Bounding the Data-Delivery Latency of DDS Messages in Real-Time Applications

Gerlando Sciangula
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Huawei Research Center, Pisa, Italy

Daniel Casini
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Alessandro Biondi
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Claudio Scordino
Huawei Research Center, Pisa, Italy

Marco Di Natale
TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Abstract

Many modern applications need to run on massively interconnected sets of heterogeneous nodes, ranging from IoT devices to edge nodes up to the Cloud. In this scenario, communication is often implemented using the publish-subscribe paradigm. The Data Distribution Service (DDS) is a popular middleware specification adopting such a paradigm. The DDS is becoming a key enabler for massively distributed real-time applications, with popular frameworks such as ROS 2 and AUTOSAR Adaptive building on it. However, no formal modeling and analysis of the timing properties of DDS has been provided to date. This paper fills this gap by providing an abstract model for DDS systems that can be generalized to any implementation compliant with the specification. A concrete instance of the generic DDS model is provided for the case of eProsima's FastDDS, which is eventually used to provide a real-time analysis that bounds the data-delivery latency of DDS messages. Finally, this paper reports on an evaluation based on a representative automotive application from the WATERS 2019 challenge by Bosch.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability

Keywords and phrases DDS, real-time systems, response-time analysis, end-to-end latency, CPA

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.9

Funding This work has been supported by EIT Urban Mobility, an initiative of the European Institute of Innovation and Technology (EIT), a body of the European Union. The work has also been partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU and the European Union's Horizon Europe Framework Programme project NANCY under the grant agreement No. 101096456.

1 Introduction

The Data Distribution Service (DDS) is a standard specification by the Object Management Group (OMG) describing a transfer protocol based on a data-centric publish-subscribe pattern (DCPS) [48]. With the advent of massively distributed applications, such as autonomous driving [9, 26, 31, 34], smart cities, Industry 4.0 [61], and more, the DDS gained a renewed in-

terest in allowing communication among a vastly heterogeneous set of computing devices [42], such as those involved in the so-called IoT-to-Edge-to-Cloud compute continuum [5]. Furthermore, other popular frameworks, such as ROS 2 [12, 17, 21] and Autoware [34], build on top of the DDS to implement the publish-subscribe communication. In the automotive field, the AUTOSAR consortium has recently integrated DDS in its Adaptive Platform software standard [4]. Moreover, DDS support is being integrated in the next release of AUTOSAR Classic Platform [56]. In many of these applications, it is important to provide real-time guarantees on the delivery latency of messages passing through the DDS. However, the DDS is implemented as a complex multi-threaded middleware with threads that must be properly scheduled to achieve the desired real-time performance. These threads serve many purposes, from message dispatching, listening and liveliness monitoring, to garbage collection. Furthermore, some DDS threads implement custom, implementation-specific message queuing policies that can severely affect the message response times.

In this complex scenario, designers of real-time edge applications are called to provide proper values for several critical parameters, such as periods, application and DDS threads priorities, queue sizes, and others. Without fine-grained modeling and analysis of the system, designers can only rely on trial-and-error approaches, deploying system configurations and empirically assessing their performance, which is heavily time-consuming and error-prone.

Contribution. This paper provides a detailed modeling of DDS-enabled real-time systems. First, it provides a general model based on the DDS specification. Then, it shows how to instantiate it for the case of the eProsima's *FastDDS* [27], one of the most popular and efficient [57, 69] DDS implementation, leveraging an extensive exploration of the source code, documentation, and a set of experiments to validate the behavior inferred from the source code. Building on the model, we devise a response-time analysis for messages in a DDS-based distributed real-time system. The analysis can be used as an essential building block for future tools for design-space exploration of the system parameters, which can significantly help designers in configuring complex DDS-based systems. Finally, we evaluate our approach using the *WATERS 2019 Industrial Challenge* by Bosch [30], which represents a complex and real case of autonomous driving application, and we compare the analysis results with the latency values observed by running a simple FastDDS-enabled use-case application on a real platform.

2 Background

In this section, we review the DDS standard. Then, we highlight the peculiarities of FastDDS, i.e., the DDS implementation considered in this work, and we review the Compositional Performance Analysis (CPA) scheme, adopted in the paper.

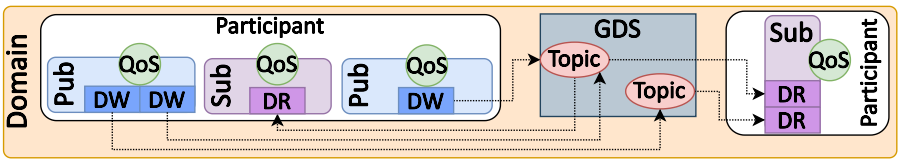

Figure 1 Example of connections between DDS participants in a domain.

2.1 The DDS Standard

The DDS standard specifies a data transfer protocol based on a Data-Centric Publisher-Subscriber (DCPS) pattern [48]. The DCPS model leverages the concept of a *Global Data Space (GDS)*, accessible to all the interested applications. Applications that provide information to the GDS declare their intent to become *Publishers*, whereas applications that want to access portions of the data space are identified as *Subscribers*. The DDS provides mechanisms for the exchange of data between these applications. Whenever a publisher publishes new data into the GDS, the middleware broadcasts this data to all interested subscribers. Moreover, the information flow is regulated by Quality of Service (QoS) policies at various levels of the communication stack [45]. According to the DDS specification, the transfer of any information happens in a logical area called *Domain*, which can be seen as a set of abstract links that connect all the communicating distributed applications. In any domain, there are several *Participants* and *Topics*. Topics are unambiguous identifiers that associate a name, unique within the Domain, to a data type and a set of attached data-specific QoS policies. Topics can be seen as channels for exchanging data. Participants are entities that can send and receive information from any topic in one Domain. A participant can include one or more publishers and/or subscribers. A publisher can send information over multiple different topics through `DataWriter (DW)` objects, and, similarly, a subscriber can receive data from different topics through `DataReader (DR)` objects. Each `DW` or `DR` object is linked to a single topic. Figure 1 shows an example of connections between DDS participants in a domain. The DDS leverages a lower-level protocol, the Real-Time Publish-Subscribe Protocol [46]. RTPS provides both best-effort and reliable publish-subscribe communications over unreliable transports, such as UDP, in both unicast and multicast settings. The OMG has standardized RTPS as the interoperability protocol for all the DDS implementations. Despite its name, RTPS does not define any real-time specific feature. The DDS operates in three main phases: **1)** *Discovery phase*, when the DDS participants find each other in the network, **2)** *Matching phase*, when the discovered participants determine if they should engage in a publish-subscribe relationship, and **3)** *Data Distribution phase*, when data is disseminated from the publishers to the matching subscribers.

2.2 The FastDDS Implementation

FastDDS is a C++ implementation of the DDS with a complex multi-threaded architecture, analyzed by means of code inspection. FastDDS threads are usually scheduled with the `SCHED_OTHER` (i.e., CFS) standard scheduler of Linux.

Publisher application. A publisher application consists of: a *publisher thread*, an *event thread*, and *meta-traffic listener threads*. The *publisher thread* is a user-level thread that manages a single publisher object. It is responsible for preparing and publishing application data on topics. The publishing of data can be **1)** *synchronous*, when it is performed by the publisher thread and **2)** *asynchronous*, when data is sent through the network on behalf of the publisher thread by a *flow-controller thread*, i.e., FastDDS internal middleware-level thread. If the publishing mode is asynchronous, the publisher thread inserts the new message into a queue of pending messages shared with the flow-controller thread. The queue contains messages related to different topics. The queue can be ordered according to three policies, i.e., `FIFO`, `RR` (round robin), `HIGH_PRIORITY` (fixed priority). The flow-controller thread is responsible for extracting data from the queue and sending it over the network. A publisher

thread can refer to multiple flow-controller threads if it publishes to multiple topics. The middleware-level *event thread* processes periodic and time-triggered events (mainly related to discovery/matching and QoS-checking services). The middleware-level *meta-traffic listener threads* manage the reception of discovery information.

Subscriber application. A subscriber application consists of: i) a *subscriber thread*, an *event thread*, a *user-traffic listener thread*, and *meta-traffic listener threads*. The *subscriber thread* is a user-level thread managing a single subscriber object, which is responsible for reading and interpreting data from topics. As for the publisher application, the subscriber application includes the middleware-level *event thread*. The middleware-level *user-traffic listener thread* manages the reception of user data (i.e., application data). The middleware-level *meta-traffic listener threads* manage incoming meta-traffic information.

Communication threads. FastDDS allows multiple publisher threads to publish data over the same topic. Similarly, multiple subscriber threads can subscribe to a specific topic. In this way, *many-to-many* communications between participants are supported. In FastDDS, the transport layer provides communication services between DDS entities, being in charge of sending and receiving messages over a physical transport [27]. Note that a listener thread is spawned for each reception channel, where the definition of channel depends on the adopted transport layer (UDP, TCP, or shared memory transport port).

2.3 Compositional Performance Analysis

CPA [32] is a framework for analyzing the timing behavior of complex heterogeneous and distributed real-time systems. CPA is built around two main concepts: *workloads* and computational *resources*. Workloads consist of tasks with precedence constraints. Applications are modeled as a direct acyclic graph (DAG) of communicating tasks. Groups of tasks execute on a *resource*, which provides the supply time and determines the resource-specific scheduling policy. In CPA, the source task of a chain is triggered according to an *externally provided event arrival curve* $\eta(\Delta)$, denoting an upper bound on the number of release events in any interval $[t, t + \Delta)$. Non-source tasks are triggered by *derived arrival curves*. Derived curves are obtained from arrival curves by accounting for the activation delay given by the completion times of predecessor tasks. The typical approach consists in accounting for a release jitter in non-source tasks that depends on predecessors' response times. The basic CPA analysis uses the sum of individual response-time bounds of each task to bound the end-to-end latency of a processing chain, while extensions have been designed to improve the precision in specific conditions [55].

3 Compositional DDS Model

Next, we model a DDS-based system in two steps. First, we provide a general and compositional model based on the DDS specification only that can be instantiated on any DDS implementation. Then, we show how to instantiate the model for the specific case of FastDDS.

