization of nodes and with callback groups
- ❏ Parallelization with Multi-Threaded Executor
- ❏ Loaned messages and zero-copy middleware
- ❏ PICAS, Events-Executor, ...



Update Rate

- ❏ Prioritization of nodes
- ❏ Prioritization with callback groups

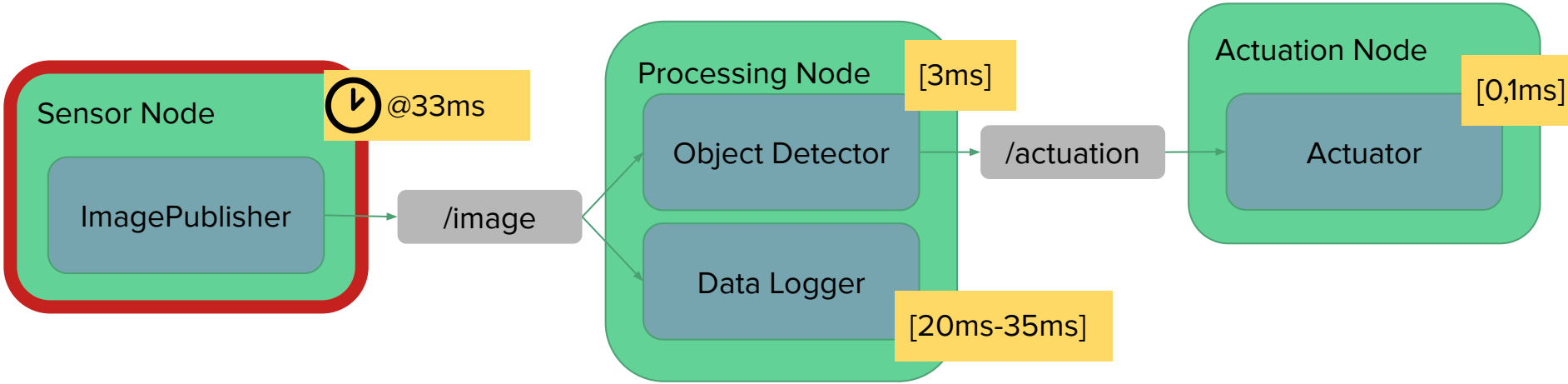
Real-time ROS 2: future work

- Dynamic memory allocation still a problem
  - shared_ptr in every callbacks
  - vectors/STL data types for all messages
  - Possible page faults
- `std::mutex` usage in the executor still a problem
  - Priority inversion

Hands-on 4

Exercise 4-1: real-time ROS 2 Node

- Objective: Run ROS 2 Node in real-time by using default Executor in real-time thread



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

Exercise 4-1: Using Default ROS 2 Executor

```
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();
 ...
}
```

One Executor with three Nodes

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

Exercise 4-1: Image publisher

```
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);
}
```

