Response-Time Analysis of ROS 2 Processing Chains under Reservation-Based Scheduling

Daniel Casini
Scuola Superiore Sant'Anna, Pisa, Italy
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
daniel.casini@sssup.it

Tobias Blaß
Corporate Research, Robert Bosch GmbH, Germany
Saarland Informatics Campus, Saarland University, Saarbrücken, Germany
tobias.blass@de.bosch.com

Ingo Lütkebohle
Corporate Research, Robert Bosch GmbH, Germany
ingo.luetkebohle@de.bosch.com

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

Bounding the end-to-end latency of processing chains in distributed real-time systems is a well-studied problem, relevant in multiple industrial fields, such as automotive systems and robotics. Nonetheless, to date, only little attention has been given to the study of the impact that specific frameworks and implementation choices have on real-time performance. This paper proposes a scheduling model and a response-time analysis for ROS 2 (specifically, version "Crystal Clemmys" released in December 2018), a popular framework for the rapid prototyping, development, and deployment of robotics applications with thousands of professional users around the world. The purpose of this paper is threefold. Firstly, it is aimed at providing to robotic engineers a practical analysis to bound the worst-case response times of their applications. Secondly, it shines a light on current ROS 2 implementation choices from a real-time perspective. Finally, it presents a realistic real-time scheduling model, which provides an opportunity for future impact on the robotics industry.

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

Keywords and phrases ROS, real-time systems, response-time analysis, robotics, resource reservation

Related Version This document is an updated revision of DOI 10.4230/LIPIcs.ECRTS.2019.6. This version has been revised to incorporate an erratum concerning Lemma 8.

Supplement Material The associated source code is available at `https://github.com/boschresearch/ros2_response_time_analysis`.

1 Introduction

ROS, the *Robot Operating System* [43], is one of the most popular frameworks for designing and developing Linux-based robots. Powering over 100 different robot designs, it is used by tens of thousands of developers and researchers in both industry and academia [4, 22]. However, after over a decade of development and in the face of increasingly demanding applications, it became clear to the ROS community that the framework is held back by several long-standing shortcomings and architectural limitations that cannot be rectified in a backwards-compatible manner. This motivated the development of ROS 2, a complete refactoring of ROS that puts the successful concept onto a modernized and improved foundation. Of particular interest to us, a major design goal of ROS 2 is to improve the real-time capabilities of the framework, enabling the implementation of time-critical control paths inside ROS [24].

Safely implementing such control paths requires predicting the end-to-end latency (or *response time*) of time-critical *processing chains*. For instance, such a chain might cover all steps from a sensor, via a controller, to the final actuator and span multiple software components, multiple cores, and even multiple hosts. Although the end-to-end latency problem is well-studied in the literature, existing work often assumes idealized scheduling models that are not always directly applicable in real systems. Case in point, the ROS[1] scheduling approach (as described in detail in Section 3) does not match any of the classic results on bounding end-to-end latencies. ROS developers therefore have to resort to fully prototyping and deploying a design to measure its timing properties, which severely limits the degree to which the design space can be explored in practice.

While the scheduling model of operating systems like Linux has been studied extensively, this is not the case for ROS. Even though it is a middleware layer and not a proper operating system, its effects on an application's runtime behavior are substantial, rivaling or even exceeding that of the underlying OS. For example, ROS multiplexes independent message handlers onto shared threads using custom scheduling policies. Consequently, applications running on top of ROS are subject to the scheduling decisions of the underlying operating system *and* the middleware layer, with complex and interdependent effects on timing.

An additional complication stems from one of the key strengths of ROS: its modular structure. ROS emphasizes composing existing, battle-tested components instead of reimplementing common subsystems for each robot from scratch. While this greatly simplifies and speeds up robot development, it also obfuscates the overall timing behavior. This problem is further aggravated by ROS' event-driven design style, which gives rise to data dependencies and potentially long processing chains. As a result, it is extremely difficult for developers to anticipate, or even just understand, the timing of processing chains that cross multiple, loosely coupled components, most of which are developed by independent teams all around the world. Realistically, automated end-to-end response-time analysis is thus required to safely employ ROS in time-critical situations.

In this paper, we seek to lay the theoretical foundations for such an automated analysis tool by exploring the temporal behavior of ROS 2 "Crystal Clemmys" (released in December 2018) [5]. We present and validate a model of ROS applications running on top of a resource reservation scheduler such as `SCHED_DEADLINE` in Linux. Based on this model, we develop an end-to-end response-time analysis for ROS processing chains that takes the peculiarities and engineering constraints of the ROS environment into account. Finally, to demonstrate the applicability of our analysis to practical ROS components, we evaluate our approach on the popular `move_base` package [3], the core of the ROS navigation stack.

2 Background

This section introduces necessary background on the three pillars on which this paper rests. First, the structure of ROS and its execution model are presented. Then, we review resource reservations, an OS-level mechanism to isolate the resource consumption of processes, and last we review the Compositional Performance Analysis approach for response-time analysis.

[1] For brevity, we omit the version number and refer to ROS 2 as ROS in the remainder of the paper.

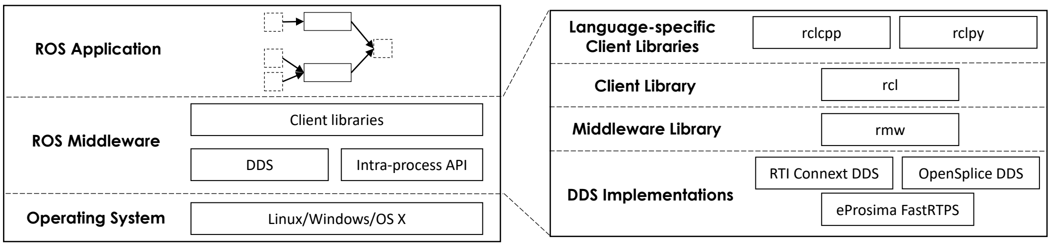


Figure 1 Layered structure of ROS.

2.1 ROS

ROS places great emphasis on modularity and composability. It therefore encourages strict separation between the logical structure of the application and the mapping of this structure onto hosts, processors, and threads. While the former is defined by the package developer, the latter is entirely up to the system integrator. This way, software modules can be developed independently of the target platform without losing the ability to taylor them to the deployment characteristics of a particular robot.

From a logical perspective, ROS applications are composed of *nodes*, the smallest self-contained units of behavior. These nodes communicate using the publish-subscribe paradigm: nodes publish messages on *topics*, which broadcast the message to all nodes that are subscribed to the topic. Nodes react to incoming messages by activating *callbacks* to process each message. Since these callbacks may publish messages themselves, complex behavior can be implemented as a network of topics and callbacks. ROS also allows callbacks to invoke remote procedure calls by means of the *service mechanism* using a continuation-passing style. Specifically, a callback can initiate a non-blocking service request to a *service* callback and specify a third *client* callback to be invoked once the response is available.

ROS seamlessly allows composing nodes written in different programming languages and using different communication backends. The ROS implementation is therefore split into multiple layers of abstraction, which are visualized in Figure 1. Each supported programming language requires a client library that provides a language-specific API to the ROS application model. The ROS project officially supports C++ and Python, with community-provided support for numerous other languages. Below the surface, these libraries use a common system model provided by the *rcl* library. This ensures consistent behavior between the languages and reduces code duplication.

Despite this unified implementation, some parts of the ROS system are allowed to differ between languages. In particular, client libraries have a lot of freedom in implementing the execution model to allow the callback graph to be expressed in the most natural way in each language. A language supporting coroutines, for example, might allow coroutines as event handlers instead of callbacks. We therefore limit the focus of this paper to the C++ interface, which we believe to be the most likely choice for time-critical components.

For inter-node communication, ROS uses the Data Distribution Service [39] (DDS), an industry standard for data distribution in real-time systems. DDS specifies a network-transparent publish-subscribe mechanism that can be adapted to the needs at hand using a rich set of *Quality-of-Service* (QoS) policies. ROS works with different, independent implementations of the DDS standard (currently, FastRTPS by eProsima [2], Connext by RTI [6], and Vortex OpenSplice by Adlink [7]), each with a different API. The *rcl* client library therefore accesses the DDS subsystem over the common *rmw* (ROS MiddleWare) interface, which provides a DDS-agnostic API to the *rcl* layer. Each supported DDS implementation requires a dedicated *rmw* implementation, which translates between the common *rmw* interface and the vendor-specific DDS API.

To deploy a ROS application, the individual nodes have to be distributed to hosts and then mapped onto operating system processes. ROS does not impose any restrictions on this mapping. Processes implement the ROS execution model by running *executors*, which receive messages from *rcl* and invoke the corresponding callbacks. ROS provides two built-in executors: a sequential one that executes callbacks in a single thread, and a parallel one that distributes the processing of pending callbacks across multiple threads. Moreover, ROS supports arbitrarily complex setups of multiple, user-defined executors.

From a real-time perspective, it is important to note that executors implement custom scheduling policies; we revisit this issue in Section 3 in detail. Furthermore, how each ROS executor's threads are scheduled by the OS has a major impact on the overall timing behavior of the application. To ensure predictable scheduling, executor threads can be bound to a reservation server, which is a pragmatic configuration approach to increasing predictability that we advocate in this paper. Next, we briefly review key aspects of resource reservations.

2.2 Resource Reservations

An ideal mechanism to ensure predictable service for ROS threads is a *resource reservation*, which is a classic OS-level abstraction that limits interference between processes by bounding their resource consumption. Resource reservations are typically implemented by reservation servers. In general, a reservation server r_i is characterized by a budget Q_i and a period P_i, and guarantees that its client threads receive Q_i units of execution time in each period.