We leverage a compositional approach to model the DDS middleware in an implementation-independent manner by mapping DDS operations to compositional *Logic Functional Blocks (LFBs)*. Each block describes the basic DDS operations. LFBs are divided into two categories: **(i)** *principal* blocks, which are directly involved in the data exchange from the publisher to the subscriber, and **(ii)** *auxiliary* blocks providing support (middleware) features such as discovery/matching, QoS-enforcement, or other implementation-specific services. Examples

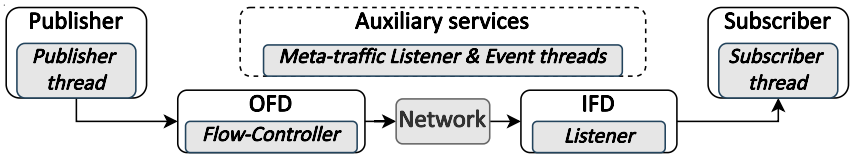

Figure 2 Instantiation of FastDDS threads on DDS model.

of DDS auxiliary implementation-specific services are represented by FastDDS's *Timed-Event handling* and Eclipse CycloneDDS's *Garbage Collector* and *Liveliness monitoring* [68]. *Publisher*, *Subscriber*, *Outgoing Flows Dispatching (OFD)*, *Network*, and *Incoming Flows Dispatching (IFD)* are principal LFBs. The Publisher and Subscriber blocks implement the fundamental publishing and subscribing operations, respectively, performed by user-level application-specific threads. The OFD block receives data from the Publisher block and controls the process of publishing it over the Network block. Note that the DDS standard does not define how data dispatching over the network should be implemented. The IFD block manages the procedure of processing messages received by the Network block. Furthermore, it is responsible to deliver messages to the Subscriber block. Finally, the Network block maps the functionalities of a network protocol and it is in charge of transmitting data over a communication link, from a source node to a destination node.

FastDDS instance of the model. Figure 2 shows the FastDDS implementation-specific instance of the abstract compositional DDS model. Each FastDDS thread we identified is mapped in its corresponding LFB. Meta-traffic listener and event threads have been mapped to the *auxiliary services block*. The publisher and subscriber threads have been mapped respectively to the *Publisher block* and *Subscriber block*. The flow-controller thread, discussed in Section 2.2, has been instantiated upon the *OFD block* when asynchronous-send mode is enabled. Similarly, the user-traffic listener (from now on, we refer to it simply as listener) thread has been mapped to the *IFD block*. Finally, the functionalities of a transport protocol and the network have been mapped onto the *Network block*.

4 FastDDS-based System Model and Problem Definition

The considered FastDDS-based system comprises a set C of cores, where each core $c_k \in C$ is possibly distributed onto multiple nodes in a distributed system.

Thread model. Four classes are used to identify the system threads: *publisher*, *flow-controller*, *listener*, and *subscriber*, contained in the sets Γ_{p}, Γ_{f}, Γ_{l}, and Γ_{s}, respectively. Threads are scheduled using a partitioned fixed-priority scheduler (each thread is statically allocated to a core). An arbitrary i-th thread belonging to each category is denoted 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 set $\Gamma_{\mathrm{all}} = \{\Gamma_{\mathrm{p}} \cup \Gamma_{\mathrm{f}} \cup \Gamma_{\mathrm{s}} \cup \Gamma_{\mathrm{l}}\}$ represents all the threads in the system. When the type of a thread is not relevant or clear from the context, the thread is simply denoted with τ_i. The set of *middleware-level* threads includes flow-controller and listener threads and is denoted with $\Gamma_{\mathrm{mw}} = \Gamma_{\mathrm{f}} \cup \Gamma_{\mathrm{l}}$. The set Γ_{all}^k includes all threads of any type running on core $c_k \in C$. $\Gamma_{\mathrm{mw}}^k \subseteq \Gamma_{\mathrm{all}}^k$ is the subset of the middleware threads on c_k. Each thread $\tau_i \in \Gamma_{\mathrm{all}}$ is associated with a unique fixed priority. Finally, we use the notation $\mathtt{hp}_{oth}^k(\tau_i) \subseteq \Gamma_{\mathrm{all}}^k \setminus \Gamma_{\mathrm{mw}}^k$ and $\mathtt{hp}_{\mathrm{mw}}^k(\tau_i) \subseteq \Gamma_{\mathrm{mw}}^k$ to indicate the set of non-middleware- and middleware-level threads, respectively, that run on core c_k and have priority higher than $\tau_i \in \Gamma_{\mathrm{all}}^k$.

Topic and message model. To define the logical communication channels between publisher and subscriber applications, we define a set of topics Θ. Each topic $\theta_j \in \Theta$ has a unique priority within the whole system, *which is independent of the priorities of the corresponding threads*, discussed in the *Thread model*. The topic priority is then inherited by each instance of any message $m_z(\tau_i^{\mathrm{p}}, \theta_j)$ published by the publisher thread τ_i^{p} over the topic θ_j. We simply use the symbol m_z whenever it is not needed to identify the publisher thread and the topic. An instance of a message m_z is said to be *pending* in a middleware-level thread $\tau_i \in \Gamma_{\mathrm{mw}}$ from when it is released in the thread to when its processing completes in the middleware-level thread. The set of messages associated with a topic θ_j is denoted with $\mathcal{M}(\theta_j)$. Each instance of the publisher thread τ_i^{p} can send up to w_i^j messages to topic θ_j. We define $\Theta(\tau_j^{\mathrm{s}}) \subseteq \Theta$ as the subset of the topics from which a subscriber thread τ_j^{s} can receive messages. $N_{\mathrm{sub}}(m_z)$ denotes the number of subscribers interested in message m_z.

Association among threads. A publisher thread τ_i^{p} can be associated to multiple flow-controller threads $\tau_i^{\mathrm{f}} \in \Gamma_{\mathrm{f}}$ if it publishes to multiple topics. A subscriber thread τ_j^{s} is associated to a unique listener thread τ_j^{l}, which can handle messages from different topics. A pair (publisher thread, topic) $(\tau_i^{\mathrm{p}}, \theta_j)$, and therefore a message $m_z(\tau_i^{\mathrm{p}}, \theta_j)$, is associated with a single flow-controller thread and a single listener thread. The association of a message to a middleware thread is denoted by $m_z \in \tau_i^{\mathrm{t}}$, with $t \in \{\mathrm{f}, \mathrm{l}\}$.

Execution times and activations. Non-middleware-level threads $\tau_j \in \Gamma_{\mathrm{all}} \setminus \Gamma_{\mathrm{mw}}$ are characterized by a worst-case execution time e_j. This paper considers a discrete-time model, i.e., all time parameters are integer multiples of a basic time unit (e.g., a processor cycle), defined as $\epsilon \triangleq 1$. Each publisher thread τ_i^{p} is characterized by an *externally-provided* event arrival curve $\eta_i^p(\Delta)$. Subscriber thread instances are triggered in a data-driven fashion. Therefore, each subscriber thread τ_j^{s} is associated with a *derived* arrival curve $\eta_j^s(\Delta)$, which depends on the response times of the message triggering the computation. We show later in Section 5.2 how to derive such curves. Differently, the worst-case execution time and activation patterns of flow-controller and listener threads are determined by the arrival patterns and message-processing delays of the messages. We denote with $\eta_{z,i}^{\mathrm{f}}(\Delta)$ and $\eta_{z,j}^{\mathrm{l}}(\Delta)$ the derived arrival curve of each message in their flow-controller thread τ_i^{f} and listener thread τ_j^{l}, respectively. Whenever it is not relevant whether τ_i is a flow-controller or a listener thread, we simply denote the arrival curve of a message with $\eta_{z,i}(\Delta)$. The parameters $\delta^{\mathrm{f}}(m_z)$ and $\delta^{\mathrm{l}}(m_z)$ denote the worst-case time required to process a message m_z in its flow-controller and listener threads, respectively, without the interference of any other message and thread. In the flow-controller, this time is required to execute a single system send call, while in the listener involves the deserialization of a single message and delivery of the message to the subscriber object. In both cases, the message size affects the parameter. The network propagation delay of a message m_z is denoted as $\delta^{\mathrm{net}}(m_z)$. It can be either pragmatically estimated or analytically bounded, depending on the underlying network [23, 35, 67].

Flow-controller scheduling policies. We define the available scheduling policies of flow-controller threads as `HP` and `F`, denoting the `HIGH_PRIORITY` and `FIFO` policies as defined by FastDDS, respectively. The analysis of the RR policy is left as future work. Within the same flow-controller, messages that have the same priority (related to the same topic) are processed in FIFO order. When using the `HP` policy, given an arbitrary message m_z and a flow-controller thread τ_i, the symbols $hp_i(m_z)$, $ep_i(m_z)$, and $lp_i(m_z)$ denote the set of all the messages with higher, equal, and lower priority than m_z in τ_i, respectively.

Table 1 Table of main symbols.

Sym.	Description	Sym.	Description
c_j	j-th physical core	τ_i^{t}	i-th thread of type $t \in \{\mathtt{p}, \mathtt{f}, \mathtt{l}, \mathtt{s}\}$
Θ	set of topics	$m_z(\tau_i^{\mathrm{p}}, \theta_j)$	z-th *msg* published by τ_i^{p} over θ_j
θ_j	j-th topic	w_i^j	max. num. *msgs* to θ_j for each τ_i^{p} instance
$\mathcal{M}(\theta_j)$	*msgs* for a topic θ_j	$\Theta(\tau_j^{\mathrm{s}})$	topics from which τ_j^{s} receives *msgs*
Γ_{t}	threads of type t	$N_{\mathrm{sub}}(m_z)$	num. of *subscribers* subscribed to m_z
Γ_{mw}	middleware threads	$\mathtt{hp}^k_{\mathrm{mw}}(\tau_i)$	mw-thrds with pr. higher than τ_i on c_k
Γ_{all}	all threads	$\mathtt{hp}^k_{oth}(\tau_i)$	non-mw-thrds with pr. higher than τ_i on c_k
Γ_{t}^k	threads of type t on c_k	$rbf_i(\Delta)$	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} arr. curve, $t \in \{\mathtt{p}, \mathtt{s}\}$	$\eta_{z,i}^{\mathrm{t}}(\Delta)$	arrival curve of m_z in τ_i^{t}, $t \in \{\mathtt{f}, \mathtt{l}\}$

Static discovery. In this paper, we consider a static network of publishers and subscribers, meaning that no new participant join at run-time. Under this assumption, the overhead due to the discovery mechanism becomes negligible by leveraging the FastDDS *Static Discovery* [27]. This configuration implies that, after static discovery is over, the network of entities is fixed, and no other discovery messages are exchanged among them. Thus, delay due to meta-traffic listener and event threads becomes negligible (auxiliary services block in Fig. 2), since they are responsible for processing discovery periodic events (e.g., sending of heartbeat messages for remote node liveliness) and QoS-checking services.

