A Survey of Real-Time Support, Analysis, and Advancements in ROS 2

Daniel Casini
Scuola Superiore Sant'Anna, Pisa, Italy

Jian-Jia Chen
TU Dortmund University, Germany

Jing Li
New Jersey Institute of Technology, Newark, NJ, USA

Federico Reghenzani
Politecnico di Milano, Italy

Harun Teper
TU Dortmund University, Germany

Abstract

The Robot Operating System 2 (ROS 2) has emerged as a relevant middleware framework for robotic applications, offering modularity, distributed execution, and communication. In the last six years, ROS 2 has drawn increasing attention from the real-time systems community and industry. This survey presents a comprehensive overview of research efforts that analyze, enhance, and extend ROS 2 to support real-time execution. We first provide a detailed description of the internal scheduling mechanisms of ROS 2 and its layered architecture, including the interaction with DDS-based communication and other communication middleware. We then review key contributions from the literature, covering timing analysis for both single- and multi-threaded executors, metrics such as response time, reaction time, and data age, and different communication modes. The survey also discusses community-driven enhancements to the ROS 2 runtime, including new executor algorithm designs, real-time GPU management, and microcontroller support via micro-ROS. Furthermore, we summarize techniques for bounding DDS communication delays, message filters, and profiling tools that have been developed to support analysis and experimentation. To help systematize this growing body of work, we introduce taxonomies that classify the surveyed contributions based on different criteria. This survey aims to guide both researchers and practitioners in understanding and improving the real-time capabilities of ROS 2.

2012 ACM Subject Classification Software and its engineering → Real-time systems software; Software and its engineering → Real-time schedulability; Computer systems organization → Real-time operating systems; Computer systems organization → Robotics

Keywords and phrases ROS 2, middleware, real-time, timing predictability, publish-subscribe

Digital Object Identifier 10.4230/LITES.11.1.1

Funding This work was partially supported by: the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU; the US National Science Foundation (Grant No. CNS-2340171) and Department of Energy (Award No. DE-SC0024424); the NGI Enrichers Transatlantic Fellowship Programme and National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data and Quantum Computing;the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038; the Federal Ministry of Research, Technology and Space (BMFTR) via the *6GEM+* transfer hub under funding references 16KIS2412. This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170).

Received 2025-12-09 **Accepted** 2026-03-10 **Published** 2026-05-21

1 Introduction

The Robot Operating System (ROS) [77] has been established as one of the most popular middleware frameworks for prototyping, developing, and deploying robotics applications. With its first release in 2007, it has become more and more widespread, thanks to a modular architecture, a considerable community support, extensive libraries, and excellent interoperability. Its impact goes beyond robotics, with applications including, among others, industrial automation, smart infrastructure, and automotive [35, 50, 74, 84]. Notably, Autoware [50], the largest and most popular open-source autonomous driving framework, builds upon the ROS 2 software stack to modularly integrate perception, location, planning, and control functionalities.

After about 10 years since the first release, it became clear that the original ROS was constrained by several architectural limitations and shortcomings, which were difficult to address without a complete redesign of the framework. To overcome these issues with a completely new design, ROS 2 was released in 2017. One of the key objectives of ROS 2 was to provide support for real-time computations, thus allowing the implementation of time-critical applications [71]. Therefore, the real-time capabilities of ROS 2 have drawn increasing attention from both the academic community and industry.

Since then, the real-time systems community has intensively studied the topic of guaranteeing timing constraints under ROS 2, with particular focus on ensuring the end-to-end timing constraints of *processing chains* that may span from sensing to actuation through multiple software components, cores, and possibly distributed processing platforms. Indeed, providing such guarantees is essential for safely integrating ROS 2 applications into cyber-physical systems, which are characterized by tight interaction of the processing systems with the physical world through sensing, computation, and actuation.

Since the very first work [23], it has become evident that ROS 2 was characterized by a unique scheduling behavior that did not match any state-of-the-art scheduling algorithm previously studied in the real-time systems literature. Therefore, over the last years, the community has largely explored the topic by proposing:

- real-time analysis methods to analyze end-to-end latency metrics such as response times, reaction times, and data age of ROS 2 applications, which are the key building blocks to enable principled design-space exploration without needing the full prototyping and deployment of the system;
- customized schedulers, aimed at improving the timing predictability of the default ROS 2 scheduling algorithm;
- tools to help the development of real-time applications using ROS 2, as well as system-level enhancements to support the interaction between ROS 2 and hardware accelerators or other frameworks; and
- techniques for bounding ROS-related communication delays (e.g., DDS), message filters, and profiling tools that have been developed to support analysis and experimentation.

After more than 6 years of work by the real-time systems communities towards these goals, we believe it is essential to understand what we have accomplished and how far we are from achieving the ambitious goal of supporting real-time computing in ROS 2.

This survey aims to guide both researchers and practitioners in understanding the literature about the real-time capabilities of ROS 2. To help systematize this growing body of work, several taxonomies have been introduced in this survey to classify the surveyed contributions based on different criteria, including scheduling strategies, communication mechanisms, and timing models.

Survey structure

The discussion of this survey is divided into the following parts: Section 2 summarizes the necessary background on the real-time aspects of ROS 2, covering its internal scheduling mechanisms, layered architecture, and interaction with DDS-based communication and other middleware backends; Section 3 discusses the articles on scheduling analyses on native ROS 2. Section 4 discusses novel approaches that extend the ROS 2 scheduling approaches and introduces GPU and latency management techniques; Section 5 recaps the papers dealing with tools, profiling, and empirical analyses; Section 6 concludes the paper.

2 Background

We start the survey by providing the necessary background. In particular, Section 2.1 summarizes the layered architecture of ROS 2, discusses its scheduling mechanism, and reports the differences between the different executor types natively available in ROS 2. Section 2.2 reports some relevant information regarding the interaction of ROS 2 with lower-level middleware that are used to dispatch messages, such as the DDS, and describes the *micro-ROS* initiative, aimed at using ROS 2 on microcontrollers. Finally, Section 2.3 briefly summarizes some approaches that have been presented in the state-of-the-art to improve the real-time performance of ROS 1, so as to provide a historical note.

2.1 ROS 2

In this section, we provide an overview of the ROS 2 architecture, the system design, and the basic ROS 2 scheduling and communication mechanisms.

Overview. As shown in Figure 1, ROS 2 acts as a middleware between the application and the operating system, providing a set of tools and libraries to build robotic applications, as well as interfaces for inter-process and inter-machine communication. ROS 2 itself is composed of several layers that enable its modular architecture. At the top, there are programming-language-specific layers: the two most popular ones are `rclpy` for Python and `rclcpp` for C++, respectively. The literature on real-time performance of ROS 2 assumes the use of the C++ interface, since C++ is best suited for time-critical applications. Below, the `rcl` layer provides the core functionalities of ROS 2 and the `rmw` layer provides a unified API for the underlying middleware used for message dispatching (DDS [76] or Zenoh [25]).

ROS 2 applications are realized through a distributed network of components, as illustrated in Figure 2. Each component is realized as a node, which is the fundamental unit of composition in ROS 2 systems and encapsulates a set of related functionalities. The functionalities of a node are implemented as **callbacks**, which are functions that are executed in response to specific events. These events are triggered by different task types, such as subscribers that receive messages (*event-driven*) or timers that elapse after a specified time interval (*time-driven*).

In ROS 2, nodes can communicate with each other using the *publish-subscribe* paradigm. Each node can include publishers and subscribers that allow them to communicate via messages through so-called *topics*. When a callback *publishes* on a topic, it sends a message in broadcast to all the *subscribers* that subscribe to the same topic. The reception of the message triggers a corresponding activation, which eventually leads to the execution of the subscriber callback.

Another mechanism to activate callbacks is via *timers*, which trigger the execution of a callback periodically, according to a specified time interval. Furthermore, ROS 2 also provides a mechanism for remote procedure calls (RPCs) through *services*. These feature a client-server architecture, where a client callback sends a request to a service server, which processes the request and returns a response, which is then processed by a client callback, as shown in Figure 2.

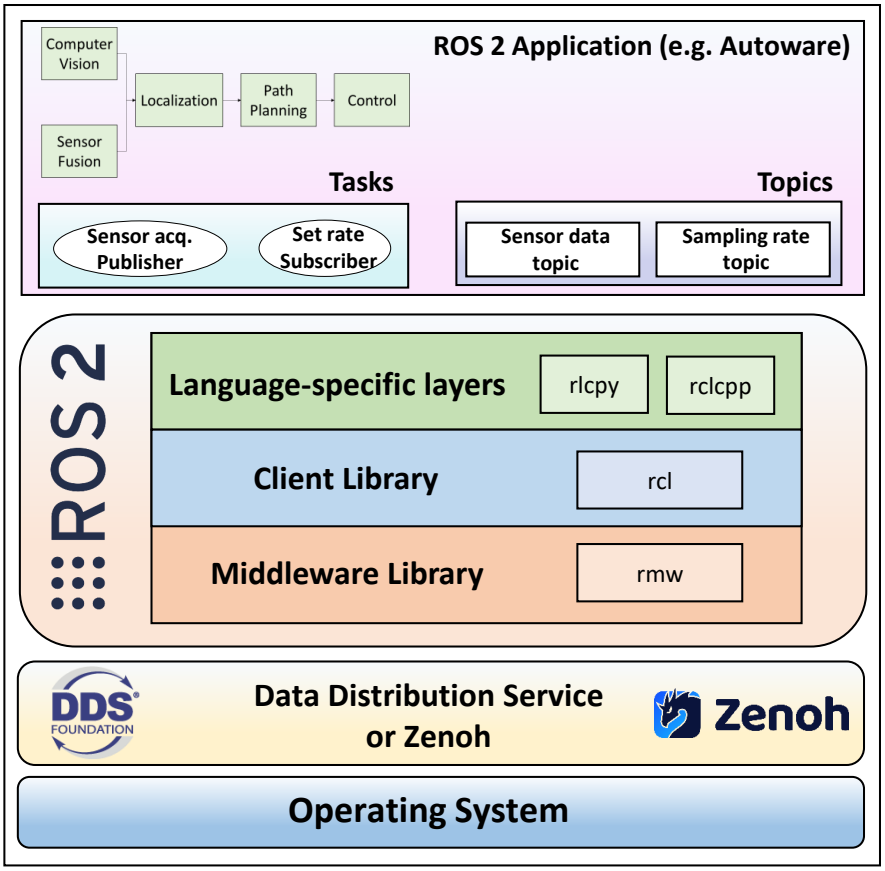

Figure 1 ROS 2, with its layers and interactions.

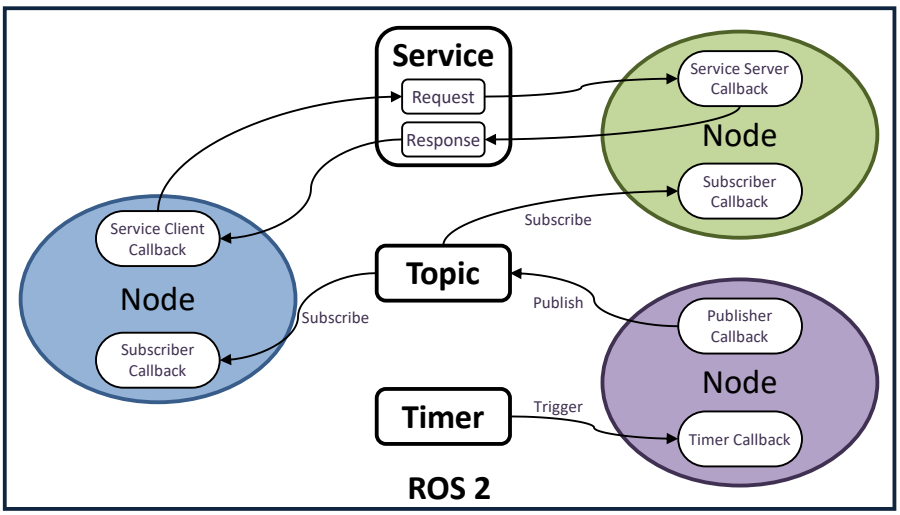

Figure 2 ROS 2 components.

Finally, ROS 2 also includes the concept of *waitables*, a general mechanism for integrating custom asynchronous events into the executor. Waitables enable developers to define new sources of execution that can be triggered by external events, such as file descriptors or other system-level notifications, and can be seamlessly integrated in ROS 2 alongside traditional mechanisms.

Scheduling in ROS 2. In ROS 2, the callbacks are computational activities of interest for bounding real-time performance metrics, such as response times and end-to-end latencies. However, callbacks are functions dispatched by the ROS 2 middleware and are subject to a two-level scheduling hierarchy.

On the one hand, callbacks are scheduled by ROS 2 threads, which implement their own scheduling mechanism at the level of C++ functions, as extensively discussed in the following. On the other hand, ROS 2 threads are scheduled by the underlying Operating System (OS), which most commonly is Linux[1].

The scheduling behavior of OS-level schedulers has been largely studied for several decades. For example, both Linux and QNX provide priority-based schedulers and reservation-based schedulers, for which the timing behavior was largely explored [26, 58]. Conversely, ROS 2 implements custom internal scheduling mechanisms that have only been studied since 2019 [23]. As detailed below, the scheduling logic of ROS 2 is significantly different from any other scheduler that has been studied before. In ROS 2, the scheduling algorithm is implemented within the *executor*, the C++ function dispatcher at the middleware level. Three default executor algorithms are implemented in the standard distribution: **(1)** the single-threaded executor, **(2)** the multi-threaded executor, and **(3)** the static single-threaded executor. The static single-threaded executor is a variant of the single-threaded executor, optimized for reduced overhead, in which the dynamic task addition is not supported.

Single-threaded executor in ROS 2. The single-threaded executor is the default callback scheduling mechanism in ROS 2. Its behavior, outlined in Algorithm 1, differs significantly from classical priority-based schedulers. Instead of managing a global list of all ready callbacks, it manages a local set, referred to as *wait set*, that is periodically updated from the underlying layers. The executor's core logic to update the wait set revolves around a repeating cycle of two phases: polling and processing.

Polling and Processing Cycle. The time between two polling events is known as *processing window* [23]. During a processing window, the executor executes callbacks from its local wait set. When the wait set becomes empty, a *polling point* occurs. At this point, the executor repopulates its wait set by querying the underlying layers for all callbacks currently ready (Algorithm 1, line 5). All callbacks that follow this update behavior are typically called *polled callbacks* [19].

Once the wait set is populated, the executor begins executing the stored callbacks sequentially. The selection follows a fixed-priority, non-preemptive policy based on a two-level hierarchy:

1. **Callback Type:** Callbacks are processed in a predefined order, thus implementing an implicit per-type priority ordering: timers, subscriptions, service handlers, service clients, and waitables (Algorithm 1 Line 7).
2. **Registration Order:** Callbacks of the same type are prioritized depending on the order in which they were created and added to the executor.

[1] Other OSes, such as QNX and VxWorks, FreeRTOS, are also supported.

