Internet of Things 36 (2026) 101853

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

End-To-end response-time analysis of DDS-based real-time applications

Gerlando Sciangula [a,b,*], Daniel Casini [a], Alessandro Biondi [a], Claudio Scordino [b], Marco Di Natale [a,b]

[a] Scuola Superiore Sant'Anna, Pisa, Italy
[b] Evidence SRL, Pisa, Italy

ARTICLE INFO

Keywords:
DDS
Real-time systems
End-to-end latency
Edge computing
Response times

ABSTRACT

The Data Distribution Service (DDS) is established as a middleware communication standard based on a data-centric publish-subscribe protocol. This standard is pivotal for applications in autonomous driving, smart cities, and Industry 4.0, facilitating communication among diverse devices across the IoT-to-Edge-to-Cloud continuum. Particularly in the automotive industry, modern autonomous systems, built on top of frameworks like ROS 2 and Autoware, heavily rely on DDS for real-time data exchange across distributed software components. The DDS is however typically implemented with a multithreaded software structure and leverages middleware-specific policies for message dispatching, posing considerable challenges in guaranteeing timing constraints. This paper fills significant gaps in the current understanding of DDS's real-time performance. We introduce a comprehensive DDS model that includes both synchronous and asynchronous communication under various dispatching policies. The model is then used to derive a holistic response-time analysis capable of bounding the end-to-end latency of DDS-enabled real-time applications. Furthermore, we integrate our analysis with a state-of-the-art executor-based analysis for ROS2-based systems. The effectiveness of our approach is validated through experiments on a real platform using FastDDS, a popular DDS implementation, and a modern automotive testbed taken from the WATERS 2019 Industrial Challenge by Bosch. Finally, our analysis method is evaluated with both a ROS2 case-study application and the Autoware reference system, a realistic testbed from the open-source Autoware.Auto framework for autonomous driving.

1. Introduction

The Object Management Group's (OMG) Data Distribution Service (DDS) [1] defines a middleware communication standard positioned between application software and network transport layers. The DDS alleviates the developers' effort to realize system-specific data distribution mechanisms and corresponding support. As a data-centric publish-subscribe protocol, DDS has gained traction with the advent of massively distributed applications in fields like autonomous driving [2–5], smart cities [6], and Industry 4.0 [7], since it facilitates communication among diverse computing devices across the so-called IoT-to-Edge-to-Cloud continuum [8]. The advancement of modern, complex, and autonomous systems, often designed as a network of interconnected and distributed software components, demands seamless integration and close cooperation akin to the operation of a single, cohesive entity. This trend is particularly prominent in the automotive industry, where sophisticated driver assistance systems (ADAS) – including functionalities

* Corresponding author.
E-mail address: gerlando.sciangula1@huawei.com (G. Sciangula).

https://doi.org/10.1016/j.iot.2025.101853
Received 18 March 2025; Received in revised form 5 June 2025; Accepted 10 December 2025
Available online 17 December 2025

such as lane-keeping, adaptive cruise control, and anti-lock braking systems – are distributed across multiple electronic control units (ECUs) and interconnected via in-vehicle networks. Many of these functionalities are based on frameworks like ROS 2 [9–11] and Autoware [5], which in turn are built on top of the DDS for publish-subscribe communications. Moreover, the AUTOSAR consortium has integrated DDS into the specifications of both their Adaptive and Classic platforms [12].

Typically, distributed real-time applications are implemented as chains of computations, facing with data dependencies between threads, which form the so-called *cause-effect thread chains* (or simply *thread chains*). A cause-effect chain represents a sequence of threads that are executed to achieve a given functionality. A typical example is a sensor-to-actuator cause-effect chain, which consists of a first thread that reads the sensor (cause), one or more threads that process the value produced by the preceding thread(s), and a final thread that writes the output to an actuator (effect). Often, such chains need to be completed within timing constraints, often referred to as *deadlines*. However, guaranteeing deadlines in this context requires accounting for DDS specificities, which span from the fundamentals of the DDS standard up to the peculiarities of specific implementations. For example, several DDS implementations rely on a complex multithreaded software architecture, requiring proper thread scheduling to achieve the desired timing performance. These threads handle various tasks, including synchronous or asynchronous message dispatching, listening, liveliness monitoring, and garbage collection. Custom, implementation-specific message queuing policies can further and severely affect message response times.

To properly support the designers of these application, it is essential to have an accurate model for DDS-enabled real-time systems to assess whether a given configuration can allow meeting a set of timing constraints *upfront*, without calling for trial-and-error testing campaigns that involve deploying applications on the target platform many times.

To this aim, we propose a fine-grained modeling and end-to-end response-time analysis for DDS-enabled real-time applications, focusing on the popular FastDDS implementation [13] when dealing with all implementation-specific aspects.

This paper extends a previous conference version [14] by: **(i)** expanding the model and analysis capabilities by supporting the round-robin policy of flow-controller threads (instead of fixed priority and FIFO only), **(ii)** supporting the DDS synchronous sending mode, **(iii)** allowing to bound the end-to-end latency of thread chains rather than computing only the data delivery latency of individual pairs of producers and consumers, **(iv)** discussing how to integrate the DDS-based analysis with a state-of-the-art analysis for ROS2, **(v)** considerably extending the evaluation to include new experiments, including a new case study on the Autoware autonomous driving framework [5] and other experiments that consider the integrated DDS-ROS2 analysis.

Contribution. This paper proposes a compositional model to describe the peculiarities of a complex DDS-based multithreaded application. The compositional model is based on the DDS specification only and is not tied to any specific implementation.

Later, we propose a Fast-DDS-specific instantiation of the model, which can more precisely model and capture the relevant timing aspects based on the specific implementation choices.

Building on the model, we propose a holistic response-time analysis to upper bound the end-to-end latency of thread-chains for FastDDS-based systems, under both synchronous and asynchronous sending mode. Additionally, we discuss how to integrate the DDS analysis with state-of-the-art real-time analysis methods for ROS 2.

Finally, we compare our analysis results with latency measurements observed by running a FastDDS-enabled use-case application on a real platform, demonstrating the efficacy of our methods. We further evaluate the approach based on two well-known realistic automotive test-beds, taken from the WATERS 2019 Industrial Challenge by Bosch [15,16] and the Autoware reference system [17], from the Autoware.auto autonomous driving framework.

2. Background

The structure of this section is as follows. First, we provide a review of the DDS standard, as defined by the Object Management Group (OMG). Second, we provide a description of the features offered by FastDDS, the reference implementation of the DDS considered in this work. Subsequently, a brief overview of the ROS 2 autonomous framework and its interaction with FastDDS is presented. Finally, to establish a foundation for the analysis presented later, we introduce some core concepts of the Compositional Performance Analysis (CPA) approach [18].

2.1. The DDS standard

The Data Distribution Service (DDS) standard offers a powerful publish-subscribe protocol for data exchange [1]. This approach centers on a shared pool of information called the *Global Data Space (GDS)*, accessible to all participating applications. Applications can act as either publishers, contributing data to the GDS, or subscribers, expressing interest in specific portions of the data space. DDS manages the flow of data between these roles, ensuring efficient dissemination. Whenever a publisher updates the GDS, the underlying middleware automatically broadcasts the new information to all relevant subscribers. This data exchange adheres to pre-defined Quality of Service (QoS) policies that can be configured at various levels for optimized communication [19].

DDS establishes a communication network for the distributed applications. Within this network, logical areas called *domains* act as interconnected channels. Two key elements facilitate communication within a domain: *topics* and *participants*. Topics function as unique identifiers, combining a descriptive name with the specific data type it represents and any associated QoS policies. Topics serve as channels dedicated to specific data types. Participants, on the other hand, are the entities that interact with the topic data. A participant can be a publisher, a subscriber, or even both. Publishers leverage DataWriter (DW) objects to disseminate information across various topics. Similarly, subscribers utilize DataReader (DR) objects to receive data from their chosen topics. Each DW and DR object maintains a one-to-one relationship with a single topic (refer to Fig. 1 for a visual representation of participant connections within a domain).

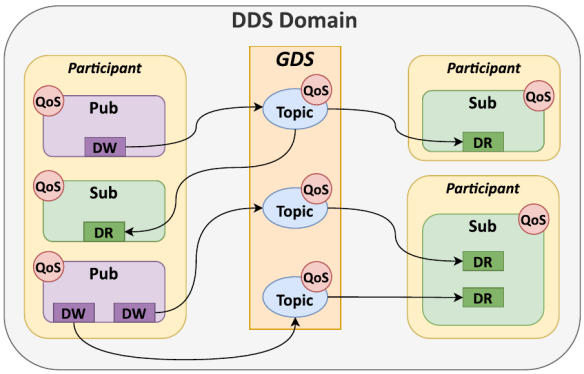

Fig. 1. Example of connections between DDS participants in a domain.

The Real-Time Publish-Subscribe Protocol (RTPS) [20] is underlying the communication within DDS. RTPS provides both reliable and best-effort communication mechanisms, even when utilizing unreliable transport protocols like UDP. This flexibility allows for operation in both unicast (one-to-one) and multicast (one-to-many) settings. It is important to note that, despite its name, RTPS does not inherently guarantee real-time behavior or timing constraints in general. DDS operates with a three-phase process. The first phase involves discovery, where participants locate each other on the network. During the matching phase, discovered participants determine if they should establish a publish-subscribe relationship based on their data needs. Finally, the data distribution phase sees the dissemination of data from publishers to the matching subscribers. This structured approach ensures efficient and targeted data exchange within the DDS framework.

2.2. The FastDDS implementation

One of the most popular and efficient C++ implementations of DDS is FastDDS by eProsima [21], which is currently the default option within the ROS 2 framework [22]. It employs a multi-threaded architecture to guarantee efficient and robust data communication.

At the heart of a publisher application lies the *publisher thread*, a user-level actor responsible for preparing application data and publishing it onto designated topics. The publishing process itself exhibits flexibility, offering two distinct sending modes: *synchronous* and *asynchronous*. In synchronous mode, the publisher thread directly transmits the data. The asynchronous mode, on the other hand, introduces a separate internal thread known as the *flow-controller thread*. This thread takes over the network transmission duties, freeing the publisher thread to focus on preparing more data. The flow-controller thread operates at the middleware level, constantly monitoring a message queue. Whenever new data arrives in the queue, the flow-controller thread extracts it and transmits it over the network according to three different policies, i.e., `FIFO`, `RR` (i.e., round-robin), or `HIGH_PRIORITY` (i.e., fixed priority). Notably, publishers can leverage multiple flow controllers if they manage data publication across various topics, further enhancing scalability.

Beyond the core data publishing functionality, publisher applications also rely on a dedicated middleware-level *event thread*. This thread processes events such as **(i)** discovery events (participants locating each other on the network), **(ii)** matching events (participants determining if they should establish a data exchange relationship), and **(iii)** QoS verification events (ensuring the data delivery adheres to pre-defined quality of service parameters). Additionally, FastDDS provides a *meta-traffic listener thread* for receiving discovery information, which is required for establishing communication paths between participants within the DDS network.

Subscriber applications mirror the publisher application thread structure. A subscriber thread, operating at the user level, acts as the information receiver within the system. The subscriber thread's primary function revolves around reading and interpreting data that arrives from topics of interest. Similar to publisher applications, subscriber applications also incorporate a dedicated middleware-level event thread. This thread works identically to its counterpart in the publisher application, ensuring consistent event handling across both sides of the communication spectrum.

Another key thread within subscriber's applications is the *user-traffic listener thread*. This thread plays a critical role in managing the reception of user data, the very essence of the communication. It listens for incoming data packets containing the application data published by various sources. This data is processed and delivered to the corresponding subscriber thread. FastDDS empowers a versatile data exchange pattern by enabling many-to-many communications. This means that multiple publisher threads can publish data on the same topic concurrently. Likewise, multiple subscriber threads can subscribe to a specific topic, allowing them to receive and process the published data.

The transport layer acts as the communication backbone for DDS entities [13]. It assumes the responsibility of sending and receiving messages over the chosen physical transport protocol, which can be UDP, TCP, or shared memory. Notably, a dedicated listener thread is spawned for each reception channel. The definition of this channel depends on the specific transport layer protocol utilized, ensuring optimal performance and resource utilization based on the chosen communication method.

Thread chains. FastDDS leverages an event-driven (data-driven) design style, leading to data dependencies and potentially long processing chains. A processing chain is composed of coordinated threads that exchange information to achieve a specific behavior.

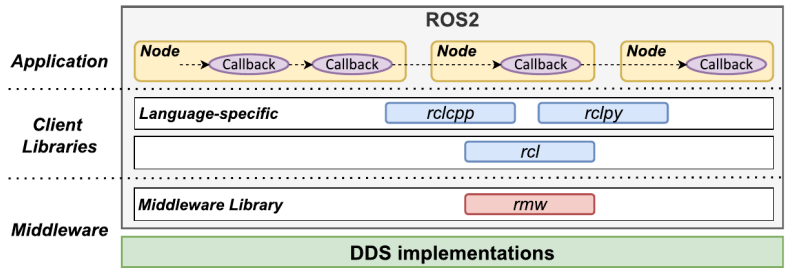

Fig. 2. ROS2 architecture.

FastDDS facilitates application design through these chains, enabling a subscriber thread to receive data and subsequently publish it to another subscriber. Each chain starts with a *source* publisher thread and ends with a *sink* subscriber thread. Middleware-level threads are involved in the middle of the chain.

2.3. ROS 2

ROS 2 is a framework for developing autonomous applications on top of an operating system. An application consists of individual nodes running on separate processes across various hosts. Each node can be composed of multiple publishers and subscribers that send and receive messages to/from specific topics, leveraging a specific DDS implementation for the message exchange. When a node receives a message, it triggers a callback function for processing, i.e., subscriber callback. Callbacks can also publish new messages, i.e., thus acting both as a subscriber and a publisher.

ROS2 provides a unified stack that integrates multiple layers of abstraction, as depicted in Fig. 2. At its core, ROS2 applications rely on language-specific client libraries–*rclcpp* for C++ and *rclpy* for Python–both of which manage callback execution within processes. These client libraries interface with the ROS client library (*rcl*), which offers a consistent set of APIs to ensure uniform behavior across programs written in different languages. Below this, the ROS middleware layer (*rmw*) serves as a communication interface between *rcl* and the underlying DDS implementations provided by specific vendors.

Recently, an additional *rmw* implementation that integrates *Zenoh* [23] as underlying communication protocol has been included in ROS2, to provide a more flexible communication for ROS2 nodes, especially in environments where traditional DDS-based *rmws* may face limitations [24]. This paper focuses, instead, on DDS-based solutions that are enabled by default.

ROS2 exhibits a unique scheduling policy for handling callbacks, which differs from the traditional scheduling models found in classical operating systems (OSes). The policy is enforced by an abstraction of an OS process called *executor*, which is in charge of managing the callbacks (e.g., C++ functions) that implement the functional behavior of ROS2 nodes.

ROS 2 offers two types of executors: i) single-threaded, which processes callbacks sequentially, and ii) multi-threaded, which distributes callbacks across multiple threads. This paper considers the single-threaded executor, which is also the option enabled by default.

The custom policy leverages a cache of ready callbacks, which is updated by interacting with the communication layer when the cache becomes empty. The cache can contain at most one active instance of each callback. The executor processes the cache and executes the contained callbacks in *priority order*.[1] The presence of the cache of active callbacks gives rise to several challenges in the real-time analysis. The first formal analysis to study the ROS2's executor-based scheduling behavior, aimed at ensuring real-time guarantees, was conducted by Casini et al. in [10].

Furthermore, ROS 2 supports two primary communication modes: intra-process and inter-process. Intra-process communication allows direct message exchange between publishers and subscribers within the same executor, leveraging shared-memory communications. Since this method does not involve DDS, it is not considered in this work. Inter-process communication, on the other hand, occurs between publishers and subscribers that are managed by different executors, either on the same machine (local) or separate machines (remote). This communication can be synchronous or asynchronous, depending on the system's needs.

Synchronous communication requires the ROS 2 callback to wait until the DDS layer finishes sending the message; conversely, asynchronous communication prioritizes efficiency and allows publishers to continue even if the DDS has not yet sent a message.

2.4. ROS2-FastDDS interaction

A notable design decision in ROS 2 is its avoidance of the traditional DDS publisher mechanism, which is typically used to manage message publication for multiple DataWriters. Instead, ROS 2 directly accesses individual DataWriters. According to this design, each ROS 2 publisher is strictly linked to a single DataWriter, and likewise, each ROS 2 subscriber is connected to a single DataReader. This design contrasts with FastDDS, where a single publisher or subscriber can manage multiple DWs or DRs. By opting for a more granular,

[1] In ROS2, the priority depends on the callback type (timers, subscriptions, services, and clients, prioritized in this order), and ties are broken according to the registration order.

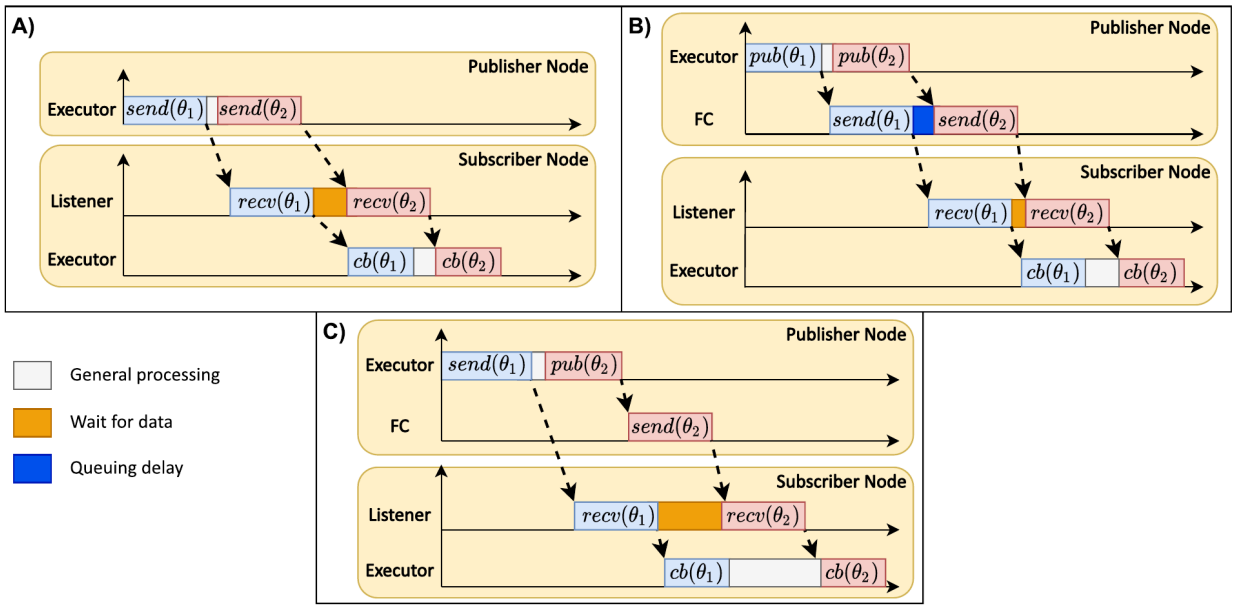

Fig. 3. A) Sync-Sync scenario. No flow-controller thread is spawned, hence messages are sent by the publisher node in the context of the executor. B) Async-Async scenario. The executor prepares the messages to send. Messages are then inserted in the queue of pending messages shared with the flow-controller, notifying it. Subsequently, messages are sent in the context of flow-controller thread. C) Async-Sync scenario. The first publisher data is sent directly through the network, in the context of the executor, while data related to the second publisher are first processed by the flow-controller and then sent through the network.

one-to-one relationship between publishers and DWs (as well as between subscribers and DRs), ROS 2 simplifies the structure of the communication flow. Specifically, this is beneficial in complex robotic systems, where predictable and independent communication channels are often critical for real-time, distributed computing.

As already introduced, FastDDS provides two primary publication modes: synchronous and asynchronous. The middleware implementation of FastDDS within ROS 2 is named `rmw_fastrtps`. It offers a simple means to tune the publication mode through an application configuration XML file. The XML file accepts the following values to set the publication mode of each publisher:

1. `ASYNCHRONOUS`: When an executor serves a callback that initiates a publish operation, the data is copied into a queue. Subsequently, the insertion in the queue is notified, relinquishing thread control back to the user before the actual transmission of data. An asynchronous thread (i.e., flow-controller thread) manages the data consumption from the queue and dispatches data to all matched readers.
2. `SYNCHRONOUS`: This mode facilitates synchronous publication. When enabled, data is directly transmitted within the executor's context. Consequently, any blocking calls during a write operation would stall the executor thread, potentially interrupting the application's flow. This mode tends to achieve higher throughput rates and lower latencies due to the absence of notifications and thread context switching. When a publisher sends messages related to different topics, the sending operation is done sequentially, one message per topic at a time. In this case, it is not possible to prioritize sending one message over another. However, an order can be foreseen at the application code level, such that a message is sent first, while all the others will have to wait until the sending operation is completed.

