Performance Engineering for Real-Time Autonomous Applications

A thesis approved for the academic degree of

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

at the

Faculty of Electrical Engineering and Information Technology
Technical University Dortmund

by

Hazem Abaza
Supervisor
Prof. Dr.-Ing. Selma Saidi

Second Examiner
Prof. Dr. Frank Hoffmann

Leistungstechnik für echtzeitfähige autonome Anwendungen

Eine Dissertation zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

an der

Fakultät für Elektrotechnik und Informationstechnik
Technische Universität Dortmund

von

Hazem Abaza

Referentin

Prof. Dr.-Ing. Selma Saidi

Korreferent

Prof. Dr. Frank Hoffmann

Abstract

In the dynamic landscape of autonomous system development, there is an increasing need to enhance the integration and interaction between software and hardware components. These systems engage in intricate internal communications to actively perceive and adapt to changes in the external environment. Given their critical role in safety, they are bound by rigorous timing constraints to maintain both temporal and functional integrity. Achieving high precision for these systems necessitates a deep understanding of their functional requirements and operational domain. This foundational insight is crucial for meeting their stringent timing requirements and expanding their computational prowess. Such advancements could be further realized by adopting more sophisticated, decentralized, and interconnected architectural frameworks, which would enable these systems to integrate and react seamlessly to the complexities of real-world environments. Therefore, the design of such safety-critical autonomous systems presents significant challenges when addressing the system's real-time properties, requiring innovative solutions to ensure reliability and responsiveness under all conditions.

Challenges towards modeling autonomous systems: Modeling autonomous systems comprehensively presents several challenges. Firstly, understanding the interactions between system components is not straightforward, given their complexity and the multitude of functionalities that may interact within these systems. Secondly, the behavior of these systems is influenced by changes in the physical environment. While these systems are typically periodic, their execution can vary depending on computational load and environmental conditions. This thesis presents our approach to addressing these challenges through a generic methodology for extracting timing models for middleware-based autonomous functionalities. This is achieved without compromising their performance or altering their predefined confidentiality constraints, which is particularly important when dealing with third-party developed components. We demonstrate our approach capabilities by synthesizing the timing model of a real-world benchmark implementing LIDAR-based Autonomous Valet Parking application.

Deterministic computation of autonomous systems: Designs of autonomous systems are often developed along separate design trajectories before being integrated. This integration can significantly disrupt performance and timing properties when components are consolidated onto the same hardware platform, i.e., sharing the same computational resources. In this thesis, we delve into the significance of timing models for identifying bottlenecks within the integrated system. We have analyzed the performance of these systems under various scheduling algorithms to ensure determinism in execution. To manage the performance

from sensing to actuation in integrated applications, we adopted a scheduler-middleware co-design approach to achieve deterministic and jitters-free execution. Our methodology shows a significant improvement in performance over existing state-of-the-art approaches, with our findings also benchmarked on a LIDAR-based Autonomous Valet Parking application.

Deterministic communication in autonomous systems: As autonomous systems evolve, ensuring performance and safety on hardware where resources are limited becomes increasingly challenging. The shift towards distributed architectures emerges as a logical response to enhance performance and safety. Integrating multiple components on constrained embedded systems introduces more bottlenecks, complicating safety assurances. However, the success of distributed architectures hinges on deterministic networks capable of CPU offloading, i.e., networks that do not require CPU intervention for every data transfer. In this context, we have developed a network protocol that ensures determinism. This work evaluates the lack of determinism in existing protocols and introduces a lightweight multi-layer communication stack designed to imbue real-time capabilities with minimal overhead. Our stack offers straightforward interfaces, allowing applications to leverage low-level protocols for communication, thereby ensuring adherence to real-time requirements throughout the communication process. Our experiments demonstrate that our stack significantly reduces both communication latency and CPU utilization in comparison to existing protocols. Moreover, we achieve an overhead of less than 1% relative to real-time unaware communication when executing a neural network-based object detection application.

Summary: This thesis proposes a holistic solution to ensure determinism in both computation and communication within autonomous systems, addressing the integration and interaction challenges between software and hardware components. It delves into the importance of understanding system components' interactions and the influence of the physical environment on system behavior, highlighting the necessity of timing models to identify bottlenecks and maintain performance under varying operational conditions. By adopting a scheduler-middleware co-design approach, the thesis achieves deterministic and jitter-free execution, demonstrating performance improvement over existing methodologies. Furthermore, it introduces a deterministic multi-layer communication stack that reduces communication latency and CPU utilization, ensuring real-time capabilities with minimal overhead. This innovative approach not only enhances the performance and safety of autonomous systems but also addresses the challenges posed by distributed architectures and the need for CPU offloading, achieving minimum overhead in real-world applications.

Zusammenfassung

In der dynamischen Landschaft der Entwicklung autonomer Systeme besteht ein zunehmender Bedarf, die Integration und Interaktion zwischen Software- und Hardwarekomponenten zu verbessern. Diese Systeme führen eine komplizierte interne Kommunikation durch, um Veränderungen in der äuSSeren Umgebung aktiv wahrzunehmen und sich an sie anzupassen. In Anbetracht ihrer kritischen Rolle für die Sicherheit sind sie an strenge Zeitvorgaben gebunden, um sowohl die zeitliche als auch die funktionale Integrität zu wahren. Um eine hohe Präzision für diese Systeme zu erreichen, ist ein tiefes Verständnis ihrer funktionalen Anforderungen und ihres Einsatzbereichs erforderlich. Diese grundlegenden Erkenntnisse sind entscheidend, um die strengen Zeitvorgaben zu erfüllen und ihre Rechenleistung zu erweitern. Derartige Fortschritte könnten durch die Einführung ausgefeilterer, dezentralisierter und vernetzter Architekturen erzielt werden, die es diesen Systemen ermöglichen, sich nahtlos in die komplexen Umgebungen der realen Welt zu integrieren und darauf zu reagieren. Die Entwicklung solcher sicherheitskritischen autonomen Systeme stellt daher eine groSSe Herausforderung dar, wenn es um die Echtzeiteigenschaften des Systems geht, und erfordert innovative Lösungen, um Zuverlässigkeit und Reaktionsfähigkeit unter allen Bedingungen zu gewährleisten.

Herausforderungen bei der Modellierung autonomen Systeme: Die umfassende Modellierung autonomer Systeme ist mit mehreren Herausforderungen verbunden. Erstens ist das Verständnis der Wechselwirkungen zwischen den Systemkomponenten angesichts ihrer Komplexität und der Vielzahl von Funktionen, die in diesen Systemen interagieren können, nicht ganz einfach. Zweitens wird das Verhalten dieser Systeme durch Veränderungen in der physikalischen Umgebung beeinflusst. Während diese Systeme in der Regel peridoisch sind, kann ihre Ausführung je nach Rechenlast und Umgebungsbedingungen variieren. In dieser Arbeit wird unser Ansatz zur Bewältigung dieser Herausforderungen durch eine generische Methodik zur Extraktion von Zeitmodellen für Middleware-basierte autonome Funktionalitäten vorgestellt. Dies wird erreicht, ohne deren Leistung zu beeinträchtigen oder ihre vordefinierten Vertraulichkeitsbedingungen zu ändern, was besonders wichtig ist, wenn es sich um von Dritten entwickelte Komponenten handelt. Wir demonstrieren Fähigkeiten unseres Ansatzes durch die Synthese des Zeitmodells eines einer realen Benchmark-Anwendung, die eine autonome Parkservice-Anwendung implementiert.