Listener threads. Each listener thread handles one network socket through which the thread receives data related to different topics, possibly sent by different publishers. Incoming messages are processed in `FIFO` order.

Queues. When using the `FIFO` policy, each flow-controller (or listener) thread $\tau_j^{\mathrm{t}} \in \Gamma_{\mathrm{mw}}$, with $t \in \{f, l\}$, manages one queue of pending messages that can contain at most $M_j^{\mathtt{F}}$ messages. Differently, using the `HP` policy, each priority-level i corresponds to a queue of size $M_j^{\mathtt{HP},i}$. Note that `HP` is only used for flow-controller threads. Buffers should be large enough so that no messages are dropped at both the sender and receiver sides.

Supply-bound function. In this work, analysis and results rely on the existence of a supply-bound function $sbf_k(\Delta)$ that denotes the minimum time of processor service provided by a core $c_k \in C$ in any time window of length Δ, [16, 40, 60]. This abstraction is useful to make the analysis extensible with reservation-based scheduling mechanisms [1], such as those implemented by the `SCHED_DEADLINE` scheduling class of Linux [39] or by the QNX Adaptive Partitioning Scheduler [22], and naturally generalizes to the case without reservation (if a core is fully available, $sbf_k(\Delta) = \Delta$).

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

4.1 Problem Statement

The metric of interest for the analysis in this paper is:

▶ **Definition 1** (Data Delivery Latency). *The Data Delivery Latency (DDL) L_z 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 in which m_z is sent by the publisher and the time instant when the corresponding instance of the subscriber thread is released.*

Note that, as described next in Section 4 (Execution times and activations), for each subscriber thread, *exactly* one subscriber thread instance is triggered for each received message instance. Moreover, the DDL only accounts for the time spent by a message in the middleware-level threads, and does not include the time in which the target instance of the subscriber is waiting for being scheduled. Our goal is to leverage the model to devise a real-time analysis capable of bounding the worst-case *data delivery latency* of any DDS message.

4.2 Thread behavioral rules

Next, we formalize the behavior of the FastDDS middleware implementation through a set of rules, considering the interactions between the modeled threads.
R1 – Pub-to-Flow: When a publisher thread needs to send data over a topic, it performs a `write` operation and notifies the corresponding flow-controller thread.
R2 – Flow-to-Net: If the queue of pending messages is not empty, the flow-controller thread extracts a message from the head of the queue, arranges the RTPS packet (serialization), and performs a system network send operation for each interested subscriber. The number of send operations corresponds to the number of message copies to be sent to each subscriber, expressed by the parameter $N_{\text{sub}}(m_z)$. When the queue is empty, the flow-controller thread blocks until its associated publisher thread notifies it with new data to send.
R3 – Net-to-List: A listener thread performs a blocking system network `receive` operation on a socket. Whenever a message is received and written to a socket buffer by the system network functionalities, the listener thread is woken up and becomes ready to process incoming messages.
R4 – List-to-Sub: The listener thread takes a message from the socket buffer, following a `FIFO` pattern. Then, it deserializes the message. The message is then delivered to the subscriber thread, which is notified of the new message presence.
R5 – Non-preemptiveness: The send operation of a message is *non-preemptive*, meaning that, if a message has been extracted from the queue, it and all of its copies to different subscribers are sent over the network, even if a higher priority message has arrived in the meantime.
R6 – `HIGH_PRIORITY (HP)` policy: Under this policy, each message is assigned to a priority inherited from the corresponding topic. Messages are handled in the flow-controller thread in priority order, from the highest to the lowest.
R7 – `FIFO (F)` policy: Under this policy, the flow-controller thread handles each message in a first-in-first-out fashion.
R8 – Work-conservation: Flow-controller and listener threads never become idle if there are messages to be served.

4.3 Model Validation

The model and the above behavioral rules have been derived with a deep inspection of the FastDDS documentation and source code (*GitHub* repository [28]). To corroborate our findings with empirical evidence, we performed several experiments to figure out interactions between threads by focusing on shared data structures and condition variables. To this end, an application constituted of three publishers and one subscriber exchanging data over three topics was executed on two desktop machines running Ubuntu 20.04 and interconnected through a point-to-point Ethernet link using UDP communication. Furthermore, we designed ad-hoc experiments to corroborate the behavior of the two scheduling policies of the flow controller. In each experiment, the subscriber is subscribed to three topics (θ_1, θ_2, θ_3) on

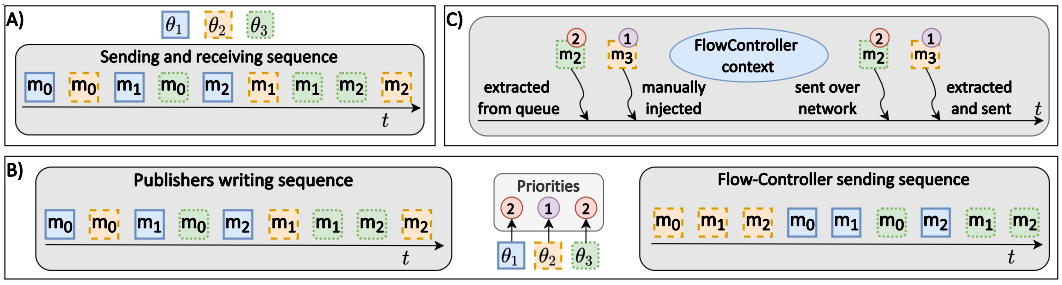

Figure 3 Validation experiments for R5 (C), R6 (B), and R7 (A) rules.

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. Subscribers and publishers run on different machines.

`FIFO` policy. First, the flow-controller was configured to work with the `FIFO` policy. To corroborate rule **R7**, we checked that at the subscriber listener side, messages were received from the oldest timestamp to the most recent one. Figure 3 (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[1].

`HIGH_PRIORITY` policy. A similar experiment was performed to check the behavior of the `HIGH_PRIORITY` policy (rule **R6**). 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. Figure 3 (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.

Non-preemptiveness. We checked the non-preemptiveness (rule **R5**) of the sending operation, modifying the previous experiment. Referring to Figure 3 (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 Figure 3 (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.

Making FastDDS more predictable. We introduced two features to improve FastDDS predictability, considering a Linux-based system, and to show that it is possible to make FastDDS fully compliant with our model. We leveraged these changes in the comparison of analysis-driven and empirical latencies in Section 6.2. First, FastDDS does not provide any mechanism to set scheduling properties for its internal threads, such as thread priorities and usage of the Linux's fixed-priority scheduler to schedule such threads. Therefore, we modified FastDDS to introduce a new scheduling service able to initialize these parameters at the system start time. Second, message queues (e.g., in the flow-controller) are unbounded

[1] In principle, it would have been possible to observe out-of-order delivery due to data streams following multiple paths through the network and the lack of flow control mechanisms in UDP protocol [70]. Our experiment leveraged a point-to-point connection to mitigate the issue.

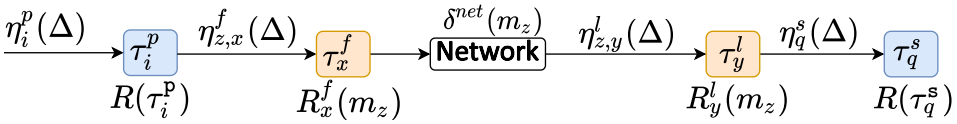

Figure 4 Source-Destination data path and arrival curve propagation.

by default, which may lead to unbounded growth of memory if the system is flooded with a considerably large amount of messages (e.g., due to a distributed denial-of-service attack). Hence, we modified the source code to comply with a limited size. Methods to derive a suitable size for such queues will be derived in future work.

5 Data-Delivery Latency Analysis

Figure 4 leverages the FastDDS instance of the compositional model to summarize the message path from the publisher to the subscriber. Once the publisher thread prepares new data to transmit, such data is inserted in a queue of pending messages managed by the flow controller thread, which sends messages through the network. When the subscriber listener thread receives the message, it is processed and delivered to the user-level subscriber thread. To bound the DDL of an arbitrary message m_z we provide bounds for the worst-case response-time experienced by each message in each middleware thread of interest, namely, the flow-controller and listener threads. The worst-case response time of a message m_z in a middleware (either the flow-controller or the listener) thread τ_i^{t}, with $t \in \{f, l\}$, is the longest time span from the release of the message instance in the thread to when the message instance processing completes. We denote with the symbols $R_x^{\mathrm{f}}(m_z)$ and $R_y^{\mathrm{l}}(m_z)$ a *response-time bound* for message m_z in the associated flow-controller thread τ_x^{f} and listener thread τ_y^{l}, respectively. Whenever specifying the involved thread is not needed or clear from the context, we simply write $R(m_z)$. Note that all the threads involved in the communication can be allocated to arbitrary cores. The following analysis leverages the knowledge of arrival curves of messages at the flow-controller and listener threads: we show later in Section 5.2 how to derive them. Following CPA [32] and by rules **R1-R4**, the DDL L_z of an arbitrary message m_z can be bounded as the sum of the individual worst-case delays experienced in the network and flow-controller and listener threads, i.e.,

$$L_z = R_x^{\mathrm{f}}(m_z) + R_y^{\mathrm{l}}(m_z) + \delta^{\mathrm{net}}(m_z). \tag{1}$$

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 returns the maximum processor time needed by the thread instances of τ_i in any interval of length Δ, i.e., $rbf_i(\Delta) = \eta_i^t(\Delta) \cdot e_i$, with $t \in \{p, s\}$ [15, 17]. The sum of request-bound functions of an arbitrary set of threads Γ' is referred to as $RBF(\Gamma', \Delta) = \sum_{\tau_j \in \Gamma'} rbf_j(\Delta)$.