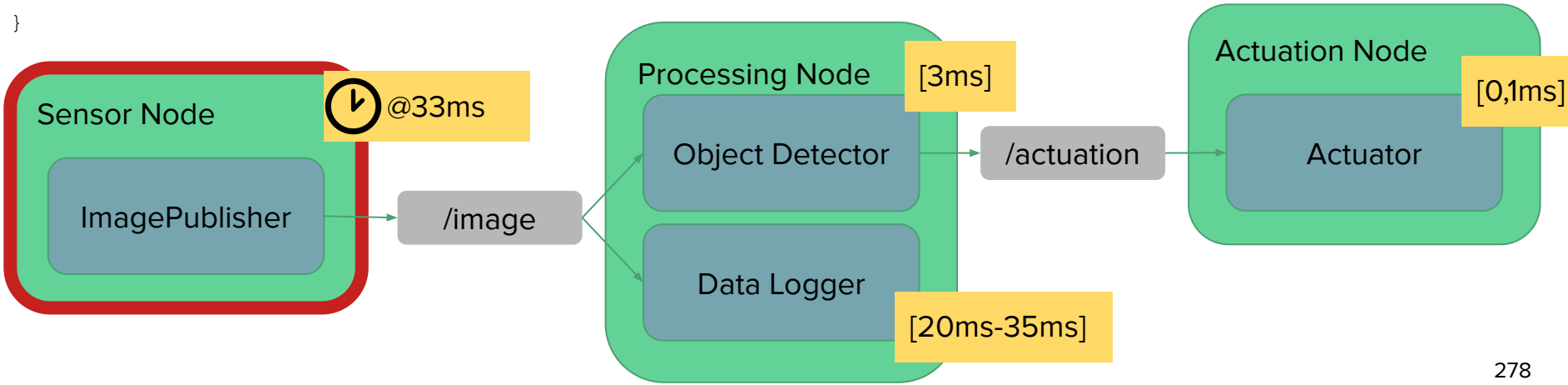
Publisher Node with a Timer

Timer callback

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

Exercise 4-1: best effort callback with large variation time

```
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));
 . . .
}
```



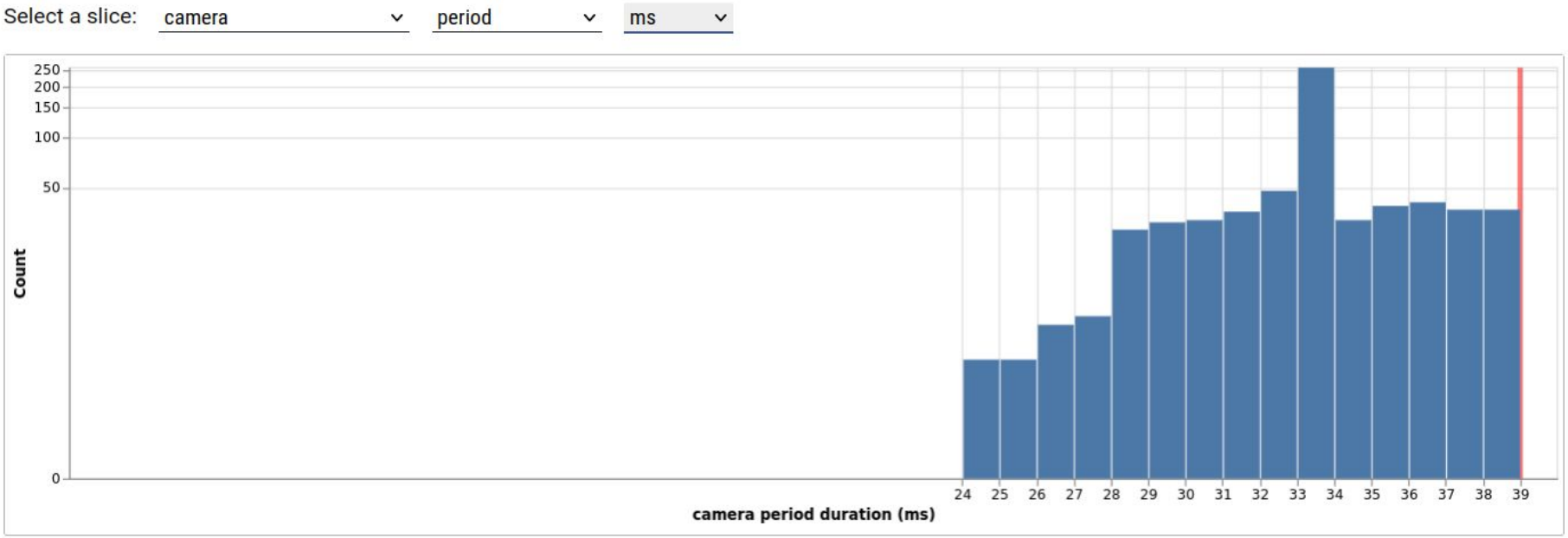
Exercise 4-1: Build and Run

In a terminal:

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

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

Exercise 4-1: Result default ROS 2 Executor



period: [24ms, 39ms]

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

Exercise 4-1: Update source code

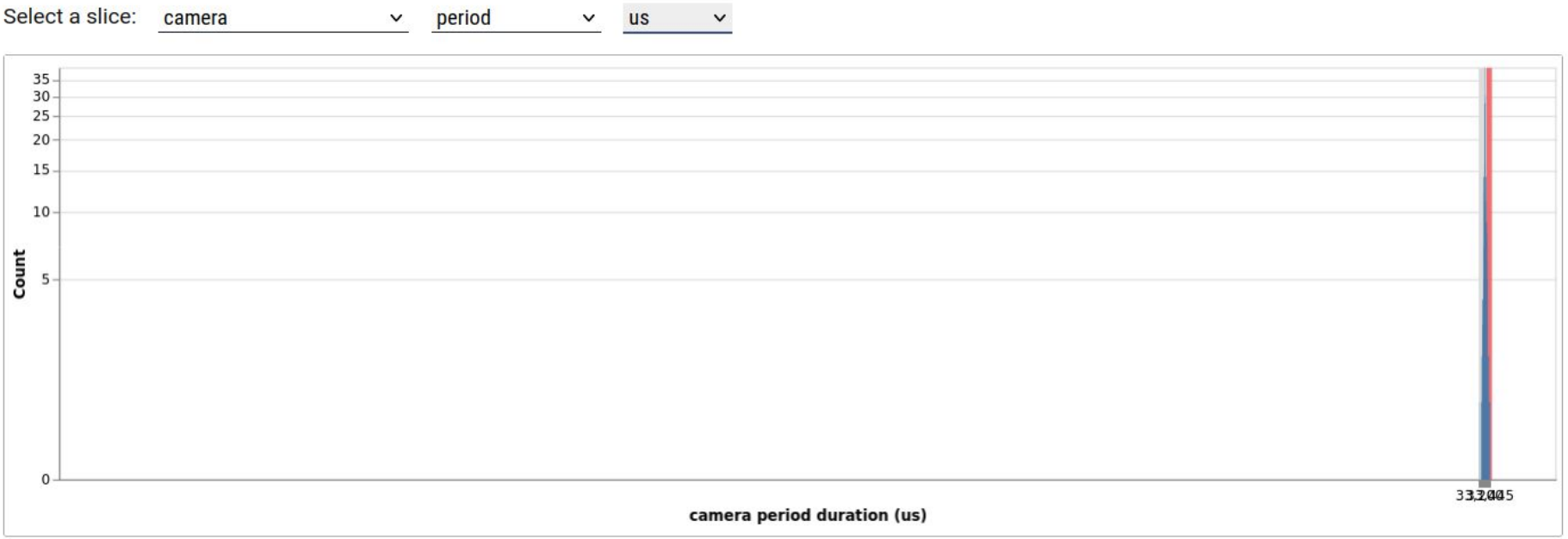
```
29 // Exercise 4-1 : comment-out next line
30 executor.add_node(camera_node);
…
35 // Exercise 4-1 uncomment next block
36 rclcpp::executors::SingleThreadedExecutor image_pub_executor;
37 image_pub_executor.add_node(camera_node);
...
41 thr = std::thread([&image_pub_executor] {
42   sched_param sch;
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: ") +
std::strerror(errno)};
46   }
47    image_pub_executor.spin();
48 });
54 // Exercise 4-1 (4): uncomment next block
55 thr.join();
```

terminal 1:

colcon build
./run.sh (for 15sec)

