Real-Time Execution Management in the ROS 2 Framework

Tobias Stark (né Blaß)

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr. Ing.)
an der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

Saarbrücken, 2022

Dekan:
Univ.-Prof. Dr. Jürgen Steimle

Prüfungsausschuss:
Prof. Dr. Jan Reineke (Vorsitz)
Dr. Björn Brandenburg (Berichterstatter)
Prof. Dr. Martina Maggio (Berichterstatter)
Prof. Dr.-Ing. Rolf Ernst (Berichterstatter)
Dr. Hadar Frenkel (Beisitzer)

Tag des Kolloquiums: 04.07.2023

Textfassung vom 02.08.2023

Abstract

Over the past decade, the ROS ecosystem has emerged as the most popular repository of open-source robotics software. As a result, many robots rely on ROS-based software to make timing-critical decisions in real time. However, there is little evidence that real-time theory is used to analytically bound or control the worst-case response time in ROS components. The recent transition to ROS 2, the next generation of the ROS framework, has not changed the situation, even though ROS 2 explicitly aims to improve support for real-time control [52].

This dissertation identifies three main hurdles to adopt real-time theory in the context of ROS 2: first, the complex and non-obvious timing effects introduced by the ROS 2 framework; second, the expertise required to use real-time scheduling mechanisms correctly; and third, the inherent unpredictability of typical robotics workloads, which defy static provisioning.

These hurdles are overcome in two steps. First, the dissertation introduces a timing model for ROS 2 applications that accounts for the framework's implicit timing effects. Based on this model, a response-time analysis is developed that allows robotics developers to bound the worst-case response time of individual components and multi-component processing chains.

However, modeling and provisioning ROS 2 systems remains a cumbersome and error-prone task. In a second step, the dissertation hence proposes *ROS-Llama*, an *automatic latency manager* for ROS 2. ROS-Llama automatically controls the latency of a ROS 2 system through real-time scheduling, while requiring only little effort and no real-time scheduling expertise by the user. It runs in parallel with the deployed application and can therefore measure all required information without user involvement and adapt to changes at runtime. As part of ROS-Llama's design, the dissertation discusses the conceptual and practical challenges in developing such an automatic tool, identifying relevant properties of ROS 2 and essential requirements of the robotics domain.

Experiments on a mobile robot demonstrate the effectiveness of the timing model and the response-time analysis in real-world settings. They further confirm the viability of the ROS-Llama approach and show that ROS-Llama reduces the maximum observed latency under load compared to the default Linux scheduler.

Kurzzusammenfassung

In den letzten zehn Jahren hat sich ROS und das ROS-Software-Ökosystem zur populärsten Quelle von Open-Source Robotik-Software entwickelt. Viele Roboter verwenden daher ROS-basierte Softwarekomponenten um zeitkritische Entscheidungen zu treffen. Trotzdem gibt es kaum Indizien, dass die Reaktionszeit solcher Komponenten durch Echtzeittheorie bestimmt oder kontrolliert wird. Die Migration zur nächsten Generation des Frameworks, ROS 2, hat an dieser Tatsache nichts geändert, obwohl ROS 2 mit dem expliziten Ziel entwickelt wurde, Echtzeitsoftware besser zu unterstützen [52].

In dieser Dissertation werden drei Hürden identifiziert, die der Anwendung von Echtzeittheorie auf ROS-Systeme im Wege stehen. Erstens, die komplexen und versteckten Effekte, die das Framework selbst auf die Laufzeit von ROS-Komponenten hat; zweitens, die Expertise die ein Entwickler bräuchte, um Echtzeit-Schedulingverfahren korrekt zu verwenden; und drittens, die inhärente Unvorhersehbarkeit und Dynamik typischer Robotik-Anwendungen.

Diese Hürden werden in zwei Schritten addressiert: zunächst definiert die Dissertation ein Laufzeitmodell für ROS 2-Anwendungen, das die impliziten Effekte des Frameworks beschreibt. Basierend auf diesem Modell definiert die Dissertation eine Antwortzeitanalyse (*response-time analysis*), mithilfe derer Entwickler eine obere Schranke auf die maximale Reaktionszeit bestimmen können. Die Analyse kann sowohl auf einzelne Komponenten als auch auf komponentenübergreifende Reaktionsketten angewendet werden.

Trotz dieser Verbesserungen bleibt es schwierig und fehleranfällig, ROS 2-Systeme korrekt zu modellieren und die Komponenten sinnvoll mit Ressourcen zu versehen. Dieses Problem wird mit der Entwicklung von ROS-Llama addressiert, einem automatischen Latenzmanager für ROS 2. ROS-Llama kontrolliert vollautomatisch die Latenz zeitkritischer Komponenten und Reaktionsketten. Es konfiguriert den Echtzeitscheduler des Betriebssystems und teilt den Komponenten Rechenzeit zu, ohne sich auf manuelle Interventionen oder sonstige Expertise des Benutzers zu verlassen. Dazu läuft es parallel zur verwalteten Anwendung und misst die benötigten Informationen zur Laufzeit. Mithilfe der gemessenen Parameter passt sich ROS-Llama automatisch an dynamische Änderungen im System an. Als Teil der Diskussion zu ROS-Llama werden die konzeptuellen und praktischen Herausforderungen eines solchen voll-automatisierten Werkzeugs diskutiert. Dabei werden relevante Eigenschaften von ROS 2 sowie die wichtigsten Anforderungen auf Seite der Robotik-Entwickler identifiziert.

Experimente an einem selbstfahrenden Roboter demonstrieren die Effektivität des Laufzeitmodells und der Antwortzeitanalyse in realistischen Situationen. Sie bestätigen außerdem die praktische Machbarkeit des ROS-Llama-Ansatzes und zeigen, dass ROS-Llama die beobachtete Latenz unter Last besser reduzieren kann als der in Linux mitgelieferte Standard-Scheduler.

Acknowledgements

First and foremost, I want to thank my academic advisor, Björn B. Brandenburg, for teaching and guiding me on the path toward this PhD. I learned a lot over the years and am highly grateful for your patience and time.

Second, I thank Robert Bosch GmbH for funding my PhD and giving me the opportunity to do my research in an industrial and applied context. Most of all, my thanks go to my group leader Dirk Ziegenbein and my advisors, Arne Hamann and Ralph Lange. You always managed to carve time for me out of your very busy schedules, and your willingness to share your experience and perspectives with me helped me out a great deal during this PhD.

Third, I would like to thank Rolf Ernst and Martina Maggio for reviewing this dissertation.

My PhD would not have been the same if it weren't for my colleagues at Bosch Corporate Research. I particularly want to thank Ingo Lütkebohle and Jan Staschulat for helping me to understand robotics software engineering better.

Thanks also to my fellow students at Max Planck Institute, particularly my office mate Felix and the members of the real-time systems group: Arpan, Cédric, Georg, Marco, Marco, and Sergey. I especially want to thank Manohar Vanga for sharing his experience and always having an open ear for my concerns.

Further thanks go to Daniel Casini for being an experienced and highly effective co-author over these years. Your hands-on teaching on how to turn a research idea into a finished paper has helped me a great deal.

Special thanks also go to Sergey Bozhko, Tina Jung, Marco Perronet, and Kathrin Stark for reading earlier drafts of this thesis. Your advice has greatly improved this work.

Further, a big thanks to my family and friends. Thank you for pub quizzes, Staden barbecues, board game nights, vacations; in short: countless hours of fun, both online and in person. Thank you for supporting me through all these years.

Finally, I want to thank my wife, Kathrin, for her never-ending love and support. You've cheered me up when I was down and pushed me forward when I was tempted to give up, despite the ocean between us.

Publications

Parts of this dissertation have previously appeared in the following publications:

[27] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B. Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 6:1–6:23, 2019

[14] 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)*, pages 264–277, 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 *Proceedings of the 42nd Real-time Systems Symposium (RTSS)*, pages 41–53, 2021

Contents

List of Figures

1 Introduction

Robots continuously interact with their environment. The correctness of a robot's control software therefore depends not only on whether the robot makes the *right* decision but also on whether it makes the decision *quickly enough*. Such a timing constraint is known as a *real-time constraint*.

One example of a common computation with real-time constraints is obstacle detection, an essential part of any mobile robot. When a robot perceives an obstacle in its path, it has limited time to identify the obstacle, plan a new course, and put the plan into action. Taking too long to compute the plan may lead to unwanted behavior like jerky movements, frequent emergency braking, or even collisions.

A similar situation occurs in the control of robotic arms [109, pp. 153 ff.]. To pick up an object, the robot needs to first locate the target and plan a suitable arm movement. While the movement is ongoing, the robot needs to continuously monitor the situation and react to any contingencies such as imprecise arm movements or changes in the environment. Failure to correct for these changes in time will leave the arm in the wrong position or force sudden and jerky corrections.

Real-time requirements are hardly unique to robots, and methods to control and predict worst-case software latency have been the subject of real-time systems research for about fifty years [71]. The required real-time schedulers are widely available, not only in special-purpose real-time operating systems but also in general-purpose operating systems like Linux. In principle, the tools to ensure that robots adhere to their real-time requirements are readily available.

Unfortunately, those tools turn out to be extremely difficult to apply in the robotics context. A major reason is that many robots are developed with frameworks such as *ROS 2* to simplify

and speed up development. ROS 2 abstracts away tedious low-level details and provides a portable high-level API that is designed with common robotics issues in mind. Most importantly, it provides a module system that gives developers access to a large ecosystem of third-party packages with ready-made solutions to many robotics problems. The advantages are numerous and easy to see. For instance, why painstakingly develop a new navigation subsystem if a complete navigation stack with multiple state-of-the-art path-planning algorithms and 3-D visualization support is just one download away?

Unfortunately, many of the design decisions that make robotics frameworks so useful also make it difficult to apply established real-time techniques. By abstracting from low-level details, frameworks complicate and obfuscate the timing behavior of the application. For example, ROS 2 multiplexes independent message handlers onto shared executor threads using custom scheduling policies. Consequently, applications running on top of the framework are subject to the scheduling decisions of the underlying operating system *and* the framework, with complex and interdependent effects on timing.

In this dissertation, we bridge this gap and make established real-time techniques applicable to ROS 2 systems. We describe the timing behavior of the ROS 2 framework and develop a *response-time analysis* that computes safe upper bounds on software latency in the system.

Although such techniques make it *possible* to control and bound software latency, applying them requires significant expertise and effort. The extensive modeling effort required by traditional real-time techniques proves particularly onerous once third-party modules are involved.

To address these issues, we propose ROS-Llama, an *automatic latency manager* for ROS 2 that automatically enforces real-time constraints based on a simple specification of latency goals, which can be written with little effort and expertise. ROS-Llama autonomously extracts a system model at runtime and uses it to configure the system's real-time scheduler such that the user's latency goals are fulfilled.

1.1 Why ROS 2?

Among the various robotics frameworks available today (surveyed in Section 2.4), this dissertation focuses exclusively on ROS 2. There are two major reasons for this decision: its popularity, and its explicit goal to support timing-sensitive workloads.

Popularity. First released in 2007, ROS 1 [97] has become extremely popular in academia and industry. The ROS website lists over 150 different robots that have been built with ROS [49], the community metrics indicate tens of thousands of developers worldwide [43], and the initial paper [97] has been cited over 8,850 times according to Google Scholar.

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 backward-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.

In the four years since its first release, ROS 2 has firmly established itself as the future of ROS. It is backed by many major robotics companies [47] and Open Robotics, the organization maintaining the ROS framework, announced in 2020 that they will cease development of ROS 1 and focus their attention on ROS 2 exclusively [48].

Commitment to real-time support. One of the major improvements of ROS 2 compared to ROS 1 is improved support for real-time control [52]. Toward this goal, the core ROS libraries allocate memory only through user-provided allocators and avoid memory allocations altogether on critical paths. This improves determinism, as memory allocations are a major source of unpredictable delays.

ROS 2 further implements node communication through the real-time-capable DDS [84] middleware. Although none of these implementation changes guarantees timing predictability on its own, they are all necessary steps towards enabling users to write real-time software with ROS.

In this dissertation, we therefore do not consider ROS 1 and focus on ROS 2 only. Unless specified otherwise, we use the term ROS hereafter to refer to ROS 2 version "*Foxy Fitzroy*", released in June 2020, which is the long-term support version at the time of writing.

The decision to focus exclusively on the ROS framework might seem surprising considering the various challenges and obstacles described so far, and even more so as various additional challenges become apparent throughout this dissertation. Would it not be easier to give up on compatibility with mainstream ROS and extend it with well-known techniques from real-time systems research? Would it not be more effective to build a new framework that is explicitly designed for easy and reliable real-time support?

We believe that breaking compatibility with ROS is not an effective way to bring improved real-time support to robotics software. Even though more predictable and real-time friendly robotics frameworks are freely available, for example Orocos [22] and Fawkes [82], and even though many of those frameworks are older than ROS, none of them has ever found the overwhelming popularity that ROS enjoys. We consider this strong evidence that the robotics community is unwilling to switch away from ROS just for the promise of better real-time support. In this dissertation, we therefore take the existing ROS design and implementation as a given and investigate ways to apply real-time systems principles within the constraints of the ROS framework.

1.2 Contributions

This dissertation makes three main contributions:

A timing model of ROS applications. As already alluded to, ROS hides various details on how ROS applications are executed behind an abstraction layer. It encourages an event-driven design style, which gives rise to data dependencies and potentially long processing chains. The event handlers, called *callbacks*, are multiplexed onto shared threads using custom scheduling

policies. 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, many of which are developed by independent developers all around the world.

In this dissertation, we provide a detailed description of the timing behavior and the callback scheduling policies of the ROS implementation. To ensure the correctness of these findings, we provide an automated model validator that confirms our assessment.

Based on the description of the ROS timing behavior, we then define a *real-time task model* of ROS applications. The model represents the application as a graph of communicating callbacks. As the evaluation shows, the model is expressive enough to represent real-world ROS packages, but also simple enough to allow for response-time analysis.

A response-time analysis for ROS. The complex interactions between ROS and the operating system, the event-driven development style, and the extensive use of third-party components make it difficult for developers to predict the timing behavior of their application.

In this dissertation, we develop a *response-time analysis* that computes a safe upper bound on the worst-case delay between activation and completion of individual callbacks or entire chains of callbacks. The analysis automatically accounts for the properties and quirks of the ROS implementation. Since it is defined using the *supply bound* abstraction [118], it is compatible with common real-time schedulers, including all real-time schedulers supported in Linux.

The analysis combines a traditional busy-window analysis (*e.g.*, [16, 21, 58, 61, 86, 87, 105], to name a few examples) with a novel analysis approach that relies on the round-robin-like behavior of the callback scheduler. It further contains an optimized analysis for communication within the same thread that reduces the pessimism for intra-executor callback chains.

An automatic latency manager for ROS. The specific requirements of ROS in particular, and the robotics domain in general, make it difficult or even impossible to apply ahead-of-time response-time analysis to ROS systems in practice.

In this dissertation, we identify nine requirements that a practical solution to the ROS latency

control problem needs to fulfill. Briefly, a solution needs to work with unmodified ROS workloads; it must be able to intentionally provision the system below worst-case requirements and ensure controlled degradation if system resources turn out to be insufficient; and finally, it must require little expertise and effort from the user.

To address these requirements we propose ROS-Llama, the ROS **l**ive **la**tency **ma**nager. ROS-Llama allows users to control the worst-case latency of their ROS system based on a simple, declarative configuration that can be specified with little effort and without real-time expertise.

ROS-Llama runs alongside the managed system and periodically extracts a ROS timing model from the running application. The extracted model is then used to identify a suitable scheduler configuration, and to validate that the configuration ensures the user's latency goals. An evaluation on a ROS robot using popular ROS packages confirms that ROS-Llama controls the timeliness of the evaluation workload better than comparable approaches.

1.3 Organization

The remainder of this dissertation is organized as follows: Chapter 2 reviews the background on ROS, real-time scheduling, and timing analysis, followed by a discussion of related work.

Chapter 3 then describes the timing behavior of the ROS framework and defines the real-time model of ROS applications. This model is used in Chapter 4 to define a response-time analysis.

Chapter 5 presents the automatic latency manager ROS-Llama. It first discusses the case for automatic latency management and lists the requirements an automatic latency manager must fulfill. It then describes the timing model extractor, the scheduling strategy, and the budget manager that computes the scheduling parameters.

Chapter 6 reports the results of our evaluation, which uses both synthetic and real workloads to judge the practical effectiveness and applicability of our contributions. Finally, Chapter 7 summarizes the dissertation and discusses potential future work.

2 Background

This chapter first reviews some essential background, covering the ROS framework (Section 2.1), real-time scheduling (Section 2.2), and response-time analysis (Section 2.3). Finally, Section 2.4 discusses related work and gives a historical perspective.

2.1 The ROS Framework

A typical ROS system comprises multiple small and self-contained modules called *nodes*. Similar to an object in an object-oriented programming language, a node encapsulates an opaque inner implementation through a compact external interface. In ROS, this external interface is built on top of three primitives: *topics*, *services*, and *timers*.

Topics follow the *publish-subscribe* paradigm [85]: a node that seeks to share a piece of information with the rest of the system *publishes* a message to a topic. The message is then broadcasted to all nodes *subscribed* to that topic. Each topic is identified by a unique name. For example, consider a node that estimates the robot's current position and orientation (also known as the robot's *pose*). The node publishes each new estimate on the `/pose` topic. Any other node that needs to know the robot's pose then subscribes to the `/pose` topic and is notified whenever a new pose estimate is available.

The main advantage of the publish-subscribe paradigm is that it decouples producers and consumers. Both producers and consumers can be added, removed, or replaced without any changes to the rest of the system.

Services follow the *remote procedure call* paradigm: one node *requests* a service from another

node and receives a reply when the request completes. As an example, a map management node might offer a service called `/load_map` that loads a prerecorded map from disk and reports the success or failure of the operation to the caller.

Finally, **timers** allow a node to trigger itself periodically. For example, a node interacting with a hardware sensor might use a timer to periodically poll an external sensor.

On top of the three basic primitives, ROS provides standardized interfaces for common use cases. These include standardized *parameter management*, support for long-running operations through *action servers*, and standardized *lifecycle management*.

ROS parameters are a standardized way to expose a node's configuration parameters to the rest of the ROS system. Nodes participating in this mechanism declare their parameters during initialization. The built-in parameters library then automatically creates a set of parameter-related services. Other nodes can use these services to query and update parameters, for example as part of an interactive configuration tool or a central configuration management system.

Action servers are a kind of remote procedure call that is optimized for long-running operations. Like a regular service, an action server is invoked by some node and returns a reply when it completes. In addition, the server can provide status updates or other intermediate results to their callers through a feedback topic. For example, a navigation component might provide a `/navigate_to_goal` action. As the robot moves, the navigation action can use the feedback topic to regularly notify its caller about the estimated distance and time to arrival. Furthermore, action servers provide a way to cancel or preempt an action before it is complete. Internally, action servers are implemented as a set of topics, timers, and services.

Finally, **lifecycle management** is an optional extension of regular ROS nodes. It enables a more structured and synchronized initialization- and reconfiguration process. A so-called *lifecycle node* can be in one of four states: *unconfigured*, *inactive*, *active*, or *shut-down*. Other nodes, for example a central orchestration node, use a per-node ROS service to order state transitions.

As these three examples show, the ROS communication primitives are simple but versatile tools that can be used to build complex and expressive interfaces. While this dissertation considers

only the three ROS primitives (services, topics, and timers), it is worth remembering that this covers not only explicit uses of the ROS communication primitives but also *implicit* uses like the ROS standard interfaces discussed above.

2.1.1 Execution Model

ROS nodes do not have an explicit control flow. Instead they provide *callback functions* (or *callbacks* for short) that are invoked whenever an event of interest occurs. For example, each subscription is associated with a *subscription* callback, which is invoked whenever a new message is published to the subscribed topic. Each service is associated with a *service handler* callback, which is triggered on each service request. Even waiting for service replies is facilitated through callbacks: the caller provides a *client* callback as part of the service request, which is triggered asynchronously when the reply arrives.

The callbacks are executed by an *executor thread* (or *executor* for short). The executor is responsible for monitoring the timers and communication primitives of its assigned nodes. Once an event of interest occurs, the executor runs the callback assigned to this event. This event-handling loop is referred to as *spinning*. An executor thread may be responsible for more than one node, but each node belongs to only one executor.

Due to this event-based design, the callback execution order is determined by the executor, not the developer. The executor's implementation thus significantly impacts the timing behavior of the system. Out of the box, ROS provides two executors: a single-threaded implementation and a multi-threaded implementation. Furthermore, ROS allows users to implement custom executors. In this dissertation, we consider only the single-threaded executor, which we discuss in more detail in Section 3.1.

Although nodes are the preferred way to implement ROS modules, it is sometimes more convenient to keep control of the execution flow. A blocking read from a device file, for example, cannot be implemented as a ROS callback without blocking the entire executor indefinitely. ROS therefore allows arbitrary threads to interact with the ROS communication facilities, albeit in

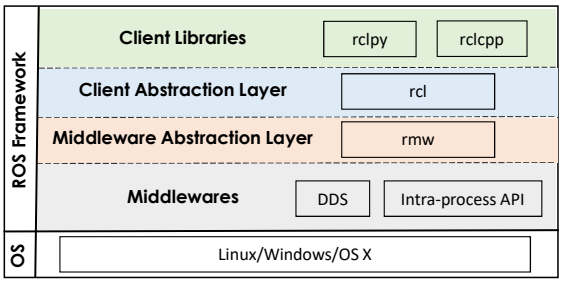


Figure 2.1: The multi-layered ROS architecture.

a limited way. Specifically, non-executor threads can publish to topics but cannot receive any messages themselves.

2.1.2 Implementation

The ROS implementation is split into multiple layers of abstraction, which are visualized in Figure 2.1. The top layer contains the API for ROS users. Because ROS supports nodes implemented in different programming languages, each supported programming language requires a separate language-specific API. 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, which allows them to represent the callback mechanism in the most natural way in each language. A language with coroutines, for example, might allow coroutines as event handlers instead of callbacks. This dissertation focuses on the C++ interface, which we presently consider the most suitable choice for time-sensitive applications.

ROS does not implement inter-node communication by itself. Instead, it relies on a communication middleware to communicate between threads. To allow users to choose this middleware freely, ROS abstracts from the underlying implementation through the *rmw* layer. The *rmw*

API specifies a small set of communication primitives that are used to implement the *rcl* layer. Each middleware implementation needs an adapter to translate between the *rmw* API and the middleware's API. As of now, each of the officially supported *rmw* adapters targets one of the various implementations of the *Data Distribution Service (DDS)* [84], an industry standard for network-transparent data distribution in real-time systems.

It is outside the scope of this dissertation to investigate, describe, and model DDS. We therefore do not assume any particular middleware. Instead, we assume that the middleware fulfills three properties, which we refer to as the *basic middleware requirements*. The requirements are:

1. **Reliable communication.** Within the boundaries of a shared-memory system, the middleware neither loses nor re-orders any messages.

2. **Executors dominate latency.** Within the boundaries of a shared-memory system, if the middleware is not starved, middleware-induced communication delays do not form a bottleneck (relative to the delay attributable to executors).

3. **Read your own writes.** Any thread that publishes and subscribes to the same topic always observes its own messages immediately after publication.

Apart from the impact of the ROS implementation, the timing behavior of a ROS application is also affected by the timing behavior and the scheduling policies of the underlying operating system (OS). While ROS supports various operating systems—Windows, MacOS, and Linux—this dissertation focuses exclusively on Linux, which we consider the most suitable choice for real-time applications among the three options. In preparation, the next section provides background on real-time scheduling theory and its application to Linux systems.

2.2 Real-Time Scheduling

The purpose of the operating system's thread scheduler (or *OS scheduler* for short) is to execute the runnable OS threads on the available processor cores. In the (typical) case where there are

more runnable threads than processor cores, the scheduler thus has to choose which thread to run. In a dynamic system like Linux, this decision must necessarily be made at runtime. An OS scheduler can thus be seen as an online algorithm that takes a list of runnable threads and a fixed number of cores as input and outputs for each time t which of the runnable threads runs on each core.

From a real-time perspective, the most important effect of the OS scheduler is that it determines the order in which threads run on the processor cores. This order impacts the timeliness of the computations performed by these threads. As a result, the real-time community has developed and studied various *real-time schedulers*, i.e., schedulers that exhibit a predictable worst-case behavior and are amenable to ahead-of-time analysis. Since some of these schedulers are implemented in Linux, they are an essential part of understanding and shaping the worst-case latency of ROS systems.

2.2.1 The Sporadic Real-Time Task Model

We begin with a brief introduction to real-time systems theory. Real-time systems theory abstracts from concrete operating system threads and discusses abstract *tasks* instead.[1] While there are many task and system models of varying complexity and sophistication in the literature, this introduction focuses on one of the simplest task models, the *sporadic task model* [79]. For now we further assume one of the simplest system models, the *uniprocessor system*, which models a single processor core that runs a single task at a time.

The sporadic task model represents a system as a set of *tasks*. Each task is triggered by some external event and reacts to each event by activating a *job*. A task τ_i is characterized by three parameters: a *worst-case execution time (WCET)* C_i, a *relative deadline* D_i, and a *minimum inter-arrival time* T_i. The WCET is an upper bound on the processor time needed by each job of τ_i. The relative deadline represents the task's timing constraint: a job of τ_i that is activated at time t must complete by time $t + D_i$. The time $t + D_i$ is also known as the *absolute deadline* of

[1] In Linux kernel jargon, the words *task* and *thread* are used interchangeably. In this dissertation, *task* refers to the abstract tasks of real-time theory and *thread* to the concrete Linux implementation.

the job. The time between activation and completion of a job is called the *response time* of the job. The longest possible response time experienced by a job of τ_i is called the *worst-case response time* of τ_i. Finally, the minimum inter-arrival time T_i lower-bounds the separation between two consecutive activations of τ_i. Throughout this dissertation we assume a discrete time model wherein all time parameters are integer multiples of a time unit $\epsilon \triangleq 1$ (e.g., a processor cycle).

A useful derived metric is the *utilization* u_i of τ_i, which is defined as the ratio $\frac{C_i}{T_i}$. Intuitively, u_i measures the fraction of a processor's total capacity that may be consumed by τ_i in the long run. For example, a task with utilization 0.5 consumes up to 50% of the compute capacity provided by one core.

The principal objective of a real-time scheduler is to ensure that all jobs finish before their absolute deadline under all circumstances, i.e., to guarantee that the worst-case response time of each task is no larger than its relative deadline. If this condition is fulfilled, a task set is *schedulable* under the scheduling algorithm.

One of the most commonly implemented real-time scheduling algorithms is the *fixed-priority (FP)* algorithm. Under fixed-priority scheduling, each task is assigned a numeric priority, and each job inherits the priority of its task. At any time, the scheduler runs the job with the highest priority. Another common real-time scheduling algorithm is the *earliest-deadline-first (EDF)* algorithm, which always runs the job with the earliest absolute deadline.

For many scheduling algorithms, including FP and EDF, the schedulability of a task set can be established with a so-called *schedulability test*, an algorithm that receives the task set as input and decides whether the task set is definitely schedulable or potentially unschedulable.

One approach to determine whether a task set fulfills all its timing constraints is to find an explicit upper bound on the worst-case response time of each task. An algorithm computing such an upper bound is called a *response-time analysis*. Since response-time analyses report explicit timing bounds instead of just a binary decision, they are particularly useful in more open contexts where not all tasks have clear deadlines or where timing constraints extend over chains of multiple tasks. We will discuss techniques for response-time analysis in more detail in Section 2.3.

2.2.2 Resource Reservations

The guarantees provided by response-time analyses and schedulability tests are only valid if all tasks behave as described by the model. As soon as a single task deviates from the model, for example by running for more than its assumed WCET, the schedulability guarantees no longer hold. In practice, this fragility is a significant limitation. Allowing errors in one subsystem to induce timing errors in nominally independent subsystems forces developers to vet every component with the vigor and reliability required for the most critical component. For larger systems, this is extremely expensive and makes it difficult to integrate third-party components into the system.

To address this problem, it is desirable to provide *timing isolation* between independent tasks. Timing isolation between two tasks τ_i and τ_j means that the worst-case response time of τ_i does not depend on the properties of τ_j and vice versa. An error in τ_i's specification therefore does not jeopardize the timing guarantees for τ_j.

A classic OS-level abstraction to achieve timing isolation is the *resource reservation* [77]. A resource reservation limits interference between tasks by limiting their resource consumption. Resource reservations are typically implemented as a second-level scheduler called a *reservation server*. Once a task is assigned to a reservation server, its jobs are no longer scheduled directly by the OS scheduler. Instead, the OS scheduler schedules the reservation server, and the reservation server schedules the jobs of the tasks assigned to it.

Many different reservation algorithms have been designed and developed over the last 30 years [25], with various different features and support for different scheduling algorithms. Of particular relevance to this dissertation is the *Constant Bandwidth Server (CBS)* [4, 7, 11], which is the only reservation server implemented in Linux. A CBS reservation r is characterized by an execution-time budget $budget(r)$, a period $period(r)$, and a deadline $dl(r)$. The reservation guarantees to provide its client tasks $budget(r)$ units of execution time in every interval of length $period(r)$, which are available at most $dl(r)$ time units after the beginning of the period. The ratio $\frac{budget(r)}{period(r)}$ is also known as the *bandwidth* $bw(r)$.

It is no accident that the parameters of a CBS reservation are similar to the task parameters in the sporadic task model. From the point of view of the OS scheduler, a CBS reservation behaves like a task with WCET $budget(r)$, minimum inter-arrival time $period(r)$, and deadline $dl(r)$. This property is useful to prove that the servers' supply guarantees hold: a set of CBS reservations always fulfill their supply guarantees if and only if the corresponding sporadic task system is schedulable.

Due to this correspondence, the supply guarantees of a CBS reservation can be verified through a simple schedulability test. A set of k CBS reservations $r_1, \ldots, r_k$ with implicit deadlines (i.e. $\forall r_i.\ dl(r_i) = period(r_i)$) is always schedulable on a uniprocessor as long as the committed bandwidth is below 100%, i.e., $\sum_{i=1}^{k} bw(r_i) \leq 1$.

Once the schedulability of the reservations is established, the worst-case response time of the tasks in one reservation server is independent of interference by tasks in other reservations. The response-time analysis can analyze each reservation in isolation.

Resource reservations can be divided into *hard* and *soft* reservations [99]. A hard reservation is not allowed to exceed its budget under any circumstances, even if only non-critical background tasks are waiting for processor time. In contrast, a soft reservation can overrun its budget as long as this does not prevent any other reservation from receiving its reserved execution time. The consumed budget is still deducted from the reservation; essentially, the reservation borrows execution time and pays it back later. As a result, hard reservations provide temporal isolation within the reservation server itself: the same budget is available during each period, independent of the behavior of previous periods. In contrast, soft reservations do *not* provide temporal isolation within a task: the processor time of a task may be diminished if previous jobs of the same tasks have overdrawn their budget. In exchange, soft reservations can opportunistically consume processor time that would otherwise go to less critical tasks and may reduce response times in the average case.

2.2.3 Multiprocessor Scheduling

The CBS algorithm was originally built around the uniprocessor EDF scheduling policy [4]. On multicore systems, this algorithm needs to be adapted to more than one core. There are four general approaches to extend uniprocessor policies to multiple processors: The most straightforward way is *partitioned scheduling*. Under partitioned scheduling, each task is assigned to exactly one core, and each core is then scheduled using the uniprocessor scheduling policy. Effectively, partitioned scheduling treats a system with N cores as N parallel systems with one core each. Another approach is to adjust the scheduling policy to allocate pending tasks to multiple cores. For example, a fixed-priority scheduler for N cores runs the N highest-priority jobs, an EDF scheduler for N cores runs the tasks with the N earliest deadlines, *etc.* This is known as *global scheduling*. An intermediate form between the two is *clustered scheduling*, where tasks are partitioned onto multiple clusters, each of which runs a separate multiprocessor scheduler. All of these options are subsumed by *arbitrary-processor-affinities (APA) scheduling*, where tasks are scheduled by a single scheduler, but each task may be restricted to an arbitrary subset of the available cores.

In this dissertation, we use partitioned scheduling due to its better analyzability and lower overheads. Brandenburg and Gül [19] showed that partitioned scheduling works well even on highly utilized systems and therefore recommend it over more complex approaches like global scheduling.[2] In particular, partitioned scheduling retains the uniprocessor schedulability test for CBS servers: as long as $\sum bw(r_i) \leq 1$ holds for each core, the reservations are schedulable.

However, the advantages of partitioned scheduling do not come for free, as it requires users to map each task or reservation to a core in advance. This is an instance of the *bin-packing problem*: each task τ_i is an item of size u_i, each core is a bin of size 1, and tasks need to be assigned to cores without exceeding the size constraint. Although the bin-packing problem is NP-complete, Brandenburg and Gül showed that even at 90% utilization, more than 80% of their (randomly-generated) systems could be partitioned with simple heuristics like *first-fit-decreasing*

[2] According to the paper, *semi-partitioned scheduling* performs even better. Unfortunately, it is not available in Linux.

(FFD) and *worst-fit-decreasing (WFD)* [64]. In this dissertation, we therefore use these two heuristics to partition workloads.

Both FFD and WFD are single-pass heuristics: they consider each task only once and make a final allocation decision before continuing with the next task. Both heuristics proceed in decreasing order of utilization. Each task is then assigned to the *first* core with enough capacity (FFD) or to the *worst* core, i.e., the core with the largest remaining capacity (WFD).

Since both heuristics only need to perform a few simple utilization comparisons for each task, they are extremely cheap to compute. We therefore try both heuristics to increase the chances of finding a feasible mapping.

2.2.4 Thread Scheduling in Linux

The Linux kernel implements six scheduling policies. Three of these, namely SCHED_IDLE, SCHED_BATCH, and SCHED_OTHER, are variants of the so-called *Completely Fair Scheduler (CFS)*, a non-real-time scheduler and the default setting for Linux threads. The other three policies are real-time schedulers. Two of these, namely SCHED_RR and SCHED_FIFO, are variants of fixed-priority scheduling (FP), while SCHED_DEADLINE is a resource reservation scheduler using the CBS algorithm.

The three *scheduling classes*—CFS, FP, and SCHED_DEADLINE—are ordered in a strict priority hierarchy. The lowest-priority scheduler is CFS, followed by the FP schedulers and finally SCHED_DEADLINE at the top. This means that FP threads *always* take priority over CFS threads and SCHED_DEADLINE threads always take priority over both CFS and FP threads. The hierarchy is hardcoded in the kernel and cannot be changed.

CFS. The CFS is the default Linux scheduler. It assigns CPU time to threads such that the total available processor time is distributed in approximately equal parts to all runnable threads. Users can skew the distribution towards individual threads by adjusting the *weight* of a thread, for example through the *nice* mechanism.

Since the CFS policy is not intended for real-time threads and does not provide any scheduling guarantees, the exact algorithm and the difference between the three CFS variants is not relevant to this dissertation. More details on CFS can be found in the Linux kernel documentation [1].

FP. Under the fixed-priority scheduler, each thread is assigned a numeric priority. On a uniprocessor system, the scheduler always runs the highest-priority thread in the system. If two threads have the same priority, a tie-breaking rule is employed. The two variants use different tie-breakers: SCHED_FIFO prioritizes the thread that entered the runqueue first, i.e., it runs threads of the same priority in FIFO order. SCHED_RR also runs threads in FIFO order but adds a time-slicing mechanism: once a thread has consumed a (configurable) amount of processor time, it is descheduled and moved to the end of the FIFO queue. This gives other threads at the same priority level a chance to run.

SCHED_DEADLINE. The SCHED_DEADLINE scheduler [39] implements the Hard Constant Bandwidth Server (H-CBS) algorithm [11]. As described in Section 2.2.2, H-CBS provides temporal isolation through resource reservations servers. Unlike soft CBS, hard CBS is unable to opportunistically reroute spare bandwidth between reservations. This shortcoming is addressed through the GRUB [68] bandwidth reclamation mechanism. Since GRUB improves the average-case performance and does not affect the worst case, it does not need to be considered in worst-case analysis.

The SCHED_DEADLINE implementation diverges from the theory (i.e., H-CBS) in one major aspect: SCHED_DEADLINE does not implement reservation servers as first-class objects. Instead, it treats each thread as a separate CBS reservation with a separate budget, period, and deadline. As a result, it is impossible to share budgets among multiple threads.

Overall, the techniques presented so far all make the timing behavior of the threads more predictable. In the next section, we discuss some important techniques to exploit this predictability and bound worst-case response times.

2.3 Techniques for Response-Time Analysis

The timing correctness of a real-time system can be proven through a *response-time analysis*. A response-time analysis is an algorithm to prove the timing correctness of a system by computing an upper bound on the response time of a task or chain of tasks (Section 2.2.1).

Although the response-time analyses for various task models differ in important details, they share common techniques that are applicable to a wide range of systems. In this section, we discuss three standard techniques that underlie the response-time analysis for ROS systems introduced in Chapter 4.

First, we describe a technique to abstract from scheduler and task model details with *supply-* and *request-bound functions* (Section 2.3.1). These functions allow a more general analysis formulation, thereby making the same analysis applicable to multiple scheduler- and task models.

Second, we describe the *busy-window principle*, a technique to bound a task's response time by analyzing a window of time where the tasks' processor is continuously busy (Section 2.3.2).

And finally, we describe the *compositional performance analysis*, an efficient response-time analysis technique for large and interconnected modular systems (Section 2.3.3).

2.3.1 Supply and Demand

Many response-time analyses are general enough to apply to a wide range of task models and schedulers. No matter whether the scheduler is a fixed-priority scheduler or an EDF scheduler, and no matter whether the task model uses scalar WCETs or more complex execution-time models, in many cases the analysis needs to know only two quantities: the amount of processor time needed to complete a task (the *demand*), and the amount of processor time available to execute said task (the *supply*).

Of particular relevance to a worst-case analysis is the *highest possible* demand during a time window and the *lowest possible* supply during the same time window. This motivates the abstraction of scheduler and task model details through the *request-bound function* [9, 118] and the *supply-bound function* [108, 118].

The **request-bound function** $rbf(\Delta)$ (also called *demand-bound function*) of a task τ_i maps any duration Δ to an upper bound on the total processor time consumption of all jobs of τ_i that are activated during an arbitrary time interval of length Δ.

For example, a request-bound function for a task τ_i in the sporadic task model can be defined based on the WCET C_i and the period T_i as

$$rbf_i(\Delta) \triangleq C_i \cdot \left\lceil \frac{\Delta}{T_i} \right\rceil \qquad \text{(\textit{request bound for sporadic tasks})}$$

The **supply-bound function** $sbf(\Delta)$ of a task τ_i maps any duration Δ to a lower bound on the processor time available to τ_i during an arbitrary interval of length Δ. The scheduler may supply processor time at arbitrary times during the interval; τ_i forfeits any supply that is provided while τ_i is not ready to run.

As an example, consider a preemptive fixed-priority scheduler. Let π_i denote the priority of task τ_i, and assume for simplicity that each task has a unique priority. Then in any window of length Δ, the processor time available to τ_i consists of the total supply distributed by the fixed-priority scheduler (Δ), minus the processor time consumed by higher-priority tasks. A supply-bound function thus can be defined as [118]:

$$sbf_i(\Delta) \triangleq \max(0, \Delta - \sum_{\{\tau_x \mid \pi_x \geq \pi_i\}} rbf_x(\Delta)). \qquad \text{(\textit{supply bound for FP tasks})}$$

The supply-bound abstraction is particularly useful to describe the supply guarantees of resource reservations [108]. Due to the temporal isolation property, the supply-bound function of a reservation depends only on the reservation parameters: budget, deadline, and period.

For example, consider the sbf of a CBS reservation r_k with $period(k) = dl(k)$. Following Shin and Lee [108], we first identify the *bounded delay* (or *blackout window*) of the reservation. The bounded delay describes the longest possible waiting time until the reservation provides the first processor cycle of supply. Figure 2.2 gives a graphical illustration of this delay for general resource reservations. The longest possible time without any supply occurs if the budget during

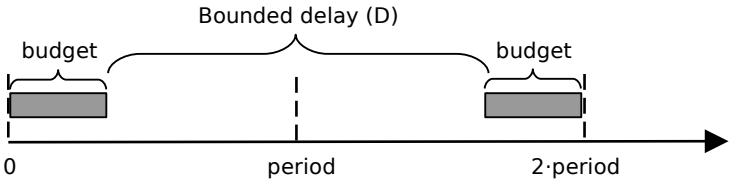


Figure 2.2: Bounded delay in CBS reservations.

one period is provided at the earliest possible time and the budget of the subsequent period is available at the latest possible time.

Accounting for this blackout window, a supply-bound function for a CBS reservation r_k can be defined as [12]:

$$
\begin{aligned}
sbf_k(\Delta) \triangleq & N \cdot budget(r_k) + F \\
\text{where} \quad & D = 2 \cdot (period(r_k) - budget(r_k)), \\
& N = \left\lfloor \frac{\Delta - D}{period(r_k)} \right\rfloor, \text{ and} \\
& F = \max(0, \Delta - (N \cdot period(r_k) + 2D)).
\end{aligned}
$$

Intuitively, the term D upper-bounds the bounded delay, N lower-bounds the number of complete reservation periods within the interval of length Δ (during which the full budget is supplied by the reservation), and F lower-bounds the supply by the remaining fractional period.

2.3.2 The Busy-Window Principle

Bounding the worst-case response time of a task τ_i requires an argument about the worst-case scenario a job of τ_i might find itself in. This is commonly done using the *busy-window principle* (*e.g.*, [16, 21, 58, 61, 86, 87, 105]). We refer to Bozhko and Brandenburg [16] for a formal description of the busy-window principle that generalizes to a wide range of schedulers. As a concrete example, and to provide intuition on how the busy-window principle works, we present the special case of a simple fixed-priority preemptive scheduler.

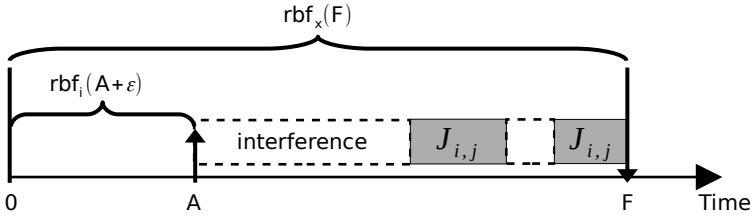


Figure 2.3: The busy window of a job $J_{i,j}$. The job is activated at time A and completes at time F. Self-interfering jobs must be activated in $[0, A + \epsilon)$, generating $rbf_i(A + \epsilon)$ time units of interference. Interfering jobs from any other task τ_x must be activated in $[0, F)$, generating $rbf_x(F)$ time units of interference.

Informally speaking, a busy window is a time window during which the processor is continuously busy. To find a response-time bound for a task τ_i, the analysis considers an arbitrary job of τ_i (called the *job under analysis*), and considers the busy window surrounding this job (Fig. 2.3). The busy window starts at time 0 and ends at time t_2. At time A, the job under analysis is released. The analysis then identifies the equilibrium point $F \in [0, t_2)$ where the processor time supply is guaranteed to match the demand. Specifically, F is the point where $sbf(|[0, F)|)$ suffices to run the job under analysis as well as all jobs that could possibly interfere with it. Since the processor never idles during a busy window, the job under analysis must be complete by time F.

Definition of the busy window. Let $J_{i,j}$ denote the j-th job of task τ_i. Recall from Section 2.2.1 that the fixed-priority preemptive scheduler requires a priority for each task and always runs a job of the task with the highest priority. Let π_i denote the priority of τ_i; for simplicity, we assume that each task has a unique priority.