Algorithm 1 Single-threaded Executor Scheduling Pseudocode.

```
1:  function SingleThreadedExecutorScheduler
2:      Initialize wait set as empty set
3:      while true do
4:          if wait set is empty then
5:              wait set ← PollCallbacks()                    ▷ Polling point
6:          for all callback in wait set ordered by priority do
7:              ▷ Priority order is: timers, subscriptions, service requests, service reply, waitables
8:              execute(callback)                    ▷ Callbacks executed non-preemptively
9:              remove(callback) from wait set
10: function PollCallbacks
11:     Initialize newWaitSet as empty set
12:     for all callback c_i in readyCallbacks do ▷ query readyCallbacks from the underlying layer
13:         if newWaitSet does not contain callback c_i then
14:             Add(callback) to newWaitSet                   ▷ Ensure unique callbacks in wait set
15:     return newWaitSet
```

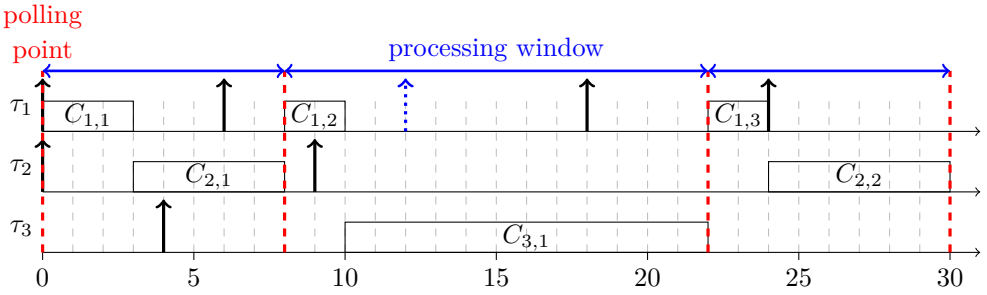

Figure 3 ROS 2 polling and processing cycle. Polling points are represented with vertical red dashed lines. The processing window length is instead represented with horizontal, double-arrowed, blue lines.

Once selected, a callback is executed to completion non-preemptively (Algorithm 1 Line 8). However, the executor's thread itself can still be preempted by the underlying operating system.

Figure 3 shows an example in which there are three callbacks: c_1, c_2, and c_3. At the beginning, only c_1 and c_2 have ready callbacks (i.e., $c_{1,1}$ and $c_{2,1}$). Hence, they are sampled at the first polling point, they are added to the wait set, and they are executed. Within the first processing window, callback instances $c_{1,2}$ of c_1 and $c_{3,1}$ of c_3 are released, but they are sampled only at the second polling point, thus being added to the wait set. In the second processing window, $c_{1,2}$ is executed first, followed by $c_{3,1}$. During the second processing window, two releases of c_1 and one release of c_2 occur, which are then executed in the third processing window.

Importantly, there are two critical aspects of this design that significantly affect the timing behavior of ROS 2 applications, (1) delays of lower-priority callbacks and (2) loss of releases.

- If we assume lower indices to correspond to higher priorities, the polling-point mechanism causes a more serious issue in the second processing window. Indeed, instance $c_{2,2}$ is released during $c_{1,2}$'s execution; however, since $c_{2,2}$ was not ready yet at the previous polling point, it is not present in the wait set; and, when $c_{1,2}$ finishes, $c_{3,1}$ with a lower priority is selected to run instead. Since another instance of c_1 is released in the meantime, with higher priority, it further delays the execution of $c_{2,2}$, which is executed only later in the third processing window.

- The wait set is fundamentally different than list-based ready queues used in classical real-time scheduling theory. In particular, the wait set enforces *uniqueness* of callbacks (Algorithm 1 Line 14), i.e., it can only contain at most one instance of each ready callback in any processing window. This means that at each polling point, *at most one instance* of each callback can be added to the wait set, regardless of how many events of a specific callback (i.e., messages for subscriptions or timer elapses) have occurred for it since the last polling point (Algorithm 1 Line 14). For instance, callback C_1 in Figure 3 arrives twice during the second processing window, but only one instance $C_{1,3}$ is added to the wait set and executed during the third processing window.

Historical Context for Polling. As highlighted by Teper et al. [100], the polling mechanism of ROS 2 is significantly different from classical scheduling theory of periodic tasks, which assumes every job will be executed. Indeed, for ROS 2 versions after "Dashing", timer callbacks are be skipped if the executor detects that the time between the activation time of the timer callback's previous job and the start time of its next job is longer than the period of the timer callback itself.

Compared to classical real-time scheduling, ROS 2 has the unique aspect of handling multiple types of callbacks, each with its own activation mechanism, including time-triggered and event-triggered ones. Finally, ROS 2 also needs to provide a scheduling solution that is both capable of handling real-time critical tasks, as well as being flexible and easy to use for a wide range of robotic applications. Thus, finding the right balance between real-time guarantees and usability has been a key design goal of ROS 2.

The exact behavior described in this section about the executor describes the behavior during the releases from 2020 (Foxy) to 2024 (Iron). It is worth noting that this behavior has not always been consistent across all callback types. In earlier ROS 2 distributions (up to "Dashing"), timer callbacks were treated as *privileged callbacks* [17], i.e., they were not subject to this polling mechanism and executed normally. Some of these cross-version changes are documented in the paper by Dust et al. [34], who have conducted experiments comparing the behavior of ROS 2 Dashing to later versions. They show that the timer execution can be significantly delayed in newer ROS 2 distributions compared to ROS 2 Dashing due to timers being subject to the polling point and processing window approach. This timer issue could be alleviated by using a multi-threaded executor with dedicated executor threads for timers. However, for complex systems, this approach may not always be feasible.

Furthermore, there have been recent developments in the distribution ROS 2 Jazzy towards completely removing the ordering inside the wait set and instead following a FIFO order [83]. It remains uncertain if this behavior will be adopted and kept in future ROS 2 distributions.

Multi-threaded executor. The multi-threaded executor extends the polling-based principle of its single-threaded counterpart to enable parallel execution. It utilizes a pool of threads to process callbacks concurrently. While the fundamental process of polling for ready callbacks and populating a shared wait set remains, the multi-threaded executor can distribute these polled callbacks among its available threads, allowing them to be executed in parallel.

Concurrency Management with Callback Groups. To manage concurrency and prevent data races when parallel execution is not desirable, ROS 2 introduces the concept of **callback groups**. Every callback within a node is assigned to a specific callback group, which affects its execution policy. There are two types of callback groups:

- **Mutually Exclusive:** Callbacks within this type of callback groups are guaranteed not to execute in parallel with each other. The executor only schedules one callback from the group at a time, effectively serializing their execution, much like the single-threaded executor.

- **Reentrant:** This group type allows multiple callbacks assigned to it to be executed concurrently by different threads. It even permits multiple instances of the same callback to run in parallel if possible, e.g., if it is triggered multiple times in quick succession.

The scope of these rules is the group itself. Callbacks belonging to different callback groups or to different nodes can always be executed in parallel, regardless of their group type, if they are assigned to the same executor. By default, all callbacks in a node are assigned to a single, default callback group that is mutually exclusive, unless explicitly configured otherwise. An improper assignment of callbacks to callback groups can lead to performance bottlenecks in a multi-threaded context, as callbacks that could safely run in parallel are instead forced to wait. For performance-critical applications, developers should strategically assign callbacks to different groups or use reentrant groups to maximize parallelism and avoid unnecessary delays.

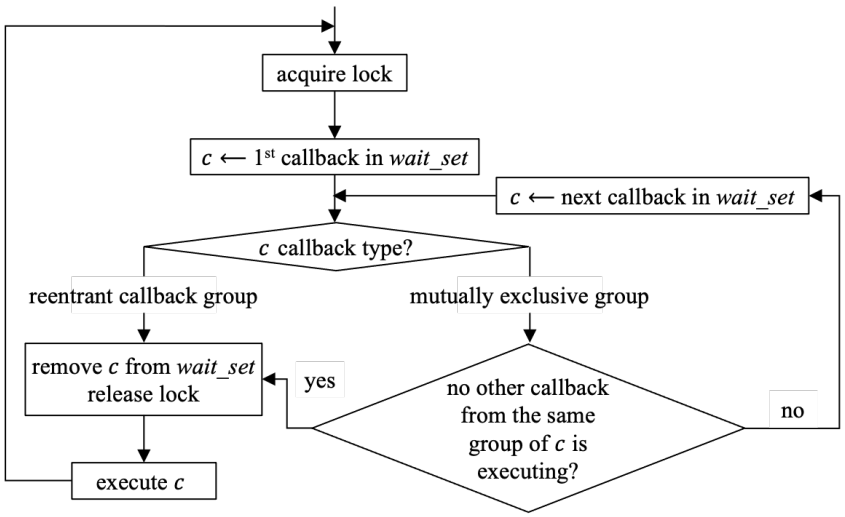

Figure 4 Thread workflow in a multi-threaded executor.

Callback Group-Aware Polling. The multi-threaded executor coordinates its thread pool using locking mechanisms to ensure thread-safe access to the shared wait set. Figure 4 summarizes the behavior of such a mechanism. As depicted in Figure 4, a worker thread must first acquire a lock to gain exclusive access to the wait set. Once it holds the lock, the thread searches for a task to execute in the wait set, following the same priority order as the single-threaded executor, but including the callback group constraints. A ready callback in a reentrant callback group can be selected freely, while one in a mutually exclusive group can only be chosen if no other callback from that same group is currently executing. After selecting a callback, the thread releases the lock and begins execution of the callback. If no executable callback is found, the thread that acquired the lock waits for new callbacks to get ready, then fills the wait set again, and repeats the selection process.

2.2 Beyond ROS 2

Next, we discuss the lower-level middleware used by ROS 2 to handle publish/subscribe communications and variants of ROS 2 for microcontrollers.

Lower-level Pub/Sub middleware: the DDS and Zenoh. The inter-process communication in ROS 2 is built upon the Data Distribution Service (DDS). DDS is an open standard from the Object Management Group (OMG) that specifies a data-centric, publish-subscribe middleware protocol. Its adoption in ROS 2 was driven by its proven track record in mission-critical distributed systems such as aerospace, autonomous vehicles, robotics, Internet of Things (IoT), and Industry 4.0/5.0 [39, 105, 119]. Furthermore, it is also compatible with the AUTOSAR Adaptive standard [8] for automotive systems.

ROS 2 interfaces with the underlying communication system through a dedicated abstraction layer, the ROS Middleware Interface (*rmw*). This design decouples the ROS 2 client libraries from the specific middleware, allowing developers to choose the DDS implementation best suited for their application. ROS 2 supports multiple DDS implementations, each with a corresponding rmw package. The primary options are DDS-based and involve *FastDDS*, *CycloneDDS*, *GurumDDS*, and *ConnextDDS*. Beyond DDS, ROS 2 is also compatible with newer protocols like Zenoh [25], accessible via the *rmw_zenoh package*. Zenoh provides a more lightweight publish-subscribe and distributed query protocol.

The choice of middleware has significant implications for system performance and real-time behavior. Because each implementation has its own internal architecture, threading model, and configuration parameters, a generic timing analysis is insufficient. For example, Sciangula et al. [88] provided a compositional model that generalizes beyond the specific implementation, taking into account the common aspects. Additionally, they provide a concrete instantiation of the model for FastDDS to devise a response-time analysis. This work is further extended in [89] to also support all the policies of FastDDS and to include multiple communication modes, as well as integrating the analysis with ROS 2. Understanding and modeling the specific timing properties of the chosen middleware is crucial for accurately predicting end-to-end latencies and ensuring that real-time requirements are met.

To illustrate the complexity of the DDS configuration, we now consider FastDDS. One of the most critical settings is the message dispatch mode, which dictates how data is sent. There are two modes:

- **Synchronous**: The publishing ROS 2 callback directly handles the message-sending operation, blocking its thread until the message publication is complete.
- **Asynchronous**: The message-sending operation is offloaded to a dedicated service thread, allowing the publishing callback to complete without waiting for the message to be sent. In FastDDS, this service thread is called the flow-controller thread.

We illustrate these behaviors using the example system in Figures 5 and 6, which consist of a callback C_1 that publishes three messages to a subscriber callback C_2 via DDS. Each callback is run by its own executor. We assume that the executor of C_2 starts after time point 6, and that each job of callback C_1 publishes at the end of its execution.

Figure 5 illustrates the timing behavior of DDS-based synchronous communication in ROS 2. As soon as a job of callback C_1 finishes its publish operation, the message is guaranteed to be stored in the buffer of C_2. Since messages may arrive while the executor of C_2 is already processing a previous message, the buffer stores multiple messages that are then processed in FIFO order, one per processing window.

Figure 6 illustrates the timing behavior of DDS-based asynchronous communication for the same scenario. In contrast to the synchronous case, the publishing callback C_1 does not wait for the message to be sent. Instead, the message is handed off to the flow-controller thread, which handles the actual transmission. As a result, the execution time of each C_1 instance is shorter, since it no longer includes the time spent on the publish operation. However, the subscriber's buffer is only updated once the flow-controller thread has completed the transmission, which may

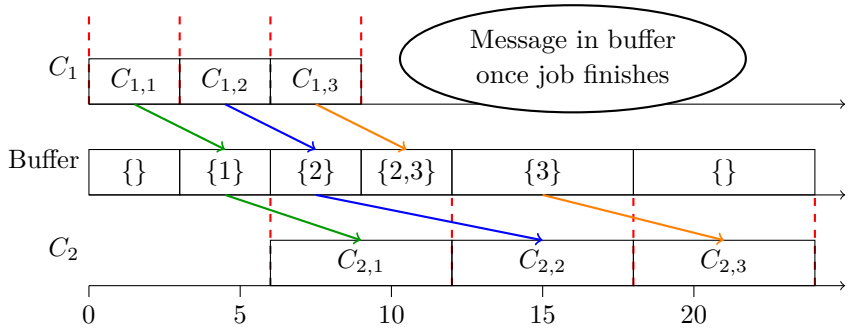

Figure 5 DDS-based synchronous inter-node communication.

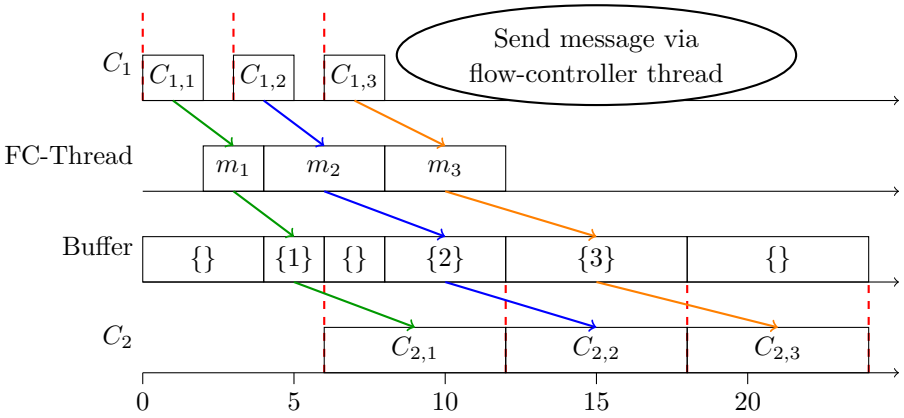

Figure 6 DDS-based asynchronous inter-node communication.

introduce additional latency compared to the synchronous mode. This is illustrated between time points 6 and 8, in which the buffer is empty, while message m_2 is still being processed by the flow-controller thread.

Moreover, this flow-controller thread can itself be configured with different scheduling policies, such as `HIGH_PRIORITY` (i.e., fixed-priority), `FIFO`, or `ROUND-ROBIN`. The choice between synchronous and asynchronous mode, combined with the scheduling policy of the flow-controller thread, can substantially impact the overall latency and determinism of message delivery in ROS 2 systems. Furthermore, on the receiver side, messages are typically delivered by a DDS listener thread to the ROS 2 executor in FIFO order, adding another stage to the communication pipeline that must be considered in a full system analysis.

micro-ROS. micro-ROS is the official extension of ROS 2 designed to run on resource-constrained microcontrollers (MCUs). Primarily developed by eProsima, Bosch, the FIWARE Foundation, PIAP, and Acutronic Robotics, with support from European research projects such as H-ROS [107] and OFERA [1], its goal is to bring the standard ROS 2 APIs and tools to embedded systems.