Deterministische Berechnungen in autonomen Systemen: Die Entwürfe für autonome Systeme werden häufig in getrennten Entwicklungsphasen entwickelt, bevor sie integriert werden. Diese Integration kann die Leistungs- und Timing-Eigenschaften beeinträchtigen, insbesondere wenn die Komponenten auf derselben Hardware-Plattform konsolidiert werden, d.h. dieselben Rechenressourcen nutzen. In dieser Arbeit befassen wir uns

mit der Bedeutung von Timing-Modellen für die Identifizierung von Engpässen innerhalb des integrierten Systems. Wir haben die Leistung dieser Systeme unter verschiedenen Zeitplanungsalgorithmen analysiert, um Determinismus bei der Ausführung zu gewährleisten. Um die Leistung von der Erfassung bis zur Betätigung in integrierten Anwendungen zu steuern, haben wir einen Scheduler-Middleware-Co-Design-Ansatz gewählt, um eine deterministische und ruckelfreie Ausführung zu erreichen. Unsere Methodik zeigt eine 13%ige Leistungsverbesserung im Vergleich zu bestehenden State-of-the-Art-Ansätzen, wobei unsere Ergebnisse auch in einer autonomen Valet-Parking-Anwendung getestet wurden.

Deterministische Kommunikation in autonomen Systemen: Mit der Weiterentwicklung autonomer Systeme wird es immer schwieriger, Leistung und Sicherheit auf Hardware mit begrenzten Ressourcen zu gewährleisten. Der Wechsel zu verteilten Architekturen ist eine logische Reaktion, um sowohl die Leistung als auch die Sicherheit zu verbessern. Die Integration mehrerer Komponenten in begrenzte eingebettete Systeme führt zu weiteren Engpässen und erschwert die Sicherheitsgarantien. Der Erfolg verteilter Architekturen hängt jedoch von deterministischen Netzwerken ab, die in der Lage sind, die CPU zu entlasten, d. h. von Netzwerken, die nicht für jede Datenübertragung einen Eingriff der CPU erfordern. In diesem Zusammenhang haben wir ein Netzwerkprotokoll entwickelt, das Determinismus gewährleistet. Diese Arbeit bewertet den Mangel an Determinismus in bestehenden Protokollen und führt einen leichtgewichtigen, mehrschichtigen Kommunikationsstack ein, der darauf ausgelegt ist, Echtzeitfähigkeiten mit minimalem Overhead zu vermitteln. Unser Stack bietet einfache Schnittstellen, die es Anwendungen ermöglichen, Low-Level-Protokolle für die Kommunikation zu nutzen und so die Einhaltung von Echtzeitanforderungen während des gesamten Kommunikationsprozesses zu gewährleisten. Unsere Experimente zeigen, dass unser Stack sowohl die Kommunikationslatenz als auch die CPU-Auslastung im Vergleich zu bestehenden Protokollen deutlich reduziert. Darüber hinaus erreichen wir bei der Ausführung einer auf einem neuronalen Netz basierenden Objekterkennungsanwendung einen Overhead von weniger als 1%.

Übersicht: In dieser Arbeit wird eine ganzheitliche Lösung vorgeschlagen, um Determinismus sowohl bei der Berechnung als auch bei der Kommunikation in autonomen Systemen zu gewährleisten, wobei die Herausforderungen der Integration und Interaktion zwischen Software- und Hardwarekomponenten berücksichtigt werden. Sie befasst sich mit der Bedeutung des Verständnisses der Interaktionen von Systemkomponenten und des Einflusses der physikalischen Umgebung auf das Systemverhalten und unterstreicht die Notwendigkeit von Timing-Modellen zur Identifizierung von Engpässen und zur Aufrechterhaltung der Leistung unter verschiedenen Betriebsbedingungen. Durch die Anwendung eines Scheduler-Middleware-Co-Design-Ansatzes wird in dieser Arbeit eine deterministische und Jitter-freie Ausführung erreicht, was eine signifikante Leistungsverbesserung gegenüber bestehenden Methoden darstellt. Darüber hinaus wird ein deterministischer, mehrschichtiger Kommunikationsstack eingeführt, der die Kommunikationslatenz und die CPU-Auslastung reduziert und so die Echtzeitfähigkeit bei minimalem Overhead gewährleistet. Dieser innovative Ansatz verbessert nicht nur die Leistung und Sicherheit autonomer Systeme, sondern geht auch auf die Herausforderungen ein, die sich aus verteilten Architekturen und der Notwendigkeit der CPU-Auslastung ergeben.

Contents

List of Figures

List of Tables

Abbreviations

ADAS Autonomous Driving-Assisted Systems

CAN Controller Area Network

CB Callback

CPU Central Processing Unit

DAG Data Acyclic Graph

DDS Data Distributed System

DSP Digital Signal Processor

GPU Graphical Processing Unit

ISP Image Signal Processor

LET Logical Execution Time

LIN Local Interconnect Network

OEM Original Equipment Manufacturer

QoS Quality of Service

RDMA Remote Direct Memory Access

RoCE RDMA over Converged Ethernet

ROS2 Robot Operating System version 2

SOA Service-Oriented Architecture

SoC System-on-Chip

TCP Transmission Control Protocol

TSN Time Sensitive Network

Chapter 1

Introduction

The rise and development of autonomous technologies have marked a significant milestone in the evolution of the tech landscape, integrating deeply with key sectors such as automation, artificial intelligence (AI), and cyber-physical systems. Unlike traditional automated setups, autonomous technologies are defined by two main features: self-governance and self-management. Self-governance denotes the technology's capability for independent operation and decision-making, allowing it to adapt its actions based on an understanding of its environment and internal workings. This adaptability is critical for the technology's ability to respond to changing situations without human intervention. On the other hand, self-management highlights the technology's capacity for self-sufficiency and self-repair, which is vital for maintaining continuous operation and resilience against unexpected issues or failures. This self-reliant nature is the foundation of an autonomous system's operational integrity, ensuring its reliability in diverse conditions [116].

The progression toward incorporating autonomous systems in vehicles has its origins in early automotive safety and convenience innovations. Initially, efforts were centered around fundamental safety measures such as seat belts and anti-lock braking systems (ABS), aimed at protecting passengers and improving vehicle handling [119]. With advances in sensor technology, computing, and algorithms, the auto industry shifted toward more complex systems designed to prevent accidents and reduce the driver's workload. This led to the introduction of advanced features like electronic stability control and automated alerts, setting the stage for today's sophisticated autonomous driving-assisted systems (ADAS) technologies. These systems, which leverage multiple sensors and data sources, represent a significant step toward fully autonomous vehicles, offering a broad spectrum of safety and convenience functionalities [35] [97] [4].

Autonomous platforms are intricate ecosystems of hardware and software, designed to process extensive environmental data and execute complex decisions instantly. At their heart are sophisticated software algorithms essential for interpreting sensor data, encompassing everything from basic operating systems to advanced applications for image recognition and decision-making. This software infrastructure is critical for the seamless integration of data and functionalities, employing AI and machine learning to adapt to dynamic driving conditions. Hardware-wise, autonomous systems rely on various sensors such as cameras,

LiDAR, radar, and ultrasonic sensors, each contributing uniquely to the vehicles perception capabilities. These sensors generate a considerable amount of data, requiring powerful processors to analyze and act on this information promptly and reliably [100] [144].