Tracing ROS2 publisher and subscriber applications. To test the two modes, we implemented a simple ROS2 example that consists of two applications, using ROS2 *Iron Irwini release* [25] upon FastDDS *v2.14.0* [26]. The first one includes a node with two publishers and the second one a node that comprises two subscribers. The applications communicate by exchanging data through two topics (θ_1, θ_2). Each publisher publishes a message per topic periodically every 500 ms. Our goal here is to understand which thread is in charge of sending messages under `SYNCHRONOUS` or `ASYNCHRONOUS` publishing modes. To this end, we tested three different scenarios:

- **Scenario A)**: both publishers publish using the `SYNCHRONOUS` publishing mode. In this case, as shown in Fig. 3A, all the messages related to the two topics are processed within the context of the executor of the publisher node.
- **Scenario B)**: both publishers publish using `ASYNCHRONOUS` publishing mode. In this case, all the messages related to the two topics are sent within the context of the flow-controller thread, as depicted in the Fig. 3B.
- **Scenario C)**: the publisher related to θ_1 publishes data using `SYNCHRONOUS` mode, while the other publisher, attached to topic θ_2, sends data under `ASYNCHRONOUS` mode. From the Fig. 3C, we can observe that data for θ_1 is sent in the context of executor, whereas data related to θ_2 is sent in the context of the flow-controller thread.

Hence, whenever the asynchronous mode is enabled for one or more publishers, automatically, the FastDDS layer spawns the flow-controller thread, managing data coming from the specific publisher(s). Instead, listener threads at the receiving side are used in both

communication modes. Note that the graphs reported in Fig. 3 are based on traces obtained using *LTTng* tracing framework [27], analyzed by means of the *Trace Compass* tool [28].

2.5. Compositional performance analysis

CPA [18] is a powerful framework for evaluating the timing behavior of complex real-time systems. These systems are often heterogeneous, meaning they consist of diverse components, and distributed, meaning they operate across a network.

CPA breaks down the analysis into two core elements: workloads and computational resources. Workloads represent the tasks the system needs to execute, modeled as a directed acyclic graph (DAG) of communicating tasks. Groups of tasks are assigned to specific resources, which define the scheduling policy for those tasks and their processing time, i.e., supply time. One of CPA's key concepts is the *event arrival curve*, denoted as $\eta(\Delta)$, which represents an upper bound on the number of release events for a task within any time interval of length Δ, i.e., $[t, t+\Delta)$ for any t. For the first task in a chain (source task), the arrival curve is provided externally. Subsequent (non-source) tasks are triggered based on the completion times of their predecessors, taking into account any potential delays caused by those predecessors and any network-related delay (e.g., representing such a delay as a release jitter). This creates a cyclic dependency: to bound the worst case response times (WCRTs), arrival curves for all tasks are required, but those of non-source tasks also require WCRTs bounds to be computed. The cycle is broken [14,18] by initially setting the release jitter to zero and computing WCRT bounds iteratively until convergence is reached.

To analyze overall system performance, CPA initially calculates the maximum response time for each individual task. The sum of these individual response times provides a basic estimate of the end-to-end latency (total response time) for a processing chain. CPA also offers extensions for specific scenarios that can improve the precision of these latency estimations [29]. This approach allows for a modular analysis, where the timing behavior of individual components is assessed and then combined to understand the overall system performance.

3. A compositional DDS model

Before delving into the various implementation details studied to analyze the concrete behavior of a DDS instance, we start by providing a general and compositional DDS model. This model is built based solely on the DDS specifications, making it adaptable across different DDS implementations. As a concrete example of an instance that can be derived from the compositional model, we show how it applies to the FastDDS implementation, which is studied in this work. Specifically, by leveraging a *compositional* approach, we show how to support both the asynchronous and synchronous sending modes specified by FastDDS.

DDS functionalities can be divided into fundamental building blocks called Logic Functional Blocks (LFBs). Each LFB represents a basic DDS operation. These LFBs fall into two categories:

1. *Principal blocks*, which are directly involved in data exchange between publishers and subscribers, and
2. *Auxiliary blocks*, which support middleware functionalities like discovery/matching (finding participants in the network), enforcing quality-of-service (QoS) requirements, or other features specific to a particular DDS implementation. Examples include FastDDS's timed-event handling and Eclipse CycloneDDS's garbage collector and liveliness monitoring [30].

The principal blocks for data exchange include:

(i) the *Publisher* and *Subscriber* blocks, which handle, respectively, publishing and subscribing operations initiated by user-level application threads.
(ii) the *Outgoing Flows Dispatching (OFD)* block, that takes data from the Publisher block and controls how it is published over the network (notably, the DDS standard does not dictate how network data dispatching should be implemented);
(iii) the *Incoming Flows Dispatching (IFD)* block that manages incoming messages received by the Network block and delivers them to the Subscriber block, and
(iv) the *Network* block, that represents the functionalities of a network protocol, responsible for transmitting data across a communication link between source and destination nodes.

FastDDS instance of the model. Now, we show how this general model applies specifically to FastDDS, when both asynchronous and synchronous sending modes are specifically considered. Fig. 4 illustrates this mapping. Each thread identified within FastDDS is assigned to its corresponding LFB in the model.

In this paper, we consider a static network of publishers and subscribers, meaning that no new publisher or subscriber can join at run-time and the topology of the network of participants is known a priori. We consider real-time applications that require predictable timing guarantees; in such settings, FastDDS's *Static Discovery* is preferred. After the initial discovery phase, and by configuring a high discovery announcement period for each participant [13], the discovery meta-traffic is extremely low at runtime. Consequently, the extra cost of the meta-traffic listener and event threads–responsible for processing discovery heartbeats, participant announcements, and QoS checks–becomes negligible in steady state. We therefore classify these threads as *auxiliary services*, grouping them under the auxiliary services block in Fig. 4 and excluding them from the system model in Section 4.

Publisher and subscriber threads are mapped directly to their respective LFBs (*Publisher* and *Subscriber* blocks). The choice of sending mode (synchronous or asynchronous) affects the *OFD* block. Under synchronous mode, the publisher thread handles both data publishing and sending, thus no separate thread is needed in the OFD block. However, when asynchronous mode is enabled,

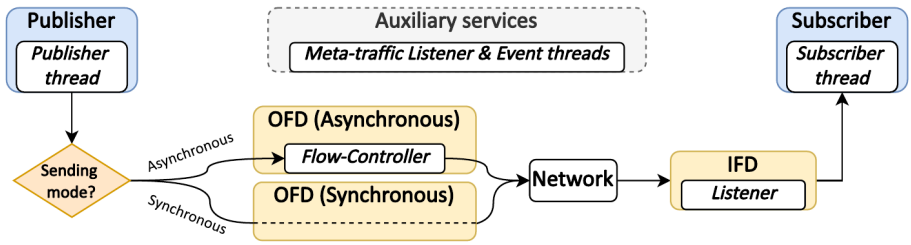

Fig. 4. Instantiation of FastDDS threads on DDS model under both asynchronous and synchronous sending mode.

the flow-controller thread (discussed in Section 2.2) takes over data sending and resides in the *OFD* block. Similarly, the user-traffic listener thread (simplified to *listener* for brevity) is mapped to the *IFD* block. Finally, the functionalities of the chosen transport protocol and the network itself are mapped to the *Network* block.

4. FastDDS-based system model and problem definition

We consider a distributed system consisting of $\mathcal{N}$ interconnected computing platforms (i.e., machines), which overall offer a set C of processing cores. Next, we model application and middleware threads for FastDDS only, leaving out ROS2 callbacks, which will be introduced later on.

Thread model. For a FastDDS-based system, we identify four classes of threads: *publisher*, *flow-controller*, *listener*, and *subscriber*, contained within the sets Γ_{p}, Γ_{f}, Γ_{l}, and Γ_{s}, respectively. These threads are managed using a partitioned fixed-priority scheduler, ensuring that each thread is statically allocated to a specific core. An arbitrary thread from each category is represented as $\tau_i^{\mathrm{p}} \in \Gamma_{\mathrm{p}}$, $\tau_i^{\mathrm{f}} \in \Gamma_{\mathrm{f}}$, $\tau_i^{\mathrm{l}} \in \Gamma_{\mathrm{l}}$, or $\tau_i^{\mathrm{s}} \in \Gamma_{\mathrm{s}}$, respectively. The collective set of all threads in the system is denoted by $\Gamma_{\mathrm{all}} = \{\Gamma_{\mathrm{p}} \cup \Gamma_{\mathrm{f}} \cup \Gamma_{\mathrm{s}} \cup \Gamma_{\mathrm{l}}\}$. Note that it could be that $\Gamma_{\mathrm{p}} \cap \Gamma_{\mathrm{s}} \neq \emptyset$, since some threads can be both publishers and subscribers. When the specific type of thread is irrelevant or clear from the context, it is simply referred to as τ_i.

Threads are also grouped into two sets: **(i)** *application-* and **(ii)** *middleware- level* threads. The former set includes all publishers and subscribers. The latter set, comprising flow-controller and listener threads, is denoted by $\Gamma_{\mathrm{mw}} = \Gamma_{\mathrm{f}} \cup \Gamma_{\mathrm{l}}$. Furthermore, the set Γ_{all}^k includes all threads running on core $c_k \in C$, while $\Gamma_{\mathrm{mw}}^k \subseteq \Gamma_{\mathrm{all}}^k$ represents the subset of middleware threads on c_k. Each thread $\tau_i \in \Gamma_{\mathrm{all}}$ is associated with a unique fixed priority. The notations $\mathrm{hp}_{oth}^k(\tau_i) \subseteq \Gamma_{\mathrm{all}}^k \setminus \Gamma_{\mathrm{mw}}^k$ and $\mathrm{hp}_{\mathrm{mw}}^k(\tau_i) \subseteq \Gamma_{\mathrm{mw}}^k$ denote the sets of non-middleware and middleware threads, respectively, on core c_k with priorities higher than τ_i.

Topic and message model. To define the logical communication channels between publisher and subscriber applications, a set of topics Θ is introduced. Each topic $\theta_j \in \Theta$ has a unique priority π_j across the system, independent of thread priorities. This priority is inherited by any message instance $m_z(\tau_i^{\mathrm{p}}, \theta_j)$ published by a publisher thread τ_i^{p} on topic θ_j. When identifying the publisher thread and topic is unnecessary, the message is simply denoted as m_z. A message instance m_z is considered pending in a middleware-level thread $\tau_i \in \Gamma_{\mathrm{mw}}$ from its release until its processing is completed. The set of messages associated with a topic θ_j is denoted by $\mathcal{M}(\theta_j)$. Each instance of a publisher thread τ_i^{p} can send up to w_i^j messages to topic θ_j, while $\Theta(\tau_i^{\mathrm{p}})$ and $\Theta(\tau_j^{\mathrm{s}})$ represent the subset of topics from which a publisher thread τ_i^{p} and a subscriber thread τ_j^{s} can send and receive messages, respectively. Similarly, $\Theta(\tau_i^{\mathrm{f}})$ and $\Theta(\tau_j^{\mathrm{l}})$ represent the subset of topics managed by, respectively, a flow-controller thread τ_i^{f} and a listener thread τ_j^{l}. The number of subscribers interested in a message m_z is denoted by $N_{\mathrm{sub}}(m_z)$.

Association among threads. A publisher thread τ_i^{p} can be linked to multiple flow-controller threads $\tau_i^{\mathrm{f}} \in \Gamma_{\mathrm{f}}$ if it publishes to multiple topics in asynchronous sending mode. Otherwise, the publisher thread directly handles data sending. A subscriber thread τ_j^{s} is uniquely associated with a listener thread τ_j^{l}, which manages messages from various topics. Each pair $(\tau_i^{\mathrm{p}}, \theta_j)$, and thus each message $m_z(\tau_i^{\mathrm{p}}, \theta_j)$, is associated with a single flow-controller and listener thread, if the asynchronous sending mode is used. Otherwise, we have a direct association between τ_i^{p} and the listener thread. With a slight abuse of notation, we also use $m_z \in \tau_i^t$, where $t \in \{\mathrm{f}, \mathrm{l}\}$, to indicate that message m_z is handled by middleware thread τ_i^t.

Thread chains. Thread chains, represented by the set $\mathcal{G}$, are sequences of communicating threads, with the first thread acting as a source publisher and the last one as a sink subscriber. Different from most works in classical real-time systems literature [29,31,32], when dealing with DDS-enabled systems, thread chains do not include application-level threads only (i.e., publishers and subscribers) but also middleware-level threads. As discussed, the kind of middleware-level threads change depending on the sending mode, i.e., synchronous or asynchronous. To abstract this complexity, we introduce the concept of *subchain*: each chain $\gamma_i \in \mathcal{G}$ can include k subchains, i.e., $\gamma_i = \{\xi_1^t, \ldots, \xi_k^t\}$, where $t \in \{\mathtt{async}, \mathtt{sync}\}$. When the specific type of subchain is irrelevant or clear from the context, it is simply referred to as ξ_i. Each subchain ξ_i is defined as tuples of threads. In asynchronous sending mode, the symbol $\xi_i^{\mathtt{async}}$ denotes an asynchronous subchain that includes a publisher, flow-controller, listener, and subscriber. In synchronous mode, the flow-controller is not involved, so that the publisher is directly linked to the listener thread of the subscriber, thereby composing a synchronous subchain denoted by the symbol $\xi_i^{\mathtt{sync}}$.

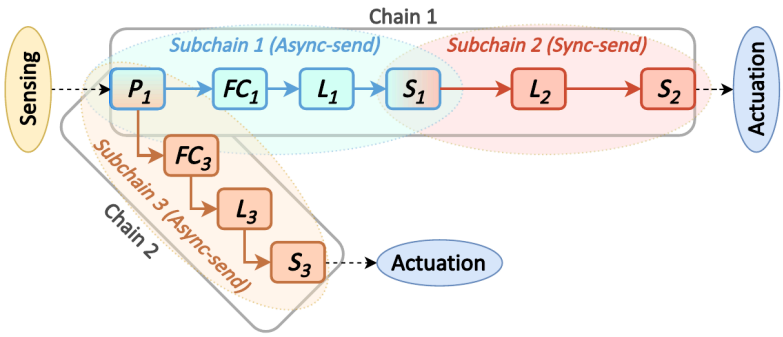

Fig. 5. Two chains activated by the same publisher thread. Chain 1 contains two subchains: Subchain 1 (where P_1 sends data asynchronously) and Subchain 2 (where S_1 communicates with S_2 by means of a sychnronous communication).

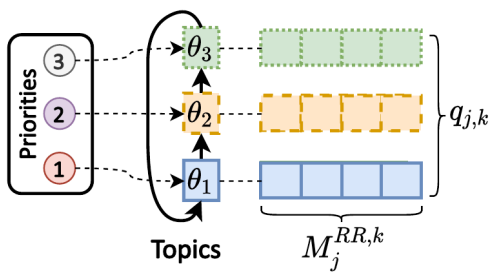

Fig. 6. Round-Robin scheduling policy. A priority is assigned to each topic, where lower value denotes higher priority. Each topic θ_k is then paired with its own message queue $q_{i,k}$ of size $M_j^{\mathrm{RR},k}$. Queues are processed in a circular order, following a sequence that respects the priority of each topic.

Two consecutive subchains of a chain, ξ_i and ξ_{i+1}, are linked by a subscriber $\tau_j^{\mathrm{s}} \in \Gamma_{\mathrm{s}}$ of the first subchain ξ_i acting as publisher for the second ξ_{i+1}, so that τ_j^{s} is also a publisher with respect to ξ_{i+1}, i.e., $\tau_j^{\mathrm{s}} \in \Gamma_{\mathrm{p}}$, too. The set of all chains forms a directed acyclic graph (DAG) of threads (vertices) and communications (edges), with multiple sources and sinks, as shown in Fig. 5.

Execution times and activations. This work employs a discrete-time model, with all time parameters as integer multiples of a basic time unit $\epsilon \triangleq 1$. Execution times vary across thread types. Symbols $\delta^{\mathrm{f}}(m_z)$, $\delta^{\mathrm{s}}(m_z)$, and $\delta^{\mathrm{l}}(m_z)$ indicate the worst-case time required, without any interference from other messages or threads (i.e., isolation), to process a message m_z in the flow-controller (for asynchronous sending), publisher (for synchronous sending), and listener, respectively. These processing times are all affected by the message size, hence their dependency on m_z. The application-level threads $\tau_j \in \Gamma_{\mathrm{all}} \setminus \Gamma_{\mathrm{mw}}$ have a worst-case execution time (WCET) e_j. When a publisher sends data synchronously, its WCET e_j also accounts for the time needed to complete the send operation for each topic, denoted by e_j^{sync}.

Publisher *source* threads (i.e., those that originate at least one chain) are characterized by externally-provided event arrival curves $\eta_i^p(\Delta)$, bounding the number of instances released in any interval of time Δ. All the other application-level threads' instances are activated by following data-driven patterns, which are characterized by derived arrival curves $\eta_j(\Delta)$. Derived arrival curves depend on the arrival patterns of predecessor threads and processing delays. With $\eta_{z,i}^{\mathrm{f}}(\Delta)$ and $\eta_{z,j}^{\mathrm{l}}(\Delta)$ we denote the arrival curve of each message within flow-controller (if any) and listener thread, respectively. Finally, the worst-case network propagation delay is denoted by $\delta_{\mathtt{net}}^{\nu_i,\nu_j}(m_z)$ for a message m_z from machine ν_i to ν_j. It can be either estimated experimentally or analytically bounded, depending on the underlying network [33–35].

Flow-controller scheduling policies. Flow-controller policies are denoted by `HP`, `F`, and `RR`, corresponding to `HIGH_PRIORITY`, `FIFO`, and `ROUND_ROBIN` policies, respectively. Messages of the same priority within a flow controller are processed in FIFO order. For `HP` policy, sets $hp_i(m_z)$, $ep_i(m_z)$, and $lp_i(m_z)$ denote messages with higher, equal, and lower priority than m_z in τ_i, respectively. Under `RR` policy, each topic θ_k is associated with a message queue $q_{i,k}$, processed in circular order according to topic priority (see Fig. 6).

Queues. Listener threads manage one network socket each, processing incoming messages from various topics in FIFO order. For FIFO policy, each flow-controller or listener thread $\tau_j^{\mathrm{t}} \in \Gamma_{\mathrm{mw}}$ maintains a single queue of up to M_j^{F} messages. Under `HP` policy, each priority level i has a queue of size $M_j^{\mathrm{HP},i}$. For `RR` policy, $M_j^{\mathrm{RR},k}$ represents the queue size for topic θ_k within middleware thread τ_j^{t}. Please, note that `HP` and `RR` are only used for flow-controller threads, while `FIFO` is also used in listener threads (the details are provided later). Buffers must be sufficiently large to prevent message loss on both sender and receiver sides.

Supply-bound function. The analysis relies on a supply-bound function $sbf_k(\Delta)$, representing the minimum processor service time provided by core $c_k \in C$ in any time window of length Δ, [36–38]. This function supports analysis extensibility with reservation-based scheduling mechanisms [39], such as those available in Linux (`SCHED_DEADLINE` scheduling class) [40] and QNX (Adaptive Partitioning Scheduler) [41], and generalizes to schedulers without reservation by setting $sbf_k(\Delta) = \Delta$.

Table 1
Table of main symbols.

Symbol	Description
c_j	jth physical core
ν_j	jth machine of a network of interconnected platforms
τ_i^t	ith thread of type $t \in \{\mathtt{p}, \mathtt{f}, \mathtt{l}, \mathtt{s}\}$
Γ_t	set of threads of type t
Γ_{all}	set of all threads in the system
Γ_{mw}	set of middleware threads in the system
Γ_t^k	threads of type t on c_k
$\mathtt{hp}_{mw}^k(\tau_i)$	middleware threads with priority higher than τ_i on c_k
$\mathtt{hp}_{oth}^k(\tau_i)$	non-middleware threads with priority higher than τ_i on c_k
Θ	set of topics
θ_j	jth topic
π_j	priority of the topic θ_j
$m_z(\tau_i^p, \theta_j)$	zth *message* published by τ_i^p over θ_j
w_i^j	max number of *messages* to θ_j for each τ_i^p instance
$\mathcal{M}(\theta_j)$	*messages* for a topic θ_j
$\Theta(\tau_j^p)$	topics to which τ_j^p publishes on
$\Theta(\tau_j^s)$	topics from which τ_j^s subscribes to
$N_{sub}(m_z)$	number of *subscribers* interested in m_z
$\mathcal{G}$	set of thread chains in the system
γ_i	ith thread chain
ξ_i^t	ith subchain of type $t \in \{\mathtt{async}, \mathtt{sync}\}$
$rbf_i(\Delta)$	request-bound function of τ_i
e_j	WCET of τ_j^t, $t \in \{\mathtt{p}, \mathtt{s}\}$
$sbf_k(\Delta)$	supply-bound function of c_k
$\eta_i^t(\Delta)$	τ_i^t arrival curve, $t \in \{\mathtt{p}, \mathtt{s}\}$
$\eta_{z,i}^t(\Delta)$	arrival curve of m_z in τ_i^t, $t \in \{\mathtt{f}, \mathtt{l}\}$
$\delta^s(m_z)$	processing time of m_z in τ^p
$\delta^f(m_z)$	processing time of m_z in τ^f
$\delta^l(m_z)$	processing time of m_z in τ^l
$\delta_{\mathtt{net}}^{\nu_i,\nu_j}(m_j)$	network propagation delay for a m_z from machine ν_i to ν_j

Table of symbols. Table 1 summarizes the main symbols introduced in this paper.

4.1. Problem statement

This paper aims to bound the end-to-end latency of a chain of threads using the DDS to communicate in a data-driven manner. We decompose this problem into two subproblems: first, we study the *Data Delivery Latency* of two communicating threads (a publisher and a subscriber), which includes all the additional scheduling delays due to the DDS middleware. Second, we derive worst-case response times of threads with the data delivery latency to obtain an overall end-to-end latency bound.