5.1 Response-Time Analysis for a Fast-DDS message

Definitions. To bound the worst-case response time of a message m_z while being processed by middleware thread τ_i^{t}, with $t \in \{f, l\}$, or simply τ_i if the type is not needed, we start defining the sources of interference that can delay m_z, and the corresponding bounds. We start from the *thread-level* interference, which depends on higher-priority non-middleware threads running on the same core.

▶ **Definition 2** (Thread-level Interference). *The thread-level interference $I_{i,z}^{thread}(\Delta)$ is an upper bound on the delay suffered by an arbitrary instance of m_z while pending in middleware thread $\tau_i \in \Gamma_{mw}^k$, in any time interval of length Δ, due to non-middleware threads $\tau_j \in \Gamma_{all}^k \setminus \Gamma_{mw}^k$ allocated on the same core c_k.*

Other sources of interference are due to messages. This interference can be either due to: **(i)** messages handled in other middleware threads with higher priority running on the same core, or **(ii)** messages handled in the same middleware thread τ_i under analysis. We call (i) *inter-thread message interference*, and (ii) *intra-thread message interference*.

▶ **Definition 3** (Inter-Thread Message Interference). *The inter-thread message interference $I_{i,z}^{inter}(\Delta)$ is an upper bound on the delay suffered by an arbitrary instance of m_z while being pending in middleware thread $\tau_i \in \Gamma_{mw}^k$, in any time interval of length Δ, due to the processing of other messages by high-priority middleware-level threads $\tau_j \in \boldsymbol{hp}_{mw}^k(\tau_i)$ on the same core c_k.*

▶ **Definition 4** (Intra-Thread Message Interference). *The intra-thread message interference $I_{i,z}^{intra}(\Delta)$ is an upper bound on the delay suffered by an arbitrary instance of m_z, in any time interval of length Δ, due to messages processed by the same middleware-level thread $\tau_i \in \Gamma_{mw}^k$ where m_z is pending.*

Note that Definition 4 includes both interference due to instances of other messages $m_r \neq m_z$, and from other instances (previously released) of the same message under analysis. We call the latter *self-interference*, and the corresponding instances *self-interfering instances*.

Policy-independent bounds. Now, we instantiate the previously defined interference bounds, and we finally derive a generic response-time bound for a message in a middleware-level thread, which can be used for both flow-controller and listener threads. We start presenting bounds for the thread-level and inter-thread message interference, which are independent of the scheduling policy adopted in the middleware-level thread. To this end, Lemma 5 bounds the number of pending message instances in a middleware-level thread.

▶ **Lemma 5.** *Let $\overline{R}(m_z)$ be a response-time bound for m_z in an arbitrary middleware-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. ◀

While Lemma 5 is presented as a mean to derive a response-time bound for m_z, it requires, in turn, a pre-existing response-time bound $\overline{R}(m_z)$, hence introducing a circular dependency. The same notation for pre-existing bounds is used also in the presentation of the following results: the dependency can be solved by using standard real-time analysis techniques [32] that provide an outer response-time analysis loop and initially set $\overline{R}(m_z) = 0$, deriving a response-time estimate at every iteration, and updating $\overline{R}(m_z)$ until a global fixed-point is achieved. The procedure is guaranteed to converge since response-time estimates never decrease [32]. Further details are provided next in Section 5.3.

Next, Lemma 6 bounds the thread-level interference.

▶ **Lemma 6.** *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*

$$I_{i,z}^{thread}(\Delta) \triangleq RBF(\boldsymbol{hp}_{oth}^{k}(\tau_i), \Delta). \tag{2}$$

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 $\mathtt{hp}_{oth}^{k}(\tau_i)$. The lemma follows by noting that $RBF(\mathtt{hp}_{oth}^{k}(\tau_i), \Delta)$ sums all terms $rbf_h(\Delta)$ due to each $\tau_h \in \mathtt{hp}_{oth}^{k}(\tau_i)$, where each term $rbf_h(\Delta)$ bounds the individual demand due to τ_h , in any interval of length $\Delta > 0$. ◀

Next, we consider the interference due to messages. Before starting, we derive a bound on the delay due to each message processed by a flow-controller or listener thread.

▶ **Lemma 7.** *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*

$$\delta_i^t(m_z) \triangleq \begin{cases} \delta^f(m_z) \cdot N_{sub}(m_z) & \text{if } t = f, \\ \delta^l(m_z) & \text{if } t = l. \end{cases} \tag{3}$$

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 **R2**, 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 Equation (3). The second branch follows by noting that, due to rule **R4**, for each message instance processed by the listener only one message instance at a time is forwarded to the subscriber. ◀

With the previous result in place, Lemma 8 bounds the inter-thread message interference experienced by an arbitrary message m_z under analysis.

▶ **Lemma 8.** *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 $\tau_j^t \in \boldsymbol{hp}_{mw}^{k}(\tau_i^t)$. In any window of length $\Delta > 0$, the inter-thread message interference of an instance of m_z is bounded by:*

$$I_{i,z}^{inter}(\Delta) \triangleq \sum_{\tau_j^t \in \boldsymbol{hp}_{mw}^{k}(\tau_i^t)} \sum_{m_r \in \tau_j^t} \eta_{r,j}(\Delta + \overline{R}_j^t(m_r) - \epsilon) \cdot \delta_j^t(m_r), \text{ with } t \in \{f, l\} \tag{4}$$

Proof. By definition, $I_{i,z}^{\mathrm{inter}}(\Delta)$ includes all the interference due to messages in other middleware level threads $\tau_j^{\mathrm{t}} \in \mathtt{hp}_{\mathrm{mw}}^{k}(\tau_i^{\mathrm{t}})$ 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 5 and 7, 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. Next, we present the bounds on the intra-thread message interference, which depends on the policy used in the middleware-level thread. Before proceeding, we bound the number of self-interfering instances in Lemma 9.

▶ **Lemma 9.** *Let $\overline{R}(m_z)$ be a response-time bound for m_z in an arbitrary middleware-level thread. In any interval of length Δ, the number of self-interfering instances to an arbitrary instance of a message m_z in a middleware-level thread $\tau_i \in \Gamma_{mw}$ is bounded by*

$$si_i(m_z, \Delta) \triangleq \max(0, \eta_{z,i}(\Delta + \overline{R}(m_z) - \epsilon) - 1). \tag{5}$$

Proof. The lemma follows from Lemma 5 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 precision of the self-interference bound can be further tightened at the cost of testing a search space of multiple message release times in a busy window [17], thus complicating the analysis and requiring additional running time. To keep the analysis simple, this potential improvement is left as future work.

`HIGH_PRIORITY` policy. Under this policy, each instance of m_z under analysis can be delayed by: **(i)** low-priority messages causing delay due to non-preemptive message handling (rule **R5**), **(ii)** equal-priority messages enqueued before and thus being prioritized by the FIFO tie-break, 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, so 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 considering equal-priority messages in Lemma 10.

▶ **Lemma 10.** *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*[2]

$$\mathcal{D}^{ep}_i(\Delta) = \biguplus_{m_r \in ep_i(m_z)} \{\delta^t_i(m_r)\} \otimes \overline{\eta}_{r,i}(\Delta), \text{ with } t \in \{f, l\} \tag{6}$$

where

$$\overline{\eta}_{r,i}(\Delta) \triangleq \begin{cases} si_i(m_z, \Delta) & \text{if } z = r, \\ \eta_{r,i}(\Delta + \overline{R}^t_i(m_r) - \epsilon) & \text{otherwise}, \end{cases} \tag{7}$$

where $\delta^t_i(m_r)$ is given by Lemma 7 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 7, each message contributes with a delay of at most $\delta^t_i(m_r)$. By Lemma 9, 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 5. ◀

Hereafter, the notation $\Sigma(x, M)$ denotes the sum of the x *largest* elements in a multiset M. If M contains less than x elements, all elements in M are summed.

▶ **Lemma 11.** *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*

$$I^{ep}_{i,z}(\Delta) \triangleq \Sigma(M^{HP,j}_i - 1, \mathcal{D}^{ep}_i(\Delta)). \tag{8}$$

[2] 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., $\{1, 4\} \otimes 2 = \{1, 1, 4, 4\}$.

Proof. By definition, the j-th priority queue of τ_i^{t} has size $M_i^{\mathtt{HP},j}$. Therefore, at most $M_i^{\mathtt{HP},j}-1$ message instances with same priority can interfere with the one under analysis. By Lemma 10, the delays of message instances with same priority that may contribute to the intra-thread interference of m_z are contained into the multiset $\mathcal{D}_i^{\mathrm{ep}}(\Delta)$. Hence $\Sigma(M_i^{\mathtt{HP},j}-1, \mathcal{D}_i^{\mathrm{ep}}(\Delta))$ bounds the intra-thread interference generated by messages with the same priority of m_z. ◀

Next, we provide a bound to the intra-thread interference due to messages with lower priority, which occurs because each message is handled in a non-preemptive manner [23, 43] locally to each thread.

▶ **Lemma 12.** *Consider an instance of a message m_z in a middleware-level thread using the* `HIGH_PRIORITY` *policy $\tau_i^t \in \Gamma_{mw}$ and a time window of length Δ. It holds*

$$I_{i,z}^{lp}(\Delta) \triangleq \max_{m_r \in lp_i(m_z)} \delta_i^t(m_r), \text{ with } t = f. \tag{9}$$

Proof. Low-priority messages can contribute with at most *one* instance due to the non-preemptive handling of messages (rule **R5**). The corresponding delay can be at most equal to the longest delay $\delta_i^t(m_r)$, yielding the bound $I_{i,z}^{\mathrm{lp}}(\Delta)$. ◀

Differently from equal-priority messages (Lemma 11), the bound for higher-priority messages cannot rely on message queue sizes. This is because the message under analysis is placed in a different queue: thus, the bound can only leverage the message arrival curves on the middleware-level thread under consideration. A bound on the intra-thread interference due to higher-priority messages is reported in Lemma 13.