We say that a job $J_{i,j}$ is *pending* at time t if it is activated at or before time t and completes after time t. If the job is pending at both t and $t - \epsilon$ it is *carried in* at time t.

A job $J_{i,j}$ suffers *interference* from another job $J_{k,l}$ at time t if $J_{k,l}$ runs at time t but $J_{i,j}$ is not complete at time t. A time instant t is called a *quiet time* for a job $J_{i,j}$ if neither $J_{i,j}$ nor any other job that may interfere with $J_{i,j}$ are carried in at t. Finally, an interval $[t_1, t_2)$ is a *busy window* for $J_{i,j}$ if both t_1 and t_2 are quiet times for $J_{i,j}$, there is no quiet time for $J_{i,j}$ between t_1 and t_2, and $J_{i,j}$ is activated in $[t_1, t_2)$.

The busy-window analysis. In the fixed-priority preemptive scheduler, an arbitrary job $J_{i,j}$ cannot suffer interference from jobs that have lower priority than τ_i. In any interval of length Δ, the total processor demand of all jobs interfering with $J_{i,j}$ is thus bounded by $\sum_{\{\tau_x \mid \pi_x \geq \pi_i\}} rbf_x(\Delta)$. Now let A be the activation time of $J_{i,j}$. Then any positive value F that satisfies

$$sbf(F) \geq rbf_i(A + \epsilon) + \sum_{\{\tau_x \mid \pi_x \geq \pi_i\}} rbf_x(F)$$

is an upper bound for the completion time of $J_{i,j}$. The response time of $J_{i,j}$ is thus bounded by $F - A$. As depicted in Fig. 2.3, $rbf_i(A + \epsilon)$ bounds the demand by both $J_{i,j}$ itself and any other jobs of τ_i that are activated before $J_{i,j}$'s activation. $\sum_{\{\tau_x \mid \pi_x \geq \pi_i\}} rbf_x(F)$ bounds the interference by higher-priority jobs, which can interfere with $J_{i,j}$ even if they are activated after time A.

Note that the resulting response-time bound only holds for one particular activation offset A. To obtain a response-time bound for arbitrary activation offsets, it is necessary to compute the above bound for all possible offsets. But since there is an unbounded number of such offsets, a practical analysis needs to constrain the number of offsets to consider. This happens in two steps: first, the analysis bounds the search space from above by upper-bounding the length of the busy window. Second, the analysis identifies redundant offsets, i.e., offsets that are guaranteed not to yield the maximal response time. The resulting search space of non-redundant offsets is finite and sparse enough to allow for an exhaustive search.

For the fixed-priority scheduler, the length of the busy window (and thus the search space) is bounded by the least positive value L that satisfies

$$sbf(L) \geq \sum_{\{\tau_x \mid \pi_x \geq \pi_i\}} rbf_x(L).$$

Within this search space $[0, L)$, any value A for which $rbf_i(A - \epsilon) = rbf_i(A)$ holds is redundant, i.e., does not yield a maximal response-time bound [16]. As a result, only offsets in the set $\mathcal{A} \triangleq \{0\} \cup \{A \mid 0 < A < L \wedge rbf_i(A - \epsilon) \neq rbf_i(A)\}$ need to be considered.

2.3.3 Compositional Performance Analysis

In the sporadic model, job activation times are independent of the other jobs in the system. This is not the case in a data-driven system, where some tasks are triggered by the completion of another job. If jobs of a task τ_i trigger jobs of another task τ_j, then the analysis can no longer consider both independently, even if the two tasks do not interfere with each other: the timing of τ_i determines the number of activations of τ_j.

Compositional performance analysis (CPA) [61] is a method to analyze such systems. It restores the ability to analyze τ_i and τ_j independently by abstracting τ_i's impact onto τ_j through an *event model*, usually an *activation curve*. An activation curve (or *arrival curve*) maps a duration Δ to an upper bound on the number of activations of τ_j that may occur during any interval of length Δ [67]. Given this curve, an analysis of τ_j no longer has to consider τ_i explicitly.

Isolating the impact of trigger relationships into the activation curve allows CPA to split the response-time analysis of a complex inter-dependent system into two separate steps: a *local analysis* step and an *activation curve propagation* step. The local analysis operates on each component individually; it assumes an activation curve for each task as given and computes a response-time bound for each task based on this activation curve. The activation curve propagation step operates on the connection between the components; it assumes a response-time bound for each task as given and computes an activation curve for each task based on this response-time bound. Overall, each step computes the other step's input but requires the other step's output.

CPA resolves this circular dependency by alternating between the two steps in a fixed-point iteration. The analysis computes a series of response-time bounds $(R_i^0, R_i^1, \ldots, R_i^n)$ and a series of activation curves $(\eta_j^0, \eta_j^1, \ldots, \eta_j^n)$. For any $k > 0$, the computation of R_i^k (respectively, η_j^k) uses the result of the previous iteration η_j^{k-1} (respectively, R_i^{k-1}); R_i^0 is initialized as 0, η_j^0 is initialized as η_i. Since R_i^k grows monotonically in k, the computed response-time bounds either grow without bounds or converge to a fixed point. If the iteration diverges, the analysis fails and reports that the response time of τ_i is unbounded. If the iteration converges, the fixed point R_i^∞ is a response-time bound for τ_i.

Activation curve propagation. We conclude this section by formally defining the activation curve propagation process. Consider two tasks τ_i and τ_j, where each job of τ_i activates a job of τ_j during its runtime. Assume further that τ_i's activation curve $\eta_i(\Delta)$ and a response-time bound R_i are known. Activation curve propagation then needs to find an activation curve $\eta_j(\Delta)$ such that $\eta_j(\Delta)$ upper-bounds the number of activations of τ_j in any time interval of length Δ. We also say that the analysis *propagates* $\eta_j(\Delta)$ from τ_i to τ_j.

The propagation mechanism exploits that any job of τ_i that is activated at a time t must activate an job of τ_j before time $t + R_i$, since R_i is a response-time bound of τ_i. Thus, the number of jobs of τ_j activated in an arbitrary time window $[t_1, t_1 + \Delta)$ is upper-bounded by the number of jobs of τ_i activated in the time window $(t_1 - R_i, t_1 + \Delta)$, which is again upper-bounded by $\eta_i(\Delta + R_i - \epsilon)$.

The activation curve of τ_j can thus be derived from R_i and η_i [37]:

$$\eta_j(\Delta) \triangleq \eta_i(\Delta + R_i - \epsilon). \qquad \text{(\textit{activation-curve propagation})}$$

An additional propagation delay δ between τ_i and τ_j can be incorporated by adding δ to the response-time bound R_i. If τ_j may be activated by more than one task, the propagated curves from each activating task are summed up.

2.4 Related Work

In this section, we review prior work and place the contributions of this dissertation into context.

Alternative ROS executors. Choi et al., Staschulat et al., and Arafat et al. explored alternative ROS executor designs with improved time-predictability. Choi et al. [30] developed a new ROS executor called *PiCAS* that supports explicit prioritization of callbacks. They developed an automatic priority assignment scheme that prioritizes callbacks and executor threads minimize chain response times. The priority assignment is complemented by an allocation scheme that

optimizes the assignment of nodes onto executors and of executors onto processor cores.

As their evaluation shows, the resulting system exhibits much lower observed latency and response-time bounds for high-priority chains. If the final integrator of a ROS system can replace all executors and rearrange the nodes at will, their approach provides significant benefits. However, unlike our work, it is not directly applicable to unmodified systems.

Staschulat et al. [113] developed a new ROS executor called *rclc executor* that retains the round-robin aspect of the ROS executor but avoids much of the resulting pessimism by giving the user much more control over the scheduling. Compared to the default ROS executor, their variant introduces three main changes. First, it allows users to assign arbitrary priorities to callbacks. Second, it implements more complex activation conditions for callbacks. The executor can, for example, be configured to wait until *all* assigned callbacks have been triggered or until one particular callback has been triggered. Third, it (optionally) enforces *logical execution-time (LET)* semantics [62], in which all communication occurs at pre-defined times to increase predictability.

The extended scheduling mechanisms provided by the rclc executor require explicit configuration by the user. Like PiCAS, the rclc executor is therefore not applicable to unmodified ROS components.

Finally, concurrent work by Arafat et al. [8] proposed a ROS executor that supports dynamic-priority scheduling for callbacks and processing chains. The executor uses a ready queue instead of the default executor's *readySet* and schedules waiting callbacks with the EDF scheduling algorithm. Their results confirm that their scheduler achieves lower average end-to-end latencies for chains that are assigned a low priority by PiCAS' priority assignment scheme while achieving similar latencies for chains that are assigned a high priority by PiCAS' priority assignment.

Since multiple executors can co-exist in different threads on the same system, it would be possible to extend ROS-Llama to support alternative executors. Future work could integrate the additional per-executor response-time analyses into ROS-Llama and extend the executor implementations with ROS-Llama instrumentation. This is likely straightforward for the PiCAS executor and the deadline-based executor, as both come with a response-time analysis in their

paper that uses a similar analysis technique as the analysis in Chapter 4. Integrating the rclc executor this way would be more difficult; particularly the more complex activation conditions would require significant extensions to the model.

Empirical latency measurements. A range of prior work empirically measured control latency in ROS systems to estimate its applicability to real-time workloads. Park et al. [91] evaluated the predictability of ROS 2 in comparison to ROS 1. Their experiments focused exclusively on the jitter and response time of the periodic sensor nodes; the complex and computation-intensive navigation stack was offloaded to a backend PC. Their results confirm that the ROS 2 implementation improves timing determinism compared to ROS 1. The results further demonstrate the importance of timing by showing that the higher timing precision of ROS 2 directly impacts the path-following precision of the robot.

Gutiérrez et al. [56] also investigated the real-time performance of ROS 2, but focused on the DDS communication instead. They demonstrated how to reduce the worst-case communication latency by tuning the implementation-specific real-time settings of three DDS implementations.

Real-time issues have also been studied in the context of ROS 1, e.g. [102, 115]. Since real-time support has changed significantly between ROS 1 and ROS 2, it is unclear whether these results still apply.

Latency monitoring. Schlatow [104, Chapter 5] developed a runtime-monitoring system for ROS 2 and evaluated it on the *Autoware.Auto* software [44]. Instead of developing a fine-grained timing model of ROS systems, their runtime monitor observes only the communication layer of the system and informs the ROS system about any observed latency violations. It is the job of the ROS component developer to specify an appropriate reaction.

It would be interesting to explore if access to an automatically extracted timing model would allow for more automated reactions. Alternatively, ROS-Llama might be able to automatically generate intermediate timing constraints for Schlatow's monitoring system, potentially allowing the system to react more promptly to emerging timing issues.

Response-time analysis for ROS. Concurrently to this dissertation, Tang et al. [116] also developed a response-time analysis for ROS 2. Compared to this work, their analysis focuses on finding more precise analysis bounds for the special case of independent linear processing chains. They further investigate how ROS developers should prioritize their callbacks within each ROS executor to minimize end-to-end response times. Unfortunately, Tang et al.'s analysis does not apply to the workloads considered in Chapter 6, which contain branching processing chains and are therefore incompatible with their linear processing chain assumption.

In more distantly related work, Tang et al. [117] developed an analysis for a ROS-inspired scheduler that also exhibits a round-robin-like property. As their analysis considers only systems of independent tasks, it does not transfer easily to the general ROS systems considered herein.

Non-ROS robotics. The real-time needs of robotics workloads have long been a subject of intense study, with classic papers in this area defining suitable APIs and runtime systems. Better-known examples include the robotics operating systems HARTIK [23, 24], ETHNOS [94, 95], and XO/2 [20]. Notably, HARTIK introduced an approach to controlled degradation that is similar to the one we integrate into ROS-Llama (Chapter 5).

More recent examples usually do not implement a separate operating system kernel, but, similar to ROS, are frameworks running on general-purpose operating systems like Linux. Examples include YARP [78], MRPT [31], Orocos [22] and Fawkes [82]. Except for MRPT, which is a curated collection of libraries rather than a framework, all these frameworks define a communication mechanism and encourage the integration of third-party components.

In the context of this dissertation, the Orocos and Fawkes frameworks are particularly interesting due to their focus on real-time safety and their approach to execution management. The Fawkes framework relies mainly on the *sense-plan-act* pipeline, a cyclic executive that periodically iterates through a set of 10 stages. Components add callback functions at the appropriate points in this pipeline. Orocos leaves much more of the callback scheduling to the component developer and allows them to define the execution order of the callbacks through a state machine.

In this dissertation, we focus exclusively on ROS. As discussed in Section 1.1, ROS's popularity strongly suggests that the robotics community is unwilling to switch away from ROS just because of better real-time support. This dissertation therefore focuses on existing ROS applications and therefore cannot impose a custom API.

Compositional timing analysis. To improve the scalability of timing analysis in distributed systems, various *compositional* analysis approaches have been proposed [92]. A compositional analysis considers the system as a collection of components, each of which is analyzed separately. Dependencies between components are addressed by propagating event models, which describe the activation pattern of individual tasks.

The timing analysis in this dissertation is inspired by the *compositional performance analysis (CPA)* approach for distributed systems (cf. Section 2.3.3). Since the original proposal by Henia et al. [61], CPA has received various extensions and improvements over the years; Hofmann et al. [63] provide an overview over the main developments.

Another compositional approach is *real-time calculus (RTC)* [118] and its extension *modular performance analysis with RTC (MPA-RTC)* [119]. RTC models real-time systems as a set of *performance components* connected by supply streams. Each resource receives a supply stream, consumes a part of it, and outputs the unused part as a *residual supply curve*. This is used to model inter-task interference and the effect of schedulers. It further receives a demand stream, processes some of it, and outputs a *delivered computation* stream. The interaction between the supply stream and the demand stream is described through equations adopted from network calculus [67].

MPA-RTC extends RTC with activation relationships between components. The delivered computation stream of the activating component is transformed into a stream of discrete activation events and feeds into the demand stream of the activated component. The resulting network of supply- and demand streams thus represents both interference and activation relationships between the components.

While real-time calculus enables an elegant formulation for some schedulers like preemptive fixed-priority schedulers, it is not immediately obvious how to extend it to support the ROS callback scheduler. We therefore opted for the CPA approach instead.

A third compositional approach uses *timed automata* [60]. Each computational resource and each communication link is modeled as a timed automaton. Combining the automata for all resource and communication links in the system yields a network of timed automata, whose timing properties can be analyzed by a model checker.

While the timed-automata approach produces more precise results than CPA and RTC [92], it also tends to be more computationally expensive than the other two approaches [92]. Since ROS-Llama needs to perform timing analysis at runtime, we opted for the CPA approach to avoid excessive analysis overhead.

Finding reservation parameters. ROS-Llama needs to determine suitable resource reservation parameters for various ROS executors automatically. Although the problem has been extensively studied before, none of the existing approaches is directly applicable to ROS.

Lipari and Bini [69, 70] considered the relationship between the reservation's bandwidth and its period for a set of independent periodic tasks. The challenge in finding suitable parameters is that there are two objectives to fulfill: a reservation should have a small bandwidth to minimize the claimed processor utilization but should also have a large period to minimize scheduling overheads. Unfortunately, these two goals conflict: a larger period usually requires additional bandwidth to compensate for the larger initial supply delay and vice versa. Lipari and Bini formulated a joint optimization problem to find the optimal combination of the two parameters. Unfortunately, their approach is not directly applicable to ROS due to complex interdependencies in the ROS callback graph, which we will discuss in Section 5.4.2.

Buttazzo and Bini [26] evaluated the benefits of choosing a period that matches the period of the underlying workload. They considered the scheduling rules of the Constant Bandwidth Server directly, without using the supply-bound function abstraction. By choosing the right

reservation period, the replenishment time of the server was made to coincide with the task arrival period, which eliminates the bounded delay entirely. The work is not directly applicable to our analysis since the supply-bound function abstraction cannot exploit the relative timing between task activations and reservation replenishment. Furthermore, ROS executors serve different callbacks, which tend to have different activation periods. Nevertheless, future work to adapt this approach to ROS-specific challenges could significantly reduce the pessimism incurred by the supply abstraction.

Palopoli and Abeni [88] developed a method to automatically determine a suitable budget for a set of "legacy applications" (by which the paper means applications that are not structured as a series of jobs). The budget is chosen and adapted continuously by a feedback controller that attempts to minimize both the budget and the number of budget overruns. It is therefore a cheaper and simpler alternative to ROS-Llama's budget assignment process (Section 5.4.2). However, since the approach lacks an internal model of the involved threads, it is purely reactive; unlike ROS-Llama, the feedback controller will necessarily underestimate the required budget occasionally and therefore continuously runs the risk of exceeding the allowed latency.

Adaptation and graceful degradation. Section 5.1 identifies a set of requirements an automatic latency manager has to fulfill. Two of the requirements call for *unsurprising overloads*, *i.e.*, a pre-defined and controlled *graceful degradation process* that sacrifices less critical workloads in favor of more critical workloads in case of an overload, and support for *unpredictable environments*, *i.e.*, a way to *adapt* to changing behavior and changing demands of the workload.

The general ideas of graceful degradation in the face of transient overload and adaptation towards changing circumstances have also been explored from many angles in prior work. A very well-explored approach is the concept of *service levels* [5, 10, 15, 55, 59, 74, 75, 83], which relies on application developers to explicitly define different operating modes and a notion of utility associated with each mode. The service manager can then choose appropriate levels of service throughout the application by solving a utility maximization problem.

This dissertation does not consider service levels since we target unmodified ROS workloads, which generally do not expose multiple operating modes.

Another common adaptation strategy is to rely on feedback-control theory to adjust scheduling parameters. Prior work has explored approaches to directly control periods [28, 29], QoS-levels [15, 28, 75, 112], and reservation budgets [5, 6, 38, 41, 89, 90, 107], to name a few. Such approaches could in principle be transferred to ROS but, to our knowledge, have not yet been systematically studied in that context.

3 A Timing Model of ROS Applications

As the first step towards controlling the worst-case latency of ROS systems, this chapter presents a timing model for ROS applications. We first identify and describe the scheduling algorithm used by the ROS executor (Section 3.1). Although this algorithm is crucial to understanding the order in which callbacks are executed, it was not specified in the ROS documentation prior to our work[1]. It therefore had to be reverse-engineered from the implementation. We then develop a real-time task model for ROS systems (Section 3.2). The model represents a ROS system as a network of callbacks scheduled by a two-level hierarchical scheduler. The resulting model is simple enough to allow efficient response-time analysis but comprehensive enough to cover many real-world ROS packages.

3.1 The ROS Executor

As discussed in Chapter 2, ROS packages consist of a collection of callbacks, which are grouped into one or more nodes. Each node is assigned to an executor thread, which is responsible for running the callbacks of its assigned nodes. To do so, the executor continuously monitors the communications layer for events and invokes the associated callback function whenever an event occurs. The executor thus functions as a callback scheduler: it runs the runnable callbacks on the available executor threads and decides which callbacks to run if there are not enough threads to run them all. Understanding the executor is therefore a crucial step towards understanding the timing behavior of ROS applications.

[1] By now, the ROS documentation has been extended with the results in this chapter.

The ROS C++ library provides its executor in a single-threaded and a multi-threaded variant. In this dissertation, we focus on the simpler and more predictable single-threaded executor.

The next section (Section 3.1.1) describes the scheduling algorithm used by this single-threaded executor. Since the algorithm is not specified in the ROS documentation, the description is based on a careful study of the ROS source code. To confirm the correctness of our description, a second part validates the observations with an experiment that demonstrates and corroborates the findings on a concrete example (Section 3.1.2).

3.1.1 The Algorithm

The executor is responsible for taking messages from the input queues of the middleware layer (via 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 callbacks, which it updates in irregular, execution-dependent intervals. We refer to callbacks that are subject to the *readySet* as *polled* callbacks. Other callbacks circumvent the *readySet*; their readiness is checked directly at the source. We refer to these callbacks as *privileged* callbacks.

Whether a callback is privileged or polled is determined by the ROS implementation and the callback's type. In older ROS versions (up to "*Dashing Diademata*"), timers are privileged and all other callbacks are polled. In newer ROS versions ("*Eloquent Elusor*" and newer), timers have lost their privileged status and are also polled.

The complete algorithm is depicted in Figure 3.1. If the executor is idle, it updates its *readySet*. This is the only step in which the executor interacts with the underlying communication layer. It then looks for a callback to execute by searching through the five callback categories[2]. It first checks whether any of the privileged callbacks are ready. It then searches the *readySet* for timers, subscriptions, services, and clients (in this order). Evaluating the queues in a fixed order has the

[2]The executor blocks if there is nothing to do; this optimization has been omitted for clarity.

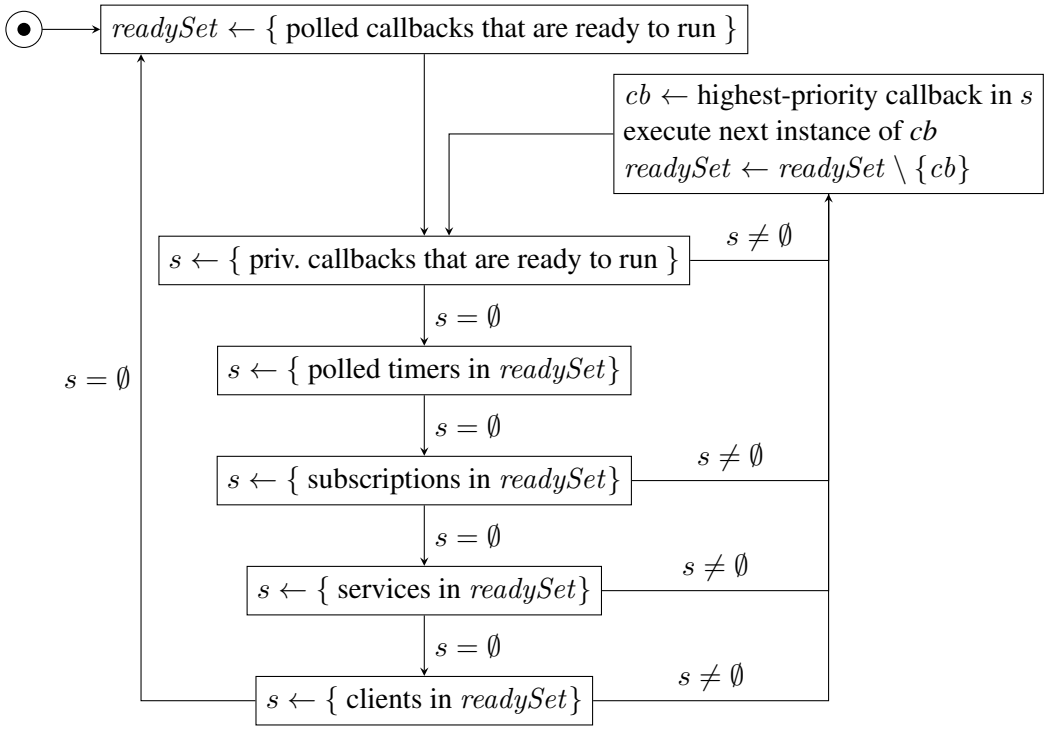


Figure 3.1: The executor scheduling algorithm.

intrinsic effect of assigning each queue a different priority (i.e., the privileged queue is examined first and 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 to the interval between two polling points as a *processing window*.

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 delaying the next polling point.

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 polled callback are ready. It therefore 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-privileged callback instance might be blocked by multiple instances of the same lower-priority callback.

Third, users cannot freely adjust callback priorities to the application's needs. Although it is theoretically possible to control the relative priority within a callback type by registering callbacks in a particular order, there is no way to change the relative priority between two callbacks of different types.

Even within a type, the registration order of the callbacks is restricted by the node structure. When a developer adds a node to an executor, that node's callbacks are immediately registered with the executor. The developer can choose the relative order of the nodes and the relative order of callbacks within a node, but there is no way to register a callback with the executor without registering all other callbacks of the same node as well.

3.1.2 Model Validation

The above description of the ROS scheduler is based on manual code inspection. In a system as complex as ROS 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

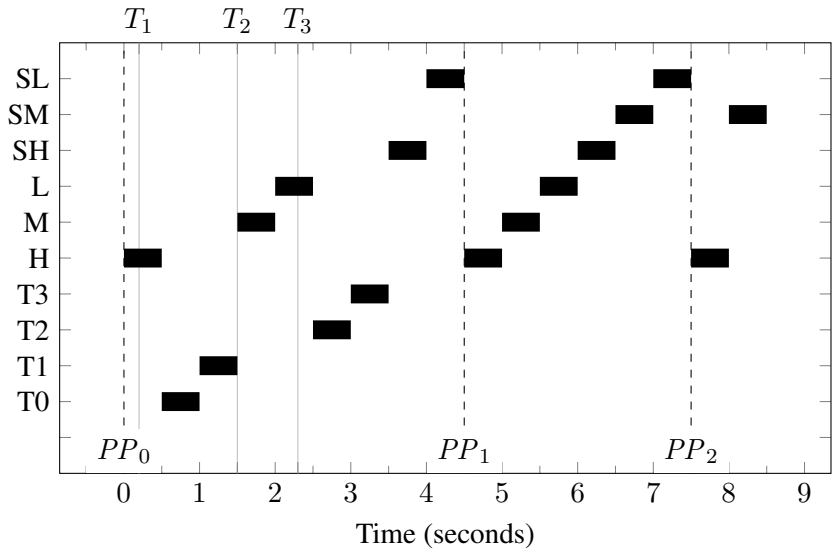


Figure 3.2: Gantt-Chart of the scheduler validation test for ROS 2 "*Dashing Diademata*". At times T_1 and T_3, the timers trigger. At time T_2, the second batch of service requests and messages is submitted.

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_1) and two timers at 2300ms (T_3). It then sends the message sequence $\langle$*L M H SH SL L M H SH SL*$\rangle$, waits for 1.5 seconds (T_2), and then sends $\langle$*SM SM H*$\rangle$. The result is visualized in Fig. 3.2 and Fig. 3.3. The former depicts the behavior in older versions of ROS (here: "*Dashing Diademata*"), the latter depicts the behavior in newer versions of ROS (here: "*Foxy Fitzroy*"). Polling points (*PP*) are marked with vertical dashed lines. Note that the test does not report the location of the polling points; rather, they are inferred from the resulting timing behavior.

Figure 3.2 confirms that the scheduler executes only a single instance of each ready callback, even if multiple messages have been queued up; this is especially apparent between time 4.5 (PP_1) and 7.5 (PP_2), where only one of callback H's two ready instances is executed. The result of this round-robin policy is a characteristic staircase pattern.

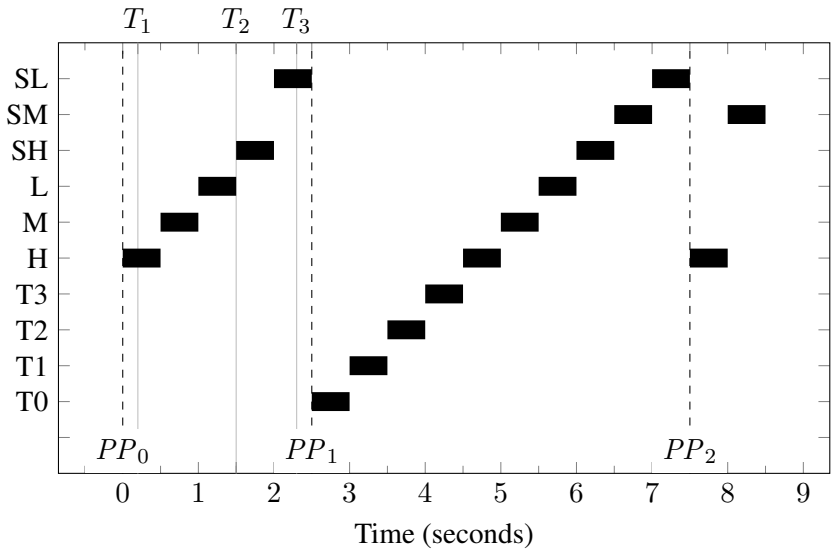


Figure 3.3: Gantt-Chart of the scheduler validation test for ROS 2 "*Foxy Fitzroy*". At times T_1 and T_3, the timers trigger. At time T_2, the second batch of service requests and messages is submitted.

The experiment also confirms that incoming messages need to wait for a polling point before the scheduler takes them into account. For example, service SM is activated for the first time at time 1.5 (T_2) but then visibly skipped at time 4. The SM callback eventually runs at time 6.5, supporting the claim that the executor only learns of SM's arrival at the polling point PP_1.

The timers, however, are not subject to these polling points: both t_2 and t_3 arrive later than *SM* but run before PP_1, i.e., during the first processing window.

Figure 3.3 shows the callback execution order for the same activation sequence in newer ROS versions. The situation of callbacks H and SM looks similar. However, the privileged status of timers is gone. None of the four timers gets to run during the first processing window. Instead, the activations at time T_2 and T_3 are noticed only after the lowest-priority callback of the first processing window completes at time 2.5 (PP_1). Apart from this difference, the callback ordering is the same as in Fig. 3.2, including the staircase pattern characteristic for a round-robin scheduling policy.

3.2 System Model

Having established ROS executor's scheduling algorithm, we are now ready to formulate a real-time task model for ROS systems. We begin with a model for executors only; support for non-executor threads will be added in Section 3.2.3.

We model a ROS system as a set of callbacks $\mathcal{C}$, each of which activates a potentially infinite sequence of instances. Each callback $c_i \in \mathcal{C}$ is statically assigned to one of k single-threaded executors $E_1, \ldots, E_k$; for notational convenience, we let e_i denote callback c_i's assigned executor. We assume a steady-state system (i.e., callbacks neither leave nor join the system at runtime) and a discrete-time model wherein all time parameters are integer multiples of a basic time unit $\epsilon \triangleq 1$ (e.g., a processor cycle).

We divide the set of all callbacks by type and let $\mathcal{C}^{\mathrm{tmr}}$, $\mathcal{C}^{\mathrm{sub}}$, $\mathcal{C}^{\mathrm{srv}}$, and $\mathcal{C}^{\mathrm{clt}}$ denote the set of all timers, subscribers, services, and clients, respectively. Subscribers, services, and clients collectively form the set of *message-driven callbacks* $\mathcal{C}^{\mathrm{msg}}$. We further let $\mathcal{C}_k$ denote the subset of all callbacks assigned to executor E_k, so that for instance $\mathcal{C}_k^{\mathrm{tmr}}$ denotes the subset of all timers assigned to executor E_k. Finally, $lp_k(c_i)$ and $hp_k(c_i)$ denote all callbacks in $\mathcal{C}_k$ with lower or higher priority than c_i, respectively.

We distinguish between the set of *polled callbacks* $\mathcal{C}^{\mathrm{pol}}$ and the set of *privileged callbacks* $\mathcal{C}^{\mathrm{prv}}$, with $\mathcal{C}^{\mathrm{pol}} \cup \mathcal{C}^{\mathrm{prv}} = \mathcal{C}$. Polled callbacks are subject to polling points and have to pass through the *readySet*; privileged callbacks are exempt from polling points. For ROS versions up to "*Dashing Diademata*", all timers are privileged, i.e., $\mathcal{C}^{\mathrm{prv}} \triangleq \mathcal{C}^{\mathrm{tmr}}$. Later ROS versions do not have any privileged callbacks, i.e., $\mathcal{C}^{\mathrm{prv}} \triangleq \emptyset$.

Each callback c_i is described by an *activation curve* $\eta_i(\Delta)$. Recall from Section 2.3.3 that an activation curve upper-bounds the number of activations during an arbitrary interval of length Δ. For timer callbacks, the activation curve must be given as part of the system specification. For message-driven callbacks, activation curves can alternatively be derived from the timing behavior of the messaging callbacks. This derivation process is part of the response-time analysis and will be discussed in Section 4.1.

For notational convenience, we define the activation curve for negative arguments as well, with $\eta_i(\Delta) = 0$ if $\Delta < 0$. We further stipulate that each callback can be activated, i.e., $\eta_i(\epsilon) > 0$. This does not limit the generality of the model, since a callback that is never activated can be removed from the model without any effect on the result.

3.2.1 The ROS Executor

Callbacks are executed by their assigned executor, which itself is scheduled by the OS thread scheduler. The combination of the OS scheduler and the executor thus forms a two-level scheduler hierarchy. The model abstracts the OS scheduler through a supply-bound function $sbf_k(\Delta)$, which makes the model compatible with a wide range of OS schedulers (including the three Linux real-time schedulers). The ROS executor is modeled explicitly as a callback scheduler, which receives supply from the OS scheduler and distributes it among its callbacks according to the algorithm described in Section 3.1.

Since it is unwieldy to reason about the executor behavior in algorithmic form, we abstract from the exact executor algorithm by identifying six properties of any schedule produced by the executor. The timing model refers only to these properties and is oblivious to any other details of the scheduler. The properties are as follows:

SQ *Sequentiality*: Different instances of the same callback are executed in activation order.

NP *Non-Preemptiveness*: The executor selects a new instance only when the previous instance has completed.

WC *Work-Conservation*: The executor never idles if an instance is pending.

PP *Polling Points*: A polling point occurs when the executor needs to select the next instance to execute but no sampled instance is available.

SM *Sampling*: At a polling point, the executor samples up to one instance of each polled callback in activation order. Instances of privileged callbacks are sampled immediately upon activation.

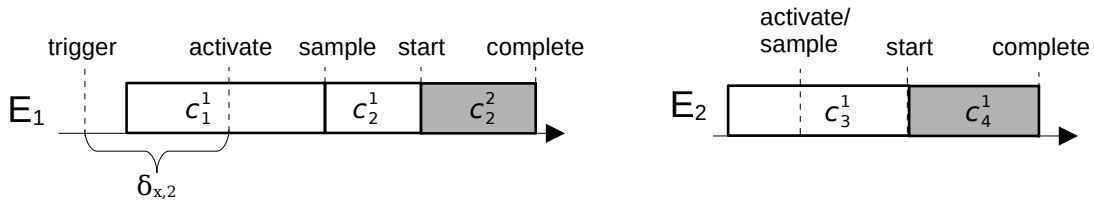


Figure 3.4: Lifecycle of a polled callback c_2 and a privileged callback c_4.

PR *Selective prioritization*: The executor runs sampled instances (and only sampled instances) in priority order.

While the algorithm shares the first three properties with a standard non-preemptive fixed-priority scheduler, the last three properties represent the unique scheduling behavior of the ROS executor.

3.2.2 The Callback Instance Lifecycle

Each callback activates an instance whenever its *activation event* occurs. For timers, the activation event is the start of a new period; for message-driven callbacks, the activation event is the arrival of an *activation message*. The activation message is either a new publication on the subscribed topic in the case of subscribers, a service request to the offered service in the case of service handlers, or a reply to a previous service call in the case of clients. We refer to the k-th instance of a callback c_i as c_i^k. Each instance passes through several phases, which are illustrated in Fig. 3.4.

Conceptually, an instance of a message-driven callback c_i^k is *triggered* when the activation message is sent. The activation message may incur a *propagation delay* while it traverses the ROS stack and potentially the network. Consequently, the instance c_i^k *activates* only some time later when the activation message arrives at c_i's assigned executor e_i. Note that even a small propagation delay may have an outsized effect by forcing c_i^k into the subsequent processing window. We let $\delta_{i,j}$ denote an upper bound on the propagation delay between any two callbacks c_i and c_j. Based on the read-your-own-writes assumption (Section 2.1.2), we further assume that there is negligible propagation delay within the same executor, i.e., $e_i = e_j \Rightarrow \delta_{i,j} = 0$.

Once activated, an instance c_i^k is said to be *pending* until it completes. If c_i is a polled callback

then c_i^k resides in a middleware buffer after activation until it is *sampled* by its executor at a future polling point. Once sampled, c_i^k becomes eligible for execution in the subsequent processing window. After all higher-priority sampled instances have executed, the executor then *selects* c_i^k and runs it until c_i^k *completes*. Due to the polling-point semantics, there is at most one sampled instance of a polled callback at a time.

The lifecycle of a polled, message-driven callback instance is illustrated in the top part of Fig. 3.4, where instance c_2^1 is triggered by a message published to c_2's topic. The message suffers from up to $\delta_{1,2}$ time units of propagation delay and then arrives at E_1, where it activates c_2^1. It is then sampled at the next polling point and executes during the subsequent processing window.

A privileged callback's lifecycle is somewhat simpler since it does not involve a separate sampling step. There is hence no difference between the time of activation and the time of sampling. In contrast to polled callbacks, there can be multiple simultaneously sampled instances of a privileged callback.

The bottom part of Fig. 3.4 illustrates the lifecycle of a privileged timer callback c_4. Executor E_2 immediately samples c_4^1 upon activation, and, once the current instance c_3^1 completes, the executor selects and starts c_4^1 without further delay.

3.2.3 Event Sources

Although the ROS system is built around callbacks and executors, developers may also include non-executor threads in their systems. These threads communicate over the same ROS communication primitives but retain full control over their control flow.

There are two main reasons for including non-executor threads in a ROS system. The first is to react to external hardware. Communicating with devices may require blocking operations, for example a blocking read from a device file. Such operations cannot be implemented with executors and callbacks because ROS does not support triggering callbacks based on file state changes. Component developers usually address this problem by writing a *driver thread*, which reads the required data from the device, interprets it, and sends the interpreted result to a topic.

```
1  rclcpp::Rate loop_rate(freq);
2  while (rclcpp::ok()) {
3    ...
4    loop_rate.sleep()
5  }
6
7
```

```
node = create_node()
timer = node->create_wall_timer(
           seconds(1.0/freq),
           []() { ... });
executor.add_node(node);
while (rclcpp::ok())
  executor.spin();
```

Listing 3.5: Implementation of a time-triggered workload using a *rate* object (left) and a timer callback (right).

A second reason for avoiding executors stems from convenience rather than necessity. In nodes that periodically perform a single computation, having to wrap that computation into a timer callback is inconvenient. Instead, ROS provides so-called *rate* objects, which allow a thread to wait for the beginning of the next period.

An example for this use of *rate* objects can be seen in Listing 3.5. The *rate* object is initialized with a period in Line 1 and used to suspend until the next period in Line 4. The effect is that the computation runs periodically every $1/\text{freq}$ seconds. The same behavior could be achieved with the node-based solution on the right-hand side, albeit at the cost of more boilerplate code.

Crucially, both threads using *rate* objects and driver threads *conceptually* behave like an executor with a single callback that is activated periodically (in the case of rate objects) or in arbitrary, device-dependent patterns (in the case of external devices). The model therefore represents these threads as virtual callbacks called *event sources*. They are associated with an activation curve and execution-time demand like any other callback.

In general, modeling an arbitrary thread as a callback would make unwarranted assumptions about the internal workload scheduling of the event source thread. After all, it seems to imply that the virtual callback is scheduled by the ROS executor scheduling algorithm. The model avoids this assumption by mandating that an event source must always be the only callback within its executor. For a single callback, the executor scheduling policy is equivalent to FIFO scheduling, which matches the execution order of a sequential workload.

Formally, we denote the set of all event sources by $\mathcal{C}^{\text{evt}} \subseteq \mathcal{C}$. Like with timers, the activation

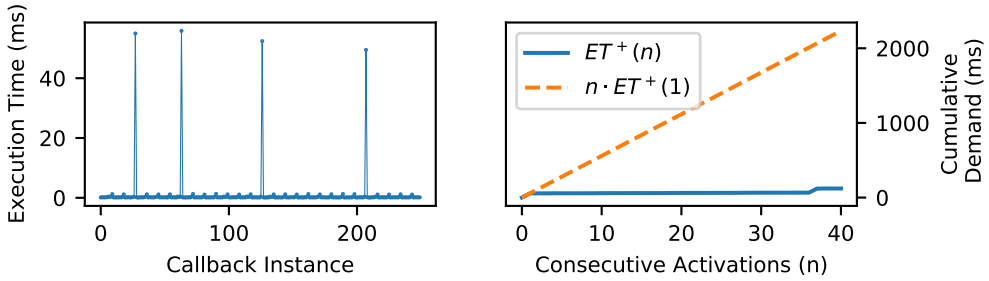


Figure 3.6: Left: 250 observed per-invocation execution times of the `/tf` callback in the *amcl* component. Right: Cumulative demand assumed by the analysis for n consecutive activations under different execution-time models.

curve of each event source must be specified as part of the model. Since an event source thread does not run the executor algorithm, it is a privileged callback. However, since an event source does not share its thread with any other callback, their privilege has no impact on the schedule. Similarly, its priority is irrelevant and arbitrarily defined as zero.

3.2.4 Execution-Time Model

In most of the real-time systems literature, execution-time requirements are modeled as scalar WCETs, meaning that the maximum cost per *single* activation is known, and the joint cost of n consecutive activations is extrapolated as the product of n and the scalar WCET. While this is both safe and convenient from an analysis perspective, it can be overly pessimistic for many ROS applications.

As an example, Fig. 3.6 shows observed execution times of the *amcl* node's `/tf` callback, which is part of the *Navigation 2* package [76]. The callback handles a diverse set of messages, which leads to varying execution-time demands depending on the content of the message. In the depicted trace, the maximum observed cost of a *single* activation was roughly 56 ms. However, it is apparent that the trace follows a pattern where such expensive activations are rare and far apart—any two peaks are separated by many "cheap" activations. Thus, assuming that *every* instance of the callback requires 56 ms would be horrendously pessimistic.

We therefore model the execution-time needs of each callback c_i as a *cumulative execution-time*

curve $ET_i(n)$ [98], which bounds the maximum cumulative execution time of any n consecutive instances of c_i. The classic scalar WCET is hence equivalent to $ET_i(1)$. The resulting gain in precision can be seen in the right-hand inset of Fig. 3.6. The $ET_i(n)$ curve correctly represents that any *single* activation might take up to 56 ms, but also shows that the cumulative execution time of any 40 consecutive executions never exceeded $ET_i(40) = 122$ ms. For the same number of activations, the scalar WCET model would predict a cost of $40 \times ET_i(1) = 2240$ ms.

Execution-time curves must satisfy one main condition: they need to be *super-additive*, *i.e.*, for any two integers m and n, $ET_i(m) + ET_i(n) \geq ET_i(m + n)$.

Execution-time curves are by no means the only model capable of representing multiple kinds of activation with different execution-time requirements. We choose execution-time curves over other established models—such as the multi-frame model [80], the recurring real-time task model [9], or the digraph real-time task model [114]—because execution-time curves can easily be derived from measurements, without requiring any user inputs. This property will be important for the model extractor in Chapter 5.

Executor overheads. In addition to the callbacks' processor demand, the timing model also needs to account for executor overheads. These overheads turned out to be far from negligible: we routinely measured worst-case overheads in the range of 1–2 ms.

We are not the first to notice surprisingly large overheads in the current executor implementation, and the issue was discussed extensively on the ROS Discourse forum [54]. The discussion suggests that there is a large optimization potential in the executor and that the overhead might be reduced in the future. However, in its current state, the executor overhead is clearly non-negligible and needs to be accounted for in the timing model.

We account for this overhead in the execution-time curve through the well-known method of *WCET inflation* (see, *e.g.*, Liu [72]). The worst-case execution time or, in this case, the execution-time curve is inflated with the worst-case overhead. The inflated system can then be safely analyzed without considering overheads.

To this end, each executor is associated with an execution-time curve describing its overheads. We refer to this execution-time curve as the executor's *overhead curve*. For any executor E_k, the overhead curve $OT_k(n)$ bounds the worst-case executor overhead required to execute a sequence of n consecutive callbacks.