Many different reservation algorithms have been designed and developed over the last 30 years [12], with various different features and support for different scheduling algorithms (e.g., fixed-priority, deadline-based scheduling, etc.). This paper does not focus on any specific algorithm, but we assume the reservation server to comply with the periodic resource model [58], namely, we require that: **(i)** each reservation has an implicit deadline, **(ii)** whenever r_i has workload, the reservation algorithm guarantees at least Q_i units of service every P_i time units, and **(iii)** there exists a bounded maximum service delay, i.e., a bounded maximum release delay that a process running in a reservation can experience because of budget exhaustion and delays due to other reservations. Under these assumptions, the minimum amount of service provided by the reservation in an interval of length Δ can be expressed with a supply-bound function $sbf(\Delta)$ [33, 58]. In the following, we assume that such a supply-bound function is known for each reservation and refer to [12, 15, 33, 58] for a discussion of how to obtain them.

On Linux systems, resource reservations are available through the `SCHED_DEADLINE` [32] scheduling class, which implements the *Constant Bandwidth Server* [8] reservation algorithm. Moreover, as a special case of practical relevance, a thread running on a dedicated core at the highest priority of the `SCHED_FIFO` scheduling class can be considered as running in a reservation with the supply-bound function $sbf(\Delta) = \Delta$.

Next, to complete the overview of needed background, we review the Compositional Performance Analysis approach, upon which we base our response-time analysis.

2.3 Compositional Performance Analysis

Compositional Performance Analysis (CPA) is an approach to analytically evaluate the performance of heterogeneous and distributed systems [27]. CPA models systems as networks of resources, and workloads as tasks with dependencies. Resources provide processing time, which is consumed by tasks. Tasks with dependencies are organized as a direct acyclic graph, and paths in the graphs are denoted as *processing chains*. Tasks sharing the same resource are scheduled according to a resource-specific scheduling policy. The source task of a chain is triggered according to an *externally provided event arrival curve* [27, 30, 62] $\eta^e(\Delta)$, which defines an upper bound on the number of events that can arrive in any time window $[t, t+\Delta)$. Event arrival curves are general enough to model both periodic and event-driven (e.g., interrupt-driven) activations [47]. For example, when a task T_x is periodically triggered, its arrival curve can be expressed as $\eta^e_x(\Delta) = \left\lceil \frac{\Delta}{period(x)} \right\rceil$. Non-source tasks are triggered according to *derived activation curves*. Activation curves are obtained from arrival curves by accounting for release jitter, which reflects the activation delay due to predecessor tasks and depends on their response times. However, response times also depend on the release jitter, thus creating a cyclic dependency. To solve this problem, the analysis starts with an initial jitter of zero, and then iteratively applies the response-time analysis and updates all jitter bounds until convergence is achieved [42, 63]. Convergence is guaranteed (for non-overloaded systems) by the monotonic dependency between response time and jitter (the more jitter, the higher the response times, and vice versa). The basic CPA approach bounds the end-to-end latency of a chain with the sum of the individual response-time bounds of each task. Extensions have been subsequently designed to improve analysis precision, e.g., [54, 56].

Since ROS provides executors that dispatch callbacks in peculiar ways using custom scheduling policies, the existing CPA literature and tooling is not a perfect match for ROS. However, we liberally take inspiration from CPA to obtain a similarly flexible timing model that reflects the idiosyncrasies of ROS, which we introduce next.

3 ROS Scheduling

As described in Section 2.1, the ROS execution model multiplexes all callbacks associated with an executor onto one or more threads. The ROS C++ library provides its built-in executor in two variants: a single-threaded and a multi-threaded one. In this initial study of the ROS timing behavior, we focus exclusively on the simpler and more predictable single-threaded executor. The following description is based on a careful study of the ROS source code and documentation, and is to our knowledge the first comprehensive description of the scheduling behavior of ROS. To validate our observations, we conclude this section with an experiment that demonstrates and corroborates our findings on a concrete example.

The executor distinguishes four categories of callbacks: *timers*, which are triggered by system-level timers, *subscribers*, which are triggered by new messages on a subscribed topic, *services*, which are triggered by service requests, and *clients*, which are triggered by responses to service requests. The executor is responsible for taking messages from the input queues of the DDS layer (by interacting with the *rcl* layer) and executing the corresponding callback. Since it executes callbacks to completion, it is a non-preemptive scheduler. However, unlike most commonly studied schedulers, it does not always consider all ready tasks for execution. Instead, it bases its decisions on the *readySet*, a cached copy of the set of ready non-timer callbacks, which it updates in irregular, execution-dependent intervals. The algorithm is depicted in Figure 2, in which we assume $\mathcal{C}$ to be the set of all callbacks assigned to the executor, and $\mathcal{C}^{\text{tmr}}$, $\mathcal{C}^{\text{sub}}$, $\mathcal{C}^{\text{srv}}$, $\mathcal{C}^{\text{clt}}$ to be the subsets of $\mathcal{C}$ consisting only of timers, subscribers, services, and clients, respectively.

If the executor is idle, it updates its *readySet*. This is the only step in which the executor interacts with the underlying communication layer (i.e., *rmw*, via *rcl*). It then looks for a callback to execute by searching through the four callback categories (for efficiency, the

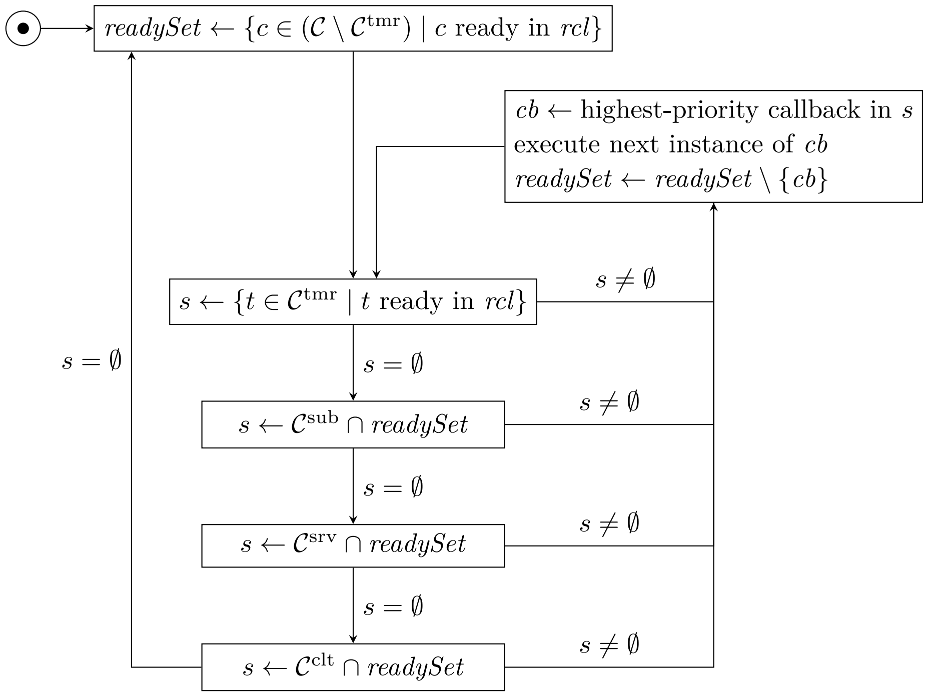


Figure 2 The executor scheduling algorithm.

executor blocks if there is nothing to do; this optimization has been omitted for clarity). It first checks whether any timers have expired. Since these are not managed by the DDS layer, this check is based on the current timer state and does not depend on the *readySet*. It then searches the *readySet* for subscriptions, services, and clients (in this order). Evaluating the queues in a fixed order has the intrinsic effect of assigning each queue a different priority (i.e., the timer queue is examined first and hence has the highest priority, and the client queue is examined last and has the lowest priority). When a queue is considered, callback instances are examined in callback registration order, i.e., the order in which the callbacks were registered with the executor. Consequently, the registration order represents a second level of priorities. Overall, the pair (callback type, registration time) is a unique priority for each callback.

Whenever a category has at least one ready callback, the highest-priority one is selected, executed, and then removed from the *readySet*. Finally, when the *readySet* is empty and no expired timers are left, the executor returns to the idle state and updates the *readySet* based on a current snapshot of the communication layer. We refer to the updating of the *readySet* as a *polling point* and the interval between two polling points as a *processing window*. The n-th polling point is referred to as PP_n, and the n-th processing window (ranging from PP_n to PP_{n+1}) as PW_n.

Compared to regular fixed-priority scheduling, this algorithm exhibits a few unusual properties. First, messages arriving during a processing window are not considered until the next polling point, which depends on *all* remaining callbacks. This leads to priority inversion, as lower-priority callbacks may implicitly block higher-priority callbacks by prolonging the current processing window.

Second, it relies on a ready *set* instead of the more usual ready *list*. This means that the algorithm cannot know how *many* instances of any non-timer callback are ready. It therefore

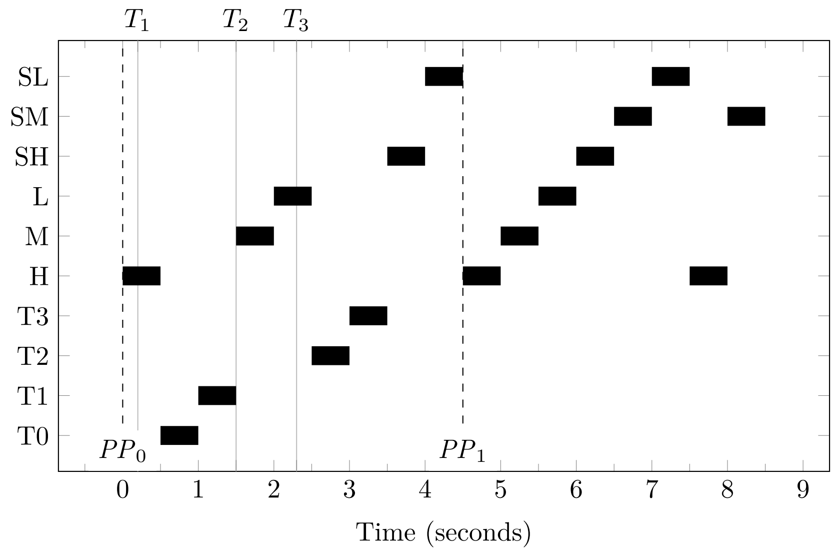


Figure 3 Gantt-Chart of the scheduler validation test. At times T_1 and T_3, the timers trigger. At time T_2, the second batch of service requests and messages is submitted.

processes at most one instance of any callback per processing window. This aggravates the priority inversion above, as a backlogged callback might have to wait for *multiple* processing windows until it is even considered for scheduling. Effectively, this means that a non-timer callback instance might be blocked by multiple instances of the same lower-priority callback.