To begin, we provide the definitions of two crucial metrics.

Definition 1 (Data Delivery Latency). The Data Delivery Latency (DDL) $L_z(\xi_i)$ experienced by a message m_z, sent by a publisher thread to a matching subscriber thread, is the longest time span elapsed between the time instant when m_z is sent by the publisher and the time instant when the corresponding instance of the subscriber is released, within a subchain ξ_i.

Note that the DDL does not include the time in which the target instance of the subscriber is waiting to be scheduled.

Definition 2 (End-to-End Latency). The End-to-End (E2E) latency $L^{\mathtt{e2e}}(\gamma_i)$ of a thread chain γ_i is defined as the longest time span elapsed between the release of the source thread of the chain to when the sink thread of γ_i completes.

Next, we build upon these definitions to provide a comprehensive method for bounding the worst-case *data delivery latency* of any DDS message within a subchain and the *end-to-end latency* of thread chains under both asynchronous and synchronous modes.

4.2. Thread behavioral rules

In this section, we detail the behavior of the FastDDS middleware implementation, formalized through a set of comprehensive rules that describe the interactions between the identified threads under both synchronous and asynchronous sending modes, within a subchain.

Rule 1. ***Pub-to-Net (sync-mode).*** *In synchronous sending mode, a publisher thread is responsible for generating the data to be sent. Upon invoking the* `publish` *operation, the data undergoes RTPS serialization. Subsequently, the serialized data is transmitted to each interested subscriber through a blocking network send operation. This sending process is conducted sequentially per topic, strictly adhering to the order specified by the application code. The total number of send operations directly corresponds to the number of message copies required for each subscriber, which is quantified by the parameter* $N_{sub}(m_z)$.

Rule 2. ***Pub-to-Flow (async-mode)***. *In asynchronous sending mode, non-blocking publish operations are used. When a publisher thread needs to send data, it initiates a* `publish` *operation by just notifying the corresponding flow-controller thread. The flow-controller then manages the actual data transmission.*

Rule 3. ***Flow-to-Net (async-mode)***. *If the flow-controller's queue of pending messages is not empty, it extracts the message from the head of the queue, performs RTPS serialization, and sends the message to each interested subscriber. Similar to the synchronous mode, the number of send operations matches the parameter* $N_{sub}(m_z)$*, ensuring each subscriber receives the required number of message copies. If the queue is empty, the flow-controller thread enters a blocked state, awaiting notification from the associated publisher thread indicating the arrival of new data.*

Rule 4. ***Net-to-List***. *Listener threads perform a blocking* `receive` *operation on a designated network socket. When the system network functionalities write a message to the socket buffer, the listener thread is activated.*

Rule 5. ***List-to-Sub***. *The Listener thread processes incoming messages by retrieving them from the socket buffer in a first-in-first-out (FIFO) manner. The message is deserialized and subsequently delivered to the appropriate subscriber thread, which is notified of the new message's arrival.*

Rule 6. ***Non-preemptiveness (async-mode)***. *The sending operation within a flow controller adheres to a non-preemptive policy. Once a message is extracted from the queue, it is fully processed and transmitted, along with all its copies to different subscribers, before addressing any newly arrived higher priority messages. This ensures that once the sending process begins, it is completed without interruption. For clarity, in this context, the term* ***non-preemptive*** *refers to the atomicity of the message-send operation within a flow-controller thread only: once the transfer begins, no other send call* in the same thread *can interrupt it (i.e., it runs to completion). It does not imply non-preemptiveness at the OS scheduler level: a higher-priority thread can still preempt the flow controller thread.*

Rule 7. `HIGH_PRIORITY (HP)` ***policy (async-mode)***. *Under the* `HP` *policy, messages are assigned priorities based on the corresponding topic. The flow-controller thread processes messages in descending order of priority, ensuring that higher priority messages are handled first. This prioritization mechanism is crucial for scenarios where certain data requires expedited delivery due to its critical nature.*

Rule 8. `FIFO (F)` ***policy (async-mode)***. *The* `FIFO` *policy dictates that messages are handled in the order they arrive, maintaining a strict first-in-first-out processing order.*

Rule 9. `ROUND-ROBIN (RR)` ***policy (async-mode)***. *The* `ROUND-ROBIN (RR)` *policy introduces a balanced approach where the flow-controller thread handles one message per topic in a round-robin fashion, following the order specified by the priority value. Messages related to the same topic are processed in a FIFO manner.*

Rule 10. ***Work-conservation***. *Both flow-controller and listener threads operate under a work-conservation principle. They remain active and continue processing messages as long as there are messages to be served.*

It is important to note that Rule 1 is mutually exclusive with Rules 2 and 3, which pertain to synchronous and asynchronous modes, respectively, since a publisher can operate in only one sending mode at a time. Additionally, Rules 6–9 apply only in the context of the asynchronous sending mode.

The model and the above behavioral rules were derived with a deep inspection of the FastDDS documentation and source code (*GitHub* repository [42]). To corroborate our findings with empirical evidence, we performed several experiments to figure out the interactions between threads by focusing on shared data structures and condition variables. To this end, an application composed of three publishers and one subscriber exchanging data over three topics was executed on two machines running Ubuntu 20.04 and interconnected through a point-to-point Ethernet link.

Furthermore, we designed ad-hoc experiments to corroborate the behavior of the three scheduling policies of the flow-controller. In each experiment, the subscriber is subscribed to three topics (θ_1, θ_2, θ_3) on which each publisher publishes three messages over one of the topics. Each message payload contains the timestamp of the moment when it is sent on the network. Publishers and the subscriber run on different machines, exchanging data by means of a UDP-based network.

4.3. Model validation

`FIFO` **policy.** First, the flow-controller was configured to work with the `FIFO` policy. To corroborate Rule 8, we checked that, at the subscriber (listener) side, messages were received from the oldest timestamp to the most recent one. Fig. 7 (A) shows the result of this experiment: the sequence of messages sent over the network by the flow-controller are in FIFO order as the order observed by the sender and the receiver corresponds[2]

[2] In principle, it would have been possible to observe out-of-order delivery due to data streams following multiple paths through the network, poorly configured queuing along the path, and the lack of flow control mechanisms in UDP protocol [43]. Our experiment leveraged a point-to-point connection to mitigate the issue.

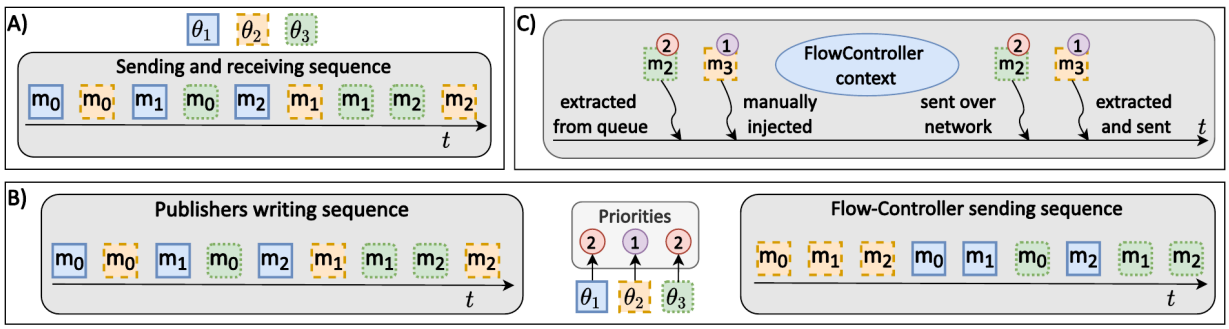

Fig. 7. Validation experiments for R6 (C), R7 (B), and R8 (A) rules.

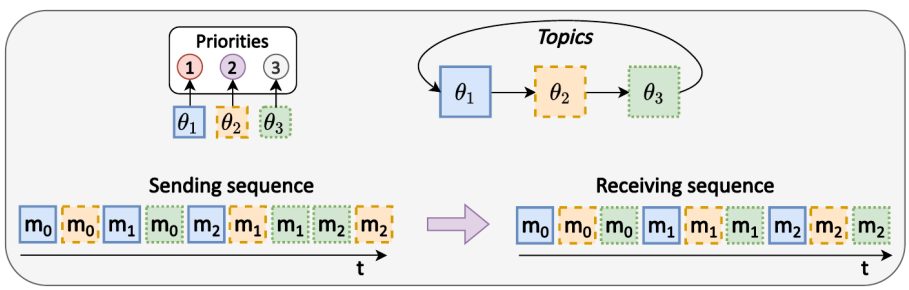

Fig. 8. Validation experiments for R9 rule.

`HIGH_PRIORITY` **policy.** A similar experiment was performed to check the behavior of the `HIGH_PRIORITY` policy (Rule 7). In this experiment, each topic is assigned to a priority. Topic θ_2 is assigned to priority 1, which is the highest of this configuration. θ_1 and θ_3 are both assigned to priority 2. Fig. 7 (B) shows the results of this experiment. As expected, messages related to topic θ_2 (with the highest priority) were sent first on the network, while the messages related to the topics with the same priority were handled in FIFO order.

`ROUND-ROBIN` **policy.** `ROUND-ROBIN` policy behavior has been tested through the same experiment used for the `HIGH_PRIORITY` policy. However, topics are assigned to a priority such that lower topic subscript identifiers indicate higher priority values. In this case, the priority values specify the starting sending order. In Fig. 8, we can observe that messages are sent over the network following a circular pattern, starting from the highest-priority topic θ_1. Considering the messages of any of the topics, messages are always sent following a first-in-first-out order, corroborating Rule 9.

Non-preemptiveness. We checked the non-preemptiveness (Rule 6) of the sending operation performed by a flow controller, modifying the previous experiment. Referring to Fig. 7 (B), when the last message related to topic θ_3 has already been extracted from the pending message queue, we manually injected, by modifying the source code, a new higher-priority message (i.e., m_3 related to topic θ_2) in the queue, as shown in Fig. 7 (C). Even if the new message should be processed first according to the priority order, the flow-controller thread waits until the current message was sent, before processing the highest-priority one, thus exhibiting a non-preemptive behavior.

5. Response-time analysis

In this section, we first provide bounds for the DDL of each message, considering the FIFO, fixed priority, and round-robin scheduling policies under both synchronous and asynchronous communications (Section 5.2). In Section 5.3, we outline how to integrate the analysis in Section 5.2 with the executor-based ROS2 analysis presented by Casini et al. in [10]. In Section 5.4, we show how to build upon the DDL analysis to bound the end-to-end latency of a chain of threads. Finally, we address arrival-curve propagation under both synchronous and asynchronous sending modes (Section 5.5) and discuss the applicability of our approach (Section 5.6).

5.1. Message data-delivery latency in a subchain

Fig. 9 illustrates the FastDDS instance of the compositional model, summarizing the message path from a publisher to a subscriber in both synchronous and asynchronous modes, within a subchain. In asynchronous mode, once the publisher thread prepares new data for transmission, the data is queued in a queue of pending messages handled by the flow controller thread, which then sends the messages through the network, according to one of the sending policies (`F`, `HP`, `RR`). Conversely, in synchronous mode, the publisher

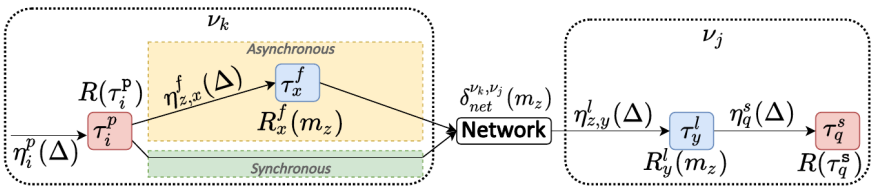

Fig. 9. Subchain data path and arrival curve propagation.

directly sends data over the network, for each topic at a time. When the listener thread receives the message, it processes and delivers it to the user-level subscriber thread.

To bound the DDL of an arbitrary message m_z, we provide worst-case response-time bounds for the publisher and for each message in listener threads for synchronous mode, or within the flow-controller and listener threads for asynchronous mode.

The *worst-case response time of a message* m_z in a middleware thread τ_i^{t}, where $t \in \{f, l\}$, is the maximum time from the release of the message instance in the thread to the completion of its processing. Similarly, in synchronous mode, the worst-case response time of a publisher thread is the longest time span from the release of the publisher thread instance to the completion of all synchronous sending operations, denoted by $R(\tau_i^{\mathrm{p}})$.

We denote the response-time bounds for message m_z in the flow-controller thread τ_x^{f} and listener thread τ_y^{l} by $R_x^{\mathrm{f}}(m_z)$ and $R_y^{\mathrm{l}}(m_z)$, respectively. When the specific middleware-level thread is clear from the context, we simplify the notation to $R(m_z)$. Please, note that all involved threads can be allocated to arbitrary cores. The only restriction is due to DDS-specific allocation constraints, i.e., each publisher (resp. subscriber) thread and the referred flow controller, if any, (resp., listener) threads must be allocated to the same machine. The analysis leverages the arrival curves of messages at the flow-controller and listener threads, as detailed in Section 5.5 for both synchronous and asynchronous cases.

Following CPA [18] and Rules 2–5, the DDL for an asynchronous subchain $L_z(\xi_i^{\mathtt{async}})$ and an arbitrary message m_z is bounded by the sum of the worst-case delays in the flow-controller and listener threads τ_x^{f} and τ_y^{l}, respectively, plus the network delay from ν_k to ν_j, where ν_k and ν_j are the machines in which the publisher and the subscriber are allocated to, respectively:

$$L_z(\xi_i^{\mathtt{async}}) = R_x^{\mathrm{f}}(m_z) + R_y^{\mathrm{l}}(m_z) + \delta_{\mathtt{net}}^{\nu_k,\nu_j}(m_z). \tag{1}$$

Under Rules 1, 4, and 5, the DDL of a synchronous subchain $L_z(\xi_q^{\mathtt{sync}})$ for an arbitrary message m_z is bounded by the sum of the worst-case delays in publisher and listener threads τ_i^{p} and τ_y^{l}, respectively, and the network from ν_k to ν_j:

$$L_z(\xi_q^{\mathtt{sync}}) = R(\tau_i^{\mathrm{p}}) + R_y^{\mathrm{l}}(m_z) + \delta_{\mathtt{net}}^{\nu_k,\nu_j}(m_z). \tag{2}$$

For each thread $\tau_i \in \Gamma_{\mathrm{all}} \setminus \Gamma_{\mathrm{mw}}$, the symbol $rbf_i(\Delta)$ denotes its *request-bound function* (RBF). The RBF bounds the maximum processor time required by τ_i within any interval of length Δ, given by $rbf_i(\Delta) = \eta_i^t(\Delta) \cdot e_i$, with $t \in \{p, s\}$ [10,44]. The sum of the request-bound functions for an arbitrary set of threads Γ' is denoted as $RBF(\Gamma', \Delta) = \sum_{\tau_j \in \Gamma'} rbf_j(\Delta)$.

5.2. Response-time analysis for a Fast-DDS message

We start bounding the DDL of messages: to this end, we first introduce some definitions.

Definitions. To bound the worst-case response time of a message m_z processed by a middleware thread τ_i^{t}, where $t \in \{f, l\}$, or simply τ_i, if the type is not needed or clear from the context, we first report the sources of interference and their corresponding bounds. We begin with *thread-level* interference, influenced by higher-priority non-middleware threads running on the same core.

Definition 3 (Thread-level Interference). The thread-level interference $I_{i,z}^{\mathrm{thread}}(\Delta)$ is an upper bound on the delay suffered by m_z, while being pending in a middleware thread $\tau_i \in \Gamma_{\mathrm{mw}}^k$ during any time interval Δ, due to non-middleware threads $\tau_j \in \Gamma_{\mathrm{all}}^k \setminus \Gamma_{\mathrm{mw}}^k$ allocated on the same core c_k.

Other sources of interference arise from messages: **(i)** messages handled in higher-priority middleware threads on the *same* core (*inter-thread message interference*), and **(ii)** messages handled within the same middleware thread τ_i (*intra-thread message interference*).

Definition 4 (Inter-Thread Message Interference). For $\tau_i \in \Gamma_{\mathrm{mw}}^k$ handling a pending instance of the message m_z, the expression $I_{i,z}^{\mathrm{inter}}(\Delta)$ bounds the delay experienced by m_z during any time interval Δ due to higher-priority middleware threads $\tau_j \in \mathrm{hp}_{\mathrm{mw}}^k(\tau_i)$ on core c_k, processing other messages.

Definition 5 (Intra-Thread Message Interference). The intra-thread message interference $I_{i,z}^{\mathrm{intra}}(\Delta)$ bounds the delay suffered by m_z during any time interval of length Δ due to messages processed within the same middleware thread $\tau_i \in \Gamma_{\mathrm{mw}}^k$ while m_z is pending.

Note that Definition 5 includes interference from both other messages $m_r \neq m_z$ and earlier-issued instances of m_z itself (i.e., *self-interference*).

Policy-independent bounds. We leverage these definitions and derive a general response-time bound for a message in middleware-level threads applicable to both flow-controller and listener threads within a subchain. We first establish bounds for thread-level

and inter-thread message interference, which are invariant under the middleware's scheduling policy. The first Lemma 1 bounds the number of pending message instances in a middleware-level thread.

Lemma 1. *Let $\overline{R}(m_z)$ be a response-time bound for m_z in an arbitrary middle-ware-level thread τ_i. In any interval of length Δ, there are at most $\eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon)$ pending instances of m_z in τ_i.*

Proof. Consider an arbitrary interval $[\bar{t}, \bar{t} + \Delta)$, with $\Delta > 0$. First, note that instances of m_z released at or after $\bar{t} + \Delta$ are not pending in $[\bar{t}, \bar{t} + \Delta)$. Note that $\eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon)$ counts all the instances released in $(\bar{t} - \overline{R}(m_z), \bar{t} + \Delta)$, which has length $\Delta + \overline{R}(m_z) - \epsilon$. By contradiction, assume there are more than $\eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon)$ pending instances in τ_i. Then it means there exists an instance of m_z released at or before $\bar{t} - \overline{R}(m_z)$ that is still pending in $[\bar{t}, \bar{t} + \Delta)$. This leads to a contradiction because $\overline{R}(m_z)$ is a response-time bound for m_z. □

Lemma 1 provides a means to derive a response-time bound for m_z. However, it relies on a pre-existing response-time bound $\overline{R}(m_z)$, thereby introducing a circular dependency. This notation for pre-existing bounds is also employed in subsequent results. To resolve this dependency, standard real-time analysis techniques [18] employ an outer response-time analysis loop. Initially, $\overline{R}(m_z)$ is set to 0, and a response-time estimate is derived iteratively, updating $\overline{R}(m_z)$ until a global fixed-point is achieved. Convergence is guaranteed since response-time estimates never decrease [18]. Further details are provided in Section 5.6.

Next, Lemma 2 bounds the thread-level interference.

Lemma 2. *Let τ_i be a thread handling an instance of message m_z (either as flow-controller or listener thread) running on c_k. In any interval of length Δ, the corresponding thread-level interference is bounded by*

$$I_{i,z}^{thread}(\Delta) \triangleq RBF(\mathrm{hp}_{oth}^{k}(\tau_i), \Delta). \tag{3}$$

Proof. By definition, the thread-level interference involves all non-middleware threads with a higher priority than τ_i. These threads are contained into the set $\mathrm{hp}_{oth}^{k}(\tau_i)$. The lemma follows by noting that $RBF(\mathrm{hp}_{oth}^{k}(\tau_i), \Delta)$ sums all terms $rbf_h(\Delta)$ due to each $\tau_h \in \mathrm{hp}_{oth}^{k}(\tau_i)$, where each term $rbf_h(\Delta)$ bounds the individual demand due to τ_h, in any interval of length $\Delta > 0$. □

Next, we analyze the interference caused by messages. Prior to this, we establish a bound on the delay due to each message processed by a flow-controller or listener thread.

Lemma 3. *The delay due to a single instance of an interfering message m_z in a middleware-level thread $\tau_i^t \in \Gamma_{mw}$ is bounded by*

$$\delta_i^t(m_z) \triangleq \begin{cases} \delta^f(m_z) \cdot N_{sub}(m_z) & \text{if } t = f, \\ \delta^l(m_z) & \text{if } t = l. \end{cases} \tag{4}$$

Proof. Recall that the delay due to an instance of message m_z is equal to $\delta^{\mathrm{f}}(m_z)$ for a flow-controller thread and $\delta^{\mathrm{l}}(m_z)$ for a listener thread. By **Rule 3**, for each instance of a message m_z sent by a publisher, the flow controller sends $N_{\mathrm{sub}}(m_z)$ copies towards subscribers. This leads to a delay of $\delta^{\mathrm{f}}(m_z) \cdot N_{\mathrm{sub}}(m_z)$, proving the first branch of Eq. (4). The second branch follows by noting that, due to **Rule 5**, for each message instance processed by the listener only one message instance at a time is forwarded to the subscriber. □

Building on the previous result, Lemma 4 bounds the inter-thread message interference experienced by the message m_z under analysis.