The inflated execution-time curve is computed as follows: Let c_i be an arbitrary callback with executor E_k. Let $ET_i^{\text{raw}}(n)$ denote the overhead-oblivious execution-time curve of c_i. Then the execution-time curve of c_i is defined as

$$ET_i(n) \triangleq ET_i^{\text{raw}}(n) + OT_k(n).$$

For executors serving more than one callback, the correctness of this method is not entirely obvious: $OT_k(n)$ bounds the overhead of n callback instances consecutively run by E_k, which might stem from different callbacks. In contrast, $ET_i^{\text{raw}}(n)$ bounds the execution time of n consecutive instances of c_i, which do not necessarily run consecutively. However, the inflation method still overestimates the total processor demand, as the following lemma shows:

Lemma 1. *Let σ be an arbitrary sequence of callback instances scheduled by an executor E_k. For any callback c_i, let n_k denote the number of instances of c_i in σ. Then the total execution-time cost of σ is upper-bounded by $\sum_{c_i \in \mathcal{C}_k} ET_i(n_i)$.*

Proof. Since E_k only runs instances of callbacks assigned to E_k, any callback $c_x \notin \mathcal{C}_k$ does not appear in σ. We thus only need to consider callbacks in $\mathcal{C}_k$. Let $|\sigma|$ denote the number of instances in σ. Then the total execution-time cost of σ is upper-bounded by $OT_k(|\sigma|) + \sum_{c_i \in \mathcal{C}_k} ET_i^{\text{raw}}(n_i)$, *i.e.*, the raw cost of all contained callback instances plus the overhead incurred by $|\sigma|$ instances. Due to super-additivity, the overhead can be split into separate terms for each callback's contribution, *i.e.*, $OT_k(|\sigma|) \leq \sum_{c_i \in \mathcal{C}_k} OT_k(n_i)$. Therefore:

$$OT_k(|\sigma|) + \sum_{c_i \in \mathcal{C}_k} ET_i^{\text{raw}}(n_i) \leq \sum_{c_i \in \mathcal{C}_k} OT_k(n_i) + \sum_{c_i \in \mathcal{C}_k} ET_i^{\text{raw}}(n_i) = \sum_{c_i \in \mathcal{C}_k} ET_i(n_i) \qquad \square$$

3.2.5 Callback Graph

Callbacks can be triggered by a wide range of trigger events. Some callbacks are triggered by messages from other callbacks, like topic updates (subscriptions), service requests (service handlers), or service reply (clients). Other callbacks are triggered periodically (timer callbacks) or by events outside the ROS framework (event sources).

The timing model abstracts from these details and models the trigger relationships among callbacks as a directed graph $\mathcal{D} = \{\mathcal{C}, \mathcal{E}\}$. An edge $(c_i, c_j) \in \mathcal{E}$ encodes that an instance of c_i may trigger one instance of c_j at some point during its execution. We say that the instance of c_j is triggered *along* the edge (c_i, c_j).

We define the set of predecessors and successors associated with each callback c_i as $pred(c_i) = \{c_j \in \mathcal{C} : \exists\, (c_j, c_i) \in \mathcal{E}\}$ and $succ(c_i) = \{c_j \in \mathcal{C} : \exists\, (c_i, c_j) \in \mathcal{E}\}$, respectively.

If a callback c_i has multiple predecessors, it activates an instance whenever *any* of its predecessors triggers it. This corresponds to what is referred to as `OR`-activation in prior work [61].

Which of c_i's successors are triggered by one of c_i's instances depends on which predecessor triggered the instance. Each incoming edge (c_h, c_i) is associated with a *trigger set* $tris_{h,i} \subseteq succ(c_i)$, which specifies the callbacks that may be triggered by any instance of c_i triggered along the (c_h, c_i) edge.

As an example, consider the callback graph in Fig. 3.7. It consists of two timer callbacks (c_1 and c_2). Their activation patterns are described by externally-provided activation curves derived from the respective timer periods. Each instance of c_1 and each instance of c_2 potentially publishes to c_3's topic, which triggers an instance of c_3. Each of these triggered instances may publish to further topics as well, thus potentially triggering instances of c_4 and of c_5 in the process.

The two edges leading to c_3 are associated with the trigger sets $tris_{1,3} = \{c_4, c_5\}$ and $tris_{2,3} = \{c_4\}$. Any instance of c_3 that was triggered by c_2 can thus trigger one instance of c_4 and one of c_5. If the instance of c_3 was triggered by c_1, however, it can trigger only an instance of c_4.

The trigger set can be used to model an internal computation of c_3 that is not visible from the

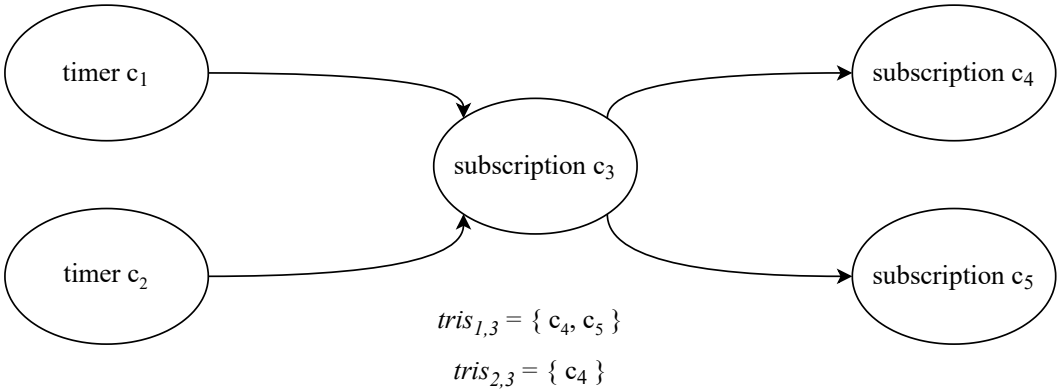


Figure 3.7: A simple callback graph of five callbacks.

callback graph alone. For example, c_3 might inspect an integer field in each incoming message and publish to c_5 only if the integer is above some threshold. If messages from c_2 are always below the threshold, there is no need to account for an activation of c_5 when c_3 is activated by c_2. The trigger set describes this underlying property without requiring an explicit model of the integer field or c_3's internal decision-making.

3.2.6 Processing Chains

In ROS, a single piece of functionality is often realized jointly by the consecutive execution of multiple callbacks. We refer to such a path through the callback graph as a *callback chain*. Given such a chain $\gamma_i = (c_s, \ldots, c_e)$, a *chain instance* $\gamma_i^k = (c_s^{l_1}, \ldots, c_e^{l_k})$ is a sequence of instances of the chain callbacks where each instance triggers the next instance in the sequence. The chain instance activates when its first element activates and completes when its last element completes.

Of particular interest to the analysis are those parts of a chain where multiple consecutive callbacks are assigned to the same executor. We call such fragments *intra-executor subchains* (or *subchains* for short).

3.3 Summary

In this chapter, we have defined a timing model for ROS applications. To this end, we first described the behavior of the ROS callback scheduler and confirmed our observations experimentally (Section 3.1). The described scheduler differs from common real-time schedulers in three major aspects. First, messages arriving during a processing window are not considered until the next polling point. The time of the next polling point depends on *all* remaining callbacks, which makes the duration of the resulting priority inversion hard to predict. Second, the scheduler runs at most one instance of each callback during a processing window. This aggravates the priority inversion, as a backlogged callback might have to wait for *multiple* processing windows until it is even considered for scheduling. Third, users cannot freely adjust callback priorities to the application's needs. As a result, the callback priorities do not necessarily reflect the true priority or urgency of their associated workload.

The proposed timing model (Section 3.2) describes this executor behavior in the form of six scheduler properties. The callbacks in the system are represented as a directed graph whose edges represent triggering relationships. The activation pattern of each callback is represented as an activation curve. In the case of message-driven callbacks, the activation curve need not be specified in the model but can be automatically derived by a response-time analysis. The execution-time demand of each callback and the overhead induced by each executor is represented as an execution-time curve.

As we demonstrate in Chapter 6, the proposed timing model is expressive enough to represent real ROS workloads. In the next chapter, we build upon this model to bound the worst-case response time of timing-sensitive processing chains.

4 A Response-Time Analysis for ROS

Having defined a timing model for ROS, we now turn towards bounding the worst-case latency of ROS callback chains. We begin by adapting the terminology on response-time analysis introduced in Section 2.2.1 to the ROS timing model.

The *worst-case response time* of a callback c_i or a chain γ_i is the largest possible time difference between activation and completion of any instance of c_i (respectively, γ_i). The *response-time bound* $R(c_i)$ (respectively, $R(\gamma_i)$) denotes the upper bound on the worst-case response time of c_i (respectively, γ_i) computed by the analysis.

A callback instance c_i^x is pending during an interval $[t_1, t_2)$ if there is an instant $t \in [t_1, t_2)$ where c_i^x is pending. c_i^x suffers *interference* from another instance c_j^y at time t if c_j^y occupies the shared executor at t and c_i^x is incomplete at t. Interference is *direct* if c_i^x is pending at time t and *indirect* if c_i^x is not yet pending. As a special case, we call interference by prior instances of c_i *self-interference*. A chain instance suffers interference if any of its callback instances suffers interference.

Overall, the analysis follows the response-time analysis approach described in Section 2.3: the analysis alternates between an *activation-curve propagation* step, which propagates each callback's activation pattern to the callback's successors, and a *local analysis* step, which analyzes each executor in isolation to obtain response-time bounds for all callbacks and processing chains served by that executor. Due to the cyclical dependency between activation curves and response times, the analysis keeps alternating between the two steps until a fixed point is reached.

We begin with a discussion of the propagation step (Section 4.1). We continue with two local

analysis approaches, one based on the round-robin-like behavior of the ROS executor (Section 4.2) and one based on the busy-window principle (Section 4.3). Since neither of the two approaches dominates the other, the final analysis uses a combination of both strategies.

4.1 Activation-Curve Propagation

The activation-curve propagation for ROS systems is similar to the traditional activation model propagation rule described in Section 2.3.3. Recall that in a task system where a task τ_j can be activated by its predecessors $pred(\tau_j)$ with propagation delay δ, τ_j's activation curve for any interval Δ can be derived as follows (where R_i denotes τ_i's response time):

$$\eta_j(\Delta) \triangleq \sum_{\tau_i \in pred(\tau_j)} \eta_i(\Delta + R_i + \delta - \epsilon) \qquad \textit{(non-ROS propagation rule)}$$

This propagation mechanism applies almost identically to ROS systems but additionally needs to account for the trigger set of the edges towards callback c_i (cf. Section 3.2.5). An activation of c_i along an edge (c_h, c_i) should be counted in the propagation rule for c_j only if c_j is in the trigger set of the edge. The analysis therefore needs to keep track of which predecessor activated an instance of c_i. To this end, we define the *edge activation curve* $\eta_{i,j}(\Delta)$, which upper-bounds the number of instances of c_j that are activated along the edge (c_i, c_j) during a time window of length Δ.

Definition 1. Let (c_i, c_j) be an arbitrary edge in the graph. If c_i is a message-driven callback, then the *edge activation curve* $\eta_{i,j}(\Delta)$ of the edge (c_i, c_j) is given by

$$\eta_{i,j}(\Delta) \triangleq \sum_{c_h \in pred(c_i)} \begin{cases} \eta_{h,i}(\Delta + R(c_i) - \epsilon + \delta_{i,j}) & \text{if } c_j \in tris_{h,i} \\ 0 & \text{otherwise.} \end{cases}$$

If c_i is not message-driven, then the edge activation curve is given by

$$\eta_{i,j}(\Delta) \triangleq \eta_i(\Delta + R(c_i) - \epsilon + \delta_{i,j}).$$

The callback activation curve is then defined in terms of the edge activation curve as follows:

Definition 2. For any message-driven callback c_j, the *activation curve* $\eta_j(\Delta)$ is given by

$$\eta_j(\Delta) \triangleq \sum_{c_i \in pred(c_j)} \eta_{i,j}(\Delta)$$

The following lemma proves the correctness of the edge activation curve. The correctness of the activation curve then follows as a corollary.

Lemma 2. *Let $\Delta > 0$ and (c_i, c_j) be an arbitrary edge in the graph. Then at most $\eta_{i,j}(\Delta)$ instances of c_j are activated along the (c_i, c_j) edge during any interval of length Δ.*

Proof. By induction over the edges of the callback graph.
Induction base: If c_i is not message-driven, then $\eta_{i,j}(\Delta) = \eta_i(\Delta + R(c_i) - \epsilon + \delta_{i,j})$. This corresponds to the traditional activation model propagation rule described in Section 2.3.3.
Induction step: Now assume that c_i is message-driven instead. Then each of c_i's instances must be activated by one of c_i's predecessors. Consider all instances activated by one such predecessor c_h. If $c_j \in tris_{h,i}$ then each activation along (c_h, c_i) during an arbitrary interval of length Δ may trigger an instance of c_j in the same interval. By the induction hypothesis there are at most $\eta_{h,i}(\Delta)$ such activations. If $c_j \notin tris_{h,i}$ then none of the activations along (c_h, c_i) triggers an instance of c_j. There are thus 0 such activations.

In conclusion, the case distinction in the definition of $\eta_{i,j}(\Delta)$ (Definition 1) upper-bounds the number of activations along the (c_i, c_j) edge that are ultimately triggered by an instance of c_h. Since each instance of c_i is activated by exactly one of its predecessors, the sum of the per-predecessor bound constitutes an overall bound on the number of activations along (c_i, c_j). □

Corollary 1. *Let $\Delta > 0$, and c_j be an arbitrary callback in the graph. Then at most $\eta_j(\Delta)$ instances of c_j are activated during any interval of length Δ.*

Proof. If c_j is not a message-driven callback, then $\eta_j(\Delta)$ is provided as part of the model and bounds the number of activated instances in the time window by definition.

Otherwise, each instance of c_j is activated by exactly one of its predecessors. Since each predecessor c_i activates c_j at most $\eta_{i,j}(\Delta)$ times during any time interval of length Δ, summing $\eta_{i,j}(\Delta)$ for all predecessors upper-bounds the number of c_j's activations. □

The activation-curve propagation step thus proceeds as follows: whenever a new response-time bound is computed by the local analysis, the analysis uses Definition 2 to update the activation curves of all successor callbacks. These updated activation curves are then used for the next iteration of the local analysis.

4.2 Round-Robin Approach

We now turn to the local analysis step, which computes a response-time bound given fixed activation curves. The step identifies the shortest time window during which the supply provided by the executor is guaranteed to exceed the processor time required by the callback or chain under analysis and its interfering callbacks.

We propose two approaches to identify interfering callbacks. In this section, we discuss the *round-robin* approach, which relies on the round-robin-like behavior of the ROS scheduler to bound interference. Specifically, it exploits that due to Property **SM** of the ROS callback scheduler, no more than one instance per polled callback runs in each processing window, independently of its priority or the number of pending instances.

This scheduling approach enforces a notion of fairness among polled callbacks. Consider two polled callbacks, c_1 and c_2. Assume c_1 is triggered periodically and has two pending instances, whereas c_2 is triggered in infrequent bursts and has ten pending instances. Due to Property **SM**, only two instances of c_2 interfere with c_1's two pending instances. A traditional busy-window

analysis would pessimistically account for ten interfering instances instead. The analysis proposed in this section improves upon busy-window approaches by bounding the number of processing windows needed to complete a chain instance.

4.2.1 Interference Bounds

In the following, we exploit Property **SM** to establish bounds on total interference. As a preliminary, Lemma 3 bounds the number of pending callback instances in arbitrary time intervals.

Lemma 3. *Let c_i be any callback. In any interval of length Δ, at most $\eta_i(\Delta + R(c_i) - \epsilon)$ instances of c_i are pending.*

Proof. Consider an arbitrary interval $[t, t + \Delta)$. If $\Delta = 0$, then the bound holds trivially, so assume $\Delta > 0$. Clearly, callback instances activated at or after time $t + \Delta$ are not pending before $t + \Delta$. By definition of the response-time bound $R(c_i)$, instances of c_i activated at or before $t - R(c_i)$ are complete by time t. Thus, only instances activated in $(t - R(c_i), t + \Delta)$ are pending during $[t, t + \Delta)$. The lemma follows since the length of $(t - R(c_i), t + \Delta)$ is $\Delta + R(c_i) - \epsilon$. □

Next, we introduce a bound that exploits Property **SM** to bound the number of callback instances that directly interfere with a callback c_i. The bound depends only on c_i's activation curve and not on the activation curve of the interfering callbacks.

In the following lemmas, let N denote a given upper bound on the number of polling points in an arbitrary interval $[t_1, t_2)$. We later show in Lemma 9 how to obtain such a bound N. For brevity, we let $[\![p]\!]_1$ denote the *indicator function* that evaluates to 1 if the predicate p is true and to 0 otherwise.

Lemma 4. *Let $c_i \in \mathcal{C}_k^{\text{pol}}$ and $c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_i\}$. Let N be a bound on the number of polling points in E_k during an interval $[t_1, t_2)$. If an instance of c_i completes at time t_2, then at most $N + [\![c_j \in hp_k(c_i)]\!]_1$ instances of c_j run during $[t_1, t_2)$.*

Proof. Consider separately the last polling point before time t_2 (called *last polling point* hereafter), and the up to $N - 1$ polling points in $[t_1, t_2)$ that precede the last polling point (called *internal*

polling points hereafter). Each of these polling points samples at most one instance of c_j (Property **SM**). In the case of internal polling points, all such instances run before the next polling point and thus before t_2. In the case of the last polling point, the instance of c_i that completes at t_2 is among the sampled instances. During the processing window, the executor E_k runs only callbacks of higher priority than c_i before c_i (Property **SM**). Thus, instances of callbacks in $lp_k(c_i)$ run after t_2, and instances in $hp_k(c_i) \cup c_i$ run before t_2. There may be further instances running in $[t_1, t_2)$ that have been sampled before t_1. These instances are all sampled at the same polling point, namely the last polling point preceding t_1 (Property **PP**). There is thus at most one such instance per callback (Property **SM**). Overall, c_j executes in $[t_1, t_2)$ up to $N - 1$ instances sampled at internal polling points, at most 1 instance sampled before t_1, and, if $c_j \in hp_k(c_i)$, at most 1 instance sampled at the last polling point, for a total of at most $N + [\![c_j \in hp_k(c_i)]\!]_1$ instances. □

Lemmas 3 and 4 bound the number of callbacks that run during an interval in different ways. The minimum of both bounds is a safe bound, too, as the following corollary notes.

Corollary 2. *Let $c_i \in \mathcal{C}_k^{\mathrm{pol}}$ and $c_j \in \mathcal{C}_k^{\mathrm{pol}} \setminus \{c_i\}$. Let N be an upper bound on the number of polling points in E_k during an interval $[t_1, t_2)$. If an instance of c_i completes at time t_2, then for any $0 \leq \Delta \leq t_2 - t_1$ at most*

$$\min\left(\eta_j(\Delta + R(c_j) - \epsilon),\ \ N + [\![c_j \in hp_k(c_i)]\!]_1\right)$$

instances of c_j run during $[t_1, t_1 + \Delta)$.

Thanks to Corollary 2, we can bound the total amount of direct interference a callback c_i suffers from other callbacks.

Definition 3. For any polled callback c_i assigned to E_k, the *direct interference bound function*

$I_i(\Delta, N)$ is given by

$$I_i(\Delta, N) \triangleq \sum_{c_j \in \mathcal{C}_k^{\text{prv}}} ET_j(\eta_j(\Delta + R(c_j) - \epsilon)) + \sum_{c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_i\}} ET_j(v_j)$$

with $v_j = \min\big(\eta_j(\Delta + R(c_j) - \epsilon), N + [\![c_j \in hp_k(c_i)]\!]_1\big)$.

Lemma 5 shows $I_i(\Delta, N)$ to be sound.

Lemma 5. *Let $c_i \in \mathcal{C}_k^{\text{pol}}$. Let N be an upper bound on the number of polling points in E_k during an interval $[t_1, t_2)$. If an instance of c_i completes at time t_2, then for any $0 \leq \Delta \leq t_2 - t_1$, instances of callbacks in $\mathcal{C}_k \setminus c_i$ consume at most $I_i(\Delta, N)$ units of processor service during $[t_1, t_1 + \Delta)$.*

Proof. First note that, w.r.t. each callback, instances running during $[t_1, t_1+\Delta)$ form a consecutive sequence of instances. Thus, if n instances of an interfering callback c_j run during $[t_1, t_1 + \Delta)$, then their total demand is bounded by $ET_j(n)$. In the case of privileged callbacks, the bound simply exploits that only pending instances can run. Therefore, Lemma 3 yields a bound on the number of callbacks that may be executed in $[t_1, t_1 + \Delta)$. In the case of polled callbacks, Corollary 2 shows that v_j bounds the number of instances that may run in $[t_1, t_1 + \Delta)$. Since each callback in $\mathcal{C}_k$ is either privileged or polled, $I_i(\Delta, N)$ bounds the total demand across all callbacks in $\mathcal{C}_k \setminus \{c_i\}$. □

Next, we bound the direct self-interference caused by earlier instances of the callback under analysis.

Definition 4. For any $c_i \in \mathcal{C}^{\text{pol}}$ and any duration Δ, the *self-interfering instances bound* is given by $si_i(\Delta) \triangleq \max(0, \eta_i(\Delta + R(c_i) - \epsilon) - 1)$.

Lemma 6 shows Definition 4 to be sound.

Lemma 6. *Let c_i be an arbitrary callback and c_i^x one of c_i's instances. During any interval $[t_1, t_1 + \Delta)$ for any $\Delta \geq 0$, at most $si_i(\Delta)$ instances directly self-interfere with c_i^x.*

Proof. By Lemma 3, at most $\eta_i(\Delta + R(c_i) - \epsilon)$ instances of c_i are pending during any interval of length Δ. If $\eta_i(\Delta + R(c_i) - \epsilon) = 0$, no instance of c_i is pending in $[t_1, t_1 + \Delta)$. c_i^x can suffer self-interference only if it is pending at some point in $[t_1, t_1 + \Delta)$. There is thus no self-interference and $si_i(\Delta) \geq 0$ is an upper bound.

If $\eta_i(\Delta + R(c_i) - \epsilon) > 0$ then $\eta_i(\Delta + R(c_i) - \epsilon) = si_i(\Delta) + 1$, meaning that at most $si_i(\Delta) + 1$ instances of c_i are pending during any interval of length Δ. To suffer direct self-interference, c_i^x must be pending at some point in $[t_1, t_1 + \Delta)$. By definition of the response-time bound $R(c_i)$, instances released at or prior to $t_1 - R(c_i)$ complete by time t_1; c_i^x is hence activated after time $t_1 - R(c_i)$. Thus, one of the $si_i(\Delta) + 1$ pending instances is c_i^x itself, which implies that at most $si_i(\Delta)$ instances cause direct self-interference. □

4.2.2 Response-Time Bound

With Definitions 3 and 4 in place, we now bound the response time of any given subchain $\gamma = (c_s, \ldots, c_e)$ that is assigned to executor E_k and ends in a polled callback c_e. Let $\gamma^a = (c_s^x, \ldots, c_e^y)$ be an arbitrary instance of this subchain. Let A denote γ^a's activation time and F its completion time, so that its response time is given by $F - A$.

As a first step towards a response-time bound, we note that throughout $[A, F)$, at least one callback of the chain under analysis is pending.

Lemma 7. *At any time in $[A, F)$, at least one of the callback instances comprising $\gamma^a = (c_s^x, \ldots, c_e^y)$ is pending.*

Proof. Since all callbacks in γ^a are assigned to E_k and the intra-executor propagation delay is zero, each callback instance $c_i^q \in \gamma^a \setminus c_e^y$ is still running when its successor $c_j^w \in \gamma^a$ is activated, with $c_j \in succ(c_i)$. Thus, at least one of the callbacks in γ^a is pending throughout $[A, F)$. □

Since at least one callback instance in γ^a is pending at every polling point during $[A, F)$, each polling point samples at least one instance of a callback in γ, which implies an upper bound on the number of polling points.

Definition 5. For any callback c_i, its *polling-point bound* $pp(c_i)$ is defined as $pp(c_i) \triangleq \eta_i(R(c_i))$ if $c_i \in \mathcal{C}^{\mathrm{pol}}$, and simply as $pp(c_i) \triangleq 0$ otherwise. For a subchain γ, the aggregate bound $pp(\gamma)$ is defined as $pp(\gamma) \triangleq \sum_{c_i \in \gamma} pp(c_i)$.

Lemmas 8 and 9 prove the correctness of these bounds.

Lemma 8. *Let c_i^x be an arbitrary callback instance. Let t_a denote c_i^x's activation time and t_f denote its completion time. There are at most $pp(c_i)$ polling points in $[t_a, t_f)$.*

Proof. If c_i is not a polled callback, c_i^x is sampled immediately upon activation. Since a polling point occurs only if there are no sampled instances (Property **PP**), there can be no polling point in $[t_a, t_f)$. If c_i is a polled callback each polling point in $[t_a, t_f)$ samples one instance of c_i since at least one instance of c_i, namely c_i^x, is pending during the entire interval $[t_a, t_f)$. The number of polling points in $[t_a, t_f)$ is therefore bounded by the number of instances of c_i that are sampled in $[t_a, t_f)$. The last of these instances is c_i^x (Property **PP**), which is pending at t_a. Due to Property **SQ**, any instance of c_i that is sampled at or after t_a but before c_i^x is sampled must be activated before c_i^x, which implies that any such instance is also pending at time t_a. The number of polling points in $[t_a, t_f)$ is thus bounded by the number of instances of c_i pending at time t_a. By Lemma 3 with $\Delta = \epsilon$, at most $\eta_i(R(c_i))$ instances of c_i are pending at time t_a (i.e., during $[t_a, t_a + \epsilon)$). □

Lemma 9. *There are at most $pp(\gamma)$ polling points in $[A, F)$.*

Proof. By Lemma 7, at every time in $[A, F)$ at least one callback of γ^a is pending. Every polling point in $[A, F)$ lies thus between activation and completion of at least one of the callbacks of γ^a (Property **SM**). By Lemma 8, $pp(c_i)$ bounds the number of polling points between the activation and the completion of each callback c_i^y of γ^a. The sum of the individual polling-point bounds of the callbacks comprising γ hence yields an upper bound on the total number of polling points between the activation and completion of γ^a. □

Since γ^a's last callback instance c_e^y completes at time F, $pp(\gamma)$ fulfills the condition on N and the associated interval $[t_1, t_2)$ in Lemmas 4 and 5 and Corollary 2.

In the last preparatory step, we observe a simple structural property of self-interference.

Lemma 10. *Let $c_i^y, \ldots, c_i^{y+n}$ be $n+1$ consecutive instances of a callback c_i. If the last instance c_i^{y+n} runs for ω_i^{y+n} time units, then the first n instances demand at most $\min\big(ET_i(n+1) - \omega_i^{y+n},\ ET_i(n)\big)$ time units of processor service.*

Proof. As $c_i^y, \ldots, c_i^{y+n}$ is a sequence of $n+1$ consecutive callback instances, their overall execution time is bounded by $ET_i(n+1)$. The last element c_i^{y+n}, by assumption, runs for ω_i^{y+n} time units. The first n elements thus run for at most $ET_i(n+1) - \omega_i^{y+n}$ time units. Similarly, $c_i^y, \ldots, c_i^{y+n-1}$ is a sequence of n consecutive callback instances; their total execution time hence is bounded by $ET_i(n)$. □

The next two lemmas finally bound the response time $F - A$ by bounding first the *start time* S of the last callback instance, and then its completion time F. The start time is a useful stepping stone because, from this point on, other callbacks can no longer interfere with γ^a (Property **NP**).

Using the established interference bounds, Lemma 11 finds the latest point in time where c_e^y, the last callback instance of γ^a, must consume its first unit of supply, which bounds S.

Lemma 11. *Let γ^a be an arbitrary instance of subchain γ, let A denote γ^a's activation time, let c_e^y denote the last callback instance in γ^a, suppose that c_e^y requires ω_e^y time units of processor service, and let $N \triangleq pp(\gamma)$. If S^* is the least positive solution (if any) of the inequality*

$$sbf_k(S^*) \geq \epsilon + I_e(S^*, N) + \min\big(ET_e(si_e(S^*) + 1) - \omega_e^y, ET_e(si_e(S^*))\big),$$

then c_e^y starts running in $[A, A + S^*)$.

Proof. By Lemmas 5 and 9, $\epsilon + I_e(S^*, pp(\gamma))$ strictly exceeds the total direct interference due to all callbacks in $\mathcal{C}_k \setminus \{c_e\}$. By Lemma 6, there are at most $si_e(S^*)$ directly self-interfering instances of c_e. Since the self-interfering instances are consecutive, the total interference due to these instances is bounded by Lemma 10 (with $n = si_e(S^*)$). We now show that c_e^y completes in $[A, A + S^*)$. Since A is the activation time of γ^a, by Lemma 7, there is always a pending callback

instance of γ^a until c_e^y completes, which implies that the executor does not idle (Property **WC**). Since S^* satisfies the stated inequality, the amount of service supplied by the executor exceeds the total demand by callback instances directly interfering with c_e^y (either due to other callbacks or self-interference) by at least ϵ units of service. It follows that the only instance that can run while c_e^y is pending without interfering with it is c_e^y itself. Therefore, c_e^y starts running in $[A, A + S^*)$. □

From S^*, we obtain a bound on the response time of γ^a.

Theorem 1. *Let γ^a be an arbitrary instance of subchain γ, let A denote γ^a's activation time, let c_e^y denote the last callback instance in γ^a, and suppose that c_e^y requires ω_e^y time units of processor service. Let S^* be defined as in Lemma 11. If R^* is the least positive solution (if any) of the inequality*

$$sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \omega_e^y,$$

then R^* is a response-time bound for γ (i.e., $F - A \leq R^*$).

Proof. Due to Property **NP**, a callback instance cannot be interfered with once it starts running. Recall from Lemma 11 that c_e^y starts running in $[A, S^*)$, and that it suffers at most $sbf_k(S^*) - \epsilon$ time units of direct interference before it starts running. By assumption, c_e^y runs for ω_e^y time units. Therefore, c_e^y necessarily completes once the executor has provided $sbf_k(S^*) - \epsilon + \omega_e^y$ units of supply. Since R^* satisfies the stated inequality, the executor provides at least $sbf_k(S^*) - \epsilon + \omega_e^y$ units of supply in $[A, A + R^*)$. Consequently, c_e^y completes in $[A, A + R^*)$ and $A + R^* - A = R^*$ is a response-time bound for γ^a. Furthermore, since γ^a is an arbitrary instance of γ upon which we have placed no restrictions, R^* bounds the response time of *any* instance of γ. □

4.2.3 Eliminating ω_e^y

Both S^* and R^* in Theorem 1 depend on ω_e^y, the exact runtime of the last component of the chain under analysis, which is unknown at analysis time. The bound thus cannot be directly applied in a response-time analysis. While ω_e^y can be trivially bounded by 0 from below and by $ET_e(1)$ from

above, such an estimate would be needlessly pessimistic. In the following, we refine Lemma 11 and Theorem 1 to be independent of ω_e^y (hereafter simply referred to as ω).

Since the argument does not depend on details of the interference bounds (and will be reused in Section 4.3), we consider a more general version of the problem. Let $f, g : \mathbb{N} \to \mathbb{N}$ be any two monotonically increasing functions with $f(t) > 0$ and $g(t) \geq 0$ for all t. For a given $\omega \in \mathbb{N}$ such that $0 \leq \omega \leq ET_e(1)$, let $s(\omega)$ denote the least positive $S^* \in \mathbb{N}$ that satisfies

$$sbf_k(S^*) \geq f(S^*) + \min(ET_e(g(S^*)), ET_e(g(S^*)+1) - \omega) \tag{4.1}$$

and let $r(\omega)$ denote the least positive $R^* \in \mathbb{N}$ that satisfies

$$sbf_k(R^*) \geq sbf_k(s(\omega)) - \epsilon + \omega. \tag{4.2}$$

We refer to Ineqs. (4.1) and (4.2) as the *defining inequality* of $s(\omega)$ and $r(\omega)$, respectively.

In the following, we derive a bound on $\max_{\omega \geq 0} r(\omega)$ for arbitrary f and g that is independent of ω. This bound is then applied to Lemma 11 and Theorem 1, which are a special case of the above system of inequalities (with $f(x) \triangleq \epsilon + I_e(x, N)$ and $g(x) \triangleq si_e(x)$). Our argument makes use of the following trivial property of supply-bound functions:

Property 1. An executor cannot provide more than ϵ units of supply in an interval of length ϵ: $\forall x.\ sbf_k(x) \leq sbf_k(x - \epsilon) + \epsilon$.

To get started, we establish that $s(\omega)$ not only satisfies its defining inequality (Ineq. (4.1)), but in fact yields an equality. For brevity, let $z(S^*)$ denote the right-hand side of Ineq. (4.1).

Lemma 12. *If* $0 \leq \omega \leq ET_e(1)$ *and Ineq.* (4.1) *has a positive solution, then* $sbf_k(s(\omega)) = z(s(\omega))$.

Proof. By contradiction: suppose $sbf_k(s(\omega)) > z(s(\omega))$. By Property 1, $sbf_k(s(\omega) - \epsilon) \geq sbf_k(s(\omega)) - \epsilon > z(s(\omega)) - \epsilon$. Since time is discrete, $sbf_k(s(\omega) - \epsilon) > z(s(\omega)) - \epsilon$ implies $sbf_k(s(\omega) - \epsilon) \geq z(s(\omega))$. $s(\omega) - \epsilon$ is thus a solution of Ineq. (4.1), too. Since $s(\omega)$ is by

definition the least *positive* solution of Ineq. (4.1), it follows that $s(\omega) - \epsilon = 0$, which implies $z(s(\omega)) = 0$ since $sbf_k(s(\omega) - \epsilon) = sbf_k(0) = 0$. However, as $f(s(\omega)) > 0$, this implies $\min(ET_e(g(s(\omega))), ET_e(g(s(\omega)) + 1) - \omega) < 0$, which is impossible since $\forall x. ET_e(x) \geq 0$ and $\omega \leq ET_e(1) \leq ET_e(g(s(\omega)) + 1)$. □

The next three lemmas characterize $s(\omega)$ by identifying a value ω^m such that s is constant up to ω^m and monotonically decreasing thereafter. Based on ω^m, we then identify the maximum of $r(\omega)$. We begin by establishing monotonicity.

Lemma 13. *$s(\omega)$ is monotonically decreasing.*

Proof. Since $\omega \in \mathbb{N}$, it suffices to establish $s(\omega + \epsilon) \leq s(\omega)$ for any $\omega \geq 0$. To this end, we show that $S^* = s(\omega)$ is a solution to the defining inequality of $s(\omega + \epsilon)$. Since $s(\omega + \epsilon)$ is, by definition, the least solution to $s(\omega + \epsilon)$'s defining inequality, this implies that $s(\omega) \geq s(\omega + \epsilon)$.

$$
\begin{aligned}
& f(S^*) + \min(ET_e(g(S^*)), ET_e(g(S^*) + 1) - (\omega + \epsilon)) \\
\leq & f(S^*) + \min(ET_e(g(S^*)), ET_e(g(S^*) + 1) - \omega) \\
= & f(s(\omega)) + \min(ET_e(g(s(\omega))), ET_e(g(s(\omega)) + 1) - \omega) \\
\leq & sbf_k(s(\omega)) = sbf_k(S^*) \qquad \{\text{Def. } s(\omega)\}
\end{aligned}
$$
□

Lemma 13 implies that s is maximized at $\omega = 0$. We next observe that there exists a "tipping point" such that the minimum term in Ineq. (4.1) resolves to its first argument below the tipping point and to its second argument otherwise.

Lemma 14. *There is an ω^m such that*

$$\forall \omega, \omega \geq \omega^m \Leftrightarrow ET_e(g(s(\omega))) \geq ET_e(g(s(\omega)) + 1) - \omega.$$

Proof. For brevity, we refer to the right-hand side of the stated equivalence as the ω^m-criterion. There is always at least one value that fulfills the ω^m-criterion, namely $ET_e(1)$, since ET_e is sub-additive ($ET_e(n) \geq ET_e(n+1) - ET_e(1)$ for any n).

Therefore, there also exists a *least* value of ω for which the ω^m-criterion holds. We now show that this least value satisfies the stated equivalence: that is, the value of ω^m is given by the least ω for which the ω^m-criterion holds.

$\Leftarrow$: follows immediately, since ω^m is the least value that satisfies the ω^m-criterion.

$\Rightarrow$: We show that ω *not* fulfilling the ω^m-criterion implies $\omega < \omega^m$. Let ω^* be a value of ω for which $ET_e(g(s(\omega^*))) < ET_e(g(s(\omega^*)) + 1) - \omega^*$. (If no such ω^* exists, then $\omega^m = 0$ and the claim holds trivially.) Then $s(\omega^*)$ fulfills $s(\omega^m)$'s defining inequality: by definition of ω^* and $s(\omega^*)$, we have $sbf_k(s(\omega^*)) \geq f(s(\omega^*)) + ET_e(g(s(\omega^*))$, and since $\forall x. ET_e(g(s(\omega^*)) \geq \min(ET_e(g(s(\omega^*)), x)$, hence also $sbf_k(s(\omega^*)) \geq f(s(\omega^*)) + \min(ET_e(g(s(\omega^*)), ET_e(g(s(\omega^*)) + 1) - \omega^m))$, which is $s(\omega^m)$'s defining inequality. Since $s(\omega^*)$ is positive and $s(\omega^m)$ is the least positive solution of ω^m's defining inequality, this implies $s(\omega^*) \geq s(\omega^m)$, which implies $\omega^* \leq \omega^m$ since s is monotonically decreasing (Lemma 13). Further, $\omega^* \neq \omega^m$ since ω^m fulfills the ω^m-criterion while ω^* does not. Therefore, $\omega^* < \omega$. □

As a result, $s(\omega)$ is a constant function for any $\omega < \omega^m$, which implies that all values in $[0, \omega^m)$ maximize s.

Lemma 15. *If* $0 \leq \omega < \omega^m$*, then* $s(\omega) = s(0)$.

Proof. If $\omega < \omega^m$, then $ET_e(g(s(\omega)) + 1) - \omega > ET_e(g(s(\omega)))$ by Lemma 14. Therefore, $s(\omega)$ is the least positive solution of the inequality $sbf_k(S^*) \geq f(S^*) + ET_e(g(S^*))$, which obviously does not depend on ω. Since by assumption $\omega^m > 0$ (otherwise $\omega < \omega^m$ does not exist since $0 \leq \omega$), we have $s(\omega) = s(0)$ for $0 \leq \omega < \omega^m$. □

Based on Lemmas 12 to 15, we now identify the possible maxima of the r function for $0 \leq \omega \leq ET_e(1)$.

Lemma 16. $\operatorname{argmax}_{0 \leq \omega \leq ET_e(1)} r(\omega) \in \{\omega^m, \omega^m - \epsilon\}$

Proof. We distinguish two cases: $\omega < \omega^m$ and $\omega \geq \omega^m$.

Case 1: If $\omega < \omega^m$, then $s(\omega) = s(0)$ (Lemma 15) and thus $sbf_k(r(\omega)) \geq sbf_k(s(0)) - \epsilon + \omega$. Since sbf_k is monotonically increasing, $r(\omega)$ is hence also monotonically increasing and is thus maximized if ω is maximized, i.e., at $\omega = \omega^m - \epsilon$.

Case 2: If $\omega \geq \omega^m$, then by Lemma 14 the minimum term in Ineq. (4.1) is equal to $ET_e(g(s(\omega))+1) - \omega$. By Lemma 12, Ineq. (4.1) is in fact an equality, which allows us to replace $sbf_k(s(\omega))$ with the right-hand side of Ineq. (4.1). Therefore, $r(\omega)$ is the least positive value satisfying:

$$\begin{aligned} sbf_k(r(\omega)) &\geq sbf_k(s(\omega)) - \epsilon + \omega \\ &= f(s(\omega)) + ET_e(g(s(\omega)) + 1) - \omega - \epsilon + \omega \\ &= f(s(\omega)) + ET_e(g(s(\omega)) + 1) - \epsilon. \end{aligned}$$

$r(\omega)$ is thus monotonically increasing in $s(\omega)$ (as sbf_k, f, g, and ET_e are all monotonically increasing). Since $s(\omega)$ is monotonically *decreasing* (Lemma 13), $s(\omega)$ is maximized if ω is minimized, i.e., at $\omega = \omega^m$. □

Based on the maxima of $r(\omega)$, the next lemma provides two simplified inequalities that no longer depend on ω.

Lemma 17. *If S^* is the least positive solution of*

$$sbf_k(S^*) \geq f(S^*) + ET_e(g(S^*)), \tag{4.3}$$

$\Omega \triangleq ET_e(g(S^*) + 1) - ET_e(g(S^*))$, and R^* is the least positive solution of

$$sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \Omega, \tag{4.4}$$

then $R^* \geq \max_{0 \leq \omega \leq ET_e(1)} r(\omega)$.

Proof. We first show that S^* upper-bounds both $s(\omega^m)$ and, if $\omega^m > 0$, $s(\omega^m - \epsilon)$. If $\omega^m = 0$, then, by Lemma 14, $ET_e(g(s(\omega^m))) \geq ET_e(g(s(\omega^m)) + 1)$. Since ET_e is monotonically

increasing, this implies $ET_e(g(s(\omega^m))) = ET_e(g(s(\omega^m)) + 1)$. As a result, Ineq. (4.3) is equivalent to the defining inequality of s (Ineq. (4.1)) for $\omega = \omega^m$ and S^* is therefore equal to $s(\omega^m)$. If $\omega^m > 0$, Ineq. (4.3) is equivalent to the defining inequality of s (Ineq. (4.1)) for $\omega = \omega^m - \epsilon$ (Lemma 14), and hence $S^* = s(\omega^m - \epsilon)$. As s is monotonically decreasing (Lemma 13), this implies $S^* \geq s(\omega^m)$.

We now show that $\Omega \geq \omega^m$. If $\omega^m = 0$, then Ω is trivially an upper bound, so assume $\omega^m > 0$ and thus $S^* = s(\omega^m - \epsilon)$. By the definition of ω^m, it then holds that:

$$
\begin{aligned}
& ET_e(g(s(\omega^m - \epsilon))) < ET_e(g(s(\omega^m - \epsilon)) + 1) - (\omega^m - \epsilon) \\
\Leftrightarrow\ & ET_e(g(S^*)) < ET_e(g(S^*) + 1) - (\omega^m - \epsilon) \\
\Leftrightarrow\ & (\omega^m - \epsilon) < ET_e(g(S^*) + 1) - ET_e(g(S^*)) \\
\Leftrightarrow\ & \omega^m < ET_e(g(S^*) + 1) - ET_e(g(S^*)) + \epsilon \\
\Leftrightarrow\ & \omega^m < \Omega + \epsilon \Leftrightarrow \omega^m \leq \Omega
\end{aligned}
$$

Finally, we establish that R^* upper-bounds both $r(\omega^m)$ and, if $\omega^m > 0$, $r(\omega^m - \epsilon)$ by showing that R^* fulfills r's defining inequality (Ineq. (4.2)) for $\omega^* \in \{\omega^m, \omega^m - \epsilon\}$.

$$
\begin{aligned}
& sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \Omega \\
\Rightarrow & sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \omega^* && \{\Omega \geq \omega^* \geq \omega^* - \epsilon\} \\
\Rightarrow & sbf_k(R^*) \geq sbf_k(s(\omega^*)) - \epsilon + \omega^* && \{S^* \geq s(\omega^*)\}
\end{aligned}
$$

Thus, by Lemma 16, $R^* \geq r(\omega)$ for $0 \leq \omega \leq ET_e(1)$. □

Having solved the general case, we now apply Lemma 17 to the response-time analysis in Lemma 11 and Theorem 1.