The presented description of the ROS scheduler is based on manual code inspection. In a system as complex as ROS, however, this is potentially error-prone, as there might be subtle interactions that are easily overlooked yet change the behavior drastically. Thus, to validate our model, we implemented a special-purpose ROS node that executes arbitrary-length callbacks in a way that allows inferring the behavior of the ROS scheduler from the resulting trace. Specifically, the node is controlled using three topics (H, M, and L), three services (SH, SM, and SL), and a special-purpose topic to create timers. Note that the chosen names assume that topics and services are prioritized in registration order; checking that topic H actually has the highest priority is part of the model validation. In the following description, time zero refers to the point in time when the first batch of validation callbacks arrives at the node. The i-th timer is denoted as t_i. For ease of visualization, all callbacks run for 500ms.

Our test first sets up two timers at 200ms (T_0) and two timers at 2300ms (T_3). It then sends the message sequence $<L\ M\ H\ SH\ SL\ L\ M\ H\ SH\ SL>$, waits for 1.5 seconds (T_2), and then sends $<SM\ SM\ H>$. The result is visualized in Figure 3. Note that the polling points are not determined by the test; rather, they are inferred from the resulting timing behavior.

One can clearly observe the scheduler executing only a single callback per ready event, even if multiple messages have been queued up; this is especially apparent after the second polling point. Furthermore SM is visibly skipped at time 4, even though it arrives earlier at T_2 (i.e., during the execution of t_1). This proves the existence of polling points. The timers, however, are clearly not subject to these polling points, since t_2 and t_3 arrive later than SM but are still executed during the first processing window.

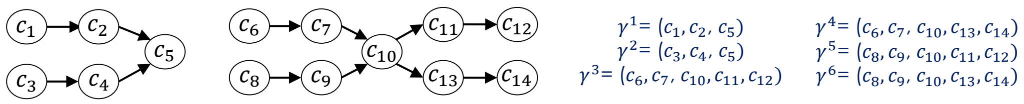


Figure 4 Example of a ROS graph. Circles represent callbacks, and edges represent communication relations among them. The corresponding processing chains are also shown.

4 System Model

In this section, we introduce a model of the timing-related aspects of a ROS system, its callbacks, and their activation relations. Table 1 summarizes our notation.

We model a ROS system as a direct acyclic graph (DAG) $\mathcal{D} = \{\mathcal{C}, \mathcal{E}\}$ composed of a set of callbacks $\mathcal{C} = \{c_1, \ldots, c_n\}$ and a set of directed edges $\mathcal{E} \subseteq \mathcal{C} \times \mathcal{C}$. We assume the graph $\mathcal{D}$ to be fixed, i.e., callbacks can neither join nor leave the system at runtime. Recall from Section 3 that $\mathcal{C}^{\text{tmr}}$, $\mathcal{C}^{\text{sub}}$, $\mathcal{C}^{\text{clt}}$, and $\mathcal{C}^{\text{srv}}$ denote the subsets of all timer, subscriber, client, and service callbacks, respectively.

Each callback $c_i \in \mathcal{C}$ has a worst-case execution time e_i, a unique priority π_i, and releases a potentially infinite sequence of instances. We assume a discrete-time model, that is, all time parameters are integer multiples of some basic time unit (e.g., a processor cycle).

Depending on its type, a callback instance is activated when the DDS layer receives a message or a timer expires. When an instance of a callback is activated, it is said to be *pending*, and it remains pending until it completes. A callback instance is said to be *ready* when it is pending but not executing. Each edge $(c_i, c_j) \in \mathcal{E}$ encodes an activation relation from callback c_i to callback c_j, meaning that during the execution of an instance of c_i it activates up to one instance of c_j (e.g., by publishing a message to the topic to which c_j is subscribed). Each callback is associated with a set of predecessors $\mathit{pred}(c_i) = \{c_j \in \mathcal{C} : \exists\ (c_j, c_i) \in \mathcal{E}\}$ and a set of successors $\mathit{succ}(c_i) = \{c_j \in \mathcal{C} : \exists\ (c_i, c_j) \in \mathcal{E}\}$. A callback without predecessors (respectively, successors) is said to be a *source callback* (respectively, *sink callback*).

Processing chains. The ROS graph $\mathcal{D}$ can have multiple source and sink callbacks. Each source originates one or more *callback chains* $\gamma^x = (c_s, \ldots, c_e)$, i.e., directed paths in the graph. The set of all chains of the graph from a source callback to any other callback is denoted by $\mathit{chains}(\mathcal{D}) = \{\gamma^1, \ldots, \gamma^s\}$. Callbacks can be shared by multiple chains. An example of a ROS graph with several chains is shown in Figure 4.

Activation model. As in CPA, each source callback c_s is associated with a given external event arrival curve $\eta_s^e(\Delta)$, denoting the maximum number of instances of c_s that can be released in any interval of length Δ, while non-source callbacks are associated with a (derived) activation curve. We assume w.l.o.g. that $\eta_s^e(\Delta) > 0$ for $\Delta > 0$.

As discussed in Section 2.1, non timer-callbacks are activated in a data-driven fashion. Our model assumes that callbacks belong to a single timer, topic, or service. (This is not a restriction, since two callbacks may execute the same code.) Consequently, a callback may have multiple incoming edges only if it subscribes to a topic with multiple publishers (similarly to what is referred to as `OR`-activation semantics in other work [27]). In this case, all subscribers are triggered once for each message published to the topic. The derivation of activation curves for callbacks with multiple incoming edges is discussed in Section 5.1.

Table 1 Summary of Notation

Symbol	Description	Symbol	Description
c_i	the i-th callback	γ^x	the x-th processing chain
r_k	the k-th reservation	$\gamma^{x,y}$	the y-th subchain of γ^x
$\mathcal{C}_k$	set of callbacks in reservation r_k	η_i^e	external arrival curve of c_i
$\delta_{i,j}$	propagation delay from c_i to c_j	η_i^a	derived activation curve of c_i
A	an offset into a busy window	$sbf_k(\Delta)$	supply-bound function of r_k
$R_i^*(A)$	least positive solution of c_i's response-time equation for offset A	$rbf_i(\Delta)$	request-bound function of c_i
		$RBF(C, \Delta)$	$\sum_{c_i \in C} rbf_i(\Delta)$

Scheduling of executors. As discussed in Section 3, this paper adopts the built-in single-threaded executor. To compel the OS to guarantee predictable service to ROS executors, we assume each executor (i.e., each thread) to be assigned to a single reservation server, and each reservation server to handle a single executor. Consequently, callbacks assigned to an executor can equivalently be considered as assigned to the corresponding reservation server.

The system comprises a set of w reservations $\mathcal{R} = \{r_1, \ldots, r_w\}$, and the set of all the callbacks assigned to reservation r_k is denoted $\mathcal{C}_k$. Analogously, the sets of all timers, subscriptions, clients and services allocated to reservation r_k are denoted as $\mathcal{C}_k^{\text{tmr}}$, $\mathcal{C}_k^{\text{sub}}$, $\mathcal{C}_k^{\text{clt}}$, and $\mathcal{C}_k^{\text{srv}}$, respectively. The symbols $lp_k(c_i)$ and $hp_k(c_i)$ denote the set of callbacks in $\mathcal{C}_k$ with lower and higher priority than π_i, respectively. The reservations are partitioned onto a set of m processors $\mathcal{P} = \{p_1, \ldots, p_m\}$, i.e., each reservation is statically assigned to a processor.

The results presented in this paper rely on the availability of a supply-bound function $sbf_k(\Delta)$ denoting the minimum service provided by a reservation r_k in any interval of length Δ (recall Section 2.2). Whenever multiple reservations are allocated to the same processor, the resource provisioning described by the supply-bound function is guaranteed only if all reservations are schedulable, i.e., they are always able to provide their complete budget during each period [33]. In this paper, reservations are assumed to be schedulable. The problem of guaranteeing reservations to meet their timing constraints is also referred to as *global schedulability* in prior works (e.g., in the context of hierarchical scheduling [9]). Many results are available to ensure global schedulability, e.g., [37].

Propagation delay. The propagation delay between the publication of a message by a sending callback and the activation of an associated receiving callback can be significant. Indeed, due to the inherently distributed topology of ROS systems, the message exchange can involve the network, introducing additional latencies. To model such a delay, each pair of reservations (r_x, r_y) is characterized by a (DDS-dependent) worst-case communication delay $\delta_{i,j}$, denoting the maximum time experienced by a message sent from the DDS layer of a sending callback c_i allocated to r_x until being received by the DDS layer of a receiving callback c_j allocated to r_y, i.e., the maximum additional delay experienced by c_j before being activated. When $r_x = r_y$, $\delta_{i,j}$ is assumed to be negligible. This delay can be either analytically upper-bounded for different types of networks (e.g., see [17, 19, 51]), or pragmatically measured, depending on the requirements of the target application domain.

Event sources. With the exception of timers, all callback types provided by ROS implement data-driven activation semantics. Consequently, all chains comprised solely of ROS callbacks are initially triggered by a timer. Nonetheless, applications often have to react to external events that are delivered asynchronously via interrupts (e.g., certain sensors, network packets delivering inputs from supervisory controllers or human operators, etc.). To integrate such events in our ROS model, we allow external threads to interact with ROS and model them as pseudo-callbacks. Specifically, we name these threads *event sources*. An event source is a regular OS-level thread that is sporadically activated, and interacts with ROS by publishing to one or more topics, thus acting as an interface or ingress point for external events. As we do in the case of executors, we assume each event source to be exclusively assigned to a dedicated reservation. For notational convenience, we let $\mathcal{C}^{\text{evt}}$ denote the set of all event sources and refer to event sources as callbacks $c_i \in \mathcal{C}^{\text{evt}}$.