Lemma 4. *Consider a message m_z in a middleware-level thread $\tau_i^t \in \Gamma_k^{mw}$. Let $\overline{R}_j^t(m_r)$ be a response-time bound for m_r in an arbitrary middleware-level thread $\tau_j^t \in \mathrm{hp}_{mw}^{k}(\tau_i^t)$. In any window of length $\Delta > 0$, the inter-thread message interference of an instance of m_z is bounded by:*

$$I_{i,z}^{inter}(\Delta) \triangleq \sum_{\tau_j^t \in \mathrm{hp}_{mw}^{k}(\tau_i^t)} \sum_{m_r \in \tau_j^t} \eta_{r,j}(\Delta + \overline{R}_j^t(m_r) - \epsilon) \cdot \delta_j^t(m_r), \text{ with } t \in \{f, l\}. \tag{5}$$

Proof. By definition, $I_{i,z}^{\mathrm{inter}}(\Delta)$ includes all the interference due to messages in other middleware level threads $\tau_j^{\mathrm{t}} \in \mathrm{hp}_{\mathrm{mw}}^{k}(\tau_i^{\mathrm{t}})$ on the same core c_k. The first summation sums over all such threads, and the second over all messages handled by each thread. The lemma follows by recalling that, by Lemmas 1 and 3, each of such messages contributes with at most $\eta_{r,j}(\Delta + \overline{R}_j^t(m_r) - \epsilon)$ instances, each one with a delay of at most $\delta_j^t(m_r)$. □

Policy-dependent bounds. We now introduce the bounds on the intra-thread message interference, which vary based on the scheduling message policy applied in the middleware-level thread. Prior to this discussion, we establish the limit on self-interfering instances for a message in Lemma 5.

Lemma 5. *Let $\overline{R}(m_z)$ be a response-time bound for m_z in an arbitrary middle-ware-level thread $\tau_i \in \Gamma_{mw}$. In any interval of length Δ, the number of self-interfering instances to an arbitrary instance of m_z in $\tau_i \in \Gamma_{mw}$, is bounded by*

$$si_i(m_z, \Delta) \triangleq \max(0, \eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon) - 1). \tag{6}$$

Proof. The lemma follows from Lemma 1 by noting that only pending instances of m_z can cause self-interference in the interval $[\bar{t}, \bar{t} + \Delta)$, with $\Delta > 0$, excluding the message instance under analysis, and noting that the number of self-interfering instances cannot be negative. □

The accuracy of the self-interference bound can be enhanced by exploring a broader range of message release times within a busy window [10]. However, this increases the complexity of the analysis and demands more computational time. This possible enhancement will be considered in future work.

`HIGH_PRIORITY` **policy.** Under this policy, each instance of the message m_z being analyzed may experience delays due to: **(i)** low-priority messages, caused by non-preemptive message dispatching (**Rule 7**); **(ii)** equal-priority messages enqueued earlier and thus prioritized by the FIFO tie-breaking; and **(iii)** higher-priority messages. Let $I^{\text{lp}}_{i,z}(\Delta)$, $I^{\text{ep}}_{i,z}(\Delta)$, and $I^{\text{hp}}_{i,z}(\Delta)$ be the interference bounds for (i), (ii), and (iii), respectively, such that $I^{\text{intra}}_{i,z}(\Delta) \triangleq I^{\text{lp}}_{i,z}(\Delta) + I^{\text{ep}}_{i,z}(\Delta) + I^{\text{hp}}_{i,z}(\Delta)$. We begin by bounding $I^{\text{ep}}_{i,z}(\Delta)$ in Lemma 6.

Lemma 6. *All the delays that may contribute to the intra-thread message interference of an instance of m_z that is pending in a middleware-level thread $\tau^t_i \in \Gamma_{mw}$, during any interval of length Δ and due to messages with same priority, are contained into the multiset*[3]

$$\mathcal{D}^{ep}_i(\Delta) = \biguplus_{m_r \in ep_i(m_z)} \{\delta^t_i(m_r)\} \otimes \overline{\eta}_{r,i}(\Delta), \ with\ t \in \{f, l\} \tag{7}$$

where

$$\overline{\eta}_{r,i}(\Delta) \triangleq \begin{cases} si_i(m_z, \Delta) & if\ z = r, \\ \eta_{r,i}(\Delta + \overline{R}^t_i(m_r) - \epsilon) & otherwise, \end{cases} \tag{8}$$

where $\delta^t_i(m_r)$ is given by Lemma 3 and $\overline{R}^t_i(m_r)$ is a response-time bound for m_r in τ^t_i.

Proof. First, note that delays due to intra-thread message interference to an instance of message m_z from messages with the same priority in a middleware-level thread τ^t_i are due to other messages $m_r \in ep_i(m_z)$ in the queue of the same thread. By Lemma 3, each message contributes with a delay of at most $\delta^t_i(m_r)$. By Lemma 5, m_z can contribute with up to $si_i(m_z, \Delta)$ interfering message instances. The lemma follows by noting that other messages can interfere only if they are pending in the same middleware-level thread, with up to $\eta_{r,i}(\Delta + \overline{R}^t_i(m_r) - \epsilon)$ due to Lemma 1. □

Hereafter, the notation $\Sigma(x, M)$ represents the sum of the x *largest* elements in the multiset M. If M contains fewer than x elements, the sum includes all elements in M.

Lemma 7. *Let j be the priority of message m_z. Consider an instance of m_z that is pending in a middleware-level thread $\tau^t_i \in \Gamma_{mw}$ that uses the `HIGH_PRIORITY` policy and an arbitrary time window of length Δ. It holds*

$$I^{ep}_{i,z}(\Delta) \triangleq \Sigma(M^{HP,j}_i - 1, \mathcal{D}^{ep}_i(\Delta)). \tag{9}$$

Proof.

By definition, the jth priority queue of τ^t_i has size $M^{\text{HP},j}_i$. Therefore, at most $M^{\text{HP},j}_i - 1$ message instances with same priority can interfere with the one under analysis. By Lemma 6, the delays of message instances with same priority that may contribute to the intra-thread interference of m_z are contained into the multiset $\mathcal{D}^{\text{ep}}_i(\Delta)$. Hence $\Sigma(M^{\text{HP},j}_i - 1, \mathcal{D}^{\text{ep}}_i(\Delta))$ bounds the intra-thread interference generated by messages with the same priority of m_z. □

Next, we report the bound related to the intra-thread interference due to messages with lower priority, which occurs since each message is handled in a non-preemptive manner [33,45], (**Rule 6**), locally to each thread.

Lemma 8. *Consider an instance of a message m_z in a middleware-level thread using the `HIGH_PRIORITY` policy $\tau^t_i \in \Gamma_{mw}$ and a time window of length Δ. It holds*

$$I^{lp}_{i,z}(\Delta) \triangleq \max_{m_r \in lp_i(m_z)} \delta^t_i(m_r), \ with\ t = f. \tag{10}$$

Proof. Low-priority messages can contribute with at most *one* instance due to the non-preemptive handling of messages (**Rule 7**). The corresponding delay can be at most equal to the longest delay $\delta^t_i(m_r)$, yielding the bound $I^{\text{lp}}_{i,z}(\Delta)$. □

Unlike equal-priority messages (Lemma 7), the bound for higher-priority messages cannot depend on message queue sizes. This is because the message under analysis is queued in a different queue. Therefore, the bound only relies on the message arrival curves of the middleware-level thread being considered. The following Lemma 9 provides a bound on the intra-thread interference caused by higher-priority messages.

Lemma 9. *Consider an instance of a message m_z in a middleware-level thread using the `HIGH_PRIORITY` policy $\tau^t_i \in \Gamma_{mw}$ and a time window of length Δ. Let $\overline{R}^t_i(m_r)$ be a response-time bound for a higher priority message m_r in τ^t_i, it holds*

$$I^{hp}_{i,z}(\Delta) \triangleq \sum_{m_r \in hp_i(m_z)} \eta_{r,i}(\Delta + \overline{R}^t_i(m_r) - \epsilon) \cdot \delta^t_i(m_r), \text{ with } t = f. \tag{11}$$

Proof. Higher-priority messages can interfere with all instances that are pending in an arbitrary interval $[\bar{t}, \bar{t} + \Delta)$. By Lemma 1, the number of such instances is bounded by $\eta_{r,i}(\Delta + \overline{R}^t_i(m_r) - \epsilon)$, each one delaying for up to $\delta^t_i(m_r)$. □

[3] The operator $\uplus$ denotes the union of multisets, e.g., $\{3,3\} \uplus \{6,2\} = \{3,3,6,2\}$, and the product operator $\otimes$ multiplies the number of instances of each element in the multiset, e.g., $\{1,4\} \otimes 2 = \{1,1,4,4\}$.

`FIFO` **policy.** Under the `FIFO` policy, all messages of a middleware-level thread τ_i^{t} are handled in a single queue of size M_i^{F}. Since the tie-breaking policy for messages with equal priority under `HIGH_PRIORITY` is FIFO too, intra-thread message interference $I_{i,z}^{\mathrm{intra}}(\Delta)$ can be bounded as $I_{i,z}^{\mathrm{ep}}(\Delta)$ in Lemma 7, but considering M_i^{F} in place of $M_i^{\mathrm{HP},j}$, and using τ_i^{t} in the union of Lemma 6 instead of $ep_i(m_z)$.

`RR` **policy.** Under the RR policy, instances of a message m_z related to a given topic θ_z in a middleware-level thread τ_i^{t} are handled in a queue of messages $q_{i,z}$ of size $M_i^{\mathrm{RR},z}$; the flow-controller processes one instance of a message per queue, in circular order. The order leveraged by the round-robin cycle is defined by topic priorities: it starts from the topic with highest priority, and proceeds in priority order of topics. Hence, each instance of m_z under analysis can be delayed by: **(i)** instances of other messages m_k related to different topics $\theta_k \neq \theta_z$, causing delay due to round-robin message handling (**Rule 9**), and **(ii)** other, previously released, instances of m_z related to the same topic θ_z, enqueued before the instance under analysis and thus being prioritized by the FIFO tie-break (**Rule 9**). We define

$$I_{i,z}^{\mathrm{rr}}(\Delta) \triangleq \sum_{\theta_k \in \Theta(\tau_i^{\mathrm{t}})} I_{i,z}^{k}(\Delta) \tag{12}$$

as the interference bound for (i) and (ii), so that $I_{i,z}^{\mathrm{intra}}(\Delta) \triangleq I_{i,z}^{\mathrm{rr}}(\Delta)$, where $I_{i,z}^{k}(\Delta)$ represents the interference due to the processing of a message m_k related to the topic θ_k on the message m_z under analysis and it is bounded by the following Lemma 10.

Lemma 10. *Consider an instance of a message m_z related to topic θ_z (with priority π_z) in a middleware-level thread using the `ROUND-ROBIN` policy $\tau_i^t \in \Gamma_{mw}$ and a time window of length Δ. Let m_k representing the message related to topic θ_k (with priority π_k) and $\overline{R}_i^t(m_z)$ be a response-time bound for the message m_z in τ_i^t, then it holds*

$$I_{i,z}^{k}(\Delta) \triangleq min(M_i^{RR,k} - x_k^*, \overline{\eta}_{k,i}(\Delta)) \cdot \delta_i^t(m_k) \tag{13}$$

where

$$\overline{\eta}_{k,i}(\Delta) \triangleq max(0, \eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - x_k^*) \tag{14}$$

with

$$x_k^* \triangleq \begin{cases} 1 & \pi_k \leq \pi_z \\ 0 & otherwise \end{cases} \tag{15}$$

Proof. By definition, the queue $q_{i,k}$ for a topic θ_k has a size of $M_i^{\mathrm{RR},k}$. We can distinguish three cases.

1. **Self-interference due to messages in the queue of the same topic**, i.e., $\theta_k = \theta_z \rightarrow \pi_k = \pi_z$. By Lemma 5 self-interfering message instances can contribute with up to $max(0, \eta_{z,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - 1)$ instances. When considering the message m_z under analysis, the term x_k^* is equal to 1 (Eq. (15)) since $\pi_k = \pi_z$. Hence, since $\pi_k = \pi_z$, then $x_k^* = 1$, and $\overline{\eta}_{k,i}(\Delta) \triangleq \max(0, \eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - x_k^*) = \max(0, \eta_{z,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - 1)$. It follows that $\overline{\eta}_{k,i}(\Delta)$ is valid bound on the number of self-interfering message instances.

   Orthogonally, recalling that the queue $q_{i,z}$ in which the instance of the message under analysis m_z is enqueued has size $M_i^{\mathrm{RR},z}$, at most $M_i^{\mathrm{RR},z} - 1$ instances can self-interfere. Thus, since $\pi_k = \pi_z$, then $x_k^* = 1$, and the term $min(M_i^{\mathrm{RR},z} - 1, \overline{\eta}_{z,i}(\Delta))$ of Eq. (13) bounds the number of self-interference instances for m_z: since both operands are valid bounds, the minimum of the two is a valid bound, too. Recalling that $M_i^{\mathrm{RR},z}$ contains messages of the same size and all characterized by the same $\delta_i^t(m_z)$, it holds that $min(M_i^{\mathrm{RR},z} - 1, \overline{\eta}_{z,i}(\Delta)) \cdot \delta_i^t(m_z)$ bounds the interference due to the processing of other message instances of the same message in the queue $q_{i,z}$.
2. **Interference due to instances of messages m_k of topic θ_k with lower priority than θ_z under analysis**, i.e., $\pi_k < \pi_z$.

   Interfering messages can interfere with all instances that are pending in an arbitrary interval $[\bar{t}, \bar{t} + \Delta)$. By Lemma 1, the number of such instances is bounded by $\eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon)$. However, due to the priority-ordered round-robin handling of the queues, the queue $q_{i,z}$ of the message under analysis is being served first; hence, in the round-robin cycle in which the message under analysis is served, messages from lower priority topics do not interfere because the queue in which the message under analysis is contained is queried first. This is captured by $x_k^* = 1$ for this case (Eq. (15)), and the bound becomes $\eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - 1)$.

   Thus, merging the two above considerations with the fact that the number of interfering instances cannot be negative, similar to the self-interference case, the instances of m_k that may contribute to interference with m_z is bounded by $\overline{\eta}_{k,i}(\Delta) \triangleq \max(0, \eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - x_k^*) = \max(0, \eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - 1)$.

   For the same reason, since the queue $q_{i,k}$ has a size of $M_i^{\mathrm{RR},k}$, m_k can interfere with at most $M_i^{\mathrm{RR},k} - 1$ instances. Therefore, similarly to the previous case, the term $min(M_i^{\mathrm{RR},k} - 1, \overline{\eta}_{k,i}(\Delta)) \cdot \delta_i^t(m_k)$ of Eq. (13) bounds the interference for m_z due to the processing of messages belonging to a lower-priority queue $q_{i,k}$.
3. **Interference due to instances of messages m_k of topic θ_k with higher priority than θ_z under analysis,** i.e., $\pi_k > \pi_z$. The bound follows analogously to the previous case, but, since the priority of $q_{i,k}$ is higher than $q_{i,z}$, all the pending message instances from $q_{i,k}$ can interfere, i.e., $\eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon)$ and $M_i^{\mathrm{RR},k}$, following the two orthogonal reasoning discussed before. Then, $min(M_i^{\mathrm{RR},k}, \max(0, \eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon)))$ is a valid bound on the number of interfering instances in this case. Since $(\pi_k > \pi_z)$, $x_k^* = 0$, also the bound of Eq. (13) is valid, because $min(M_i^{\mathrm{RR},k} - x_k^*, \overline{\eta}_{k,i}(\Delta)) \cdot \delta_i^t(m_k) = \min(M_i^{\mathrm{RR},k} - x_k^*, \max(0, \eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon) - x_k^*)) \cdot \delta_i^t(m_k) = \min(M_i^{\mathrm{RR},k}, \max(0, \eta_{k,i}(\Delta + \overline{R}_i^t(m_z) - \epsilon))) \cdot \delta_i^t(m_k)$.

The lemma follows. □

With Lemmas 7, 8, 9, and 10 in place, we have all the interference components to instantiate a response-time bound under all the policies (i.e., FIFO, HIGH_PRIORITY, and ROUND-ROBIN), which we describe later in Theorem 12.

Response-time bound. We consider a two-step approach to derive the respon-se-time bound. First, a bound for the start time of an arbitrary message instance m'_z is provided. This represents the time when the middleware-level thread begins to non-preemptively process m'_z within the scope of the thread under consideration. However, it is important to note that m'_z may still experience interference from higher-priority threads, both at the thread level and between threads. Next, the response time is computed by leveraging the previously established start-time bound. This is based on the principle that once m'_z begins to be processed, it will not encounter any intra-thread message interference. Theorem 11 provides a bound for the start time of an arbitrary message instance m'_z within a middleware-level thread τ_i.

Theorem 11. *Consider an arbitrary instance m'_z of message m_z running in a middleware-level thread τ_i released at a time A. If S^* is the least positive solution (if any) of the following inequality*

$$sbf_k(S^*) \geq \epsilon + I^{intra}_{i,z}(S^*) + I^{thread}_{i,z}(S^*) + I^{inter}_{i,z}(S^*), \tag{16}$$

then m'_z starts being processed in the middleware-level thread no later than time $A + S^*$.

Proof. By Lemmas 2 and 4, $I^{\text{thread}}_{i,z}(S^*)$ and $I^{\text{inter}}_{i,z}(S^*)$ bound the thread-level and inter-thread message interference, respectively. The intra-thread message interference is bounded by $I^{\text{intra}}_{i,z}(S^*)$ due to Lemmas 7, 8 and 9, if the middleware-level thread adopts the HIGH_PRIORITY policy, Lemma 7 (slightly modified as suggested above) if the FIFO policy is used, or Lemma 10 if ROUND-ROBIN is used. If S^* satisfies Eq. (16), then the service time $sbf_k(S^*)$ supplied by core c_k in any interval of length S^* is enough to satisfy the computational demand of the whole interference to m'_z in the same interval. Therefore, being the middleware-level thread work-conserving (**Rule 10**), m'_z starts being served in the middleware-level thread no later than time $A + S^*$ and the theorem follows. □

Finally, Theorem 12 provides a response-time bound R^*.

Theorem 12. *Consider an arbitrary instance m'_z of message m_z processed by a middleware-level thread τ^t_i released at a time A. If S^* is defined as in Theorem 11 and R^* is the least positive solution (if any) of the following inequality*

$$sbf_k(R^*) \geq \epsilon + I^{intra}_{i,z}(S^*) + I^{thread}_{i,z}(R^*) + I^{inter}_{i,z}(R^*) + \delta^t_i(m_z), \tag{17}$$

then m'_z completes no later than $A + R^*$.

Proof. By Theorem 11, m'_z starts being served no later than time $A + S^*$. After that, due to **Rule 6**, it starts being processed non-preemptively in the middleware-level thread, and it does not suffer intra-thread interference anymore. Hence, $I^{\text{intra}}_{i,z}(S^*)$ bounds the overall intra-thread interference suffered by m'_z in $[A, A + R^*)$. Inter-thread and thread-level interference in the same interval are bounded by $I^{\text{inter}}_{i,z}(R^*)$ and $I^{\text{thread}}_{i,z}(R^*)$, respectively. If R^* satisfies Eq. (17), then the service time $sbf_k(R^*)$ supplied by core c_k in any interval of length R^* is enough to satisfy the computational demand of the whole interference suffered by m'_z, plus the time $\delta^t_i(m_z)$ to process m'_z itself. Hence, the theorem follows. □

The response-time bound for the listener thread, denoted as $R^l_y(m_z)$, is essential for calculating the DDL in both the asynchronous (Eq. (1)) and synchronous (Eq. (2)) scenarios. This bound can be determined using the results presented in this section by applying the FIFO policy. Conversely, the response-time bound for the flow-controller thread, denoted as $R^f_x(m_z)$, is required solely for computing the DDL through Eq. (1) for an asynchronous subchain. The theorems provided in this section can be utilized to find the response-time bound for the flow-controller, considering any of the three scheduling policies: HIGH_PRIORITY, FIFO, or ROUND-ROBIN. Finally, the worst-case response time for the publisher thread, represented as $R(\tau^p_i)$, is necessary for the calculation of the DDL (Eq. (2)) for a synchronous subchain, where e^{sync} is considered as the WCET for the publisher. Specifically, note that the response-time bound for publishers and subscribers can be derived either **i)** with standard methods for response-time analysis under preemptive fixed-priority scheduling [46] for non-ROS2-systems or **ii)** integrating the state-of-the-art executor-based real-time analysis for ROS2-based systems (e.g., [10]) with the results reported in this section. The next Section 5.3 discusses on this integration.

5.3. Analysis integration with executor-based ROS2 analysis

As already introduced in Section 2.3, in ROS2-based systems, publisher and subscriber activities are handled within callbacks. These callbacks are then scheduled by the executor following a ROS2 specific policy.

We next discuss how to integrate our DDS analysis with the ROS2 analysis in [10], which is seamlessly compatible with our analysis. Indeed, likewise our work, the analysis in [10] is based on the CPA framework, through which a complex ROS callback graph is analyzed by computing individual worst-case response-time bounds for each callback of the graph. Specifically, the authors of that paper investigated callback scheduling behavior within a single-threaded executor and used resource reservation to model its resource availability.

5.3.1. Summary of the analysis in [10].

The analysis in [10] builds upon the concepts of quiet time and *busy window* (also called busy period), which are defined as follows.

- A time t is a *quiet time* with respect to an executor and a callback c_i if there is no pending instance of any other callback c_j that can potentially interfere with c_i arrived prior to t.
- A time interval $[t_1, t_2)$ is a *busy window* for an executor and a callback c_i if and only if t_1 and t_2 are quiet times and there is no quiet time in between t_1 and t_2.