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

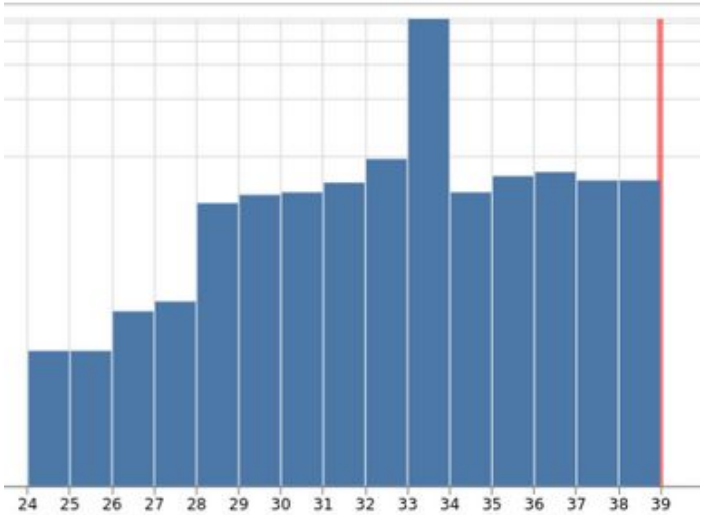
Exercise 4-1: Result Executor in real-time thread



period: [33.2ms, 33.4ms]