▶ **Lemma 13.** *Consider an instance of a message m_z in a middleware-level thread using the* `HIGH_PRIORITY` *policy $\tau_i^t \in \Gamma_{mw}$ and a time window of length Δ. Let $\overline{R}_i^t(m_r)$ be a response-time bound for a higher priority message m_r in τ_i^t, it holds*

$$I_{i,z}^{hp}(\Delta) \triangleq \sum_{m_r \in hp_i(m_z)} \eta_{r,i}(\Delta + \overline{R}_i^t(m_r) - \epsilon) \cdot \delta_i^t(m_r), \text{ with } t = f. \tag{10}$$

Proof. Higher-priority messages can interfere with all instances that are pending in an arbitrary interval $[\bar{t}, \bar{t}+\Delta)$. By Lemma 5, the number of such instances is bounded by $\eta_{r,i}(\Delta + \overline{R}_i^t(m_r) - \epsilon)$, each one delaying for up to $\delta_i^t(m_r)$. ◀

With Lemmas 11, 12, and 13 in place, we have all the interference components to instantiate a response-time bound under the `HIGH_PRIORITY` policy, which we present later in Theorem 15.

`FIFO` policy. Under the `FIFO` policy, all messages of a middleware-level thread τ_i^{t} are handled in a single queue of size $M_i^{\mathtt{F}}$. Since the tie-break 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 11, but considering $M_i^{\mathtt{F}}$ in place of $M_i^{\mathtt{HP},j}$, and using set τ_i^{t} in the union of Lemma 10 instead of $ep_i(m_z)$.

Response-time bound. We provide the response-time bound by proceeding in two steps. First, we bound the start time of an arbitrary message instance m_z' under analysis, i.e., the time in which the middleware-level thread starts serving non-preemptively m_z', locally to the middleware-level thread under consideration. However, note that m_z' can still suffer thread-level and inter-thread message interference due to higher-priority threads. Later, we bound the response time by leveraging the start-time bound and the fact that once m_z' started being served, it cannot experience intra-thread message interference.

Theorem 14 bounds the start-time of an arbitrary message instance m'_z in a middleware-level thread τ_i.

▶ **Theorem 14.** *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*

$$sbf_k(S^*) \geq \epsilon + I_{i,z}^{intra}(S^*) + I_{i,z}^{thread}(S^*) + I_{i,z}^{inter}(S^*), \tag{11}$$

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

Proof. By Lemmas 6 and 8, $I_{i,z}^{\text{thread}}(S^*)$ and $I_{i,z}^{\text{inter}}(S^*)$ bound the thread-level and inter-thread message interference, respectively. The intra-thread message interference is bounded by $I_{i,z}^{\text{intra}}(S^*)$ due to Lemmas 11, 12 and 13, if the middleware-level thread adopts the `HIGH_PRIORITY` policy, or Lemma 11 (slightly modified as suggested above) if the `FIFO` policy is used. If S^* satisfies Equation (11), 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 **R8**), m'_z starts being served in the middleware-level thread no later than time $A + S^*$ and the theorem follows. ◀

Finally, Theorem 15 provides a response-time bound R^*.

▶ **Theorem 15.** *Consider an arbitrary instance m'_z of message m_z processed by a middleware-level thread τ_i^t released at a time A. If S^* is defined as in Theorem 14 and R^* is the least positive solution (if any) of the following inequality*

$$sbf_k(R^*) \geq \epsilon + I_{i,z}^{intra}(S^*) + I_{i,z}^{thread}(R^*) + I_{i,z}^{inter}(R^*) + \delta_i^t(m_z), \tag{12}$$

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

Proof. By Theorem 14, m'_z starts being served no later than time $A + S^*$. After that, due to rule **R5**, it starts being processed non-preemptively in the middleware-level thread and it does not suffer intra-thread interference anymore. Hence $I_{i,z}^{\text{intra}}(S^*)$ 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 $I_{i,z}^{\text{inter}}(R^*)$ and $I_{i,z}^{\text{thread}}(R^*)$, respectively. If R^* satisfies Equation (12), 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 $\delta_i^t(m_z)$ to process m'_z itself. Hence, the theorem follows. ◀

Response-time bounds for the flow-controller and listener threads required to compute the DDL (see Equation (1), terms $R_i^f(m_z)$ and $R_j^l(m_z)$) can be computed with the results presented in this section, considering either the `HIGH_PRIORITY` or `FIFO` policy for the flow controller, and the `FIFO` policy for the listener.

5.2 Arrival-curve propagation

The DDL bound derived in the previous section is based on the knowledge of the arrival curves of the various threads in the system. Using the standard arrival curve propagation approach of CPA [32], they can be derived from the externally-provided arrival curves $\eta_i^p(\Delta)$ of publisher threads $\tau_i^{\text{p}} \in \Gamma_{\text{p}}$, 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 $m_z(\tau_i^{\mathrm{p}}, \theta_j)$ 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 Figure 4. 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:

$$\eta_{z,x}^{\mathrm{f}}(\Delta) = \eta_i^p(\Delta + \overline{R}(\tau_i^{\mathrm{p}}) - \epsilon) \cdot w_i^j, \tag{13}$$

where $\overline{R}(\tau_i^{\mathrm{p}})$ is a response-time bound for the publisher thread, which can be derived with standard methods for response-time analysis under preemptive fixed-priority scheduling [38]. As shown in Figure 4, the message then passes through the network, with a delay $\delta^{\mathrm{net}}(m_z)$, and it is received by the listener thread τ_y^{l}. The arrival curve of the message within the listener thread is hence computed as:

$$\eta_{z,y}^{\mathrm{l}}(\Delta) = \eta_{z,x}^{\mathrm{f}}(\Delta + \overline{R}_x^{\mathrm{f}}(m_z) + \delta^{\mathrm{net}}(m_z) - \epsilon). \tag{14}$$

Finally, the arrival curve of the subscriber thread is obtained with an OR-activation semantics [33, 32] 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.,

$$\eta_q^s(\Delta) = \sum_{\theta_j \in \Theta(\tau_q^{\mathrm{s}})} \sum_{m_z \in \mathcal{M}(\theta_j)} \eta_{z,y}^{\mathrm{l}}(\Delta + \overline{R}_y^{\mathrm{l}}(m_z) - \epsilon). \tag{15}$$

5.3 Analysis summary and its applicability

Analysis summary. Algorithm 1 summarizes the analysis proposed in this paper for the purpose of computing the DDL according to Equation (1). The pseudo-code relies on global variables to store response-time bounds and their candidates (line 2). Then, function `COMPUTE_RT_BOUNDS()` needs to be called to populate the global variables $R_x^{\mathrm{f}}(m_z)$ and $R_y^{\mathrm{l}}(m_z)$ with the response-time bounds of each message m_z in the corresponding flow-controller and listener threads, respectively.

The calculation leverages functions `ResponseTimeBound_FlowController()` (line 17) and `ResponseTimeBound_Listener()` (line 27) that properly instantiate $I_{i,z}^{\mathrm{intra}}(\Delta)$ as discussed in Section 5.1, while $I_{i,z}^{\mathrm{inter}}(\Delta)$ and $I_{i,z}^{\mathrm{thread}}(\Delta)$ are defined as in Lemmas 6 and 8 in both the cases. As discussed in Section 5.1, the computation of the response-time bounds $R_x^{\mathrm{f}}(m_z)$, $R_y^{\mathrm{l}}(m_z)$ (functions `ResponseTimeBound_FlowController()` and `ResponseTimeBound_Listener()`) cyclically depends on the existence of pre-existing response-time bounds $\overline{R}_x^{\mathrm{f}}(m_z)$, $\overline{R}_y^{\mathrm{l}}(m_z)$. The dependency is broken by initializing the bounds to zero (line 6) and performing an outer loop (lines 7-15) until a global fixed-point is reached (i.e., $R_x^{\mathrm{f}}(m_z) \neq \overline{R}_x^{\mathrm{f}}(m_z)$ and $R_y^{\mathrm{l}}(m_z) \neq \overline{R}_y^{\mathrm{l}}(m_z)$ for each message m_z) and performing arrival curve propagation (see Section 5.2) inside the loop. Convergence is guaranteed by the fact that response-time estimates (variables $\overline{R}_y^{\mathrm{l}}(m_z)$ and $\overline{R}_x^{\mathrm{f}}(m_z)$) never decreases. Response-time bounds for publisher and subscriber threads also need to be computed (according to standard fixed-priority scheduling, line 13) inside the loop, updating global response time variables (as those initialized in line 6) to be used in the arrival curve propagation process (e.g., Equation (13), not detailed in the pseudo-code for brevity). When `COMPUTE_RT_BOUNDS()` completes, the global variables are configured to the correct response-time bound values, and the function `DDL`(m_z, $\delta^{\mathrm{net}}(m_z)$), at line 33 of the Algorithm 1, can be used to compute the DDL of a given message m_z according to Equation (1) by providing the network delay $\delta^{\mathrm{net}}(m_z)$ as an input parameter.