Considering a generic callback c_i having a worst-case execution time equal to e_i, the following recurrent equation, from [10],

$$sbf_k(A + R_i^*(A)) = rbf_i(A + 1) + RBF(C_k \setminus c_i, A + R_i^*(A) - e_i + 1), \tag{18}$$

is used to bound the response time $R_i^*(A)$ experienced by the callback $c_i \in C_k$ released at time $A \geq 0$ (relative to the beginning of the current busy window), where:

- C_k is the set of all callbacks allocated to the kth executor.
- The term $sbf_k(\Delta)$ represents the minimum amount of service provided by the executor (which is included in a reservation server in [10]) in an interval of length Δ.
- The term $rbf_i(\Delta) = \eta_i(\Delta) \cdot e_i$ for a callback c_i defines the request-bound function denoting the maximum amount of processor service required by callback instances released in an interval of length Δ.
- Term $rbf_i(A + 1)$ in the equation represents the self-interference from other instances of the same callback, released before the time A in which the instance under analysis is released.
- Finally, the term $RBF(C^*, \Delta) = \sum_{c_j \in C^*} rbf_j(\Delta)$ denotes the total request-bound function of a given set of callbacks in a given interval of length Δ.

Then, the worst-case response time of the callback is bounded by

$$R_i = max\{R_i^*(A) \mid A \geq 0\}. \tag{19}$$

Eq. (19) requires checking an unbounded number of possible release offsets A. To use it to practically compute a response time bound both a bound on the analysis interval and a reduction of the search space size are required. The search space is bounded and discretized in Section 5.3 of [10] by Lemmas 6 and 7, respectively.

Next, we outline two key aspects to be examined for integrating ROS2 timing analysis with DDS analysis: **(i)** how DDS threads scheduling influence the response time of callbacks executed by the ROS2 executor, and **(ii)** how the ROS executor callback processing affects the timing of DDS messages.

5.3.2. Integration with the DDS analysis

DDS message influence on ROS2 callbacks - aspect (i).

We start discussing how to bound the response time of a ROS2 callback when including the DDS-specific aspects. In our analysis, we consider the case where at most one executor thread is assigned to each core c_k in the system and no resource reservation algorithm is used, i.e., the supply provided by the core to the executor is equal to the available time, i.e., $sbf_k(\Delta) = \Delta$. However, since in our case executors are not isolated in a reservation, callbacks processed by an executor may experience additional interference for other higher-priority threads in the system. Specifically, we integrated Eq. (18) to include **i)** a term $I^{\text{inter}}(\Delta)$, representing the Inter-Thread Message Interference (see Definition 4), and **ii)** a term $I^{\text{thread}}(\Delta)$ (see Definition 3). The former term accounts for the impact due to the processing of DDS messages handled by middleware-level threads with higher priority than that of the executor where the callback under analysis is being executed. This interference is bounded by Lemma 4 provided in this paper. Term $I^{\text{thread}}(\Delta)$ still represents the interference due to other higher-priority non-middleware threads. Hence, the response-time equation now becomes:

$$A + R_i^*(A) = rbf_i(A + 1) + RBF(C_k \setminus c_i, A + R_i^*(A) - e_i + 1) + I^{\text{inter}}(A + R_i^*(A)) + I^{\text{thread}}(A + R_i^*(A)). \tag{20}$$

When bounding the analysis interval of possible release times A of each callback, the additional interference due to the aforementioned terms leads to a longer busy window. As a result, the Lemma 6 from [10], which provides an upper-bound on the length of the analysis window, denoted as L^*, must be modified to include both $I^{\text{inter}}(\Delta)$ and $I^{\text{thread}}(\Delta)$ terms. This change increases the length of the busy window and, consequently, the number of release points $A \geq 0$ that need to be checked. However, Lemma 7 from [10] remains unchanged, since it does not depend on the new interference parameters $I^{\text{inter}}(\Delta)$ and $I^{\text{thread}}(\Delta)$. Notably, this lemma is concerned with the activation discontinuity of the callback under analysis and the length of the busy window. In fact, it excludes all the release points A where the request bound function of the callback does not change, i.e., $rbf_i(A + 1) = rbf_i(A)$, with $0 \leq A \leq L^*$.

ROS2 callbacks influence on DDS messages - aspect (ii).

We now discuss a different aspect: how the interference due to the executor thread of ROS2 can delay the DDS messages. Specifically, the term $I_{i,z}^{\text{thread}}(\Delta)$ (bounded by our Lemma 2) must be slightly modified to account for the interference caused by any callback c_j executed within an executor with higher priority than the middleware-level thread τ_i^t processing the message m_z under analysis, in a specific core c_k. If the set $\mathcal{HP}_k^{cbk}(\tau_i^t)$ denotes the set of such callbacks, the term $I_{i,z}^{\text{thread}}(\Delta)$ now becomes as follows:

$$I_{i,z}^{\text{thread}}(\Delta) \triangleq RBF(\text{hp}_{oth}^k(\tau_i^t) \cup \mathcal{HP}_k^{cbk}(\tau_i^t), \Delta).$$

The modified $I_{i,z}^{\text{thread}}(\Delta)$ now also accounts for all the interference due to the processing of all the callbacks in the system, if the executor allocated to the same core has higher priority than the DDS thread in which the DDS message is processed.

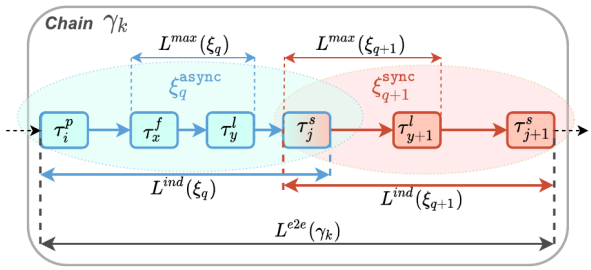

Fig. 10. Latencies within a chain composed of an asynchronous subchain and a synchronous subchain.

With these modifications in place, a ROS2-based system can be analyzed, taking into account the relevant timing effects associated with DDS message processing.

So far, we have shown how to integrate our results with an analysis for a single-threaded ROS2 executor. To the best of records, this is the first attempt in the literature to unify the two approaches. The same can be done for the case of multi-threaded ROS2 executors (e.g., as the one proposed by Jiang et al. in [47]). However, this would require some additional effort that goes beyond the scope of this work. For example, this would require: **i)** extending the higher-priority callback set $\mathcal{HP}_k^{cbk}(\tau_i^l)$, to include the correct interfering callbacks managed by a thread of the multi-threaded executor, and **ii)** incorporating our DDS interference terms $I^{\text{inter}}(\Delta)$ and $I^{\text{thread}}(\Delta)$ into the response-time bound for each callback, similar as we shown for the single-threaded case in the Eq. (20). In the light of these observations, we leave the formal integration of our results in this context as future work.

In the following section, we show how this approach can be extended to perform end-to-end response time analysis for thread chains under DDS communications.

5.4. End-to-end latency analysis

With the compositional approach of CPA [18], extending the subchain response-time analysis (described in Section 5.2) to the analysis of the end-to-end latency of thread chains requires a few steps. First, within a subchain ξ_q, the maximum DDL value among all those of the messages sent by a publisher $\tau_i \in \xi_q$ is computed as:

$$L^{\max}(\xi_q) = \max_{\forall \theta_r \in \Theta(\tau_i), \forall m_z \in \mathcal{M}(\theta_r)} \{L_z(\xi_q)\} \tag{21}$$

Then, starting from the Eq. (21), we derive the individual latency $L^{\text{ind}}(\xi_q)$ for a subchain ξ_q. Although its precise definition depends on whether the subchain ξ_q is asynchronous or synchronous, we can still give a type-agnostic definition. Namely, the individual latency $L^{\text{ind}}(\xi_q)$ of a subchain ξ_q is the longest time span elapsed between the release of the publisher thread τ_i^{p} of the subchain ξ_q to when the subscriber thread τ_j^{s} of the same subchain ξ_q completes.

For the case of an asynchronous subchain ξ_q^{async}, the latency is obtained by summing up the term $L^{\max}(\xi_q^{\text{async}})$ and the WCRT of the publisher τ_i^{p} and subscriber τ_j^{s}, within the subchain.

Similarly, for the synchronous subchain ξ_q^{sync}, the WCRT of the subscriber τ_j^{s} is added to $L^{max}(\xi_q^{\text{sync}})$. Overall, it holds

$$L^{\text{ind}}(\xi_q) = \begin{cases} L^{\max}(\xi_q) + R(\tau_i^{\text{p}}) + R(\tau_j^{\text{s}}) & \texttt{async-mode;} \\ L^{\max}(\xi_q) + R(\tau_j^{\text{s}}) & \texttt{sync-mode.} \end{cases} \tag{22}$$

In turn, the overall end-to-end latency $L^{\text{e2e}}(\gamma_k)$ of the thread chain γ_k can be bounded by summing the individual latencies of each subchain $\xi_q \in \gamma_k$ [18].

However, since two consecutive subchains $\{\xi_q, \xi_{q+1}\}$ are linked by a subscriber $\tau_j^{\text{s}} \in \xi_q$ that also acts as a publisher for the subchain ξ_{q+1}, the WCRT of τ_j^{s} is included twice in the calcution of the individual latency of the two subchains: once for ξ_q, as a subscriber, and once for ξ_{q+1}, as a publisher (please refer to Fig. 10 that should clarify this aspect).

Therefore, to accurately compute the end-to-end latency, the WCRT of each subscriber linking two subchains must be subtracted from the total of the individual latencies of the subchains, i.e.,

$$L^{\text{e2e}}(\gamma_k) = \sum_{\xi_q \in \gamma_k} L^{ind}(\xi_q) \quad - \sum_{\tau_j^{\text{s}} \in \{\Gamma_{\text{p}} \cap \Gamma_{\text{s}} \cap \gamma_k\}} R(\tau_j^{\text{s}}) \tag{23}$$

5.5. Arrival-curve propagation

All the results derived in the previous section are based on the knowledge of the arrival curves of the various threads in the system. Using the standard arrival curve propagation approach of CPA [18], they can be derived from the arrival curves $\eta_i^p(\Delta)$ of source publisher threads τ_i^{p}, response-time bounds, and network propagation delay, if any, in the path from the source to the destination. As discussed in Section 4, a message $m_z(\tau_i^{\text{p}}, \theta_j)$ is identified by its publisher τ_i^{p} and topic θ_j. Furthermore, the per-message arrival curve depends on the number of messages w_i^j published by each instance of the publisher to θ_j. The arrival curve propagation

process is shown in Fig. 9, for both asynchronous and synchronous cases. Under the asynchronous case, the first step is to compute the arrival curve of a message m_z in the flow-controller thread τ_x^{f} starting from the arrival curve of its publisher and the number of message instances sent in each publisher instance to θ_j. This can be computed as:

$$\eta_{z,x}^{\mathrm{f}}(\Delta) = \eta_i^p(\Delta + \overline{R}(\tau_i^{\mathrm{p}}) - \epsilon) \cdot w_i^j, \tag{24}$$

where $\overline{R}(\tau_i^{\mathrm{p}})$ is a response-time bound for the publisher thread, derived as described in Section 5.2. As shown in Fig. 9, the message then passes through the network, with a delay $\delta_{\mathtt{net}}^{\nu_k,\nu_j}(m_z)$, and it is received by the listener thread τ_y^{l}. Depending on the type of the publishing process, the arrival curve of a message within the listener thread is hence computed as:

$$\eta_{z,y}^{\mathrm{l}}(\Delta) = \begin{cases} \eta_{z,x}^{\mathrm{f}}(\Delta + \overline{R}_x^{\mathrm{f}}(m_z) + \delta_{\mathtt{net}}^{\nu_k,\nu_j}(m_z) - \epsilon) & \texttt{async-mode;} \\ \eta_i^p(\Delta + \overline{R}(\tau_i^{\mathrm{p}}) + \delta_{\mathtt{net}}^{\nu_k,\nu_j}(m_z) - \epsilon) & \texttt{sync-mode.} \end{cases} \tag{25}$$

Finally, since a subscriber thread τ_q^{s} may be subscribed to multiple topics, its arrival curve depends from all the activations due to all messages related to these topics. In this case, we distinguish two different semantics according to which a subscriber thread can be activated, similarly to what is referred to as an OR-Activation and an AND-Activation semantics in [18,48]. Intuitively, under the OR-Activation semantic, a subscriber thread is triggered *for each* message published to a subscribed topic. Whereas, under the AND-Activation semantic, a subscriber thread is activated *once* when at least a message for each subscribed topic has been published. Hence, the arrival curve of the subscriber thread can be obtained either considering an OR-activation or AND-activation semantics.

For the OR-activation, the curve is derived by summing all the activations due to all messages $m_z \in \mathcal{M}(\theta_j)$, from the topics $\theta_j \in \Theta(\tau_q^{\mathrm{s}})$ to which the thread τ_q^{s} subscribes to, i.e.,

$$\eta_q^s(\Delta) = \sum_{\theta_j \in \Theta(\tau_q^{\mathrm{s}})} \sum_{m_z \in \mathcal{M}(\theta_j)} \eta_{z,y}^{\mathrm{l}}(\Delta + \overline{R}_y^{\mathrm{l}}(m_z) - \epsilon). \tag{26}$$

Whereas, the derivation of the curve in presence of the AND-activation follows the Lemma 4.2 in [48], equaling the maximum among all the activations due to all messages $m_z \in \mathcal{M}(\theta_j)$, from the topics $\theta_j \in \Theta(\tau_q^{\mathrm{s}})$ to which the thread τ_q^{s} subscribes to, i.e.,

$$\eta_q^s(\Delta) = \max_{\forall \theta_j \in \Theta(\tau_q^{\mathrm{s}}), \forall m_z \in \mathcal{M}(\theta_j)} [\eta_{z,y}^{\mathrm{l}}(\Delta + \overline{R}_y^{\mathrm{l}}(m_z) - \epsilon)]. \tag{27}$$

5.6. Analysis summary and its applicability

We conclude this section by proposing a summary of the analysis and its applicability.

Analysis summary. Algorithm 1 illustrates the method presented in this paper to compute **i)** the DDL latency of messages within subchains for both asynchronous and synchronous scenarios (Eqs. (1) and (2)), and **ii)** the end-to-end latency of thread chains by means of the Eq. (23). The pseudo-code defines global variables to store response-time bounds and their candidates for both application-level threads (or callabcks when targeting ROS2-based systems) and messages handled by middleware-level threads (line 2 and line 5). To populate these variables, the `COMPUTE_RT_BOUNDS()` function is invoked (line 15). Subsequently, the bounds for each published message m_z within the middleware-level threads are computed leveraging the functions `RESPONSETIMEBOUND_FC()` (line 25) and `RESPONSETIMEBOUND_LIST()` line 38). These functions appropriately instantiate $I_{i,z}^{\mathrm{intra}}(\Delta)$ as detailed in Section 5.2, while $I_{i,z}^{\mathrm{inter}}(\Delta)$ and $I_{i,z}^{\mathrm{thread}}(\Delta)$ are defined according to Lemmas 2 and 4, respectively. As explained in Section 5.2, the computation of the response-time bounds $R_x^{\mathrm{f}}(m_z)$, $R_y^{\mathrm{l}}(m_z)$, $R(\tau_i^{\mathrm{p}})$, $R(\tau_j^{\mathrm{s}})$ cyclically depends on pre-existing bounds $\overline{R}_x^{\mathrm{f}}(m_z)$, $\overline{R}_y^{\mathrm{l}}(m_z)$, $\overline{R}(\tau_i^{\mathrm{p}})$, $\overline{R}(\tau_j^{\mathrm{s}})$. This dependency is resolved by initializing the bounds to zero (lines 9 and 11) and iterating through an outer loop (lines 12–23) until a global fixed-point is achieved, where $R_x^{\mathrm{f}}(m_z) \neq \overline{R}_x^{\mathrm{f}}(m_z)$, $R_y^{\mathrm{l}}(m_z) \neq \overline{R}_y^{\mathrm{l}}(m_z)$ for each message m_z, and $R(\tau_i^{\mathrm{p}}) \neq \overline{R}(\tau_i^{\mathrm{p}})$, $R(\tau_j^{\mathrm{s}}) \neq \overline{R}(\tau_j^{\mathrm{s}})$). At each iteration, the arrival curve propagation process (see Section 5.5) is performed (line 22). Upon completion of `COMPUTE_RT_BOUNDS()`, the global variables are set to the correct response-time bound values. At this stage, the function `COMPUTELATENCIES()`, shown at line 44 of Algorithm 1, is used to compute the subchain DDL of a given message m_z according to Eqs. (1) or (2), respectively, for asynchronous and synchronous cases (lines 50 and 52). This computation also considers the message network delay $\delta_{\mathtt{net}}^{\nu_a,\nu_b}(m_z)$ between ν_a (executing τ_x^{f}, for `async case`, or τ_i^{p}, for `sync case`) and ν_b (executing τ_y^{l}). Additionally, this function computes the end-to-end latency of all thread chains in the system (line 55), as specified in Section 5.4.

Applicability. The analysis strategy we proposed makes our method applicable to several practically useful scenarios.

- **Linux – SCHED_FIFO.** Our method is naturally suited for analyzing the timing behavior of FastDDS-based applications running under the `SCHED_FIFO` scheduling class on Linux, which employs a fixed-priority scheduler that aligns with our model's assumptions. In this scenario, each core fully supplies the scheduled applications, as no reservation-based mechanism is implemented. Consequently, the supply bound function for a core c_k is defined as $sbf_k(\Delta) = \Delta, \forall k$.
- **Linux – SCHED_DEADLINE and QNX APS.** Utilizing the supply-bound function abstraction, our analysis can also be applied to systems using the `SCHED_DEADLINE` scheduler on Linux, which is a reservation-based scheduler. Under `SCHED_DEADLINE`, each thread can be temporally isolated by associating it with a budget and period pair [39,41,49]. The definition for $sbf_k(\Delta)$ is available in the literature [37,44,50]. Due to temporal isolation, $I_{i,z}^{\mathrm{thread}}(\Delta) = 0$ and $I_{i,z}^{\mathrm{inter}}(\Delta) = 0$.

Algorithm 1 Pseudo-code of the DDL and end-to-end analysis.

global variables:
  $R(\tau_i^{\mathrm{p}}),\ \overline{R}(\tau_i^{\mathrm{p}}),\ R(\tau_j^{\mathrm{s}}),\ \overline{R}(\tau_j^{\mathrm{s}})$ ▷ to store response-time bounds and the corresponding
  ▷ candidates for application threads (or callbacks)
  $\forall\ \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$, **define**: ▷ for each message in the system
    $R_x^{\mathrm{f}}(m_z),\ R_y^{\mathrm{l}}(m_z),\ \overline{R}_x^{\mathrm{f}}(m_z),\ \overline{R}_y^{\mathrm{l}}(m_z)$ ▷ to store response-time bounds and
    ▷ the corresponding candidates for messages in middleware threads

function COMPUTE_RT_BOUNDS( )
  $R(\tau_i^{\mathrm{p}}) \leftarrow 0,\ \overline{R}(\tau_i^{\mathrm{p}}) \leftarrow 0,\ R(\tau_j^{\mathrm{s}}) \leftarrow 0,\ \overline{R}(\tau_j^{\mathrm{s}}) \leftarrow 0$
  $\forall\ \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$: ▷ for each message in the system:
    $R_x^{\mathrm{f}}(m_z) \leftarrow 0, R_y^{\mathrm{l}}(m_z) \leftarrow 0, \overline{R}_x^{\mathrm{f}}(m_z) \leftarrow 0, \overline{R}_y^{\mathrm{l}}(m_z) \leftarrow 0$
  do
    $R(\tau_i^{\mathrm{p}}) \leftarrow \overline{R}(\tau_i^{\mathrm{p}}), R(\tau_j^{\mathrm{s}}) \leftarrow \overline{R}(\tau_j^{\mathrm{s}})$
    $R(\tau_i^{\mathrm{p}}), R(\tau_j^{\mathrm{s}}) \leftarrow$ `compute bounds for application threads (or callbacks)`
    ▷ see end of Section 5.2
    for $\theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$: **do** ▷ for each message in the system
      $\overline{R}_x^{\mathrm{f}}(m_z) \leftarrow R_x^{\mathrm{f}}(m_z), \overline{R}_y^{\mathrm{l}}(m_z) \leftarrow R_y^{\mathrm{l}}(m_z)$
      if ASYNCHRONOUS: **then**
        FC_SCHED_POL $\leftarrow$ sched. policy of the flow-controller handling m_z
        $R_x^{\mathrm{f}}(m_z) \leftarrow$ RESPONSETIMEBOUND_FC(m_z, FC_SCHED_POL)
      $R_y^{\mathrm{l}}(m_z) \leftarrow$ RESPONSETIMEBOUND_LIST(m_z)
      perform arrival curve propagation ▷ see Section 5.5
  while no more response-time bounds updates

function RESPONSETIMEBOUND_FC(m_z, FC_SCHED_POL)
  switch FC_SCHED_POL **do**
    case HIGH_PRIORITY:
      Set $I_{x,z}^{\mathrm{intra}}(\Delta) \leftarrow I_{x,z}^{\mathrm{ep}}(\Delta) + I_{x,z}^{\mathrm{lp}}(\Delta) + I_{x,z}^{\mathrm{hp}}(\Delta)$ ▷ where $I_{x,z}^{\mathrm{ep}}(\Delta), I_{x,z}^{\mathrm{lp}}(\Delta), I_{x,z}^{\mathrm{hp}}(\Delta)$
      ▷ are bounded as in Lemmas 7, 8, and 9
    case FIFO:
      Set $I_{x,z}^{\mathrm{intra}}(\Delta) \leftarrow I_{x,z}^{\mathrm{ep}}(\Delta)$ ▷ using Lemma 7, with M_x^{F} in place of $M_x^{\mathrm{HP},j}$, and
      ▷ with τ_x^{f} in the union of Lemma 7 in place of $ep_x(m_z)$
    case ROUND-ROBIN:
      Set $I_{x,z}^{\mathrm{intra}}(\Delta) \leftarrow I_{x,z}^{\mathrm{rr}}(\Delta)$ ▷ where $I_{x,z}^{\mathrm{rr}}(\Delta)$ is bounded as in Lemma 10
  Compute S^*, compute R^* using S^* ▷ Theorem 11 to compute S^*, Theorem 12 to compute R^*
  return R^*

function RESPONSETIMEBOUND_LIST(m_z)
  Set $I_{y,z}^{\mathrm{intra}}(\Delta) = I_{y,z}^{\mathrm{ep}}(\Delta)$ ▷ using Lemma 7 with M_y^{F} in place of $M_y^{\mathrm{HP},j}$, and with τ_y^{l}
  ▷ in the union of Lemma 7 in place of $ep_y(m_z)$
  Compute S^*, compute R^* using S^* ▷ Theorem 11 to compute S^*, Theorem 12 to compute R^*
  return R^*

function COMPUTELATENCIES( ) ▷ to be called after COMPUTE_RT_BOUNDS()
  for γ_{ch} in the system **do** ▷ for each chain in the system
    for $\xi_q \in \gamma_{ch}$ **do** ▷ for each subchain of the chain γ_{ch}
      for $\tau_i^{\mathrm{p}} \in \xi_q, \theta_j \in \Theta(\tau_i^{\mathrm{p}}), m_z \in \mathcal{M}(\theta_j)$: **do** ▷ for each message published by τ_i^{p}
        ▷ of the subchain ξ_q
        if ASYNCHRONOUS: **then**
          $L_z(\xi_q^{\mathrm{async}}) \leftarrow R_x^f(m_z) + \delta_{\mathrm{net}}^{v_a,v_b}(m_z) + R_y^l(m_z)$
        else
          $L_z(\xi_q^{\mathrm{sync}}) \leftarrow R(\tau_i^{\mathrm{p}}) + \delta_{\mathrm{net}}^{v_a,v_b}(m_z) + R_y^l(m_z)$
      Compute $L_{max}(\xi_q)$ ▷ according to Eq. (21)
      Compute $L_{ins}(\xi_q)$ ▷ according to Eq. (22)
    Compute $L_{e2e}(\gamma_{ch})$ ▷ according to Eq. (23)

However, compared with the SCHED_FIFO case, SCHED_DEADLINE poses the additional challenge of finding a proper budget and period parameters for each reservation (and therefore for each thread), which could be arguably difficult to derive for data-driven threads. Moreover, our analysis extends to the QNX APS reservation-based scheduler [51,52] by considering only the threads allocated to the same APS partition as the thread under analysis when deriving $I_{i,z}^{\text{thread}}(\Delta)$ and $I_{i,z}^{\text{inter}}(\Delta)$.