To operate within the limited memory and processing power of an MCU, micro-ROS employs a client-agent architecture.

- **micro-ROS Client**: The micro-ROS client is a lightweight library that runs on the MCU. It implements the ROS 2 APIs but does not connect directly to the global data space.
- **micro-ROS Agent**: The micro-ROS agent runs on a more powerful machine (e.g., a single-board computer) and acts as a proxy. It connects to the main ROS 2 network via the standard DDS protocol.

Communication between the client and agent uses the DDS-XRCE (eXtremely Resource-Constrained Environments) protocol, which is a lightweight, client-server protocol specifically designed for embedded systems with limited resources. The agent effectively bridges the resource-constrained device into the ROS 2 ecosystem.

The scheduling of callbacks in micro-ROS is designed for predictability and minimal overhead, critical for embedded applications. The default executor provided by the `rclc` library (C-language client library), referred to as the `rclc` executor, operates as a simple, non-preemptive, static-priority scheduler. Its execution logic is as follows:

- **Static Ordering**: Like ROS 2 itself, micro-ROS also features timer-based and event-based triggering mechanisms through timers and subscriptions to initiate the executor's processing cycle. However, callbacks (e.g., timers, subscriptions) are executed in a fixed, sequential order based on the user-assigned priorities, making the system's behavior more predictable.
- **Subscription Semantics**: For the trigger condition of subscriptions, the developer can specify *execution semantics*, configuring if the callback of a subscription should run only if new data is available (`ON_NEW_DATA`) or if it should run in every cycle (`ALWAYS`), which is useful for polling or fixed-rate tasks.
- **Triggered Execution**: On top of the basic triggering mechanisms, the executor's processing cycle can also be initiated by a trigger condition. This allows for the implementation of complex, event-driven control logic. Key trigger mechanisms include the following policies:
  - **ANY**: Starts execution if at least one callback is ready.
  - **ALL**: Starts execution only if all specified callbacks are ready.
  - **ONE**: Starts when a message for a specific callback arrives.
  - **USER-DEFINED**: Allows custom triggering logic, including hardware interrupts.
- **Data Consistency Semantics**: To support time-triggered, real-time systems, the executor offers Logical Execution Time (LET) [44] semantics. When enabled, the executor reads and buffers all available inputs at the beginning of a cycle. All callbacks within that cycle then operate on this consistent, timestamped snapshot of data. This contrasts with the default ROS 2 semantics, where the data of a callback is fetched from the queue immediately before its execution.

For applications requiring concurrency, the `rclc` executor also provides an experimental multi-threaded model. In this architecture, a main executor thread monitors for incoming data and handles the trigger conditions. For each callback, a dedicated worker thread is spawned with configurable scheduling parameters provided by the underlying RTOS. The main executor thread then dispatches new message data to the appropriate callback function, which executes in its dedicated worker thread. This design allows developers to leverage native RTOS scheduling features such as thread priorities, scheduling policies, and advanced scheduling algorithms (including reservation-based scheduling) to manage real-time callback execution.

2.3 Overview of the ROS 1 Approaches for Real Time

A primary motivation for the development of ROS 2 was the inherent difficulty of achieving real-time performance in its predecessor, ROS 1 [51, 71]. By design, ROS 1 nodes are implemented as standard operating system threads, meaning their execution is delegated entirely to the underlying Linux scheduler. In typical installations, this means relying on general-purpose schedulers like the *Completely Fair Scheduler (CFS)* [78] or the more recently *Earliest Eligible Virtual Deadline First (EEVDF)* [94], which are optimized for general-purpose systems.

While the Linux kernel has gained powerful real-time capabilities, e.g., through tools for its fine-grained configuration [29, 30], the PREEMPT_RT patch [27, 28], and the SCHED_DEADLINE scheduler [56, 58], ROS 1 itself lacks the internal architecture and APIs needed to effectively leverage them. In contrast, the design of ROS 2 aims at explicitly targets real-time support through a modular execution model, DDS-based communication, and configurable DDS QoS policies [81]. The fundamental design limitation of ROS 1 prompted numerous research efforts to retrofit real-time capabilities onto the ROS 1 framework, which can be broadly categorized into two main strategies.

The first strategy involved architecturally isolating critical tasks using a co-kernel or multi-kernel approach to bypass the standard Linux scheduler. For example, Delgado et al. [31] used the Xenomai real-time co-kernel to execute core ROS 1 components as high-priority tasks alongside the main OS. A similar concept was explored by Wei et al. with RGMP-ROS [111] and its multi-core evolution, RT-ROS [112], which dedicated specific CPU cores to a real-time operating system to manage time-sensitive ROS nodes.

A second strategy focused on building scheduling frameworks on top of a standard Linux system. A notable example is the ROSCH framework [85], which introduced a DAG-based scheduling system, synchronization mechanisms, and fail-safe functions to manage ROS 1 nodes, later extending support to GPUs with ROSCH-G [96].

Ultimately, these approaches were workarounds for the core issue: ROS 1 was not designed with real-time performance in mind. The absence of proper APIs and internal structures to manage timing constraints meant that scientific contributions often focused on empirical latency measurements and mitigation, as exemplified by tools developed by Nishimura et al [75], rather than providing a formal scheduling analysis with provable guarantees. This gap was a key driver for the fundamental architectural redesign that led to ROS 2.

3 Analysis of Real-Time Behavior in ROS 2

We start the survey by summarizing the papers on the real-time analysis of ROS 2. Achieving predictable real-time performance in ROS 2 requires a deep understanding of the various factors that contribute to the real-time behavior. Over the past several years, a significant body of research has emerged to formally analyze these factors from different perspectives. This section provides a comprehensive overview of these analytical efforts. We begin by examining the core of ROS 2's internal scheduling logic, covering works that model the behavior of both the *single-threaded* (Section 3.1) and *multi-threaded* executors (Section 3.2). We then explore studies that focus on the communication stack, discussing techniques for bounding delays introduced by lower-level middleware like DDS and specialized message synchronization filters in Section 3.3. Subsequently, we explore an alternative approach based on *formal verification and model checking* in Section 3.4, which explores the verification of timing properties in a different way. Finally, a *taxonomy* is presented in Section 3.5 to systematically classify and compare the different analytical approaches discussed throughout this section.

3.1 Analysis of the Single-Threaded Executor

The first ROS 2 paper in real-time systems. The first work on the real-time analysis of ROS 2 systems was published by Casini et al. [23]. In their paper, they described the single-threaded executor algorithm for the first time[2] and provided a system model suitable for analyzing ROS 2

[2] The ROS 2 documentation now reports the behavior of the executor algorithm, based on [23], see `https://docs.ros.org/en/kilted/Concepts/Intermediate/About-Executors.html`

processing chains, which are sequences of callbacks that communicate (via topics) across multiple executors, spanning from sensing to actuation. They modeled these chains as a directed acyclic graph (DAG) of callbacks to capture intersection and complex data flows [23].

To analyze this model, the authors adopted *Compositional Performance Analysis (CPA)* [43], a framework for analyzing complex systems by examining individual components in isolation. In their mapping, ROS 2 executors act as components that supply CPU time, while callbacks are computations that consume it. This required modeling three key aspects:

- **Available CPU Resources:** To abstract away the OS-level scheduler, the authors of [23] used the *supply-bound function* [22, 64, 90] abstraction, denoted as $sbf_k(\Delta)$, which represents a lower bound on the processing time an executor k receives in any given time interval of length Δ, which can be derived for reservation-based schedulers like Linux's SCHED_DEADLINE [58].
- **Workload Arrival:** Callback releases were modeled using arrival curves, denoted as $\eta_i(\Delta)$, which upper-bounds the number of releases of a callback c_i in any interval of length Δ. Arrival curves for initial "source" callbacks (e.g., timers) are provided, while curves for subsequent callbacks are systematically derived by means of the arrival-curve propagation process [43].
- **Workload Execution:** The execution time of each callback was modeled using the classical *worst-case execution time (WCET)* abstraction [113].

Based on this, the paper derived a formal model and *response-time analysis* for processing chains. The analysis supports arbitrary DAG structures, including callbacks that are triggered by messages from multiple topics (an *OR* activation semantic), and accounts for external event sources like device drivers. Their baseline method computes the response-time bound by summing the individual response-time bounds of each callback in a chain.

The core of their analysis is a search for the worst-case scenario within a so-called "busy window", which can be informally defined as a period of continuous processing. The interested reader in the correct and formal definition is left to the original paper. The response-time analysis presented in the paper builds on computing the response-time of arbitrary callback instances released at an arbitrary point in time A within the busy window. The per-callback response time is then computed as the maximum among all possible instances released at arbitrary values of A. Clearly, this requires bounding and discretizing the space of possible time instants A.

First, instead of checking over an unbounded time interval (i.e., $A \geq 0$), the paper restricts the search space to the interval $[0, L)$, where L is an upper bound on the length of a busy window. Second, it discretizes this interval into a finite set of evaluation points, making the problem computationally feasible. The technique builds on previous work on the abstract response-time analysis framework [21].

Finally, the authors proposed an improved holistic analysis to mitigate pessimism from the "pay-burst-only-once" problem [20, 86] for subchains with a single executor. While this work laid the essential foundation for all subsequent ROS 2 analysis, its model only captured the negative scheduling effects of the single-threaded executor, paving the way for future refinements.

"Second-generation" analysis methods. Building on the foundational work from Casini et al. [23], subsequent research introduced more precise analysis methods. These "second generation" works focused on refining the model to capture more nuanced behaviors of the single-threaded executor, leading to tighter and more accurate latency bounds.

A key improvement by Tang et al. [97] focused on optimizing *linear processing chains*, i.e., chains that cannot include branching and each callback can belong to only one chain. Their work improved analysis precision for these common structures and introduced a priority assignment strategy. It demonstrated that a chain's end-to-end latency is primarily affected by the priority of its final (sink) callback, leading to the recommendation of promoting sink priorities.

In parallel, Blass et al. [19] enhanced the analysis for *arbitrary DAGs* by incorporating two novel concepts:

- **Execution Time Curves:** Instead of relying on a single WCET, they used execution time curves, introduced by Quinton et al. [82], which model the cumulative execution demand over multiple job releases. This better captures the variability in callback execution times common in real-world applications.
- **Starvation-Freedom Property:** They formally modeled the executor's round-robin-like nature, which prevents high-priority callbacks from indefinitely starving low-priority ones. While the original analysis treated this behavior as a source of pessimism for high-priority tasks, this work also modeled its positive impact on lower-priority callbacks, resulting in a more balanced and accurate system-wide analysis.

This approach was validated using callback graphs extracted from the real-world Turtlebot3 navigation stack. Later, Tang et al. [98] further refined the analysis for DAGs by drawing deeper comparisons between the ROS 2 executor and classical round-robin scheduling, leveraging graph decomposition techniques to better characterize workload interference.

Analysis of Reaction Time and Data Age. In addition to the traditional response-time metrics, recent research has introduced a shift to metrics that focus on data propagation for ROS 2. This line of research was initiated by Teper et al. [103], who expanded the system model to cover more diverse communication patterns. Beyond the typical publish-subscribe mechanism (termed *inter-node* communication), their work also formalized *intra-node* communication, where callbacks exchange data by storing it within the same node, and *external* communication, where callbacks interact with external data interfaces. Critically, they focus on two new performance metrics:

- **Maximum Reaction Time (MRT):** The latency from an external event to the system's corresponding reaction.
- **Maximum Data Age (MDA):** The data freshness, in which the latency is determined backward from the actuation to the cause that originated it.

This new focus was supported by a more general cause-effect chain model that allowed chains to start with a timer, with following callbacks being either subscriptions or timers, but not allowing consecutive timers. Despite these advances, this initial analysis was constrained to a single executor, excluding chains that span multiple executors and thus not supporting distributed systems. Furthermore, it does not incorporate OS-level scheduling overheads, nor support for supply-bound functions to abstract the underlying OS scheduling.

The support for multi-executor systems has been introduced in a later work by the same authors [101]. The work provided a formal upper-bound analysis for both MRT and MDA in these distributed setups, additionally lifting the prior restriction against having consecutive timers within a chain. Beyond the analysis, the paper also introduced an optimization framework using constrained programming to minimize end-to-end latencies. This framework automatically configures system parameters such as node-to-executor assignments, DDS communication modes (synchronous or asynchronous), task prioritization policies, and timer periods. Evaluated on an autonomous racing software stack, the optimization reduced the analytical upper bound by up to 50.2% and the maximum measured latency by 19.8%.

Focusing more on fundamental theory of end-to-end latencies, Günzel et al. [40] explored the precise relationship between the maximum reaction time (MRT) and maximum data age (MDA) metrics that Teper et al. analyzed. The authors formally proved that, after an initial system warm-up period, MRT and MDA are equivalent. Crucially, they demonstrated this equivalence holds under very general and non-restrictive assumptions, applying to a wide variety of systems including those with over- and undersampling, various scheduling policies (e.g., preemptive or non-preemptive, static-priority or EDF), and distributed setups. Furthermore, they provide a case study about a ROS 2 system that shows that this condition also experimentally holds.

Orthogonally, Teper et al. [102] addressed the impact of system architecture on performance issues like message loss and high end-to-end latencies. The authors provided two key analytical bounds. First, they derived a bound for timer periods to prevent sensor undersampling, where a timer fails to process all available sensor data due to interference from other callbacks on the same executor. Second, they analyzed subscription buffers, concluding that a buffer size of two is sufficient to prevent message loss for a subscription with a single synchronous publisher. They also showed that with proper callback prioritization, this could be reduced to one. Based on these findings, the paper proposed a heuristic for callback prioritization designed to reduce both buffer sizes and end-to-end latencies. To enable this heuristic, the authors suggested a minor architectural change to the ROS 2 executor: prioritizing subscription callbacks over timer callbacks by swapping their sampling order. The evaluation of these changes demonstrated a reduction in end-to-end latencies by up to 40%.

The impact of different data handling semantics was explored by Tang et al. [99], who focused on *data refreshing*, where limited buffer sizes cause fresh data to overwrite older messages. They noted that while this reduces the overall workload, it can counter-intuitively worsen end-to-end latency in some scenarios compared to the standard First-In-First-Out (FIFO) approach, necessitating a dedicated analysis. The paper provides a formal timing analysis to calculate upper bounds for both maximum reaction time and data age, specifically for chains using data refreshing, considering how message overwriting affects data propagation. Their method offers more precise bounds than existing FIFO-based analyses, which they show are safe but pessimistic. Furthermore, the authors prove that end-to-end latency can be optimized by setting the input buffer size of the first regular callback in the chain to one.

Jitter Control and the LET Paradigm. While most analyses focus on bounding the worst-case end-to-end latency, a predictable system also requires low jitter. Control systems rely on low jitter in order to provide reliable and consistent timing. To address this, Abaza et al. [3] proposed a non-intrusive method called *latency shaping*, inspired by the *Logical Execution Time (LET)* [44] paradigm, that uses table-driven reservation servers to achieve near-zero jitter. The core idea of their approach is to decouple when a task finishes from when its output is released, ensuring that data is always published at a predictable, fixed point in time. The authors use a two-server design to implement this method in ROS 2 without modifying application code:

- **High-Priority Server:** This server runs the main chain computations, ensuring that the processing is completed as quickly as possible.
- **Low-Priority Server:** This server has a precisely timed slot dedicated solely to publishing the chain's final output at a deterministic point in time.