To meet the computational needs of autonomous systems, platforms utilize System-on-Chips (SoCs) with specialized AI accelerators, essential for real-time processing of complex tasks. These SoCs combine traditional computing cores with graphics and digital signal processors, facilitating rapid data processing and neural network computations. The use of AI accelerators is essential for handling high-resolution sensor inputs, ensuring swift and precise decision-making essential for vehicle safety and performance [37] [118]. This leads to an architecture of heterogeneous platforms, which sometimes also necessitates the incorporation of networks in between to enhance their computation capabilities, increasing the complexity of the system more and more, resulting in distributed heterogeneous platforms connected over a network. This evolution not only expands the computational fabric but also embeds greater flexibility and scalability into the system architecture for the advancing of the overall system intelligence. On the software level, incorporating middlewares, especially those following Service-Oriented Architectures(SOA) as [10] and [109], introduces a new dimension of flexibility for autonomous systems abstracting the complexity of underlying distributed systems. Middleware frameworks facilitate the development of applications as a collection of loosely coupled services, which can be independently developed, deployed, and scaled. This programming model significantly enhances system modularity and agility, enabling rapid adaptation to changing requirements and technologies. However, while SOAs offer remarkable flexibility, they also introduce computation and communication determinism complexities, particularly when integrated with distributed architectures.

This hardware-software ecosystem has to follow a deterministic real-time computing paradigm to ensure that the system processes react to external inputs within a specific timeframe, vital for autonomous systems's ability to make timely adjustments to vehicle operation to ensure passenger safety. The adoption of SOAs in conjunction with distributed architectures necessitates a nuanced understanding of their impact on system performance, especially concerning real-time operation. The dynamic nature of service discovery and binding, inherent to SOAs, can introduce variability in response times, thereby affecting computation determinism. Similarly, the communication among distributed servicesoften reliant on networked interactionscan be subjected to unpredictable latencies, thus impacting communication determinism. These factors can compound the challenges of maintaining real-time performance, highlighting the need for sophisticated design strategies that balance the benefits of flexibility with the imperative of determinism. Addressing these challenges requires a holistic approach that includes optimizing middleware configurations, employing real-time communication protocols that support service-oriented interactions, and implementing network management strategies tailored to the specifics of distributed autonomous architectures. By carefully considering these aspects, it is possible to harness the flexibility offered by middleware and SOAs while mitigating their potential impacts on computation and communication determinism, thereby ensuring that the distributed system architecture effectively supports the real-time, safety-critical functionalities of autonomous systems.

The journey towards optimizing the performance of such autonomous sophisticated systems and ensuring their reliability begins with a comprehensive understanding of their operational state and interactions. Delving into the intricacies of the system's timing architecture by extracting snippets of its configuration provides an invaluable first step in this process. This initial exploration is critical for mapping out the system's interactions, setting the stage for the identification of performance bottlenecks, and facilitating targeted optimizations. Moreover, it lays the foundation for a thorough understanding of the hardware-software interplay, a crucial aspect when considering resource allocation and capability enhancement.

With a foundational understanding of the system's timing architecture in place, the next critical step involves harnessing this knowledge to enforce deterministic execution within middleware frameworks that drive autonomous applications. By meticulously analyzing the timing model extracted from the initial architecture understanding, this thesis endeavors to develop strategies that precisely control the latency and jitter inherent in these complex systems. This approach aims to achieve a level of execution predictability that is paramount for the seamless operation of autonomous vehicles and other critical applications reliant on real-time data processing and decision-making.

Building upon the basic understanding of the timing behaviors of applications, this thesis extends its exploration into proposing a deterministic communication network topology specifically designed to connect distributed architectures running autonomous applications. Recognizing the pivotal role that middleware plays in orchestrating these complex systems, the proposed network topology aims to support middleware frameworks and AI-based applications efficiently, ensuring that communication between distributed components is not only reliable but inherently deterministic. This advanced exploration seeks to address one of the fundamental challenges in the deployment of autonomous systems: the need for a communication infrastructure that can guarantee the timely and predictable exchange of data across distributed nodes while reducing CPU interference in managing the network.

This thesis addresses the challenges of the hardware-software co-design complexity by first focusing on the extraction of the timing architecture to gain insight into the system's interactions. Such an approach not only aids in pinpointing areas where efficiency can be improved but also enhances our understanding of the system's real-time operational demands. Building on this foundation, the research introduces innovative methodologies aimed at managing latency and jitter for computing and communicating tasks, thereby ensuring that real-time tasks are executed within their designated deadlines. The holistic approach detailed in the thesis encompasses the development and application of methodologies and strategies specifically designed to control and mitigate variations in execution times and data transmission delayskey factors crucial to the system's real-time performance and reliability. This research integrates deterministic principles into the computational and communicational facets of autonomous systems, presenting a comprehensive solution that significantly advances the field of autonomous systems design and operation. It highlights efforts to ensure temporal and functional integrity, allowing these systems to perform their functions with the precision and consistency vital for safety-critical applications.

1.1 Research Questions

Our research aims to address the critical challenges in achieving deterministic and real-time performance in autonomous applications, which are essential for the safety and efficiency of autonomous vehicles. This research focuses on the extraction and analysis of timing properties and architectures, the assurance of jitter-free execution with controlled latency, and the design of deterministic networks to support autonomous applications. The objective is to empower system developers and engineers with methodologies and tools for:

Extracting and understanding the timing architecture of autonomous applications to identify potential interferences and optimize system performance across the distributed network. We are leveraging extracted timing information to guarantee the execution of autonomous applications with minimal jitter and latency, ensuring real-time responsiveness. Designing and implementing deterministic network architectures that provide reliable and predictable performance for distributed autonomous applications. This leads to the principal research question: **How can system developers ensure deterministic and real-time performance in distributed autonomous environments?**

The principal research question at the heart of our investigation addresses three crucial areas that merit deeper examination: timing architecture extraction, jitter and latency management, and deterministic networking. Each area plays a vital role in optimizing the performance and reliability of autonomous systems. Firstly, timing architecture extraction is fundamental in understanding the timing behavior and interactions of autonomous applications. This process is key to identifying performance bottlenecks and enhancing the overall efficiency of the system. To achieve this, it is imperative to develop comprehensive methodologies that allow for the accurate capture and analysis of timing properties, enabling a deeper insight into system operations. Secondly, jitter and latency management focuses on developing strategies that ensure autonomous applications execute within their designated time frames, minimizing delay and variability. By leveraging detailed timing data, it is possible to refine scheduling algorithms and optimize resource allocation. This ensures that the system's responses are both timely and consistent, critical for maintaining operational integrity and performance. Lastly, deterministic networking addresses the challenge of creating network infrastructures that can meet the strict real-time requirements of distributed autonomous applications. This includes the development of specialized protocols and architectural solutions designed to deliver consistent and predictable network performance. By minimizing the impact of network-induced delays, these solutions ensure the reliable transmission of critical data across the network. Together, these three areas form the foundation of our research, guiding our efforts to create more efficient, reliable, and predictable autonomous systems. Through the meticulous exploration of timing architecture extraction, jit,ter and latency management, and deterministic networking, we aim to push the boundaries of current technology and pave the way for future advancements in autonomous applications.