5 Response-Time Analysis for Processing Chains

This section presents an analysis of the end-to-end delay (i.e., the maximum response time) of a generic ROS processing chain. Our analysis is inspired by the CPA approach (described in Section 2.3), whose event-propagation mechanism is a natural fit for the distributed and message-based nature of ROS. A discussion of possible alternatives is postponed to Section 7.

As in CPA, a complex ROS graph is analyzed by computing individual response-time upper bounds for each callback. End-to-end latencies can then be obtained by summing the individual response times of the callbacks of each chain [27]. Unfortunately, none of the existing instantiations of CPA can compute these per-callback response times, as they are unaware of the peculiarities of the ROS scheduling mechanism (e.g., polling points). We therefore present a ROS-specific response-time analysis for callbacks in Section 5.2.

Although this approach provides a safe and simple upper bound on end-to-end latencies, the resulting bounds may be overly pessimistic if arrival bursts of interfering callbacks are accounted for multiple times, once for each callback in the chain under analysis. This effect is known in the literature as the “pay-burst-only-once” problem [30, 56]. To improve the accuracy of our analysis, Section 5.4 presents a bound in which portions of chains, named *subchains*, are analyzed in a holistic way. We define the y-th subchain $\gamma^{x,y}$ of γ^x as a sequence of consecutive callbacks $c_i \in \gamma^x$ of the original chain that are allocated to a single reservation r_k, i.e., $c_i \in \gamma^{x,y} \Rightarrow c_i \in \mathcal{C}_k$. With this approach, arrival bursts are accounted for only once per subchain. The CPA approach can then be applied on a per-subchain basis, by propagating arrival curves and summing response-time bounds whenever a subchain crosses a reservation boundary, or joins with another chain in a callback with multiple predecessors.

5.1 High-Level Overview

Figure 5 shows an example that illustrates how the proposed analysis can be used to upper-bound the response time of a callback chain spanning multiple reservations. For clarity, interfering callbacks have been omitted in the figure. Response-time bounds for the various subchains of γ^x (i.e., $\gamma^{x,1} = (c_1, c_2)$, $\gamma^{x,2} = (c_3, c_4)$, $\gamma^{x,4} = (c_6, c_7)$, and $\gamma^{x,3} = (c_5)$) can be derived with the results that will be presented in Sections 5.2 and 5.4.

As discussed in Section 2.3, activation curves of non-source subchains must be derived from their predecessors and depend on both the response time of previous subchains and communication delays. In this example, $\eta^a_{x,2}(\Delta) = \eta^e_x(\Delta + R_{x,1} + \delta_{2,3})$, $\eta^a_{x,3}(\Delta) = \eta^a_{x,2}(\Delta + R_{x,2} + \delta_{4,5})$, $\eta^a_{x,4}(\Delta) = \eta^e_{x,3}(\Delta + R_{x,3} + \delta_{5,6})$, where $R_{x,y}$ is a response-time upper bound for $\gamma^{x,y}$. The response time of the chain shown in this example can then be computed as the sum of the response times of the subchains and communications delays, i.e., as $R_x = R_{x,1} + \delta_{2,3} + R_{x,2} + \delta_{4,5} + R_{x,3} + \delta_{5,6} + R_{x,4}$.

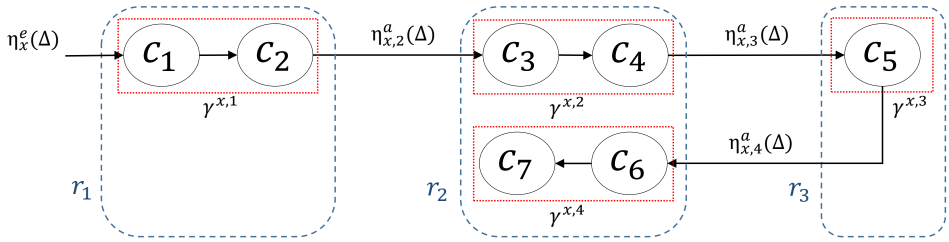


Figure 5 A callback chain crossing multiple reservations. Activations of the first subchain are bounded by an external event arrival curve, while subsequent subchains are characterized by an activation curve that depends on response times and communication delays in the previous subchains.

Processing chains sharing one or more callbacks are also supported by the analysis framework. To deal with this case, the jitter propagation approach is extended to callbacks with multiple incoming edges, i.e., multiple predecessors [29]. As discussed in Section 4, a callback with multiple incoming edges is triggered when it receives a message from *any* of its predecessors. Consequently, the activation curve of a callback (i.e., the source callback of a subchain, when the holistic approach of Section 5.4 is adopted) is derived from the activation curves of the predecessors as follows:

$$\eta_i^a(\Delta) = \sum_{c_j \in pred(c_i)} \eta_j^a(\Delta + R_j + \delta_{j,i}), \tag{1}$$

where R_j is the response time of c_j and $\delta_{j,i}$ is the propagation delay of messages from c_j to c_i. The sum in Equation (1) follows since *each* incoming message spawns a callback instance.

5.2 Analysis for Individual Callbacks

To start, we recall some classic definitions from uniprocessor schedulability analysis. A time t is a *quiet time* with respect to an executor r_k and a callback $c_i \in \mathcal{C}_k$ if there is no pending instance of any callback $c_j \in \mathcal{C}_k$ that can potentially interfere with c_i and that arrived prior to t. An interval $[t_1, t_2)$ is a *busy period* for an executor r_k and a callback c_i iff t_1 and t_2 are quiet times and there is no quiet time in between t_1 and t_2. The *response time* R_i of a callback c_i is defined as the maximum difference, over all possible instances, between the finishing time and the release time of the specific instance. For each callback c_i, the *request-bound function* $rbf_i(\Delta)$ is defined as the maximum amount of (cumulative) processor service required by callback instances released in an interval of length Δ, i.e., $rbf_i(\Delta) = \eta_i^a(\Delta) \cdot e_i$ [31]. Finally, we define the total request-bound function of a given set of callbacks $\mathcal{C}^*$ as $RBF(\mathcal{C}^*, \Delta) = \sum_{c_i \in \mathcal{C}^*} rbf_i(\Delta)$.

From a scheduling perspective, callbacks can be divided into three categories: event sources, timers, and *polling-point-based* (*pp-based*) callbacks. For convenience, in addition to the sets $\mathcal{C}_k^{\text{evt}}$ and $\mathcal{C}_k^{\text{tmr}}$ containing respectively the event sources and timers allocated to r_k, we define also the set $\mathcal{C}_k^{\text{pp}} = \mathcal{C}_k \setminus (\mathcal{C}_k^{\text{evt}} \cup \mathcal{C}_k^{\text{tmr}})$ of pp-based callbacks allocated to r_k. Event sources are the easiest to analyze since, as described in Section 4, each event source is exclusively allocated to a dedicated reservation. Building on the concept of the supply-bound function $sbf_k(\Delta)$, i.e., the minimum amount of service provided by reservation r_k in an interval of length Δ, Lemma 1 provides a response-time bound for event sources.

▶ **Lemma 1.** *If $A \geq 0$ is the time at which the instance of an event source callback $c_i \in \mathcal{C}_k^{\text{evt}}$ under analysis is released (relative to the beginning of the current busy period), and $R_i^*(A)$ is* *the least positive value that satisfies*

$$sbf_k(A + R_i^*(A)) = rbf_i(A + 1), \tag{2}$$

then $R_i = \max\{R_i^*(A) \mid A \geq 0\}$ *is a response-time bound for* c_i.

Proof. By assumption (cf. Section 4), if an event source c_i is allocated to a reservation r_k, no other callbacks are allocated to r_k. Consequently, each callback instance can suffer only self-interference from other instances of the same callback. The lemma follows since the amount of service provided by r_k in the interval $[0, A + R_i^*(A))$ is lower-bounded by $sbf_k(A + R_i^*(A))$ and the maximum amount of service required by instances of c_i released in the interval $[0, A]$ is bounded by $rbf_i(A + 1)$. ◀

Lemma 1 is not directly applicable, as it requires checking an unbounded number of possible release offsets A. To actually implement a response-time analysis, both an upper bound on the length of the analysis interval and a reduction of the number of release offsets that must be checked are needed; we revisit this issue in Section 5.3.

Next, we consider the response times of timers, which are proper callbacks and thus dispatched by ROS executors. As described in Section 3, timer scheduling is not subject to polling points. Nevertheless, since executors process callbacks non-preemptively, timers are subject to lower-priority blocking. Lemma 2 bounds the blocking experienced by a timer callback due to the lower-priority callbacks $c_j \in lp_k(c_i)$.

► **Lemma 2.** *A timer callback* $c_i \in \mathcal{C}_k$ *is blocked for at most* $B_i = \max\{e_j \mid c_j \in lp_k(c_i)\}$ *time units by lower-priority callbacks.*

Proof. First note that callbacks allocated to any reservation $r_o \neq r_k$ cannot block c_i since there is an independent executor in each reservation and, as explained in Section 3, timers are not subject to polling points. An instance of a timer callback $c_i \in \mathcal{C}_k$ can be released at time $t^* + 1$, where t^* is the time at which a lower-priority callback $c_j \in lp_k(c_i)$ started executing. Due to non-preemptive scheduling, c_i cannot start until c_j completes, i.e., after at most e_j time units. The lemma follows. ◀

With Lemma 2 in place, Lemma 3 upper-bounds the response time of timer callbacks.