Note on Priority Assignment. Priority assignment for executors, callbacks, and DDS middleware threads is an input of our analysis and remains a responsibility of the system designer. Our analysis integration, described in Section 5.3, captures all cross-domain interference regardless of how these priorities are chosen. Hence, our framework assists designers in validating and assessing their system configurations. Mechanisms for configuring DDS-based systems and setting DDS thread priorities can be found in [53].

Dynamic Discovery Extensions. Although our core model assumes *Static Discovery* (see Section 4), it can be extended in future work to consider dynamic scenarios, for example, by modeling discovery messages generated by every DDS participant as a sporadic publisher with a minimum inter-arrival time equal to the DDS announcement period. The sporadic publisher should then be included in the arrival-curve propagation and in the interference terms of Section 5.2. Despite that, dynamic systems involve new entities joining and leaving the system at run-time, mainly following non-deterministic patterns. Therefore, modeling the worst-case for arrival and leaving events of such entities could further complicate the analysis. This is an interesting direction for future work.

6. Evaluation

This section discusses the experimental results obtained with our response-time analysis, which was implemented by building upon the *pyCPA* framework [54]. We propose two sets of experiments to evaluate our analysis: DDS-applications implemented by Linux threads under SCHED_FIFO and via ROS2.

The first set includes experiments purely based on FastDDS (*v2.14.0* [26]) where the response-times for publishers and subscriber were computed with standard methods for response-time analysis under preemptive fixed-priority scheduling [46] for non-ROS2-systems, considering threads scheduled by SCHED_FIFO in Linux. In these experiments, we used a simple FastDDS real application (Section 6.2) and the WATERS 2019 Challenge benchmark by Bosch (Section 6.3). For the FastDDS application, we compared the *DDL* bounds obtained by the analysis under both synchronous and asyncrhonous cases with the empirical values, measured from the execution on a real platform. For the WATERS 2019 Challenge, we evaluated the analysis bounds to assess the applicabilty of our analysis to a complex testbed.

With the second set of experiments, we compare the analytical bounds on the end-to-end latency with empirical end-to-end values (measured from the execution on a real platform) for two case studies based on ROS2 (*ROS2 Iron Irwini release* [25]). For these experiments, response-times for publishers and subscribers were computed using the executor-based real-time analysis introduced in Section 5.3. The first testbed is an ad-hoc ROS2 application (Section 6.4), while the second is based on the well-known *Autoware Reference System (ARS)* [17] (Section 6.5), which includes a computation graph from Autoware, a popular open-source, ROS 2-based, autonomous driving framework [5].

We used an 8-core *Dell Optiplex 7070* machine running Ubuntu 20.04 (kernel v6.5.0 with fixed cores frequency at $3GHz$) as our targeted platform to run all the experiments to measure empirical values. The Linux kernel of the platform was configured with the isolcpus parameter, which allows defining a set of CPUs to be disregarded by the kernel scheduler. In all the experiments performed by running the DDS on a real system, we isolated these workloads on the set of cores $\{c_0, c_1, c_2, c_3\}$.

Before proceeding with the presentation of the results, we report on the experimental methodology we followed to estimate the analysis parameters.

6.1. Measuring message processing delays

The analysis presented in this work requires to estimate the worst-case delays $\delta^{\text{f}}(m_z)$ and $\delta^{\text{l}}(m_z)$ to process a message m_z within the flow-controller and listener threads, respectively. Moreover, for the case of a publisher τ_j sending data synchronously, the parameter e_j^{sync} depends on the parameter $\delta^{\text{s}}(m_z)$, which is also empirically estimated.

The experiments leverage an application purely based on FastDDS, which includes a publisher, its flow-controller, a subscriber, and its listener. The two application-level threads communicate through a single topic (which is varied to cope with all the message size of the WATERS Challenge, see Section 6.3), using UDP through the loopback interface. When considering asynchronous mode, the flow-controller and listener threads were mapped to two different cores with the highest priority to reduce the interference they can suffer during the experiment. Each message was processed 50000 times by each middleware thread. Middleware threads were configured to collect data about the execution time of each processed message.

When computing the parameter $\delta^{\text{s}}(m_z)$ under synchronous mode, we modified the previous publisher to send the data synchronously. Then, the publisher thread has been mapped to the highest priority within a core. It has been instrumented to gather data about the execution time of each message. Again, each message was processed 50000 times by the publisher.

The experiments in the next sections use the maximum observed values for each message payload size as the parameters $\delta^{\text{f}}(m_z)$, $\delta^{\text{s}}(m_z)$, and $\delta^{\text{l}}(m_z)$. For the sake of clarity and ease of reading, we report the message delays in Table 2, for each payload size within flow-controller (asynchronous case), publisher (synchronous case), and listener.

Table 2
Maximum-observed message delays.

Message size [kB]	Flow-controller [ms]	Listener [ms]	Publisher [ms]
1	0.062	0.224	0.098
3	0.064	0.261	0.172
5	0.099	0.346	0.339
24	0.110	0.459	0.493
500	0.759	1.084	2.046
750	1.067	1.576	2.620
1500	1.433	2.123	3.592

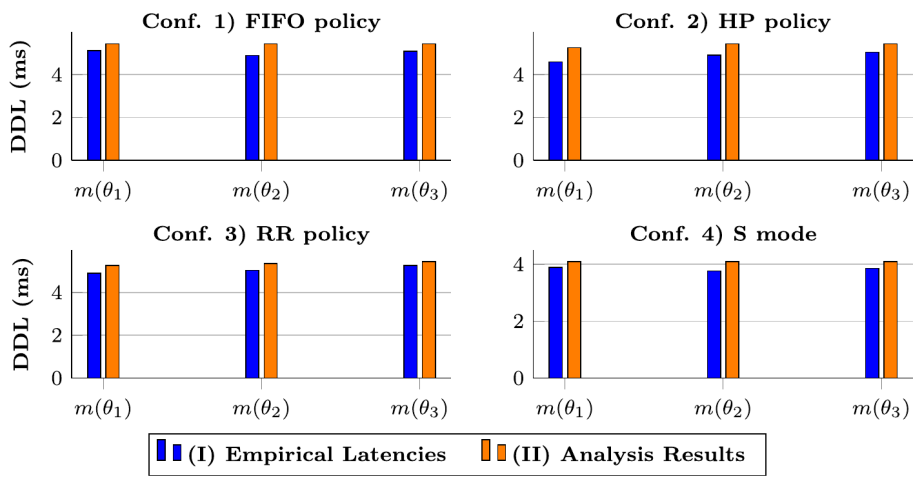

Fig. 11. Results from FastDDS app and analysis related to Configurations 1) - 4).

6.2. Comparing analysis bounds with measured DDLs (pure FastDDS)

In this section, we analyze and compare the bounds on the synchronous and asynchronous DDL policies. For the asynchronous case, we consider the various scheduling policies.

The FastDDS application involves a publisher thread, τ_1^{p}, paired with its flow-controller thread, τ_2^{f}, and a subscriber thread, τ_4^{s}, paired with its listener thread, τ_3^{l}. These components exchange data over three distinct topics: θ_1, θ_2, and θ_3. The messages, denoted as $m(\theta_1)$, $m(\theta_2)$, and $m(\theta_3)$, all have the same payload size of 1kB. The delay for each message is set according to the measurements provided in Table 2 for 1kB payloads.

Threads τ_1^{p} and τ_2^{f} are allocated to the same core, c_0, and are assigned the two highest priorities, with τ_2^{f} having a higher priority than τ_1^{p}. Meanwhile, τ_4^{s} and τ_3^{l} are mapped to cores c_2 and c_1, respectively, each with the highest priority on their respective core. Under asynchronous mode, the publisher thread τ_1^{p} is configured with execution time $e_1 = 1ms$, while, under synchronous mode, the parameter e_1^{sync} is set as $e_1 + \delta^{\mathrm{s}}(m_1) + \delta^{\mathrm{s}}(m_2) + \delta^{\mathrm{s}}(m_3)$ which is equal to $1.294ms$, being $\delta^{\mathrm{s}}(m_1) = \delta^{\mathrm{s}}(m_2) = \delta^{\mathrm{s}}(m_3) = 0.098ms$, as reported in Table 2.

The publisher thread is configured to operate periodically with a $2ms$ period, which is then characterized by an arrival curve $\eta_1^p(\Delta) = \lceil \frac{\Delta}{T} \rceil$. On the Optiplex platform, the FastDDS application is configured to measure the DDL for each message across $50,000$ samples. We evaluated four different configurations:

- **Conf. 1**: The flow-controller schedules messages using the `FIFO` policy.
- **Conf. 2** and **Conf. 3**: The flow-controller schedules messages using the `HIGH_PRIORITY` and `ROUND_ROBIN` policies, respectively. In these cases, each topic is assigned a unique priority, with lower topic subscript identifiers indicating higher priority values.
- **Conf. 4**: The publisher directly manages the message-sending operation, enabling synchronous sending mode.

Fig. 11 presents the DDL (y-axis) for each message (x-axis), comparing the measurements with the analytical bounds across all configurations.

Conf. 1: Under the `FIFO` policy, each flow-controller message has the potential to interfere with others, leading our analysis to compute the same DDL bound of $5446\mu s$ for all messages. When comparing this analytical DDL with the measured values on the Optiplex platform, it is evident that the measured values do not exceed the bound derived from our analysis, validating our analytical findings.

Conf. 2: Under the `HP` policy, both the analytical results and the empirical measurements on the Optiplex platform indicate that the DDL values are dependent on message priority. This observation empirically demonstrates that higher-priority messages benefit

Table 3
Table of percentage difference: measurements vs. analysis bounds.

Message	F	HP	RR	S
$m(\theta_1)$	6.3 %	14.5 %	11.3 %	5.2 %
$m(\theta_2)$	11.1 %	10.7 %	6.1 %	8.7 %
$m(\theta_3)$	6.9 %	7.7 %	7.8 %	6.3 %

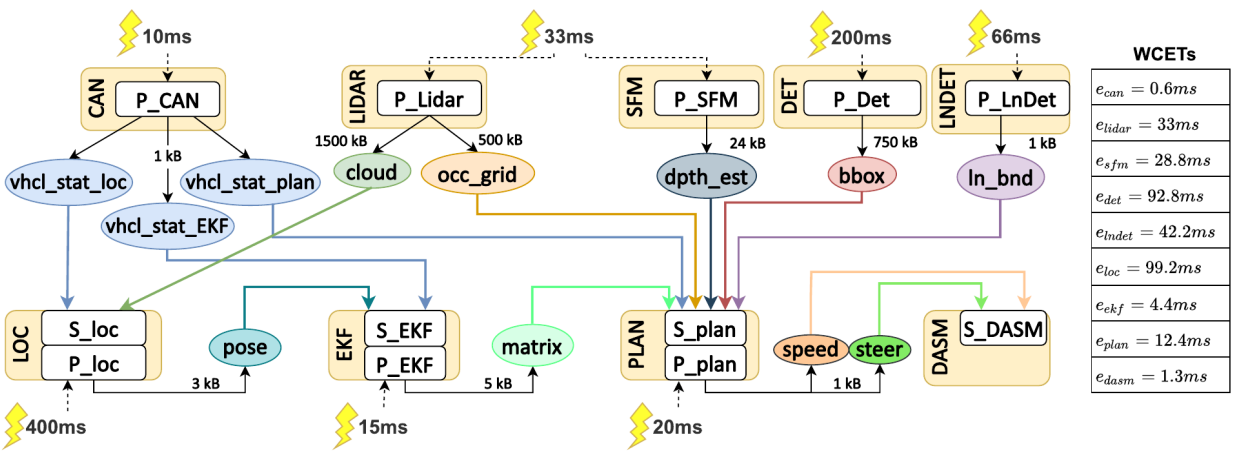

Fig. 12. WATERS 2019 Challenge adapted to use a pub/sub paradigm.

from the HP policy, and our analysis effectively captures this behavior. Additionally, for the lowest-priority message, $m(\theta_3)$, the DDL bound determined by the analysis is identical to the DDL bound computed under **Conf. 1**. Importantly, even in this case, the empirical DDL values do not violate the analytical DDL bounds.

Conf. 3: Under the RR, the DDL bounds obtained from our analysis and the execution on the platform show that the DDL values depend on the order used to query message queues, which is based on the priority of the topic/message. Specifically, messages belonging to higher-priority message queues have benefits with the RR policy, reporting lower empirical DDL values. This behavior is captured by our analysis, as shown in Fig. 11. Notably, the results also show a similar trend when comparing the empirical results with those from **Conf. 2**: specifically, higher-priority messages demonstrate a reduction of the DDLs compared to those under FIFO policy.

Conf. 4: When the synchronous sending mode (`S mode` in Fig. 11) is enabled, we observe that the analytical DDL bounds are smaller than those obtained under all asynchronous policies. This reduction is attributable to the structure of our analysis, which utilizes arrival-curve propagation to derive the arrival curves of non-source threads. This method accounts for delays introduced by predecessor threads in the chain as release jitter, which introduces pessimism in the analysis. Since `S mode` is a flow-controller-free configuration, the analysis' pessimism is mitigated by eliminating the propagation of the worst-case response time caused by the additional processing of the messages managed by the flow-controller along the subchain. This reduction in pessimism results in tighter DDL bounds for synchronous mode, with only a minimal difference between the analytical results and the empirical latencies (see Table 3).

Table 3 summarizes the relative percentage difference between the DDL bounds provided by the analysis and the corresponding measured values. This table demonstrates that the DDL bounds determined by our analysis are indeed tight and reliable for the application under consideration.

6.3. *WATERS 2019 challenge case study (pure FastDDS)*

We present the results from a modern and practical autonomous driving case study based on the WATERS 2019 Challenge. The Challenge includes nine tasks, which are involved in the entire computing process, from the environmental stimuli detected by sensors to the subsequent commands issued to actuators, such as the steering mechanism.

The model for the Challenge is provided in the form of an Amalthea model [55], defining parameters such as periods, worst-case execution times, and data exchanged among threads (i.e., shared labels) for the different threads.

The original model was not explicitly designed for use with the DDS, the target application is nonetheless well-suited for implementation using a publish/subscribe (pub/sub) paradigm. We illustrate in Fig. 12 the adaptation of the Challenge application to DDS.

In this context, shared labels are modeled as topics (depicted as ellipses) with equivalent payload sizes (in kB) to the original labels. Specifically, when a task writes to a label, it is interpreted as *publishing* a message to that topic, while a task reading from a

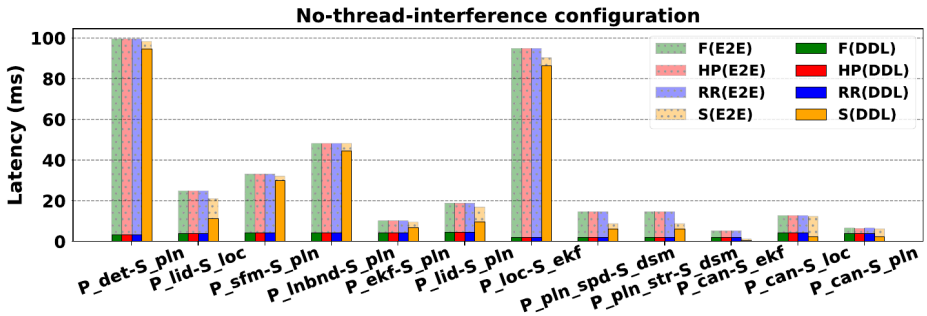

Fig. 13. Experimental results under four representative configurations of the case study.

label is considered as *subscribing* to and receiving messages from the topic. The delays associated with these messages are configured based on their payload sizes, as detailed in Table 2. Threads that only perform read operations on labels are treated as subscribers with data-driven activation triggered by their subscription to corresponding topics (e.g., the DASM task). For threads that engage in both reading and writing operations, such as LOC, EKF and PLAN, we introduced two sub-threads: one representing the subscriber role and the other the publisher role, denoted in the figure with the prefixes *"S_"* and *"P_"*, respectively. This approach preserves the original task periods specified in the Challenge for publishers, as shown in Fig. 12.

Given the original WCET e_i parameters specified in the Challenge model (reported in the table within Fig. 12), we derived individual WCETs for the corresponding publisher and subscriber sub-threads as follows: $e_i^{\text{pub}} = e_i \cdot \alpha$ and $e_i^{\text{sub}} = e_i \cdot (1-\alpha)$, where $\alpha \in [0,1]$. The e_i^{pub} constitutes the asynchronous WCET, while the synchronous WCET is obtained by adding the time required to process a message m_z, i.e., $\delta^{\text{s}}(m_z)$, to e_i^{pub}, for each topic on which the publisher publishes on. Moreover, the parameter α allows for the allocation of computational time between the publish and subscribe components of each task, thereby enabling flexible control over the distribution of execution time between these two activities. We tested a large variety of α parameters, and we report the results for $\alpha = 0.95$.

6.3.1. Analysis results

The priorities of publishers were assigned using the rate-monotonic algorithm [46]. In this case study, each topic is associated with a single publisher, allowing to directly inherit the publisher's priority for the corresponding topic and, by extension, for each message transmitted via that topic. When a publisher handles multiple topics, the topic associated with the smallest payload size is assigned the highest priority, ensuring that smaller, potentially time-sensitive messages are privileged.

For messages generated by the same publisher but with identical payload sizes, a specific priority scheme was applied: for the P_CAN publisher, the priority hierarchy is set as $\pi_{\text{vhcl\_stat\_plan}} > \pi_{\text{vhcl\_stat\_ekf}} > \pi_{\text{vhcl\_stat\_loc}}$, while for the P_plan publisher, the order is $\pi_{\text{steer}} > \pi_{\text{speed}}$. Subscribers such as S_loc, S_EKF, and S_plan inherit the priorities of their corresponding publishers, while the S_DASM subscriber, which provides the final application output, is assigned the highest system priority. Considering that all threads operate on the same computing node v_j, we set $\delta_{\text{net}}^{v_j,v_j}(m_z) = 0$, for each message m_z. Furthermore, for simplicity, we assumed a fixed queue size of 500 for all parameters M_k^{F}, $M_j^{\text{HP},k}$, and $M_j^{\text{RR},k}$.

We considered a *No-thread-interference Configuration*, where each publisher is equipped with its own flow controller when operating in asynchronous mode. Conversely, no flow-controllers are used in synchronous mode. Additionally, all tasks are exclusively allocated to individual cores, thereby eliminating thread-level interference (i.e., the reason behind the name of the configuration). The primary goal of this configuration is to compare asynchronous DDL values and E2E latencies with those in synchronous mode. The graph of Fig. 13 depicts the DDL values (in darker shades) and E2E latencies (in lighter shades) on the y-axis for each subchain in the system, with the subchains (i.e., paths from publishers to subscribers) represented along the x-axis.