This separation guarantees that even if the computation time varies, the final output is always sent with near-zero jitter. For chains using synchronous publishing, the authors architecturally extend the chain with a *republisher* node to maintain this separation without modifying application code. This technique also supports creating multiple *latency bands*, which allows a system to operate in different modes (e.g., city vs. highway driving) with a predictable latency for each mode.

In addition to the aforementioned methods, new research directions are emerging. For example, Lee et al. [57] and Han and Kim [41] propose probabilistic real-time analysis and optimization methods for end-to-end latency in autonomous-driving task graphs, using Autoware as a case study; however, these works target more general scheduling assumptions and are not specific to the ROS 2 scheduling policy.

3.2 Analysis of the Multi-Threaded Executor

Analyzing the real-time behavior of the multi-threaded executor is inherently more complex due to the increased concurrency and resource contention. Research in this area has focused on developing accurate response-time analyses that address these challenges.

The initial response-time analysis for the multi-threaded executor was presented by Jiang et al. [49]. Their work provided two different scheduling tests characterized by different accuracy versus complexity trade-off for ROS 2 processing chains under multi-threaded executors. The second one leverages a dynamic programming approach to explore the space of interfering sequences, thus providing better schedulability results in the evaluation. Conversely, the first one is simpler but more pessimistic. The experiments also identified that a careful assignment of callback groups for callbacks is essential to avoid an increased response time when switching from single to multi-threaded executors. The authors also observed that concurrency issues arise when callbacks share a common resource, potentially leading to higher response times than in the single-threaded executor case. In particular, when several callbacks belong to the same mutually exclusive group, a low-priority callback can be repeatedly removed from and re-added to the wait set together with new high-priority instances. Every time this occurs, the new high-priority callbacks are chosen first, so the low-priority callback is postponed again, increasing the response time. With a single-threaded executor, this does not occur because once a callback is in the queue, it is not removed and reinserted. Teper et al. [104] showed that this concurrency issue makes the multi-threaded executor prone to starvation. Specifically, they observed that, since high-priority callbacks in a callback group can be re-added to the wait set, higher-priority jobs would then be executed before the lower-priority jobs of the callback group that were previously added, thus causing starvation. They provided a few configurations that end up in starving tasks in ROS 2 systems, uncovering unobserved cases in the previous two works on response-time analysis for the multi-threaded executor [49, 91]. To address the problem, the authors proposed a design change to the multi-threaded executor in ROS 2, which prevents the refresh of the wait set for higher-priority tasks in a callback group. They formally and experimentally proved the absence of starvation scenarios, guaranteeing that both the single-threaded and multi-threaded executors align with respect to starvation-freedom. Sobhani et al. [91] also proposed a response time analysis for the multi-threaded executor, but compared to [49], their model accepts arbitrary deadlines and considers fixed-priority scheduling, which makes the multi-threaded executor follow the priority of the corresponding chain when scheduling each callback.

3.3 Bounding Communication and Synchronization Delays

The real-time performance of applications developed using ROS 2 is often not only affected by the scheduling mechanism of ROS 2 itself. As discussed in Section 2.2, lower-level middleware, such as the DDS, may also have a substantial impact on the timing performance since they are in charge of delivering messages according to publish-subscribe. This section first discusses them, and then presents several works related to how ROS 2 computational activities can selectively process incoming messages.

Intra-Process Communication and Lower-Level Middleware (DDS). In ROS 2, communication occurs in two main ways: *intra-process* and *inter-process* communication. Intra-process communication, which handles message exchanges within the same executor, is optimized to avoid unnecessary copies when sending messages from publishers to subscribers.

Intra-process communication has been recently characterized by Luo et al. [70], who showed that its overhead is not constant and can vary based on the message configuration, workload structure, and usage semantics. To improve scalability, the paper proposes guidelines for configuring intra-process communication and an allocation-free message pooling mechanism.

In contrast, inter-process communication manages the more complex task of connecting publishers and subscribers across different executors, often on separate machines, and relying on the underlying DDS middleware. Because the performance of distributed robotic systems hinges on the efficiency of this layer, most real-time research has focused on analyzing the overheads of inter-process communication.

A key factor in this analysis is the message exchange mode, which can be either *synchronous* or *asynchronous* [88]. In synchronous mode, a publishing callback blocks its executor thread until the message is received by all subscribers, ensuring immediate delivery but potentially introducing significant delays. Meanwhile, in asynchronous mode, the callback delegates the sending process to the DDS middleware and continues its execution without blocking. The complexities of this asynchronous mode were formally analyzed by Luo et al. [68], who provided an upper-bound for communication delay when the DDS WriterHistory of ROS 2 is configured with either the FIFO (First-In-First-Out) (used in ROS 2 Humble Hawksbill) or InterestTree policy (used in ROS 2 Foxy Fitzory). The FIFO policy ensures that messages are delivered to subscribers in the order they were sent, while the InterestTree policy maintains separate FIFO queues for each publisher, sending the messages of each publisher sequentially, and ensuring fair delivery.

For a more comprehensive model of the DDS and an end-to-end analysis for distributed applications, Sciangula et al. [88] developed a detailed, compositional DDS model using the CPA framework [43]. Their work offers a generic DDS model and a concrete instantiation for FastDDS, one of the most common implementations. Their model captures the behavior of key DDS components in asynchronous mode, including the *flow-controller threads* that dispatch messages and the *listener threads* that receive them, allowing these delays to be incorporated into a full system analysis. This understanding of the FastDDS implementation was also leveraged by Teper et al. [101] to provide an upper-bound analysis for both intra-process and inter-process communication.

Building on these detailed models, subsequent research has focused on system-level optimization and resolving cross-layer priority issues. Sciangula et al. [87] developed an optimization algorithm that uses analysis-based heuristics to configure the entire communication pipeline, including thread-to-machine and thread-to-core assignments, the number of flow control threads, message partitioning, and thread priorities. Further work [93] presented a virtualized architecture for the DDS, leveraging Xen and Linux, focusing on two design alternatives: raw sockets bypassing the network stack and standard UDP.

Message Filters and Synchronizers. Beyond the core communication stack, ROS 2 provides *message filters*, which are libraries that allow nodes to selectively process incoming messages based on specific criteria, such as timestamps or content. For instance, the *Time Sequencer* filter enforces chronological order by discarding any message with a timestamp earlier than the last received one.

The most critical use of these filters is in *Message Synchronizers*, which are essential for sensor fusion tasks where a callback must only be triggered after receiving a coherent set of inputs from multiple topics. ROS 2 offers three main synchronization policies: *ExactTime*, *ApproximateTime*, and *LatestTime*. As the name suggest, the *ExactTime* policy is a hard filter which verifies that the messages have the exact timestamp.

The *ApproximateTime* policy is the most widely analyzed. It groups messages that arrive within a specified time window, tolerating a small time disparity. Research on this policy has evolved over several years. Li et al. [60] first modeled and analyzed this time disparity, initially for systems with unlimited buffers and later extending the analysis to more realistic limited-size buffers [59]. Moreover, the *ApproximateTime* policy potentially needs messages to remain in the message synchronizer in order to work, introducing additional latencies. Li et al. [59] were the first

to formally model and experimentally measure these latencies, introducing the concepts of *passing latency* and *reaction latency*. The *passing latency* is defined as the time a message spends in the synchronizer before being output, while the *reaction latency* is the time from message arrival to output, including any necessary discarding of older messages. In 2023, Li et al. [62] provided a formal model and experimental measurements of these latencies. Later, Wu et al. [115] built upon [62] to provide more accurate estimates.

A notable limitation of the *ApproximateTime* policy is that it requires prior knowledge of incoming message timing characteristics, and any inaccuracies in this knowledge can significantly degrade performance [95]. Two distinct approaches circumvent this limitation by not relying on a priori information about the messages, though they differ fundamentally in their design goals and guarantees.

The first is SEAM [95], a custom message synchronizer that selects the earliest-arriving message from each input queue such that the time disparity among the selected messages remains within a user-specified threshold, thereby guaranteeing well-bounded time disparity without requiring any prediction of future message arrivals.

The second is the *LatestTime* policy, a more recent addition to ROS 2, which instead prioritizes maximizing output frequency. It achieves this by implementing a *zero-order-hold* mechanism, storing the latest message from each topic and publishing a complete set of the latest message per topic whenever any single topic receives a new message. However, unlike SEAM, *LatestTime* does not guarantee bounded time disparity. Wu et al. [116] were the first to model the latency of this policy and uncovered a critical flaw in its design that could lead to unbounded latencies under certain scenarios.

Finally, the impact on real-time metrics of different sensor fusion patterns has been recently explored by Sobhani et al. [92] in the context of more abstractly modeled autonomous driving systems. Here, the authors identified task types and fusion patterns, dependent on the number of inputs and the activation patterns, and evaluated their impact on the real-time performance of the system.

3.4 Formal Verification with Model Checking

A different line of work has focused on formal verification rather than analytical timing models. Dust et al. [33] proposed an approach for verifying ROS 2 applications using the UPPAAL model checker. Their key contribution is a pattern-based method that uses reusable Timed Automata (TA) templates to simplify the formal modeling of ROS 2 components, including callbacks, communication channels, and two different versions of the single-threaded executor (ExV1 from ROS 2 Dashing and ExV2 from ROS 2 Eloquent and later releases, which differs in that in ExV1 the polling of timer callbacks is done continuously, and in ExV2 it is performed only at polling points). The verification focuses on properties such as callback latency and buffer overflow. Their framework supports both a holistic approach, modeling entire processing chains, and an individual-node approach, which abstracts communication to analyze nodes in isolation. A key advantage of this method is its ability to exhaustively explore the system's state space and generate counterexample traces, which can reveal potential design flaws or critical execution paths not easily discovered through traditional testing.

This formal verification approach was later extended to the multi-threaded executor by the same authors [32]. They provide UPPAAL Timed Automata (TA) templates to model the behavior of the multi-threaded executor, including its locking mechanisms and both mutually exclusive and reentrant execution modes. A key novelty of their work is the ability to model the influence of the underlying operating system by incorporating templates for reservation-based scheduling. This allows for the verification of callback blocking and latency not just from the middleware's internal scheduling, but also from OS-level preemption and resource constraints.

3.5 Taxonomy of Analysis Methods

To provide a general overview of the large body of work discussed in this section, we provide three summary tables, each targeting a different aspect of real-time analysis in ROS 2. The tables are placed within their respective subsections to provide immediate context, but are described here together to give a complete overview.

- **Table 1 (Analysis and Verification Methods)** provides a broad comparison of the analytical and formal verification papers discussed in Sections 3.1, 3.2, and 3.4. It classifies each work based on key criteria such as the executor type, system model, timing metrics, and the assumptions made about execution times, arrivals, and the underlying OS. The criteria are as follows:
  - **Executor type** (EX.): *single-threaded* (ST) vs. *multi-threaded* (MT).
  - **System model** (Model): arbitrary *Directed Acyclic Graphs* (DAG), *linear chains* (Linear), or formal models based on *Timed Automata* (TA).
  - **Timing metric** (Metric): *end-to-end response time* (E2E-RT), *maximum data age/reaction time* (DA/RT), end-to-end timing *Jitter*, or worst-case response time of callbacks (*WCRT*, called individual callback latency in [33]).
  - **Execution time model** (ET): *worst-case execution time* (WCET), *execution time curves* (ET curv.), or measured values from profiling (Measured).
  - **External arrival model** (Arrivals): leveraging *timer-based periodicity* (Timers), based on *arrival curves* (Arr.curve), or a mix of *periodic and sporadic* (Per./Spor.).
  - **DDS communication**: DDS managed as a single *parameter* (Para), as two distinct parameters for *synchronous/asynchronous* cases (2 Para.), or no accounting of DDS (None).
  - **Operating system model** (OS): using a *supply-bound function* (SBF), using *reservation-based servers* (Server), or *not considered* (None).
- **Table 2 (DDS Communication Analysis)** focuses specifically on the papers that analyze the DDS middleware, as detailed in Section 3.3. This table highlights the communication mode, the specific DDS implementation targeted, and the primary focus of the analysis, from delay modeling to system-wide optimization.
- **Table 3 (Message Synchronization Analysis)** summarizes the research on message filters and synchronizers, discussed in Section 3.3. It categorizes papers by the synchronizer policy they analyze and key contributions, e.g., modeling time disparity or identifying design flaws.

Table 1 Comparison of real-time analysis and verification methods for ROS 2 executors.

Paper	EX.	Model	Metric	ET	Arrivals	DDS	OS
Analytical Methods							
Casini et al. [23] (2019)	ST	DAG	E2E-RT	WCET	Arr.curve	Para.	SBF
Tang et al. [97] (2020)	ST	Linear	E2E-RT	WCET	Arr.curve	None	SBF
Blass et al. [19] (2021)	ST	DAG	E2E-RT	ET curv.	Arr.curve	Para.	SBF
Tang et al. [98] (2023)	ST	DAG	E2E-RT	WCET	Arr.curve	None	SBF
Teper et al. [103] (2022)	ST	Linear	DA/RT	WCET	Timers	None	None
Teper et al. [101] (2024)	ST	Linear	DA/RT	WCET	Timers	2 Para.	None
Tang et al. [99] (2024)	ST	Linear	DA/RT	WCET	Arr.curve	None	SBF
Abaza et al. [3] (2024)	ST	Linear	Jitter	Measured	Timers	2 Para.	Server
Jiang et al. [49] (2022)	MT	Linear	E2E-RT	WCET	Timers	None	None
Sobhani et al. [91] (2023)	MT	Linear	E2E-RT	WCET	Arr.curve	None	SBF
Formal Verification Methods (Model Checking)							
Dust et al. [33] (2025)	ST	TA	WCRT	WCET	Per./Spor.	2. Para.	None
Dust et al. [32] (2024)	MT	TA	WCRT	WCET	Per./Spor.	None	Server

Table 2 Communication Analysis in ROS 2, either intra-process or DDS-based.

Paper	Communication	DDS Type	Analysis Focus
Luo et al. [70]	Intra-process	N/A	Scalability analysis
Luo et al. [68]	Async. inter-process	Generic DDS	Worst-case communication delay modeling for FIFO & InterestTree policies
Sciangula et al. [88]	Async. inter-process	FastDDS	End-to-end latency analysis with FastDDS threads modeling
Sciangula et al. [89]	Sync+Async. inter-process	FastDDS	End-to-end latency analysis with FastDDS threads modeling, and integration with ROS 2 analysis
Teper et al. [101]	Intra & inter-process	FastDDS	Formal upper-bound analysis for both communication types using FastDDS
Sciangula et al. [87]	Inter-process	FastDDS	Thread-to-core assignment optimization with analysis-based heuristics

Table 3 Message Synchronization and Filters Analysis in ROS 2.

Paper	Synchronizer	Analysis Focus	Key Contributions
Li et al. [60]	ApproximateTime	Time disparity model	Worst-case time disparity analysis for unlimited buffer size
Li et al. [59]	ApproximateTime	Buffer size constraints	Extended time disparity analysis for limited buffer sizes
Li et al. [62]	ApproximateTime	Latency analysis	Formal modeling of passing latency and reaction latency
Wu et al. [115]	ApproximateTime	Improved analysis	Reduced pessimism in ApproximateTime latency bounds
Sun et al. [95]	SEAM (Custom)	Novel synchronizer	New synchronizer without a priori knowledge, guaranteeing well-bounded time disparity
Wu et al. [116]	LatestTime	Zero-order hold	LatestTime policy analysis with unbounded latency defect identification

4 Enhancements and Modifications for Real-Time Performance

While the previous section focused on analyzing the real-time behavior of default ROS 2 systems, a parallel area of research has been dedicated to actively modifying and extending ROS 2 to improve its real-time performance. This section surveys these enhancements, which range from fundamental changes to the core scheduling logic to the addition of new system-level management capabilities, as well as heterogeneous integration with domain-specific platforms.