► **Lemma 3.** *If* $A \geq 0$ *is the time at which the instance under analysis of a timer callback* $c_i \in \mathcal{C}_k^{\text{tmr}}$ *is released (relative to the beginning of the current busy period), and* $R_i^*(A)$ *is the least positive value that satisfies*

$$sbf_k(A + R_i^*(A)) = rbf_i(A + 1) + RBF(hp_k(c_i), A + R_i^*(A) - e_i + 1) + B_i, \tag{3}$$

then $R_i = \max\{R_i^*(A) \mid A \geq 0\}$ *is a response-time bound for* c_i.

Proof. By Lemma 2, the blocking due to lower-priority callbacks experienced by c_i is bounded by B_i. Due to the priority assignment presented in Section 3, every callback with a priority higher than a timer is itself a timer, i.e., $c_j \in hp_k(c_i) \Rightarrow c_j \in \mathcal{C}_k^{\text{tmr}}$. Due to non-preemptive scheduling, as soon a callback instance starts executing it cannot be interfered with by any other callback, i.e., higher-priority callbacks can interfere only in the interval $[0, A+R_i^*(A)-e_i]$. The lemma follows by noting that: **(i)** the interference from higher-priority callbacks is bounded by the total request-bound function, i.e., $RBF(hp_k(c_i), A+R_i^*(A)-e_i+1)$, **(ii)** the callback under analysis can suffer self-interference only from instances released in $[0, A]$, and **(iii)** the amount of service provided by r_k in the interval $[0, A + R_i^*(A))$ is lower-bounded by $sbf_k(A + R_i^*(A))$, i.e., the callback under analysis completes no later than when the guaranteed minimum service matches the maximum total demand. ◀

Again, we discuss how to use Lemma 3 in a practical response-time analysis in Section 5.3. Next, we consider pp-based callbacks. Due to the unpredictable nature of dynamic polling points, pp-based callbacks suffer additional blocking. Indeed, when an instance of a pp-based callback is released, it requires the completion of one or more processing windows before being executed. A response-time bound for pp-based callbacks is provided by Lemma 4, which is illustrated in Figure 6.

▶ **Lemma 4.** *If $A \geq 0$ is the time at which the instance of a pp-based callback $c_i \in \mathcal{C}_k^{\mathrm{pp}}$ under analysis is released (relative to the beginning of the current busy period), $X \geq 0$ is the difference between time $A + R_i^*(A) - e_i$ and the last polling point before time $A + R_i^*(A) - e_i$ (see Figure 6), and $R_i^*(A)$ is the least positive value that satisfies*

$$sbf_k(A + R_i^*(A)) = rbf_i(A+1) + RBF(\mathcal{C}_k^{\mathrm{oth}}, A + R_i^*(A) - e_i - X + 1) + RBF(\mathcal{C}_k^{\mathrm{tmr}}, A + R_i^*(A) - e_i + 1), \quad (4)$$

where $\mathcal{C}_k^{\mathrm{oth}} = \mathcal{C}_k \setminus (\mathcal{C}_k^{\mathrm{tmr}} \cup \{c_i\})$ is the set of the other non-timer callbacks allocated to r_k, then $R_i = \max\{R_i^*(A) \mid A \geq 0\}$ is a response-time bound for c_i.

Proof. Due to polling points, non-timer callbacks (both of higher and lower priority) can delay the callback under analysis only with instances that have arrived by the last polling point. Note that a polling point cannot occur while a callback is executing. Consequently, the polling point at time $A + R_i^*(A) - e_i - X$ is the last polling point before $A + R_i^*(A)$, and pp-based callbacks can delay the callback instance under analysis only with instances released in $[0, A + R_i^*(A) - e_i - X)$ (note that a callback released exactly at a polling point PP_n is processed during PW_n). Due to the priority assignment discussed in Section 3, each timer callback has higher priority than any pp-based callback. It follows that, due to non-preemptive scheduling, all timer callbacks $\mathcal{C}_k^{\mathrm{tmr}}$ can interfere with the pp-based callback under analysis up to the time at which it starts executing, i.e., at any time in $[0, A + R_i^*(A) - e_i]$. The lemma then follows analogously to Lemma 3 by noting that: **(i)** the callback under analysis can suffer self-interference only from instances released in $[0, A]$, and **(ii)** the amount of service provided by r_k in the interval $[0, A + R_i^*(A))$ is lower-bounded by $sbf_k(A + R_i^*(A))$. ◀

Lemma 4 upper-bounds the response time experienced by a pp-based callback. As for the previous lemmas, we will discuss how to bound the space of possible times A in Section 5.3. Moreover, Lemma 4 depends on the time distance X between $A + R_i^*(A) - e_i$ and the last polling point before $A + R_i^*(A)$, which is generally unknown during offline analysis. Consequently, we need to determine the scenario (i.e., the value of X) that maximizes the response time. Intuitively, this case occurs when the callback c_i under analysis starts executing just after the last polling point, i.e., lower-priority callbacks can interfere with c_i throughout the time from its release until it starts executing. In this case, $X = 0$. Lemma 5 proves that $X = 0$ indeed dominates all possible values of X.

▶ **Lemma 5.** *The delay experienced by a pp-based callback $c_i \in \mathcal{C}_k \setminus (\mathcal{C}_k^{\mathrm{tmr}} \cup \mathcal{C}_k^{\mathrm{evt}})$ due to other pp-based callbacks is maximized when c_i starts executing just after the last polling point:*

$$\max_{A \geq 0, X \geq 0} RBF(\mathcal{C}_k^{\mathrm{oth}}, A + R_i^*(A) - e_i - X + 1) = \max_{A \geq 0} RBF(\mathcal{C}_k^{\mathrm{oth}}, A + R_i^*(A) - e_i + 1), \quad (5)$$

where $R_i^*(A)$, A, and X are defined as in Lemma 5.

Proof. The lemma follows by noting that $X \geq 0$ and that $RBF(\mathcal{C}_k^{\mathrm{oth}}, A + R_i^*(A) - e_i - X + 1)$ is a sum of monotonic non-decreasing functions; hence it is monotonic non-decreasing, too. ◀

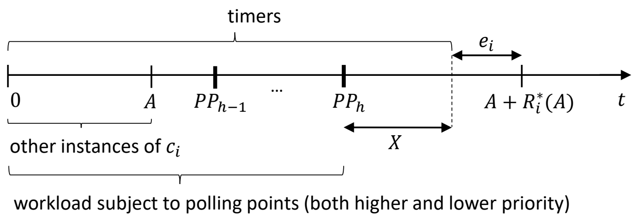


Figure 6 Time intervals in which other pp-based callbacks and timers can interfere with a pp-based callback c_i under analysis.

By Lemma 5, it follows that the amount of interference generated by timer callbacks $c_t \in \mathcal{C}_k^{\text{tmr}}$ and non-timer callbacks $c_n \in \mathcal{C}_k^{\text{oth}}$ is the same in the worst case. Consequently, we can merge the two sets, and rewrite Equation (4) in a simpler manner:

$$sbf_k(A + R_i^*(A)) = rbf_i(A+1) + RBF(\{\mathcal{C}_k \setminus c_i\}, A + R_i^*(A) - e_i + 1). \tag{6}$$

Equation (6) highlights that the scheduling policy adopted by the built-in ROS executor allows every other callback, independent of priority, to interfere with pp-based callbacks. Consequently, polling points make the priority assignment ineffective for upper-bounding the response time of pp-based callbacks. This confirms what we empirically observed during the model validation (Section 3) from an analytical perspective. Note that timer callbacks are not affected by polling points and their response-time bound is equivalent to non-preemptive fixed-priority scheduling [17], in the context of a resource reservation (Lemma 3).

5.3 Bounding the Search Space

The lemmas presented in Section 5.2 require checking Equations (3), (4) and (5) for all possible $A \geq 0$, where A represents the relative release time (with respect to the beginning of the current busy period) of the callback instance under analysis. To use the previous lemmas in a practical response-time analysis, both a bound on the analysis interval and a reduction of the search space size are required. Note that the analysis interval can be bounded by the longest interval during which a reservation r_k is busy serving higher-or-equal-priority workload, i.e., the length of the longest busy period [59], which Lemma 6 bounds.

▶ **Lemma 6.** *Let $\mathcal{C}_k^{\text{evt}}$, $\mathcal{C}_k^{\text{tmr}}$, and $\mathcal{C}_k^{\text{pp}}$ be the sets of all event source, timer, and pp-based callbacks allocated to r_k, respectively. If $c_i \in \mathcal{C}_k$ is the callback under analysis, and L^* is the least positive value that satisfies*

$$sbf_k(L^*) = \begin{cases} rbf_i(L^*) & \text{if } c_i \in \mathcal{C}_k^{\text{evt}} \\ RBF(hp_k(c_i), L^*) + B_i + rbf_i(L^*) & \text{if } c_i \in \mathcal{C}_k^{\text{tmr}} \\ RBF(\mathcal{C}_k, L^*) & \text{if } c_i \in \mathcal{C}_k^{\text{pp}}, \end{cases} \tag{7}$$

then L^* is an upper bound on the length of the longest busy period.

Proof. By contradiction, assume that there exist a busy period of c_i with length $L' > L^*$. Under this assumption, in the busy period corresponding to L' either **(i)** there are more callbacks delaying c_i, or **(ii)** callbacks execute for more time or, **(iii)** callbacks arrive more frequently than in the busy period corresponding to L^*. By Lemmas 1, 3, and 4, Equation (7) accounts for all callbacks that can delay c_i. Moreover, by definition of the request-bound function, Equation (7) is composed of a sum of products of worst-case execution times and arrival curves. By definition, no callback can execute for more than its worst-case execution time. Further, the activation curve $\eta^a(\Delta)$ defines an upper bound on the number of events that can arrive in any time window $[t, t+\Delta)$, thus leading to a contradiction. ◀

With Lemma 6 restricting the search to a finite interval, Lemma 7 below reduces the number of points contained in the search space. To this end, consider the response-time bounds computed with Equations (3), (4) and (6): each can be expressed as an instance of a *general response-time equation* $sbf_k(A+x) = rbf_i(A+1) + I(A+x) + B$, where B is a constant and the function I depends only on its argument. Equation (6), for example, can be written in this form by substituting $B = 0$ and $I(\Delta) = RBF(\{\mathcal{C}_k \setminus c_i\}, \Delta - e_i + 1)$. For any A, we let $SOL(A)$ denote the set of all positive x that satisfy the general response-time equation.