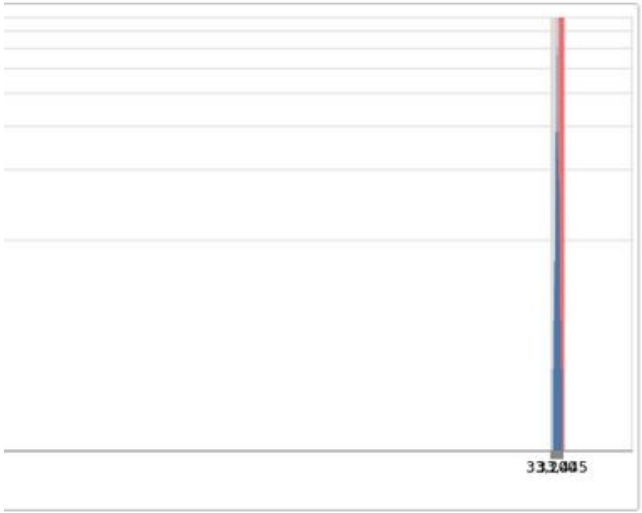
Exercise 4-1: Summary

Publisher with ROS 2 default



Large variation

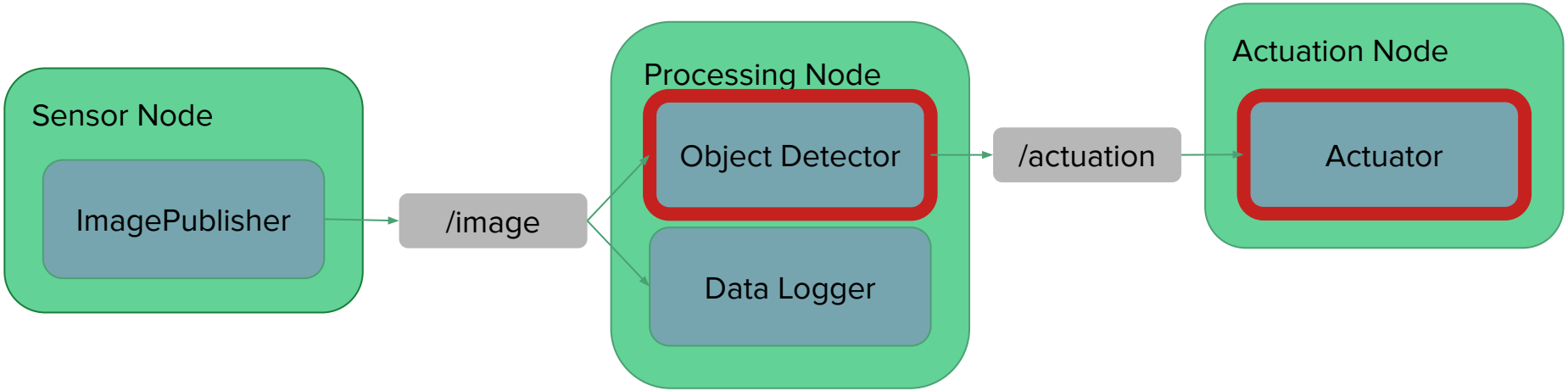
Publisher in real-time thread



Very precise

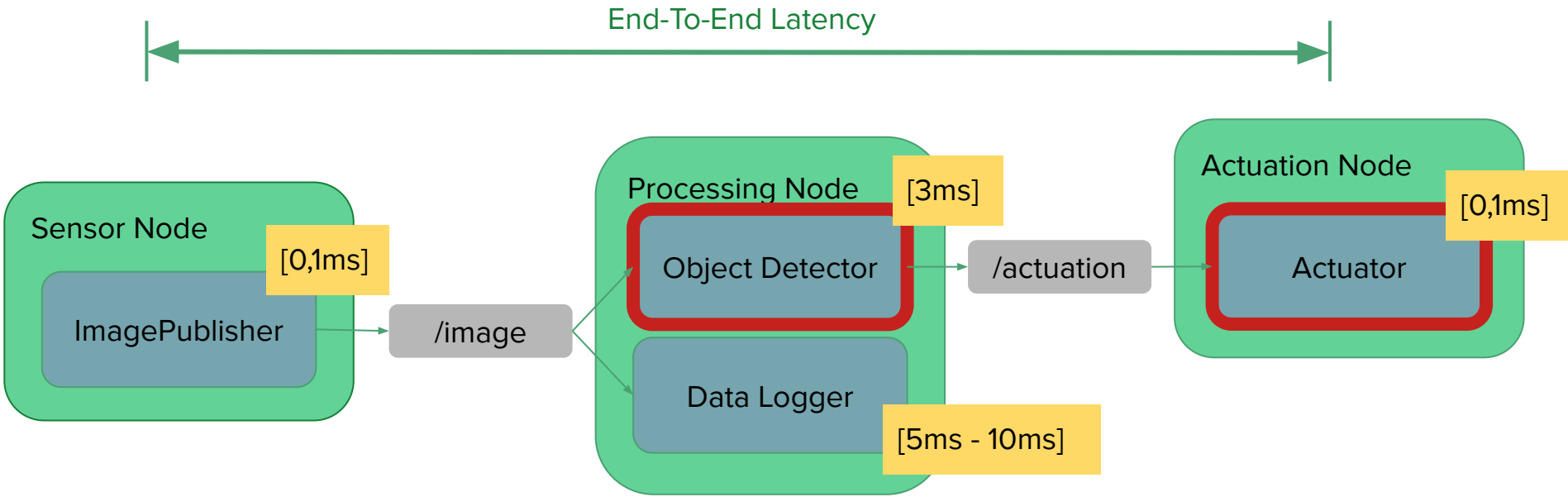
Exercise 4-2: Real-time processing chain

- Objective: Run callbacks from multiple nodes in real-time.



Exercise 4-2: Real-time processing chain

- Objective: Run callbacks from multiple nodes in real-time.