In asynchronous mode, message interference is restricted to messages within the same flow-controller thread, particularly for publishers sending multiple distinct messages. However, in synchronous mode, such interference is entirely missing. We analyzed four distinct scenarios: three corresponding to the asynchronous HP, FIFO, and RR policies of the flow-controller, and one corresponding to the synchronous mode (S). Our findings indicate that in this configuration, the asynchronous sending policies generally do not significantly impact the DDL of each subchain, with the exception of those originating from the P_CAN publisher. Specifically, for the P_CAN-S_Plan subchain (i.e., the vchl_stat_plan message), the FIFO and RR policies result in slightly higher DDLs compared to HP, since under HP, the vchl_stat_plan message is assigned the highest priority within the flow-controller. Furthermore, the graph shows another important aspect: the synchronous DDLs result to be higher than the asynchronous DDLs, for almost every subchain. This is due to the fact that when considering the synchronous DDL, the DDL includes publisher's worst-case response time (see Eq. (2) in Section 5.1).

However, when focusing on E2E latency for each subchain, synchronous E2E latencies are consistently lower than, or at worst equal to, their asynchronous equivalents. This outcome, as previously demonstrated in Section 6.2, stems from the absence of flow-controller threads in synchronous mode, which reduces the pessimism that is included in the compositional analysis.

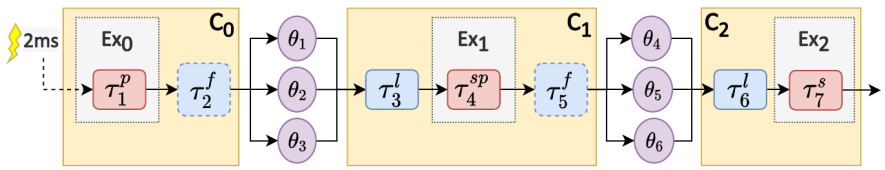

Fig. 14. Overview of the ROS2 testbed application used for E2E latency analysis.

6.4. Comparing analytical bounds with measured E2E latencies (ROS 2)

Next, we focus on evaluating the end-to-end (E2E) latency in a simple, custom, ROS2 testbed. The testbed represents a processing chain of operations. The goal of this section is to compare and analyze the analytical bounds of the E2E latency with the empirical measurements. An overview of the testbed structure is illustrated in Fig. 14. The testbed comprises three ROS2 applications, each of which runs on a dedicated core of the *Optiplex 7070 platform*. These cores are isolated from one another to ensure independent and isolated execution.

The first application has a publisher callback, τ_1^{p}, triggered every $2ms$, and it is assigned to core c_0. The second application, running on core c_1, consists of a callback τ_4^{sp}, which processes the data from the first application and then sends it to the third application. The third application contains a subscriber callback, τ_7^{s}, assigned to core c_2, which provides the system's output. Additionally, each core has an associated ROS2-related executor thread, EX_i, responsible for managing and scheduling the callback on that core. As already introduced at the beginning of Section 6, the worst-case response time of each callback is computed by leveraging the method proposed in [10], integrated into our analysis, as discussed in the Section 5.3.

To receive data, the second and third applications are equipped with listener threads τ_3^{l} and τ_6^{l}, which are allocated to the same cores as their respective callbacks. Depending on the sending mode (synchronous or asynchronous), two flow-controller threads, τ_2^{f} and τ_5^{f}, are allocated to the same core and referenced by the corresponding callback, as shown in Fig. 14. The only chain in the system is activated by the source publisher callback, hence, triggered with its period. All topics (from θ_1 to θ_6) in the testbed have a $1kB$ payload size, and the message processing delays are specified in Table 2. Each topic has a unique priority, with lower subscript values indicating higher priority. Similarly, priorities for executors, callbacks and middleware-level threads are set according to the order they appear in the Fig. 14, i.e., lower subscript values indicate higher priorities.

Under asynchronous mode, the worst-case execution times for the callbacks of the first and second applications, τ_1^{p} and τ_4^{sp}, are $e_1 = 0.1ms$ and $e_4 = 0.2ms$, respectively. For the synchronous mode, the corresponding WCETs are set as follows: $e_1^{sync} = 0.4ms$ and $e_4^{sync} = 0.5ms$.

The application running on the *Optiplex* platform is configured to measure the E2E latency for each chain activation, with $50,000$ samples collected for analysis.

We evaluated all the possible scenarios by varying the sending modes for the publishing callbacks and the policies for the two flow-controllers when the asynchronous sending mode is enabled (`F`, `HP`, `RR`). We tested sixteen different configurations, as reported in the graph of the Fig. 15. On the y-axis, the graph reports the E2E latency for each configuration (x-axis), comparing the empirical measurements with the analytical bounds. Note that, a configuration is represented as a pair "`X-Y`", where `X` refers to the sending mode, and possibly to one of the asynchronous sending policy, for the first publisher callback τ_1^{p}, while `Y` is referred to the second application callback. For example, the configuration "`F-S`" indicates that the first publisher sends data asynchronously with its flow-controller configured with `F` policy, and the second callback sends data synchronously `S` to the third application.

The results indicate that when any of the publishers sends data synchronously (`S`), both the empirical latencies and the analytical bounds are significantly lower than those in asynchronous mode. Specifically, the configuration "`S-S`," where both publishers send data synchronously, resulted in the lowest maximum E2E latency. Moreover, since the analytical calculation of E2E latency considers the maximum DDL among all the messages sent by a publisher (Eq. (21)), configurations of asynchronous policies yielded identical analytical bounds. This is true even for configuration involving the `HP` policy, as the DDL for the lowest-priority message is the same as that of the `F` and `RR` policies. While, regarding the empirical latencies, in general all the combinations with almost one `HP` policy lead to slightly lower values compared to all the other asynchronous policies, e.g., `F-HP` vs `F-F` or `F-RR`, `RR-HP` vs `RR-F` or `RR-RR`, `HP-HP` vs `HP-F` or `HP-RR`, and `S-HP` vs `S-F` or `S-RR`.

Lastly, the percentage difference between the analytical E2E bounds and the empirical values is provided above each bar pair in the graph. The analytical bounds were generally tighter for combinations with at least one synchronous (`S`) mode compared to fully asynchronous combinations. For instance, the maximum relative difference was $10.8\,\%$ for `HP-S`, while the minimum was $6.1\,\%$ for `S-S`. In contrast, `HP-HP` showed the highest relative difference ($16.0\,\%$).

6.5. Autoware reference system case study (ROS 2)

In this last experimental campaign, we evaluate the effectiveness of the proposed end-to-end analysis approach using a case study based on the *Autoware Reference System (ARS)* [17]. ARS is a realistic testbed derived from the open-source *Autoware.Auto* autonomous driving framework developed by the Autoware Foundation [56]. It offers a reference implementation for testing and benchmarking

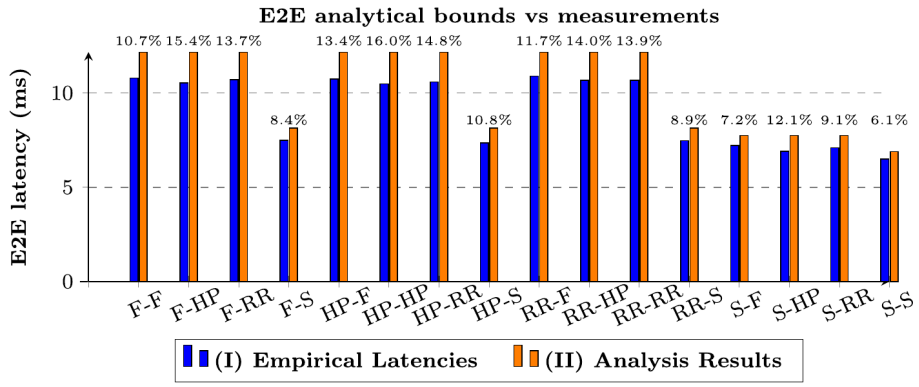

Fig. 15. Comparison between E2E analytical bounds and empirical latencies under different sending modes/policies.

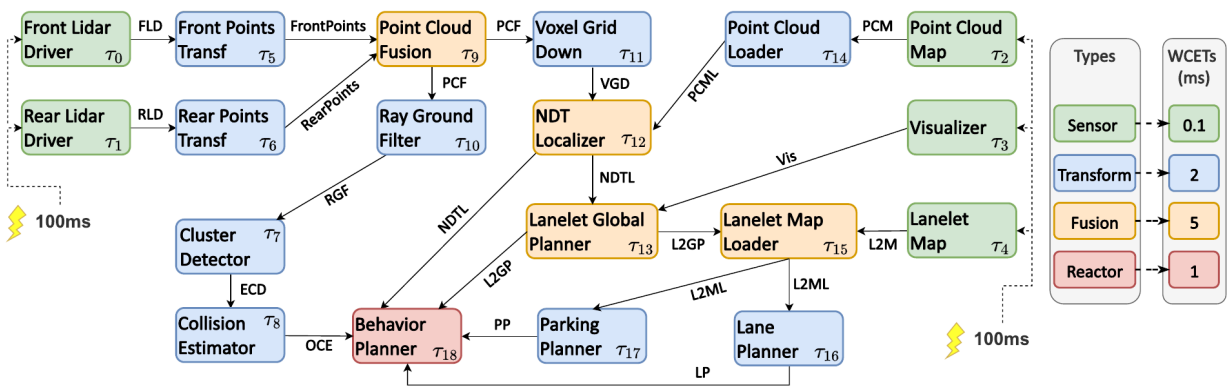

Fig. 16. Overview of the autoware reference system (ARS).

real-time computing performance in ROS 2 applications. For example, it is commonly used to enable the evaluation and performance comparison for various ROS 2 executors [57].

ARS comprises several node types (i.e., *Sensor*, *Transform*, *Fusion* and *Reactor*), each of which is intended to emulate a computational node from the Autoware.Auto project's LiDAR pipeline as shown in Fig. 16. Each node is identified by a name and an alias τ with a subscript, associated with a WCET reported in the figure. Moreover, the subscript represents the callback priority of the node, hence, the lower the value the higher the priority.

For clarity, middleware-level threads between node pairs are not shown in the figure. Incoming edges represent the topics a node subscribes to, while outgoing edges indicate the topics a node publishes. Each edge is labeled with the corresponding topic's name, and the priority of each topic is inherited from the callbacks that publish on it.

The *Sensor* nodes τ_0, τ_1, τ_2, τ_3, and τ_4 are triggered periodically with a frequency of $100ms$, serving as the source publishers for each chain in the system. *Transform* nodes have a single subscriber that listens to one topic and publishes to another, initiating processing immediately after receiving a message.

In contrast, *Fusion* nodes subscribe to two different topics and process data only after receiving messages from both (i.e., AND-Activation). Finally, the *Reactor* node subscribes to five topics but starts executing upon receiving a message from any one of these subscriptions (i.e., OR-Activation).

All publishers in this setup send messages of the same size ($3kB$). Therefore, message delay parameters are configured according to the values provided in Table 2. The system is configured to run on a single machine v_j with four processing cores. Since all nodes are allocated to the same machine, the network propagation delay for any message m_z, i.e., $\delta_{\text{net}}^{v_j,v_j}(m_z)$, is considered negligible and set to zero. Callbacks from the *Sensor* and *Transform* nodes are executed by a single-threaded executor running on core c_0, while all other callbacks run on an executor assigned to core c_1. Listener threads for the subscriber callbacks are allocated to core c_2. In cases where asynchronous sending mode is enabled, all publishers use a common flow-controller thread, which is mapped to core c_3. Each *Sensor* source publisher node τ_i initiates a set of processing chains $\mathcal{G}_i$, where $i \in \{0,1,2,3,4\}$, and each chain $\gamma \in \mathcal{G}_i$ terminates with node τ_{18}. Hence, set $\mathcal{G}_i$ contains the chains representing all the paths connecting τ_i to τ_{18}. To simplify the plots of Fig. 17, we consider each chain set $\mathcal{G}_i$ being populated with a single chain γ_i, reporting the worst end-to-end latency among all the chains in $\mathcal{G}_i$, as shown in Table 4, where the symbol $\rightarrow$ indicate a publisher-subscriber relationship between two nodes.

The goal of this experiment is to examine this complex, modern system and demonstrate that our E2E analysis approach can effectively and efficiently compute tight bounds for the E2E latency of each chain.

Table 4
Considered chains.

ChainSet	Chain	Nodes
$\mathcal{C}_0$	γ_0	$\tau_0 \rightarrow \tau_5 \rightarrow \tau_9 \rightarrow \tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13} \rightarrow \tau_{15} \rightarrow \tau_{16} \rightarrow \tau_{18}$
$\mathcal{C}_1$	γ_1	$\tau_1 \rightarrow \tau_6 \rightarrow \tau_9 \rightarrow \tau_{11} \rightarrow \tau_{12} \rightarrow \tau_{13} \rightarrow \tau_{15} \rightarrow \tau_{16} \rightarrow \tau_{18}$
$\mathcal{C}_2$	γ_2	$\tau_2 \rightarrow \tau_{14} \rightarrow \tau_{12} \rightarrow \tau_{13} \rightarrow \tau_{15} \rightarrow \tau_{17} \rightarrow \tau_{18}$
$\mathcal{C}_3$	γ_3	$\tau_3 \rightarrow \tau_{13} \rightarrow \tau_{15} \rightarrow \tau_{17} \rightarrow \tau_{18}$
$\mathcal{C}_4$	γ_4	$\tau_4 \rightarrow \tau_{15} \rightarrow \tau_{17} \rightarrow \tau_{18}$

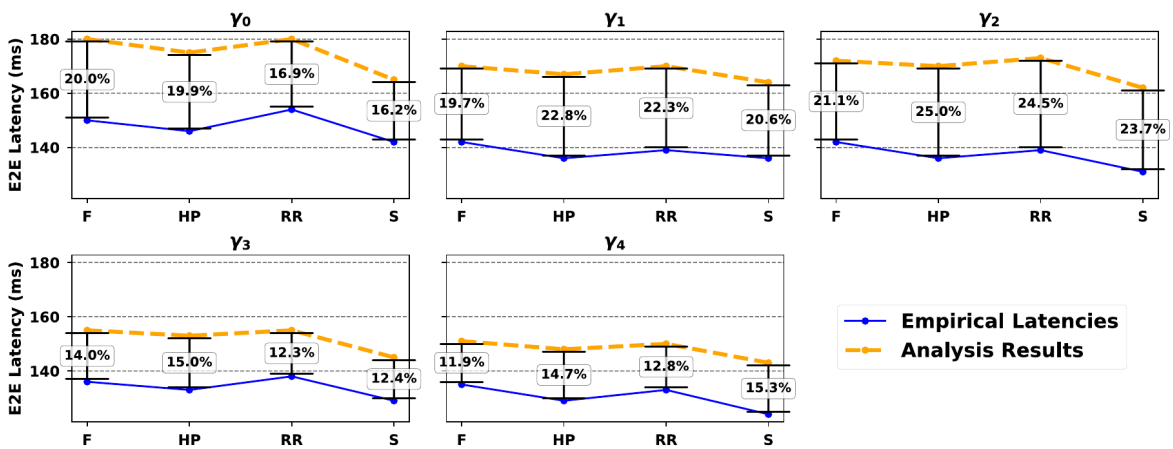

Fig. 17. Autoware Reference System testbed E2E latency results for each considered chains under both synchronous and asynchronous cases.

The Autoware Reference System has been executed on the *Optiplex* platform and modified to measure the E2E latency for all the chains. For each chain activation, we sampled $50,000$ values. Of course, the maximum E2E latency value has also been considered for this experiment.

As in the previous experiment (Section 6.4), we considered four scenarios by varying the sending modes (synchronous 'S' vs asynchronous) for all the publishing callbacks and the policies for the flow-controller when the asynchronous sending mode is enabled (`F`, `HP`, `RR`).

Fig. 17 presents a set of graphs for each chain under investigation. Each graph illustrates the E2E latency on the y-axis and the specific scenario on the x-axis, providing insight into how the empirical latencies and analytical bounds vary depending on the sending mode or asynchronous policy.

Across all chains, the analytical bounds exhibit same trends across the different scenarios. Specifically, under `F` and `RR` scenarios, the same highest E2E latency value has been found within each chain; while under `HP` policy, the latency bounds turned out to be slightly lower than the other asynchronous policies, within each graph. The synchronous mode `S` consistently yields the lowest latency across all chains, making it the most favorable scenario.

This behavior is mirrored in the empirical latency data for most scenarios and chains, with only a few exceptions. For example, in chains γ_0 and γ_3, the `F` policy results in lower latencies compared to `RR`, while the opposite is true for the remaining chains. Regardless, the synchronous mode consistently produces the lowest empirical latencies across all chains.

To further compare the empirical and analytical results, each graph includes a label in the middle of the two lines, indicating the relative percentage difference between the empirical and analytical values. Our analysis tightly bounded the empirical latencies across all scenarios for chains γ_3 and γ_4, with the maximum relative difference reaching approximately 15 % in the `F` and `S` scenarios, respectively. However, γ_2 exhibited the largest relative percentage differences, with values of 21.1 %, 25 %, 24.5 %, and 23.7 % for the `F`, `HP`, `RR`, and `S` scenarios, respectively.

6.5.1. Runtimes required by the analysis on the autoware reference system

To assess the runtime required by our approach and its scalability, we measured the runtimes of the analysis applied to the ARS under varying subsystem configurations, when asynchronous mode is always enabled (which corresponds to the highest number of threads, i.e., considering the total number of system entities, including the additional flow-controller allocated to core c_3, thus requiring higher runtimes). As shown in Fig. 16, the full ARS case includes up to 19 callbacks, 2 single-threaded executors, 1 DDS flow-controller thread, and 14 DDS listener threads, for a maximum of 36 entities (DDS threads and ROS callbacks), and a total of 24 chains. We call this **24-chains configuration**. To evaluate how the runtime vary with a varying number of chains and threads, we derived the other configurations starting from the 24-chains configuration by removing one or more thread, as follows:

Table 5
Analysis runtimes on autoware reference system for varying numbers of chains and entities (DDS threads and ROS callbacks).

# Chains	# Entities	AVG runtime-Python	AVG runtime-Cython
5	26	7.12s	0.89s
6	28	8.46s	0.94s
10	29	8.64s	1.08s
15	32	11.16s	1.24s
21	35	11.68s	1.46s
24	36	14.22s	1.58s

1. **21-chains configuration**: obtained from the 24-chain configuration by removing the τ_4 from the system;
2. **15-chains configuration**: obtained from the 21-chain configuration by removing the τ_1 and τ_6 from the system;
3. **10-chains configuration**: obtained from the 15-chain configuration by removing the τ_2 and τ_{14} from the system;
4. **6-chains configuration**: obtained from the 10-chain configuration by removing the τ_3 from the system;
5. **5-chains configuration**: obtained from the 6-chain configuration by removing the τ_{16} from the system.

We measured the average analysis runtime over 50 runs in both (i) pure Python and (ii) a Cython-compiled version of our analysis. Results are reported in Table 5: the various configurations are indexed by the number of chains (first column).

As shown in Table 5, the pure Python implementation's runtime increases from $7.12s$ at 26 entities to $14.22s$ at 36 entities. This means an average slope of roughly $0.71s$ per additional entity, indicating near-linear scaling. The Cython-compiled solver exhibits an even gentler slope ($\approx 0.07s$ per entity) and achieves an $8-10X$ speed-up across all configurations. Notably, variations in the number of chains (e.g., 6 vs 10 chains at similar entity counts) have only a marginal impact on runtime, confirming that total entity count is the dominant factor.

These results demonstrate that, even for medium-sized ROS2 systems with 30–40 entities, our analysis completes in under $2s$ when compiled and under $15s$ in pure Python, making it practical for both design-space exploration of system configurations and general offline timing validation.

7. Related work

The literature regarding the real-time aspects of DDS is quite limited. To the best of our knowledge, this is the first work to model the DDS from an end-to-end real-time perspective, providing a holistic response-time analysis not only for DDS-based but also for ROS2-enabled systems. Most of the previous research on DDS focused on empirical performance measurement and optimization. For instance, Bellavista et al. [58] compared the DDS implementation OpenSlice (i.e., the predecessor of CycloneDDS by Vortex [30]) with Connext-DDS by Real-Time Innovations [59]. Bode et al. [60] introduced *DDS-Perf*, a benchmarking tool designed to analyze the performance of different DDS implementations.

Krinkin et al. [61] proposed a framework to assess the effectiveness of various DDS implementations in terms of message transport latency and throughput. Specifically, the authors targeted open-source DDS implementations such as OpenDDS [62] (by Prismtech), OpenSlice, and FastDDS [13] (by eProsima). Other works attempted to suggest potential improvements for DDS implementations. For example, Choi et al. [63] studied a real-time DDS setup over specialized packet-switching ASICs to enable Software Defined Networking (SDN). Rosa et al. [64] discussed a DDS model for mission-critical applications and proposed QoS policies to ensure data consistency. Gavrilov et al. [65] presented the performance of different communication protocols used for DDS and showed the benefits of RTPS. Peeck et al. [66] presented a UDP-based protocol for effective error correction with integrity guarantees that considers the DDS as the middleware for data-centric embedded systems. Agarwal et al. [67] proposed the integration of a DDS implementation with a TSN protocol for real-time data transfers. Perez et al. [68] examined the challenges of the integration of the DDS with the *Avionics Full Duplex Switched Ethernet (AFDX)* communication network for safety-critical avionics applications. Stevanato et al. [69] proposed a reference architecture for implementing virtualized DDS communications in a hypervisor-based multi-domain system. Scordino et al. [12] implemented in hardware some DDS functionalities to ensure more deterministic communication times.