Theorem 2. *Let γ^a be an arbitrary subchain instance. Let A be γ^a's activation time, let F be γ^a's completion time, and let c_e^y denote the last callback instance in γ^a. Let S^* be the least*

positive solution (if any) of the following inequality.

$$sbf_k(S^*) \geq \epsilon + I_e(S^*, pp(\gamma)) + ET_e(si_e(S^*)) \tag{4.5}$$

Let $\Omega \triangleq ET_e(si_e(S^*) + 1) - ET_e(si_e(S^*))$*, and let* R^* *be the least positive solution (if any) of the following inequality.*

$$sbf_k(R^*) \geq sbf_k(S^*) - \epsilon + \Omega \tag{4.6}$$

Then R^* *is a response-time bound for* γ*:* $F - A \leq R^*$*.*

Proof. Follows from Lemma 17 for $f(x) \triangleq \epsilon + I_e(x, N)$ and $g(x) \triangleq si_e(x)$, Lemma 11, and Theorem 1. □

Since Theorem 2 does not depend on ω_e^y, it is suitable for an *a priori* response-time analysis. Specifically, an implementation can find the least solution S^* through fixed-point iteration and then compute a response-time bound. If a solution S^* cannot be found, then Theorem 2 is not applicable.

Furthermore, since the computation of γ^a's response-time depends on other response-time bounds in a cyclical fashion (e.g., Definitions 3 and 4), another outer fixed-point iteration is necessary [61] until the response-time bounds for all callbacks have reached a global fixed point (or until some predefined threshold is exceeded). This process always terminates since the response-time estimates never decrease during the search.

4.3 Busy-Window Approach

We now integrate the busy-window principle with the preceding analysis. A key benefit of considering an entire busy window at a time is that it enables more precise callback activation curves, as we show in this section.

We begin with some preliminary definitions. The definitions are conceptually identical to the standard terms (cf. Section 2.3.2) but are adapted to the ROS timing model.

We say that a callback instance is *carried in* at time t if it is pending at both t and $t-1$. An instant t is a *quiet time* of executor E_k if no instances assigned to E_k are carried-in. An interval $[t_1, t_2)$ is a *busy window* w.r.t. a callback instance c_i^x if both t_1 and t_2 are quiet times of c_i^x's executor, no quiet time of c_i^x's executor occurs between t_1 and t_2, and c_i^x is activated in $[t_1, t_2)$.

Since the propagation delay within executors is zero, all components of a subchain instance γ^x share the same busy window. We thus define the busy window of a chain instance γ^x as the busy window of its components.

We again refer to the subchain instance under analysis as γ^a, which is activated at time A and completes at time F. For simplicity (and w.l.o.g.), we assume that the time axis is normalized such that γ^a's busy window starts at time zero.

Under the busy-window assumption, the analysis can locally improve upon the generic activation-curve propagation method in Section 4.1 in the special case of intra-executor edges. The analysis exploits that no instance served by γ_x's executor is carried in at time 0. If two callbacks c_i and c_j are assigned to γ_x's executor, then instances of c_i activated before time 0 cannot activate instances of c_j at or after time 0.

This insight is captured in the following lemma:

Lemma 18. *Let $\Delta > 0$ and (c_i, c_j) be an arbitrary edge in the graph. If time 0 is a quiet time of c_j's executor, and c_i and c_j are assigned to the same executor, then any instance of c_j that is activated along the (c_i, c_j) edge during $[0, \Delta)$ was triggered by an instance of c_i that was also activated in $[0, \Delta)$.*

Proof. Since c_i and c_j share an executor the propagation delay is zero. Therefore, only instances of c_i that are pending in $[0, \Delta)$ can trigger an instance of c_j during $[0, \Delta)$.

Since $[0, \Delta)$ is a busy window of c_i's and c_j's shared executor, no instance of c_i is carried in at time 0. Thus, all instances of c_i pending in $[0, \Delta)$ are also activated in $[0, \Delta)$. □

As a result, the number of instances activated along any intra-executor edge (c_i, c_j) during the interval $[0, \Delta)$ is upper-bounded by the number of instances activated in $[0, \Delta)$, not those

activated in $[-(R(i) - \epsilon), \Delta)$ as assumed by the general edge activation-curve in Definition 1. Definition 6 defines a specialized edge activation curve exploiting that exploits this property within γ_a's executor and falls back to $\eta_{i,j}(\Delta)$ otherwise.

Definition 6. For any message-driven callback c_j assigned to γ_a's executor and any predecessor c_i of c_j, the *busy-window edge activation curve* $\eta^b_{i,j}(\Delta)$ is given by

$$\eta^b_{i,j}(\Delta) \triangleq \sum_{c_h \in pred(c_i)} \begin{cases} \eta^b_{h,i}(\Delta) & \text{if } e_i = e_j \wedge c_j \in tris_{h,i} \\ 0 & \text{if } e_i = e_j \wedge c_j \notin tris_{h,i} \\ \eta_{i,j}(\Delta) & \text{otherwise.} \end{cases}$$

If c_i is not a message-driven callback, then $\eta^b_{i,j}(\Delta) \triangleq \eta_i(\Delta)$.

The per-callback case is analogous to the general case (Definition 2) but defers to the busy-window edge activation curve instead of the general edge activation curve.

Definition 7. For a message-driven callback c_j assigned to γ_a's executor the *busy-window activation curve* $\eta^b_j(\Delta)$ is given by

$$\eta^b_j(\Delta) \triangleq \sum_{c_i \in pred(c_j)} \eta^b_{i,j}(\Delta).$$

If c_j is not a message-driven callback, then $\eta^b_j(\Delta) \triangleq \eta_j(\Delta)$ instead.

The following lemma proves the correctness of the edge activation bound; the correctness of the callback activation bound follows as a corollary.

Lemma 19. *Let $\Delta > 0$, $c_j \in \mathcal{C}_k$, and $c_i \in pred(c_j)$. If time 0 is a quiet time of c_j's executor, then at most $\eta^b_{i,j}(\Delta)$ instances of c_j are activated along the (c_i, c_j) edge during $[0, \Delta)$.*

Proof. By induction over the edges in the callback graph.

Induction base: If c_i is not message-driven, $\eta_j^b(\Delta) = \eta_i(\Delta)$, which bounds the number of activations of c_i in $[0, \Delta)$. By Lemma 18, this also upper-bounds the number of activations of c_j along the (c_i, c_j) edge.

Induction step: Consider any edge (c_i, c_j). If c_i and c_j do not share an executor, then $\eta_{i,j}^b(\Delta) = \eta_{i,j}(\Delta)$, which upper-bounds the number of activations along the edge by Lemma 2.

If c_i and c_j share an executor, then any activation of c_j along the (c_i, c_j) edge during $[0, \Delta)$ is triggered by an instance of c_i that was activated in $[0, \Delta)$ (Lemma 18). The remainder of the argument proceeds analogously to Lemma 2. Each instance of c_i activated in $[0, \Delta)$ is triggered by one of c_i's predecessors. Consider one such predecessor c_h. If $c_j \in tris_{h,i}$ then all of these activations may trigger an instance of c_j. By the induction hypothesis this happens up to $\eta_{i,j}(\Delta)$ times. If $c_j \notin tris_{h,i}$ then none of the activations along (c_h, c_i) triggers an instance of c_j.

In conclusion, the case distinction in the definition of $\eta_{i,j}^b(\Delta)$ (Definition 6) upper-bounds the number of activations along the (c_i, c_j) edge that were indirectly triggered by an instance of c_h. Since each instance of c_i is activated by exactly one of its predecessors, the sum of the per-predecessor bound constitutes an overall bound on the number of activations along (c_i, c_j) during $[0, \Delta)$. □

The correctness of $\eta_j^b(\Delta)$ follows as a corollary.

Corollary 3. *Let $\Delta > 0$, and c_j be an arbitrary callback in the graph. Then at most $\eta_j^b(\Delta)$ instances of c_j are activated during the interval $[0, \Delta)$.*

Proof. If c_j is not message-driven, then $\eta_j^b(\Delta) = \eta_j(\Delta)$, which bounds the number of activations of c_j due to Corollary 1.

If c_j is message-driven, then each instance of c_j is activated by exactly one of its predecessors. Since each predecessor c_i activates c_j at most $\eta_{i,j}^b(\Delta)$ times during the time interval, summing $\eta_{i,j}^b(\Delta)$ for all predecessors upper-bounds the number of c_j's activations. □

4.3.1 Response-Time Bound

The improved activation curve replaces η_i in the interference bound to reduce pessimism. In addition to Δ (the length of the interval to consider) and N (a bound on the number of polling points), the function takes c_i^x's activation time t_a as a third parameter. It then combines two ways to bound the number of instances of a polled callback c_j during $[0, \Delta)$. The first way is to bound all activations in $[0, \Delta)$ as $\eta_j^b(\Delta)$, which exploits that the considered interval starts at a quiet time but does not exploit Property **SM**. The second way is to bound all activations in $[0, t_a)$ as $\eta_j^b(t_a)$, and to then bound the interference in $[t_a, \Delta)$ (i.e., after c_i^x's activation) under consideration of Property **SM**.

Definition 8. For any polled callback c_i assigned to E_k, the *busy-window interference bound* is given by

$$I_i^b(\Delta, N, t_a) \triangleq \sum_{c_j \in \mathcal{C}_k^{\mathrm{prv}}} ET_j(\eta_j^b(\Delta)) + \sum_{c_j \in \mathcal{C}_k^{\mathrm{pol}} \setminus \{c_i\}} \min \begin{cases} ET_j(\eta_j^b(\Delta)) \\ ET_j(v_j) \end{cases}$$

with $v_j = \eta_j^b(t_a) + N + [\![c_j \in hp_k(c_i)]\!]_1$.

Lemma 20 proves the bound to be sound.

Lemma 20. *Let $c_i \in \mathcal{C}_k^{\mathrm{pol}}$. Let t_a and t_f denote the activation time and completion time of an instance of c_i. Let N upper-bound the number of polling points in E_k during the interval $[t_a, t_f)$. If time 0 is a quiet time of E_k, then for any $0 \leq \Delta \leq t_f$, instances of callbacks in $\mathcal{C}_k \setminus \{c_i\}$ consume at most $I_i^b(\Delta, N, t_a)$ units of processor service during $[0, \Delta)$.*

Proof. Since I_i^b sums over all callbacks in $\mathcal{C}_k$, it suffices to show that both $\eta_j^b(\Delta)$ and, in the case of polled callbacks, v_j bound how many instances of each interfering callback c_j run during $[0, \Delta)$. Since 0 is a quiet time of E_k, the number of callbacks that run during $[0, \Delta)$ is upper-bounded by the number of callbacks activated during $[0, \Delta)$.

By Lemma 19, $\eta_j^b(\Delta)$ bounds how many instances of c_j are activated in the interval $[0, \Delta)$ (since, by assumption, time 0 is a quiet time of E_k).

It remains to be shown that v_j is a valid upper bound, too. Lemma 19 again shows that $\eta_j^b(t_a)$ bounds how many instances of c_j are activated during $[0, t_a)$. By Lemma 4 and Corollary 2, $N + [\![c_j \in hp_k(c_i)]\!]_1$ upper-bounds the number of instances running in $[t_a, t_f)$. Their sum therefore bounds the number of instances running in $[0, t_a) \cup [t_a, t_f) = [0, t_f) \supseteq [0, \Delta)$. □

Analogously to Definition 4, we also define a busy-window-aware self-interference bound $si_i^b(t_a) \triangleq \eta_i^b(t_a + \epsilon) - 1$, using $si_i^b(t_a)$ in place of $si_i(\Delta)$ to leverage the busy-window activation curve $\eta_i^b(\Delta)$. Unlike in Definition 4, no $\max(0, \ldots)$ term is required since $t_a + \epsilon > 0$ and therefore $\eta_i^b(t_a + \epsilon) \geq 1$.

Put together, the two new interference bounds yield a result similar to Theorem 2.

Theorem 3. *Let γ^a be an arbitrary instance of subchain γ. Let A denote γ^a's activation time, F its completion time, c_e^y the last callback instance in γ^a, and t_a the activation time of c_e^y. If time 0 is a quiet time of γ's executor, S^* is the least positive solution (if any) of*

$$sbf_k(S^*) \geq \epsilon + I_e^b(S^*, pp(\gamma), t_a) + ET_e(si_e^b(t_a)),$$

$\Omega \triangleq ET_e(si_e^b(t_a) + 1) - ET_e(si_e^b(t_a))$, and F^* is the least positive solution (if any) of the inequality

$$sbf_k(F^*) \geq sbf_k(S^*) - \epsilon + \Omega,$$

then $F \leq F^*$ and $F^* - A$ is a response-time bound for γ.

Proof. Recall there are at most $pp(\gamma)$ polling points in $[A, F)$ (Lemma 9), hence also in $[t_a, F) \subseteq [A, F)$ ($t_a \geq A$, since c_e^y is part of γ^a). Due to Lemma 20, $\epsilon + I_e^b(S^*, pp(\gamma), t_a)$ hence strictly exceeds the total interference due to all callbacks in $\mathcal{C}_k \setminus \{c_e\}$ during $[0, S^*)$. Since no instance of c_e is carried in at time 0, $si_e^b(t_a)$ bounds the number of instances of c_e except c_e^y that are pending (and hence self-interfering) in $[0, t_a]$. By Lemma 10, the total self-interference is then given

by $\min\big(ET_e(si_e^b(t_a)+1)-\omega_e^y,\, ET_e(si_e^b(t_a))\big)$, where ω_e^y is c_e^y's execution cost. Analogously to the proofs of Lemma 11 and Theorems 1 and 2, the claim then follows via Lemma 17 with $f(x) \triangleq \epsilon + I_e^b(x, pp(\gamma), t_a)$ and $g(x) \triangleq si_e^b(t_a)$. □

Theorem 3 yields a response-time bound, namely $F^* - A$, which however depends on two unknown offsets, A and t_a. In the case where γ consists of a single callback, A is equal to t_a and does not need to be considered separately. Otherwise, it suffices to consider $A = 0$; A does not influence the computation of F^*, and $A = 0$ therefore maximizes the term $F^* - A$. We next derive a sparse, finite set of offsets that suffice to be considered.

4.3.2 The Search Space for the Activation Offset t_a

To start, Lemma 21 derives an upper bound on t_a.

Lemma 21. *Let c_i be an arbitrary callback and let t_a^* denote the least positive solution (if any) of*

$$sbf_k(t_a^*) \geq \epsilon + I_i^b(t_a^*, pp(c_i), t_a^*) + ET_i(\eta_i^b(t_a^*)).$$

Then any instance of c_i is activated strictly less than t_a^* time units after the beginning of its busy window.

Proof. By contradiction: suppose an instance c_i^a is activated at time $t_a \geq t_a^*$. W.l.o.g. let time 0 denote the start of c_i^a's busy window. By Lemma 20, $I_i^b(t_a^*, pp(c_i), t_a)$ bounds the total interference that callbacks other than c_i impose upon c_i^a during the time window $[0, t_a^*)$. $ET_i(\eta_i^b(t_a^*))$ bounds the demand that c_i imposes upon c_i^a during $[0, t_a^*)$. By the inequality in the lemma statement, the supply available to callbacks served by E_k during $[0, t_a^*)$ then necessarily exceeds the possible total demand during that time. Since c_i^a is not pending before $t_a \geq t_a^*$, this implies that the executor must idle at some point during $[0, t_a^*)$, i.e., there is a quiet time in $(0, t_a^*)$. Since c_i^a's busy window starts at time 0, this implies that c_i^a's busy window ends before c_i^a's own activation, which is a contradiction. Therefore, any instance c_i^a activates at most $t_a^* - \epsilon$ time units after the beginning of its busy window. □

To obtain a sparse search space, the next lemma identifies that only "steps" in the η_j^b bounds for the callbacks in E_k need to be considered to find a response-time bound.

Lemma 22. *Let c_i^x be an arbitrary instance of c_i and let t_a denote c_i^x's activation time. If $t_a > 0$,*

$$\forall c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_i\} \,.\, \eta_j^b(t_a) = \eta_j^b(t_a - \epsilon), \textit{ and}$$

$$\eta_i^b(t_a) = \eta_i^b(t_a + \epsilon),$$

then the response-time bound (given by Theorem 3) for $\gamma^a = \langle c_i^x \rangle$ is lower than that for an instance activated at time $t_a - \epsilon$.

Proof. Let c_i^w be an instance of c_i activated at time $t_a - \epsilon$. We show that $I_j^b(\Delta, N, t_a) = I_j^b(\Delta, N, t_a - \epsilon)$ and $si_i^b(t_a) = si_i^b(t_a - \epsilon)$. Hence F^* as defined in Theorem 3 (with γ being the chain consisting of only c_i) is the same for c_i^w and c_i^x. The lemma follows since $F^* - t_a < F^* - (t_a - \epsilon)$.

In the definition of $I_i^b(\Delta, N, t_a)$ only the term $ET_j(v_j)$ depends on the t_a parameter. This term appears for any $c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_i\}$. Since for each such c_j by assumption $\eta_j^b(t_a) = \eta_j^b(t_a - \epsilon)$, it follows that v_j is equal for both activation times, t_a and $t_a - \epsilon$. Therefore, $I_j^b(\Delta, N, t_a) = I_j^b(\Delta, N, t_a - \epsilon)$.

In the definition of $si_i^b(t_a)$, only the term $\eta_i^b(t_a + \epsilon)$ depends on t_a. Since $\eta_i^b(t_a + \epsilon) = \eta_i^b(t_a)$ it follows that $si_i^b(t_a) = si_i^b(t_a - \epsilon)$ □

The search space for activation offsets of a callback c_i is thus defined as

$$\mathcal{A} \triangleq \{0\} \cup \{t_a < t_a^* \mid \ \eta_i^b(t_a) \neq \eta_i^b(t_a + \epsilon) \ \vee \exists c_j \in \mathcal{C}_k^{\text{pol}} \setminus \{c_i\} : \eta_j^b(t_a) \neq \eta_j^b(t_a - \epsilon)\}.$$

The analysis needs to consider only $\mathcal{A}$ as possible values of t_a. The response-time bound for the subchain under analysis is then given by the maximum result obtained via Theorem 3 for each $t_a \in \mathcal{A}$. If t_a^*, or either of the fixed point solutions F^* and S^* for any t_a, cannot be found, then Theorem 3 is not applicable.

4.3.3 Combined Analysis

Theorems 2 and 3 are independent analyses that should be used jointly: as neither dominates the other, it is generally advisable to apply both analyses to each subchain and to use the lesser of the two bounds on a per-subchain basis.

In the derivations of Theorems 2 and 3, no assumptions have been placed on the number of callbacks in the chain under analysis γ. Both analyses can therefore also be used to bound the response time of an individual polled callback by interpreting the callback under analysis as a single-element chain.

4.4 Summary

In this chapter, we have introduced two strategies to bound the worst-case response time of a ROS callback or processing chain. The first exploits the round-robin-like nature of the ROS executor; the second uses the busy-window principle. A busy-window-aware activation-curve propagation rule further improves analysis precision for intra-executor subchains.

The evaluation in Chapter 6 confirms that none of the two approaches dominates the other. The round-robin approach is more resistant to bursty callbacks but cannot exploit the busy-window principle. On the other hand, the busy-window approach does exploit the busy-window principle but yields much more pessimistic estimates if the callback under analysis is interfered with by a bursty callback. It is therefore advisable to combine both approaches.

5 An Automatic Latency Manager for ROS

The response-time analysis described in the previous chapters requires a detailed timing model of the ROS system to compute response-time bounds. Unfortunately, constructing such a timing model for ROS systems can be difficult in practice.

A major challenge is that ROS systems are usually not developed by a single person or group. After all, it is one of the main selling points of ROS that third-party components can be integrated easily. As a result, the various components comprising a single ROS system are usually developed in isolation by multiple independent *component developers* who do not necessarily know (of) each other. Similarly, the *system integrator*, who composes the selected components on a deployment platform with application- and mission-specific logic and "glue code," usually does not coordinate closely with the respective component developers.

In such a setup, no single actor has all the information required to perform a response-time analysis. Real-time analysis presumes in-depth knowledge of many low-level system details such as the number of concurrent tasks, their activation semantics and functional interactions, arrival patterns of messages, worst-case execution times, *etc.* Since ROS components do not come with a manifest that would provide this kind of information, system integrators would have to reverse-engineer these details from the components. This is a risky endeavor, considering how poorly real-time analyses cope with faulty or incomplete information. A single mistake or oversight while manually reverse-engineering a third-party component for modeling purposes could silently invalidate the entire effort.

The component developer does not have the required information either. The timing behavior of

individual components depends too much on the wider system context, which only the integrator knows. One reason is that many robotics algorithms exhibit vastly varying execution times and activation patterns that depend on use-case- and platform-specific aspects. For example, consider a generic object-tracking component that identifies objects in a video stream and infers their trajectories (*e.g.*, cars in a neighboring lane). The execution time of this functionality depends on a variety of parameters: the frame rate, resolution, and codec of the video stream as well as various other parameters related to the specific tracking algorithm. None of these parameters can be known or fixed upfront by the developer of a generic object-tracking component. Such use-case-specific information is only known to the integrator building a specific robot, who in turn is not necessarily an expert in object tracking or real-time systems and thus cannot always predict the impact of specific configuration choices. The resource demands and real-time behavior of a component must therefore always be evaluated in the context of its use in a specific deployment, which is not compatible with the modular reuse of "black box" components that underlies the popularity of the ROS framework.

Yet even if the integrator were to discuss each component with the respective experts and would somehow obtain all details necessary for a timing analysis, a second fundamental problem remains: the resource requirements and performance characteristics of many components inherently depend on a robot's dynamic environment and thus vary over time, rendering a static (worst-case) resource provisioning infeasible.

For instance, consider the object-tracking component again, and suppose the robot also relies on a landmark-based self-localization component. On the one hand, the object tracker will demand much more processor time moving through a bustling city than through sparsely populated countryside. On the other hand, self-localization is likely much easier in a city with its many recognizable landmarks than in a mostly uniform landscape. To guarantee sufficient resources in both situations, the system integrator would have to provision the system for bustling cities consisting of barren countryside.

In robotics, such pessimistic system dimensioning is bound to run into practical limitations.

Instead, to remain practical and cost-efficient, robotics systems must be provisioned for the expected peak *joint* resource demands, rather than the sum of each component's individual peak demands.

In this chapter, we propose to overcome these challenges with an *automatic latency manager*, which automatically and dynamically re-provisions the system at runtime. To this end, we present the ***ROS L**ive **la**tency **ma**nager* (*ROS-Llama*), an automatic latency manager based on the timing model and response-time analysis of the previous chapters. ROS-Llama automatically extracts a timing model from the running system and uses response-time analysis to compute scheduling parameters that ensure the user's latency goals are fulfilled.

ROS-Llama allows system integrators to control the worst-case latency of timing-critical processing chains while avoiding the problems identified above. Its automation allows system integrators to exploit a detailed and dynamic system timing model, which neither the system integrator nor individual component developers could provide on their own.

Unlike the system integrator, ROS-Llama does not have to ask the component developer about intricate system parameters but relies solely on introspection and automatically determines all required timing model parameters at runtime.

Unlike the component developer, ROS-Llama has a holistic view of the system and measures all components in their specific context. Consider again an object-tracking component whose timing behavior depends on application-specific properties of the video stream. The measurements reflect the one configuration that is actually in use, no matter how many different configurations and parameters a component supports in principle.

Furthermore, ROS-Llama can dynamically adapt its models and scheduling decisions over time. Unlike a fixed deployment, it can thus adapt to dynamically-changing environments and software components.

5.1 Requirements and Constraints

ROS-Llama's design is shaped by various implicit and explicit requirements that an automatic latency manager needs to fulfill. Specifically, we identify the following nine requirements:

(I) Form does not follow function. It is common in the classic real-time literature to assume that a system's functionalities and requirements are neatly reflected in its implementation as a set of executable tasks at the OS level. Correspondingly, central notions such as response time, priority, and criticality are usually associated with specific tasks, and hence the problem of ensuring correct timing for a given functionality is implicitly understood to be equivalent to the problem of properly provisioning the corresponding task.

As should be evident by now, this is far from the case in ROS. Latency-critical functionality is rarely contained to a single component, cause-effect chains usually extend across many executors (and hence threads), and shared executors frequently serve multiple chains with vastly different latency needs and activation patterns.

A latency manager must hence consider ROS systems holistically and cannot provision individual tasks, threads, or other OS entities in isolation.

(II) Do no harm. ROS is popular because, empirically, it works well (enough) for many workloads. By default, ROS relies on Linux's best-effort CFS scheduler, which requires no configuration whatsoever. To state the obvious: active latency management should *not* result in *worse* compliance with latency goals than observed under CFS. This, however, is far less trivial than it sounds since a poorly configured real-time scheduler easily performs much worse than the default CFS scheduler.

A latency manager should hence be self-aware and refrain from enacting configuration changes of uncertain benefit.

(III) No exotic kernel patches. Robotics engineers are generally not willing to switch away from officially supported, "battle-tested" platforms just because of a promise of better

real-time support. The gain in predictability rarely outweighs the lack of tooling, the difficulty in obtaining support, or the (perceived) lack of code maturity with its implied risks of rare bugs and untested corner cases. This rules out the use of bespoke patches augmenting a kernel's real-time capabilities.

A practical solution is hence limited to the facilities found in a stock Linux kernel (and its widely used PREEMPT_RT variant).

(IV) No universal buy-in. The ROS ecosystem is inherently heterogeneous, and development proceeds in an asynchronous, agile fashion, marked by frequent component updates. It is hence not realistic to expect all (or even any) component developers to invest effort into supporting any particular latency management approach, nor is it reasonable to expect system integrators to fill in such support where it is lacking.

In particular, this means a practical solution cannot rely on source-level annotations, presuppose the use of custom APIs, or change how ROS works in fundamental ways.

(V) Ease of adoption. In a similar vein, a latency manager must minimize the upfront configuration and continuous maintenance burden incurred by system integrators. This is especially true given that the baseline choice—the default CFS scheduler—requires no setup at all.

A system integrator usually has a high-level understanding of robot- and mission-specific end-to-end latency requirements but cannot reasonably be expected to know low-level system internals such as how the various ROS components interact precisely, how frequently they do so, how many executors there are, how callbacks are scheduled by the ROS executors, or how Linux's real-time scheduling facilities work in detail.

To minimize the barrier to adoption, a practical latency manager should thus rely as much as possible on dynamic introspection rather than on upfront configuration (or costly static analysis) and favor configuration by means of declarative goals over mechanism-specific options.

(VI) Unpredictable environments. A dynamic, introspection-based approach is also advisable due to the inherently uncertain and shifting resource needs in dynamic environments. Furthermore, latency goals may change as mission profiles evolve and robots adapt their behavior, which reinforces the need for a high-level, goal-oriented approach.

(VII) Nice-to-have payloads. Closely related to the prior point, it would be naïve and misguided to assume that a robot is actually equipped with sufficient compute resources to sustain all software functions in all situations. To the contrary, especially in mobile robots subject to space, weight, and power (SWaP) constraints, it is not uncommon to include "nice-to-have" functionalities that should work "most of the time," but which are not strictly essential and fully expected to operate at a degraded level (or not at all) when conditions become challenging (*e.g.*, mission- but not safety-critical payloads). A practical latency manager must be cognizant of such intentional under-provisioning of non-critical functionalities.

(VIII) Unsurprising overload behavior. Conversely, it is not out of the ordinary for robots to experience periods of transient overload. Such periods are the most challenging situations for an automatic latency manager and necessitate hard choices, as not all latency goals can be satisfied simultaneously. A practical latency manager must not devolve to erratic decision making or otherwise unstable behavior under overload. Rather, it should avoid "surprises" by degrading the latency of cause-effect chains predictably and gracefully.

(IX) Earn your keep. Last but not least, it is worth emphasizing that every processor cycle spent on the latency manager is a cycle not spent on the workload, especially during periods of overload. Since, pragmatically speaking, latency issues under CFS can often be attenuated simply by making additional resources available, it is not a given that the presence of an active latency manager is actually beneficial in terms of latency goal compliance. In other words, a latency manager must yield sufficient benefits to compensate for the cost of running it in the first place. Implementation efficiency and the runtimes of any employed analyses are hence crucial.

5.2 The ROS-Llama Approach

Guided by the just-discussed observations and considerations, we designed the ***ROS L**ive **la**tency **ma**nager* (*ROS-Llama*). ROS-Llama operates largely automatically and follows a *purely declarative* configuration approach.

In terms of necessary setup, ROS-Llama requires a *latency goal* for each cause-effect chain that the system integrator deems latency-sensitive (*i.e.*, in need of active latency management), and a *degradation order* among all latency-sensitive cause-effect chains, which is consulted in case of transient overload.

Each latency-sensitive cause-effect chain is identified solely by its start- and endpoints. For example, a user might specify that at most 200 ms may pass between the arrival of a new measurement at the laser scanner callback and the completion of the callback registering the detected obstacle in the map. It is ROS-Llama's responsibility to determine how these callbacks are connected, how frequently the chain is triggered, how much processor time each callback requires, and ultimately how the involved threads must be scheduled to achieve the latency goal.

The degradation order allows system integrators to configure a policy for controlled degradation, which is also defined in terms of processing chains. This addresses the *dynamic-environment* requirement (Req. VI), the *nice-to-have-payload* requirement (Req. VII), and the the *unsurprising-overload-behavior* requirement (Req. VIII). If ROS-Llama determines that it cannot guarantee all latency goals simultaneously, it degrades some chains to *best-effort mode*. The provided degradation policy determines the order in which ROS-Llama will provision chains. This guarantees predictable behavior under overload and allows system integrators to ensure that critical chains are never degraded to best-effort status in favor of lower-importance chains.

To realize the configured goals, ROS-Llama must solve two main problems. First, it needs to *extract* a model of the running ROS system, including all topics, callbacks, executors, resource needs, *etc.* Second, it needs to *schedule* all threads such that the configured latency goals are satisfied to the extent possible and decide if any chains need to be degraded to best-effort mode.

ROS-Llama's architecture mirrors this structure and consists of two components, each solving

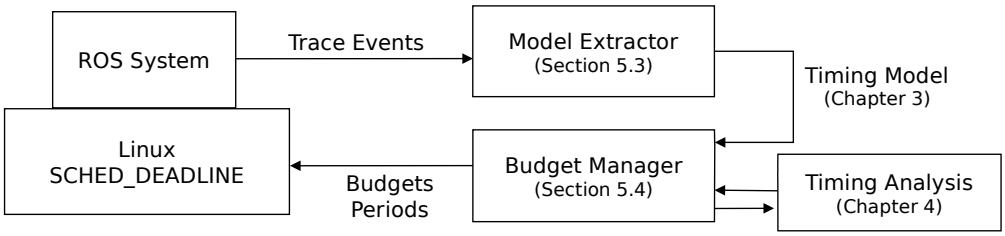


Figure 5.1: Overview of ROS-Llama. The model extractor derives a timing model of the ROS system in a continuous manner and makes it available to the budget manager, which provisions the executor threads based on a response-time analysis of each cause-effect chain. Finally, the new budgets are enacted.

one of the problems: a model extractor and a budget manager (Fig. 5.1).

The **model extractor** (Section 5.3) receives a stream of trace events from the running ROS threads and uses them to derive a timing model of the running system. To keep up with changing environments (Req. VI), the model extractor continuously updates the model at runtime. Effectively, the model extractor represents a dynamic ROS system as a series of static systems over time.

Periodically, the **budget manager** (Section 5.4) takes a snapshot of this model and computes a new set of scheduling parameters, mainly in the form of budgets and periods for the SCHED_DEADLINE scheduler. In our case study, we chose to recompute the budgets every six seconds. To find these budgets, the budget manager relies on the response-time analysis defined in Chapter 4 to predict the impact of the scheduling parameters onto the timing-critical chains. This way, the budget manager always considers the worst case and anticipates adverse activation sequences and task interactions before they even occur.

Once computed, the budgets are used to configure the Linux scheduler. Timing-critical ROS executors are scheduled using the SCHED_DEADLINE scheduler, some system threads use the fixed-priority SCHED_RR scheduler, and the remaining threads are scheduled with the default CFS scheduler.

5.3 Model Extractor

The model extractor derives a timing model of the running system by observing and measuring the running ROS threads. The required information is provided by the ROS components themselves through a tracing mechanism. Since ROS-Llama cannot rely on developers to instrument their code due to the *no-buy-in* requirement (Req. IV), the tracing infrastructure is integrated into the core ROS libraries. As a result, instrumenting an arbitrary (C++-based) ROS component takes only a simple recompilation and no additional developer involvement.

When an instrumented ROS process initializes, it first establishes a communication channel with a central model extractor daemon (Section 5.3.1). The channel allows an arbitrary number of ROS threads to inform the model extractor about any events of interest.

This communication infrastructure is used by eleven ROS-Llama tracepoints in the ROS core libraries (Section 5.3.2). Whenever one of the tracepoints is triggered, it emits a trace event containing the name of the tracepoint, a timestamp, and additional tracepoint-specific parameters.

The trace events are then used by the model extractor to find relevant threads in the system, to recognize which callbacks they serve, and to detect when callback instances start and end (Section 5.3.3). This allows the model extractor to identify all nodes of the callback graph and to associate each trace event with a callback instance.

Once a callback instance completes, its trace events are analyzed to derive the edges of the callback graph as well as activation- and execution-time curves (Section 5.3.4). The result is a complete timing model of the ROS system that grows increasingly accurate as more and more trace events are integrated.

In a highly dynamic ROS system, however, relying too much on previous observations can become a liability as initialization behavior or other non-representative circumstances may come to dominate the timing model. To accommodate such systems, ROS-Llama applies two heuristics that detect the end of the initialization phase and other phase changes in the system. Each phase change allows ROS-Llama to discard measurements taken prior to the change (Section 5.3.5).

In cases where these heuristics do not suffice, ROS-Llama further provides a *data aging*

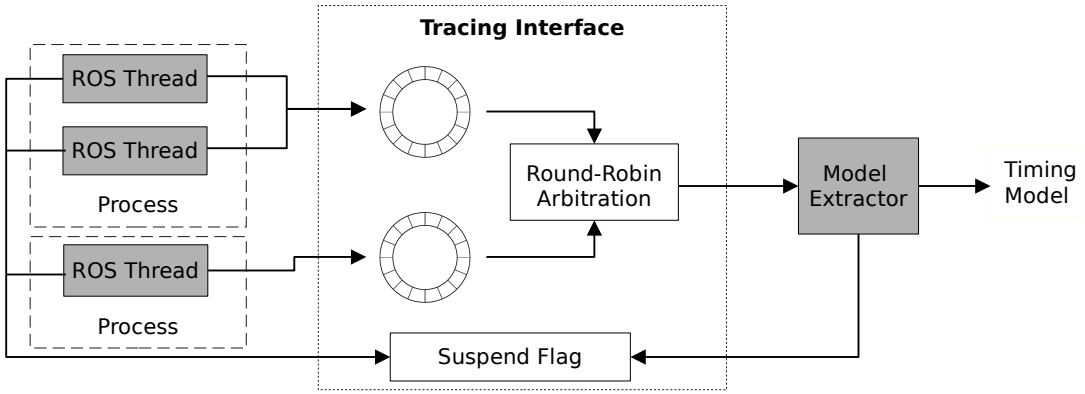


Figure 5.2: The trace event communication infrastructure.

mechanism that automatically "ages out" old measurements. While it requires tuning by the system integrator, it allows the measured timing model to automatically adapt to changing circumstances based solely on changes in a callback's execution time (Section 5.3.6).

5.3.1 Transmitting Trace Events

The communication channel between the ROS threads in the system and the central model extractor daemon needs to satisfy four requirements: first, it needs to support multiple **concurrent writers**, which may join and leave over time. Second, it must be **efficient** enough to transmit hundreds of messages per second and participant. Third, it must allow for **non-blocking writes**, as ROS-Llama's tracepoints may be triggered during latency-critical situations, where a blocking tracepoint would introduce additional latency into the system and violate the *do-no-harm* requirement (Req. II). Fourth, it must support **blocking reads** since the model extractor cannot anticipate the rate of event arrivals and needs to perform well for both high and low arrival rates.

ROS-Llama's implementation of such a communication channel (Fig. 5.2) is based on the *Feather-Trace* toolkit [17]. When a ROS process initializes, it registers with the model extractor daemon through a Unix-domain socket and sets up a shared-memory interface. Both sides map a fixed-size Feather-Trace ring buffer into their address space. In our experiments, the ring buffer held 1024 fixed-size messages of 256 bytes each. If a process contains multiple ROS threads

```
1  read_next_event() {
2    if (a ringbuffer is nonempty) {
3      data = read_from_next_ringbuffer_rr();
4      suspend_flag = 0;
5      return data; }
6    suspend_flag = 1;
7    // Check again, in case a writer raced with our flag change
8    if (a ringbuffer is nonempty) {
9      data = read_from_next_ringbuffer_rr();
10     suspend_flag = 0;
11     return data; }
12   wait_until_unequal(suspend_flag, 1);} // uses the futex syscall
13
14 write_event() {
15   if (buffer_full) {
16     suspend_flag = 0;
17     return fail; }
18   write_event_into_ringbuffer();
19   suspend_flag = 0;
20   return success; }
```

Listing 5.3: Suspension protocol added to the Feather-Trace implementation.

(*e.g.*, multiple independent executors), those threads share the same ring buffer. Additionally, all participants—the daemon and *all* ROS processes in the system—map a shared suspension flag into their address space to allow for blocking reads by the daemon.

To multiplex between the ROS processes, the extractor daemon cycles through the ring buffers in a round-robin fashion. Empty buffers are skipped in this process. If all buffers are empty, the reader suspends until new data becomes available.

Once the reader suspends, it needs to be woken up by one of the writers once new messages are available. This is where the shared suspension flag comes into play. The flag consists of an integer stating whether the reader should be sleeping ($= 1$) or awake ($= 0$). Writers reset the flag upon each write; the reader sets the flag to 1 when it suspends.

The full protocol is defined in Listing 5.3. The reader sets the flag to 0 upon a successful read and to 1 if it finds all buffers empty and intends to suspend (Line 6). To avoid race conditions, the reader checks the buffers a second time before actually suspending; this ensures that no writer added a message between the previous check and the setting of the suspend flag. If the

second check also comes up empty, the reader suspends using the *futex-wait* operation [51], which suspends the calling thread until the given memory location deviates from the given reference value (Line 12). Writers clear the flag upon each completed write, no matter whether the write succeeded (Line 19) or not (Line 16). The reader thus wakes up when the next message arrives.

To sum up, the described implementation provides a multi-writer, single-reader communication channel. The instrumented ROS threads use this channel to inform ROS-Llama whenever a tracepoint is triggered.

5.3.2 Tracepoints

ROS-Llama introduces eleven tracepoints into the ROS core libraries `rclcpp` and `rcl` (Fig. 5.4). The first four tracepoints monitor the ROS executor implementation and report when a callback is registered, invoked, or completed. Tracepoints 5 and 6 monitor the communication layer to report when a callback publishes to a topic or sends a service request. Tracepoints 7–9 report uses of the *rate* API, which is commonly used to implement periodic activations. A thread using the *rate* API is represented as an event source in the callback graph (cf. Section 3.2.3). Finally, tracepoints 10 and 11 warn ROS-Llama about the use of certain executor functions that are not supported by the ROS timing model. These tracepoints allow ROS-Llama to detect threads that it cannot predict reliably and prevents it from provisioning such threads based on false premises.

Whenever the control flow passes a tracepoint, a short trace event—containing the tracepoint ID, the thread ID of the triggering thread, a timestamp in wall-clock time, a timestamp in CPU-clock time, and a few tracepoint-specific parameters—is transmitted to a model extractor daemon. The extractor daemon collects and evaluates the incoming trace data and transforms the stream of events into a single coherent timing model.

5.3.3 Recognizing Callback Instances

The next stage of the model extractor turns the unstructured stream of trace events into a sequence of callback instances. To this end, the extractor needs to identify the threads in the system,

Table 5.4: List of tracepoints in `rcl` and `rclcpp`. See Appendix A for details on their location.

#	Tracepoint	Description
1	register-callback	A callback is registered with a node.
2	start-callback	A callback starts running.
3	end-callback	A callback completes.
4	executor-spin	An executor thread starts processing callbacks ("*spins*").
5	publish	A thread publishes to a topic.
6	send-request	A thread sends a request to a service.
7	rate::sleep	A thread suspends using a *rate* object.
8	rate::wakeup	A thread wakes up after a *rate*-based suspension.
9	rate::stop	A *rate* object is destroyed.
10	limited-spin	An executor thread spins for limited time.
11	spin-until-future-complete	Executor thread spins until a *future* resolves.

recognize which callbacks they serve, and identify when the callback instances begin and end.

A crucial part of this process is to identify the *type* of each thread. The model extractor distinguishes four types of threads. The first type is the *executor thread*, a thread that runs a ROS executor. The next two types are *event source* threads, *i.e.*, threads that do not run a ROS executor but still can be represented in the timing model as an event source callback. Following the two event-source types described in Section 3.2.3, we distinguish *periodic event source threads*, which use the *rate* mechanism to run periodic workloads and *data-driven event source threads*, which use an unobserved mechanism like file-system operations to suspend. Finally, threads may be *unclassifiable*, *i.e.*, not representable in the model. To ROS-Llama, such threads are unpredictable and are monitored to ensure that they do not interact with any timing-sensitive components.

Each type of thread requires a different mechanism to identify callbacks and instance boundaries. For each type of thread, ROS-Llama needs separate rules on **(a)** how to detect that a thread is of that type, **(b)** how to identify which callbacks the thread serves and what their relative priority is, and **(c)** how to recognize when an instance begins and ends.

In the following, we discuss each type of thread in detail and derive the required rules. We then discuss how threads can change their classification over time and conclude with a brief discussion on the limitations and assumptions of the data-driven event source classification.

```
1  main() {
2    node1->create_subscription("/topic", subCallback);
3    node2->create_wall_timer(10ms, timerCallback);
4    executor.add_node(node2);
5    executor.add_node(node1);
6    /* end of initialization, begin main loop */
7    executor.spin();
8  }
9
10 subCallback(message) {
11   next_message = process(message)
12   publish("/other-topic", next_message);
13 }
14
15 timerCallback() {
16   ...
17 }
```

Listing 5.5: Definition of a subscription and a timer callback using the ROS callback API.

```
register-callback(subscription, node1, /topic, 0x1234)
register-callback(timer, node2, 10ms, 0x1235)

executor-spin([0x1235, 0x1234])

start-callback(0x1234)
publish(/other-topic)
end-callback(0x1234)
```

Listing 5.6: The trace events emitted by the example in Listing 5.5.

5.3.3.1 Executor Threads

Executor threads explicitly interact with the ROS callback API and are therefore the easiest type to identify.

Example. Consider the two-callback executor setup in Listing 5.5. The thread first creates two callbacks (Lines 2 and 3) as part of the two nodes *node1* and *node2*. It then adds both nodes to the executor *executor* (Lines 4 and 5) and begins the callback processing loop (Line 7). From this point on, the thread waits for activations of the two callbacks and runs the corresponding callbacks as described in Chapter 3.