In Section 4.1, we review the design of *customized executors*, a research direction that replaces the default schedulers with alternatives that are either completely new or based on classical real-time scheduling algorithms, such as fixed-priority (FP) or earliest deadline first (EDF) [65].

Next, in Section 4.2, we explore *system-level enhancements* that address challenges beyond pure CPU callback scheduling. This includes frameworks for managing shared resources like *GPUs and hardware accelerators* to prevent common issues like priority inversion and provide end-to-end timing guarantees for heterogeneous computations. We also cover systems for *automatic latency management* that dynamically adjust ROS 2 configurations at run-time to meet high-level performance goals.

Finally, in Section 4.3, we examine frameworks for heterogeneous integration, which bridge ROS 2 with specialized platforms like the automotive-grade AUTOSAR AP and the determinism-focused Lingua Franca, enabling a smoother transition from research to production environments.

4.1 Customized Executor Design

The first work in this area was *PiCAS* by Choi et al. [24], who proposed a single-threaded executor based on a fixed-priority, chain-aware policy designed to minimize end-to-end latency. This work includes a formal latency analysis and a corresponding priority assignment scheme, based on the timing requirements and criticalities of the system. The paper showed how the standard executor algorithm can jeopardize the schedulability of time-critical callbacks, and proposes to adopt a fixed-priority scheduling algorithm within a single-threaded executor to improve real-time performance. Additionally, the paper proposes a chain-aware node allocation algorithm to minimize interference between chains. This seminal paper on customizing the executor's behavior to favor real-time predictability sparked a long series of new executor proposals.

Building directly on this foundation, Sobhani et al. [91] extended the priority-driven approach to the multi-threaded executor, providing an analysis that accounts for the performance effects of mutually-exclusive callback groups. It first analyzed the behavior of the default multi-threaded executor scheduling algorithm, then it extended it to priority-driven scheduling, allowing to analyze chains with both arbitrary and constrained deadlines.

In parallel with these fixed-priority designs, several works have explored dynamic-priority scheduling. Arafat et al. [7] first introduced Earliest Deadline First (EDF) scheduling by modifying the single-threaded executor, replacing its ready set with a priority queue. Furthermore, they provide a response-time analysis for the modified executor. In subsequent work by Arafat et al. [4], they extended this concept to the multi-threaded executor by implementing the global EDF scheduling policy [65] to enable parallel execution. Furthermore, a response-time analysis for ROS 2 callbacks under the presented executor is proposed, which also accounts for delays due to shared resources. The paper also establishes model equivalence between the proposed executor (when considered without callback groups) and the fixed-preemption point EDF [120], enabling reuse of the corresponding response-time analysis.

As introduced in Section 2, the polling-based behavior of the default ROS 2 executor is very different from traditional fixed-priority or dynamic-priority schedulers for periodically released tasks. Rather than replacing the executor entirely, Teper et al. [100] sought to bridge the gap between ROS 2's unique processing-window behavior and classical real-time theory by making only minor modifications to the recently introduced ROS 2 events executor. In ROS 2, the default events executor introduces an events queue and allows scheduling decisions to be made immediately after a job completes. In order to make this design more aligned with classical real-time scheduling, the authors proposed to implement a priority queue for scheduling released jobs and to manage job release and scheduling through separate queues. This way, their design is analytically equivalent to a classical non-preemptive, work-conserving, priority-based scheduler. Furthermore, their approach supports both fixed-priority and dynamic priority policies and yields tighter, more predictable analytical bounds and real-world performance compared to the default ROS 2 executor.

To achieve fully-preemptive scheduling, a goal not addressed by the previous non-preemptive designs, Wilson et al. [114] developed a modified ROS 2 executor that assigns each callback to its own dedicated thread that is then managed by Linux's SCHED_DEADLINE scheduling class. This callback-per-thread design supports preemptive EDF scheduling and is particularly suited for mixed-criticality systems. Specifically, when a callback is first invoked, the executor creates a thread, sets its scheduling parameters (period, runtime, and deadline) using the SCHED_DEADLINE

scheduling class, and associates it with the callback. To reuse threads for subsequent executions of the same callback, the executor maintains a mapping between callbacks and their threads and passes a reference to the callback's triggering event (e.g., timer or subscriber) to the associated thread. However, their implementation requires that callbacks are non-reentrant to ensure that only one instance of a callback can execute at any given time. Furthermore, it also assumes that each topic is published by only a single publisher per period.

Focusing on the limitations of the default multi-threaded executor, Liu et al. [67] identified that the lock-protected wait_set is a major bottleneck that causes blocking and context switches. Additionally, due to the wait_set only being updated when it becomes empty, high-priority callbacks may experience arbitrary delays, leading to interference in end-to-end latency for time-sensitive chains. To address this issue, they proposed *RTeX*, a new multi-threaded executor design that replaces the wait_set with a lock-free, atomic-based ready list. This design allows multiple threads to access ready callbacks concurrently, significantly improving efficiency and timing predictability. Furthermore, RTeX supports priority scheduling across chains, ensuring that the callback in the highest-priority chain is always selected. It also integrates a timer monitor that checks for expired timers after each callback execution, updating the ready list in real time based on priority. This design reduces blocking overhead, eliminates unnecessary context switches, and ensures prompt and priority-respecting scheduling, resulting in improved real-time performance for ROS 2 applications.

Addressing the prioritization for the multi-threaded executor, Wu et al. [117] proposed a finer-grained chain-instance-level scheduling EDF scheduling scheme for the multi-threaded executor. Unlike traditional chain-level scheduling, their approach allows finer-grained priority assignment at the level of individual chain instances. Specifically, chain instances are prioritized based on their deadlines, with earlier deadlines receiving higher priority. Each callback instance inherits the priority of its corresponding chain instance, and executor threads always schedule the highest-priority callback available. The authors also provide a response time analysis model for the proposed chain-instance-level EDF scheduling strategy.

Building on this to address mixed-criticality workloads, they later proposed a *Hybrid Scheduling Executor (HSE)* [118] to address the challenge of managing mixed workloads with both critical (real-time) and ordinary (best-effort) tasks, maximizing the performance of ordinary tasks while guaranteeing that all real-time constraints for critical tasks are met. The HSE integrates two distinct schedulers: a real-time scheduler that uses the EDF strategy for critical chains, and a fair scheduler that uses a round-robin policy for ordinary chains. To prevent interference, the HSE introduces two types of threads: exclusive threads, which are dedicated solely to executing critical tasks, and shared threads, which can run tasks from both schedulers to improve CPU utilization. To complement this executor, the authors also developed the Chain-Aware Thread Mapping Strategy (CATMS), an approach for mapping the HSE's threads to a given number of CPU cores.

In 2025, Ishikawa et al. [47] proposed the middleware-transparent ROS 2 executor, which establishes a persistent one-to-one mapping between callbacks and Linux threads, superseding the executor's behavior and allowing callbacks to be executed directly by the OS's scheduling policy. Very recently, Liu et al. [66] proposed ROS$^{\mathrm{RT}}$, a new execution paradigm for ROS 2 leveraging multiple worker threads that supports preemptive scheduling and supports both fixed-priority and EDF scheduling for arbitrary workloads.

micro-ROS. Finally, these principles of priority-driven scheduling have also been adapted to resource-constrained environments such as *micro-ROS*. The default executor in micro-ROS, known as the RCLC Executor, suffers from two critical issues highlighted in [110]: due to its FIFO-based data processing and batch-based callback scheduling, it may violate priority ordering, leading to priority inversion during callback execution and data processing. Additionally, the executor

enters a blocking state during data transmission, leading to inefficient CPU utilization and wasted resources. To address these issues at the system level, Wang et al. [109] proposed PoDS, a *Priority-Driven chain-aware Scheduling* system that integrates the Timing-Deterministic and Efficient (TIDE) executor introduced in [110]. The TIDE Executor introduces two priority-aware queues: a timer queue sorted by timestamps and a ready queue sorted by priority. It also incorporates a priority-based data-processing mechanism and a parallel transmission model to minimize blocking and reduce priority inversion. These modifications make callback execution and data handling more deterministic and efficient. Experimental results in [109] confirm that PoDS significantly outperforms the default micro-ROS in terms of runtime predictability and real-time stability, making it a promising advancement for low-resource robotic systems.

Taxonomy of customized executors, compatibility, and current status. Table 4 summarizes the main customized executors discussed in this section. The table reports, for each contribution, whether the executor is single-threaded (ST) or multi-threaded (MT), the scheduling algorithm implemented, and the main context or key features.

Table 5, instead, reports the current implementation status of the customized executors. For each work, the table reports the date of the last public code commit, the targeted ROS 2 distribution, and the corresponding repository link.

Table 4 Overview of Customized Executors for ROS 2.

Paper	Executor	Scheduling Algorithm	Context/Key Features
Choi et al. [24]	ST	Fixed Priority (FP)	Priority-driven chain-aware scheduling with latency analysis
Sobhani et al. [91]	MT	Fixed Priority (FP)	Priority-driven chain-aware scheduling for multi-threaded executors
Teper et al. [100]	ST	FP & EDF	Modified events executor for non-preemptive periodic task scheduling
Liu et al. [66]	MT	FP & EDF	Callbacks dispatched to OS worker threads, preemptive (no wait set)
Wilson et al. [114]	MT	Preemptive EDF	Physics-informed preemptive mixed-criticality scheduling using Linux
Liu et al. [67]	MT	Priority-driven	Lock-free atomic executor for chain-aware priority-based scheduling
Wu et al. [117]	MT	Chain-instance EDF	Chain-instance level scheduling with deadline-based priorities
Wang et al. [109]	ST	Priority-driven	Priority-driven chain-aware scheduling for micro-ROS
Arafat et al. [7]	ST	EDF	Dynamic priority scheduling with a priority queue implementation
Arafat et al. [4]	MT	Global EDF	Dynamic priority scheduling for multi-threaded executors

Table 5 Status of Customized Executors in ROS 2. The last commit refers to the time of writing (October 2025).

Paper	Last commit	ROS 2 Version	Repository link
Choi et al. [24] Sobhani et al. [91]	February, 2023	*Eloquent*	`https://github.com/rtenlab/ros2-picas`
Teper et al. [100]	March, 2025	*Rolling*	`https://github.com/tu-dortmund-ls12-rt/ros2_executor_evaluations`
Wilson et al. [114]	February, 2025	*Foxy*	`https://github.com/RTIS-Lab/ROS-Phys-MC`
Liu et al. [67]	June, 2024	*Humble*	`https://github.com/ESLab2012/RTeX`
Wu et al. [117]	December, 2024	Unknown	`https://github.com/Wuzhengda55/CIL-EDF-for-ROS2-multi-threaded-Executors`
Wang et al. [109]	October, 2025	Micro-ROS	`https://github.com/ESLab2012/PoDS/tree/main`
Arafat et al. [7]	July, 2025	*Foxy*	`https://github.com/RTIS-UCF/ros_edf/`
Arafat et al. [4]	March, 2025	Unknown	`https://github.com/RTIS-Lab/ROS-Dynamic-Executor`
Liu et al. [66]	May, 2025	*Humble*	`https://github.com/sizheliu-unc/rclcpp`

4.2 System-Level Enhancements

Next, we discuss papers addressing inter-process communication in ROS 2, its interaction with GPU and accelerators, its integration in containerized systems and orchestration mechanisms, and mechanisms for automated latency management.

Optimizing Inter-Process Communication. Beyond CPU scheduling, researchers have proposed system-level enhancements targeting the performance and predictability of ROS 2's inter-process communication (IPC), often focusing on limitations within the default DDS middleware or exploring alternatives. Kim et al. [52] identified the lack of priority propagation across ROS 2's multi-layered architecture, leading to priority inversion where low-priority DDS or kernel threads delay high-priority application messages. To address this, they developed the *Cros-Rt* framework, which explicitly aligns thread priorities across the application, middleware, and kernel layers to ensure consistent communication across layers that uphold priority ordering. Orthogonally, Ishikawa et al. [46] proposed the *ROS 2 Agnocast* framework to optimize the data transfer efficiency within ROS 2 by using zero-copy shared memory for IPC. They observed that current applications are often bottlenecked by data serialization and copying overheads in DDS, especially for large messages, but also identified key limitations in existing zero-copy DDS implementations, such as the inability to handle unsized message types (e.g., images or point clouds). They introduce a novel publish/subscribe solution through Agnocast using shared memory and pointer-swapping that supports unsized message types, enabling true zero-copy IPC and further enhancing communication efficiency through latency reduction and throughput improvement. Specific middleware protocols for large data objects have been investigated also by Peeck et al. [79, 80] in the context of the DDS. Luo et al. [69] proposed a multi-stage approach for enabling efficient zero-copy inter-

process communication in ROS 2, with a focus on supporting dynamically structured data fields. The approach builds upon the mini memory management system component, allowing multiple processes to access the same message instance from a shared memory space, and the message propagation is adapted, which ensures compatibility with the existing ROS 2 communication framework.

Enhancing GPU and Accelerator Management. ROS 2 serves as a powerful resource manager for CPU tasks, but it lacks built-in support for GPU resource management. Furthermore, the vast majority of the discussed works focus on CPU scheduling only and neglect the presence of hardware accelerators. Nevertheless, many critical functionalities in autonomous systems, such as perception and intelligent control, require intensive GPU computation. However, under default ROS 2, these components must submit their GPU workloads independently, without coordination.

As shown by Ali et al. [6] for a ROS 2 based drone system, this unmanaged approach can lead to severe performance degradation and unpredictable behavior. They identified critical issues arising from naive utilization of CPU cores and GPUs: tasks experience undue scheduling delays when accessing busy compute hardware resources, interference occurs when concurrently running workloads contend for shared resources and slow each other down, and tasks miss deadlines due to blocked and interrupted GPU accesses. This lack of real-time scheduling guarantees makes it difficult to provide end-to-end timing guarantees for chains that use accelerators.

To address these challenges, several frameworks have been proposed to manage accelerator access at the middleware level. One such solution, ROSGM [61], was proposed as an extension to ROS 2, introducing customizable GPU management policies and enabling dynamic switching between different policies at runtime. ROSGM allows developers to implement various GPU scheduling strategies as plug-ins and consists of three key components: (1) a registration table for GPU callbacks, (2) a waiting queue pool that organizes requests based on the selected policy, and (3) a high-priority scheduler thread that processes the requests. A dynamic policy switching mechanism is implemented using ROS 2's publish-subscribe model, and the framework supports three empirically evaluated policies: *exclusive*, *sharing*, and *preemption*.

An alternative architecture for managed accelerator access was proposed by Enright et al. [37] with the Priority-driven Accelerator Access Management (PAAM) framework, which treats accelerators as a schedulable service. Instead of having callbacks invoke hardware directly, PAAM uses a standalone ROS 2 executor that acts as a dedicated accelerator resource server. Client callbacks send requests to this server, which arbitrates access based on the priorities of the originating chains, thus resolving the priority inversion problem inherent in direct, unmanaged access. PAAM employs a two-level scheduling hierarchy: requests are first mapped to hardware-level priority "buckets" (e.g., CUDA streams) and then scheduled within each bucket according to their fine-grained chain priority. To minimize communication overhead, the framework uses a separate data plane with zero-copy shared memory for large kernel data, while small control messages are sent via DDS. The work also provides a worst-case response time analysis for chains with accelerator segments and demonstrated up to a 91% reduction in end-to-end latency for critical chains.