The principal question itself can be further broken down into three sub-questions, each targeting different aspects of the main research question:

Research Question 1 (Extract): How can timing properties and architectures of middleware-based autonomous applications be effectively extracted and analyzed to optimize system performance and minimize inter-application interference?

This research question's goal is to delve into the development of comprehensive methodologies for the extraction and analysis of timing properties and architectural details of applications running within autonomous systems. The aim is to uncover the intricate dynamics of application interactions, identify potential bottlenecks, and understand how different components' timing behaviors impact overall system performance. By accurately capturing and analyzing these properties, the research seeks to develop optimization strategies that can minimize inter-application interference, thereby enhancing the predictability and reliability of autonomous functionalities.

Research Question 2 (Manage): How can jitter-free execution with controlled latency be guaranteed for autonomous applications using extracted timing information?

Building on the foundational understanding of autonomous systems timing architectures, this question explores the application of extracted timing data towards achieving deterministic execution of system functions, with a particular focus on minimizing jitter and managing latency effectively. The challenge lies in adapting methodologies that can accommodate the diverse and dynamic nature of autonomous workloads while ensuring that critical tasks meet their timing constraints. This includes concerted adaptive policies that can respond to real-time changes in system state and workload, methods for predicting and mitigating potential latency spikes, and techniques for optimizing resource allocation in a way that balances performance with real-time responsiveness. The goal is to ensure that autonomous systems can reliably perform their critical functions within the required time frames, even in the face of complex and unpredictable operating environments.

Research Question 3 (Exploration): How can network architectures be optimized to support the real-time requirements of distributed autonomous applications, mitigating the CPU centrism effect in a network-connected distributed architecture?

This question delves into the underlying network infrastructure essential for distributed autonomous systems, aiming to find solutions for developing network architectures that can fulfill the demanding real-time communication requirements while minimizing the CPU centrism effect. The primary objective is to design network protocols that guarantee timely and predictable data delivery, with an overarching goal of offloading the CPU from unnecessary excessive data copying. This involves creating network topologies that reduce latency and transmission variability and implementing Quality of Service (QoS) mechanisms to prioritize network traffic, all tailored to the critical nature of autonomous functionalities. The research focus is to lay the foundation for a network infrastructure that satisfies the immediate real-time communication needs of autonomous applications and remains flexible and scalable for future advancements in autonomous vehicle technology.

The goal of this research is to develop methodologies and tools that ensure deterministic and real-time performance for autonomous applications. This involves a focused exploration into three key areas: the extraction and detailed analysis of timing architecture to understand the system performance and mark potential interferences, the assurance of jitter-free execution with minimized latency to ensure real-time responsiveness, and the design of deterministic network architectures that cater to the real-time requirements of distributed applications. Through this comprehensive approach, the research aims to provide system developers and engineers with the necessary insights and instruments to enhance the safety, efficiency, and reliability of autonomous systems, aligning with the stringent standards required for automotive safety and functionality.

1.2 publications

The main contributions of this thesis result from Huawei Dresden Research Center-funded research projects that were presented in peer-reviewed publications at international top-tier venues which we describe in the following:

- Hazem Abaza, Debayan Roy, Antonios Motakis and Selma Saidi **Trace-enabled Timing Model Synthesis for ROS2-based Autonomous Applications**, 2024, Design Automation and Test in Europe (DATE), Valencia, Spain
- Hazem Abaza, Debayan Roy, Bohdan Trach, Selma Saidi, and others **Mechanism to Shape End-to-End Latency of ROS2 Computation Chains**, 2024, The 32nd International Conference on Real-Time Networks and Systems (RTNS), Porto, Portugal
- Hazem Abaza, Debayan Roy, Selma Saidi and others **RDMA-Based Deterministic and Low-Latency Communication Architecture for Autonomous Driving Applications**, 2023, IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Niigata, Japan
- **Patent:** Method and Related Apparatus for Controlling Delay Jitter in a Task Chain Debayan Roy, Sergey Tverdyshev, Hazem Abaza, **Case No.: 10000514298831 Application Protocol Volume Number: S23P004103, The State Intellectual Property, Beijing, China**

1.3 Thesis organization

The structure of this thesis unfolds progressively, beginning with a foundational exploration of state-of-the-art technologies and methodologies in Chapter 2, where signal-to-service-oriented architectures, middleware technologies such as Adaptive AUTOSAR and ROS2, and timing models in ADAS and autonomous applications are thoroughly examined. This foundational knowledge serves as a precursor to Chapter 3, which dives into the specifics of system model extraction for ROS2 autonomous applications, highlighting the crucial role of application tracing using eBPF and the development of a trace-based timing model extraction framework. Chapter 4 transitions into the practical realm, focusing on the management of latency and jitter within ROS2 applications through innovative scheduling techniques and latency shaping methods, enriched with case studies to demonstrate applicability. Chapter 5 ventures into network-level optimizations, advocating for the adoption of RDMA technology in automotive systems, supported by experimental evidence of Soft-RoCEs advantages and proposing a multi-layer communication stack for deterministic, low-latency communication. The thesis culminates in Chapter 6, which synthesizes the key findings and contributions, thereby highlighting the research approach to ensuring determinism in autonomous systems. It outlines the contributions across chapters, from foundational explorations to practical implementations, and sets forth directions for future research to further advance the field.

Chapter 2

State of the Art

2.1 From Signal to Service-Oriented architectures

The evolution of the automotive industry, marked by the transition from signal to service-oriented architectures, represents a fundamental shift in the design and functionality of vehicles. This change has been driven by the rapid adoption of electrical and electronic components within automotive systems. Historically, these systems were based on signal-oriented architectures, which relied on direct, predefined connections between sensors and actuators. Such a model was adequate for more straightforward, more deterministic vehicle interactions, utilizing networks like Controller Area Network (CAN) [69] and Local Interconnect Network (LIN) [114] to ensure the real-time capabilities of applications. However, as vehicles began to incorporate advanced driver-assistance systems (ADAS) and move towards autonomous operation, the limitations of signal-oriented architectures became evident. Introducing approximately 100 Electrical/Electronic Control Units (ECUs) in modern vehicles significantly increased the complexity and bandwidth demands of automotive E/E architectures [123].

Service-oriented architectures (SOA) have emerged as a pivotal solution to these challenges, characterized by dynamic interactions between software components that offer services over a network. Unlike signal-oriented architectures, SOAs define services through interfaces, enabling more flexible, scalable, and modular system designs. This approach allows for the dynamic discovery and utilization of services, making it well-suited to the complex, interconnected systems found in modern vehicles [54]. The comparison between signal-oriented and service-oriented architectures, particularly in the context of functional safety and timing requirements as per ISO 26262 [122], highlights the advantages of SOAs. Signal-oriented systems, with their tight coupling between safety functions and hardware components, offer limited flexibility, complicating the isolation of safety-critical functions and compliance with safety standards. The static nature of signal pathways in these architectures also restricts the system's ability to adapt to changing timing demands, potentially

compromising the execution of safety-critical functions. Conversely, SOAs facilitate a more precise separation and prioritization of safety-critical functions, streamlining the integration and management of functional safety measures. The modular and dynamic nature of SOAs enhances the system's capacity to meet stringent timing requirements, enabling flexible service scheduling and execution. This adaptability ensures that safety-critical functions are adequately prioritized, addressing system resource management more effectively.