Finally, a recent work presented methods to optimize the end-to-end latency of DDS-based systems [53], based on the work in [14], which also provides the foundation from which we extend. However, even compared to [14], as extensively discussed in the introduction, we make significant research advancements. Indeed, none of these works analyzes nor formalizes the synchronous sending mode of data, asynchronous sending under round-robin policy, and the end-to-end latency of DDS-based processing thread chains.

Regarding ROS2 framework, several works analyzed its real-time performance. Casini et al. [10] conducted the first work on formal modeling and analysis of ROS2 executor, laying a foundation for subsequent analysis. Tang et al. [70] improved the response time analysis techniques and proposed priority assignment strategies. Blaß et al. improved [10] by considering the variance of actual execution time and further exploring scheduling features in ROS 2 [71]. Most recently, Jiang et al. [47] conducted the first study on formal modeling and analysis of multi-threaded executors in ROS2. In 2022, Teper et al. [72] addressed the end-to-end timing analysis of ROS 2-based applications executed on a single electronic control unit, leveraging a single-threaded executor. Further studies by Teper et al. [73] analyzed the behavior of ROS2 timers and subscriptions, and Betz et al. [74] studied the effects of the ROS 2 system configuration on end-to-end latencies. Empirical evaluations of ROS 2 over different DDS implementations have been carried out by

Maruyama et al. [75] and Kronauer et al. [76], providing guidelines on designing ROS 2 applications to minimize latencies. Other works considered other middlewares, e.g., OpenMP [77,78], or the ROS-based framework Apex.OS [79].

However, all the previous works on the ROS2 framework did not address the DDS-specific timing properties, often neglecting it from the analysis process. In this context, our work represents the first attempt to propose a holistic approach for ROS2-enabled systems taking also into account the timing peculiarities of the DDS message dispatching, from an analytical point of view.

8. Conclusion and future work

In this paper, we first proposed a general, comprehensive, and compositional model (adaptable to any implementation) based on the DDS specification for studying its timing behavior in real-time distributed systems. Then, we showed how to instantiate the compositional model for the case of the eProsima's FastDDS, one of the most popular DDS implementations, leveraging an extensive exploration of the source code, documentation, and a set of experiments to validate the behavior inferred from the source code.

Building upon the model, we developed the first holistic response-time analysis to upper-bound the data delivery latency for messages and the end-to-end latency of thread chains in purely DDS-based systems, applicable to both synchronous and asynchronous message dispatching. Furthermore, we demonstrated the integration of our analysis with a real-time analysis method for ROS2-based systems [10]. This integration enables ROS2 system designers with a comprehensive end-to-end analysis that accounts for DDS-related timing features, thus bridging a critical gap in the existing literature.

To evaluate the analytical bounds, we conducted experiments comparing our analysis results with measured latency values from a simple FastDDS-enabled application running on a real platform. The close alignment between our analytical predictions and the empirical data corroborated the accuracy of our approach. Additionally, we evaluated our methods using two well-known realistic automotive testbeds: the WATERS 2019 Industrial Challenge by Bosch [15], adapted to be a DDS-based system, and the Autoware reference system [17] from the Autoware autonomous driving framework, which is natively based on ROS2 and the DDS. These evaluations further demonstrated the applicability of our end-to-end latency analysis methods in practical, industry-relevant scenarios.

We believe that the proposed analysis will significantly benefit system designers in configuring both purely DDS-based and ROS2-enabled systems. It provides guidance on critical design decisions such as thread-to-core allocation and priority assignments from a timing-constraints-driven perspective. Our analysis is readily integrable with minimum effort into existing analysis-driven allocation algorithms for DDS [53] and ROS2 [72] systems, facilitating adoption in current development workflows.

Looking forward, future research directions include enhancing the precision of our analysis by combining it with other techniques [80], and holistically considering additional factors such as scheduling effects due to DDS and OS overheads [81], I/O and memory-related contention delays [82–85], and network delays [86–89], with special emphasis on Time-Sensitive Networking [90]. These extensions aim to further refine the analysis and broaden its applicability to a wider range of distributed real-time systems.

9. Acknowledgements

This work has been partially supported by the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456, and the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

CRediT authorship contribution statement

Gerlando Sciangula: Data curation, Conceptualization, Visualization, Software, Methodology, Investigation, Formal analysis; **Daniel Casini:** Supervision, Writing – review & editing; **Alessandro Biondi:** Supervision, Writing – review & editing; **Claudio Scordino:** Conceptualization; **Marco Di Natale:** Conceptualization.

Data availability

The authors do not have permission to share data.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] G. Pardo-Castellote, OMG data distribution service: architectural overview, in: IEEE Military Communications Conference, 2003. MILCOM 2003., 1, 2003, pp. 242–247 Vol.1.

[2] L. Belluardo, A. Stevanato, D. Casini, G. Cicero, A. Biondi, G. Buttazzo, A multi-domain software architecture for safe and secure autonomous driving, in: 2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2021, pp. 73–82.

[3] Z. Dong, W. Shi, G. Tong, K. Yang, Collaborative autonomous driving: vision and challenges, in: 2020 International Conference on Connected and Autonomous Driving (MetroCAD), IEEE, 2020, pp. 17–26.

[4] A. Hamann, S. Saidi, D. Ginthoer, C. Wietfeld, D. Ziegenbein, Building end-to-end IoT applications with QoS guarantees, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6.

[5] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, T. Azumi, Autoware on board: enabling autonomous vehicles with embedded systems, in: 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), 2018, pp. 287–296.
[6] C. Sethukkarasi, K. Palamutha, P. Poonguzhali, S. Sugumaran, Data distribution platform for smart city applications, 2021. https://doi.org/10.1109/IoTaIS53735.2021.9628758
[7] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, M. Gidlund, Industrial internet of things: challenges, opportunities, and directions, IEEE Trans. Ind. Inf. 14 (11) (2018) 4724–4734.
[8] D. Balouek-Thomert, A.R.Z. Eduard Gibert Renart, M.P. Anthony Simonet, Towards a computing continuum: enabling edge-to-cloud integration for data-driven workflows, Int. J. High Perform. Comput. Appl. 33 (6) (2019) 1159–1174.
[9] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, B.B. Brandenburg, Automatic latency management for ROS 2: benefits, challenges, and open problems, in: Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2021.
[10] D. Casini, T. Blaß, I. Lütkebohle, B.B. Brandenburg, Response-time analysis of ROS 2 processing chains under reservation-based scheduling, in: 31st ECRTS 2019, 2019.
[11] R.D. Cuebas, S. Park, Y. Cho, D. Park, C.-G. Lee, Extension of functionally and temporally correct simulation of cyber-systems of automotive systems based on ROS system, Korean Information Science Society Academic Papers (2019) 1174–1176.
[12] C. Scordino, A.G. Mariño, F. Fons, Hardware acceleration of data distribution service (DDS) for automotive communication and computing, IEEE Access 10 (2022) 109626–109651.
[13] eProsima, Fast-DDS, 2024, https://fast-dds.docs.eprosima.com.
[14] G. Sciangula, D. Casini, A. Biondi, C. Scordino, M. Di Natale, Bounding the data-delivery latency of DDS messages in real-time applications, in: 35th ECRTS 2023, (LIPIcs), 2023.
[15] A. Hamann, D. Dasari, F. Wurst, I. Sañudo, N. Capodieci, P. Burgio, M. Bertogna, WATERS Industrial Challenge 2019, 2019a.
[16] A. Hamann, D. Dasari, F. Wurst, I. Sañudo, N. Capodieci, P. Burgio, M. Bertogna, WATERS Industrial Challenge 2019, 2019b. https://archives.ecrts.org/fileadmin/WebsitesArchiv/ecrts2019/waters/waters-industrial-challenge/index.html.
[17] Reference System, https://github.com/ros-realtime/reference-system.
[18] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, R. Ernst, System level performance analysis - the SymTA/S approach, IEEE Proceedings - Computers and Digital Techniques (2005).
[19] OMG, Supported QoS, 2015, https://www.omg.org/spec/DDS/1.4/PDF.
[20] OMG, The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol Specification (v2.5), 2021, https://www.omg.org/spec/DDSI-RTPS/2.5/PDF.
[21] V. Bode, C. Trinitis, M. Schulz, D. Buettner, T. Preclik, DDS implementations as real-time middleware – A systematic evaluation, in: 2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2023, pp. 186–195. https://doi.org/10.1109/RTCSA58653.2023.00030
[22] . Robot Operating System, Default DDS Implementation in ROS2, 2024, https://docs.ros.org/en/iron/Installation/DDS-Implementations.html.
[23] E. Foundation, Zenoh, https://zenoh.io/.
[24] A. Corsaro, L. Cominardi, O. Hecart, G. Baldoni, J.E.P. Avital, J. Loudet, C. Guimares, M. Ilyin, D. Bannov, Zenoh: unifying communication, storage and computation from the cloud to the microcontroller, in: 2023 26th Euromicro Conference on Digital System Design (DSD), 2023, pp. 422–428. https://doi.org/10.1109/DSD60849.2023.00065
[25] Robot Operating System, ROS2 Iron documentation, 2024, https://docs.ros.org/en/iron/index.html.
[26] eProsima, Fast-DDS v2.14.0, 2024, https://fast-dds.docs.eprosima.com/en/v2.14.0/index.html.
[27] T.L. Project, LTTng, https://lttng.org/docs/v2.13/.
[28] E. Foundation, Eclipse Trace Compass, https://eclipse.dev/tracecompass/.
[29] J. Schlatow, R. Ernst, Response-time analysis for task chains with complex precedence and blocking relations, ACM Trans. Embed. Comput. Syst. (2017).
[30] EclipseFoundation, Cyclone-DDS, 2021, https://github.com/eclipse-cyclonedds/cyclonedds.
[31] M. Becker, D. Dasari, S. Mubeen, M. Behnam, T. Nolte, End-to-end timing analysis of cause-effect chains in automotive embedded systems, J. Syst. Archit. 80 (2017) 104–113.
[32] T. Kloda, A. Bertout, Y. Sorel, Latency analysis for data chains of real-time periodic tasks, in: 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), 2018. https://doi.org/10.1109/ETFA.2018.8502498
[33] R.I. Davis, A. Burns, R.J. Bril, J.J. Lukkien, Controller area network (CAN) schedulability analysis: refuted, revisited and revised, Real-Time Syst. 35 (2007) 239–272.
[34] B. Kim, K. Park, Probabilistic delay model of dynamic message frame in flexray protocol, IEEE Trans. Consum. Electron. 55 (1) (2009) 77–82.
[35] L. Thomas, A. Mifdaoui, J.-Y.L. Boudec, Worst-case delay bounds in time-sensitive networks with packet replication and elimination, IEEE/ACM Trans. Netw. (2022) 1–15.
[36] D. Casini, L. Abeni, A. Biondi, T. Cucinotta, G. Buttazzo, Constant bandwidth servers with constrained deadlines, in: Proceedings of the 25th International Conference on Real-Time Networks and Systems, 2017, pp. 68–77.
[37] G. Lipari, E. Bini, Resource partitioning among real-time applications, in: 15th Euromicro Conference on Real-Time Systems, 2003. Proceedings., 2003, pp. 151–158.
[38] I. Shin, I. Lee, Periodic resource model for compositional real-time guarantees, in: RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003, 2003, pp. 2–13.
[39] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279), 1998.
[40] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the linux kernel, Softw. Pract. Exper. 46 (6) (2016) 821–839.
[41] D. Dasari, M. Becker, D. Casini, T. 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), 2022, pp. 214–227.
[42] eProsima, Fast-DDS Github repository, 2023, https://github.com/eProsima/Fast-DDS.
[43] X. Zhou, P. Van Mieghem, Reordering of IP packets in internet, in: C. Barakat, I. Pratt (Eds.), Passive and Active Network Measurement, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 237–246.
[44] G.C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Springer Publishing Company, Incorporated, 3rd edition, 2011.
[45] M. Nasri, B.B. Brandenburg, An exact and sustainable analysis of non-preemptive scheduling, in: 2017 IEEE Real-Time Systems Symposium (RTSS), IEEE, 2017, pp. 12–23.
[46] J. Lehoczky, L. Sha, Y. Ding, The rate monotonic scheduling algorithm: exact characterization and average case behavior, in: [1989] Proceedings. Real-Time Systems Symposium, 1989, pp. 166–171.
[47] X. Jiang, D. Ji, N. Guan, R. Li, Y. Tang, Y. Wang, Real-time scheduling and analysis of processing chains on multi-threaded executor in ROS 2, in: 2022 IEEE Real-Time Systems Symposium (RTSS), 2022, pp. 27–39. https://doi.org/10.1109/RTSS55097.2022.00013
[48] M. Jersak, Compositional Performance Analysis for Complex Embedded Applications, Ph.D. thesis, Technical University of Braunschweig, 2004.
[49] A. Biondi, A. Melani, M. Bertogna, Hard constant bandwidth server: comprehensive formulation and critical scenarios, in: Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014), 2014, pp. 29–37.
[50] A. Biondi, G.C. Buttazzo, M. Bertogna, Schedulability analysis of hierarchical real-time systems under shared resources, IEEE Trans. Comput. 65 (5) (2016) 1593–1605.
[51] M. Becker, D. Casini, The material framework: modeling and automatic code generation of edge real-time applications under the QNX RTOS, J. Syst. Archit. 154 (2024) 103219.
[52] M. Becker, D. Dasari, D. Casini, On the QNX IPC: assessing predictability for local and distributed real-time systems, in: 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2023.

[53] G. Sciangula, D. Casini, A. Biondi, C. Scordino, End-to-end latency optimization of thread chains under the DDS publish/subscribe middleware, in: 2024 Design, Automation, Test in Europe Conference and Exhibition (DATE), 2024, pp. 1–6.
[54] J. Diemer, P. Axer, R. Ernst, Compositional performance analysis in python with pyCPA, in: In Proceedings of WATERS'12, 2012.
[55] E. Foundation, APP4MC, https://eclipse.dev/app4mc/.
[56] A. Foundation, Autoware, https://autoware.org/.
[57] ApexAI, Autoware Reference System, 2024, ttps://github.com/ApexAI/reference-system-autoware.
[58] P. Bellavista, A. Corradi, L. Foschini, A. Pernafini, Data distribution service (DDS): a performance comparison of opensplice and RTI implementations, in: 2013 IEEE Symposium on Computers and Communications (ISCC), IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 000377–000383.
[59] RTI, Connext-DDS, 2013, https://www.rti.com/products/dds-standard.
[60] V. Bode, D. Buettner, T. Preclik, C. Trinitis, M. Schulz, Systematic analysis of DDS implementations, in: Proceedings of the 24th International Middleware Conference, Middleware '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 234–246. https://doi.org/10.1145/3590140.3629118
[61] K. Krinkin, A. Filatov, A. Filatov, O. Kurishev, A. Lyanguzov, Data distribution services performance evaluation framework, in: 2018 22nd Conference of Open Innovations Association (FRUCT), 2018, pp. 94–100.
[62] Prismtech, Open-DDS, 2021, https://download.objectcomputing.com/OpenDDS/OpenDDS-latest.pdf.
[63] H.-Y. Choi, A.L. King, I. Lee, Making DDS really real-time with OpenFlow, in: 2016 International Conference on Embedded Software (EMSOFT), 2016.
[64] L. Rosa, W. Song, L. Foschini, A. Corradi, K. Birman, DerechoDDS: strongly consistent data distribution for mission-critical applications, in: MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM), 2021, pp. 684–689. https://doi.org/10.1109/MILCOM52596.2021.9653032
[65] A. Gavrilov, M. Bergaliyev, S. Tinyakov, K. Krinkin, P. Popov, Using IoT protocols in real-time systems: protocol analysis and evaluation of data transmission characteristics, J. Comput. Netw. Commun. 2022 (2022) 1–18. https://doi.org/10.1155/2022/7368691
[66] J. Peeck, M. Möstl, T. Ishigooka, R. Ernst, A middleware protocol for time-critical wireless communication of large data samples, in: 2021 IEEE Real-Time Systems Symposium (RTSS), 2021, pp. 1–13.
[67] T. Agarwal, P. Niknejad, M. Barzegaran, L. Vanfretti, Multi-level time-sensitive networking (TSN) using the data distribution services (DDS) for synchronized three-Phase measurement data transfer, IEEE Access PP (2019) 131407–131417.
[68] H. Pérez, J.J. Gutiérrez, On the integration of DDS and AFDX standards, in: 2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2024, pp. 73–84. https://doi.org/10.1109/RTCSA62462.2024.00020
[69] A. Stevanato, A. Biondi, A. Biasci, B. Morelli, Virtualized DDS communication for multi-domain systems: architecture and performance evaluation of design alternatives, in: Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), San Antonio, USA, May 9–12, 2023, 2023.
[70] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, W. Yi, Response time analysis and priority assignment of processing chains on ROS2 executors, in: 2020 IEEE Real-Time Systems Symposium (RTSS), 2020, pp. 231–243.
[71] T. Blaß, D. Casini, S. Bozhko, B.B. Brandenburg, A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance, in: 2021 IEEE Real-Time Systems Symposium (RTSS), 2021, pp. 41–53.
[72] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, J. Chen, End-To-end timing analysis in ROS2, in: 2022 IEEE Real-Time Systems Symposium (RTSS), 2022.
[73] H. Teper, T. Betz, G. Von Der Brüggen, K.-H. Chen, J. Betz, J.-J. 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), 2023, pp. 206–215. https://doi.org/10.1109/RTCSA58653.2023.00032
[74] T. Betz, M. Schmeller, H. Teper, J. Betz, How fast is my software? Latency evaluation for a ROS 2 autonomous driving software, in: 2023 IEEE Intelligent Vehicles Symposium (IV), 2023, pp. 1–6. https://doi.org/10.1109/IV55152.2023.10186585
[75] Y. Maruyama, S. Kato, T. Azumi, Exploring the performance of ROS2, in: EMSOFT '16, Association for Computing Machinery, New York, NY, USA, 2016.
[76] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, G.P. Fettweis, Latency analysis of ROS2 multi-node systems, 2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI) (2021) 1–7.
[77] M.A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, E. Quinones, Timing characterization of openMP4 tasking model, in: 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), IEEE, 2015, pp. 157–166.
[78] J. Sun, N. Guan, Z. Guo, Y. Xue, J. He, G. Tan, Calculating worst-case response time bounds for openMP programs with loop structures, in: 2021 IEEE Real-Time Systems Symposium (RTSS), IEEE, 2021, pp. 123–135.
[79] M. Pöhnl, A. Tamisier, T. Blass, A middleware journey from microcontrollers to microprocessors, in: 2022 Design, Automation and Test in Europe Conference and Exhibition (DATE), 2022, pp. 282–286.
[80] J.C. Palencia, M. Gonzalez Harbour, Schedulability analysis for tasks with static and dynamic offsets, in: Proceedings 19th IEEE Real-Time Systems Symposium, 1998.
[81] D.B. de Oliveira, D. Casini, R.S. de Oliveira, T. Cucinotta, Demystifying the real-time linux scheduling latency, in: 32nd Euromicro Conference on Real-Time Systems (ECRTS 2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
[82] A. Serrano-Cases, J.M. Reina, J. Abella, E. Mezzetti, F.J. Cazorla, Leveraging hardware QoS to control contention in the xilinx zynq ultrascale+ MPSoC, in: 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
[83] M. Zini, G. Cicero, D. Casini, A. Biondi, Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms, Softw. Pract. Exper. 52 (5) (2022) 1095–1113.
[84] R. Giannessi, A. Biondi, A. Biasci, RT-Mimalloc: A new look at dynamic memory allocation for real-time systems, in: 2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2024, pp. 173–185. https://doi.org/10.1109/RTAS61025.2024.00022
[85] A. Stevanato, M. Zini, A. Biondi, B. Morelli, A. Biasci, Learning memory-contention timing models with automated platform profiling, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 43 (11) (2024) 3816–3827. https://doi.org/10.1109/TCAD.2024.3449237
[86] C. Blumschein, I. Behnke, L. Thamsen, O. Kao, Differentiating network flows for priority-aware scheduling of incoming packets in real-time IoT systems, in: 2022 IEEE 25th International Symposium on Real-Time Distributed Computing (ISORC), IEEE Computer Society, Los Alamitos, CA, USA, 2022, pp. 1–8.
[87] P.-J. Chaine, M. Boyer, C. Pagetti, F. Wartel, Egress-TT configurations for TSN networks, in: Proceedings of the 30th International Conference on Real-Time Networks and Systems, RTNS 2022, 2022.
[88] R.S. Oliver, G. Fohler, Probabilistic estimation of end-to-end path latency in wireless sensor networks, in: 2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, IEEE, 2009, pp. 423–431.
[89] H. Rashtian, S. Gopalakrishnan, Balancing message criticality and timeliness in IoT networks, IEEE Access 7 (2019) 145738–145745.
[90] G. Patti, L. Lo Bello, L. Leonardi, Deadline-aware online scheduling of TSN flows for automotive applications, IEEE Trans. Ind. Inf. 19 (2022) 5774–5784.