Beyond arbitrating access to accelerator computation, another line of work has focused on optimizing the efficiency of data movement between CPUs and accelerators. While ROS 2 provides zero-copy semantics for multi-core CPUs, Hazcat [12] extends this capability to heterogeneous computing platforms. It provides a heterogeneity-aware memory management framework that enables portable zero-copy semantics across devices by eliminating unnecessary data movement. Hazcat introduces specialized allocators that automatically perform copy operations only when required in a decentralized manner. Its implementation uses a shared message queue with reference counting to manage memory that may have multiple copies for different devices, with garbage collection triggered when all subscribers have processed the message.

ROS with Containers, in the Edge, Fog, and Cloud. The first efforts on leveraging ROS with containers have been spent for ROS 1: in this context, Aldegheri et al [5] presented a toolchain based on container-based packaging in which applications are orchestrated across heterogeneous hardware architectures. The paper uses Docker and Kubernetes and designs the framework to be used in the context of ROS, also discussing communication mechanisms among distributed and containerized nodes. A more recent effort is due to Ichnowski et al. [45], who presented FogROS2. FogROS2 extends the ROS 2 launch system, allowing the deployment of ROS 2 nodes in the cloud and in the fog, automating the configuration (including DDS), provisioning, and data compression so that computationally intensive components can be transparently offloaded to nearby or remote servers while preserving the standard ROS 2 programming model. It uses Docker containers, and it is compatible with AWS EC2 and Kubernetes.

Betz et al. [16] proposed a containerized microservice architecture for ROS 2–based autonomous driving software by redesigning Autoware into a set of Docker microservices orchestrated with lightweight Kubernetes (k3s). The paper compares different settings: bare-metal, single-container, and multi-container deployments on both x86 and Arm platforms, evaluating end-to-end latency, jitter, and resource usage using DDS benchmarks, a perception pipeline, and full closed-loop Autoware simulation. Their results show that containerized deployments can match or even improve end-to-end latency and jitter compared to bare-metal, while reducing CPU/memory usage and startup time, supporting the viability of microservice architectures for real-time robotic stacks.

Very recently, Betz et al. [15] presented an optimization framework for containerized ROS 2 autonomous driving software, targeting the reduction of end-to-end latency across multiple cause–effect chains in an autonomous racing stack. The paper formulates a constraint-optimization problem that jointly decides the assignment of nodes to executors, processes, and containers, whether to use DDS or intra-process communication, and possibly suitable timer periods by leveraging the timing analysis in [101]. The approach is evaluated on TUM Autonomous Motorsport's ROS 2 Humble–based racing software, deployed as Docker microservices orchestrated via docker-compose and tested in a closed-loop Unity simulation with LiDAR/RADAR sensors.

Automatic Latency Management. Applying formal real-time theory often requires detailed, static knowledge of the system, such as worst-case execution times and message arrival patterns. This approach is sometimes impractical for ROS 2 applications, which rely on reusable "black box" components whose activation semantics, functional interactions, message arrival patterns, and worst-case execution times are either not well understood or subject to dynamic changes. To address this challenge, ROS-Llama [18] was developed as an automatic latency manager that provisions ROS 2 applications dynamically at runtime. Instead of requiring complex, low-level parameters, ROS-Llama uses a declarative approach where the user only specifies high-level latency goals for each critical processing chain and a degradation policy that manages priorities in case of transient overloads. It operates through introspection, automatically extracting a system model at runtime, estimating the necessary parameters, and adjusting the scheduling configurations (using Linux's SCHED_DEADLINE) as conditions evolve to decide whether any chains should be degraded to best-effort mode. An evaluation on a TurtleBot3 demonstrated that ROS-Llama provides superior latency control under load compared to both the default Linux CFS scheduler and a criticality-monotonic SCHED_RR baseline. Very recently, Enright et al. [36] proposed a Latency Management Executor (LaME), which leverages SCHED_DEADLINE applied to an extended multi-threaded executor, and determines thread budget of individual executor threads, chain-to-threadclasses (a threadclass is a group of executor threads) allocation, and considers an affinity-based scheduling in which chains can execute either on a single threads, according to partitioned scheduling, or to multiple threads, according to constrained global scheduling.

A different approach to achieving predictable execution focuses on the choice of programming language and its concurrency model. The increasing popularity of the Rust programming language for robotics has driven interest in its real-time capabilities, as its asynchronous programming paradigm differs significantly from the thread-based model of C++ ROS 2. Skoudlil et al. [108] explored the execution model of R2R, a community-supported asynchronous Rust client library for ROS 2. They compared various asynchronous Rust runtimes (like Tokio and futures) and proposed a specific application structure designed for deterministic, real-time operation, which involves separating the event-sampling and callback-execution tasks into different threads with distinct priorities. Their experiments showed that this proposed structure significantly outperforms other configurations and can achieve bounded response times that align with theoretical real-time analysis.

4.3 Heterogeneous Integration

This section discusses how ROS 2 has been integrated with other frameworks and languages, such as AUTOSAR Adaptive and Lingua Franca.

AUTOSAR Integration. A key challenge in deploying autonomous systems is bridging the significant "platform gap" between academic research and industrial automotive production. While researchers favor the open-source and flexible ROS 2 for rapid prototyping, the automotive industry relies on standardized platforms like AUTOSAR Adaptive Platform (AP) [13, 38] to meet the strict requirements for safety, security, and real-time performance necessary for commercial vehicles. To bridge this gap, Iwakami et al. [48] proposed a collaboration framework that enables communication between ROS 2 and AUTOSAR AP applications, particularly in cloud-based development environments for Software-Defined Vehicles (SDV). The framework addresses the fundamental communication incompatibility between AUTOSAR AP's SOMEIP protocol and ROS 2's DDS middleware by implementing a bridge converter that operates as a ROS 2 node, enabling bidirectional data exchange through protocol conversion and service discovery mechanisms. As a result, it enables seamless use of ROS 2 development tools for visualization and debugging, like RViz and ROSbag, with AUTOSAR AP applications, facilitating rapid prototyping in cloud environments while maintaining compatibility with automotive industry standards. The bridge converter incorporates SOMEIP-SD functionality for service registration and discovery, performs type conversion between the different data formats, and leverages CommonAPI and vsomeip for industry-standard SOMEIP implementation. A key innovation is the Bridge Generator, a GUI-based tool that automatically generates conversion code and configuration files from minimal user input with support for complex message types. Evaluation on AWS EC2 instances demonstrated communication latency scaling from 5 ms for small messages to approximately 80 ms for 4 MB payloads, with throughput exceeding 40 MB/s, which is sufficient for high-resolution LiDAR sensors like the VLS-128.

Lingua Franca Integration. Another approach addresses the inherent nondeterminism of ROS 2's standard publish-subscribe model by integrating it with Lingua Franca (LF) [73], a deterministic coordination language designed for real-time and cyber-physical systems. Kwok et al. [55] introduced the High-Performance Robotic Middleware (HPRM), a framework built on top of Lingua Franca, which leverages LF's reactor model to provide deterministic event processing through logical time coordination. It offers both centralized and decentralized coordination strategies where the Runtime Infrastructure (RTI) manages global event ordering or federates coordinate directly through peer-to-peer communication with Safe-to-Process (STP) offsets. The middleware incorporates three key optimizations: an in-memory object store (Plasma) for zero-copy

transfer of payloads larger than 64KB, adaptive serialization that uses out-of-band techniques for large data structures like NumPy arrays while maintaining in-band serialization for smaller data types, and an eager protocol with real-time sockets that pre-allocates 64KB buffers and disables Nagle's algorithm to minimize handshake latency. As a result, it overcomes the high communication latency in ROS 2 when handling large data payloads. Benchmark results demonstrate HPRM achieves up to 173× lower latency than ROS 2 when broadcasting large messages to multiple nodes, with particularly significant improvements for 10MB camera images (77× faster) and 50MB payloads (173× faster). In real-world validation using CARLA autonomous driving simulations with parallel reinforcement learning agents and YOLO object detection, HPRM achieved 91.1% lower latency compared to ROS 2, demonstrating its effectiveness in complex, multi-modal robotic applications requiring deterministic real-time performance.

5 Tools, Profiling, and Experimentation

While the aforementioned theoretical methods are necessary to provide formal real-time guarantees for ROS 2 systems, real-world systems also require robust evaluation of their performance through comprehensive tooling for tracing, latency measurement, and real-time benchmarking. These tools enable visibility into message flows, callback scheduling, and end-to-end delays under diverse workloads and deployment scenarios. In Section 5.1, we review profiling and tracing frameworks that instrument ROS 2 middleware and applications. We then examine application-level latency studies leveraging these tools to characterize communication and computation overheads across hardware platforms and runtime configurations in Section 5.2.

Table 6 Overview of Profiling Tools for ROS 2.

Paper	Key Features	Compatibility	Source Code
Bédard et al. [11] (ros2_tracing)	Instrument and analyze using ROS 2 tracepoints and LTTng	Rolling, Dashing, Eloquent, Foxy, Galactic, Humble	`https://gitlab.com/ros-tracing/ros2_tracing`
Bédard et al. [10]	Message flow analysis using ros2_tracing and Eclipse Trace Compass	Rolling, Dashing, Eloquent, Foxy, Galactic, Humble	`https://github.com/christophebedard/ros2-message-flow-analysis`
Li et al. [63] (Autoware Perf)	End-to-end latency analysis using ros2_tracing	Foxy, Galactic	`https://github.com/azu-lab/ROS2-E2E-Evaluation`
Abaza et al. [2]	Tracing and timing analysis using eBPF	Foxy	Not available
Kuboichi et al. [54] (CARET)	End-to-end latency measurement using LTTng with additional tracepoints	Galactic, Humble, Iron, Jazzy	`https://github.com/tier4/caret`
He et al. [42] (TILDE)	Dynamic message tracking using TILDE embedded nodes	Humble	`https://github.com/tier4/TILDE`

5.1 Profiling Tools

Bédard et al. [11] developed `ros2_tracing`, a low-overhead tracing tool that enables multipurpose instrumentation for ROS 2, capturing key events such as message publications, callbacks, services, executor states, and lifecycle states. It achieves this through tracepoint calls for both initialization

and runtime events, allowing for detailed execution analysis of applications. The tool employs the `low-overhead Linux Trace Toolkit: Next Generation` (LTTng) tracer as the default tracing backend and leverages the `tracetools` package to support tracepoint calls. To evaluate the overhead introduced by `ros2_tracing`, experiments measured the time between publishing a message and its reception by the subscription callback with and without the tool. Results indicate that the mean latency overhead remains below 15%, with each tracepoint call introducing overhead at the microsecond level.

The `ros2_tracing` tool was later extended to extract and visualize message flows across distributed ROS 2 systems [10]. This extension builds an intermediate execution database from trace data collected in distributed systems, detailing ROS 2 objects (nodes, publishers, subscriptions, timers) and events (message publication, reception instances, and callback executions). The extended tool can automatically detect one-to-one and one-to-many causal links between received and published messages for direct links and uses user-level annotations to identify more complex, indirect causal relationships. To illustrate how the extended tool can analyze ROS 2 and application-level logic, two experiments were performed. The first experiment used the Autoware reference system, splitting its nodes across multiple executables on two hosts within the same network. The second experiment used the RTAB-Map simultaneous localization and mapping (SLAM) system and distributed it across two computers connected over a wireless network.

Li et al. [63] developed `Autoware Perf`, a tracing and performance analysis framework for ROS 2 applications, including Autoware.Auto. It extends `ros2_tracing` by allowing users to selectively trace specific nodes by manually inputting information about chosen nodes and their callback dependencies. It aims to analyze end-to-end latency by measuring node execution time and publish/subscribe communication time. Since `Autoware Perf` does not support tracing individual messages, the end-to-end latency, accounting for lost messages during measurement, is analytically estimated using the convolution integral of the discretized probability distribution.

Abaza et al. [2] proposed a tracing and measurement framework for ROS 2-based applications. The framework utilizes an `extended Berkeley Packet Filter` (eBPF)-based tracer, which probes various functions within the ROS 2 middleware and extracts their arguments or return values to analyze the data flow within applications. By integrating event traces from both ROS 2 and the operating system, the framework constructs a directed acyclic graph that visualizes ROS 2 callbacks, their precedence relationships, and associated timing attributes. To evaluate its effectiveness, experiments were conducted using a real-world benchmark implementing LIDAR-based localization in Autoware's Autonomous Valet Parking system.

In addition to measuring end-to-end latency, `CARET` [54] was developed to facilitate detailed tracking and analysis of message flow in ROS 2 applications. By adding additional tracepoints related to the message flow, `CARET` can detect lost messages that never reach their destination, thereby enabling more accurate calculation of end-to-end latency. This tool has been used in experiments with Autoware.Universe to identify system bottlenecks and understand their underlying causes.

To support online tracking of message flow and deadline monitoring during system execution, `TILDE` [42] was developed as a dynamic message tracking infrastructure for ROS 2 applications. `TILDE` augments the ROS 2 system by publishing specialized messages called MessageTrackingTags alongside the original ROS 2 messages to log the publishing and subscribing relationships between nodes. A special deadline detector node is also added at the end of the data flow to collect these tags, computes end-to-end latencies, and reports deadline misses.

5.2 Profiling Various Latencies using Different Applications

The first work exploring the performance of ROS 2 was by Maruyama et al. [72], who described the differences between ROS 1 and ROS 2 and evaluated ROS 2 performance using an alpha version available in 2016. Kronauer et al. [53] conducted empirical experiments on desktop PCs and Raspberry Pis to profile communication latency in ROS 2 across multiple hardware nodes. Their experiments focused on a use case where a sensor reading is published in one node and then processed through a chain of nodes, with variables including publishing frequency, payload size, number of computing nodes, and quality of service (QoS) settings. They measured the 95th percentile of end-to-end latency in ROS 2 distributed systems using default settings and different DDS middleware, including Connext, FastRTPS, and CycloneDDS, focusing specifically on latency attributed to ROS 2 rather than the DDS middleware itself. To capture intra-layer communication latency, they employed the `ros2profiling` package to obtain precise timestamps. The results reveal that end-to-end latency decreases as frequency increases, especially with longer data-processing pipelines. This effect may be due to unknown energy-saving features in other hardware components or changes in thread scheduling priorities. For payload sizes exceeding the UDP fragmentation threshold, latency increases with payload size. The most significant sources of latency were attributed to DDS and rclcpp notification delay. Notably, ROS 2 introduces up to 50% additional latency over raw DDS for small messages (128 B), while latency for larger messages (500 KB) is largely dominated by the DDS middleware. Furthermore, DDS middleware performance differs between Raspberry Pis and desktop PCs, with Raspberry Pis exhibiting higher latency and greater fluctuation in latency compared to desktop PCs.

Betz et al. [14] conducted latency measurements using the `ros2_tracing` tool and hardware-in-the-loop (HiL) simulation of `Autoware.Universe`, an open-source autonomous driving software stack. Their study focused on the impact of timer frequencies, executor priorities in ROS 2, DDS implementations, operating system (OS) scheduling, and various hardware configurations. The evaluated path in Autoware starts from receiving LiDAR data and ends at the controller output via EKF localization, involving 12 nodes and 15 callbacks, including 3 timer callbacks. Performance metrics measured include end-to-end latency of the evaluated path, communication latency between ROS 2 nodes, compute latency for callback processing, and idle latency due to intra-node timers. Results indicate that prioritizing subscription callbacks over timer callbacks increases both idle time and end-to-end latency. The findings suggest that increasing timer frequencies, as computational resources allow, can reduce idle latency. While different DDS implementations show minimal impact on communication latency, variations such as whether separate DDS processes are created do influence callback execution, leading to differences in end-to-end latency. Additionally, different OS schedulers achieve relatively similar throughput, though the choice of round-robin quantum affects end-to-end latency. Lastly, experiments show that hyperthreading and variable CPU clock speeds increase end-to-end latency due to delays in executing timer callbacks.