To ROS-Llama, the example would appear as the sequence of events in Listing 5.6. The first two trace events are emitted by Lines 2 and 3. Each *register-callback* trace event comes with four parameters: first, the callback type (subscription and timer, respectively); second, the name of the callback's node; third, the activation condition, either in the form of a period (timers) or in the form of an activating topic (message-driven callbacks); and fourth, an identifier, which is used to refer to the callback from other tracepoints and is derived from the memory address of the callback in the emitting thread's address space.

The next trace event, *executor-spin*, is emitted when the thread begins its callback processing loop (Line 7). The trace event carries one parameter, a list of callbacks served by this executor in priority order.[1]

At this point, the thread starts to process callbacks. When a callback is activated, the executor implementation emits the *start-callback* event and runs the callback. The argument identifies the started callback (here: *0x1234*) as the subscription callback registered in line Line 2. As the callback runs, it publishes to `/other-topic` (Line 12), which results in a *publish* trace event. Finally, the callback completes, causing the executor implementation to emit the *end-callback* event.

The example motivates the following general rules for identifying executor threads, their callbacks, and the boundaries between instances.

Identifying the type. The type of an executor thread can be uniquely identified by the *executor-spin* trace event. Since the *spin* function does not return until the program terminates, observing an *executor-spin* guarantees that the thread remains an executor forever. It further guarantees that the thread exclusively runs the callback scheduler described in Section 3.1.1, as required by the timing model.

[1] Technically, it contains the list of nodes registered with the executor and leaves it to ROS-Llama to remember in which order the callbacks were registered with their nodes. This technical detail is omitted for simplicity.

```
1  function() {
2    rclcpp::Rate loop_rate(125ms);
3    while (condition) {
4      data = compute();
5      some_topic->publish(data);
6      loop_rate.sleep()
7    }
8  }
```

Listing 5.7: Example of a periodic event source.

Identifying the callbacks. The callbacks served by the thread are identified by the arguments of the *executor-spin* trace event, which also reports their relative priority. Other callback properties, for example the callback type and the activation criterion, are specified as part of the earlier *register-callback* trace event.

Identifying callback boundaries. Callback instances are delineated by explicit *start-callback* and *end-callback* trace events, which also identify which callback the instance belongs to.

5.3.3.2 Periodic Event Source Threads

Periodic event sources are slightly harder to recognize, as the event source callback does not use the ROS callback API and thus does not explicitly register in advance. However, tracepoints in the *rate* API still allow ROS-Llama to detect instance boundaries, as the following example demonstrates.

Example. Consider the periodic event source in Listing 5.7. To recall from Section 3.2.3, the *rate* object is initialized with a period, in this example 125 ms (Line 2). Each invocation of the *sleep* function (Line 6) then suspends until the beginning of the next 125 ms-period.

To ROS-Llama, the example would appear as the sequence of events in Listing 5.8. As the periodic event source enters the first iteration of its loop, it first triggers the *publish* tracepoint (Line 5). It then reaches the end of the loop where it suspends until the next period, emitting the

```
publish(/some_topic)
rate::sleep(125ms)
rate::wakeup()
publish(/some_topic)
rate::sleep(125ms)
rate::wakeup()
...
rate::stop()
```

Listing 5.8: The trace events emitted by the periodic event source in Listing 5.7.

rate::sleep event in the process (Line 6). At the beginning of the next period, the *sleep* function returns and the *rate::wakeup* event is emitted.

The periodic cycle continues until the condition in Line 3 evaluates to false and the control flow leaves the function. When the *rate* object goes out of scope, the *rate::stop* event is emitted to mark the end of the loop.

As in the case of executors, the observations in the example allow us to derive general rules:

Identifying the type. The type of an executor thread can be uniquely identified by the *rate::sleep* trace event. However, unlike in executor threads, the component developer retains full control over the control flow and may terminate the periodic loop at any point. In that case, the *rate::stop* event informs ROS-Llama that the thread should no longer be considered a periodic event source.

Identifying the callbacks. An event source thread serves only its event source callback.

Identifying callback boundaries. Callback instances are delineated by explicit *rate::wakeup* and *rate::sleep* trace events, which identify when an instance of the single callback starts and stops. An instance also ends when the *rate::stop* event is emitted, which signals that the periodic loop has ended.

```
1    function() {
2      while (condition) {
3        input = read(device_file);
4        data = compute(input);
5        some_topic->publish(data);
6      } }
```

Listing 5.9: Example of a data-driven event source.

A minor limitation of the thread identification approach is that the first iteration of the loop is lost. Since the thread is only identified as a periodic event source thread at the first *rate::sleep* event (Line 6), the first iteration of the loop—which *precedes* the sleep event—cannot be measured. However, losing a single iteration of the loop is unlikely to matter in the long run.

An alternative approach, that does not suffer from this limitation, would be to instrument the creation of the *rclcpp::Rate* object (Line 2) instead. However, this runs the risk of including initialization code that runs after the creation of the *rate* object but before the start of the periodic loop into the measurement. We consider the loss of the first iteration an acceptable price to pay if it prevents initialization workloads from being integrated into the execution-time curve or other callback properties.

5.3.3.3 Data-Driven Event Source Threads

Data-driven event sources are the hardest type to recognize. Activations and suspensions of the data-driven event sources do not involve the ROS API and do not emit trace events. As the following example demonstrates, ROS-Llama therefore needs to rely on heuristics to approximate instance boundaries.

Example. Consider the data-driven event source in Listing 5.9. The program follows the typical structure of a device driver thread. The thread waits for some input from the device file (Line 3), processes the incoming data, and forwards the processed data to a ROS topic (Line 5).

To ROS-Llama, the example would appear as the sequence of events in Listing 5.10. Since the thread does not interact with the ROS API except for its publications, the model extractor

```
publish(/some_topic)
publish(/some_topic)
publish(/some_topic)
...
```

Listing 5.10: The trace events emitted by the data-driven event source in Listing 5.9.

only observes executions of Line 5. As a consequence, the general rules rely exclusively on the *publish* trace event to identify instance boundaries.

Identifying the type. The type of a data-driven event source is determined by elimination: any thread that emits *publish* events while being neither an executor nor a periodic event source is assumed to be a data-driven event source.

Identifying the callbacks. An event source thread serves only its event source callback.

Identifying callback boundaries. From ROS-Llama's point of view, a data-driven event source thread emits only *publish* events. ROS-Llama therefore does not receive any information on the activation or completion of the instance.

To capture the behavior of data-driven event source threads despite those limitations, ROS-Llama assumes that each instance completes shortly after the publication. Since data-driven event sources usually serve as a bridge between an external source and a ROS topic, it is reasonable to assume that an instance has fulfilled its purpose after publication. The *publish* event is therefore treated as the end of an instance and the beginning of the next one.

While this heuristic gives a good estimate of the execution time requirements of the event source, it does not necessarily capture the activation pattern well. Consider, for example, an event source that is triggered every 200 ms. Assume that this event source's first instance completes after 10 ms while the second instance completes after 150 ms. Then ROS-Llama receives *publish* events at time $t = 10$ ms and at time $t = 350$ ms, inferring an inter-arrival time of 340 ms. If the two instances were swapped, ROS-Llama would receive *publish* events at time $t = 150$ ms and

$t = 210$ ms, respectively, and would infer an inter-arrival time of 60 ms. The measurements are distorted by the event source's response-time jitter.

Given enough time, such measurement errors lead ROS-Llama to underestimate the true inter-arrival time since ROS-Llama remembers only the *minimal* inter-arrival times. In the example above, ROS-Llama remembers the 60 ms measurement and forgets about the 340 ms measurement, thereby underestimating the true inter-arrival time by 140 ms. If the data-driven event source is triggered periodically (like the laser-scanner driver in the evaluation platform in Chapter 6), the measurement error thus reduces precision but does not compromise the correctness of the predictions.

Overall, the case of data-driven event sources illustrates that there are limits to the degree of automation that can be realistically attained. In cases where our heuristic does not suffice, the developer or system integrator needs to compromise on automation and assist ROS-Llama by manually adding tracepoints. Such semi-automatic modes of operation would be an interesting area of future work. In this dissertation, we simply rely on the heuristic presented above, which works well for the periodically-triggered sensor driver in the evaluation robot.

5.3.3.4 Unclassifiable Threads

Finally, ROS systems may contain threads that cannot be classified as any of the three categories above. A thread that uses the *spin_until_future_complete* function, for example, temporarily behaves as an executor thread until an unrelated blocking operation completes. This operation cannot be represented in our timing model. If ROS-Llama detects such unsupported behavior, it classifies the thread as an *unclassifiable thread*. ROS-Llama still records the publications and service requests for each unclassifiable thread but does not attempt to extract an execution-time curve.

The unclassifiable thread is added to the ROS callback graph as a set of *opaque senders*, one per topic. Opaque senders have no defined semantics in the timing model; to the analysis, they simply serve as markers that the callbacks subscribing to these topics and handling these services receive

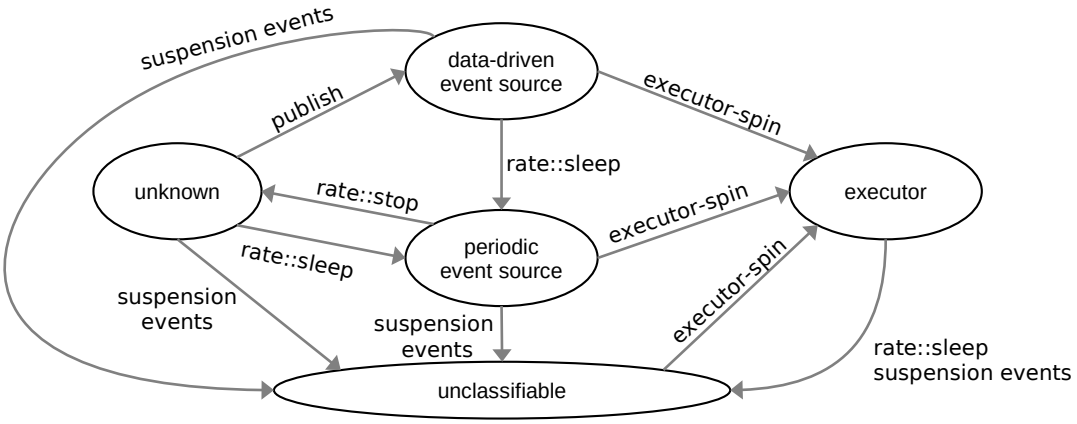


Figure 5.11: Possible transitions between thread types.

requests from an unclassifiable source. This allows ROS-Llama to check that no timing-critical callback is affected by an unclassifiable thread.

5.3.3.5 Thread Type Transitions

The thread type detection rules leave the question of what happens if a thread triggers the tracepoint associated with more than one thread type. How is a thread classified that, for example, first emits a *publish* event (indicating a data-driven event source) and then invokes an *executor-spin* event? To resolve such conflicts, ROS-Llama defines a set of possible transitions between thread types (Fig. 5.11) that list the ways in which threads can be reclassified over time.

Each thread starts in the *unknown* state. When ROS-Llama receives the *publish*, *rate::sleep*, or *executor-spin* event, the thread is reclassified as described in the thread identification rules.

Any thread, independent of its classification, is reclassified as an executor thread if it emits the *executor-spin* event. Since the executor implementation takes over the control flow from the code calling the *spin* function, any observed behavior prior to the *executor-spin* event is unlikely to correctly reflect the thread type.

Similarly, any thread is regarded as unclassifiable if it emits one of the unsupported events (*limited-spin* and *spin-until-future-complete*). An executor thread is also regarded as unclassifiable if it emits a *rate::sleep*, as this indicates a suspension within a ROS callback. Once marked

unclassifiable, a thread can only leave the state through the *executor-spin* event, which, again, takes over the control flow of the thread and is therefore strong evidence for a fundamental change in behavior.

A thread in the *data-driven* state can be reclassified as a periodic event source if the extractor observes the *rate::sleep* event. This commonly happens during initialization, for example if a thread publishes to a topic during setup before it enters a *rate*-based periodic loop.

A thread in the *periodic event source* state is reclassified as *unknown* if it emits the *rate::stop* event, which indicates that the C++ destructor of the *rate* object has been invoked, *i.e.*, that the *rate* object has gone out of scope. This happens, for example, if the *rate* object was used as an improvised timeout device during initialization.

5.3.4 Measuring Callback Properties

With the trace stream separated into callback instances, the model extractor can now measure the required properties of these callback instances. Specifically, three such properties are needed for the model: the *successors* in the callback graph, the *execution-time curve*, and the *activation curve*.

5.3.4.1 Identifying Graph Successors

For each callback c_i, the model extractor records all topics c_i ever published to. When the budget manager requests a new snapshot, the model extractor creates an edge from c_i to any callback c_j subscribing to one of these topics. Each such edge is created with a full trigger set, *i.e.*, $tris_{i,j} = succ(j)$. An exception is made for *self-activations*, which we discuss below.

The model extractor further records all service requests c_i makes and, for each request, records the associated client callback c_c. The model extractor then creates an edge from c_i to the service handler c_s and a return edge (c_s, c_c). The trigger set of (c_i, c_s) is set to $(succ(c_s) \setminus \mathcal{C}^{\mathrm{clt}}) \cup \{c_c\}$, *i.e.*, all successors of c_s except for clients related to other service requests. The return edge (c_s, c_c) receives the full trigger set, just like a publication.

The model extractor chooses a more narrow trigger set if a callback publishes to a topic that it has itself subscribed to. Although it may seem like such a self-activation serves no purpose, they do occur in practice, for example in the self-localization component *amcl* of the ROS navigation stack.

In short, this component publishes the robot's estimated position through the widely used TF coordinate transform library [42], a core ROS library for managing the various coordinate transformations encountered in robotics applications. However, *amcl* computes these position estimates from odometry updates, which are also distributed by TF. The *amcl* component hence both subscribes and publishes to the `/tf` topic. Why does this not trigger an infinite loop? The answer lies in the *content* of the messages published on the `/tf` topic: *amcl*'s position update is triggered only by messages updating the odometry coordinate frame. Translations involving any other coordinate frames, like the position update published by *amcl* itself, are ignored and can therefore not trigger a cycle.

In conclusion, while there is no point in intentionally triggering c_i from itself, it may well happen that c_i triggers itself as a side effect of triggering other subscribers of the same topic. To handle such cases, ROS-Llama assumes that no callback intends to activate itself with a publication and that the developer prevents the apparent self-activation through some other means (e.g., based on the content of the message like in the TF case). It therefore associates any self-loop in the graph with an empty trigger set (*i.e.*, $tris_{i,i} = \emptyset$).

5.3.4.2 Measuring the Execution-Time Curve

The execution time of the callback is given by the processor time consumed by the thread between the beginning and the end of a callback instance. To measure the elapsed time, each trace event contains a timestamp derived from the thread's *CPU-time clock*. Linux provides a separate CPU-time clock for each thread (called CLOCK_THREAD_CPUTIME_ID). The clock advances only while the thread actively runs on the processor and is paused when the thread is descheduled from the processor, for example if it is preempted by a higher-priority thread. Deriving the

Algorithm 1: The execution-time curve update rule.

Input: t: new execution time measurement
Data: $W[(-L+1)\dots 0]$: sliding window of the last L execution times
$ET^*[1\dots n]$: current execution-time curve prefix
Result: t is integrated into W and ET^*

```
// Shift the window left and add t
```
1 $W[-len(W)\dots -1] \leftarrow W[-len(W)+1\dots 0]$
2 $W[0] \leftarrow t$

```
// Update the execution-time curve prefix
```
3 **for** *n* **from** *0* **to** *len(W)* **do**
4 $\quad ET^*[n] \leftarrow \max(ET^*[n], \sum_{i=n-1}^{0} W[i])$

```
// Shrink the window back to length L
```
5 **if** $len(W) > L$ **then** $discard(W[-L])$

execution time measurement from the CPU-time clock thus measures only the raw execution time and is not distorted by scheduling interference or thread suspensions.

Each instance of a callback contributes one execution-time measurement. The model extractor aggregates those execution-time measurements over time into a single execution-time curve prefix ET^*, as shown in Algorithm 1.

The algorithm considers a sliding window of the last L instances' execution time (called W). Each new execution-time measurement t is first added to the window (lines 1–2). Then for each n, the existing bound is raised if it failed to bound the cumulative execution time of the last n instances (lines 3–4). Finally, the sliding window is truncated back to L elements again (line 5).

Since the update procedure is applied for each new execution-time measurement, the resulting execution-time curve upper-bounds the cumulative execution time of all observed n-element sequences. The value of L compromises between accuracy and overhead: larger values of L reduce the amount of (pessimistic) extrapolation but are more costly to update and vice versa. In our experience, $L = 64$ is long enough to analyze most callbacks without extrapolation but short enough that curve updates have a reasonable cost.

Execution-time bounds for sequences longer than L are extrapolated from the L-element prefix using the super-additivity property of execution-time curves. Recall from Section 3.2.4 that the super-additivity property states that, for any two integers m and n, $ET_i(m) + ET_i(n) \geq$

$ET_i(m+n)$. Using this property, the prefix is extrapolated as follows:

Definition 9. Let ET^* denote the measured execution-time curve prefix. The execution-time curve is then extrapolated as

$$ET(n) \triangleq \begin{cases} ET^*[n] & n \leq L \\ \min\{ET(n-a) + ET(a) \mid a \in [1, n-1]\} & \text{otherwise} \end{cases}$$

The correctness of the extrapolation is shown in the following lemma:

Lemma 23. *Let $ET^* : [0, L] \mapsto \mathbb{N}$ be an arbitrary super-additive function. Let $ET(n)$ be derived from ET^* as described in Definition 9. Then $ET(n)$ is super-additive and a (pointwise) upper bound for any execution-time curve with prefix ET^*.*

Proof. We first show that $ET(n)$ is super-additive by construction. For $n \leq L$, this follows directly from ET^*'s super-additivity. For $n > L$, $ET(n)$ is super-additive by definition: for any set S, $\min S$ is less or equal to any member of S. Therefore, $\min\{ET(n-a) + ET(a) \mid a \in [1, n-1]\}$ is guaranteed to be less or equal to $ET(n-a) + ET(a)$ for all $a \in [1, n-1]$.

We then show that $ET(n)$ upper-bounds any execution-time curve with prefix ET^*. Let f be an arbitrary super-additive function with prefix ET^*. For any $n \leq L$, $ET(n)$ is equal to $ET^*(n)$ and thus a trivial upper bound for $f(n) = ET^*(n)$.

Starting from this base case, the remaining $n > L$ are proven by (complete) induction on n: we show that the induction hypothesis $(\forall n' < n.\ ET(n') \geq f(n'))$ implies $ET(n) \geq f(n)$.

$$\begin{aligned} ET(n) &= \min\{ET(n-a) + ET(a) \mid a \in [1, n-1]\} \\ &\geq \min\{f(n-a) + f(a) \mid a \in [1, n-1]\} && \text{(induction hypothesis)} \\ &\geq \min\{f(n) \mid a \in [1, n-1]\} && (f \text{ super-additive}) \\ &= f(n) && \square \end{aligned}$$

The recursive extrapolation rule can be efficiently implemented through dynamic programming,

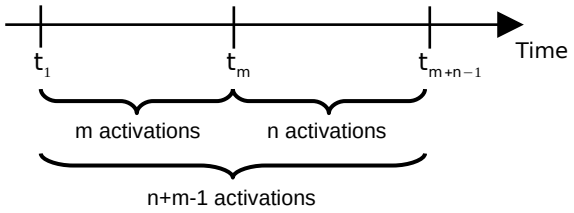


Figure 5.12: Argument offset in $\delta^{\min}$ curves.

i.e., by memoizing the values of $ET(n-a)$ and $ET(a)$ during the recursive evaluation.

The execution-time curve measurement mechanism is also used to measure the executor's overhead curve. An overhead measurement consists of the CPU clock difference between two callback instances, *i.e.*, between the end of one and the beginning of the subsequent instance. Each such measurement is integrated into an overhead-curve prefix for the executor using Algorithm 1, which is further extrapolated using the super-additivity property if needed.

5.3.4.3 Measuring the Activation Curve

Timers and periodic event sources report their period as part of the *register-callback* and *rate::sleep* trace events, respectively. For these callbacks the activation curve is trivially derived as $\eta(\Delta) = \lceil \Delta / \text{period} \rceil$. For other callbacks, namely data-driven event sources, the activation curve needs to be measured from the observed inter-arrival time between callback instances.

The inter-arrival time between two instances is measured using a secondary timestamp carried by each event. The secondary timestamp is taken from the *monotonic wall-time clock* (called CLOCK_MONOTONIC in Linux) at the time the event was emitted. The monotonic wall-time clock counts the wall-clock time elapsed since an unspecified time in the past, usually system startup. Unlike the CPU-time clock, the wall-time keeps advancing while the thread is suspended.

Like with execution-time curves, an incremental update procedure aggregates the individual measurements into a single activation curve. Instead of directly measuring this activation curve $\eta(\Delta)$, the tracer records $\eta(\Delta)$'s pseudo-inverse, the minimum inter-arrival curve $\delta^{\min}(n)$ [101]. $\delta^{\min}(n)$ maps a number of instances n to the first duration Δ where $\eta(\Delta) \geq n$. It thus contains

Algorithm 2: The activation-curve update rule.

Input: t: new activation time
Data: $W[(-L+1)\dots 0]$: sliding window of the last L activation times
$\delta^{min^*}[1\dots n]$: current execution-time curve prefix
Result: t is integrated into W and δ^{min^*}

```
// Shift the window left and add t
```
1 $W[-len(W)\dots -1] \leftarrow W[-len(W)+1\dots 0]$
2 $W[0] \leftarrow t$

```
// Update the curve prefix
```
3 **for** *n* **from** *0* **to** *len(W)* **do**
4 $\quad \delta^{min^*}[n] \leftarrow \min(\delta^{min^*}[n], W[0] - W[-n+1]$

```
// Shrink the window back to length L
```
5 **if** $len(W) > L$ **then** $discard(W[-L])$

the same information as $\eta(\Delta)$, but allows for a more compact representation and makes it easier to see the similarities to the execution-time curve measurements.

The minimum inter-arrival-time curve is quasi-sub-additive in the sense that $\delta^{\min}(m) + \delta^{\min}(n) \leq \delta^{\min}(m+n-1)$. The quasi-sub-additivity property allows ROS-Llama to extrapolate $\delta^{\min}(n)$ for $n > L$ from the L-element prefix. The reason for the offset of -1 is depicted in Fig. 5.12: when the inter-arrival time of $m+n-1$ instances is split into the inter-arrival time of the first m instances and the inter-arrival time of the last n instances, one activation (t_m) is part of *both* sequences.

Like in the case of the execution-time curve, the extractor maintains a sliding window of the last $L = 64$ activations of c_i and derives a prefix of the activation curve (Algorithm 2).

The algorithm considers a sliding window of the last L activations (called W). Each new activation time t is first added to the window (lines 1–2). Then for each n, the existing inter-arrival time bound is lowered if it failed to bound the inter-arrival time of the last n instances (lines 3–4). Finally, the sliding window is truncated back to L elements again (line 5).

The resulting activation-curve prefix is then extrapolated as follows:

Definition 10. Let δ^{min^*} denote the measured activation-curve prefix. The minimum inter-arrival-

time curve is then extrapolated as

$$\delta^{\min}(n) \triangleq \begin{cases} \delta^{min^*}(n) & n \leq L \\ \max(\delta^{\min}(n-a+1) + \delta^{\min}(a) \mid a \in [2, n-1]) & \text{otherwise} \end{cases}$$

The correctness of the extrapolation is proven by the following lemma:

Lemma 24. *Let $\delta^{min^*} : [1, L] \mapsto \mathbb{N}$ be an arbitrary quasi-sub-additive function. Let $\delta^{\min}(n)$ be extrapolated from δ^{min^*} as described in Definition 10. Then $\delta^{\min}$ is quasi-sub-additive and a (pointwise) lower bound for any activation curve with prefix δ^{min^*}.*

Proof. The argument is analogous to Lemma 23.

We first show that $\delta^{\min}(n)$ is quasi-sub-additive by construction. For $n \leq L$, this follows directly from δ^{min^*}'s quasi-sub-additivity. For $n > L$, $\delta^{\min}(n)$ is quasi-sub-additive by definition: for any set S, $\max S$ is greater or equal to any member of S. Therefore, $\max(\{\delta^{\min}(n-a+1) + \delta^{\min}(a) \mid a \in [2, n-1]\})$ is guaranteed to be greater or equal to $\delta^{\min}(n-a+1) + \delta^{\min}(a)$ for all $a \in [2, n-1]$.

We then show that $\delta^{\min}(n)$ lower-bounds any activation curve with prefix δ^{min^*} Let f be an arbitrary quasi-sub-additive function with prefix δ^{min^*}. For any $n \leq L$, $\delta^{\min}(n)$ is equal to $\delta^{min^*}(n)$ and thus a trivial lower bound for $f(n) = \delta^{min^*}(n)$.

Starting from this base case, the remaining $n > L$ are proven by (complete) induction on n: we show that the induction hypothesis $(\forall n' < n.\ \delta^{\min}(n') \leq f(n'))$ implies $\delta^{\min}(n) \leq f(n)$.

$$\begin{aligned} \delta^{\min}(n) &= \max\{\delta^{\min}(n-a+1) + \delta^{\min}(a) \mid a \in [2, n-1]\} \\ &\leq \max\{f(n-a+1) + f(a) \mid a \in [2, n-1]\} && \text{(induction hypothesis)} \\ &\leq \max\{f(n) \mid a \in [2, n-1]\} && (f \text{ quasi-sub-additive}) \\ &= f(n) \end{aligned}$$ □

Like for execution-time curves, the recursive evaluation rule can be computed efficiently through dynamic programming.

Naively applying the above update rule might yield pessimistic results for rarely-occurring events. Consider an event source that activates five minutes after startup and again one millisecond later. Extrapolating from just these two cases, the activation curve would predict a new activation every millisecond, which is unacceptably pessimistic. The fact that it took five minutes until the first event occurred needs to be incorporated into the activation curve.

ROS-Llama therefore counts the system startup as a zeroth event (i.e., the startup time appears as t_0 in the update rule). In the example above, the activation curve would state that the inter-arrival time for two events is one millisecond, but the inter-arrival time for three events is 5 minutes + 1 millisecond. This prevents overly pessimistic extrapolation of the arrival curve for rare events (e.g., interactive commands) and during the initial time window until L events have been observed. If the event source *actually* activates frequently, t_0 is quickly shifted out of the sliding window, at which point it no longer affects the result.

To sum up, the model extractor extracts graph edges, execution-time curves, and activation curves by aggregating measurements over time. It monitors the resulting timing model, which grows more pessimistic over time as necessary.

While this approach is sufficient for a static system, it can be too pessimistic in a dynamic system where threads transition through different phases. In the following, we discuss how to strategically discard existing measurements in this situation.

5.3.5 Detecting Initialization Phases

ROS-Llama's timing model describes a static system. This keeps the model simple enough to perform response-time analysis but fails to account for transient states in the system, for example initialization phases or transitions between different steady states. We call such a change in the internal state of the thread a *phase change*, as the thread enters a new phase of its lifecycle. Measurements taken before a phase change are rarely indicative of behavior after the phase change and may lead to excessive pessimism.

ROS-Llama attempts to discover phase changes through a set of heuristics. If any of these

heuristics detects a phase change, ROS-Llama purges the sliding windows, resets the activation- and execution-time curve estimates to the initial state, and discards all outgoing edges of the callback. Effectively, all callbacks of the node are reset to their initial state.

In the following, we discuss the two heuristics proposed in this dissertation: the *initialization-topic heuristic* and the *thread-type-change heuristic*.

5.3.5.1 Initialization-Topic Heuristic

As discussed in Section 2.1, ROS comes with standardized interfaces for parameter and lifecycle management. If ROS-Llama observes activity on the topics and services associated with these interfaces, it can reasonably assume that the node just underwent a phase change At this point, past measurements are therefore unlikely to predict future activity.

Specifically, ROS-Llama monitors one topic and one set of services. First, ROS-Llama monitors the `/parameter_events` topic. The parameter system uses this topic to inform the system about any changes to the configuration parameters. Any node that publishes to this topic has recently changed its internal configuration and is therefore assumed to still be initializing.

ROS-Llama further monitors the `change_state` service of the lifecycle management library, which is offered by every node using the lifecycle mechanism. A request to this service commands the node to transition between lifecycle states, for example from the *inactive* state to the *active* state. If a node receives such a request, it just entered a new phase in its lifecycle and thus likely underwent a phase change.

5.3.5.2 Thread-Type-Change Heuristic

The second heuristic monitors changes in the thread type. If, for example, a periodic event source thread suddenly emits an *executor-spin* event, it is clear that the behavior of the thread changed fundamentally. ROS-Llama therefore interprets each thread type change as indicative of a phase change.

However, there is one scenario where thread type changes do not necessarily indicate a phase

```
1  while (true) {
2    wait_for_command();
3    {
4      rclcpp::Rate loop_rate(125ms);
5      while (!arrived) {
6        cmd = compute_vel_cmd();
7        cmd_vel->publish(cmd);
8        loop_rate.sleep()
9      } }
10   cmd_vel->publish(stop_cmd); }
```

Listing 5.13: The *dwb_controller* (simplified).

change. Consider, for example, the *dwb_controller* node in the ROS navigation stack (see Listing 5.13 for a simplified representation). The controller waits for navigation commands. Whenever it receives a command, it enters a *rate*-based loop. Once the goal is reached, it leaves the loop again, publishes a stop command to the engine, and waits for the next command.

Over time, this thread oscillates between three thread types. It is classified as a *periodic event source* while the robot moves. When the *rate* loop ends it is reclassified as *unknown* until it publishes the stop command and is classified as a *data-driven event source*. If ROS-Llama interpreted each transition as a phase transition, the thread would oscillate between the three states and ROS-Llama would repeatedly discard everything it learned about the involved callbacks.

To avoid this oscillating behavior, ROS-Llama can be configured to classify any such oscillating thread functions as a periodic event source. In this case, ROS-Llama disables the transition from the *periodic event source* state to the *unknown* state after it has been taken once. Once a thread enters the *periodic event source* state for the second time, the *rate::stop* event no longer induces a state transition but simply marks the end of the periodic callback instance.

The oscillation suppression mechanism is motivated by the behavior of the *dwb_controller* and is enabled in our evaluation setup. However, it is not obvious whether this is the right choice for *all* workloads. Extending the timing model to represent such oscillating workloads remains future work.

5.3.6 Data Aging

Even outside of initialization phases, it is often useful to deliberately disregard earlier measurements. As discussed in the *dynamic-environment* requirement (Req. VI), ROS-Llama should adapt to changing demands in execution-time: if an object-tracking component tracks fewer objects now than it did in the past, it should also receive a lower budget than it did in the past. More generally speaking, ROS-Llama needs a way to adjust model parameters towards smaller execution-time bounds and needs to avoid budgeting for past measurements that no longer reflect the current workload.

Unfortunately, such an adaptation risks unsafe execution-time bounds. Adapting the system to a low-processor-demand environment involves choosing a budget that ROS-Llama *knows* to be insufficient in a previously observed environment. With the information available to ROS-Llama, any adaptation to reduce demand runs the risk of causing a latency goal violation if demand suddenly picks up again.

The appropriate trade-off between adaptivity and safety heavily depends on the use case in question and requires a value judgment by the system integrator. A comprehensive solution for the problem would need to explore this trade-off across different case studies, which is out of the scope of this dissertation. However, as a first step towards a more comprehensive solution, ROS-Llama compromises on the *ease-of-adoption* requirement (Req. V) and provides a generic solution that exposes explicit tuning knobs to the user. The solution consists of an extension to the execution-time curve measuring process, which we call *data aging*.

The principal goal of data aging is to reduce a callback's execution-time bounds if the measured execution times remain significantly below the bound for extended periods of time. A previously observed high-load situation would thus be "aged out" after the robot spends enough time in a low-demand setting. An execution-time demand increase, on the other hand, should raise the execution-time bound immediately. The execution-time curve thus *slowly* adapts downwards but *immediately* adapts upwards.

The aging mechanism. If data aging is enabled, ROS-Llama tracks the recent processor demand of each callback c_i through a separate *short-term execution-time curve* $ET_i^s(n)$. The curve is updated and maintained like the regular execution-time curve but is reset periodically. The short-term curve thus provides an upper bound on the execution-time demand during the recent past.

The updating mechanism is controlled by four parameters: the *merging period* $T \in \mathbb{N}_{\geq L}$, which determines how frequently the short-term curve is reset, the *merging weight* $\alpha \in [0, 1]$, which determines how much weight recent measurements should have compared to older measurements, the *trigger threshold* $G \in [0, 1]$, which determines which ratio between observed execution time and execution-time bound is considered "significant overestimation", and the *safety margin* $S \in \mathbb{R}$, which increases the execution-time bound by a fixed percentage for improved protection against underestimation.

If the ratio ${}^{ET_i^s(1)}/_{ET_i(1)}$ is below G, *i.e.*, if the maximum processor demand of a single activation during the last interval was significantly lower than indicated by the execution-time curve, the short-term curve is *merged* into the regular execution-time curve. The merging process adjusts $ET_i(n)$ downwards by setting it to a point between the previous value of $ET_i(n)$ and the short-term bound $ET_i^s(n)$.

The merging process follows the well-known *exponentially-weighted moving average (EWMA)* algorithm. The EWMA at time t of a sequence $(a_0, a_1, \ldots)$ is defined as

$$EWMA_t = \begin{cases} a_0 & t = 0 \\ \alpha \cdot a_t + (1 - \alpha) \cdot EWMA_{t-1} & \text{otherwise} \end{cases}$$

The merging weight α determines how quickly older values are discounted. High values of α discount older values more rapidly, which makes the EWMA more volatile but quicker to adapt to changes in the data. Low values of α discount older values more slowly, which leads to slower but less abrupt changes.

Overall, the merging process can thus be described as follows:

$$ET(n) \leftarrow \begin{cases} ET(n) & \frac{ET^s(1)}{ET(1)} < G \\ \lceil \alpha \cdot ET^s(n) + (1-\alpha) \cdot ET(n) \rceil & \text{otherwise.} \end{cases}$$

Both $ET(n)$ and $ET^s(n)$ are defined for all $n \in [0, L)$ since $L \leq T$. The EWMA is rounded up after the merge to ensure that $ET(n)$ remains integer.

The following lemma shows that the merging process preserves super-additivity:

Lemma 25. *Let f and g be two arbitrary super-additive functions, and $0 \leq \alpha \leq 1$. Then*

$$h(x) \triangleq \lceil \alpha \cdot f(x) + (1-\alpha) \cdot g(x) \rceil$$

is also super-additive.

Proof. We need to show that $\forall x, y.\ h(x) + h(y) \geq h(x+y)$. Substituting h's definition yields:

$$\begin{aligned}
&h(x) + h(y) \\
&= \lceil \alpha \cdot f(x) + (1-\alpha) \cdot g(x) \rceil + \lceil \alpha \cdot f(y) + (1-\alpha) \cdot g(y) \rceil \\
&\geq \lceil \alpha \cdot f(x) + (1-\alpha) \cdot g(x) + \alpha \cdot f(y) + (1-\alpha) \cdot g(y) \rceil \\
&= \lceil \alpha \cdot (f(x) + f(y)) + (1-\alpha)(g(x) + g(y)) \rceil \\
&\geq \lceil \alpha \cdot f(x+y) + (1-\alpha) g(x+y) \rceil && \text{(super-additivity of } f \text{ and } g\text{)} \\
&= h(x+y) && \square
\end{aligned}$$

The safety margin. To reduce the risk of underestimating a callback's execution time, the *safety margin* S can be used to add a fixed margin to the execution-time curve. The safety margin is implemented by multiplying each new execution-time measurement by $(1+S)$.

At first glance, the safety margin might seem to counteract the purpose of data aging. Wouldn't the safety margin *increase* pessimism compared to the variant without data aging? In mostly-static systems, where data aging has little positive effect, this is indeed the case. However, in sufficiently

dynamic systems, where data aging makes a large difference, the gains from data aging can more than compensate for the safety margin, while the safety margin significantly reduces the underestimation risk of the aging mechanism.

The experiments in Section 6.5 show that a safety margin is particularly important if the workload ramps up slowly, as it allows the execution-time curve to gradually increase over time without underestimating the observed processor demand in the meantime.

This concludes our discussion of the model extractor. We now turn towards the budget manager, which regularly requests the extracted model from the model extractor in order to make suitable scheduling decisions for the system.

5.4 Budget Manager

Recall from Section 5.2 that ROS-Llama consists of two main components: the model extractor, which derives a model of the running ROS system, and the budget manager, which computes and applies scheduling parameters to ensure that the targeted latency goals are fulfilled. The budget manager is further responsible for enforcing the degradation order. It needs to detect if a latency goal cannot be fulfilled and, if so, needs to proactively degrade chains earlier in the degradation order to ensure that chains later in the degradation order complete in time. It thereby fulfills the *nice-to-have-payload* requirement (Req. VII) and the *unsurprising-overload-behavior* requirement (Req. VIII).

The first part of this section (Section 5.4.1) describes the scheduling approach taken by ROS-Llama. We survey various options to schedule ROS applications on Linux and conclude that the most suitable solution is to schedule ROS executors as SCHED_DEADLINE reservations under partitioned scheduling. The ROS-Llama infrastructure and several critical non-ROS threads are isolated on a separate *system core*.

The second part of this section (Section 5.4.2) describes how the necessary configuration, namely thread-to-core assignments and reservation parameters, are determined by ROS-Llama.

5.4.1 Scheduling Strategy

Before taking a detailed look at the budget manager, it is necessary to understand how provisioned threads are actually scheduled by ROS-Llama. There are three design decisions to make. First, which Linux scheduler should ROS-Llama use to schedule timing-critical ROS threads? Second, should ROS-Llama use partitioned scheduling, global scheduling, or something in-between? And third, how should ROS-Llama schedule non-ROS threads such as kernel threads, middleware threads, or ROS-Llama's own threads?

Which scheduler to use? Due to the *stock-kernel* requirement (Req. III), ROS-Llama must use one of the schedulers available in Linux: either the default CFS scheduler, or a fixed-priority scheduler (*i.e.*, the SCHED_FIFO or SCHED_RR policies), or Linux's more recent reservation-based SCHED_DEADLINE scheduler.

We choose SCHED_DEADLINE as the scheduler for ROS executors. Compared to the other options, this provides two main advantages: *analyzability* and *containment*.

Analyzability means that ROS-Llama can use a response-time analysis to predict the effect of its provisioning on the worst-case latency of the associated chains.It enables ROS-Llama to decide with confidence whether a cause-effect chain can be guaranteed, or else needs to be degraded to best-effort mode in accordance with the *do-no-harm* requirement (Req. II).

Containment means that a thread experiencing an unexpected surge in processor demand cannot prevent other threads from receiving their provisioned budget in a timely manner. This property is beneficial in case of transient overload or when resource demand increases due to changes in the robot's environment. It is hence well-aligned with the *dynamic-environment* requirement (Req. VI) and the *unsurprising-overload-behavior* requirement (Req. VIII). In such situations, containment keeps the system in a stable and predictable state until ROS-Llama computes a new set of budgets that accounts for the increased execution-time demand.

Of the three available schedulers—CFS, fixed-priority scheduling, and SCHED_DEADLINE—only SCHED_DEADLINE guarantees both of these properties.

CFS provides a certain degree of containment in that it does not allow any thread to exceed its "fair" share in the long run. However, the size of the share is hard to predict in advance and depends on various factors, including the number and weight of other threads in the system. It is therefore not obvious how to predict the effects of a surge under CFS. Furthermore, CFS provides no analyzability since no response-time analysis for CFS is known.

The fixed-priority schedulers provide analyzability but no containment. If a thread increases its processor demand, the supply available to lower-priority threads shrinks by the same amount. Although there is a mechanism called *"real-time group scheduling"* [2] that limits the processor time consumed by all fixed-priority threads collectively, it is far too coarse-grained for our purposes as it throttles all fixed-priority threads equally, independent of priority.

SCHED_DEADLINE provides both analyzability and containment. It allows ROS-Llama to predict whether the chain goals will be fulfilled (assuming the latest timing model is correct) *and* contains any deviations from the timing model to the executor that caused it.

However, these advantages do not apply to degraded chains, *i.e.*, chains that ROS-Llama cannot provision with enough budget to always reach their latency goal. If ROS-Llama were to schedule a degraded chain's threads with SCHED_DEADLINE, it would need to deliberately under-provision them. This turns the containment guarantee provided by the hard reservation mechanism into a liability, as executors would be throttled prematurely once their budget runs out, even if enough capacity is available to complete. This violates the *do-no-harm* requirement (Req. II). ROS-Llama hence uses CFS to schedule executors that only serve degraded chains, which lets the degraded chain be served in a best-effort manner rather than risk providing a thread with an insufficient budget. The chain continues to operate without interruption and may still complete in time, but ROS-Llama cannot guarantee that it does.

Partitioned or global scheduling? As discussed in Section 2.2.3, Linux provides two ways to instantiate SCHED_DEADLINE on multicore platforms: *global scheduling*, where the scheduler migrates threads freely among cores depending on current availability, and *partitioned*

scheduling, where each thread is assigned to a specific core on which it remains even when other cores are idle.

Prior work [19] has shown that partitioned scheduling achieves higher schedulability for most workloads, *i.e.*, it is much more effective at admitting and guaranteeing reservations (Section 2.2.3). However, its effectiveness depends heavily on the mapping from tasks to cores, which places an additional burden on the user, especially in dynamic environments. Linux avoids this burden by defaulting to global scheduling. ROS-Llama, on the other hand, has sufficient information to determine a suitable mapping automatically and therefore uses partitioned scheduling without imposing any additional burden on the system integrator.

How to schedule non-ROS threads? ROS threads are not the only timing-critical threads in the system. ROS-Llama also needs to account for critical kernel and middleware threads. We refer to these critical non-ROS threads as *system threads*. Ensuring that these system threads receive enough supply is at least as important as supplying the executors: starving the wrong kernel threads can lead to I/O delays or even to kernel panics. Similarly, starving the middleware threads may interfere with the communication between callbacks and risks excessive delays.

Since ROS-Llama cannot predict the processor demand of system threads, scheduling them with SCHED_DEADLINE is out of the question. Any misjudgment of the processor demand for one of these threads would lead to premature throttling once the thread exceeds its budget. ROS-Llama therefore schedules system threads using the SCHED_RR fixed-priority scheduler.

Unfortunately, this makes it impossible to run the system threads on the same core as ROS threads. In Linux, SCHED_DEADLINE threads always take priority over fixed-priority threads (cf. Section 2.2.4). Any ROS executor, no matter how early in the degradation order, could thus starve any system thread. ROS-Llama addresses this issue by isolating the system threads on a separate *system core*. Kernel threads run at their default priority; middleware threads run at a low real-time priority. Additionally, the system core runs the ROS-Llama infrastructure, which is not time-critical and therefore scheduled under CFS. The remaining cores run the ROS threads.

Summary. Overall, ROS-Llama schedules ROS systems through a combination of reservation-based, fixed-priority, and best-effort scheduling. ROS threads, *i.e.*, executors and event sources, are scheduled with SCHED_DEADLINE and are partitioned across all but one of the processor cores. Timing-critical system threads are scheduled with SCHED_RR on a separate system core. This core also hosts the ROS-Llama infrastructure, which uses the CFS scheduler.

5.4.2 Budgeting Heuristic

Based on the extracted timing model, the budget manager is responsible for finding a scheduler configuration—a budget and period for each reservation and a feasible mapping of reservations to cores—that ensures the timely completion of the configured chains. If it cannot find a configuration that fulfills all latency goals, it must initiate the controlled degradation process and ensure that the system abides by the configured degradation order.