▶ **Lemma 7.** *For a callback $c_i \in \mathcal{C}_k$ under analysis, let $\mathcal{A}_i^- = \{A > 0 \mid rbf_i(A+1) = rbf_i(A)\}$ denote the points where $rbf_i(A)$ stays constant. For any $a \in \mathcal{A}_i^-$, $R_i^*(a) \neq \max_{A \geq 0} R_i^*(A)$.*

Proof. We prove that $R_i^*(a)$ is strictly less than its "neighbor" $R_i^*(a-1) \in SOL(a-1)$, and hence necessarily also less than $\max_{A \geq 0} R_i^*(A)$. To this end, we establish that $R_i^*(a-1) = R_i^*(a) + 1$ (which is well-defined since $0 \notin \mathcal{A}_i^-$) by showing that **(i)** $R_i^*(a) + 1 \in SOL(a-1)$ and **(ii)** $R_i^*(a) + 1 \leq a'$ for any $a' \in SOL(a-1)$.

Step (i): By definition, $R_i^*(a) \in SOL(a)$, and thus $sbf_k(a + R_i^*(a)) = rbf_i(a+1) + I(a + R_i^*(a)) + B$. By adding $0 = 1-1$ and using the fact that $a \in \mathcal{A}_i^-$ and hence $rbf_i(a+1) = rbf_i(a)$, we equivalently obtain $sbf_k(a - 1 + R_i^*(a) + 1) = rbf_i(a-1+1) + I(a - 1 + R_i^*(a) + 1) + B$. This is the definition of $SOL(a-1)$ and hence proves $R_i^*(a) + 1 \in SOL(a-1)$.

Step (ii): Consider any $a' \in SOL(a-1)$. Then $sbf_k((a-1)+a') = rbf_k(a) + I((a-1)+a') + B$. Using again that $rbf_i(a+1) = rbf_i(a)$, this is equivalent to $sbf_k(a+(a'-1)) = rbf_k(a+1) + I(a + (a'-1)) + B$, which matches the definition of $SOL(a)$ and hence $a'-1 \in SOL(a)$. By definition, $R_i^*(a) = \min\{x \mid x \in SOL(a)\}$, and thus we have $a' - 1 \geq R_i^*(a) \Leftrightarrow R_i^*(a) + 1 \leq a'$. ◀

Together, Lemmas 6 and 7 enable an efficient implementation of the response-time analysis by restricting the required search space $\mathcal{A}_i$ (w.r.t. a callback c_i) to

$$\mathcal{A}_i = \{A \mid 0 \leq A \leq L^*\} \setminus \mathcal{A}_i^- = \{0 \leq A \leq L^* \mid rbf_i(A+1) \neq rbf_i(A)\} \cup \{0\}.$$

To further reduce the effects of arrival bursts, we next provide a joint response-time bound for a sequence of callbacks in a single reservation.

5.4 Analysis for Processing Chains

This section provides an end-to-end analysis for linear subchains composed of multiple callbacks, where each subchain does not cross reservation boundaries. To this end, we extend the notion of request bound functions to subchains as $rbf^{x,y}(\Delta) = \eta_s^a(\Delta) \cdot e^{x,y}$, where c_s is the first callback of the subchain $\gamma^{x,y}$, and $e^{x,y} = \sum_{c_i \in \gamma^{x,y}} e_i$ is the cumulative worst-case execution time of the subchain. Consequently, $RBF^{\gamma}(\Gamma_k, \Delta) = \sum_{\forall \gamma^{x,y} \in \Gamma_k} rbf^{x,y}(\Delta)$, where Γ_k is the set of subchains allocated to r_k. For simplicity, the following analysis assumes that all subchains are disjoint, i.e., every callback is part of exactly one subchain (possibly consisting of only itself). This restriction is easy to lift at the expense of more involved notation. Lemma 8 allows us to compute a response-time bound for a subchain composed of multiple callbacks (if a subchain consists of only a single callback, its response time can be computed with the results of Section 5.2).[2]

[2] Addendum: compared to the conference version of this paper, Lemma 8 has been updated to explicitly depend on the offset A, analogously to the preceding lemmas.

▶ **Lemma 8.** *If $A \geq 0$ is the time at which an instance of the subchain $\gamma^{x,y} = (c_s, \dots, c_e)$ under analysis is released (relative to the beginning of the current busy period), $\gamma^{x,y}$ is composed of $|\gamma^{x,y}| \geq 2$ callbacks, $\gamma^{x,y'}$ is the prefix of $\gamma^{x,y}$ without c_e, Γ_k is the set of subchains allocated to r_k, and $R^*_{x,y}(A)$ is the least positive value that satisfies*

$$sbf_k(A + R^*_{x,y}(A)) = \eta^a_s(A+1) \cdot e_e + rbf^{x,y'}(A + R^*_{x,y}(A) - e_e + 1) + RBF^{\gamma}(\Gamma_k \setminus \gamma^{x,y}, A + R^*_{x,y}(A) - e_e + 1), \quad (8)$$

then $R_{x,y} = \max\{R^*_{x,y}(A) \mid A \geq 0\}$ is a response-time bound for $\gamma^{x,y}$.

Proof. Since $|\gamma^{x,y}| \geq 2$, the last callback in a subchain is a pp-based callback (timers and event sources are necessarily source callbacks). Due to polling points, every other callback can interfere with a pp-based callback (both of higher and lower priority) with instances that have arrived by the last polling point. The subchain under analysis can be delayed by **(1)** earlier instances of itself (i.e., the subchain under analysis), **(2)** instances of other subchains allocated to the same executor, and **(3)** due to supply unavailability. The response time of the subchain is maximized if the last callback in the instance under analysis executes for its full WCET. Under this condition, due to Lemma 5 and since a polling point cannot occur while a callback is executing, the last polling point occurs no later than at time $A + R^*_{x,y}(A) - e_e$. Delays due to cause (1) are composed of two mutually-exclusive contributions: **(1-a)** delays due to the last callback $c_e \in \gamma^{x,y}$ and **(1-b)** delays due to other callbacks $c_c \in \gamma^{x,y'}$ in the subchain under analysis. The total delay due to cause (1-a) is bounded by $\eta^a_s(A+1) \cdot e_e$ since the last callback instance under analysis $c_e \in \gamma^{x,y}$ can suffer self-interference only due to prior instances of c_e released no later than itself in the same busy period, i.e., during $[0, A]$. The total delay due to cause (1-b) is bounded by $rbf^{x,y'}(A + R^*_{x,y}(A) - e_e + 1)$ since other callbacks $c_c \in \gamma^{x,y'}$ can interfere with the subchain instance under analysis only if they are released in the same busy period and no later than by the last polling point, i.e, at any time in $[0, A + R^*_{x,y}(A) - e_e]$. Similarly, the total delay due to cause (2) is bounded by $RBF^{\gamma}(\Gamma_k \setminus \gamma^{x,y}, A + R^*_{x,y}(A) - e_e + 1)$, as again other subchains can interfere with the subchain instance under analysis only if they are released in the same busy period and no later than by the last polling point, i.e., at any time in $[0, A + R^*_{x,y}(A) - e_e]$. Finally, delays due to cause (3) are accounted for by the supply-bound function, which lower-bounds the service provided by r_k in the interval $[0, A + R^*_{x,y}(A))$. The lemma follows. ◀

Analogously to Lemma 6, the analysis interval (i.e., the maximum busy period) is upper-bounded by the fixed point $sbf_k(L^*) = RBF^{\gamma}(\Gamma_k, L^*)$ and discretized as in Lemma 7 considering the activation curve of the subchain under analysis.

Lemma 8 extends Lemma 4 for subchains.[2] Since arrival bursts of interfering callbacks are accounted only for once per subchain, analyzing a subchain holistically results in improved analysis accuracy for long subchains. As previously observed in Section 5.2, the presence of pp-based callbacks renders any priority assignment ineffective, and consequently callback priorities are not reflected in Equation (8).

5.5 Analysis Summary

The results presented in this section allows analyzing ROS systems under reservation-based scheduling. Specifically, Section 5.2 proposed a response-time analysis for single callbacks, and Section 5.4 extended it to subchains allocated to a single reservation. As discussed in Section 5.1, both approaches allow to compute a safe end-to-end latency for generic processing chains by propagating arrival curves and summing individual response-time

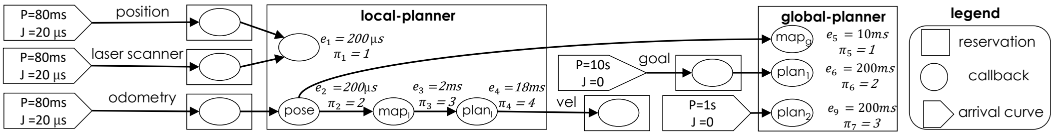


Figure 7 Callback graph of the event-driven `move_base` system.

bounds. Specifically, the effects of predecessor callbacks are accounted for as release jitter in the activation curves of non-source callbacks. Such release jitter depends on the response times of predecessors, but also response times depend on jitter in a circular manner. As in the CPA approach, this problem can be solved by iteratively searching for a global fixed point at which all jitter terms and response times are consistent.