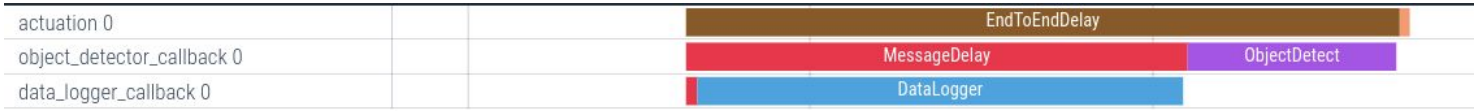
Exercise 4-2: Build and Run

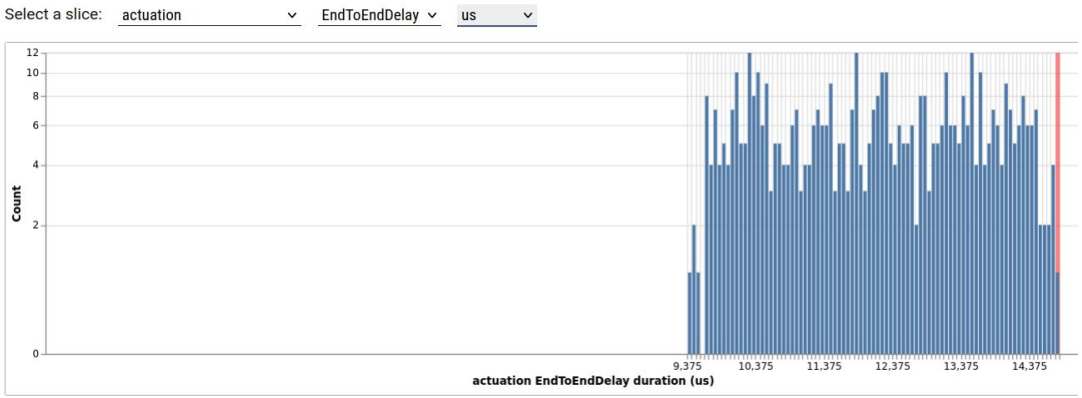
In a terminal:

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

Press Ctrl + C
Open perfetto (actuation, EndToEndDelay, us)

Exercise 4-2: Result: End-To-End Delay without Real-Time





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

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

Exercise 4-2: Code walk-through

```
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/src/application_nodes.cc

Exercise 4-2: Code walk-through

```
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_;
}
```

exercise4-2/src/camera_demo/src/main.cc

Exercise 4-2: Code walk through

```
1  int main(int argc, char** argv) {
2    // initialization of ROS and DDS middleware
3    rclcpp::init(argc, argv);
4
5    StartImagePublisherNode();
6
7    auto actuation_tracer = std::make_shared<cactus_rt::tracing::ThreadTracer>("actuation");
8    auto actuation_node = std::make_shared<ActuationNode>(actuation_tracer);
9
10   auto camera_processing_node = std::make_shared<CameraProcessingNode>(
11                                     object_detector_tracer, data_logger_tracer);
12   auto data_logger_group = camera_processing_node->get_besteffort_cbg();
13   auto object_detector_group = camera_processing_node->get_realtime_cbg();
14
15   StartTracing("camera_demo_4_2", "exercise4-2.perfetto");
16   RegisterThreadTracer(actuation_tracer);
17   RegisterThreadTracer(object_detector_tracer);
18   RegisterThreadTracer(data_logger_tracer);
19
20   rclcpp::executors::SingleThreadedExecutor real_time_executor;
21   rclcpp::executors::SingleThreadedExecutor best_effort_executor;
```

Exercise 4-2: Code walk-through

```
30  best_effort_executor.add_callback_group(data_logger_group,
31    camera_processing_node->get_node_base_interface());
32  real_time_executor.add_callback_group(object_detector_group,
33    camera_processing_node->get_node_base_interface());
34  real_time_executor.add_node(actuation_node);
35
36  // Launch real-time Executor in a thread
37  std::thread real_time_thread([&real_time_executor]() {
38    sched_param sch;
39    sch.sched_priority = 60;
40    if (sched_setscheduler(0, SCHED_FIFO, &sch) == -1) {
41      perror("sched_setscheduler failed");
42      exit(-1);
43    }
44    real_time_executor.spin();
45  });
46
47  best_effort_executor.spin();
48
49  rclcpp::shutdown();
50  StopTracing();
```

Exercise 4-2: Code walk-through