We begin with the **period**. As discussed in Section 2.4, a reservation's period should ideally be derived from the underlying periodicity of the task. If that is not possible, the period choice becomes a trade-off: increasing a reservation's period increases the initial supply delay but decreases scheduling overheads. Periods should thus be set to the shortest value that still yields acceptable scheduling overheads.

For ROS-Llama, matching the period to the underlying periodicity of the reservation is not possible in general because executors do not necessarily *have* an underlying periodicity. The callbacks assigned to an executor are often not triggered periodically. Even if they are periodic, they do not necessarily have the same period: due to the *form-deviates-from-function* requirement (Req. I), a single executor often influences multiple independent chains.

ROS-Llama thus simply assigns all reservations a uniform period that is significantly shorter than the tightest latency goal but sufficiently long to avoid undue context-switching overheads. Based on experiments taken on our evaluation platform (Section 6.4), we selected a uniform reservation period of 5 ms. Finding a method to automatically fine-tune the reservation periods to individual workloads remains future work (cf. Section 7.2.4).

The **budget** is chosen based on the worst-case processor demand that may be exhibited by each thread, as determined by the extracted timing model. ROS-Llama relies on the response-time analysis presented in Chapter 4 to identify whether a set of budget assignments fulfills the desired latency goals.

Unfortunately, it is far from obvious how to find a budget assignment that fulfills this test. Reservation budgets cannot be chosen in isolation but require solving a global co-optimization problem due to the interconnected nature of the callback graph. The underlying intuition is easy to see: in a callback chain, the response-time bound of an "upstream" callback determines the activation jitter of any "downstream" callbacks. Changing the budget of one executor affects the response-time bounds of *all* callbacks that it handles and can thus induce changes in demand in any number of executors serving "downstream" callbacks. This, in turn, affects response-time bounds in those executors, at which point the propagation effect repeats. Worse, influence *cycles* are possible: though each callback chain is cycle-free, it is possible for two executors to be connected by multiple callback chains in opposite directions.

Arbitrary budget dependencies may thus exist among executors. Optimally solving such a complex optimization problem would be much too expensive at runtime, especially given the *earn-your-keep* requirement (Req. IX). ROS-Llama therefore attempts to find a (non-optimal) solution with an iterative heuristic search. Developing a more refined or even optimal budget assignment remains future work (cf. Section 7.2.4).

The search proceeds in multiple rounds, one for each latency goal. Goals are considered in reverse degradation order. Each round consists of two steps. The heuristic first establishes a *starting point*, a budget assignment that guarantees finite response times without necessarily fulfilling the latency goal. Based on this initial budget, an iterative *refinement step* repeatedly increases individual executor budgets until the latency goal is fulfilled.

Step 1: finding a starting point. Algorithm 3 is used to find an initial budget assignment $bw(e)$ for each executor e that **(a)** has not received a budget during the previous rounds and **(b)**

Algorithm 3: The initial budget estimate.

```
1  for all influencing executors e without a budget do
2  |   needed ← Σ_{c served by e} rbf(c, horizon)
3  |   bw(e) ← needed / horizon
4  while there is a callback c with unbounded response time do
5  |   e ← executor serving c
6  |   needed ← rbf(c, horizon) − sbf(e, horizon)
7  |   if bw(e) = 1 then
8  |   |   degrade chain
9  |   else
10 |   |   bw(e) ← min(bw(e) + needed / horizon, 1)
11 if no partitioning for budget found then degrade chain
```

influences the response-time bound of the chain considered during this round. Each such executor receives an initial budget that reflects the maximum *longterm* processor demand of each executor. Since later steps will only add bandwidth but never remove it, the budgeting heuristic attempts to start the search with budgets that are as low as possible but still ensure finite response-time bounds.

The algorithm first estimates the executor's long-term demand by computing the cumulative demand over a long time interval called the *horizon* (Line 2). In our case study (Chapter 6), we arbitrarily chose a horizon of 10 seconds, which exceeded all latency goals and typical busy-window lengths. To break the aforementioned dependency cycle—actual request-bound functions depend on the budget of other executors—Line 2 is computed under the simplifying assumption that all *other* executors receive 100% budget. Line 3 configures the resulting initial budget estimate.

Having assigned an initial budget, the algorithm now drops the assumption that other executors have 100% bandwidth. As a result, some callbacks likely become unschedulable due to increased jitter effects. To cope, Lines 4 to 10 iteratively increase the bandwidth of the corresponding executors until the long-term demand at the horizon is met. If this requires bandwidths above 100%, the degradation process is started since it is impossible to guarantee bounded response times.

Algorithm 4: The assignment improvement heuristic.

```
1  res ← set of "upstream" executors that affect response times
2  while chain latency > goal do
3      for e in res do
4          d(e) ← Σ_{c served by e} (RT(c) − RT^100%(c))
5      for e in res by decreasing d(e) do
6          if bw(e) = 1 then
7              remove e from res
8              continue
9          bw(e) ← min(bw(e) + 5%, 1)
10         if partitioning for budget found then
11             break
12         remove e from res and restore old value of bw(e)
13 if no candidate found then degrade chain
```

The degradation process is also started if ROS-Llama cannot find a feasible reservation-to-processor mapping for the assignment (Line 11).

As discussed in Section 2.2.3, finding such a mapping requires solving a bin-packing problem. Each core is a bin with a capacity of one, and each reservation is an item with the reservation bandwidth as weight. ROS-Llama looks for solutions by trying the worst-fit-decreasing (WFD) and first-fit-decreasing (FFD) heuristics [64], in that order.

Step 2: refining the budget. Based on the initial budget assignment that (barely) achieves finite response-time bounds, Algorithm 4 refines executor budgets until the processing chain's response-time bound no longer exceeds the chain's configured latency goal (Line 2). To this end, ROS-Llama relies on what we refer to as the *budget-shortage delay* heuristic $d(e)$, which is the total increase in response time attributable to executors having less than 100% bandwidth (Line 4). Here, $RT(c)$ denotes the actual current response-time bound (Chapter 4), whereas $RT^{100\%}(c)$ denotes the response-time bound obtained by assuming 100% bandwidth. A large budget-shortage delay indicates that increasing the budget of this executor is likely to have large positive effects on response times in the system.

Following this heuristic, ROS-Llama considers the influencing executors in order of their

shortage delay (Lines 5 to 12). For each executor, the algorithm tries to increase the bandwidth by a fixed step size (Line 9) until the chain's latency goal is reached. We chose 5% as a compromise between the speed of convergence and the quality of the result. If no candidate for a budget increase can be found while the latency goal remains unmet, the degradation process is started (Line 13).

5.5 Summary

In this chapter, we presented ROS-Llama, the first automatic latency manager for ROS 2. Its design has been shaped by a careful analysis of the requirements and constraints of the ROS ecosystem (Section 5.1). To address these requirements, ROS-Llama operates largely automatically, based on just a simple and declarative configuration of high-level latency goals and the desired degradation order in case of overload.

To fulfill these latency goals, ROS-Llama relies on two main components: the model extractor, described in Section 5.3, and the budget manager, described in Section 5.4.

The **model extractor** observes the behavior of the running ROS system through a set of tracepoints in the core ROS libraries (Fig. 5.4). From the stream of events, it derives a timing model covering both ROS executors and event source threads (Sections 5.3.3 and 5.3.4).

In a dynamic system, the extracted timing model needs to deal with transition phases and adapt to changes in the workload. The model extractor therefore needs to be able to discard stale measurements taken during earlier states or transition periods. We proposed two adaptation heuristics, the initialization-phase heuristic and the data-aging heuristic (Section 5.3.5). The initialization-phase heuristic requires no configuration and automatically discards initialization behavior from the model. The much more powerful data-aging heuristic requires tuning by the user but can adapt to varying execution-time demands in the system.

The model is used by the **budget manager** to find a suitable scheduler configuration automatically. It schedules ROS executors using the SCHED_DEADLINE scheduler while isolating

essential system threads onto a separate system core (Section 5.4.1). The required budgets and core assignments are computed by an iterative budget assignment heuristic (Section 5.4.2).

Overall, ROS-Llama enables users to enforce their desired timing constraints on the system through real-time scheduling without requiring significant effort or expertise. In the next chapter, we evaluate the timing analysis and ROS-Llama on a realistic system and demonstrate that the proposed approach is effective in practice.

6 Evaluation

In this chapter, we empirically evaluate the proposed response-time analysis and the ROS-Llama latency manager. To test both mechanisms under realistic conditions, we chose a *Turtlebot 3* robot running off-the-shelf ROS packages as the main evaluation workload. The evaluation platform is described in detail in Section 6.1.

We first evaluate the response-time analysis. Since Chapter 4 already proved the correctness of the analysis, this chapter mainly evaluates the *effectiveness* of the analysis. Experiments with a synthetic workload and the realistic Turtlebot workload show that the round-robin and the busy-window analysis provided precision gains in different settings. We conclude that it is therefore advisable to combine both approaches in practice.

The experiments further confirm that ROS systems should be modeled using execution-time curves instead of traditional scalar worst-case execution times: the extracted Turtlebot workload is unschedulable in a WCET-based model, even if each executor runs on a dedicated CPU core. In a scaled-down version of the system, where all execution-time demands are reduced to 10% of their original value, execution-time curves still reduce response-time bounds by up to 75%.

In a second part, we evaluate the automatic latency manager ROS-Llama. The scope and goals of ROS-Llama make it difficult to prove its properties from first principles. Both correctness and suitability therefore need to be shown empirically. To this end, the evaluation aims to answer three main questions: Does ROS-Llama successfully keep the latency of the managed processing chains below the goal? Does the system degrade gracefully under overload? And finally, does ROS-Llama achieve these results at acceptable cost?

To answer these questions, Section 6.3 reports on an evaluation of ROS-Llama on the Turtlebot evaluation platform. The experiment let the robot drive through a repeatable scenario and recorded the observed response times. The results show that ROS-Llama controlled the latency of timing-critical chains better than comparable approaches. Although the observed runtime costs were substantial, ROS-Llama's improved latency management turned out to be worth the price.

The experiments further exposed that the chosen DDS middleware did not satisfy the implicit middleware requirements identified in Chapter 2. Our findings suggest that investigating the system interactions below the ROS level would be a promising next step in this line of work.

We conclude with an evaluation of the parameter choices taken in ROS-Llama. In Section 6.4, we investigate the effects of using different reservation periods and justify why 5 ms is a good choice for the fixed period length. We then turn toward the data-aging mechanism and investigate choices for the four aging parameters. The results demonstrate that the data-aging mechanism, properly configured, allows ROS-Llama to dynamically adapt to changing workloads while still providing stable bounds for stable workloads.

6.1 Evaluation Platform

We evaluated the timing model, response-time analysis, and ROS-Llama latency manager on a Turtlebot 3 "*Burger*" (Fig. 6.1) controlled by a Raspberry Pi 4B. The Raspberry Pi features an ARM A72 CPU with four cores clocked at 600 MHz.[1] The system ran a standard Linux kernel with the PREEMPT_RT patch (version *4.19.71-rt24-raspi2*). The ROS workload ran on ROS 2 "*Dashing Diademata*" and Eclipse's Cyclone DDS (version 0.5.1-1). In the "*Dashing Diademata*" version, timers are privileged (cf. Section 3.1.1).

[1] The processor also supports a 1.2 GHz setting. However, this frequency cannot be sustained in continuous operation due to overheating, quickly leading to unpredictable thermal throttling of the cores. As dynamic frequency scaling is beyond the scope of this dissertation, we focus on the stable 600 MHz setting.

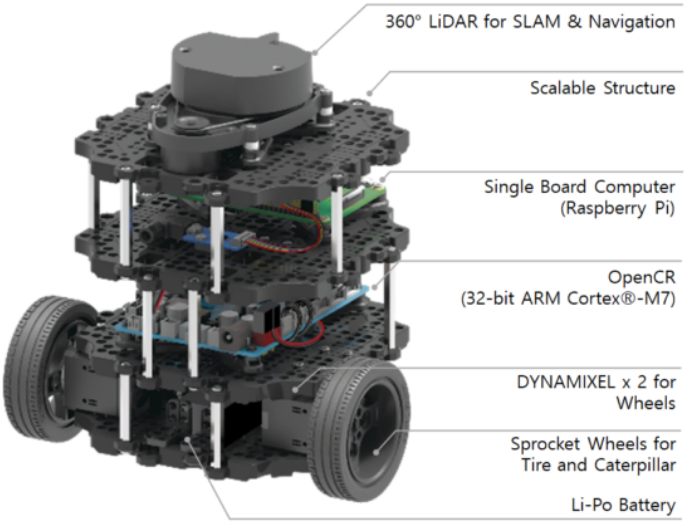


Figure 6.1: The Turtlebot 3 "*Burger*", our evaluation platform. Picture taken from the Turtlebot 3 e-manual [32].

6.1.1 ROS Components

The Turtlebot ran a workload comprising three open-source ROS components: a set of drivers for the Turtlebot, the ROS navigation stack, and an object tracker payload.

The **Turtlebot drivers** are two small packages that provide a ROS interface to the robot's hardware. They publish any measurements from the laser scanner on the `/scan` topic and odometry measurements on the `/odom` and `/tf` topics. Furthermore, they provide the `/cmd_vel` topic to submit velocity commands.

The **ROS navigation stack** [76] implements generic navigation primitives for wheeled mobile robots, including navigation planning, path following, self-localization, and cost-map management. It is a typical example of a standardized, generic, and reusable component in the ROS ecosystem that can be integrated into a wide range of robots.

The **object tracker** [33] follows a number of designated objects through a video sequence and serves as a typical example of a computationally demanding mission- but not safety-critical payload. In our setup, we simulated a camera by repeatedly playing the *car1* video taken from the VOT 2018 challenge [65, 66]. The object tracker is tasked with following cars through the

Table 6.2: Configured processing chains in reverse degradation order.

Name	Purpose	Length	Degradation Order	Goal (ms)
heartbeat	Keep-alive signal	1	(last) 7	100
pilot	Navigation & control	2	6	125
odometry-nav	Odometry (navigation)	2	5	75
laser-scanner	Sensor data acquisition	2	4	150
localization	Self-localization	1	3	450
odometry-loc	Odometry (localization)	2	2	100
tracker	Track objects	2	(first) 1	990

scene. To compensate for the large performance difference between the Raspberry Pi and the hardware recommended by the package developers (an Intel *i7-6700HQ* with four cores clocked at 2.6 Ghz), we downsampled the video to about one frame per second.

There are various tuning parameters to adapt the components to the available computational resources, physical characteristics of the robot, and navigation precision demands. We used the default configuration provided with the Turtlebot, except for the period of the local planner in the navigation stack, which we increased slightly from 100 ms to 125 ms as this proved sufficient for our scenario and induced less load. The increased period is still well within the supported parameter range for the planner [50].

6.1.2 Processing Chains

Within the described ROS system, we identified seven timing-critical processing chains. The chains are listed in Fig. 6.2. The last two columns show the configuration in the experiments.

The *heartbeat* chain simply manages a watchdog timer with a period of 100 ms that prevents the hardware from resetting. The first two functional chains are concerned with the movement of the robot's wheels. The *pilot* chain consists of a computation-intensive local planner callback responsible for computing the next motor command, followed by a shorter callback that encodes the command for transmission to the electric motor. The 125 ms latency goal ensures the motor receives the local planner's command once per period. The *odometry-nav* chain reports the measured wheel movements to the planner every 50 ms. We set a latency goal of 75 ms to ensure

that an odometry update arrives every period, *i.e.*, that the gap between two measurements, including the sampling delay of up to 50 ms, remains below 125 ms.

The next three chains cover the self-localization of the robot. The localization component relies on laser scans and an internal map to narrow down plausible estimates of the robot's current location. Since the laser moves with the robot, interpreting these scans also requires information about the robot's movement and orientation, *i.e.*, odometry. The two inputs are provided by the *laser-scanner* and *odometry-loc* chains. The *localization* chain covers the merging of the inputs and the computation of a position estimate.

Ideally, it would not have been necessary to define the *localization* chain separately, as it is conceptually just an extension of the *laser-scanner* and *odometry-loc* chains. However, the *message_filters* library used to implement the matching and merging of the inputs does not clearly expose the conceptual data flow at the callback graph level. The data flow therefore does not directly correspond to a single ROS callback chain.

To resolve this ambiguity, we configured the *localization* chain as a separate chain with its own latency goal. We will revisit the question of how to improve support for utility libraries like *message_filters* in the discussion on future work (Section 7.2.3).

The localization estimate expires after one second, counting from the time the underlying laser scan was taken. This imposes a timing constraint on the localization component: once the localization estimate expires, the robot cannot navigate and performs an emergency stop. We thus had to arrange for an end-to-end latency of at most one second between the laser scanner and the final localization callback.

The laser scanner rotates at 5 Hz, allowing it to produce one scan every 200 ms. In practice, we found that individual scans are occasionally transmitted incompletely by the hardware and cannot be interpreted. Accounting for such skipped scans yields a worst-case sampling delay of 400 ms, leaving 600 ms for processing. We assigned 150 ms to the *laser scanner* chain and 450 ms to the *localization* chain. For the odometry, we had to account for an additional 50 ms of sampling delay, leaving 100 ms for the *odometry-loc* chain.

Finally, the *tracker* chain covers the image tracker. The chain covers the (simulated) camera, which periodically acquires a frame (from disk) and sends it to the `/rgb` topic, and the tracker, which follows marked objects and outputs their position in the latest frame. The assigned latency goal ensures that every frame is processed before the next frame arrives, ensuring that the tracker does not fall behind under normal conditions. However, the *tracker* chain is also first in the degradation order, which reflects that its output is "nice to have" but not essential for the correct operation of the robot.

We stress that all latency goals derive purely from high-level functional considerations and hardware properties. They do not depend on detailed knowledge of component or system internals and would therefore be known to a system integrator.

6.2 Response-Time Analysis

We evaluated the response-time analysis in two case studies: a synthetic callback graph, designed to assess each analysis's advantages and disadvantages, and a real-world callback graph, to evaluate the analyses under realistic conditions. We compared three implementations: The round-robin analysis in Theorem 2 (*RR-only*), the busy-window analysis in Theorem 3 (*BW-only*), and the combined analysis, which computes the minimum of both approaches (*combined-analysis*).

Case study 1: synthetic workload. The first case study (Fig. 6.3) consists of a single executor containing only polled callbacks. One callback, c_0, is triggered in bursts of up to b activations at once. The bursts are separated by at least 10 ms. A second set of callbacks, c_1 to c_l, forms an intra-executor chain of length $l = 6$. Callback c_1 is activated by f callbacks located in the same executor, each with an activation curve that mandates a $10\,\mu s$ distance between any two activations but 10 ms between any three activations. We call the parameter f the *fan-in*. For simplicity, the setup uses scalar WCETs only. The callback c_0 has a WCET of $10\,\mu s$, each callback in the chain has a WCET of $50\,\mu s$, and the f predecessor callbacks of c_1 have a WCET of 1 µs each. The executor is provisioned with a periodic supply of 700 µs every one ms.

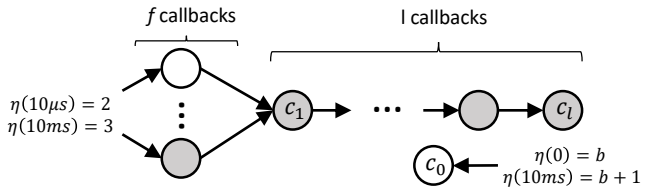
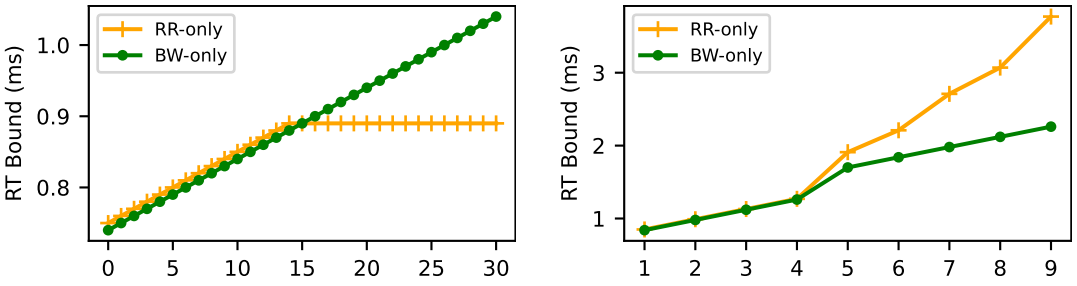


Figure 6.3: Synthetic setup. The chain under analysis is shaded gray.



(a) By burst length of c_0.

(b) By fan-in.

Figure 6.4: Response-time bound of the chain (synthetic workload).

The experiment varied two parameters: b, the burst length of c_0, and f, the fan-in. As the parameters varied, we observed the response-time bound of the chain marked in Fig. 6.3.

Figure 6.4a shows the response-time bounds reported by the *RR-only* and *BW-only* analyses as b changes (for a fixed fan-in $f = 1$). Since *combined-analysis* is simply the minimum of *RR-only* and *BW-only*, it is not shown separately. The plot shows a steadily increasing response-time bound for *BW-only*. In contrast, the *RR-only* bound stays flat after $b = 14$. At this point, all processing windows involved in the chain are saturated with instances of c_0. Activating more instances of c_0 therefore does not increase the interference suffered by the chain. The results show the importance of accounting for starvation freedom, as the *BW-only* analysis significantly overestimated the interference from c_0.

In a second experiment (Fig. 6.4b) we varied f, the fan-in, for a fixed burst length of $b = 10$. The plot shows that all variants achieved similar bounds at $f = 1$. As the fan-in increased, the *BW-only* response-time bound also increased in reaction to the increased activation rate of the chain. The *RR-only* analysis started out similarly but suffered a rapid increase starting at $f = 5$. At $f = 9$, the *RR-only* bound was already twice as large as the *BW-only* bound.

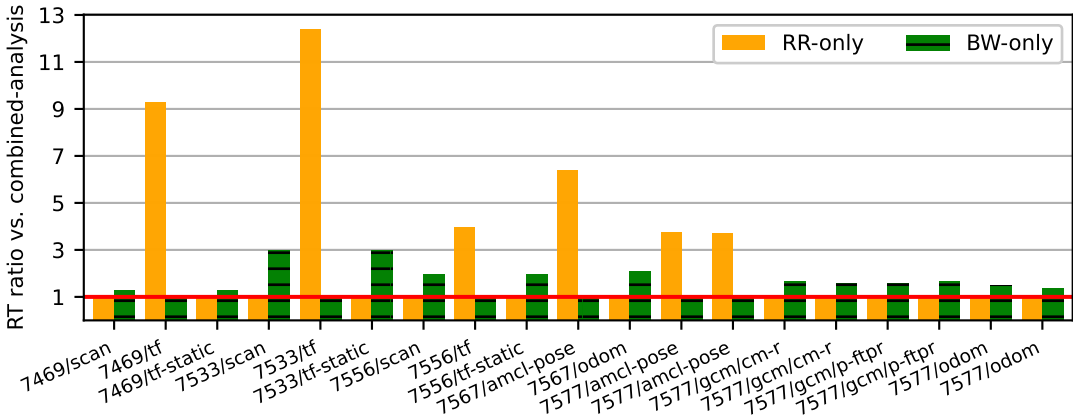


Figure 6.5: Response-time bound of callbacks compared to the proposed analysis (lower is better).

Overall, the two experiments show that both analysis approaches, *RR-only* (Theorem 2) and *BW-only* (Theorem 3) excel in different situations. To ensure low response-time bounds in both situations, it is therefore advisable to try both approaches and take the minimum of the two bounds.

Case study 2: real workload. In the second case study, we evaluated the analyses on a callback graph of the Turtlebot evaluation platform, which we extracted using the ROS-Llama model extractor. Each executor was provisioned with 100% bandwidth.

Figure 6.5 shows the results for the 19 (out of 54) callbacks in the system for which the response-time bounds are not trivial (e.g., due to lack of interference or extremely sparse activations). Since the absolute values of the bounds varied heavily between the callbacks, we instead show each bound as a ratio normalized by the *combined-analysis* bound. A ratio of 1 (marked by the red line) indicates that the analysis produced the same bound as the combined analysis; ratios above one indicate that the analysis produced a higher bound than the combined analysis. Since the *combined-analysis* bound is the minimum of the two depicted approaches, one of the two bars is always at $y = 1$, *i.e.*, equal to *combined-analysis*.

The figure confirms that the combined analysis improves significantly over both of its constituent approaches. For some callbacks (*e.g.*, *7533/scan*), the round-robin analysis produced the

tighter bound, beating the *BW-only* bound by a factor of 3. In other callbacks (*e.g.*, *7533/tf*), the *BW-only* analysis proved more effective, beating the *RR-only* analysis by a factor of 12.

Unsurprisingly, the *BW-only* analysis showed the greatest benefits in callbacks that are frequently invoked, particularly the various */tf* callbacks. Such callbacks need to account for large amounts of self-interference and therefore do not benefit from the bound on processing windows. The round-robin analysis primarily benefits callbacks that share an executor with one of the frequently-invoked callbacks (*e.g.*, the two other callbacks in executor 7533).

The importance of execution-time curves. Analyzing the extracted timing model also presented an opportunity to evaluate the importance of modeling execution times as execution-time curves instead of worst-case execution times. To this end, we compared the response-time bounds produced by the combined analysis to a WCET-only variant. The WCET-only variant truncated execution-time curves after $ET(1)$ and then extrapolated the curve linearly.

The resulting WCET-based model of the Turtlebot system turned out to be unschedulable. Even with each reservation receiving a budget of 100%, the analysis did not converge within the analysis horizon (10 s).

This observation already demonstrates the importance of using execution-time curves. However, it makes it difficult to quantify the difference between the two models. We therefore scaled down the execution times in the model linearly until the response-time analysis converged for the WCET-based system, which happened at a factor of 10%.

Figure 6.6 shows the response-time bound ratio for this downscaled system. Similar to Fig. 6.5, the combined analysis with execution-time curves serves as a baseline. A bound ratio of $y = 1$ (marked by the red line) indicates that the bounds are equal; a bound ratio above the x-axis indicates that the execution-time curve variant is better.

The figure shows that execution-time curves improved the response-time bounds by a factor of 10 and more in 2 out of 19 cases, with two more callbacks achieving improvements just slightly below 2. In the most extreme case (*7469/tf*), execution-time curves improved the bound by almost

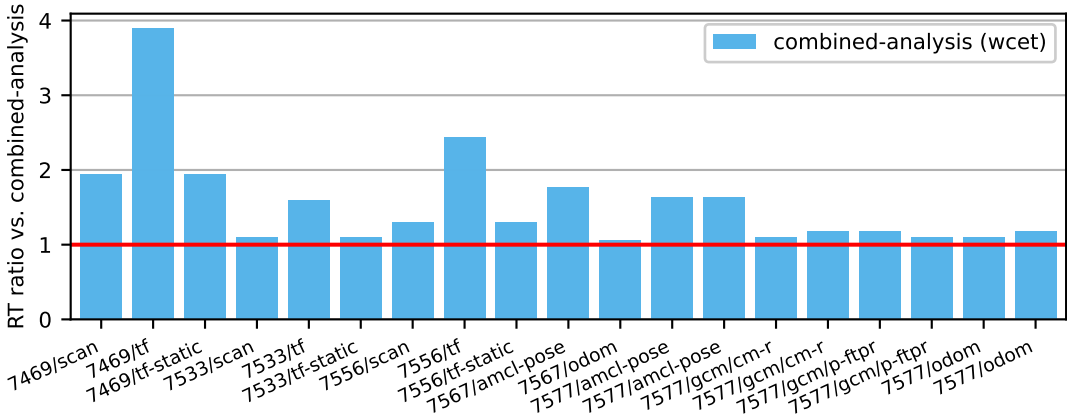


Figure 6.6: Response-time bound of callbacks modeled with scalar WCETs compared to execution-time curves (lower is better). Execution times are scaled to 10% of their original value.

a factor of 4. This callback is therefore likely responsible for the non-convergence of the analysis at higher scaling factors. Among the other 15 callbacks, four more callbacks improved by a factor of about 1.5, while the reminder improved by a factor of just slightly above one. Overall, the experiment confirms that a response-time analysis for ROS systems requires a more sophisticated execution-time model than scalar worst-case execution times.

6.3 Automatic Latency Management

Having evaluated the response-time analysis, we now turn towards the automatic latency manager ROS-Llama. This section aims to determine whether ROS-Llama fulfills the requirements in Section 5.1. Specifically, we aim to answer three main questions: does ROS-Llama ensure that the latency of the designated time-critical processing chains remains within the latency goal? Does ROS-Llama ensure the graceful degradation of the system as one of the managed chains starts consuming more and more computation time? And finally, how much overhead does ROS-Llama impose?

We answer these questions by letting the evaluation robot drive through a repeatable test scenario. In the scenario, the Turtlebot patrolled between two fixed locations while tracking a

number of objects in the video stream. The experiment progressed through three phases: *no load*, *normal load*, and *high load*. In the first phase, the object tracker did not follow any objects. In the following phases, the number of objects to track increased to simulate the effects of an increasingly crowded environment. This increased the execution time of the tracker from a barely noticeable to an unsustainable load that forced video frames to be discarded. We observed the impact of this demand increase on the other chains.

Each experiment ran for 150 seconds and was repeated four times. During this time, we recorded the beginning and the end of the processing chains described above and checked whether the chain completed in time. The results are compared against two baselines:

Baseline 1: CFS. The first baseline is a standard Linux setup, where threads are scheduled globally across all four cores using CFS without any of the ROS-Llama infrastructure present. This is the default ROS setup and arguably the only other available choice that does not require real-time expertise on behalf of the system integrator and component developers. It provides a fair baseline with regard to Req. IX (*i.e.*, the question of whether the cost of running ROS-Llama outweighs its benefits) since it does not incur any of ROS-Llama's overhead.

Baseline 2: SCHED_RR. The second baseline is a primitive variant of ROS-Llama that does not use response-time analysis. Instead, it schedules latency-critical executors with the SCHED_RR fixed-priority scheduler. It aims for graceful degradation by assigning priorities according to the degradation order, *i.e.*, it prioritizes executor threads serving chains later in the degradation order over executors serving exclusively chains earlier in the order. This is arguably the most straightforward approach to implement controlled degradation without analysis, but still cumbersome and error-prone to realize manually[2] as it requires correct identification of all callbacks, chains, and threads. We use this baseline to evaluate to what extent the response-time analysis improves the decisions of ROS-Llama. We use SCHED_RR rather than

[2]That is, without a tool providing automatic introspection capabilities such as those provided by ROS-Llama's automatic model extractor.

Figure 6.7: Number of goal violations per chain.

	ROS-Llama		CFS		SCHED_RR	
	violations	count	violations	count	violations	count
Name						
heartbeat	0	7,989	0	7,979	0	7,975
pilot	0	5,712	32	5,480	66	5,429
odometry-nav	0	15,980	0	15,959	105	15,946
laser-scanner	0	3,968	0	3,969	5	3,961
localization	0	3,968	0	3,969	0	3,958
odometry-loc	0	15,980	0	15,959	76	15,946
tracker	211	809	175	808	179	808

SCHED_DEADLINE for this baseline because, without analysis, we have no way of assigning sensible reservation budgets.

6.3.1 Latency Goal Compliance

We first evaluate how well the three system configurations (ROS-Llama, CFS, and SCHED_RR) controlled the latency of the considered chains. Figure 6.7 shows the number of observed chain completions with end-to-end latency exceeding the configured goal. In all configurations, the *tracker* chain exceeded its bound frequently (between 175 and 215 times out of about 800 activations). These violations are expected as a result of the unsustainable load in the last phase. Controlled degradation should ensure that this overload does not affect the other chains. Still, under both baselines other chains exceeded their goal latency. The *pilot* chain violated the latency goal 32 and 66 times under CFS and SCHED_RR, respectively. SCHED_RR further led to large chain violations in the two odometry chains and the *laser-scanner* chain.

In contrast, ROS-Llama successfully protected the more critical chains from undue interference by the *tracker* chain, degrading the system gracefully in the face of the surge.

While there was a notable difference in the total number of completed *pilot* instances, these differences were, as far as we can tell, not related to timing issues. The robot's software temporarily disables the *pilot* chain if it cannot find a path to the configured goal. This can happen

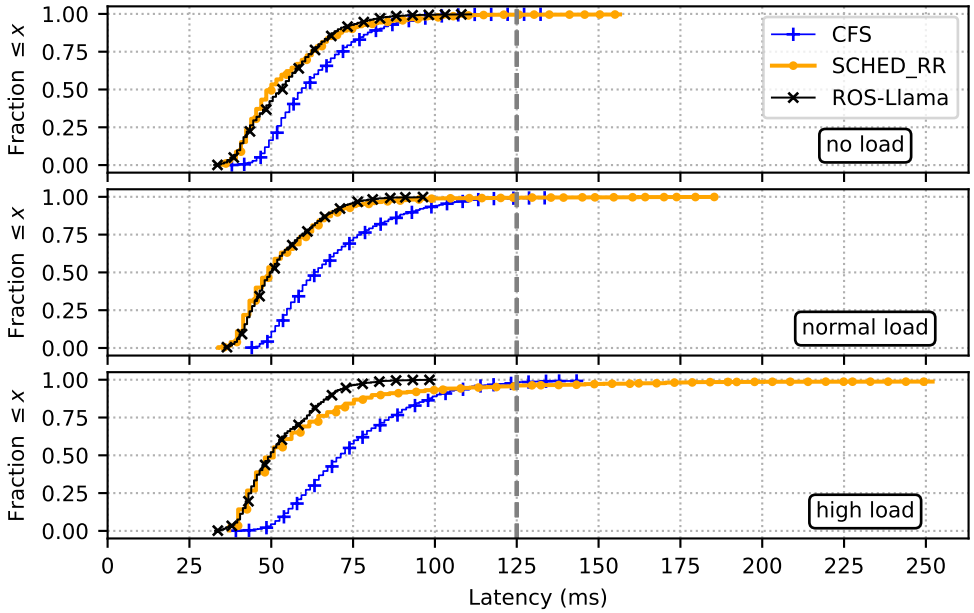


Figure 6.8: CDFs of the latency of the *pilot* chains, separated by phase. ROS-Llama keeps the tail latency below the 125 ms goal (dashed line), even under heavy load. Both baselines exhibit tail latencies in excess of the goal.

due to inaccuracies in self-localization, for example. If the robot finds itself "stuck", for example because it believes that the path to the objective is blocked by a perceived obstacle, it can take some time until the error is corrected.

The latency distribution in detail. To understand the reasons for the observed latency goal violations, we investigated the affected chains in more detail. Figure 6.8 shows a CDF of the observed end-to-end latencies in the *pilot* chain, separated by phase. The dashed vertical line marks the latency goal: the goal is always met if a curve's points are all to the left of this line (*i.e.*, if the observed maximum end-to-end latency does not exceed the chain's latency goal).

The first observation is that both baselines exceed the latency goal in the worst observed case. Only ROS-Llama ensures that the *pilot* chain completes within 125 ms throughout the experiment.

Looking at the results in more detail, we further observe that the CFS curve grows wider as the load increases, indicating that high-latency results become more prevalent. For example, while

less than 25% of activations exceed 75 ms in the first phase, almost 50% do under high load. In the last phase, this widening curve also results in higher peak latency, reaching almost 150 ms. In contrast, the ROS-Llama curve does not visibly change throughout the phases.

These observations demonstrate the risk posed by CFS's lack of temporal isolation: the non-essential *tracker* chain is *functionally* completely unrelated to the critical *pilot* chain, but still the overload experienced in the *tracker* chain heavily impacted the observed end-to-end latency of the critical chain, increasing both median and worst observed latency by over 10 ms.

The root cause is that both chains contain computationally intensive callbacks, and hence both are entitled to an equal share of the available resources under the default CFS timesharing policy. Oblivious to their respective latency requirements and their relative importance to the robot's overall correct operation, CFS has no way of inferring which of the two components it should prioritize and, as a result, *both* chains exhibited latency goal violations. Fair sharing of resources, the core principle underlying CFS, is obviously and demonstrably not the appropriate policy under transient overload conditions.

As one might hope, the SCHED_RR baseline kept latency more stable *most* of the time. However, we also observed large outliers in the *pilot* chain that exceeded the latency goal by over 100 ms. This is surprising given that, according to the degradation order, the *pilot* chain should have higher priority than the overloaded *tracker* chain (the chains do not share any executors).

An even clearer example of this issue is the *odometry-loc* chain (Fig. 6.9). While both CFS and ROS-Llama completed the chain within 60 ms at most, the chain suffered latency spikes of hundreds of milliseconds under SCHED_RR.

There are even fewer confounding factors in this case: the chain consists of two callbacks, neither of which performs heavy computation. The first callback in the chain extracts the position and state of the wheel joints from the robot hardware and transmits it to the second callback. The second callback, which is alone in its executor, receives this message, applies a few simple transformations and transmits the resulting odometry data. In such a simple and low-demand setup, latency spikes of hundreds of milliseconds should be impossible.

To further investigate the source of the latency spike, Fig. 6.10 shows the distribution of the delay between the transmission of the sensor data and the activation of the processing callback. Since the timestamp marking the start of the chain was taken after the sensor data was published, the depicted latency *underestimates* the true latency and might even be negative.

One can observe the same spikes as in the full *odometry-loc* chain. Yet neither of the two involved executors can be responsible for this delay. The receiving executor only served a single callback, which is therefore unaffected by intra-executor interference. The only sampling delay on the receiver side is thus self-interference. Measurements from the model extractor show that the receiver callback was not active during the spike, though, which rules out self-interference as a factor.

The sending executor cannot be at fault either, since the latency depicted in Fig. 6.10 is counted from a point in time when the message has already been published. The excessive latency spike therefore has to be the product of the DDS middleware layer.

The cause of the delay is subtle and can be understood as a kind of "collateral damage": because the *tracker* chain was overloaded *at real-time priority*, it induced a bursty overload also into the DDS layer, which in turn negatively affected the intra-chain communication delays of the *pilot* and *odometry-loc* chains. The ROS communication layer evidently does not cope well with overload situations at real-time priority. We will further discuss this phenomenon in Section 6.3.3.

This shows that graceful degradation is essential for ROS systems. ROS-Llama prevents such DDS overloads by recognizing in advance that the load produced by the *tracker* chain is unsustainable and by *proactively* degrading the overloaded chain before it can harm other chains.

Conclusion. Our case study shows that neither avoiding real-time scheduling altogether (the default CFS baseline) nor assigning real-time priorities in a "blind", purely heuristic-driven manner without a backing analysis (the SCHED_RR baseline) leads to satisfactory results.

The CFS baseline performed well for most callbacks but failed to meet the latency goal of the computation-heavy *pilot* chain under load. It shared the available supply equally between the

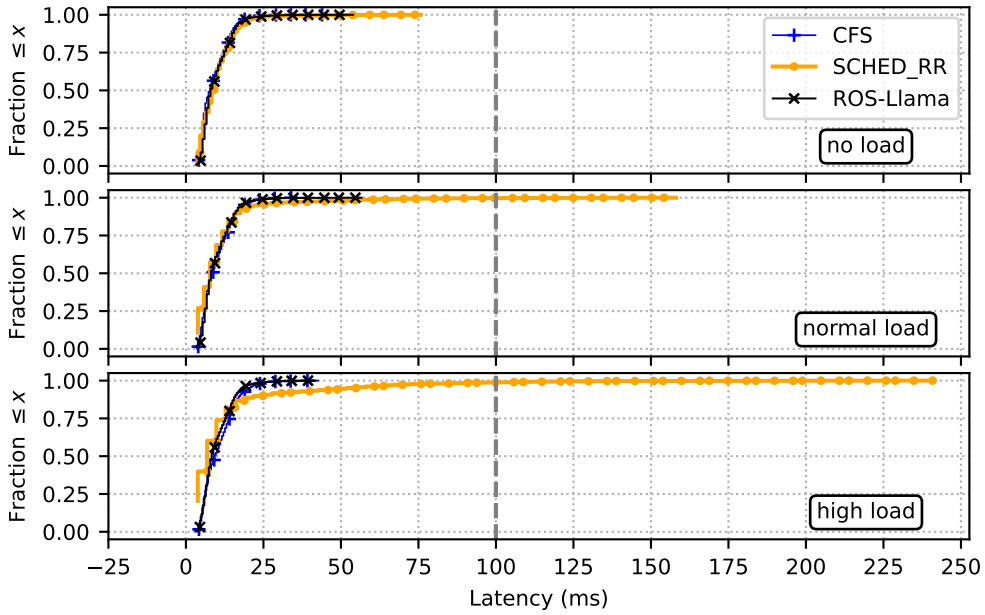


Figure 6.9: CDFs of the latency of the *odometry-loc* chains, separated by phase. ROS-Llama and the CFS baseline comply with the 100 ms latency goal, whereas the SCHED_RR baseline exhibits a large tail-latency spike.

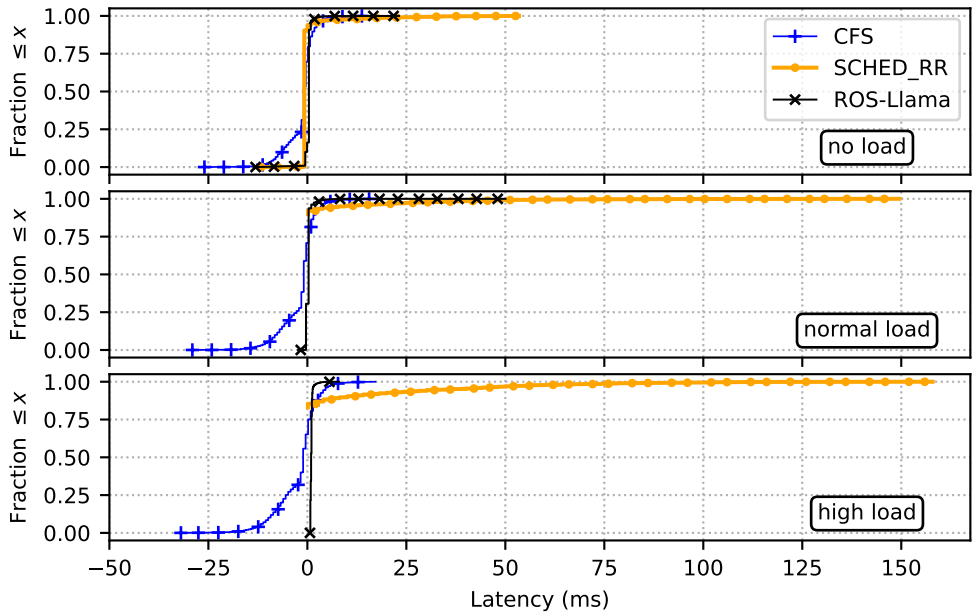


Figure 6.10: CDFs of the transmission delay in the *odometry-loc* chain, separated by phase. The delay measurement method underestimates the true delay, and might therefore be negative.

Figure 6.11: Average ROS-Llama overhead by component per phase.

Phase	No load	Medium load	High load
Model Extractor (Go)	0.95 sec (51%)	1.11 sec (42%)	1.15 sec (42%)
Model Processing (Python)	0.37 sec (20%)	0.44 sec (17%)	0.46 sec (17%)
Budget Selection (Python)	0.05 sec (3%)	0.11 sec (5%)	0.11 sec (4%)
Timing Analysis (Rust)	0.49 sec (26%)	0.97 sec (37%)	1.00 sec (37%)
Total	1.86 sec (100%)	2.63 sec (100%)	2.71 sec (100%)

pilot and the *tracker* chains and thereby completed neither chain in time.