Algorithm 1 Pseudo-code of the DDL analysis.

global variables $\forall\ \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$, define ▷ for each message in the system
$R^{\mathrm{f}}_x(m_z)$, $R^{\mathrm{l}}_y(m_z)$, $\overline{R}^{\mathrm{f}}_x(m_z)$, $\overline{R}^{\mathrm{l}}_y(m_z)$ to store response-time bounds and the corresponding candidates

function `COMPUTE_RT_BOUNDS`( )
  $\forall\ \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$: ▷ for each message in the system:
    $R^{\mathrm{f}}_x(m_z) \leftarrow 0, R^{\mathrm{l}}_y(m_z) \leftarrow 0, \overline{R}^{\mathrm{f}}_x(m_z) \leftarrow 0, \overline{R}^{\mathrm{l}}_y(m_z) \leftarrow 0$
  do
    $\forall\ \theta_j \in \Theta, m_z \in \mathcal{M}(\theta_j)$: ▷ for each message in the system
      $\overline{R}^{\mathrm{f}}_x(m_z) \leftarrow R^{\mathrm{f}}_x(m_z), \overline{R}^{\mathrm{l}}_y(m_z) \leftarrow R^{\mathrm{l}}_y(m_z)$
      `FC_SCHED_POL` $\leftarrow$ scheduling policy of the flow-controller thread handling m_z
      $R^{\mathrm{f}}_x(m_z) \leftarrow$ `ResponseTimeBound_FlowController`(m_z, `FC_SCHED_POL`)
      $R^{\mathrm{l}}_y(m_z) \leftarrow$ `ResponseTimeBound_Listener`(m_z)
      `compute response time bounds for application threads`
      `perform arrival curve propagation` ▷ see Section 5.2
  while `no more response-time bounds updates` $\forall\ m_z$

function `ResponseTimeBound_FlowController`(m_z, `FC_SCHED_POL`)
  switch `FC_SCHED_POL` **do**
    case `HIGH_PRIORITY`:
      Set $I^{\mathrm{intra}}_{x,z}(\Delta) \leftarrow I^{\mathrm{ep}}_{x,z}(\Delta) + I^{\mathrm{lp}}_{x,z}(\Delta) + I^{\mathrm{hp}}_{x,z}(\Delta)$
        ▷ where $I^{\mathrm{ep}}_{x,z}(\Delta), I^{\mathrm{lp}}_{x,z}(\Delta), I^{\mathrm{hp}}_{x,z}(\Delta)$ are bounded as in Lemmas 11, 12, and 13
    case `FIFO`:
      Set $I^{\mathrm{intra}}_{x,z}(\Delta) = I^{\mathrm{ep}}_{x,z}(\Delta)$ ▷ using Lemma 11
        ▷ with $M^{\mathtt{F}}_x$ in place of $M^{\mathtt{HP},j}_x$, and with τ^{f}_x in the union of Lemma 11 in place of $ep_x(m_z)$
  Compute S^* using Theorem 14, compute R^* using Theorem 15

function `ResponseTimeBound_Listener`(m_z)
  Set $I^{\mathrm{intra}}_{y,z}(\Delta) = I^{\mathrm{ep}}_{y,z}(\Delta)$ ▷ using Lemma 11
    ▷ with $M^{\mathtt{F}}_y$ in place of $M^{\mathtt{HP},j}_y$, and with τ^{l}_y in the union of Lemma 11 in place of $ep_y(m_z)$
  Compute S^* using Theorem 14, compute R^* using Theorem 15
  return R^*

function `DDL`(m_z, $\delta^{\mathrm{net}}(m_z)$) ▷ to be called after `COMPUTE_RT_BOUNDS()`
  return $R^f_x(m_z) + \delta^{\mathrm{net}}(m_z) + R^l_y(m_z)$

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

Linux – SCHED_FIFO. A natural fit for our method is to analyze the timing behavior of FastDDS-based applications running on the `SCHED_FIFO` scheduling class of Linux, which provides a fixed-priority scheduler fulfilling the assumptions of our model. In this case, each core provides the full supply to the scheduled applications, since no reservation-based mechanism is provided. Hence, $sbf_k(\Delta) = \Delta, \forall k$.

Linux – SCHED_DEADLINE and QNX APS. Thanks to the supply-bound function abstraction, our analysis is suitable also for being applied to systems using the `SCHED_DEADLINE` scheduler of Linux, a reservation-based scheduler. Under `SCHED_DEADLINE`, each thread can be individually isolated from a temporal perspective by associating it with a budget and period pair [1, 11]. The corresponding definition for $sbf_k(\Delta)$ is available in the literature [10, 15, 40]. Thanks to the temporal isolation, $I^{\mathrm{thread}}_{i,z}(\Delta) = 0$ and $I^{\mathrm{inter}}_{i,z}(\Delta) = 0$. Furthermore, our analysis also generalizes to the QNX APS reservation-based scheduler [6, 22], by considering only threads allocated to the same APS partition of the thread under analysis when deriving $I^{\mathrm{thread}}_{i,z}(\Delta)$ and $I^{\mathrm{inter}}_{i,z}(\Delta)$.

Processing chains and ROS. Thanks to the compositionality of our approach, our analysis can be used to study the end-to-end response-time of data-driven distributed applications [7, 29, 64, 65]. Furthermore, our approach makes few assumptions on how application-level threads are scheduled, making it easily extensible to work with methods to study the response-time of processing chains using ROS 2 [13, 17], which leverages the DDS as a lower-layer middleware and hence it is a practical use case for the DDS. However, none of the analyses for ROS 2 currently available in the literature provides a method to bound the DDS-related delay. For example, the analysis in [17] for ROS 2 models the delay due to the DDS as a single parameter and suggests estimating it empirically. The analysis of this paper (Equation (1)) presents the first analytical solution to provide a theoretically-sound bound on the single-parameter DDS-related delay of [17], which can therefore be used as a complement of previous work on ROS 2 processing chains.

6 Evaluation

First, we evaluated our analysis on a case study based on the WATERS 2019 Challenge by Bosch [30], which consists of a representative autonomous driving application. Then, we report on a comparison between the analysis results and actual measurements on a real platform running FastDDS and a relatively simple application.

6.1 WATERS 2019 Challenge case study

The model for the Challenge provides parameters such as periods, worst-case execution times, and data exchanged among threads (i.e., shared labels) for nine threads. While the challenge model was not designed to work with the DDS, the target application is a good fit to work with a pub/sub paradigm. Figure 5 illustrates how the Challenge application was adapted to work with the DDS. Shared labels were modeled as topics (represented by ellipses) having the same payload size as labels, with the following meaning: when a task writes on a label is *publishing* messages on that label, i.e., topic, and a task reading on a label is *receiving* messages from that topic. Threads performing only reads on labels were considered as subscribers with data-driven activation based on the subscription to the corresponding topics (e.g., see the `DASM` task). For threads that are both reading and writing labels (`LOC`, `EKF`, and `PLAN`), two sub-threads were identified, representing the subscriber and a publisher (denoted in the figure with the prefixes "S_" and "P_" respectively). In this way, the original periods of the Challenge were preserved for the publishers. Given the original WCET e_i of the challenge model, individual WCETs of the corresponding publisher and subscriber thread were derived as $e_i^{\text{pub}} = e_i \cdot \alpha$ and $e_i^{\text{sub}} = e_i \cdot (1 - \alpha)$, with $\alpha \in [0, 1]$. The other threads were left unaltered. The parameter α is introduced to split the WATERS Challenge's threads into publisher-subscriber pairs and can be used to regulate how much computation time to assign to publish and subscribe parts of each thread.

Message delays in the flow-controller and listener. To run the analysis, it is necessary to know the worst-case delays $\delta^{\text{f}}(m_z)$ and $\delta^{\text{l}}(m_z)$ to process a message in the flow-controller and listener thread, respectively. To estimate such parameters, we developed a FastDDS application consisting of a publisher, its flow-controller, a subscriber, and its listener. The two application-level threads communicate through a single topic, using UDP through the loopback interface. The application was executed on an 8-core *Dell Optiplex 7070* machine running Ubuntu 20.04. The flow-controller and listener threads were mapped to two different cores with the highest priority. Each message was processed 50000 times by each middleware

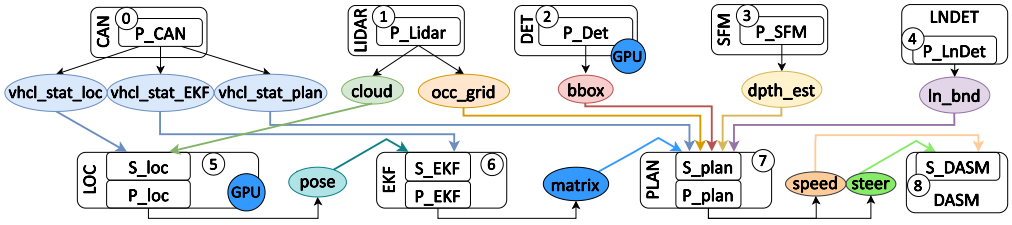

Figure 5 WATERS 2019 Challenge adapted to use a pub/sub paradigm.

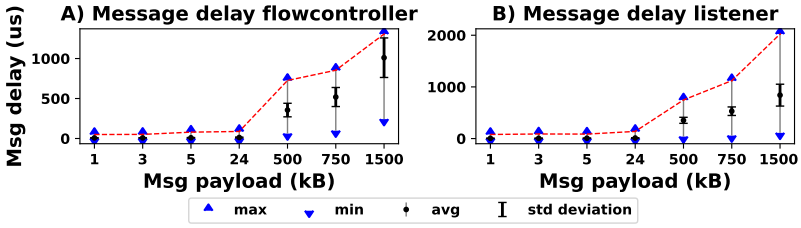

Figure 6 Estimation of the message delay.

thread. Middleware threads were configured to collect data about the execution time of each processed message. Figure 6 illustrates the results. For each payload size (x-axis) used in the WATERS 2019 Challenge, the graph shows the minimum, maximum, average, and standard deviation of message delays (y-axis). Both graphs show the same trend. For payload sizes $\leq$ 24kB, the execution time trend is fairly constant and does not exceed 125μs and 200μs for flow-controller and listener messages, respectively. As payload size exceeds loopback MTU (64kB), the fragmentation and reassembly of UDP packets was found to cause relevant overheads, affecting the processing time of the messages.

Analysis results. Next, we discuss the results of the analysis of this case study, which was implemented using the *pyCPA* framework [25]. The priorities of publishers were set according to rate-monotonic. In this case study, at most one thread publishes on each topic. Therefore, we assign to each topic the same priority of the corresponding publisher (which is then inherited by each message sent through the topic). Whenever a publisher publishes on multiple topics, the topic (i.e. message) associated with a smaller payload size is assigned a higher priority. Subscribers `S_loc, S_EKF, S_plan` inherited the priority of the corresponding publishers, while `S_DASM` priority was set to the highest priority in the system because of providing the application's output. We set $\delta^{\text{net}}(m) = 0$ since we studied threads running on the same computing node. We evaluated the analysis on a vast range of configurations, where multiple design-level parameters were varied: **(i)** the task-to-core assignment, **(ii)** the priorities of application-level and middleware-level threads, **(iii)** the number of flow-controllers and the topics-to-flow-controller assignments, **(iv)** the message priorities in the flow-controller threads.