2.2 Middlewares for Autonomous Systems

Adaptive AUTOSAR [10] and ROS2 [109] have emerged as foundational technologies in the shift towards SOA, equipping the automotive industry with robust frameworks for dynamic and complex E/E systems. These technologies facilitate the creation of flexible SW components capable of independent updates and deployments across various ECUs. The evolution of ROS2 and Adaptive AUTOSAR mirrors the automotive sector's efforts to navigate the increasing complexity and connectivity demands of contemporary vehicles, emphasizing the significance of SOA in driving automotive innovation. The following subsection will detail the architectures of both Adaptive AUTOSAR and ROS2, pinpointing their differences and showcasing how each framework addresses the current and future requirements of automotive software development.

2.2.1 Adaptive AUTOSAR

Adaptive AUTOSAR is a standardized automotive software architecture developed by the AUTOSAR partnership to address the needs of highly automated and autonomous driving systems. Unlike its predecessor, the Classic AUTOSAR [11], which targets more static and less complex control applications, Adaptive AUTOSAR provides a flexible software infrastructure capable of handling high levels of computing power and complex software functions essential for modern automotive applications. The architecture of Adaptive AUTOSAR is designed to support dynamic and high-performance automotive applications. It is based on a service-oriented architecture (SOA) that allows for the dynamic discovery and binding of services at runtime. This architecture consists of several key layers:

- Application Layer: Contains the application software components developed by the OEMs or third-party suppliers.
- Service Layer: Provides standard services such as communication, diagnostics, and network management to the application layer and other components.
- Adaptive Platform Foundation: contains the core components of the adaptive platform, such as tracing and communication.

The Adaptive AUTOSAR platform marks a significant evolution in automotive software architecture, tailored to meet the intricate and dynamic needs of modern and future vehicle systems. Unlike the Classic AUTOSAR, designed for static, embedded systems in vehicles,

the Adaptive AUTOSAR is a dynamic, service-oriented architecture crafted to support high-performance computing applications such as autonomous driving and enable functionalities like over-the-air (OTA) updates.

As in Figure 2.2, central to distinguishing the Adaptive AUTOSAR architecture are the services and foundational components it comprises. Services like Diagnostics, Security, and Software Configuration are integral to the platform, each serving a distinct purpose. Diagnostics services offer mechanisms for system monitoring and troubleshooting, Security services include measures for safeguarding the system against unauthorized access and cyber threats, and Software Configuration services allow for the dynamic adjustment and management of software settings. Built upon these services are the core components of the Adaptive AUTOSAR platform: Execution Management, Time Management, Platform Health Management, Communication, Hardware Acceleration, Bootloader and OS, and Persistency. Each component is pivotal to the architecture, ensuring a solid foundation for executing automotive applications. Component as Execution Management oversees the lifecycle management of applications, orchestrating their runtime environment to guarantee that resources are allocated efficiently and applications are executed in a controlled manner. Time Management component plays a crucial role in maintaining synchronized time across the system, essential for the coordination and scheduling of tasks, especially in applications demanding precise timing. Platform Health Management is responsible for monitoring the system's health, implementing fault detection, logging, and recovery mechanisms to uphold system reliability and safety. The Communication component ensures seamless inter-component and external communication, supporting various protocols and interfaces necessary for data exchange within the vehicle's electronic systems and with external entities. Hardware Acceleration leverages specialized hardware, such as GPUs or DSPs, for performance-critical operations, facilitating efficient processing of compute-intensive tasks. The Bootloader and OS component manages system startup, firmware updates playing a vital role in system integrity through secure boot mechanisms and lifecycle management of the OS. Lastly, Persistency deals with data storage and retrieval, ensuring data integrity and availability across system restarts, essential for maintaining settings, calibration data, and other information [9].

In summary, The Adaptive AUTOSAR platform's architecture is adeptly designed to cater to the demanding requirements of contemporary automotive systems. By distinguishing between services and foundational components, the platform underscores its capability to facilitate the development of complex, safety-critical automotive applications. Each component within the architecture contributes uniquely to the system's overall functionality and performance, ensuring flexibility, performance, and reliability. The strengths of the Adaptive AUTOSAR platform can be correlated to its standardization, interoperability, consistent specifications within the industry, and satisfaction of automotive requirements, especially regarding security. The platform also provides functionalities related to platform and application lifecycle management, execution management, communication management, and core types. Additionally, it allows for reduced overhead where dynamic communication is not required. These strengths make Adaptive AUTOSAR a valuable platform for addressing the evolving needs of the automotive industry [59].

2.2.2 Robotic Operating System (ROS2)

Robotic Operating System version 2 (ROS2) serves as a potent middleware, facilitating the effortless development and composition of software components for the creation of autonomous applications. It is a widely used open-source framework designed for the development of modular, secure, and scalable autonomous robotic and automotive systems. it decouples the application logic from the mapping of this logic on the target hardware encouraging independent workflow between the application developer and the system integrator. ROS2 is designed as a multi-layer structure as shown in Figure 2.1. The upper layer is the application layer that contains the application's functional implementation. The application uses ROS APIs from the language-specific client libraries *rclcpp* and *rclpy*layer that supports languages such as C++ and Python respectively. The rcl is a C wrapper layer that guarantees comparable performance between applications developed in different languages. The ROS middleware layer (rmw) is used to provide communication between the application entities *nodes*. To support modularity, the rmw layer abstracts the different communication standards implemented underneath and provides a unified interface to the rcl layer.

Generally, each ROS application is structured as a set of nodes each of them is a modular partition for an application feature implemented by a collection of *callback functions*. The callback functions are responsible for computing the node functionality and they can be time-triggered or event-triggered functions. The time-triggered callbacks are periodically invoked per their own rate while event-triggered ones are invoked on an event occurrence. The application nodes communicate with each other using the publish-subscribe paradigm through which nodes publish and subscribe to topics. Whenever a node publishes a message to a topic, subscribing nodes with their callbacks are invoked to trigger a computation or publish a successive message. Nodes communicate with each other using the Data Distribution Service (DDS) [94] which is a network transparent publishing-subscribing mechanism for distributing data with quality of service (QoS) policies in real-time systems. Currently, ROS supports different independent implementations for the DDS standard as FastRTPS [108] and Eclipse Cyclon DDS [40]. ROS systems are constructed by connecting the callbacks of one node to the topic of another node resulting in networks of connected nodes and topics called *chains*. Data are then transferred through these chains in an event-triggered manner crossing the nodes' boundaries to perform the application functionality.

In ROS2, the executor is a key component that allocates callback functions to operating system threads for execution. It manages callbacks including messages from subscriptions, timers, service servers, and action servers, using one or more OS threads. When new messages arrive at a node, the executor queues the related callback functions in its ready queues. The simplest executor, the Single-Threaded Executor, uses the main thread for processing, as in `rclcpp::spin()`. More complex variants, like the Multi-Threaded Executor and the Static Single-Threaded Executor, offer parallel processing or optimized runtime costs by scanning node structures only once.