Barut et al. [9] conducted a communication benchmark involving node/component communication typical in robotics to compare the real-time performance (in terms of message latency and jitter) of ROS 2 and the earlier Open Robot Control Software (OROCOS) framework. The benchmark involved periodically sending a request with echo data that was immediately responded to upon receipt. Experiments were performed on both vanilla and PREEMPT_RT patched Linux kernels under normal conditions and stress scenarios induced by the standard Linux `stress` tool. The results show that, under vanilla Linux without stress, ROS 2 and OROCOS exhibit comparable message latencies, although ROS 2 shows higher jitter. Under stress, OROCOS experiences significant latency spikes. In comparison, under vanilla Linux with stress, ROS 2 suffers a linear increase in latency of sending messages periodically, likely due to dropped messages

under high load. When using the PREEMPT_RT patched kernel, ROS 2 performance improves markedly, showing bounded latencies. These findings highlight the critical importance of stress testing in evaluating system performance, particularly for real-time applications in robotics.

6 Conclusions

In this survey, we reviewed the state of the art on real-time ROS 2, addressing both real-time analysis techniques and a wide range of practical enhancements to the ROS 2 software stack proposed over the years. We first provided the necessary background on the internal scheduling mechanisms of ROS 2 executors and their interactions with various communication middleware, such as the DDS. Later, we classified and discussed existing works on timing analysis for single- and multi-threaded executors.

We then surveyed a broad spectrum of runtime extensions and system-level mechanisms proposed to improve the predictability and performance of ROS 2 applications. These works include custom executor algorithms, runtime mechanisms for heterogeneous computing platforms (e.g., GPUs and accelerators), support for resource-constrained devices via micro-ROS, and profiling and tracing tools that enable systematic experimentation.

We also summarized state-of-the-art techniques for bounding communication delays, including DDS-level modeling and message-filtering strategies. The taxonomies introduced in this survey help to provide a structured view of a rapidly evolving research landscape.

Research directions for future work are manifold. For example, studying mechanisms to enable predictable interaction between ROS 2 and hardware accelerators for AI can yield impactful research. From a theoretical standpoint, it can be interesting to target the impact of model uncertainty (e.g., uncertain execution time estimates [121] or uncertain arrival patterns [106]) on timing metrics.

However, most importantly, future efforts of the community should enable technology transfer from the research community to the ROS 2 mainline by first achieving a common view of how the new ROS 2 executor should be designed and implemented to guarantee timing constraints.

References

1 OFERA: Open Framework for Embedded Robot Applications. https://www.ofera.eu/, 2018. Horizon 2020 project, Grant Agreement No. 780785.

2 Hazem Abaza, Debayan Roy, Shiqing Fan, Selma Saidi, and Antonios Motakis. Trace-enabled timing model synthesis for ros2-based autonomous applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024. doi:10.23919/DATE58400.2024.10546872.

3 Hazem Abaza, Debayan Roy, Bohdan Trach, Wanli Chang, Selma Saidi, Antonios Motakis, Wei Ren, and Yutao Liu. Managing end-to-end timing jitters in ros2 computation chains. In *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, pages 229–241, 2024. doi:10.1145/3696355.3696363.

4 Abdullah Al Arafat, Kurt Wilson, Kecheng Yang, and Zhishan Guo. Dynamic priority scheduling of multithreaded ros 2 executor with shared resources. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3732–3743, 2024. doi:10.1109/TCAD.2024.3445259.

5 Stefano Aldegheri, Nicola Bombieri, Franco Fummi, Simone Girardi, Riccardo Muradore, and Nicola Piccinelli. Late breaking results: Enabling containerized computing and orchestration of ros-based robotic sw applications on cloud-server-edge architectures. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–2, 2020. doi:10.1109/DAC18072.2020.9218659.

6 Syed W Ali, Angelos Angelopoulos, Denver Massey, Sarah Haddix, Alexander Georgiev, Joseph Goh, Rohan Wagle, Prakash Sarathy, James H Anderson, and Ron Alterovitz. On the necessity of real-time principles in gpu-driven autonomous robots. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8086–8092. IEEE, 2025. doi:10.1109/ICRA55743.2025.11128627.

7 Abdullah Al Arafat, Sudharsan Vaidhun, Kurt M Wilson, Jinghao Sun, and Zhishan Guo. Response time analysis for dynamic priority scheduling in ros2. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 301–306, 2022. doi:10.1145/3489517.3530447.

8 AUTOSAR. Specification of communication management. Technical report, Autosar consortium, 2020. Accessed: 2025-03-03. URL: https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_SWS_CommunicationManagement.pdf.

9 Sinan Barut, Marco Boneberger, Pouya Mohammadi, and Jochen J Steil. Benchmarking real-time capabilities of ros 2 and orocos for robotics applications. In *International Conference on Robotics and Automation (ICRA)*, pages 708–714. IEEE, 2021. doi:10.1109/ICRA48506.2021.9561026.

10 Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, and Michel Dagenais. Message flow analysis with complex causal links for distributed ros 2 systems. *Robotics and Autonomous Systems*, 161:104361, 2023. doi:10.1016/J.ROBOT.2022.104361.

11 Christophe Bédard, Ingo Lütkebohle, and Michel Dagenais. ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ros 2. *Robotics and Automation Letters*, 7(3):6511–6518, 2022. doi:10.1109/LRA.2022.3174346.

12 Oren Bell, Chris Gill, and Xuan Zhang. Hardware acceleration with zero-copy memory management for heterogeneous computing. In *29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 28–37. IEEE, 2023. doi:10.1109/RTCSA58653.2023.00013.

13 Davide Bellassai, Claudio Scordino, Daniel Casini, and Alessandro Biondi. Ap-let: Enabling deterministic pub/sub communication in autosar adaptive. *Journal of Systems Architecture*, 162:103390, 2025. doi:10.1016/J.SYSARC.2025.103390.

14 Tobias Betz, Maximilian Schmeller, Harun Teper, and Johannes Betz. How fast is my software? latency evaluation for a ros 2 autonomous driving software. In *Intelligent Vehicles Symposium (IV)*, pages 1–6. IEEE, 2023. doi:10.1109/IV55152.2023.10186585.

15 Tobias Betz, Harun Teper, Dominic Ebner, Maximilian Leitenstern, Simon Sagmeister, Marcel Weinmann, Jian-Jia Chen, and Markus Lienkamp. End-to-end latency optimization for containerized ros 2 autonomous driving software. *IEEE Access*, 13:112654–112672, 2025. doi:10.1109/ACCESS.2025.3582868.

16 Tobias Betz, Long Wen, Fengjunjie Pan, Gemb Kaljavesi, Alexander Zuepke, Andrea Bastoni, Marco Caccamo, Alois Knoll, and Johannes Betz. A containerized microservice architecture for a ros 2 autonomous driving software: An end-to-end latency evaluation. In *2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 57–66, 2024. doi:10.1109/RTCSA62462.2024.00018.

17 T. Blass. *Real-time Execution Management in the ROS 2 Framework*. Ph.d. dissertation, Robert Bosch GmbH and Saarland University, Saarland Informatics Campus (SIC), Germany, 2022.

18 Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn B Brandenburg. Automatic latency management for ros 2: Benefits, challenges, and open problems. In *27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 264–277. IEEE, 2021.

19 Tobias Blaß, Daniel Casini, Sergey Bozhko, and Björn B. Brandenburg. A ros 2 response-time analysis exploiting starvation freedom and execution-time variance. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 41–53, 2021. doi:10.1109/RTSS52674.2021.00016.

20 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

21 Sergey Bozhko and Björn B. Brandenburg. Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2020.22.

22 Daniel Casini, Luca Abeni, Alessandro Biondi, Tommaso Cucinotta, and Giorgio Buttazzo. Constant bandwidth servers with constrained deadlines. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 68–77, 2017. doi:10.1145/3139258.3139285.

23 Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B. Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2019.6.

24 Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. Picas: New design of priority-driven chain-aware scheduling for ros2. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–263. IEEE, 2021. doi:10.1109/RTAS52030.2021.00028.

25 Angelo Corsaro, Luca Cominardi, Olivier Hecart, Gabriele Baldoni, Julien Enoch Pierre Avital, Julien Loudet, Carlos Guimares, Michael Ilyin, and Dmitrii Bannov. Zenoh: Unifying communication, storage and computation from the cloud to the microcontroller. In *2023 26th Euromicro Conference on Digital System Design (DSD)*, pages 422–428, 2023. doi:10.1109/DSD60849.2023.00065.

26 Dakshina Dasari, Matthias Becker, Daniel Casini, and Tobias Blaß. End-to-end analysis of event chains under the QNX adaptive partitioning scheduler. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 214–227, 2022. doi:10.1109/RTAS54340.2022.00025.

27 Daniel B de Oliveira, Rômulo S de Oliveira, and Tommaso Cucinotta. A thread synchronization model for the preempt_rt linux kernel. *Journal of Systems Architecture*, 107:101729, 2020. doi:10.1016/J.SYSARC.2020.101729.

28 Daniel B de Oliveira, Rômulo S de Oliveira, Tommaso Cucinotta, and Luca Abeni. Automata-based

modeling of interrupts in the linux preempt rt kernel. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

29 Daniel Bristot de Oliveira, Daniel Casini, and Tommaso Cucinotta. Operating system noise in the linux kernel. *IEEE Transactions on Computers*, 72(1):196–207, 2022. doi:10.1109/TC.2022.3187351.

30 Daniel Bristot De Oliveira, Daniel Casini, Juri Lelli, and Tommaso Cucinotta. Timerlat: Real-time linux scheduling latency measurements, tracing, and analysis. *IEEE Transactions on Computers*, 2025.

31 Raimarius Delgado, Bum-Jae You, and Byoung Wook Choi. Real-time control architecture based on xenomai using ros packages for a service robot. *Journal of Systems and Software*, 151:8–19, 2019. doi:10.1016/j.jss.2019.01.052.

32 Lukas Dust, Rong Gu, Cristina Seceleanu, Mikael Ekström, and Saad Mubeen. Uppaal-based modeling and verification of ros 2 multi-threaded execution and operating system reservations. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 40–59. Springer, 2024.

33 Lukas Dust, Rong Gu, Cristina Seceleanu, Mikael Ekström, and Saad Mubeen. Pattern-based verification of ros 2 applications using uppaal. *International Journal on Software Tools for Technology Transfer*, pages 1–20, 2025.

34 Lukas Johannes Dust, Emil Persson, Mikael Ekström, Saad Mubeen, Cristina Seceleanu, and Rong Gu. Experimental evaluation of callback behavior in ros 2 executors. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2023. doi:10.1109/ETFA54631.2023.10275668.

35 Salvatore D'Avella, Carlo Alberto Avizzano, and Paolo Tripicchio. Ros-industrial based robotic cell for industry 4.0: Eye-in-hand stereo camera and visual servoing for flexible, fast, and accurate picking and hooking in the production line. *Robotics and Computer-Integrated Manufacturing*, 80:102453, 2023. doi:10.1016/J.RCIM.2022.102453.

36 Daniel Enright, Hoora Sobhani, and Hyoseung Kim. Theory-guided adaptive scheduling for ros 2. In *Proceedings of the 33rd International Conference on Real-Time Networks and Systems (RTNS 2025)*, Lecture Notes in Computer Science, Pisa, Italy, 2025. Springer.

37 Daniel Enright, Yecheng Xiang, Hyunjong Choi, and Hyoseung Kim. Paam: A framework for coordinated and priority-driven accelerator management in ros 2. *arXiv preprint arXiv:2404.06452*, 2024. doi:10.48550/arXiv.2404.06452.

38 Simon Fürst and Markus Bechter. Autosar for connected and autonomous vehicles: The autosar adaptive platform. In *2016 46th annual IEEE/IFIP international conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 215–217. IEEE, 2016. doi:10.1109/DSN-W.2016.24.

39 Muhammad Liman Gambo, Abubakar Danasabe, Basem Almadani, Farouq Aliyu, Abdulrahman Aliyu, and Esam Al-Nahari. A systematic literature review of dds middleware in robotic systems. *Robotics*, 14(5):63, 2025. doi:10.3390/ROBOTICS14050063.

40 Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. On the equivalence of maximum reaction time and maximum data age for cause-effect chains. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, pages 10–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ECRTS.2023.10.

41 Taeho Han and Kanghee Kim. Minimizing probabilistic end-to-end latencies of autonomous driving systems. in 2023 ieee 29th real-time and embedded technology and applications symposium (rtas). 27–39, 2023.

42 Xuankeng He, Hiromi Sato, Yoshikazu Okumura, and Takuya Azumi. Tilde: Topic-tracking infrastructure for dynamic message latency and deadline evaluator for ros 2 application. In *27th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–9. IEEE/ACM, 2023. doi:10.1109/DS-RT58998.2023.00010.

43 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEEE Proceedings - Computers and Digital Techniques*, March 2005.

44 Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Embedded control systems development with giotto. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 64–72, 2001. doi:10.1145/384198.384208.

45 Jeffrey Ichnowski, Kaiyuan Chen, Karthik Dharmarajan, Simeon Adebola, Michael Danielczuk, Víctor Mayoral-Vilches, Nikhil Jha, Hugo Zhan, Edith Llontop, Derek Xu, Camilo Buscaron, John Kubiatowicz, Ion Stoica, Joseph Gonzalez, and Ken Goldberg. Fogros2: An adaptive platform for cloud and fog robotics using ros 2. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5493–5500, 2023. doi:10.1109/ICRA48891.2023.10161307.

46 Takahiro Ishikawa-Aso and Shinpei Kato. Ros 2 agnocast: Supporting unsized message types for true zero-copy publish/subscribe ipc, 2025. doi:10.48550/arXiv.2506.16882.

47 Takahiro Ishikawa-Aso, Atsushi Yano, Takuya Azumi, and Shinpei Kato. Work in progress: Middleware-transparent callback enforcement in commoditized component-oriented real-time systems. In *Proceedings of the 31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 426–429. IEEE, 2025. doi:10.1109/RTAS65571.2025.00017.

48 Ryudai Iwakami, Bo Peng, Hiroyuki Hanyu, Tasuku Ishigooka, and Takuya Azumi. Autosar ap and ros 2 collaboration framework. In *2024 27th Euromicro Conference on Digital System Design (DSD)*, pages 319–326. IEEE, 2024. doi:10.1109/DSD64264.2024.00050.

49 Xu Jiang, Dong Ji, Nan Guan, Ruoxiang Li, Yue Tang, and Yi Wang. Real-time scheduling and

analysis of processing chains on multi-threaded executor in ros 2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–39. IEEE, 2022. doi:10.1109/RTSS55097.2022.00013.

50 Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018. doi:10.1109/ICCPS.2018.00035.

51 Jackie Kay and Adolfo Rodriguez Tsouroukdissian. Real-time control in ros and ros 2.0. Online, 2015. URL: https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf.

52 Sohyun Kim, Juho Song, Kilho Lee, Sangeun Oh, and Hoon Sung Chwa. Cros-rt: Cross-layer priority scheduling for predictable inter-process communication in ros 2. In *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 202–214, 2025. doi:10.1109/RTAS65571.2025.00029.