Among them, we selected four relevant system configurations (with $\alpha = 0.95$) and we discuss their trade-offs:

(A) A configuration in which each publisher has its own flow-controller, and all the threads are exclusively assigned to a core, and no thread-level interference can occur;

(B) A configuration with *eight* flow-controllers, where flow-controllers and listeners are in the same core of their publishers and subscribers, respectively;

(C) A configuration with *two* flow-controllers, in which one flow-controller manages messages with lower payload ({1, 3, 5, 24} kB), while the other handles messages with higher payload ({500, 750, 1500} kB).

(D) A configuration with *one* flow-controller handling all messages, allocated on a dedicated core.

In (C) and (D), listeners are allocated to the same core of corresponding subscribers. For each configuration, Figure 7 shows the DDL (y-axis) for each message (x-axis).

Configuration A. Figure 7 (A) shows the results of the configuration, where threads execute exclusively on a core. Thus, a message can only suffer interference due to messages within the same flow-controller thread, which occurs for publishers sending multiple different messages. We considered two scenarios for the `HP` and `F` policies of the flow-controller. In this configuration, both sending policies found not to have any significant effects on the DDL. For the `vchl_stat_plan` message, the `F` policy generates a slightly higher DDL than the `HP`, due to the fact that, under `HP`, this message has the highest priority within the flow-controller.

Configuration B. In this configuration, we evaluated the effects of the relative priority between a publisher and its flow-controller thread. To this end, we considered four different cases in Figure 7 (B): **(i)** the HP policy of the flow-controller (settings `1-HP` and `2-HP`), **(ii)** the FIFO policy of the flow-controller (`1-F` and `2-F`), **(iii)** flow-controllers with higher priorities than publishers (`1-HP` and `1-F`), **(iv)** flow-controllers with lower priorities than publishers (`2-HP` and `2-F`). When flow-controllers have a high priority, results show the same behavior and same values of Figure 7 (A). When we consider lower-priority flow-controllers in the cases `2-HP` and `2-F`, the DDL increases for each message that can suffer from the publisher execution interference. The longest DDLs ($16.5ms$ for `2-HP` and $17ms$ for `2-F`) are observed for the `occ_grid` message, with a 3x latency increment w.r.t. (A).

Configurations C and D. Figures 7 (C) and 7 (D) show the results of the configurations with two and one flow-controller, respectively. Scenarios `1-HP-L>S` and `1-F-L>S` consider the HP and FIFO policies of the flow-controllers when listeners have higher priorities than their subscribers. Scenarios `2-HP-L<S` and `2-F-L<S` consider listeners with lower priorities than their subscribers. Considering `1-HP-L>S` and `2-HP-L<S`, the configuration in inset (D) leads low-priority messages (e.g., `ln_bnd`, `pose`, `dpth_est`) to suffer from a significant interference from the processing of high-priority messages compared to configuration in inset (C). Moreover, due to the non-preemptiveness of the sending operation, for each message, the DDL of the configuration (D) always accounts the time needed to process the highest-payload message (`cloud` message, with size $1500kB$). Differently, in configuration (C), lower-payload messages just suffer at most from of non-preemptiveness processing delay due to messages with size $24kB$. This is an advantage due to having two flow-controllers that manage different data flows. In both configurations, lower-priority listeners cause larger DDL for all messages due to subscriber-related interference.

6.2 Comparing analysis bounds with measured DDLs

Next, we compare the results of our analysis with empirical measurements collected from a real FastDDS application executed on the previously mentioned *Optiplex 7070* platform, running Ubuntu 20.04. The considered application consists of a publisher τ_1^{p}, with its flow-controller τ_2^{f}, and a subscriber τ_4^{s}, with its listener τ_3^{l}, exchanging data over three different topics (θ_1, θ_2, θ_3). Messages (named $m(\theta_1)$, $m(\theta_2)$, $m(\theta_3)$) have the same payload size ($1kB$). Message delays are set according to the measurements reported in Figure 6 for $1kB$ payloads. Threads τ_1^{p} and τ_2^{f} are allocated to the same core c_1 and assigned to the two

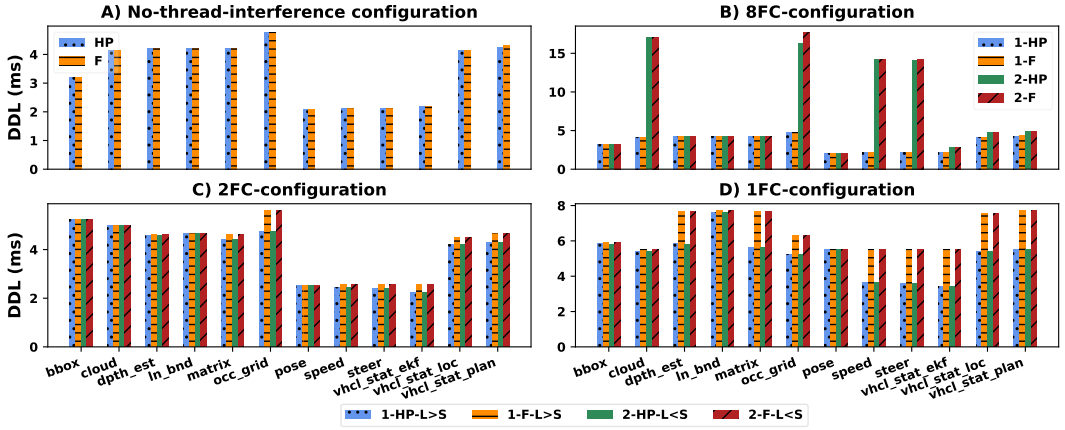

Figure 7 Experimental results under four representative configurations of the case study.

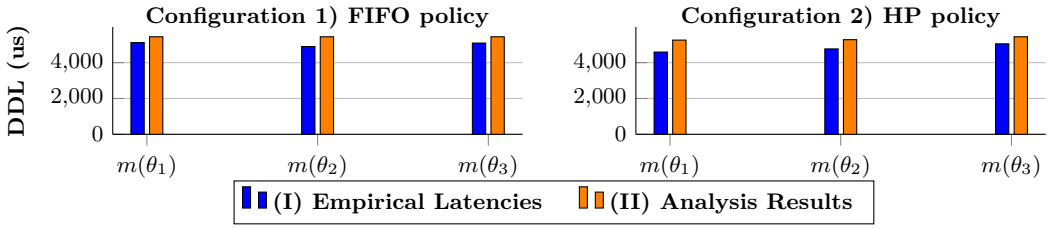

Figure 8 Results from FastDDS app and analysis related to Configurations 1) and 2).

highest priorities, with τ_2^{f} having higher priority than τ_1^{p}. τ_4^{s} and τ_3^{l} are mapped, respectively, to cores c_3 and c_2, and set with the highest priority in their respective core. τ_1^{p} is configured as a periodic thread with period $2ms$. Therefore, it is characterized by an arrival curve $\eta_1^p(\Delta) = \lceil \frac{\Delta}{T} \rceil$. On the Optiplex platform, middleware threads of the FastDDS application are configured to measure the DDL for each message over 50000 samples. We tested two different configurations: **Conf. 1)**, in which messages in the flow-controller are scheduled under `FIFO` policy; **Conf. 2)** in which the flow-controller schedules messages under the `HIGH_PRIORITY` policy and topics are assigned to a unique priority such that lower topic subscript identifiers indicate higher priority values. Figure 8 shows the DDL (y-axis) for each message (x-axis) obtained through measurements and by the analysis for both configurations.

Conf. 1). Under `FIFO` policy, each flow-controller message can interfere with others. Our analysis accordingly computes the same DDL bound ($5446us$) for all of them. Comparing the DDL bound with the measured values on the Optiplex platform we can observe that values do not exceed the DDL bound found by the analysis, corroborating its validity.

Conf. 2). Under `HP` policy, both the analysis and the measurements on the Optiplex platform show DDL values that depend upon the message priority. Moreover, we can observe that for the lowest priority message (i.e., $m(\theta_3)$), the DDL bound of the analysis equals the DDL bound found in the **Conf. 1)**. Also in this case, empirical DDL values do not exceed the DDL bounds found with the analysis.

In Table 2, we reported the relative distance, in percentage, between the DDL bound provided by the analysis and the measured value, showing that the DDL bounds found are tight for the considered application.

Table 2 Table of percentage relative distances: measurements vs. analysis bound.

Message	Configuration (1) FIFO	Configuration (2) HIGH_PRIORITY
$m(\theta_1)$	6.3 %	14.5 %
$m(\theta_2)$	11.1 %	10.7 %
$m(\theta_3)$	6.9 %	7.7 %

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 attempt to model the DDS from a real-time perspective and provide real-time analysis for DDS-based communications.

Most of the previous research on DDS focused on empirical performance measurement. For instance, Bellavista et al. [8] compared the DDS implementation OpenSlice with Connext-DDS by Real-Time Innovations [54]. Krinkin et al. [36] proposed a framework to assess the effectiveness of various DDS implementations in terms of message transport latency and throughput. Other works attempted to suggest potential improvements for DDS implementations. For example, Choi et al. [19] studied a real-time DDS setup over specialized packet-switching ASICs to enable Software Defined Networking (SDN). Peeck et al. [50] 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. [2] proposed the integration of a DDS implementation with a TSN protocol for real-time data transfers. Stevanato et al. [62] proposed a reference architecture for implementing virtualized DDS communications in a hypervisor-based multi-domain system. Finally, Scordino et al. [56] implemented in hardware some DDS functionalities. Other works considered other middlewares, e.g., OpenMP [58, 63], ROS 2 [3, 17, 20, 64, 66], the ROS-based framework Apex.OS [51], and RT-Appia [53]. However, none of the works addressing the analysis of ROS 2 provides analytical methods to bound the data-delivery latency of the DDS. Empirical evaluations of ROS 2 over different DDS implementations have been carried out by Maruyama et al. [41] and Kronauer et al. [37], providing guidelines on designing ROS 2 applications to minimize latencies.