ROS-Llama, in contrast, exploited the capabilities offered by Linux's SCHED_DEADLINE scheduler to isolate the *pilot* chain from the overloaded *tracker* chain. It achieved these results with only little additional effort compared to the CFS baseline and without requiring any expertise in real-time scheduling: all required information was collected autonomously through introspection and response-time analysis.

However, the DDS overload observed in the SCHED_RR baseline shows that the ROS communication layer may deviate from the basic middleware conditions under overload conditions. Although ROS-Llama successfully prevented these overloads, the results stress the importance of selecting a well-behaved communication middleware. We revisit this issue in Section 6.3.3.

6.3.2 ROS-Llama Runtime Costs

ROS-Llama's memory footprint is negligible relative to the footprint of ROS. In our discussion of runtime overheads, we hence focus on processor time. Figure 6.11 reports the average per-invocation cost of ROS-Llama's components. Each entry shows the processor time in seconds spent in each component per ROS-Llama activation, averaged over all activations during the respective phase. The percentage number in parentheses reports how much of the total average overhead was spent in each individual component. Recall that we configured ROS-Llama to recompute the budget every six seconds; in total, ROS-Llama as a whole thus consumed 30–45% of one CPU.

The lower analysis- and budgeting overhead in the first phase results from ROS-Llama op-

portunistically reusing cached budget assignments in a low-load scenario. The model extractor, which runs continuously alongside the system, accounts for about 40–50% of the total cost of ROS-Llama. The cost of preparing the timing model for analysis causes about 20% of the overhead. The remainder of the runtime costs are due to the budget selection process, separated into the budgeting heuristics (≈5%) and the response-time analysis (≈25–35%).

Conclusion. The results show that ROS-Llama introduces a noticeable overhead. However, despite this overhead, ROS-Llama ensured better latency goal compliance and more graceful degradation behavior than the CFS baseline. In other words, ROS-Llama satisfies the *earn-your-keep* requirement (Req. IX): it comes with significant costs but provides sufficient benefits to realize a favorable trade-off between performance and predictability.

In the current prototype, the model extractor causes most of the overhead due to our unoptimized tracing implementation. Integrating ROS-Llama with a mature tracing system like LTTng [36] would likely lower the cost of running ROS-Llama.

6.3.3 Unpredictable Middleware Implementation

A surprising discovery during the experiments was that the implementation of the DDS middleware violated some of the basic middleware requirements that we expect from a middleware for timing-critical communication. Recall the three requirements from Section 2.1.2: **(a)** communication is reliable, **(b)** executors dominate latency, and **(c)** executors read their own writes. We observed violations of both (b) and (c).

The first requirement violation we noticed is that messages sometimes get "stuck" in transmission for long periods of time, sometimes over 100 ms. As demonstrated in Section 6.3.1 (Fig. 6.10), the delay cannot have been caused by a lack of supply in the executors but instead stemmed from the communication layer. The latency was thus not necessarily dominated by the executors. Since the DDS transmission threads are the highest-priority threads on the system core, we conjecture that the delay results from shared locks and possibly priority inversion.

A likely cause is the complex interaction among DDS threads and between DDS threads and the associated executors. These threads all use locks to synchronize access to critical sections in the DDS implementations. Locks are well-known to cause outsized or even unbounded priority inversion unless developers keep critical sections short and use appropriate locking protocols. This leaves ROS-Llama in a conundrum: The limitations in the Linux scheduler force it to isolate the DDS threads on a separate core, yet the only available lock sharing protocol, the priority inheritance protocol [106], does not guarantee bounded priority inversion in partitioned multicore settings [18].

The second requirement violation we noticed is that executors sometimes did not read their own writes. This behavior has also been observed in concurrent work [116]. Such a delay might leave space for a quiet time during the execution of an intra-executor chain, which violates an assumption of the busy-window-aware chain analysis. It further reveals complex and fragile interactions among the DDS threads, even within a single process.

In summary, it has become apparent that the employed DDS implementation may violate basic middleware requirements under overload conditions. During those periods, the response-time analysis may temporarily not apply. To avoid these intermittent losses of latency control, the system integrator should ensure the robot uses an analyzable and predictable middleware. Identifying such an implementation among the various middlewares supported by ROS and identifying and documenting its real-time properties remains future work.

6.4 Tuning the Reservation Period

As described in Chapter 2, a SCHED_DEADLINE reservation is characterized by two parameters: the budget and the period. ROS-Llama selects the budget according to the execution-time needs of the reservation. However, the right choice for the period is less clear. As discussed in Section 5.4, ROS-Llama therefore chooses a fixed period of 5 ms for all reservations.

Like many other parameters, choosing the right reservation period is a trade-off. On the one

hand, shorter periods reduce the reservation blackout window and guarantee prompter supply delivery. For example, consider two reservations, r_a and r_b, with the same bandwidth of 50% but different periods: r_a guarantees a budget of 5 ms every 10 ms, while r_b guarantees a budget of 2.5 ms every 5 ms. Then r_a is guaranteed to provide a millisecond of computation time after 11 ms (*i.e.*, $sbf_a(11\,\text{ms}) = 1\,\text{ms}$), while r_b already supplies a millisecond of budget within 6 ms (*i.e.*, $sbf_b(6\,\text{ms}) = 1\,\text{ms}$). Choosing a shorter period thus strengthens the supply guarantee at no cost in processor bandwidth.

On the other hand, shorter periods increase scheduling overhead. The shorter the reservation period, the more frequently the OS scheduler needs to switch between the reservations to fulfill the supply guarantees. At some point, such overheads become non-negligible. ROS-Llama thus needs to choose a reservation period that is as short as possible but still entails negligible overheads.

We evaluated this trade-off on a synthetic ROS model. The system consisted of 10 timers, each served by a dedicated executor. The executors were all assigned to the same CPU; each received a bandwidth of 10% while the period varied between 0.1 ms and 10 ms. Figure 6.12 compares the measured response time of the first timer[3] as a boxplot. Each box covers the range between the 25-percentile and the 75-percentile, with the median marked with an orange line. The whiskers show the minimal and maximal observed response times. The dashed line marks the response-time bound predicted by the analysis.

The plot shows the two competing effects discussed above. The response-time bound grows larger as the period increases due to the weakening supply guarantee. Towards shorter periods, the graph shows the harmful effects of scheduling overhead. Starting with a period length of 2 ms, such overheads become significant enough that they can no longer be neglected by the response-time analysis. As a result, the observed response time exceeds the computed bound. ROS-Llama has to choose a period that stays clear of these short periods to ensure the response-time analysis remains applicable. We chose 5 ms as the default period; given that 2 ms is the highest period for

[3]Since the system setup is symmetrical, it makes no difference which timer is measured.

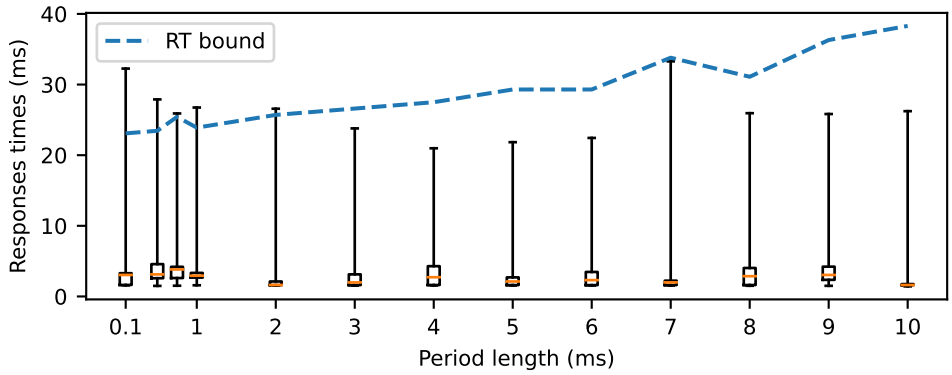


Figure 6.12: Observed response time under various periods.

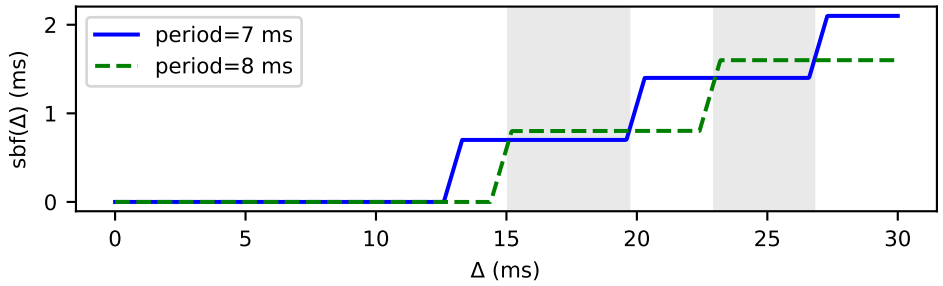


Figure 6.13: Supply-bound functions for two reservations with the same bandwidth (10%) but different periods. The reservation with the longer period has a longer blackout window but still surpasses the other reservation's supply in the shaded areas.

which we observed the response time to exceed the bound, this provides a safety margin of 3 ms (*i.e.*, a factor of 2.5).

Figure 6.12 further hints toward another benefit of a better period choice, as the response-time bound curve is not monotonic. A period of 8 ms, for example, yields a lower response-time bound *and* lower scheduling overheads than a period of 7 ms.

The reason for this effect can be observed in Fig. 6.13, which shows the supply-bound function for the two reservations. Although the 8 ms-reservation (dashed line) exhibits a longer blackout window, it guarantees more supply than the 7 ms-reservation for some interval lengths (shaded in gray). If the worst-case demand of a reservation's callbacks lies in the space between the solid and the dashed line during one of those shaded areas, the response-time bound for the

8 ms-reservation will be lower than the response-time bound for the 7 ms-reservation.

These results suggest that ROS-Llama might be able to achieve both lower response-time bounds and lower scheduling overheads with a more dynamic period assignment scheme. We will revisit this avenue for future work in Chapter 7.

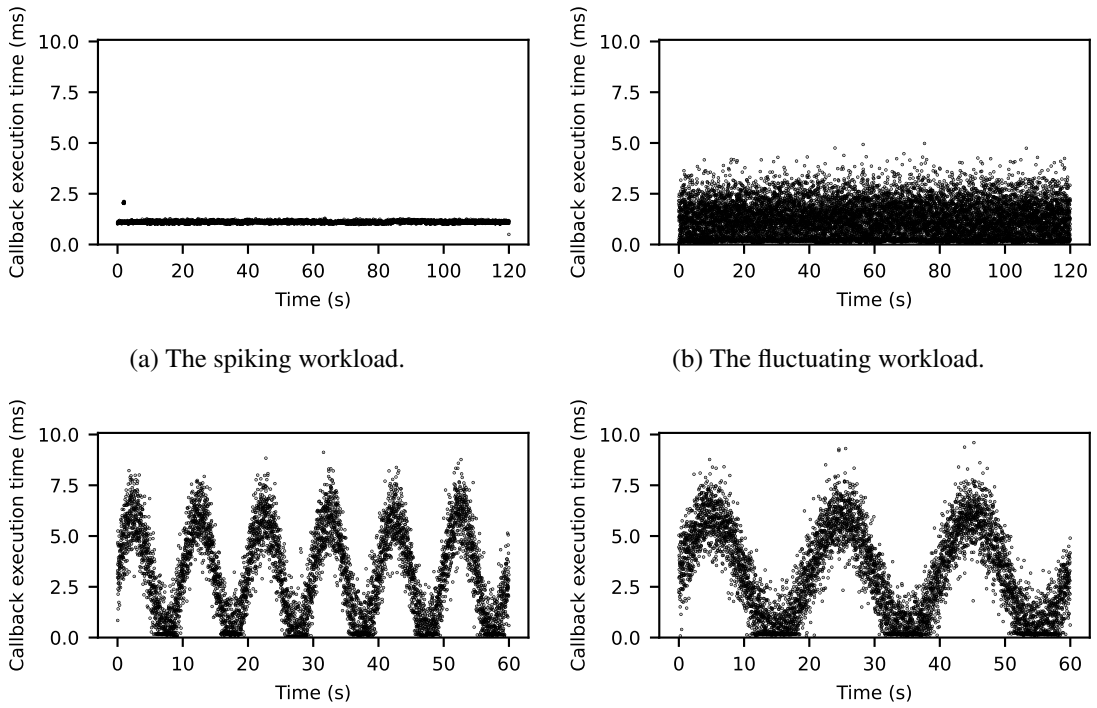
6.5 Data Aging

Recall from Section 5.3.6 that data aging is configured with four parameters: the merging period T, the merging weight α, the trigger threshold G, and the safety margin S. Unfortunately, it is not obvious how to configure these parameters since the configuration has to strike a balance between two conflicting goals. On the one hand, an execution-time curve should reliably upper-bound the processor time consumed by its callback (it should have *predictive power*). On the other hand, an execution-time curve should not be overly pessimistic and should gradually adapt to reductions in execution-time demand (it should be *adaptive*).

The right balance between predictive power and adaptivity depends on both the workload and the requirements of the robot. In the current version, ROS-Llama therefore leaves the configuration of this mechanism to the system integrator. As a first step towards a more automated solution, this section evaluates how well ROS-Llama's data-aging mechanism allows the system integrator to achieve the desired trade-off between the two goals.

Evaluation setup. We evaluated data aging on a synthetic example, consisting of a periodic event source with period 10 ms and a pre-defined execution-time demand over time. Such a synthetic benchmark is precisely reproducible and allows us to isolate the effects of data aging from any external measurement noise or variation in system behavior. It thus serves as an initial feasibility study for the data aging mechanisms and lays the groundwork for future exploration.

In this evaluation, we consider four execution-time demand patterns for the event source (Fig. 6.14): a *spiking* pattern, a *fluctuating* pattern, a *rapidly oscillating* pattern, and a *slowly oscillating* pattern. Each of the patterns exercises a different aspect of the data-aging mechanism.



(a) The spiking workload.

(b) The fluctuating workload.

(c) The rapidly oscillating workload (0.1 Hz).

(d) The slowly oscillating workload (0.05 Hz).

Figure 6.14: Consumed processor time per invocation for the four scenarios.

In the **spiking workload**, the execution-time demand remained at a fixed level, except for a brief spike in the beginning, during which the demand doubles. In our evaluation scenario, the event source busy-waited for approximately one millisecond in each iteration. At some point (between 1.8 and 2.1 seconds into the experiment), the event source consumed twice as much computation time and reverted to the original runtime afterward.

A non-adaptive execution-time curve would account for the spike's peak demand during the entire experiment. Data aging allows the tracer to discount the spike and gradually revert to the original bound. The spiking workload thus evaluates how effectively the data-aging mechanism exploits changes in demand.

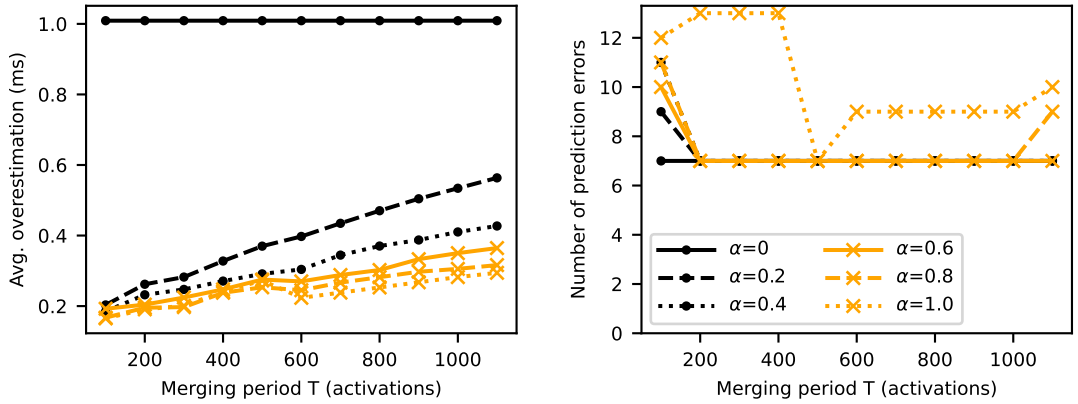
In the **fluctuating workload**, the execution-time demand was drawn from a static distribution. In our evaluation scenario, the event source drew the execution-time requirements from a Gaussian distribution with a mean and standard deviation of 1 ms. The execution-time behavior thus

fluctuated between iterations but was drawn from the same distribution throughout the experiment. Attempting to adapt to temporary and random lulls in the demand would lead to prediction errors. This workload thus evaluates whether the data-aging mechanism is robust to noise.

The two **oscillating workloads** combine random noise, which the data-aging mechanism should *not* adapt to, with a slowly-changing demand, which the data-aging mechanism *should* adapt to. The system integrator should be able to configure ROS-Llama to adapt to the slow, long-term demand changes but not to any higher-frequency fluctuations.

In both oscillating workloads, the event source drew the execution-time requirement from a Gaussian distribution with a standard deviation of one millisecond and a time-varying mean. The mean followed a sine curve that oscillated around the center point 3 ms with an amplitude of 3 ms and a frequency of 0.1 Hz for the rapidly oscillating workload and 0.05 Hz for the slowly oscillating workload.

We ran each of the workloads for two minutes, corresponding to 12,000 activations. To ensure that the results depend only on the data-aging parameters, we executed each workload once, recorded the produced trace stream, and replayed it for each considered set of data-aging parameters. The tracer thus received the same data every time. In the following, we first evaluate how the choice of α and T influences the effectiveness of the data-aging mechanism for the spiking workload (Section 6.5.1). In a second step, we evaluate how those same parameters fare under the fluctuating workload and demonstrate how the safety margin and the trigger threshold can be used to prevent overfitting to noise (Section 6.5.2). Finally, we evaluate the parameters under the oscillating workload. We demonstrate which parameters strike a balance between exploiting demand changes and protecting against overfitting and explore how well the data-aging mechanism is able to discriminate between low-frequency demand changes and high-frequency noise (Section 6.5.3).



(a) Adaptivity.

(b) Predictive power.

Figure 6.15: Data-aging performance on the spiking workload ($G = 90\%$, $S = 0$).

6.5.1 Spiking Workloads

In our evaluation of the spiking workload, we evaluated various settings of the two parameters T and α in terms of their adaptivity and predictive power (Fig. 6.15). G and S are fixed at 90% and 0%, respectively. The case $\alpha = 0$, which disables data aging entirely, serves as a baseline.

The adaptivity (Fig. 6.15a) is measured through the average execution-time overestimation, *i.e.*, the average difference between $ET(1)$ and the actual execution time of the instance. Cases where the actual execution time was higher than $ET(1)$ are counted as zero for this purpose. Successful adaptation leads to tighter bounds and thus lower average overestimation, while overly timid adaptation leads to more pessimistic bounds and a higher average overestimation.

The predictive power (Fig. 6.15b) is measured by counting how many callback instances exceeded the current execution-time bound at the time of their activation. A successful adaptation should produce as few prediction errors as possible.

Compared to the baseline $\alpha = 0$, data aging resulted in significantly lower execution-time overestimation in the spiking workload scenario. The non-adaptive algorithm never recovered from the spike and overestimated the execution time by the size of the spike (1 ms) during the entire experiment. In contrast, choosing $\alpha = 0.8$ and $T = 200$ reduced the average overestimation by 80% without increasing the number of prediction failures.

Further note how both shorter merging intervals and lower merging weights had a similar effect of reducing overestimation: a merging weight of $\alpha = 0.2$ at $T = 300$ yielded a similar overestimation reduction as a merging weight of $\alpha = 0.6$ at $T = 700$. The reason is easy to see: every time the short-term curve is merged into the execution-time curve, older measurements are diluted. Reducing the merging period hence increases the adaptivity unless the merging weight is reduced to compensate.

The number of prediction errors in the spiking workload was the same as in the non-adaptive case for most weights. However, for $\alpha \geq 0.8$ and $T = 100$, configurations appear to overfit to random noise in the callback execution times. These settings are clearly too aggressive.

6.5.2 Fluctuating Workloads

Figure 6.16 compares both the adaptivity and predictive power under varying values of α and T. The adaptivity was again measured through the average overestimation (top row), while the predictive power was measured through the number of prediction errors (bottom row). We first consider only $S = 0\%$, which is shown in the left column of the figure.

The non-adaptive setting ($\alpha = 0$) achieved an average overestimation close to 3 ms, *i.e.*, three standard deviations. This is about the expected overestimation for a bound that covers 99.7% of samples in a normal distribution according to the three-sigma rule [96].

With $T \leq 600$, the data-aging mechanism attempted to adapt to the noise, resulting in lower overestimation but more prediction errors. With $T \geq 600$, no such adaptation happens; the setting performs about the same as the non-adaptive setting. A merging period of $T = 600$ thus avoided overfitting in the fluctuating workload while still reducing the average overestimation by 60–80% in the spiking workload.

In addition to a longer adaptation period, there are two other parameters that can help to reduce the number of prediction errors: the safety margin S and the trigger threshold G. We begin with the safety margin.

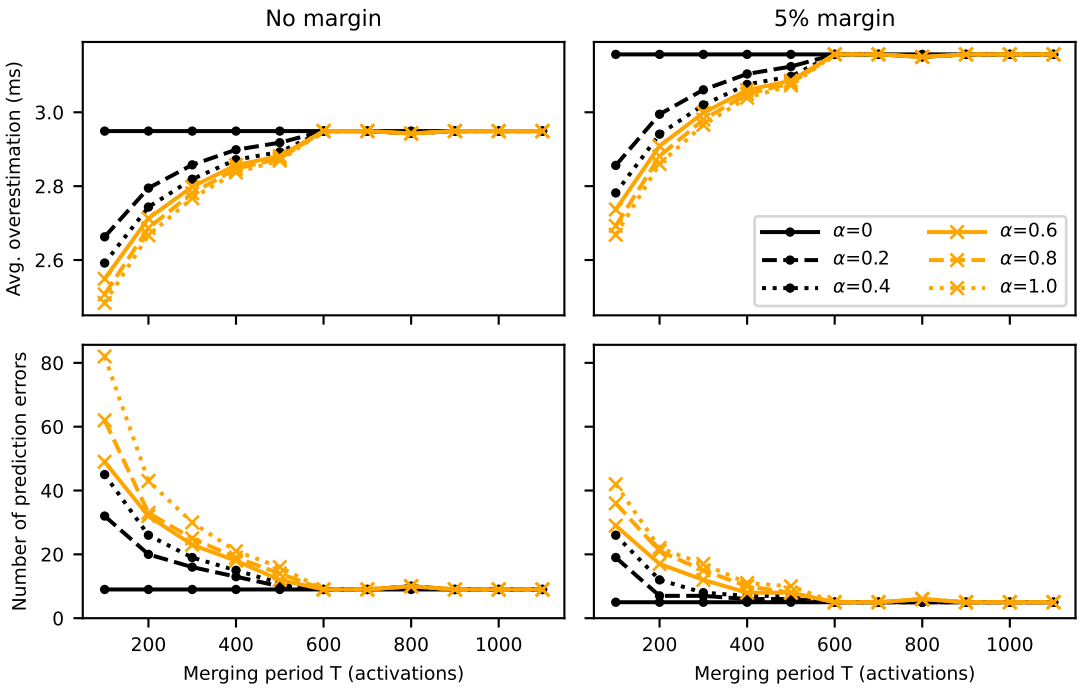
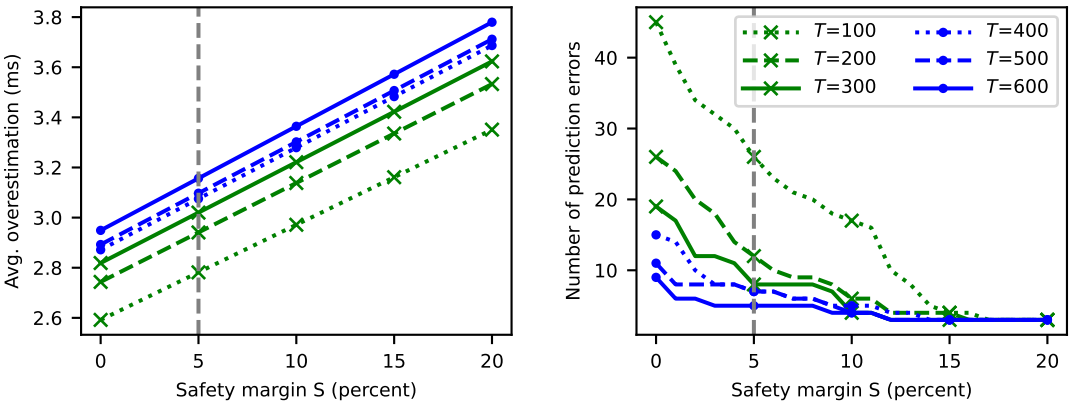


Figure 6.16: Data-aging performance on the fluctuating workload ($G = 90\%$).



(a) Adaptivity.

(b) Predictive power.

Figure 6.17: Effect of safety margins under the fluctuating workload ($G = 90\%$, $\alpha = 0.4$).

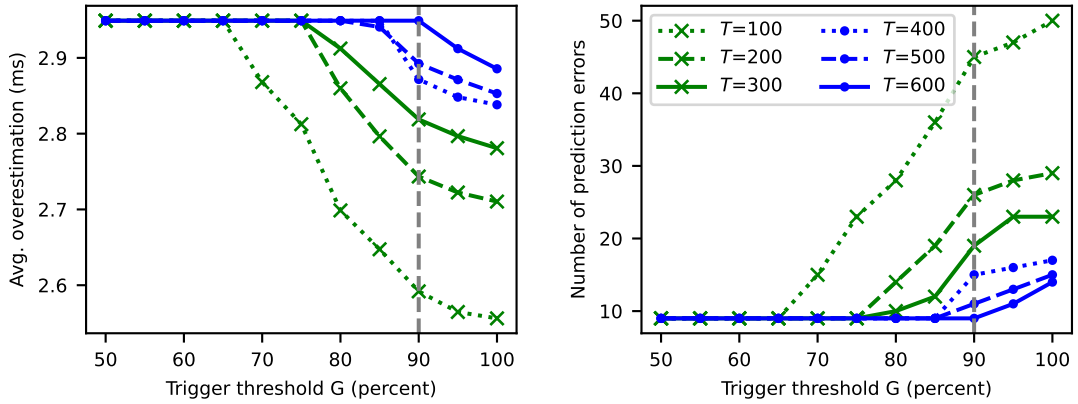
Safety margin. The effects of the safety margin (here: $S = 5\%$) are depicted in the right column of Fig. 6.16. The average overestimation is shifted upwards by about 0.2 ms compared to the $S = 0\%$ setting. The number of prediction errors drops nonlinearly, *i.e.*, more strongly at shorter merging periods than at longer merging periods. The safety margin thus shifts the average overestimation by a fixed amount but superlinearly dampens the mispredictions of more aggressive adaptation policies.

The effect can be observed more closely in Fig. 6.17, which shows the development of the overestimation and the prediction errors under different safety margins and merging periods for $\alpha = 0.4$. The overestimation increases linearly, while the number of mispredictions decreases more quickly for shorter merging periods than for longer merging periods. However, particularly for the longer merging periods, there are visible plateaus where increases in the safety margin do not improve the predictive power (but do increase the average overestimation). For example, $T = 500$ does not reduce its prediction errors at all between 0.5% and 4.5%.

The experiments also show that it can be better to choose a more aggressive adaptation policy with a larger safety margin than a less aggressive adaptation policy with a smaller safety margin. As Fig. 6.16 shows, choosing the parameters $\alpha = 0.2$ and $T = 200$ with a safety margin of $S = 5\%$ yields the same adaptivity and predictive power as the non-adaptive $\alpha = 0$ configuration without a safety margin. The adaptive variant thus performed about equally well on the fluctuating workload but would perform significantly better on the spiking workload.

Trigger threshold. The second parameter that protects against overfitting is the trigger threshold G. To compare it to the safety margin, we compare the predictive power and the adaptivity for a range of merging intervals and trigger thresholds (Fig. 6.18). The gray vertical line marks the setting $G = 90\%$ (the setting used in Fig. 6.16) and corresponds to the $S = 0$ setting in Fig. 6.17.

The results show that the trigger threshold reduced the number of prediction errors significantly without the increase in overestimation that accompanies the safety margin. For example, a trigger



(a) Adaptivity.

(b) Predictive power.

Figure 6.18: Effect of trigger thresholds under the fluctuating workload ($S=0, \alpha=0.4$).

threshold of 75% reduced the prediction error of the $T = 200$ and $T = 300$ configurations to the level of the $T = 600$ configuration, which is equal to the non-adaptive error rate (Fig. 6.16). The trigger threshold thus effectively disabled aging for the fluctuating workload (where it is ineffective), but kept data aging enabled in the spiking workload (where it is highly beneficial).

Conclusion. For the fluctuating workload, the trigger threshold was more effective than the safety margin. Both prevented overfitting in the fluctuating workload while still adapting to the changing demand in the spiking workload, but only the trigger threshold achieved this result without any additional overestimation. For $T=200$, a trigger threshold of 75% reduced prediction errors to the level of the non-adaptive setting without any additional overestimation. To achieve the same result with a safety margin, a margin of 15% would have been needed (Fig. 6.17), which would have increased the average overestimation by 10% (0.3 ms).

6.5.3 Oscillating Workloads

So far, we only considered extreme cases: either any form of adaptation was beneficial without any drawbacks (as in the spiking workload), or it was always harmful (as in the fluctuating workload). In contrast, the oscillating workloads require a more controlled and targeted adaptation. The

execution-time curve should follow the peaks and troughs of the sine wave to avoid undue pessimism but should not overfit to the Gaussian noise.

We first compare the adaptivity and predictive power without a safety margin (Fig. 6.19). The left column shows the rapidly oscillating workload (0.1 Hz) and the right column shows the slowly oscillating workload (0.05 Hz).

As we can see in the bottom row, all settings caused a large number of prediction errors. Even in the non-adaptive case, there were about 50 errors, while adaptive configuration settings (*i.e.*, $T \leq 500$) caused hundreds of prediction errors. These errors stem from the upwards slope of the sine wave. Once the bound had been reduced below the peaks of the sine wave, it underestimated the execution-time demand when the upwards slope arrived. However, since the execution-time demand grew gradually, each prediction error during the slope only slightly increased the execution-time estimate. This led to another prediction error shortly thereafter: the estimate kept lagging behind the processor demand.

The number of mispredictions drops to levels comparable to the non-adaptive setting only at $T = 600$ for the rapidly oscillating workload and beyond 700 in the slowly oscillating workload. However, as shown by the top row, the average overestimation reaches the level of the non-adaptive case as well: choosing $T \geq 600$ is thus equivalent to disabling the data-aging mechanism entirely.

Safety margin. To address the large number of mispredictions we increased the safety margin to 5% (Fig. 6.20). We keep the results of the non-adaptive variant without the safety margin as a baseline, which is depicted as a horizontal dotted line.

The rapidly oscillating workload produced significantly fewer prediction errors with $S = 5\%$ than without a safety margin. For $\alpha \leq 0.4$, prediction errors were below the non-adaptive baseline. Even at the highest merging weight $\alpha = 1$, prediction errors remained below the baseline for $T \geq 250$.

At the same time, for any $T \leq 600$, and particularly for $T \leq 400$, the overestimation is still

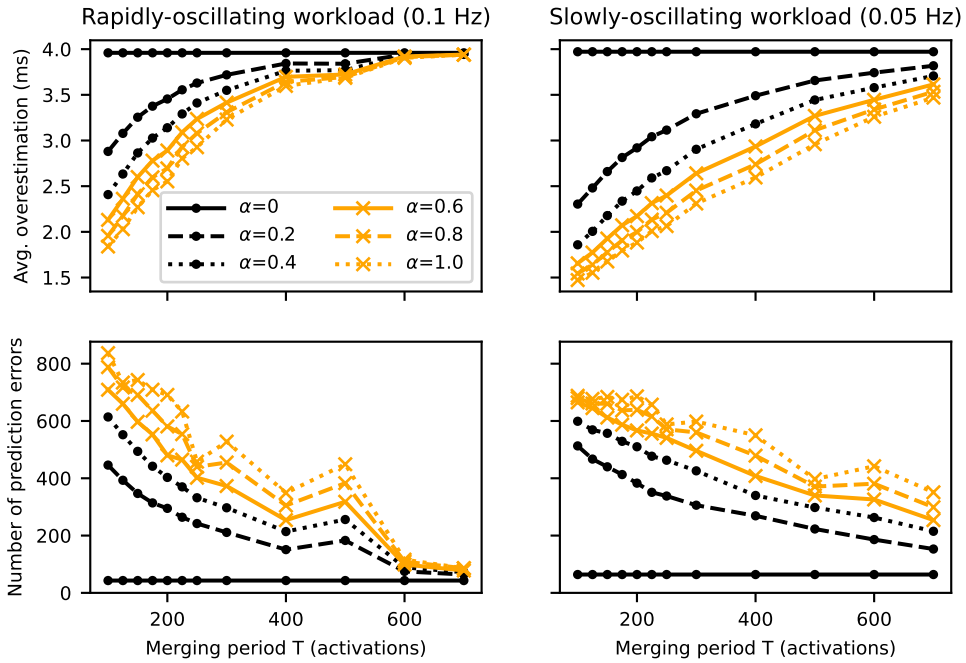


Figure 6.19: Data-aging performance on the oscillating workload with a 0% safety margin.

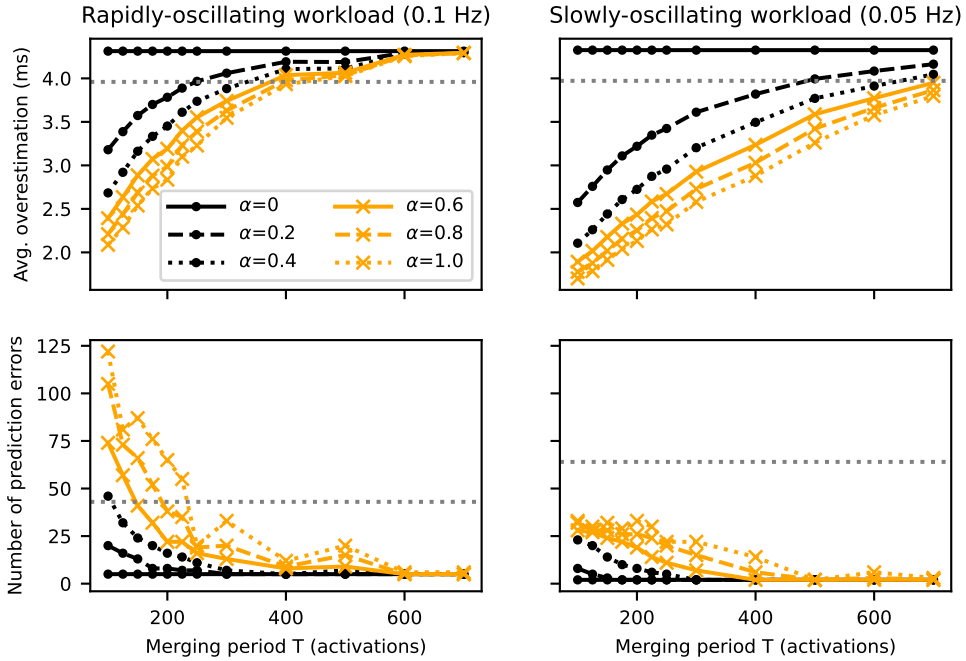
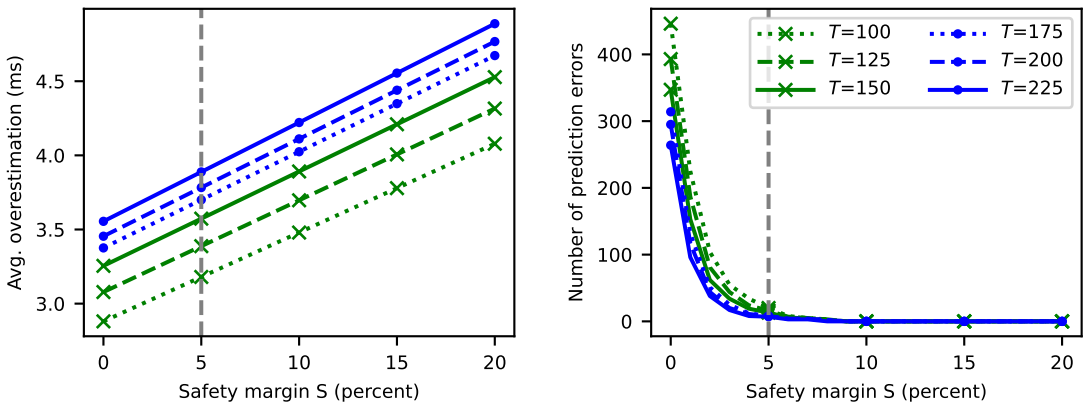


Figure 6.20: Data-aging performance on the oscillating workload with a 5% safety margin.



(a) Adaptivity.

(b) Predictive power.

Figure 6.21: Effect of safety margins under the slowly oscillating workload ($G = 90\%$, $\alpha = 0.2$).

significantly reduced compared to the non-adaptive case, demonstrating that the data-aging process adapted to the sine wave pattern.

Overall, data aging with the safety margin thus improved upon the baseline in this scenario. For $T = 175$ and $\alpha = 0.2$, prediction errors remained below 10 while the average overestimation was reduced by 6% (0.25 ms).

We can conclude that the safety margin allowed ROS-Llama to anticipate demand increases. Although the execution-time bound was still raised in small steps as the demand slowly increased, the bound no longer lagged behind the demand but stayed one step ahead.

The improvement is even clearer in the slowly oscillating workload. Here, choosing $T = 175$ and $\alpha = 0.2$ resulted in zero prediction errors and an average overestimation that is 18% (0.75 ms) below the baseline. Overall, the baseline without safety margins performed worse on both metrics than all observed configurations with $T \leq 600$.

To observe how sensitive this improvement is to the value of the safety margin, we again compare the performance of the slowly oscillating workload under various safety margins (Fig. 6.21). The result shows that the range between 0 and 5% was responsible for the significant reduction in prediction errors, reducing the number of errors from at least 250 errors to the 0–25 errors observed in Fig. 6.20. However, further increases had only limited benefits.

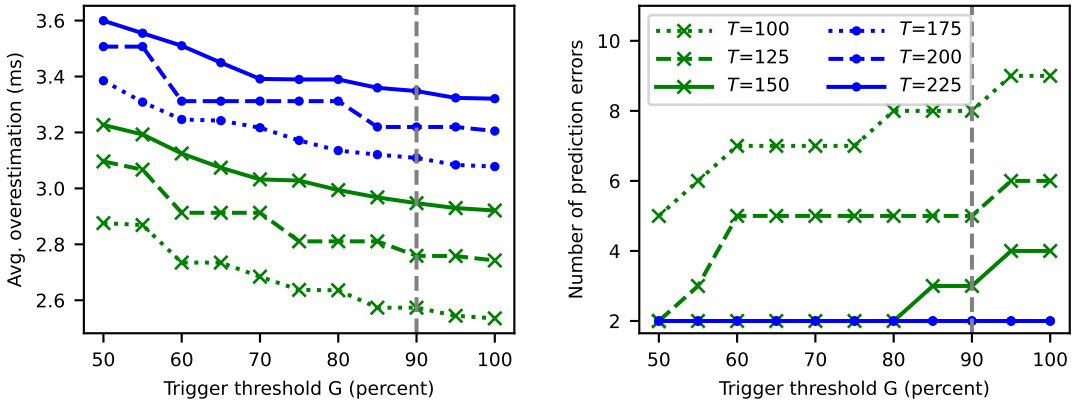
Frequency effects. Comparing the two oscillating workloads, it is striking that the slowly oscillating workload achieved better results along both dimensions, predictive power and adaptivity, than the rapidly oscillating scenario: data aging seems to perform better for slowly-changing demand than for quickly-changing demand.

The reason is that the slowly-changing demand exhibits a larger frequency difference between the long-term processor demand changes and the random noise. To avoid overfitting on the noise, the model extractor needs to use an adaptation period that is large enough that high-frequency noise cancels out, as we observed in the case of the fluctuating workload. However, if the merging interval becomes too large, the low-frequency variation cancels out as well. This turns the data-aging mechanism ineffective. A smaller frequency difference between the noise and the demand change hence leaves a smaller range of possible merging intervals.

For example, recall that the experiments on the fluctuating workload showed that a merging period of $T = 600$ reliably avoided overfitting. However, this merging period was too long to capture the 0.1 Hz-fluctuation of the rapidly oscillating workload and therefore did not improve upon the baseline in that scenario (Fig. 6.19). If the data-aging mechanism has to follow the rapidly oscillating workload, the system integrator has to compromise on robustness against noise. A workload that changes more slowly does not have this issue and allows the system integrator to optimize the merging period for robustness only.

Trigger threshold. In the experiments with the fluctuating workload, the trigger threshold prevented overfitting at a lower cost in overestimation than the safety margin. To evaluate whether the results also apply in the oscillating setting, we again compare the adaptation performance under various trigger thresholds and with $\alpha = 0.2$ (Fig. 6.22). The safety margin was set to 5% in order to adapt to the slowly-increasing demand on the upwards slopes of the sine curve, which cannot be addressed by the trigger threshold.

The results show that reducing the trigger threshold improved predictive power for more aggressive adaptation policies (*i.e.*, shorter merging intervals) but also increased average overesti-



(a) Adaptivity.

(b) Predictive power.

Figure 6.22: Effect of trigger thresholds under the slowly oscillating workload ($S = 5\%$, $\alpha = 0.2$).

mation. In some cases, tuning the trigger threshold realized this trade-off more favorably than other parameters. For example, with a trigger threshold of 80%, the $T = 150$ setting made the same number of prediction errors as settings with longer merging periods while providing lower average overestimation. In other cases, tuning the trigger threshold turns out to be less effective than tuning the merging period. Reducing the threshold below 80%, for example, increased the average overestimation of the $T = 150$ setting without any gain in prediction precision relative to higher merging periods.

All in all, the evaluation shows that there is no clear guideline in which cases the trigger threshold improves results. Although we can identify individual cases where one configuration is superior to the other on both adaptivity and predictive power, there is no obvious general rule.

Notably, there are various ranges where a lower trigger threshold did not improve prediction accuracy at all but increases average overestimation (*e.g.*, $T = 125$ and $G \in [60\%, 90\%]$, or $T = 150$ and $G \in [50\%, 80\%]$). Future work that identifies when such plateaus occur might yield a more general rule for selecting a suitable trigger threshold.

6.5.4 Summary

The experiments show that ROS-Llama's data-aging method significantly improves upon the non-adaptive execution-time curve measurement. We considered four synthetic workloads with different properties, which allowed us to observe the effect of the data aging mechanism under controlled and repeatable circumstances, without distortions from sensor noise or varying system behavior. For the spiking and the two oscillating workloads, we found configuration settings that improve upon the non-adaptive baseline on predictive power, adaptivity, or both. For the fluctuating workload, which cannot possibly benefit from data aging, we found configurations that perform no worse than the non-adaptive baseline.

Our initial investigation yields some suggestions on how the parameters should be chosen to address certain workload properties. A workload that is prone to gradual increases should use a safety margin to reduce prediction errors during the upwards slope. Workloads with highly varying and noisy execution times should use long merging periods or low trigger thresholds. Finally, the merging weight α can be used to fine-tune the aggressiveness of the adaptation mechanism.

Overall, data aging is an effective approach to trade predictive power against adaptivity and vice versa. The four parameters allow system integrators to adapt to various workloads, in particular spiking, fluctuating, and oscillating workloads. However, the interaction between the parameters and the workload is complex, and it is not obvious how a suitable set of parameters could be found automatically. Automatically finding a suitable configuration for data aging therefore remains a promising area of future work.