6 Case Study

Our analysis seeks to enable ROS developers to easily and quickly try different designs and explore various what-if scenarios. To evaluate the suitability of our approach for that purpose, we analyzed a safety-critical processing chain in the popular `move_base` package, the central part of the ROS navigation stack for wheeled robots, using sensor rates and (observed) maximum execution times from a Bosch-internal case study. Since `move_base` has not been ported to ROS 2 yet, we model the ROS 1 version as if it ran on a ROS 2 system.[3]

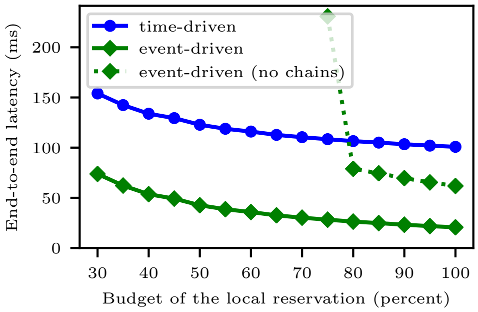
The `move_base` package addresses the *path planning* problem: given a map of the environment, first find a path to the goal location (*global planning*), and then control the robot's velocity to follow that path while avoiding obstacles (*local planning*). Both planners base their decision on internal maps, which reflect the component's knowledge of obstacles and properties of the environment. As the robot moves through the environment, these maps are continuously updated based on the most recent sensor data.

The `move_base` callback graph is illustrated in Figure 7. The incoming sensor and position data is normalized to absolute coordinates based on the robot's *pose* and then integrated into the respective maps. The local planner then updates its plan based on the new information.

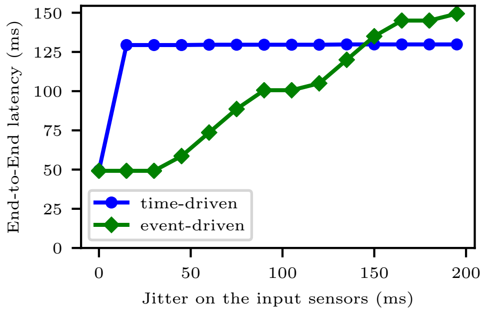
From a timing perspective, the execution time of the global planner stands out. Unlike the local planner, the global planner is difficult to predict and its execution time depends heavily on the path-finding difficulty. The global planner's map is also significantly larger and often updated only partially, further reducing predictability. In the Bosch case study, global planning times reached up to 200 ms, more than ten times the local planner's runtime. Fortunately, the global planning is not time-critical. At worst, computing the global plan too late may cause the robot to take a detour for a few moments. We therefore separate the global planning callbacks from the time-critical local planning callbacks using reservations. This not only isolates the more unpredictable components from the critical path, but also helps to limit the effects of the ROS executor scheduling policy.

Internally, the `move_base` subsystem is completely time-driven, using ROS topics only to communicate with other components. We configured the local planner to run in sync with the fixed sensor rate of 12.5 Hz, and the global planner to run more rarely at 1 Hz. While this setup makes for a quite predictable system, it is also very inflexible and makes it difficult to

[3] Addendum: compared to the conference version of this paper, the case study has been revised to reflect the change in Lemma 8 and to incorporate implementation improvements. The overall conclusions of the evaluation remain unchanged.



(a) Effects of reservation dimensioning on the critical path latency



(b) Effects of input jitter on the critical path latency, using a budget of 45%.

Figure 8 Experimental results

ensure that components do not consume stale data. For comparison, we therefore modeled two variants: the original, time-driven version, and an event-driven alternative design that models the activation dependency explicitly using internal topics.

In our case study, we are interested in the end-to-end latency of the path from the odometry input to the wheel command output (denoted “vel” for velocity). This latency determines, for example, how long the robot takes to react to an obstacle suddenly appearing in front of it, and hence is safety-relevant. In the event-driven setup, this is defined as the worst-case response time from activation to completion of the chain. For the time-driven setup, we compute the response time of the local planner and, if there is jitter on the sensor inputs, add the activation period as worst-case sampling delay. Although generally speaking the response time of the last chain element is not necessarily identical to the chain's overall latency, it happens to coincide here: by prioritizing the chain in decreasing order, and triggering all tasks at the same time, no task can run before its predecessor completes. The completion of the local planner thus implies the completion of the entire processing chain.

One of the most difficult steps in using reservation-based scheduling is dimensioning the reservations correctly. When isolating the local from the global planner, one would like to give the local planner just enough budget to complete in time, leaving as much execution time as possible for the global planner. To this end, we prototyped our analysis in the pyCPA framework [18].[3] Figure 8(a) shows the results for both the time-driven setup and the event-driven setup. In case of the event-driven setup, we also included the analysis results when disabling the whole-chain analysis described in Section 5.4. The graph shows the entire range of local planner budgets in percent of the total core bandwidth.

The graph clearly shows a similar effect of budgeting on the time-driven and event-driven system. However, due to the worst-case sampling delay, the time-driven system remains one sampling period of 80 ms above the event-driven latencies. Clearly, the event-driven approach is advisable in this setup, allowing the system to wait until the sensor results arrive instead of commencing planning based on stale values. One can also observe the beneficial effects of the whole-chain analysis; when disabled, the chain's self-interference significantly inflates the predicted response-time bounds. Since the analysis conservatively assumes that every callback is blocked by every other callback, actual interference is over-counted four-fold. As a result of this pessimism, the search for an upper bound on the maximum busy period length (Lemma 6) does not converge at lower bandwidths if the whole-chain analysis is disabled.

Another important property of any control system is how it copes with input jitter.

While the previous experiment modeled the sensors as strictly following the 12.5 Hz schedule with only 200 μs of jitter, Figure 8(b) shows the predicted end-to-end latency as input jitter increases, using a local planning budget of 45%. Here, one clearly observes the main benefits of purely time-driven systems; they are very robust to input jitter, mainly because they are not influenced by bursts. For the event-driven system, one can observe a significant rise between 35 ms and 80 ms, repeating every 80 ms. These are the regions where the analysis has to account for interference by one more instance of the chain released before the instance under analysis. The event-driven system remains superior below 150 ms of jitter, but succumbs to self-interference at larger jitters. Without a systematic analysis, such tradeoffs are extremely difficult to anticipate.

In conclusion, this case study highlights the benefits of automated response-time analysis. Without implementing a single line of ROS code, we are able to reason about the worst-case latencies of two quite different `move_base` designs, noting the advantages and disadvantages in different scenarios. Having a fully-integrated and automatic version of this analysis would clearly be a major aid to ROS developers, allowing response times to be treated as a measurable design constraint instead of relying solely on intuition, trial-and-error, or post-implementation experimentation. Extrapolating a bit further, it might even allow one day to reason about the latency of external dependencies, enabling the safe and easy reuse of well-tested components for time-critical purposes.

7 Related Work

The literature concerning the real-time aspects of ROS 2 processing chains is quite limited. To the best of our knowledge, this is the first paper modeling a ROS system from a real-time perspective and proposing a response-time analysis for ROS processing chains.

Most of the existing work on ROS targets ROS 1 systems, mainly conducting empirical performance measurement and proposing possible improvements. For instance, Saito et al. [53] proposed a priority-based message transmission algorithm for ROS 1 that allows publishers to send data to multiple subscribers according to their priorities, and a mechanism for synchronizing communications among multiple nodes running at different frequencies. Suzuki et al. [61] presented a mechanism for coordinating CPU and GPU execution of ROS 1 nodes, and an offline scheduling algorithm that assigns priorities to nodes according to their laxities. Maruyama et al. [36] conducted an experimental study aimed at comparing the performance of ROS 1 and a preliminary version of ROS 2 under different DDS implementations. Gutiérrez et al. [25] performed a similar evaluation of ROS 2 "Ardent Apalone" under Linux with the PREEMPT-RT patch. Concerning more general robotic systems, Lotz et al. [34, 35] presented a meta-model for designing non-functional aspects of robotics systems such that the resulting models can be analyzed with the SymTA/S timing analysis tool [27], which is based on the CPA approach [29, 46, 48, 49, 65].

Concerning the analysis of processing chains in distributed systems, one of the first proposals to verify end-to-end timing constraints is due to Fohler and Ramamritham [20], who proposed an approach for obtaining a static schedule composed of tasks with precedence constraints. In the context of non-static scheduling, prior work can be divided into two main categories: those based on CPA [27] and those adopting an holistic approach [42, 63]. The first method adopts arbitrary arrival curves [47] and analyzes chains crossing different nodes of a distributed system individually (by means of a *local component analysis*), and propagates the event model (i.e., activation curves) until convergence is achieved [50]. Different local analyses have been designed over years. For instance, Schlatow and Ernst [54, 55] proposed a local analysis for chains entirely contained in a single resource (e.g., a processing node) where tasks along the chain can have arbitrary priorities, under preemptive scheduling. Other authors [26, 28, 52, 56, 64] improved the analysis precision by accounting for correlations among events in different components. Previously, Thiele et al. [62] proposed Real-Time Calculus, an approach similar to CPA in which the service demand of the workload is modeled with arrival curves, and service curves model the processing capacity of local components. As in Network Calculus [30], arrival curves and services curve are combined together by means of a max-plus algebra, thereby obtaining the timing behavior of the component. Concerning the holistic approach, the seminal work is due to Tindell and Clark [63], who proposed a schedulability analysis for transactions, i.e., sporadically triggered sequences of events, scheduled under fixed-priority preemptive scheduling. Their analysis has been refined by Palencia et al. considering offsets [42] and precedence relations [41]. More details about the transactional task model can be found in a survey by Rahni et al. [44].

Only little attention has been given to date on how specific frameworks affect worst-case response times. To the best of our knowledge, all of them target the OpenMP framework [40], which is usually used for globally scheduled parallel tasks. For example, Serrano et al. [57] distinguished between tied and untied sub-tasks in OpenMP, proposing a response-time analysis for a parallel task composed of untied sub-tasks. While untied nodes have no particular scheduling restrictions, tied sub-tasks are OpenMP-specific and consist of a subgraph whose nodes must all execute on a single thread. Subsequently, Sun et al. [60] proposed an improvement of the OpenMP scheduling policy. To the best of our knowledge, the present paper is the first to systematically study the temporal behavior of ROS.