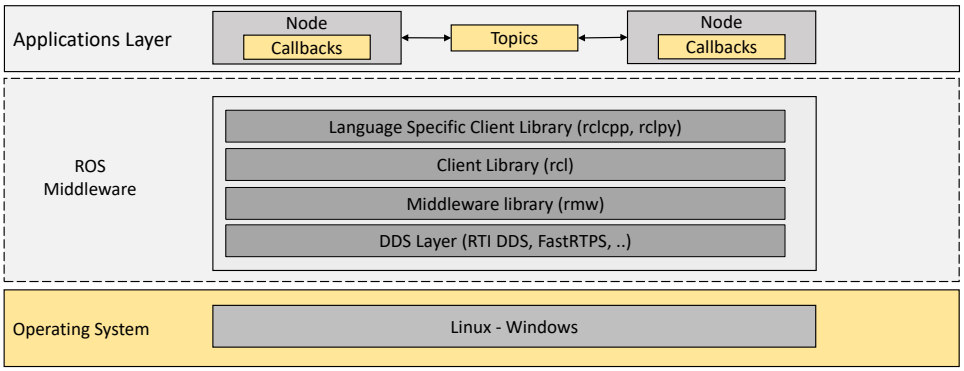


Figure 2.1: ROS2 architecture.

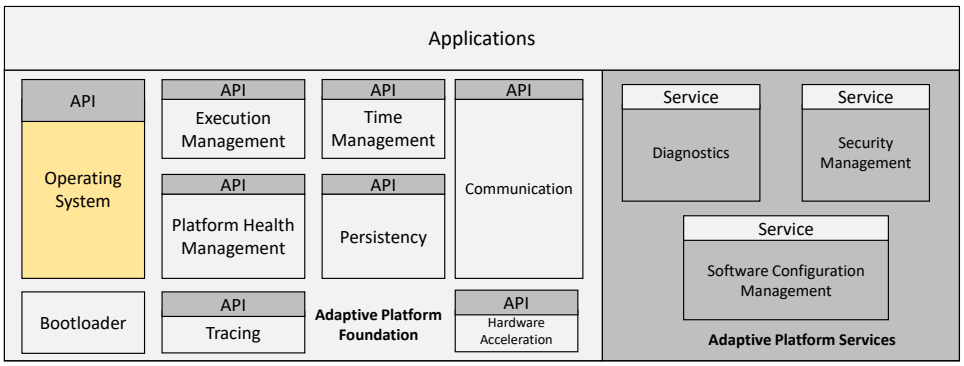


Figure 2.2: Adaptive AUTOSAR architecture.

2.2.3 ROS2 vs Adaptive AUTOSAR Adaptive

The comparison between ROS2 and Adaptive AUTOSAR unveils distinct perspectives on developing software architectures for autonomous vehicles, underlining their strengths and limitations. Adaptive AUTOSAR, with its foundation in the automotive industry, is meticulously designed to meet the stringent requirements of vehicle systems, emphasizing standardization, security, and interoperability. Its comprehensive framework encompasses a wide array of APIs and services tailored to automotive applications, ensuring a high degree of reliability and safety. The platform's structured approach, coupled with industry-wide support, facilitates a unified development process, albeit with a complexity and cost that may pose challenges for newcomers or smaller projects.

On the other hand, ROS2 represents a versatile, open-source alternative that extends beyond its robotics origins to address automotive needs. Its adaptability, supported by a vibrant community and a wealth of readily available packages, allows for rapid development and innovation. ROS2's service-oriented architecture and use of DDS for middleware communication enhance its flexibility and scalability, crucial for the diverse and dynamic demands of autonomous driving applications. However, it lacks the automotive-specific standardization and certification found in Adaptive AUTOSAR, which may necessitate additional effort to achieve compliance with automotive safety and security standards.

The distinctions between ROS2 and Adaptive AUTOSAR are primarily influenced by their respective focuses and underlying standards, illustrating their unique adaptability to different facets of software development within automotive and robotics sectors. Adaptive AUTOSAR stands out for its robust standardization, offering detailed development process standards, interoperability, and consistent specifications crucial for the automotive industry, particularly emphasizing security requirements. In contrast, ROS2, while not as prescriptive in services, APIs, or process standards, champions flexibility through its support for customizable packages and settings, allowing it to meet specific automotive functionalities when adjusted accordingly. Originally designed with robotics applications in mind, ROS2 extends its utility through community contributions, enhancing its adaptability across various industries, unlike Adaptive AUTOSAR, which is specifically crafted to address the evolving service-oriented architectures in the automotive sector. This focus ensures Adaptive AUTOSAR's alignment with automotive-specific standards such as Unified Diagnostic Services (UDS), an area where ROS2 does not inherently offer compliance. Consequently, while both platforms find their relevance in automotive contexts, Adaptive AUTOSAR offers a more focused and standardized framework tailored for automotive software architecture development, whereas ROS2 provides a more versatile foundation, adaptable to a broad range of applications through community-driven enhancements [59].

The comparison further illuminates the operational and developmental paradigms of both platforms, noting an industry trend towards adopting these technologies. Adaptive AUTOSAR's license-based model provides a well-documented, standardized development environment, significantly reducing fragmentation and enhancing compatibility across the automotive sector. This approach has been utilized by multiple OEMs in the industry, evidencing its reliability and effectiveness in meeting automotive standards as in [104] [91] [115]. On the other hand, ROS2's open-source model has gained traction for promoting accessibility and collaborative development, which encourages innovation through community contributions. Its adoption is trending among various companies due to the flexibility it offers for rapid prototyping and deployment. However, it necessitates a concerted effort to align solutions with the stringent safety and security standards of automotive applications. The increasing use of ROS2 in the industry, backed by references from leading automotive companies and tech innovators, underscores the growing recognition of its benefits for automotive development, including its adaptability, scalability, and the fostering of an innovative ecosystem that accelerates technological advancements in autonomous driving systems [105] [59].

ROS2's popularity in the automotive and autonomous application domains is largely due to its comprehensive support for real-time and safety-critical requirements, which are paramount in these fields. Unlike its predecessor, ROS1, which fell short in meeting the stringent real-time and safety standards necessary for automotive applications, ROS2 bridges this gap by offering enhanced service-oriented communication and compatibility with crucial safety standards, such as ISO26262 and the Automotive Safety Integrity Level (ASIL). This evolution makes ROS2 an ideal choice for developing ADAS and connected vehicle technologies, which rely heavily on robust and flexible service-oriented communication frameworks. Furthermore, ROS2's wide adoption in scientific and research environments,

due to its capabilities for software encapsulation and efficient communication, underscores its versatility beyond the E/E automotive architectures, extending into industrial applications. This broad applicability, coupled with certifications that affirm its suitability for automotive use, underscores ROS2's critical role in advancing automotive and industrial applications. **Considering the industrial nature of this thesis, our focus will be on exploring the potential of ROS2 for real-time applications and investigating ways to enhance its capabilities, reflecting its significance in both automotive and broader industrial contexts**.

2.3 Timing Models for Autonomous Applications

Timing models play a pivotal role in ensuring autonomus applications meet all functional requirements and prevent failures, especially in safety-critical systems where demonstrating the correctness of implementation is indispensable. The correctness of the implementation of such applications depends not only on the computed results but also on the point in time when the results are available. Timing requirements are used to specify critical execution paths, maximum durations, and functional constraints. Adhering to these requirements is fundamental for meeting the ISO26262 [122] standard, which is essential for the certification of safety-critical automotive applications. To improve the determinism in the execution of ADAS applications, [85] proposed the adoption of a deterministic reactive programming model, specifically utilizing the concept of reactors within the Adaptive AUTOSAR framework. This approach addresses the inherent nondeterminism challenges by coordinating software components under a discrete-event semantics model. By leveraging the DEAR (Discrete Events for AUTOSAR) framework, the method attaches tags to communications between distributed components, ensuring ordered processing and deterministic interactions. This methodology not only mitigates nondeterminism but also aligns with the safety and reliability requirements critical for ADAS applications, offering a structured solution to enhance testing, maintenance, and overall system robustness.