```
30  best_effort_executor.add_callback_group(data_logger_group,
31    camera_processing_node->get_node_base_interface());
32  real_time_executor.add_callback_group(object_detector_group,
33    camera_processing_nod
34  real_time_executor.add_
35
36  // Launch real-time Exe
37  std::thread real_time_t
38    sched_param sch;
39    sch.sched_priority =
40    if (sched_setschedule
41      perror("sched_setsc
42      exit(-1);
43    }
44    real_time_executor.sp
45  });
46
47  best_effort_executor.sp
48
49  rclcpp::shutdown();
50  StopTracing();
```

terminal 1:

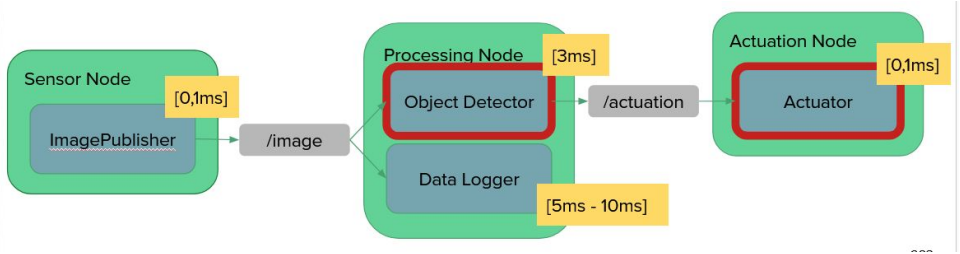
```
cp ./solutions/* ./src/camera_demo/src

colcon build
./run.sh (for 15sec)
```

Press Ctrl + C
Open perfetto
(actuation, EndToEndDelay, us)

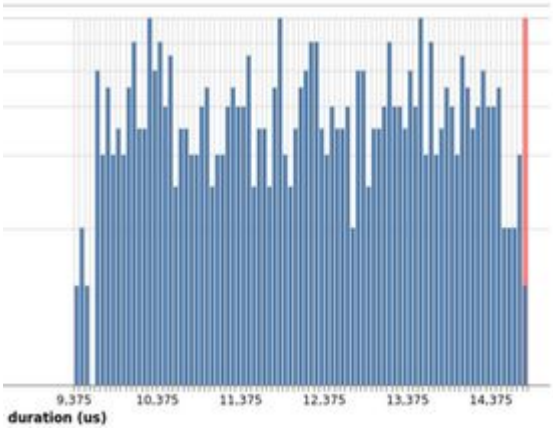
Exercise 4-2: Result: EndToEndDelay with real-time setup

EndToEnd Delay [3.1ms, 3.8ms]

Exercise 4-2: Summary

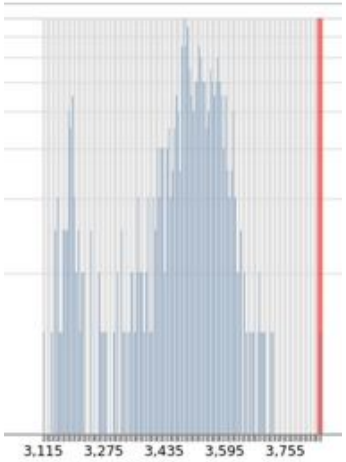


End-To-End Latency with default ROS 2



High latency

End-To-End Latency in real-time thread



Low latency

Summary

What have we discussed?

Morning

- What is “real-time”?
- Hardware, OS, application latency
- Real-time schedulers and PREEMPT_RT
- Real-time software architecture and programming techniques
- Locks and priority inversion

Afternoon

- ROS 2 Execution Management
- Impact on real-time behavior
- Overview of different Executors
- Experimental real-time Executor
- Best practices for real-time in ROS 2: thread prioritization, callback groups
- Loaned messages, zero-copy middleware
- Tooling and Benchmarks

Take away

Morning

- Ensuring real-time behavior requires ensuring every software layer is real-time.
- Real-time programming requires specialized techniques w.r.t. memory management, concurrent programming, and logging.
- Real-time threads can corporate with non-real-time ROS threads

Afternoon

- ROS 2 Execution Management has a complex semantics and is not real-time capable.
- Experimental real-time Executor proposed with simple API.
- Real-time in ROS 2 can still be achieved by using advanced features, like real-time threads, callback groups, and zero-copy memory mechanisms.