8 Limitations, Extensions, and Conclusions

This initial work on the timing analysis of ROS 2 processing chains can already handle practical components (such as `move_base`), and provides a rich foundation for future developments. Nonetheless, given the inevitable complexities associated with a mature, flexible, and widely used framework, we had to elide certain infrequently used aspects of ROS. In the following, we discuss these limitations and highlight promising direction for future extensions.

This paper considers the built-in single-threaded ROS executor. ROS also provides a multi-threaded variant of that executor, and additionally allows the definition of arbitrary special-purpose executors. Being able to easily integrate special-purpose schedulers tailored to specific robot needs would allow for interesting domain-specific research in the future.

When using multiple executor threads in a shared process, concurrency problems arise. ROS introduces *mutually-exclusive callback groups* to address this problem, and guarantees that callbacks in the same group are never executed concurrently. Extending our analysis to handle blocking relationships among callbacks remains future work.

This paper assumed the graph of the callbacks to be fixed. However, ROS allows nodes to dynamically join and leave, as well as to subscribe to and unsubscribe from topics dynamically at runtime, which is particularly useful for implementing different operating modes. This problem is referred to as *mode changes* in the literature [38, 45]. Our analysis can be applied to each mode in stable operation, but not does account for transient effects during mode changes. The design of new analysis techniques accounting for mode changes (e.g., extending [10, 11, 13] to ROS systems) represents another relevant future direction.

We modeled the overhead of network delays and the underlying DDS implementation as a single variable $\delta_{i,j}$, which allows for a safe and simple accounting for network-related delays in the overall response time by summing the communication delay every time the network is crossed. An opportunity for future improvements would be to integrate network analysis to eliminate the pessimism induced by the pay-burst-only-once problem when the network is crossed multiple times. Furthermore, a detailed study of available DDS implementations would allow for a more precise modeling of message processing overheads.

In addition to topics and services, ROS also provides a *waitable* callback type. This type is intended to implement more complex communication primitives like the long-running and high-level *actions* known from ROS 1 [1]. Since this mechanism was only introduced in the latest release, there are no known users of this mechanism as of now. It will be necessary to extend our analysis to these additional methods as and when they are adopted in ROS 2.

We assumed each callback to trigger an activation of all its successors at most once per execution. As a future improvement, we would like to extend the proposed analysis to allow a callback to trigger its successors only after having executed a predefined number of instances, or to trigger multiple instances of each successors in a single execution [23].

Our analysis based on the CPA approach allows to simply and efficiently analyze a real-world system, limiting the complexity by considering reservations individually. The analysis accuracy can be further improved by considering correlations among activation events in the chain, thus reducing the "pay-burst-only-once" problem also for chains spanning multiple reservations. A possible research direction for future work consists in extending the approaches presented by Fonseca et al. [21] and Casini et al. [14] (in the context of preemptive and non-preemptive fixed-priority scheduling of parallel tasks, respectively), based on which chains crossing multiple reservations could be modeled by means of self-suspending tasks [16]. In this way, arrival bursts can be considered only once per reservation, thus improving the analysis precision for chains crossing the same reservation multiple times.

To conclude, we have presented the first comprehensive scheduling model of ROS 2 systems, based on a review of its source code and documentation. We derived a response-time analysis for processing chains that takes the specific properties of the ROS framework into account and applied to a realistic case study. While there remain ample opportunities for future extensions, our contributions represent the first steps towards an automated analysis tool that could allow ROS users without expert knowledge in real-time systems to quickly and conveniently determine temporal safety and latency properties of their applications.

References

1. Action Lib. URL: `http://wiki.ros.org/actionlib`.
2. eProsima FastRTPS. URL: `https://www.eprosima.com/index.php/products-all/eprosima-fast-rtps`.
3. The `move_base` package. URL: `http://wiki.ros.org/move_base`.
4. Robots using ROS. URL: `http://robots.ros.org`.
5. ROS 2 "Crystal Clemmys". URL: `http://www.ros.org/news/2018/12/ros-2-crystal-clemmys-released.html`.
6. RTI Connext DDS. URL: `https://www.rti.com/products/connext-dds-professional`.
7. Vortex OpenSplice. URL: `http://www.prismtech.com/vortex/vortex-opensplice`.
8. L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, Madrid, Spain, December 2-4 1998.
9. A. Biondi, G. C. Buttazzo, and M. Bertogna. Schedulability analysis of hierarchical real-time systems under shared resources. *IEEE Transactions on Computers*, 65, May 2016.
10. A. Block and J. H. Anderson. Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms. In *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)*, Minneapolis, USA, July, 12-15 2006.

11 A. Block, J. H. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, Hong Kong, China, July, 17-19 2005.

12 G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*. Springer, 2011.

13 D. Casini, A. Biondi, and G. C. Buttazzo. Handling transients of dynamic real-time workload under EDF scheduling. *IEEE Transactions on Computers*, 2018.

14 D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2018.

15 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*, RTNS '17, 2017.

16 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, Sep 2018.

17 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*, 35(3):239–272, Apr 2007.

18 Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional performance analysis in python with pyCPA. In *In Proceedings of WATERS'12*, 2012.

19 Jonas Diemer, Jonas Rox, and Rolf Ernst. Modeling of Ethernet AVB networks for worst-case timing analysis. *IFAC Proceedings Volumes*, 2012. 7th Vienna International Conference on Mathematical Modelling.

20 G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *Proceedings 9th Euromicro Workshop on Real Time Systems*, Jun 1997.

21 J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016.

22 T. Foote. ROS community metrics report. URL: `http://download.ros.org/downloads/metrics/metrics-report-2018-07.pdf`.

23 J. J. G. Garcia, J. C. P. Gutierrez, and M. G. Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, 2000.

24 B. Gerkey. Why ROS 2.0? URL: `http://design.ros2.org/articles/why_ros2.html`.

25 C. Gutierrez, L. Juan, I. Uguarte, and V. Vilches. Towards a distributed and real-time framework for robotsç evaluation of ROS 2.0 communications for real-time robotics applications. Techical report, Erle Robotics S.L., 2018.

26 R. Henia and R. Ernst. Context-aware scheduling analysis of distributed systems with tree-shaped task-dependencies. In *Design, Automation and Test in Europe*, March 2005.

27 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.

28 K. Huang, L. Thiele, T. Stefanov, and E. Deprettere. Performance analysis of multimedia applications using correlated streams. In *2007 Design, Automation Test in Europe Conference Exhibition*, 2007.

29 M. Jersak. *Compositional Performance Analysis for Complex Embedded Applications*.

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

31 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*, 1989.

32 J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

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

34 A. Lotz, A. Hamann, R. Lange, C. Heinzemann, J. Staschulat, V. Kesel, D. Stampfer, M. Lutz, and C. Schlegel. Combining robotics component-based model-driven development with a model-based performance analysis. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Dec 2016.

35 Alex Lotz, Arne Hamann, Ingo Lütkebohle, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. Modeling non-functional application domain constraints for component-based robotics software systems. In *6th International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob'15)*, 2015.

36 Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software*, page 5. ACM, 2016.

37 L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: a new reclaiming algorithm for server-based real-time systems. In *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, May 2004.

38 M. Negrean, M. Neukirchner, S. Stein, S. Schliecker, and R. Ernst. Bounding mode change transition latencies for multi-mode real-time distributed applications. In *ETFA2011*, 2011.

39 Object Management Group. *Data Distribution Service (DDS)*, 1.4 edition, 2015.

40 OpenMP. *OpenMP Application Program Interface, Version 4.0.*, 2013.

41 J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Proceedings 20th IEEE Real-Time Systems Symposium*, 1999.

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

43 Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

44 Ahmed Rahni, Emmanuel Grolleau, Michaël Richard, and Pascal Richard. Feasibility analysis of real-time transactions. *Real-Time Syst.*, 48(3), 2012.

45 J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

46 K. Richter. *Compositional Scheduling Analysis Using Standard Event Models: The SymTA/S Approach*.

47 K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 506–513, 2002.

48 K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *Computer*, 36(4), 2003.

49 K. Richter, R. Racu, and R. Ernst. Scheduling analysis integration for heterogeneous multiprocessor SoC. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS)*, 2003.

50 K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceedings 2002 Design Automation Conference*, June 2002.

51 Jonas Rox and Rolf Ernst. Formal timing analysis of full duplex switched based ethernet network architectures. In *SAE Technical Paper*, 2010.

52 Jonas Rox and Rolf Ernst. Compositional performance analysis with improved analysis techniques for obtaining viable end-to-end latencies in distributed embedded systems. *Int. J. Softw. Tools Technol. Transf.*, 15(3), 2013.

53 Yukihiro Saito, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Priority and synchronization support for ROS. In *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 77–82. IEEE, 2016.

54 J. Schlatow and R. Ernst. Response-time analysis for task chains in communicating threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

55 Johannes Schlatow and Rolf Ernst. Response-time analysis for task chains with complex precedence and blocking relations. *ACM Trans. Embed. Comput. Syst.*, 16(5s), September 2017.

56 Simon Schliecker and Rolf Ernst. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, 2009.

57 M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones. Timing characterization of openmp4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

58 Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *24th IEEE Real-Time Systems Symposium*, 2003.

59 Marco Spuri. Analysis of deadline scheduled real-time systems. Technical report, RR-2772, INRIA, 1996.

60 J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi. Real-time scheduling and analysis of OpenMP task systems with tied tasks. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017.

61 Yuhei Suzuki, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Real-Time ROS extension on transparent CPU/GPU coordination mechanism. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 184–192. IEEE, 2018.

62 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings*, May 2000.

63 Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 2012.

64 Ernesto Wandeler and Lothar Thiele. Workload correlations in multi-processor hard real-time systems. *J. Comput. Syst. Sci.*, 73(2), 2007.

65 D. Ziegenbein, M. Jersak, K. Richter, and R. Ernst. Breaking down complexity for reliable system-level timing validation. In *Ninth IEEE/DATC Electronic Design Processes Workshop (EDP'02)*, 2002.