ROS2 is a widely adopted framework for robotic applications as with its available components, it facilitates development, prototyping, and deployment across diverse platforms. However, a notable drawback lies in the absence of real-time guarantees, challenging the assurance of safety for ROS-based robotic applications concerning timed properties, particularly when it is used for safety-critical applications such as autonomous driving applications. Formal modeling and verification of real-time behaviors within a ROS2 application, considering DDS and ROS2 Executor requires more dynamic and new approaches due to the difficulty associated with these mechanisms.

Efforts have been made to create methods for analyzing and optimizing timing in ROS2-based applications, e.g., [31] [127] [33] [5] [21]. These methods presuppose well-defined application *models*, meaning that the execution times and precedence relations among ROS2 functionalities are known and defined. In many scenarios, the application development process follows the separation of concerns, and only during system integration, the applications are composed. Moreover, during system integration, acquiring comprehensive details, par-

ticularly at the callback level, proves challenging, primarily due to confidentiality concerns.

In conjunction with model-based techniques, *tracing* ROS2-based applications have gained also attention. Within this context, *ros2_tracing* introduces a framework based on the Linux Trace Toolkit:next generation (LTTng) [17]. It has tracepoints in ROS2 functions to identify callbacks and topics and it also enables collecting runtime execution information. *Autoware_Perf* [80] has been developed for tracing and performance analysis in ROS2 applications. The frameworks extend the capabilities of tools like ros_tracing, allowing for more detailed analysis and visualization of message flow across distributed ROS2 systems. This development is particularly beneficial for complex applications like autonomous vehicles, where understanding message flow and system performance is crucial. *CARET* [76] add more tracepoints and use trace data to measure the response time of a callback, the communication latency between a pair of callbacks, and the end-to-end latency of a callback chain. However, this advancement underscores the notable gap in ROS2 diagnostics: the lack of a tool capable of treating ROS systems as black boxes and extracting timing information without necessitating any modifications. Such tools are very crucial for non-intrusive timing analysis, particularly for system integrators working at the binary level without access to source code. A tool that can analyze ROS2 systems as black boxes and extract timing information without requiring tracepoints or source code modifications would be invaluable. This would empower system integrators to efficiently diagnose and optimize ROS2 systems, offering insights into system behavior without the complexity of direct code intervention.

2.4 Real-Time Performance of ROS2 Applications

Many publications considered studying the real-time behavior of ROS-based time-sensitive applications. The primary research focus in the context of real-time ROS has been to control the latency and introduce determinism. Several investigations have been done to visualize and profile the latency of ROS stacks [89] [75]. Response time analysis techniques have been developed to bound the Worst-Case Response Time (WCRT) of ROS-based systems [128]. Timing model extraction and latency analysis have been also proposed to point out the bottlenecks affecting the chain's latency and automatically manage it [23].

When it comes to schedulability, several approaches have been introduced to improve the determinism and schedulability of ROS applications. To overcome the CFS scheduler problems, a priority-based scheduling mechanism has been introduced to improve the predictability and the end-to-end latency of the critical chains [34]. Further, to overcome the executor scheduling problem, real-time executors have been adopted to achieve deterministic execution [124] [138]. To impose control on the operating system scheduler and the hardware, a composable hardware-software architecture was proposed to manage the worst-case execution and response time for critical applications [39].

Servers-based schedulers have been adopted for real-time applications as they allow applications to run in their own resource space. with a pre-determined finite effect, isolation, and control on the resources and other running applications [101] [125]. They also showed optimal predictable and analyzable performance when adapted for the execution of real-

time applications in a virtualized environment[1]. To tackle the scheduling challenges in ROS applications, several works have evaluated the efficacy of using the constant bandwidth reservation scheduler in comparison to the CFS Linux scheduler that is utilized by the native ROS schedulers [31]. These works show substantial improvement in the determinism and the end-to-end latency over the default ROS scheduler. However, to the best of our knowledge, none of the conducted research has considered using static-based time-partitioned reservation stations. Operating systems such as QNX and PikeOS have introduced the idea of time-partitioning to provide guarantees on the execution time of chain-based applications however they did not provide a methodology to tackle jitter variations [38] [70].

2.5 Methods to control timing variations

Several works in the control systems community study the impact of software and network timings on control performance. JITTERBUG [82] and JITTERTIME [32] are tools to estimate control performance in non-ideal timing conditions, e.g., jitters, execution overruns, dropped control samples, random sampling, and so on. It is widely accepted that the performance of a control application degrades with increasing sensing-to-actuation delay jitters. Hence, in the real-time systems community, mechanisms have been proposed to reduce jitters.

A popular mechanism for this purpose is the implementation of the logical execution time (LET) concept [60], which states that a task reads its inputs at the beginning of the period and produces the output at the end of the period, i.e., the response time of a task is exactly equal to one period. Towards an LET implementation, Biondi et al. in [19] have proposed to schedule high-priority tasks at the beginning of the period to copy data (i) from the local memory of the producer task to the global memory and (ii) from the global memory to the local memory of the consumer task. Similarly, Pazzaglia et al. in [95, 96] have proposed to use DMA for time-efficient data transfers between tasks in different processing cores following LET paradigm. Further, in [43, 51, 52], a system-level LET concept is introduced for distributed applications in which an interconnect task is considered for exchanging data between two processors in a non-negligible yet constant time. However, existing LET implementations cannot be applied trivially to ROS2 chains. Also, our proposed two-server scheduling mechanism allows a more *flexible* control over the end-to-end timings of such chains, in particular, can produce multiple short latency bands.

Besides LET implementations, Buttazzo and Cervin have proposed three more jitter control mechanisms in [30]: task splitting, advancing deadlines, and non-preemption. The latter two mechanisms are not effective when the tasks have large variations in the execution time. The first method proposes to split a task into several parts and schedule the first sub-task at the beginning of the period and the last sub-task at the end of the period, which is similar to an LET implementation while also requiring to modify a task implementation. However, our proposed mechanism *does not require to modify or recompile the application source code*.

In recent years, there have been significant efforts to study the real-time behavior of ROS2 applications. Casini et al. in [31] provided the details on how ROS2 executors schedule different signal handlers, i.e., timer, subscriber, server and client callbacks. In the same work, the authors pointed out that ROS2 applications might contain chains of such callbacks, and introduced a compositional performance analysis (CPA) technique to bound the end-to-end latency of these chains when each ROS2 executor uses a constant-bandwidth server (CBS) to schedule callbacks. Later, Tang et al. proposed a new technique to improve the precision of the worst-case end-to-end latency analysis [127] for the same system model. Blaß et al. in [21] improved these results by (i) further reducing the pessimism in the callback activation model, (ii) considering a cumulative processor demand function for each callback, and (iii) making the timing analysis aware of the starvation freedom offered by ROS2 executors.