53 Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard Fettweis. Latency analysis of ROS2 multi-node systems. In *international conference on multisensor fusion and integration for intelligent systems (MFI)*, pages 1–7. IEEE, 2021. doi:10.1109/MFI52462.2021.9591166.

54 Takahisa Kuboichi, Atsushi Hasegawa, Bo Peng, Keita Miura, Kenji Funaoka, Shinpei Kato, and Takuya Azumi. Caret: Chain-aware ros 2 evaluation tool. In *20th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–8. IEEE, 2022. doi:10.1109/EUC57774.2022.00010.

55 Jacky Kwok, Shulu Li, Marten Lohstroh, and Edward A Lee. Hprm: High-performance robotic middleware for intelligent autonomous systems. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8093–8099. IEEE, 2025. doi:10.1109/ICRA55743.2025.11128519.

56 Michael Larabel. Real-time "PREEMPT_RT" support merged for linux 6.12, 2024. Accessed: 2025-03-03. URL: https://www.phoronix.com/news/Linux-6.12-Does-Real-Time.

57 Hyoeun Lee, Youngjoon Choi, Taeho Han, and Kanghee Kim. Probabilistically guaranteeing end-to-end latencies in autonomous vehicle computing systems. *IEEE Transactions on Computers*, 71(12):3361–3374, 2022. doi:10.1109/TC.2022.3152105.

58 Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Softw. Pract. Exper.*, 46(6):821–839, June 2016. doi:10.1002/SPE.2335.

59 Ruoxiang Li, Zheng Dong, Jen-Ming Wu, Chun Jason Xue, and Nan Guan. Modeling and property analysis of the message synchronization policy in ros. In *International Conference on Mobility, Operations, Services and Technologies (MOST)*, pages 71–82. IEEE, 2023. doi:10.1109/MOST57249.2023.00016.

60 Ruoxiang Li, Nan Guan, Xu Jiang, Zhishan Guo, Zheng Dong, and Mingsong Lv. Worst-case time disparity analysis of message synchronization in ros. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 40–52. IEEE, 2022. doi:10.1109/RTSS55097.2022.00014.

61 Ruoxiang Li, Tao Hu, Xu Jiang, Laiwen Li, Wenxuan Xing, Qingxu Deng, and Nan Guan. Rosgm: A real-time gpu management framework with plug-in policies for ros 2. In *29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 93–105. IEEE, 2023. doi:10.1109/RTAS58335.2023.00015.

62 Ruoxiang Li, Xu Jiang, Zheng Dong, Jen-Ming Wu, Chun Jason Xue, and Nan Guan. Worst-case latency analysis of message synchronization in ros. In *Real-Time Systems Symposium (RTSS)*, pages 185–197. IEEE, 2023. doi:10.1109/RTSS59052.2023.00025.

63 Zihang Li, Atsushi Hasegawa, and Takuya Azumi. Autoware_perf: A tracing and performance analysis framework for ros 2 applications. *Journal of Systems Architecture*, 123:102341, 2022. doi:10.1016/J.SYSARC.2021.102341.

64 G. Lipari and E. Bini. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 151–158, 2003. doi:10.1109/EMRTS.2003.1212738.

65 Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

66 Sizhe Liu, Rohan Wagle, Shareef Ahmed, Zelin Tong, and James Anderson. ROSRT: Enabling flexible scheduling in ros 2. In *Proceedings of the 46th IEEE Real-Time Systems Symposium*, 2025.

67 Songran Liu, Xu Jiang, Nan Guan, Zilong Wang, Minghe Yu, and Wang Yi. Rtex: An efficient and timing-predictable multithreaded executor for ros 2. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(9):2578–2591, 2024. doi:10.1109/TCAD.2024.3380551.

68 Xiantong Luo, Xu Jiang, Nan Guan, Haochun Liang, Songran Liu, and Wang Yi. Modeling and analysis of inter-process communication delay in ros 2. In *Real-Time Systems Symposium (RTSS)*, pages 198–209. IEEE, 2023. doi:10.1109/RTSS59052.2023.00026.

69 Xiantong Luo, Xu Jiang, Haochun Liang, Yue Tang, Nan Guan, and Wang Yi. Flexible zero-copy ipc for processing chains in ros 2. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

70 Xiantong Luo, Xu Jiang, Yichen Liu, Nan Guan, Yue Tang, and Shaoshuai Zhang. On the scalability and efficiency of intra-process communication in ros 2. In *2025 IEEE Real-Time Systems Symposium (RTSS)*, pages 55–67. IEEE, 2025.

71 Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science robotics*, 7(66):eabm6074, 2022. doi:10.1126/SCIROBOTICS.ABM6074.

72 Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016. doi:10.1145/2968478.2968502.

73 Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, et al. High-performance deterministic concurrency using lingua franca. *ACM Transactions on Architecture and Code Optimization*, 20(4):1–29, 2023. doi:10.1145/3617687.

74 Giacomo Nabissi, Sauro Longhi, and Andrea Bonci. Ros-based condition monitoring architecture enabling automatic faults detection in industrial collaborative robots. *Applied Sciences*, 13(1):143, 2022.

75 Keisuke Nishimura, Takahiro Ishikawa, Hiroshi Sasaki, and Shinpei Kato. Raplet: Demystifying publish/subscribe latency for ros applications. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 41–50, 2021. doi:10.1109/RTCSA52859.2021.00013.

76 Object Management Group. Data Distribution Service (DDS) v1.4. https://www.omg.org/spec/DDS/1.4, 2015. OMG Specification.

77 Open Source Robotics Foundation. Robot Operating System (ROS). https://www.ros.org/, 2025. Accessed: 19 November 2025.

78 Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

79 Jonas Peeck, Mischa Möstl, Tasuku Ishigooka, and Rolf Ernst. A middleware protocol for time-critical wireless communication of large data samples. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–13, 2021. doi:10.1109/RTSS52674.2021.00013.

80 Jonas Peeck, Mischa Möstl, Tasuku Ishigooka, and Rolf Ernst. A protocol for reliable real-time wireless communication of large data samples. *IEEE Transactions on Vehicular Technology*, 72(10):13146–13161, 2023. doi:10.1109/TVT.2023.3275300.

81 Héctor Pérez and J Javier Gutiérrez. Modeling the qos parameters of dds for event-driven real-time applications. *Journal of Systems and Software*, 104:126–140, 2015. doi:10.1016/J.JSS.2015.03.008.

82 Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 515–520. IEEE, 2012. doi:10.1109/DATE.2012.6176523.

83 ROS 2 Community. Callback group priority or order. GitHub Issue, 2023. Accessed: 2025-09-09. URL: https://github.com/ros2/rclcpp/issues/2532.

84 ROS-Industrial Consortium. Ros-industrial: Applying the robot operating system (ros) to industrial applications. https://rosindustrial.org/, 2024. Open-source initiative for industrial robotics.

85 Yukihiro Saito, Futoshi Sato, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Rosch:real-time scheduling framework for ros. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 52–58, 2018. doi:10.1109/RTCSA.2018.00015.

86 Simon Schliecker and Rolf Ernst. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '09)*, pages 183–192. ACM, 2009. doi:10.1145/1629435.1629469.

87 Gerlando Sciangula, Daniel Casini, Alessandro Biondi, and Claudio Scordino. End-to-end latency optimization of thread chains under the dds publish/subscribe middleware. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024. doi:10.23919/DATE58400.2024.10546636.

88 Gerlando Sciangula, Daniel Casini, Alessandro Biondi, Claudio Scordino, and Marco Di Natale. Bounding the data-delivery latency of dds messages in real-time applications. In *35th Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ECRTS.2023.9.

89 Gerlando Sciangula, Daniel Casini, Alessandro Biondi, Claudio Scordino, and Marco Di Natale. End-to-end response-time analysis of dds-based real-time applications. *Internet of Things*, 36:101853, 2026. doi:10.1016/j.iot.2025.101853.

90 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13, 2003. doi:10.1109/REAL.2003.1253249.

91 Hoora Sobhani, Hyunjong Choi, and Hyoseung Kim. Timing analysis and priority-driven enhancements of ros 2 multi-threaded executors. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 106–118, 2023. doi:10.1109/RTAS58335.2023.00016.

92 Hoora Sobhani and Hyoseung Kim. Modeling and scheduling of fusion patterns in autonomous driving systems. In *Proceedings of the 33rd International Conference on Real-Time Networks and Systems (RTNS 2025)*, Pisa, Italy, 2025.

93 Andrea Stevanato, Alessandro Biondi, Alessandro Biasci, and Bruno Morelli. Virtualized dds communication for multi-domain systems: Architecture and performance evaluation of design alternatives. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 316–328. IEEE, 2023. doi:10.1109/RTAS58335.2023.00032.

94 I. Stoica and H. Abdel-Wahab. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical report, Old Dominion University, USA, 1995.

95 Jinghao Sun, Tianyi Wang, Yang Li, Nan Guan, Zhishan Guo, and Guozhen Tan. Seam: An optimal message synchronizer in ros with well-bounded time disparity. In *Real-Time Systems*

Symposium (RTSS), pages 172–184. IEEE, 2023. doi:10.1109/RTSS59052.2023.00024.

96 Yuhei Suzuki, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Real-time ros extension on transparent cpu/gpu coordination mechanism. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 184–192, 2018. doi:10.1109/ISORC.2018.00035.

97 Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ros2 executors. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–243, 2020. doi:10.1109/RTSS49844.2020.00030.

98 Yue Tang, Nan Guan, Xu Jiang, Xiantong Luo, and Wang Yi. Real-time performance analysis of processing systems on ros 2 executors. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 80–92. IEEE, 2023. doi:10.1109/RTAS58335.2023.00014.

99 Yue Tang, Xu Jiang, Nan Guan, Xiantong Luo, Maolin Yang, and Wang Yi. Timing analysis of processing chains with data refreshing in ros 2. *Journal of Systems Architecture*, 155:103259, 2024. doi:10.1016/J.SYSARC.2024.103259.

100 Harun Teper, Oren Bell, Mario Günzel, Chris Gill, and Jian-Jia Chen. Reconciling ros 2 with classical real-time scheduling of periodic tasks. In *31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 177–189. IEEE, 2025. doi:10.1109/RTAS65571.2025.00027.

101 Harun Teper, Tobias Betz, Mario Günzel, Dominic Ebner, Georg Von Der Brüggen, Johannes Betz, and Jian-Jia Chen. End-to-end timing analysis and optimization of multi-executor ros 2 systems. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 212–224, 2024. doi:10.1109/RTAS61025.2024.00025.

102 Harun Teper, Tobias Betz, Georg Von Der Brüggen, Kuan-Hsun Chen, Johannes Betz, and Jian-Jia Chen. Timing-aware ros 2 architecture and system optimization. In *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 206–215, 2023. doi:10.1109/RTCSA58653.2023.00032.

103 Harun Teper, Mario Günzel, Niklas Ueter, Georg von der Brüggen, and Jian-Jia Chen. End-to-end timing analysis in ros2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 53–65, 2022. doi:10.1109/RTSS55097.2022.00015.

104 Harun Teper, Daniel Kuhse, Mario Günzel, Georg von der Brüggen, Falk Howar, and Jian-Jia Chen. Thread carefully: Preventing starvation in the ros 2 multithreaded executor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3588–3599, 2024. doi:10.1109/TCAD.2024.3446865.

105 Ioan Ungurean and Nicoleta Cristina Gaitan. A software architecture for the industrial internet of things—a conceptual model. *Sensors*, 20(19):5603, 2020. doi:10.3390/S20195603.

106 Şerban Vădineanu and Mitra Nasri. Robust and accurate regression-based techniques for period inference in real-time systems. *Real-Time Systems*, 58(3):313–357, 2022. doi:10.1007/S11241-022-09385-8.

107 Víctor Mayoral Vilches. Hardware robot operating system (h-ros). https://www.ros.org/news/2016/10/hardware-robot-operating-system-h-ros.html. ROS Blog, accessed 19 November 2025.

108 Martin Škoudlil, Michal Sojka, and Zdeněk Hanzálek. A First Look at ROS 2 Applications Written in Asynchronous Rust. In *37th Euromicro Conference on Real-Time Systems (ECRTS 2025)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ECRTS.2025.1.

109 Zilong Wang, Songran Liu, Dong Ji, and Wang Yi. Improving real-time performance of micro-ros with priority-driven chain-aware scheduling. *Electronics*, 13(9):1658, 2024.

110 Zilong Wang, Songran Liu, Xu Jiang, Dong Ji, and Yi Wang. Tide: a timing-deterministic and efficient executor for micro-ros. In *2023 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, pages 487–494. IEEE, 2023. doi:10.1109/ITHINGS-GREENCOM-CPSCOM-SMARTDATA-CYBERMATICS60724.2023.00096.

111 Hongxing Wei, Zhen Huang, Qiang Yu, Miao Liu, Yong Guan, and Jindong Tan. Rgmp-ros: A real-time ros architecture of hybrid rtos and gpos on multi-core processor. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2482–2487, 2014. doi:10.1109/ICRA.2014.6907205.

112 Hongxing Wei, Zhenzhou Shao, Zhen Huang, Renhai Chen, Yong Guan, Jindong Tan, and Zili Shao. Rt-ros: A real-time ros architecture on multi-core processors. *Future Generation Computer Systems*, 56:171–178, 2016. doi:10.1016/j.future.2015.05.008.

113 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM transactions on embedded computing systems (TECS)*, 7(3):1–53, 2008. doi:10.1145/1347375.1347389.

114 Kurt Wilson, Abdullah Al Arafat, John Baugh, Ruozhou Yu, and Zhishan Guo. Physics-informed mixed-criticality scheduling for f1tenth cars with preemptable ros 2 executors. In *31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 215–227. IEEE, 2025. doi:10.1109/RTAS65571.2025.00030.

115 Chenhao Wu, Ruoxiang Li, Naijun Zhan, and Nan Guan. Improving the reaction latency analysis of message synchronization in ros. In *2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 43–48, 2024. doi:10.1109/RTCSA62462.2024.00016.

116 Chenhao Wu, Ruoxiang Li, Naijun Zhan, and Nan Guan. Modeling and analysis of the latest-

time message synchronization policy in ros. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3576–3587, 2024. doi:10.1109/TCAD.2024.3446709.

117 Zhengda Wu, Yixiao Feng, Mingtai Lv, Sining Yang, and Bo Zhang. Deadline-driven enhancements and response time analysis of ros2 multithreaded executors. In *European Conference on Parallel Processing*, pages 298–312. Springer, 2024. doi:10.1007/978-3-031-69577-3_21.

118 Zhengda Wu, Haining Hu, Sining Yang, and Bo Zhang. Hybrid scheduling executor and chain-aware thread mapping strategy for ros 2. In *2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 310–318, 2024. doi:10.1109/ISPA63168.2024.00047.

119 Jinsong Yang, Kristian Sandström, Thomas Nolte, and Moris Behnam. Data distribution service for industrial automation. In *Proceedings of 2012 IEEE 17th international conference on emerging technologies & factory automation (ETFA 2012)*, pages 1–8. IEEE, 2012. doi:10.1109/ETFA.2012.6489544.

120 Quan Zhou, Guohui Li, Jianjun Li, Chenggang Deng, and Ling Yuan. Response time analysis for tasks with fixed preemption points under global scheduling. *ACM Trans. Embed. Comput. Syst.*, 18(5), October 2019. doi:10.1145/3360513.

121 Matteo Zini, Filip Marković, Daniel Casini, Alessandro Biondi, and Björn B Brandenburg. In search of butterflies: Exceedance analysis for real-time systems under transient overload. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 229–242. IEEE, 2024. doi:10.1109/RTSS62706.2024.00028.