7 Conclusion

In this dissertation, we investigated the timing behavior of the ROS framework and proposed solutions to enforce timing correctness for ROS applications. To this end, we first investigated the timing behavior of the ROS executor and proposed a real-time model that takes this behavior into account (Chapter 3). We then developed a response-time analysis for this timing model (Chapter 4). However, applying response-time analysis to ROS systems is challenging in practice due to the requirements of the robotics domain and the distributed development style of the ROS ecosystem. In Chapter 5 we addressed these challenges with ROS-Llama, an automatic latency manager for ROS that automatically configures real-time scheduling parameters for a ROS system based on only a simple, declarative specification of latency goals.

In this final chapter, we first summarize the main contributions and results of the thesis and then conclude with a discussion of future work.

7.1 Summary of Results

This dissertation provides three main contributions. First, it characterizes the timing behavior of the ROS framework and defines a system model that incorporates these properties. Second, it defines a response-time analysis to bound the end-to-end response time of processing chains within this model. Third, it identifies properties of ROS and requirements of the robotics domain that make it challenging to apply static response-time analysis in practice. To overcome these challenges, the dissertation then proposes ROS-Llama, an automatic latency manager for ROS systems. In the following, we briefly recapitulate the main insights and results.

7.1.1 Modeling the ROS framework

ROS offers a uniform and high-level interface to ROS developers. Unfortunately, this abstraction obscures the timing behavior of the system. To address this issue, Chapter 3 contains a detailed description of the ROS framework's timing behavior. We identify the unusual scheduling algorithm implemented by the standard ROS executor. To model its effect on the callback execution order, we identify six basic properties that describe the executor's behavior.

The analysis of the model reveals that the executor algorithm has changed in subtle ways over time. More specifically, timers previously enjoyed a privileged status that has been silently removed in the "*Eloquent Elusor*" version. Given the continuous developments and improvements in the ROS infrastructure, such changes are easy to miss. We therefore develop a model validation tool that automatically verifies the derived scheduler properties by sending a specific sequence of messages and observing the resulting callback execution order. Using this validation tool, we confirm the correctness of our characterization for the "*Dashing Diademata*" and "*Foxy Fitzroy*" versions.

Based on these properties, we construct a real-time system model for ROS applications (Section 3.2). The model represents a ROS system as a directed graph of callbacks. Each callback is triggered periodically or by an external event and may trigger other callbacks during its runtime. Each callback is further assigned an executor, which it may share with other callbacks. Non-executor threads are integrated as virtual callbacks called *event sources*. As the evaluation shows, the model is expressive enough to represent real-world ROS packages yet simple enough to allow for efficient response-time analysis.

To cope with the large execution-time variance prevalent in ROS systems, a callback's execution-time demand is described as an *execution-time curve* instead of the more traditional worst-case execution time (WCET). Although execution-time curves increase the complexity of the model and particularly the analysis, the evaluation shows that the results are well worth the cost. In a realistic ROS callback graph, the more sophisticated execution-time model improves response times by a factor of up to 60.

The evaluation confirms that the model is expressive and accurate enough to model real-world ROS systems (Section 6.1). However, the evaluation reveals that the implementation of the communication layer may not behave as a transparent and reliable transport medium under load (Section 6.3.3). Specifically, we observed message propagation delays even if the sender and receiver of the message are the same thread and encountered excessive latency spikes in overload situations. Timing-sensitive ROS systems thus need to ensure that the underlying communication layer fulfills the basic middleware requirements and provides timely message delivery even under load.

7.1.2 Response-time Analysis

To predict the timing behavior of ROS applications, Chapter 4 develops a worst-case response time analysis for the ROS timing model. The analysis improves upon established techniques in three main areas: support for execution-time curves, a round-robin analysis, and a busy-window analysis with improved support for intra-executor chains.

Support for execution-time curves. The analysis explicitly supports execution-time curves instead of scalar WCETs. As part of this support, Section 4.2.3 proves how to exploit the non-preemptive policy of the ROS executor to reduce the amount of interference to consider. Compared to WCET-based analyses, this is particularly challenging since execution-time curves blur the line between the cost of self-interfering instances and the cost of the instance under analysis itself.

The round-robin approach. Section 4.2 describes a novel analysis to exploit the round-robin property of the ROS executor. Compared to busy-window-based approaches, the round-robin bound limits the possible interference by bursty callbacks much more precisely. In the evaluation, the round-robin bound reduces the response-time bound by a factor of up to 3 compared to the competing, busy-window-based analysis approach proposed in this dissertation.

The busy-window approach. Finally, Section 4.3 describes a competing analysis approach based on the busy-window principle. The main contribution over prior work is an improved handling of intra-executor chains, which exploits that the callback instances contained in an intra-executor chain instance share a common busy window. This insight allows for a more precise activation curve derivation and reduced chain response times.

The evaluation in Section 6.2 shows that neither the round-robin approach nor the busy-window approach dominates the other. In practice, both approaches should be combined to exploit the strengths and compensate for the weaknesses of both approaches.

7.1.3 Automatic Latency Management

Applying a response-time analysis to ROS systems is difficult in practice. To bridge the gap between the two—and make formal response-time analysis not only possible but also *practical* in ROS systems—Chapter 5 first identifies the main hurdles and requirements that make response-time analysis difficult to apply. Based on the resulting nine requirements (Section 5.1), we conclude that a static provisioning and verification approach is not feasible for common ROS systems and that a more dynamic and adaptive approach is needed.

To address this challenge, we then propose ROS-Llama, the ROS Live Latency Manager. The main contribution of ROS-Llama is that it brings real-time scheduling to ROS in a package that is easy to use. The user provides only a simple and declarative latency goal specification stating the latency goals and desired degradation order. Crucially, the specification presupposes neither real-time systems expertise nor a detailed understanding of the system internals.

The implementation of ROS-Llama consists of two main components: a *model extractor* that automatically derives a timing model of the running application and a *budget manager* that uses the derived timing model to provision the ROS threads in accordance with the latency specification.

The Model Extractor. The implementation of the model extractor and the technical challenges involved are described in Section 5.3. The extractor is informed of important events, such as the completion of a callback, by a set of tracepoints in the ROS core libraries. This way, the extractor can extract a model without requiring any modification to the system under analysis. The extractor then constructs a timing model of the system based on the stream of trace events. While regular callbacks can be extracted precisely, the extraction of event sources, which interact less with the ROS library, requires a few heuristics.

A challenging aspect of model extraction is that the extractor needs to derive a static model for a dynamic system. To follow and adapt to dynamic changes, Section 5.3.5 describes the heuristics used by the model extractor to identify phase changes and other reasons to discard previously collected information.

However, such automatic heuristics are usually not able to follow more implicit changes in the execution-time demand, for example those caused by changes in the environment. A more comprehensive solution is provided by the *data aging* mechanism, which enables ROS-Llama to slowly phase out existing execution-time measurements if the system behavior demonstrates significant overestimation.

The evaluation on a set of synthetic workloads (Section 6.5) shows that the proposed data aging mechanism, properly configured, can significantly reduce overestimation with only minor effects on the misprediction rate. The data aging parameters allow the user to fine-tune the mechanism for the intended workload and target different points on the tradeoff between accuracy and adaptability.

Due to the complexity of the problem and the situation-dependent tradeoff between bound accuracy and adaptation speed, the data aging mechanism requires manual configuration by the user, though. Tuning the data aging mechanism automatically or based on only a high-level declarative specification remains future work.

The Budget Manager. The ROS-Llama budget manager uses the extracted model to automatically find a set of real-time scheduling parameters for the running system. As described in Section 5.4, the budget manager uses SCHED_DEADLINE in a partitioned configuration to provision appropriate budgets for the ROS executors. Due to limitations in the Linux scheduler, critical system threads are isolated on a separate core, which they share with the ROS-Llama infrastructure.

The appropriate budget for each executor is then derived using response-time analysis. For each chain, the budget manager first identifies a minimal schedulable set of budgets without considering the chain latency constraints. It then incrementally increases the budgets until the chain is guaranteed to fulfill its latency goal. If timeliness cannot be guaranteed this way, the budget manager degrades the chain and all chains below it in the degradation order to ensure the continued timeliness of the earlier chains.

The evaluation shows that ROS-Llama successfully controls the end-to-end latency of critical processing chains and ensures controlled degradation under load. In an experiment with a mobile robot using three real-world ROS packages, ROS-Llama achieves fewer latency goal violations than the default CFS system and a criticality-monotonic SCHED_RR assignment. Although ROS-Llama incurs significant costs, the results demonstrate that the benefits outweigh the costs.

7.2 Future Work

The evaluation has demonstrated that the ROS timing model and ROS-Llama are effective ways to predict and control the worst-case latency of real-world ROS systems. In the following, we suggest six promising directions to expand further upon these initial results.

The first two suggestions address limitations in the platform layers beneath the ROS framework. We first reiterate the need to identify a well-behaved middleware (Section 7.2.1). We then point out four limitations in the Linux kernel that were not obvious to us initially and that hinder ROS-Llama's efforts to enforce timing isolation on ROS workloads (Section 7.2.2).

The final four suggestions address improvements in the timing model and ROS-Llama. We first list a few promising extensions to the timing model that would further expand its scope and expressiveness (Section 7.2.3). We then suggest potential improvements to ROS-Llama's budget manager (Section 7.2.4) and how a stochastic response-time analysis might reduce the inherent pessimism in the current response-time analysis approach (Section 7.2.5). Finally, we discuss the next steps towards a more automatic data aging mechanism, which would make ROS-Llama more adaptive and allow it to exploit changes in the managed workload more aggressively Section 7.2.6.

7.2.1 Towards a Well-Behaved Middleware

The evaluation (Section 6.3) demonstrated that the employed DDS implementation does not fulfill the basic middleware requirements identified in Section 2.1.2. Although ROS-Llama managed to prevent the extreme latency spikes observed in the SCHED_RR baseline by proactively degrading the *tracker* chain, the fact that those latency spikes *can* happen threatens to undermine timing isolation between the executors.

The logical step to address this issue is to replace the middleware with another implementation that *does* fulfill the basic middleware requirements. The DDS implementation we used in the evaluation, Cyclone DDS, is under active development and recently integrated the *iceoryx* [46] middleware for in-memory communication [45]. It is possible that the middleware limitations observed in the evaluation are going to be resolved with this change. It is also worth investigating other DDS vendors and the various non-DDS middlewares supported by ROS.

Still, even with a middleware that fulfills the basic middleware requirements, the timing analysis treats the middleware as a black box. It therefore cannot account for the numerous Quality-of-Service (QoS) options offered by DDS; the documentation of one implementation [100] lists alone 53 parameters affecting, among other things, the number of threads created, the scheduling priority of several DDS support threads, or the message transmission order.

Modeling the underlying communication infrastructure in more detail would allow ROS-Llama to make more informed choices towards scheduling the DDS communication threads,

and might even allow ROS-Llama to automatically select appropriate QoS options depending on the latency requirement. A good starting point along those lines is prior work on analyzing implementation-independent QoS options [57, 93].

It is worth noting that such a model would most likely not generalize across DDS implementations. Although many QoS options are standardized, other aspects like the scheduling policy for DDS threads are intentionally left aside by the DDS standard as implementation-defined.

7.2.2 Addressing Limitations in Linux

ROS-Llama benefits from Linux's real-time capabilities to a great extent. The timing isolation afforded by SCHED_DEADLINE is central to ROS-Llama's approach. Nonetheless, certain limitations in the Linux kernel posed surprising challenges.

High-latency I/O. In our experiments, we found that laser-scanner and odometry data would sometimes arrive at the Turtlebot driver threads only after excessive delays. It turned out that data arriving on USB serial ports traverses the TTY layer, which involves CFS-scheduled kernel threads (even in a PREEMPT_RT kernel) that are easily starved by real-time processes. Although we were ultimately able to sidestep this problem by forcing Linux's "unbound *kworker* threads" onto the system processor, it serves as a reminder that real-time I/O remains a frequently overlooked and understudied problem.

Scheduler inversion. SCHED_DEADLINE threads *always* take priority over fixed-priority threads. This simplifies the analysis but also causes many practical issues. Although the assigned reservation bandwidths limit this delay somewhat, it can still be substantial (*i.e.*, the maximum EDF busy-window length). This design is particularly unfortunate since many system-critical kernel threads (*e.g.*, disk drivers) are scheduled with SCHED_FIFO or SCHED_RR priorities. Assigning "too much" bandwidth to reservations can thus starve critical kernel threads and actually lead to kernel panics. A principled solution might be to introduce "scheduling-class

reservations" that explicitly reserve processor time for SCHED_FIFO, SCHED_RR, and CFS in the SCHED_DEADLINE schedule.

Soft reservations needed. Mainline Linux presently supports only hard reservations, which unconditionally cut off a thread that exhausts its reservation's budget from processor service until the next replenishment time (cf. Section 2.2.2). This rate-limiting behavior can be highly problematic: *underestimating* the required budget even by a minuscule amount results in a massive latency increase, since once a thread's under-dimensioned budget runs out, it must wait *even if it could complete if it were a* CFS *thread*. ROS-Llama can therefore use reservations only if it is certain that the budget suffices for the latency goal. In contrast, soft reservations would allow under-provisioned threads to receive at least *some* guaranteed bandwidth and then progress on a best-effort basis, which would allow ROS-Llama to partially provision degraded chains.

Threads vs. reservations. SCHED_DEADLINE ties reservation parameters to individual threads. It is thus not possible to share a budget among multiple threads, making it exceedingly inefficient to apply reservations to multi-threaded applications that distribute work dynamically among threads (*e.g.*, virtually all DDS middlewares). Popular libraries for asynchronous programming like *boost::asio* or the C++ *std::async* API are also impossible to provision. First-class reservations supporting multiple client threads would be a relief.

These four limitations should not just be considered in isolation. One example of how these limitations interact and compound is the scheduling policy for the DDS middleware threads. Recall that ROS-Llama assigns these threads to a separate core at a fixed priority (Section 5.4.1). This is far from ideal: scheduling them together with their executor thread would strengthen timing isolation. However, the lack of first-class reservations prevents ROS-Llama from allocating a shared budget for the executor and its support threads. ROS-Llama would therefore need to provision each communication thread individually. However, without a comprehensive model of the communication thread's behavior, ROS-Llama has no solid basis for deriving a budget.

If soft reservations were available, ROS-Llama could simply estimate a budget for the communication threads and correct the estimate over time if needed. However, hard reservations harshly penalize this kind of trial and error. Under a hard reservation policy, an undersupplied support thread will be starved of supply until its budget is replenished even if a CFS thread could continue, leading to unpredictable latency spikes. As a result, ROS-Llama is forced to schedule communication support threads with SCHED_RR.

However, if communication support threads are scheduled with SCHED_RR, then they cannot run on the same core as the ROS executors. Due to the scheduler inversion, a communication thread would always have a lower priority than any ROS executor. This would not only lead to massive latency spikes in the communication layer but would also counteract the controlled degradation mechanism: an executor serving a chain early in the degradation order would take priority over all communication threads, delaying chains that are later in the degradation order. Overall, scheduling the communication threads under SCHED_RR on a separate system core remains the only option today.

To sum up, the identified scheduler limitations are not independent but interact in unfortunate ways. Addressing just one of the identified limitations might already open up a way to circumvent the remaining limitations in practice.

7.2.3 Model Extensions

The timing model in Chapter 3 is expressive enough to handle real-world ROS components like the ROS navigation stack (cf. Section 6.1). Nonetheless, given the inevitable complexities associated with a mature, flexible, and widely used framework, we had to elide certain aspects of ROS in this dissertation. In the following, we discuss these aspects and highlight promising directions for future extensions.

Other Executors. This dissertation considers only the single-threaded ROS executor. ROS also provides a multi-threaded variant of that executor and additionally allows the definition of

arbitrary special-purpose executors. It would be useful to support special-purpose schedulers tailored to specific robot or real-time needs, for example the *rclc* executor from the microROS project [113], the PiCAS executor [30], or the sense-plan-act cyclic executive from the Fawkes framework [82]. Refer to Section 2.4 for a discussion of these executors.

Message Buffer limits. ROS offers quality-of-service (QoS) options[1] to fine-tune the performance and reliability of individual topics. One of these options, called *history*, concerns the size of the internal message buffers. Setting the history option to "keep last" with a "depth" of n allows the middleware to discard all but the latest n messages for this topic.

Such message buffer limits are useful for applications where data loss is acceptable because later messages supersede earlier messages. Typical examples are laser scanners or position estimates. In such cases, discarding stale messages (and skipping the computation triggered by those messages) can reduce the worst-case load without impacting the system's functionality.

Such communication channels with limited buffers have already been studied in prior work [81]; particularly the special case $n = 1$ has received significant attention and is also known as *register semantics* [40, 73]. Integrating these approaches with our timing model in order to take advantage of such buffer limits remains future work.

Complex Activation Semantics. In addition to regular callbacks, which are unconditionally activated upon publication of a message, ROS also provides advanced activation semantics in the form of *message filters*. Of particular interest to us are the TF-related message filters, which are, for example, used in the navigation stack and required us to add the separate *localization* chain to the evaluation workload (cf. Section 6.1.2). Conceptually, filters *select* and *combine* multiple incoming data items (from separate messages) into a single message for *joint* downstream processing based on complex rules. In terms of expressiveness, they go far beyond classic "and" activation semantics and may also depend on the *contents* of to-be-combined messages.

[1] Each of these options usually corresponds directly to a DDS option of the same name but is defined as part of the ROS API. In particular, it also applies to non-DDS middlewares.

In the navigation stack, for instance, the localization component uses message filters to ensure that the laser scanner callback is only triggered once an odometry measurement is available that is no older than the latest available laser scan sample. Such complex activation semantics cannot be represented with current modeling approaches.

In the evaluation, we avoided modeling the message filter semantics by **(a)** ensuring that no processing chain spans across the message filter and **(b)** manually assigning latency goals to the surrounding chains such that the conceptual data flow (from the laser scanner to self-localization estimate and from the odometry to the self-localization estimate, respectively) completes in time even considering worst-case delay in the message filter. However, this approach needs a level of manual intervention that is at odds with the *ease-of-adoption* requirement (Req. V).

A fully automatic way to handle message filters and similar utility libraries would most likely have to extend the timing model to incorporate the complex activation semantics. A promising path towards this goal would be to integrate the "and" semantics CPA [61] and extend it with additional constraints on the data age of incoming messages.

Self-Suspending Callbacks. The model assumes that callbacks do not self-suspend, i.e., that callbacks run to completion once they start running. This is a reasonable assumption for most callbacks: a self-suspending callback would block the entire executor, preventing the executor from running any other callback in the meantime. However, this reasoning does not apply to event sources or other callbacks that have exclusive use of their executor. Suspending the executor might also be acceptable if the developer can ensure that the suspension is short. Extending the model to account for self-suspensions would therefore further increase the model's applicability.

A particular form of self-suspension that deserves special consideration are *suspending service calls*. The ROS core libraries come with a convenient way to invoke service calls in a blocking fashion (instead of providing a client callback) using the *spin_until_future_complete* function. The model extractor can easily monitor the use of this API; it thus provides an ideal starting point to explore the implications of callback suspensions for ROS-Llama.

7.2.4 Improving the Budget Management

As of now, ROS-Llama's budget manager provisions all tasks with a fixed period of 5 ms (Section 5.4.1). However, prior work [26, 69] has shown that matching the period to the expected workload of the task can significantly improve both average and worst-case response time, as discussed in Section 2.4. Although the experiments discussed in Section 6.4 demonstrated that significantly shorter periods introduce unacceptable overheads, it stands to reason that well-chosen *longer* periods reduce scheduling overheads and thereby improve system performance.

Another potential area of improvement is the budget assignment. As of now, the budget manager uses an iterative heuristic to find a suitable set of budgets (cf. Section 5.4.2). The heuristic is needed to cope with the complex and potentially circular budget dependencies among executors. Unfortunately, each iteration of the heuristic requires a response-time analysis. The heuristic therefore needs to compromise on the quality of the budget assignment to reduce the number of iteration steps required. Examples for such compromises include relying on the budget-shortage delay to find executors in need of more budget, the greedy budget assignment strategy, and the choice to consider possible bandwidth assignments only in fixed-size steps. Although the heuristic works well enough for our purposes, there is likely much room for improvement.

Even if the intra-executor budget dependencies prove an insurmountable obstacle to an optimal solution, it is unlikely that *all* executors in the system are involved in a dependency loop. It might be worthwhile to try optimal but less general budget algorithms first and fall back to the iterative approach if the dependency loop proves intractable.

7.2.5 Below-Worst-Case Provisioning through Probabilistic Analysis

ROS-Llama assigns its budgets based on the response-time analysis presented in Chapter 4. As discussed in Chapter 5, this has the advantage of predicting latency issues before they happen and allows ROS-Llama to be *proactive* instead of reactive.

However, it also makes ROS-Llama rather pessimistic; the controlled degradation process is initiated if there is any possibility whatsoever for a chain to miss its latency goal, no matter how

unlikely. Even worse, a pessimistic response-time bound might trigger the degradation process even though there is no way for any chain to actually miss its latency goal.

Given that ROS-Llama follows a dynamic, introspection-based approach anyway, and given the inherent uncertainty in ROS systems and their dynamic environments, it is unnecessary to provision the system under worst-case assumptions. We would gladly trade some analysis certainty for much tighter response-time bounds. For example, consider a hypothetical analysis that replaces the execution-time curve with a traced "probabilistic worst-case execution-time curve" $pET(n)$, in analogy to the probabilistic WCET concept at the center of much recent attention [3, 34, 35, 53, 103, 110, 111].

Unfortunately, no such analysis exists so far, and existing stochastic analyses usually focus on much simpler and more controlled systems than ROS. However, given the inherent uncertainty in ROS systems and their dynamic environments, we believe there to be much promise in this direction.

7.2.6 Automatic Data Aging

As the evaluation showed, the data aging mechanism is powerful enough to adapt to a wide range of (synthetic) workloads (Section 6.5). However, it depends on multiple user-provided configuration parameters that demand significant expertise from the ROS-Llama user and need to be fine-tuned to the workload in question. This clearly does not conform to the *ease-of-adoption* requirement (Req. V).

To make data aging as easy to use as the remaining ROS-Llama features, ROS-Llama should configure the mechanism on its own, based on just a declarative specification of the expected demand variation pattern or even solely based on measurements. However, a better understanding of the effects of data aging under real-world conditions is needed to select suitable parameters. If the preliminary results hold up in more realistic settings, future work might be able to identify the right parameters based on a description of the expected workload pattern or even based on the observed pattern at runtime.

7.3 Closing Remarks

Throughout this work it is apparent that response-time analysis on manually specified timing models is not a practical method for enforcing real-time constraints in ROS 2-based robots. Key challenges include the complexity of ROS and ROS applications as well as the system's dynamic behavior, which defies static modeling.

ROS-Llama addresses these problems by automatically extracting a timing model of the running application. This allows modeling only those parts of the application that are actually used, ignoring components that are not active in the given deployment. It also allows ROS-Llama to adjust to dynamic changes by continuously updating the extracted model.

We have demonstrated that this approach can simplify the implementation of real-time requirements. With just a simple and declarative goal specification, ROS-Llama managed to automatically provision our evaluation robot's navigation stack and driver software such that the timing goals were fulfilled even during high background loads. When the tracker workload overwhelmed the system, ROS-Llama automatically degraded the less critical tracer chain in favor of the more critical navigation chains, ensuring controlled degradation in overload conditions. Although the automation imposes noticeable processor overheads, it stands to reason that additional compute capacity is cheaper than the developer time that would be needed to fulfill real-time requirements using traditional approaches.

However, implementing automatic real-time latency management imposes additional constraints on the underlying implementation. First, our work uncovered that various timing aspects of Linux (such as I/O paths) and popular middlewares (such as DDS) are still poorly understood. Documenting and formally describing the timing behavior of these foundational layers is an essential part of supporting automatic latency management at the higher layers.

Second, ROS-Llama demonstrates the importance of having real-time primitives that gracefully deal with uncertainty. Primitives such as hard CBS, which introduces significant performance cliffs if budgets are even slightly too small, force an automatic latency manager to be extremely conservative in its predictions. In contrast, more forgiving primitives such as soft CBS allow

the latency manager to safely choose tighter budgets and require less overprovisioning in the deployment.

Overall, we see great potential in automatic latency management. Creating and maintaining up-to-date timing models for complex software systems comes at great expense and is currently not feasible for all but the most safety-conscious development projects. By eliminating this requirement and making real-time theory *easy* to use, automatic latency management has the potential to vastly expand the applicability of real-time systems theory, particularly in non-safety-critical systems. We hope that this dissertation provides motivation and a solid base for future developments in this area.

A. List of Tracepoints

Locations of the tracepoints in `rcl` (version 0.7.5) and `rclcpp` (version 0.7.9).

#	Tracepoint	Location
1	register-callback	src/rcl/publisher.c:185
		include/rclcpp/client.hpp:167
		include/rclcpp/service.hpp:156,174,194
		include/rclcpp/subscription.hpp:109
		include/rclcpp/timer.hpp:135
2	start-callback	include/rclcpp/any_service_callback.hpp:88
		include/rclcpp/any_subscription_callback.hpp:157,181,200
		include/rclcpp/client.hpp:228
		include/rclcpp/timer.hpp:154
3	end-callback	include/rclcpp/any_service_callback.hpp:96
		include/rclcpp/any_subscription_callback.hpp:176,195,216
		include/rclcpp/client.hpp:230
		include/rclcpp/timer.hpp:155
4	executor-spin	src/rclcpp/executors/single_threaded_executor.cpp:31
5	publish	src/rcl/publisher.c:255
		src/rcl/publisher.c:290
6	send-request	src/rcl/client.c:280
7	rate::sleep	include/rclcpp/rate.hpp:59
8	rate::wakeup	include/rclcpp/rate.hpp:81,85
9	rate::stop	include/rclcpp/rate.hpp:105
10	limited-spin	src/rclcpp/executor.cpp:219,249
11	spin-until-future-complete	include/rclcpp/executor.hpp:232

Bibliography

[1] Linux kernel documentation: CFS scheduler design. `https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html`, 2021.

[2] Linux kernel documentation: Real-time group scheduling. `https://www.kernel.org/doc/html/latest/scheduler/sched-rt-group.html`, 2021.

[3] Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quinones, and Francisco J. Cazorla. On the Comparison of Deterministic and Probabilistic WCET Estimation Techniques. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 266–275, 2014.

[4] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 4–13, 1998.

[5] Luca Abeni and Giorgio Buttazzo. Adaptive Bandwidth Reservation for Multimedia Computing. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 70–77, 1999.

[6] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a Reservation-Based Feedback Scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 71–80, 2002.

[7] Luca Abeni, Giuseppe Lipari, and Juri Lelli. Constant Bandwidth Server Revisited. *ACM SIGBED Review*, 11:19–24, 2015.

[8] Abdullah A. Arafat, Sudharsan Vaidhun, Kurt M. Wilson, Jinghao Sun, and Zhishan Guo. Response Time Analysis for Dynamic Priority Scheduling in ROS 2. In *Proceedings of the 59th Design Automation Conference (DAC)*, 2022.

[9] Sanjoy Baruah. A General Model for Recurring Real-Time Tasks. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 114–122, 1998.

[10] Giuseppe Beccari, Stefano Caselli, and Francesco Zanichelli. A Technique for Adaptive Scheduling of Soft Real-Time Tasks. *Real-Time Systems*, 30:187–215, 2005.

[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)*, pages 29–37, 2014.

[12] Alessandro Biondi, Giorgio C. Buttazzo, and Marko Bertogna. Schedulability Analysis of Hierarchical Real-Time Systems under Shared Resources. *IEEE Transactions on Computers*, 65:1593–1605, 2015.

[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 *Proceedings of the 42nd Real-time Systems Symposium (RTSS)*, pages 41–53, 2021.

[14] 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)*, pages 264–277, 2021.

[15] Aaron Block, Björn B. Brandenburg, James H. Anderson, and Stephen Quint. An Adaptive

Framework for Multiprocessor Real-Time Systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23–33, 2008.

[16] Sergey Bozhko and Björn B. Brandenburg. Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle. In *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS)*, volume 6, pages 22:1–22:24, 2020.

[17] Björn B. Brandenburg and James H. Anderson. Feather-Trace: A light-weight event tracing toolkit. *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 19–28, 2007.

[18] Björn B. Brandenburg and Andrea Bastoni. The Case for Migratory Priority Inheritance in Linux: Bounded Priority Inversions on Multiprocessors. *Proceedings of the 14th Real-Time Linux Workshop (RTLWS)*, 2012.

[19] Björn B. Brandenburg and Mahircan Gül. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS)*, pages 99–110, 2016.

[20] Roberto Brega, Nicola Tomatis, and Kai O. Arras. The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 1422–1427, 2000.

[21] Reinder J. Bril, Johan J. Lukkien, and Wim F. J. Verhaegh. Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 269–279, 2007.

[22] Herman Bruyninckx. Open Robot Control Software: The OROCOS project. In *Proceedings*

of the IEEE International Conference on Robotics and Automation (ICRA), volume 3, pages 2523–2528, 2001.

[23] Giorgio Buttazzo. HARTIK: A Real-Time Kernel for Robotics Applications. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS)*, pages 201–205, 1993.

[24] Giorgio Buttazzo. Real-time Issues in Advanced Robotics Applications. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems (EMWRTS)*, pages 133–138, 1996.

[25] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Third edition, 2011.

[26] Giorgio Buttazzo and Enrico Bini. Optimal Dimensioning of a Constant Bandwidth Server. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 169–177, 2006.

[27] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B. Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 6:1–6:23, 2019.

[28] Anton Cervin and Johan Eker. Feedback Scheduling of Control Tasks. In *Proceedings of the 39th IEEE Conference on Decision and Control (CDC)*, volume 5, pages 4871–4876, 2000.

[29] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback–Feedforward Scheduling of Control Tasks. *Real-Time Systems*, 23:25–53, 2002.

[30] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2. In *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 251–263, 2021.

[31] Jose Luis Blanco Claraco. *Development of Scientific Applications with the Mobile Robot Programming Toolkit: The MRPT Reference Book*. 2010.

[32] ROBOTIS Co. Turtlebot3 e-Manual. `https://emanual.robotis.com/docs/en/platform/turtlebot3/features`.

[33] Intel Corporation. ROS 2 Object Analytics. `https://github.com/intel/ros2_object_analytics`.

[34] Robert I. Davis and Liliana Cucu-Grosjean. A Survey of Probabilistic Schedulability Analysis Techniques for Real-Time Systems. *LITES: Leibniz Transactions on Embedded Systems*, 6:04:01–04:53, 2019.

[35] Robert I. Davis and Liliana Cucu-Grosjean. A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems. *LITES: Leibniz Transactions on Embedded Systems*, 6:03:01–03:60, 2019.

[36] Mathieu Desnoyers and Michel R. Dagenais. The LTTng Tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Ottawa Linux Symposium*, volume 1, pages 209–224, 2006.

[37] Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional Performance Analysis in Python with pyCPA. *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS)*, pages 27–32, 2012.

[38] Eric Eide, Tim Stack, John Regehr, and Jay Lepreau. Dynamic CPU Management for Real-Time, Middleware-Based Systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 286–295, 2004.

[39] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. An EDF scheduling class for the Linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop (RTLWS)*, pages 1–8, 2009.

[40] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. *Proceedings of the 1st Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2008.

[41] Daniele Fontanelli, Luigi Palopoli, and Luca Greco. Deterministic and Stochastic QoS Provision for Real-Time Control Systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 103–112, 2011.

[42] Tully Foote. Tf: The Transform Library. In *Proceedings of the 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, 2013.

[43] Tully Foote. ROS Community Metrics Report. `http://download.ros.org/downloads/metrics/metrics-report-2020-07.pdf`, 2020.

[44] Autoware Foundation. Autoware.Auto. Project website: `https://www.autoware.org/autoware-auto`, 2021.

[45] Eclipse Foundation. Eclipse Cyclone DDS 0.8.0 Release Announcement. `https://projects.eclipse.org/projects/iot.cyclonedds/releases/0.8.0-replique`, 2021.

[46] Eclipse Foundation. The Iceoryx middleware. Project website: `https://iceoryx.io/v1.0.1/`, 2021.

[47] Open Source Robotics Foundation. The ROS2 Technical Steering Committee. `https://docs.ros.org/en/galactic/Governance.html`.

[48] Open Source Robotics Foundation. Noetic Ninjemys: The Last Official ROS 1 Release. Release announcement: `https://www.openrobotics.org/blog/2020/5/23/noetic-ninjemys-the-last-official-ros-1-release`, 2020.

[49] Open Source Robotics Foundation. Overview of robots built with ROS. `http://robots.ros.org`, 2021.

[50] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The Dynamic Window Approach to Collision Avoidance. *IEEE Robotics & Automation Magazine*, 4:23–33, 1997.

[51] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, pages 479–495, 2002.

[52] Brian Gerkey. Why ROS 2.0? `http://design.ros2.org/articles/why_ros2.html`, 2015.

[53] Nicolas Gobillot, Fabrice Guet, David Doose, Christophe Grand, Charles Lesire, and Luca Santinelli. Measurement-Based Real-Time Analysis of Robotic Software Architectures. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3306–3311, 2016.

[54] Ishu Goel. SingleThreadedExecutor creates a high CPU overhead in ROS 2. Discussion in the ROS 2 Discord forum, `https://discourse.ros.org/t/singlethreadedexecutor-creates-a-high-cpu-overhead-in-ros-2/10077`.

[55] Stefan Groesbrink, Luis Almeida, Mario de Sousa, and Stefan M. Petters. Towards Certifiable Adaptive Reservations for Hypervisor-Based Virtualization. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–24, 2014.

[56] Carlos San Vicente Gutiérrez, Lander Usategui San Juan, Irati Zamalloa Ugarte, and Víctor Mayoral Vilches. Towards a Distributed and Real-Time Framework for Robots: Evaluation of ROS 2.0 Communications for Real-Time Robotic Applications. arXiv:1809.02595 [cs.RO], 2018.

[57] Akram Hakiri, Pascal Berthou, Aniruddha S. Gokhale, Douglas C. Schmidt, and Thierry Gayraud. Supporting End-to-End Quality of Service Properties in OMG Data Distribution Service Publish/Subscribe Middleware over Wide Area Networks. *Journal of Systems and Software*, 86:2574–2593, 2013.

[58] Michael Gonzalez Harbour, Mark H. Klein, and John P. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. In *Proceedings of the 12th Real-Time Systems Symposium (RTSS)*, pages 116–128, 1991.

[59] Houcine Hassan, José Simó, and Alfons Crespo. Flexible Real-Time Mobile Robotic Architecture based on Behavioural Models. *Engineering Applications of Artificial Intelligence*, 14:685–702, 2001.

[60] Martijn Hendriks and Marcel Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 8–15, 2006.

[61] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System Level Performance Analysis - the SymTA/S Approach. *IEE Proceedings - Computers and Digital Techniques*, 152:148–166, 2005.

[62] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In *International Workshop on Embedded Software (EMSOFT)*, pages 166–184, 2001.

[63] Robin Hofmann, Leonie Ahrendts, and Rolf Ernst. CPA – Compositional Performance Analysis. In Jürgen Teich and Soonhoi Ha, editors, *Handbook of Hardware/Software Codesign*. 2017.

[64] David S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[65] Matej Kristan, Jiri Matas, Ales Leonardis, Tomas Vojir, Roman Pflugfelder, Gustavo Fernandez, Georg Nebehay, Fatih Porikli, and Luka Cehovin. A Novel Performance Evaluation Methodology for Single-Target Trackers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38:2137–2155, 2016.

[66] Matej Kristan, Ales Leonardis, Jiri Matas, Michael Felsberg, Roman Pfugfelder, Luka Čehovin Zajc, Tomas Vojir, Goutam Bhat, Alan Lukezic, Abdelrahman Eldesokey, and Gustavo Fernandez. The sixth visual object tracking VOT2018 challenge results. In *VOT2018 Workshop*, 2018.

[67] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. 2001.

[68] Giuseppe Lipari and Sanjoy Baruah. Greedy Reclamation of Unused Bandwidth in Constant-Bandwidth Servers. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 193–200, 2000.

[69] Giuseppe Lipari and Enrico Bini. Resource Partitioning among Real-Time Applications. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 151–158, 2003.

[70] Giuseppe Lipari and Enrico Bini. A Methodology for Designing Hierarchical Scheduling Systems. *Journal of Embedded Computing*, 1:9, 2005.

[71] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[72] Jane W. S. Liu. *Real-Time Systems*. 2000.

[73] Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. Combining robotics

component-based model-driven development with a model-based performance analysis. In *Proceedings of the 8th IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 170–176, 2016.

[74] Chenyang Lu, John A. Stankovic, Tarek F. Abdelzaher, Gang Tao, Sang Hyuk Son, and Michael Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS)*, pages 13–23, 2000.

[75] Chenyang Lu, John A. Stankovic, Sang Hyuk Son, and Gang Tao. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Real-Time Systems*, 23: 85–126, 2002.

[76] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. The Marathon 2: A Navigation System. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2718–2725, 2020.

[77] Clifford W. Mercer and Ragunathan Rajkumar. An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management. In *Proceedings of the 1st Real-Time Technology and Applications Symposium (RTAS)*, pages 134–139, 1995.

[78] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: Yet another robot platform. *International Journal of Advanced Robotic Systems*, 3:43–48, 2006.

[79] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology (MIT), 1983.

[80] Aloysius K. Mok and Deji Chen. A Multiframe Model for Real-Time Tasks. *IEEE Transactions on Software Engineering*, 23:635–645, 1997.

[81] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study. *Computer Science and Information Systems*, 10:453–482, 2013.

[82] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. Design Principles of the Component-Based Robot Software Framework Fawkes. In *Proceedings of the 2nd Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, volume 6472, pages 300–311, 2010.

[83] Vladimir Nikolov, Stefan Wesner, Eugen Frasch, and Franz J. Hauck. A Hierarchical Scheduling Model for Dynamic Soft-Realtime System. *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 7:1–7:23, 2017.

[84] Object Management Group. *Data Distribution Service (DDS)*. 2015.

[85] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 58–68, 1993.

[86] José C. Palencia and Michael G. Harbour. Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, 2003.

[87] José C. Palencia and Michael G. Harbour. Response Time Analysis of EDF Distributed Real-Time Systems. *Journal of Embedded Computing*, 1:225–237, 2005.

[88] Luigi Palopoli and Luca Abeni. Legacy Real-Time Applications in a Reservation-Based System. *IEEE Transactions on Industrial Informatics*, 5:220–228, 2009.

[89] Luigi Palopoli, Luca Abeni, and Giuseppe Lipari. On the Application of Hybrid Control to CPU Reservations. In *International Workshop on Hybrid Systems: Computation and Control*, pages 389–404, 2003.

[90] Luigi Palopoli, Tommaso Cucinotta, Luca Marzario, and Giuseppe Lipari. AQuoSA—Adaptive Quality of Service Architecture. *Software: Practice and Experience*, 39:1–31, 2009.

[91] Jaeho Park, Raimarius Delgado, and Byoung Wook Choi. Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study. *IEEE Access*, 8:154637–154651, 2020.

[92] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael G. Harbour. Influence of Different System Abstractions on the Performance Analysis of Distributed Real-Time Systems. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, pages 193–202, 2007.

[93] Héctor Pérez and J. Javier Gutiérrez. Modeling the QoS parameters of DDS for Event-Driven Real-Time Applications. *Journal of Systems and Software*, 104:126–140, 2015.

[94] Maurizio Piaggio and Renato Zaccaria. Distributing a Robotic System on a Network: The ETHNOS Approach. *Advanced Robotics*, 11:743–758, 1996.

[95] Maurizio Piaggio, Antonio Sgorbissa, and Renato Zaccaria. A Programming Environment for Real-Time Control of Distributed Multiple Robotic Systems. *Advanced Robotics*, 14: 75–86, 2000.

[96] Friedrich Pukelsheim. The Three Sigma Rule. *The American Statistician*, 48:88–91, 1994.

[97] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, 2009.

[98] Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal Analysis of Sporadic Overload in Real-Time Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 515–520, 2012.

[99] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Multimedia Computing and Networking*, volume 3310, pages 150–164, 1998.

[100] Real-Time Innovations, Inc. RTI Connext DDS – Comprehensive Summary of QoS Policies. `https://community.rti.com/static/documentation/connext-dds/5.2.0/doc/manuals/connext_dds/RTI_ConnextDDS_CoreLibraries_QoS_Reference_Guide.pdf`, 2015.

[101] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models: The SymTA/S Approach.* PhD thesis, University of Braunschweig - Institute of Technology, 2005.

[102] Yukihiro Saito, Takuya Azumi, Shinpei Kato, and Nobushiko Nishio. Priority and Synchronization Support for ROS. In *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 77–82, 2016.

[103] Luca Santinelli, Fabrice Guet, and Jerome Morio. Revising Measurement-Based Probabilistic Timing Analysis. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 199–208, 2017.

[104] Johannes Schlatow. *Enabling In-Field Integration in Critical Embedded Systems*. PhD thesis, Braunschweig University of Technology, 2021.

[105] Johannes Schlatow and Rolf Ernst. Response-Time Analysis for Task Chains in Communicating Threads. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–10, 2016.

[106] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[107] Nishanth Shankaran, Xenofon D. Koutsoukos, Douglas C. Schmidt, Yuan Xue, and Chenyang Lu. Hierarchical Control of Multiple Resources in Distributed Real-Time and Embedded Systems. *Real-Time Systems*, 39:237–282, 2008.

[108] Insik Shin and Insup Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 2–13, 2003.

[109] Bruno Siciliano and Oussama Khatib, editors. *Springer Handbook of Robotics*. 2008.

[110] Karila Palma Silva, Luis Fernando Arcaro, and Romulo Silva De Oliveira. On Using GEV or Gumbel Models When Applying EVT for Probabilistic WCET Estimation. In *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS)*, pages 220–230, 2017.

[111] Karila Palma Silva, Luis Fernando Arcaro, Daniel Bristot de Oliveira, and Romulo Silva de Oliveira. An Empirical Study on the Adequacy of MBPTA for Tasks Executed on a Complex Computer Architecture with Linux. In *Proceedings of the 23rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 321–328, 2018.

[112] Jack A. Stankovic, Chenyang Lu, Sang Hyuk Son, and Gang Tao. The Case for Feedback Control Real-Time Scheduling. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 11–20, 1999.

[113] Jan Staschulat, Ingo Lütkebohle, and Ralph Lange. The rclc Executor: Domain-Specific Deterministic Scheduling Mechanisms for ROS Applications on Microcontrollers: Work-in-progress. In *Proceedings of the 17th International Conference on Embedded Software (EMSOFT)*, pages 18–19, 2020.

[114] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The Digraph Real-Time Task Model. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[115] Yuhei Suzuki, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Real-Time ROS Extension on Transparent CPU/GPU Coordination Mechanism. In *Proceedings of the*

21st IEEE International Symposium on Real-Time Distributed Computing (ISORC), pages 184–192, 2018.

[116] Yue Tang, Feng Zhiwei, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors. *Proceedings of the 41st IEEE Real-Time Systems Symposium (RTSS)*, pages 231–243, 2020.

[117] Yue Tang, Nan Guan, Zhiwei Feng, Xu Jiang, and Wang Yi. Response Time Analysis of Lazy Round Robin. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 258–263, 2021.

[118] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[119] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, Alexander Maxiaguine, and Jonas Greutert. Embedded Software in Network Processors — Models and Algorithms. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT)*, pages 416–434, 2001.