8 Conclusion and Future Work

In this paper, we derived a compositional model for studying the timing of the DDS standard and we instantiated it for FastDDS. We inspected the FastDDS documentation and source code to build an accurate model capable of capturing the FastDDS-specific timing-related effects, and we corroborated our findings by running validation experiments on an actual FastDDS system. Building on the model, we derived an analysis to bound the data-delivery latency of messages. We evaluated our analysis based on the WATERS 2019 Industrial Challenge showing how thanks to our analysis, it becomes easily possible to compare a vast range of configurations without the need to deploy them on a real system.

Furthermore, we compared analysis results with actual measurements on a real platform running FastDDS and a relatively simple application, showing the tightness of our analysis for the specific use case. The proposed analysis will enable system designers to configure DDS-based systems, guiding choices such as thread-to-core allocation, priorities, and reservation budgets in a timing-constraints-driven perspective. It will set the foundation to account for DDS-related delays in analysis-driven orchestration algorithms, which will

be the subject of future work. Other directions for future research include combining this analysis with other analysis techniques [47] to improve the analysis precision, the holistic consideration of scheduling effects due to the DDS with OS overheads [24], I/O-related contention delays [59, 71], and network delays [14, 18, 44, 52] with special emphasis on Time-Sensitive Networking [49].

References

1 L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, 1998.

2 Tanushree Agarwal, Payam Niknejad, Mohammadreza Barzegaran, and Luigi Vanfretti. Multi-level time-sensitive networking (TSN) Using the Data Distribution Services (DDS) for synchronized three-phase measurement data transfer. *IEEE Access*, PP:1–1, September 2019.

3 Abdullah Al Arafat, Sudharsan Vaidhun, Kurt M Wilson, Jinghao Sun, and Zhishan Guo. Response time analysis for dynamic priority scheduling in ROS2. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 301–306, 2022.

4 AUTOSAR. Specification of Communication Management, 2020. URL: `https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_SWS_CommunicationManagement.pdf`.

5 Daniel Balouek-Thomert, Ali Reza Zamani Eduard Gibert Renart, and Manish Parashar Anthony Simonet. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, 33(6):1159–1174, 2019.

6 Matthias Becker, Dakshina Dasari, and Daniel 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.

7 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80:104–113, 2017.

8 P. Bellavista, A. Corradi, L. Foschini, and A. Pernafini. Data distribution service (DDS): A performance comparison of OpenSplice and RTI implementations. In *2013 IEEE Symposium on Computers and Communications (ISCC)*, pages 000377–000383, Los Alamitos, CA, USA, July 2013. IEEE Computer Society.

9 Luca Belluardo, Andrea Stevanato, Daniel Casini, Giorgiomaria Cicero, Alessandro Biondi, and Giorgio 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)*, pages 73–82, 2021.

10 Alessandro Biondi, Giorgio C. Buttazzo, and Marko Bertogna. Schedulability analysis of hierarchical real-time systems under shared resources. *IEEE Transactions on Computers*, 65(5):1593–1605, 2016.

11 Alessandro Biondi, Alessandra Melani, and Marko Bertogna. Hard constant bandwidth server: Comprehensive formulation and critical scenarios. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 29–37, 2014.

12 Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn 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.

13 Tobias Blaß, Daniel Casini, Sergey Bozhko, and Björn B. Brandenburg. A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 41–53, 2021.

14 C. Blumschein, I. Behnke, L. Thamsen, and 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), pages 1–8, Los Alamitos, CA, USA, May 2022. IEEE Computer Society.

15 Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

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

17 Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

18 Pierre-Julien Chaine, Marc Boyer, Claire Pagetti, and Franck Wartel. Egress-TT Configurations for TSN Networks. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS 2022, 2022.

19 Hyon-Young Choi, Andrew L. King, and Insup Lee. Making DDS really real-time with OpenFlow. In *2016 International Conference on Embedded Software (EMSOFT)*, 2016.

20 Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. PiCAS: New design of priority-driven chain-aware scheduling for ROS2. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–263. IEEE, 2021.

21 Rut Diane Cuebas, Seonghyeon Park, Youngeun Cho, Daechul Park, and Chang-Gun Lee. Extension of functionally and temporally correct simulation of cyber-systems of automotive systems based on ROS system. *Korean Information Science Society Academic Papers*, pages 1174–1176, 2019.

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

23 Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 2007.

24 Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira, and Tommaso 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.

25 Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional Performance Analysis in Python with pyCPA. In *In Proceedings of WATERS'12*, 2012.

26 Zheng Dong, Weisong Shi, Guangmo Tong, and Kecheng Yang. Collaborative autonomous driving: Vision and challenges. In *2020 International Conference on Connected and Autonomous Driving (MetroCAD)*, pages 17–26. IEEE, 2020.

27 eProsima. Fast-DDS, 2022. `https://fast-dds.docs.eprosima.com/en/latest/`.

28 eProsima. Fast-DDS Github repository, 2022. `https://github.com/eProsima/Fast-DDS`.

29 Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. Timing analysis of asynchronized distributed cause-effect chains. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

30 A. Hamann, D. Dasari, F. Wurst, I. Sañudo, N. Capodieci, P. Burgio, and M Bertogna. Waters industrial challenge 2019.

31 Arne Hamann, Selma Saidi, David Ginthoer, Christian Wietfeld, and Dirk Ziegenbein. Building end-to-end IoT applications with QoS guarantees. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

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

33 Marek Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, Technical University of Braunschweig, June 2004.

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

35 Bonjun Kim and Kiejin Park. Probabilistic delay model of dynamic message frame in flexray protocol. *IEEE Transactions on Consumer Electronics*, 55(1):77–82, 2009.

36 Kirill Krinkin, Antoni Filatov, Artyom Filatov, Oleg Kurishev, and Alexander Lyanguzov. Data distribution services performance evaluation framework. In *2018 22nd Conference of Open Innovations Association (FRUCT)*, pages 94–100, 2018.

37 Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard P. Fettweis. Latency analysis of ROS2 multi-node systems. *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 1–7, 2021.

38 J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171, 1989.

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

40 G. Lipari and E. Bini. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 151–158, 2003.

41 Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *EMSOFT '16*, New York, NY, USA, 2016. Association for Computing Machinery.

42 Philipp Mundhenk, Arne Hamann, Andreas Heyl, and Dirk Ziegenbein. Reliable distributed systems. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022.

43 Mitra Nasri and Bjorn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23. IEEE, 2017.

44 Ramon Serna Oliver and Gerhard 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*, pages 423–431. IEEE, 2009.

45 OMG. Supported QoS, April 2015. `https://www.omg.org/spec/DDS/1.4/PDF`.

46 OMG. The real-time publish-subscribe protocol dds interoperability wire protocol specification (v2.5), March 2021. `https://www.omg.org/spec/DDSI-RTPS/2.5/PDF`.

47 J.C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium*, pages 26–37, 1998.

48 G. Pardo-Castellote. OMG data distribution service: architectural overview. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, volume 1, pages 242–247 Vol.1, 2003.

49 Gaetano Patti, Lucia Lo Bello, and Luca Leonardi. Deadline-Aware Online scheduling of TSN flows for automotive applications. *IEEE Transactions on Industrial Informatics*, 2022.

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

51 Michael Pöhnl, Alban Tamisier, and Tobias Blass. A Middleware Journey from Microcontrollers to Microprocessors. In *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 282–286, 2022.

52 Hootan Rashtian and Sathish Gopalakrishnan. Balancing message criticality and timeliness in IoT networks. *IEEE Access*, 7:145738–145745, 2019.

53 Joao Rodrigues, Hugo Miranda, João Ventura, and Luıs Rodrigues. The design of RT-Appia. In *Proceedings Sixth International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 261–268. IEEE, 2001.

54 RTI. Connext-DDS, 2013. `https://www.rti.com/products/dds-standard`.

55 Johannes Schlatow and Rolf Ernst. Response-time analysis for task chains with complex precedence and blocking relations. *ACM Trans. Embed. Comput. Syst.*, September 2017.

56 Claudio Scordino, Angela Gonzalez Mariño, and Francesc Fons. Hardware Acceleration of Data Distribution Service (DDS) for Automotive Communication and Computing. *IEEE Access*, 10:109626–109651, 2022.

57 Katherine Scott, Chris Lalancette, and Audrow Nash. 2021 ROS Middleware Evaluation Report, 2021. `https://github.com/osrf/TSC-RMW-Reports/tree/main/humble`.

58 Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quinones. Timing characterization of OpenMP4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 157–166. IEEE, 2015.

59 Alejandro Serrano-Cases, Juan M Reina, Jaume Abella, Enrico Mezzetti, and Francisco 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.

60 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13, 2003.

61 Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. Industrial Internet of Things: Challenges, opportunities, and directions. *IEEE transactions on industrial informatics*, 14(11):4724–4734, 2018.

62 A. Stevanato, A. Biondi, A. Biasci, and 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.

63 Jinghao Sun, Nan Guan, Zhishan Guo, Yekai Xue, Jing He, and Guozhen Tan. Calculating worst-case response time bounds for OpenMP programs with loop structures. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 123–135. IEEE, 2021.

64 Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ROS2 executors. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–243, 2020.

65 Yue Tang, Xu Jiang, Nan Guan, Dong Ji, Xiantong Luo, and Wang Yi. Comparing communication paradigms in cause-effect chains. *IEEE Transactions on Computers*, 2022.

66 H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J. Chen. End-to-end timing analysis in ROS2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022.

67 Ludovic Thomas, Ahlem Mifdaoui, and Jean-Yves Le Boudec. Worst-case delay bounds in time-sensitive networks with packet replication and elimination. *IEEE/ACM Transactions on Networking*, pages 1–15, 2022.

68 Vortex. Cyclone-DDS, September 2021. `https://projects.eclipse.org/projects/iot.cyclonedds`.

69 Tianze Wu, Baofu Wu, Sa Wang, Liangkai Liu, Shaoshan Liu, Yungang Bao, and Weisong Shi. Oops! It's Too Late. Your Autonomous Driving System Needs a Faster Middleware. *IEEE Robotics and Automation Letters*, 6(4):7301–7308, 2021.

70 Xiaoming Zhou and Piet Van Mieghem. Reordering of IP packets in Internet. In Chadi Barakat and Ian Pratt, editors, *Passive and Active Network Measurement*, pages 237–246, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

71 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022.