Besides the timing analysis techniques, previous works studied different scheduling policies for ROS2 applications. To meet the worst-case end-to-end timing requirements, heterogeneous laxity-based DAG (directed acyclic graph) scheduling coupled with static priority assignment for ROS2 nodes was studied in [117]. Yang and Azumi evaluated a callback-group-level ROS2 executor in [139] where a callback is assigned to an executor based on its real-time requirement, i.e., whether it is time-critical or best-effort. PiCAS [33] minimizes the end-to-end latency of chains by statically prioritizing the execution of their callbacks. The authors propose methods to (i) assign priorities to callbacks, (ii) allocate nodes to executors, (iii) map executors to processing cores, and (iv) analyze worst-case end-to-end latency. While the aforementioned works considered static configuration of the scheduling parameters, Blaß et al. in [22] suggested to dynamically refine the budgets of CBS used by ROS2 executors based on online timing measurements to meet the worst-case end-to-end latency requirements. In the same vein, Al Arafat et al. proposed deadline-driven dynamic priority assignment to ROS2 computation chains and developed a technique to analyze their worst-case end-to-end latency [5]. Most of these works mention that ROS2 computation chains often implement sense-compute-actuate control logic. However, all of them focus on the worst-case end-to-end latency of ROS2 computation chains and *none of them consider to minimize the jitters*, which is the main focus of our work.

2.6 Communication in Disributed Architectures

In the context of autonomous vehicles, the communication paradigms are shifting from traditional, static, signal-based communication to more dynamic and flexible service-oriented architectures (SOAs). This change is driven by the requirements of Advanced Driver Assistance Systems (ADAS) functions and autonomous vehicles (AVs). The traditional approach with static E/E architectures and signal-based communication is reaching its limits in terms of efficiency, bandwidth, and flexibility. As a result, more powerful bus networks, such as Ethernet, are being introduced to provide increased bandwidth and meet the requirements of future vehicles. The introduction of SOAs enables dynamic communication through the provision of services in a distributed system. Service-consuming entities can subscribe dynamically to these services, allowing for more flexible and dynamic system interaction. This is a departure from the static C-Matrix and signal-based communication, which lack flexibility. With SOAs, communication paths are dynamically established during runtime, allowing for more adaptability in the face of changing subsystems and links. Furthermore, the deployment of software applications in a SOA is no longer statically coupled with the underlying hardware. This dynamic SW architecture requires interoperability between the service sending and receiving entities, emphasizing the need for standardization and harmonization of higher architecture abstraction layers and interfaces. the communication paradigms in the context of autonomous vehicles are changing from static, signal-based communication to dynamic and flexible service-oriented architectures, driven by the requirements of ADAS functions and AVs [59].

Autonomous driving technologies have significantly advanced, transitioning from centralized processing to distributed architectures. This shift addresses the increasing computational demands and the need for resilience and scalability in autonomous vehicles. Accelerated SoCs play a pivotal role in this evolution, offering a compact, energy-efficient solution for processing complex algorithms and sensor data in realtime. Meanwhile, communication protocols ensure reliable and secure data exchange within the vehicle's systems and with external entities [141]. Accelerated SoCs integrate various processing units, such as CPUs, GPUs, and custom accelerators, on a single chip, optimizing the performance and power efficiency of AVs. These SoCs support the execution of machine learning algorithms, sensor fusion processes, and control systems essential for autonomous applications. As the automotive industry integrates more powerful Systems on Chip (SoCs) with advanced acceleration capabilities, the demand for communication protocols that can handle significantly increased data throughput becomes paramount. These enhanced SoCs enable more complex computations, sophisticated sensor data processing, and real-time execution of machine learning algorithms, necessitating communication infrastructures that surpass the bandwidth capabilities of traditional protocols such as CAN, LIN, and FlexRay. The limitations of these conventional protocols in supporting high-bandwidth data streams drive the need for more advanced communication solutions.

Automotive Ethernet emerges as a pivotal technology in this context, offering bandwidths that can extend up to 10 Gbps and beyond, compared to the maximum 10 Mbps provided by FlexRay, the fastest among traditional protocols. This quantum leap in data handling capacity is essential for supporting the voluminous and continuous data flow from high-resolution cameras, LiDAR, radar, and other sensory equipment integral to AV operations. Furthermore, the shift towards Automotive Ethernet and similar high-bandwidth protocols is complemented by the adoption of Time-Sensitive Networking (TSN) standards, which ensure deterministic data transmission, critical for the real-time decision-making processes in autonomous driving.

RDMA technology has been widely adopted in data centers over Ethernet networks (i.e., RoCE) to tackle the challenges in data-intensive applications, e.g., big data analytics and online gaming [55] [16] [61]. Several works have evaluated the performance of RoCE in comparison to default Ethernet communication stacks with TCP/IP, e.g., [134, 15]. These works clearly show that RoCE reduces the CPU load (related to network communication) significantly compared to TCP and, at the same time, offers lower communication latency. RoCE communication has been traditionally established over RDMA-capable NICs, i.e., the main focus has been to build such hardware, e.g., [140]. Although Soft-RoCE has been developed to enable hardware-independent RDMA communication, it has not received much attention for industrial use. Considering that Soft-RoCE offers more flexibility and similar performance with respect to hardware-based RoCE [130], we advocate its use in the automotive domain.

The primary research focus in the context of RoCE has been to control packet congestion in the network switches that can lead to packet losses. To avoid packet loss, the default technique in RDMA is to use Go-Back-N protocol where only the packets following and including the lost packet are re-sent [136]. However, this protocol suffers from lower throughput and, hence, is improved to IRN where only the lost packet is re-transmitted, thereby allowing out-of-order packet delivery [88, 86]. Further, to avoid buffer overflows in network switches and NICs, priority flow control (PFC) mechanisms have been proposed [7], where sender devices are notified hop-by-hop to pause/resume sending packets based on the states of the buffers in the receiver devices. PFC is typically implemented at the port level which might lead to poor performance, e.g., unfairness and victim flow, with respect to individual data flows [143, 56]. For flow-level congestion control, quantized congestion notification (QCN) is proposed where a data flow is addressed by its identifier in addition to the MAC address [6]. Data center QCN (DCQCN) is also proposed to ensure fairness in bandwidth allocation [143]. Different algorithms are proposed for DCQCN, e.g., [135]. Note that the problem setting in these works is very different from ours. In a typical automotive setting with AD/ADAS applications, the static analysis shall be performed to estimate buffer size so that time-critical packets are not lost [18]. Our goal is to minimize the latency of high-priority data packets and, at the same time, it is acceptable to be unfair to best-effort packets. Note that these works assume that packets are sent based on the default RDMA protocol unless there is congestion, however, we want to prioritize packets even when there is no congestion and the sending side already knows the priority of the packets. These existing techniques cannot be directly applied to solve the problem at hand and they are mostly

implemented on hardware. [110] is the closest to our work where a hardware priority queue is proposed in the Queue Pair. However, our communication architecture is more flexible as we can easily add different scheduling mechanisms and QoS metrics to the software.

Full-duplex switched Ethernet [2] and BroadR-Reach [26] standards have enabled the use of Ethernet in safety-critical automotive systems. Express traffic or high-priority traffic is recommended in IEEE 802.3br [63], an amendment to Ethernet protocol. Further, the Time-Sensitive Networking (TSN) Task Group has proposed several amendments (e.g., time synchronization [65], bandwidth reservation [64], and queueing and forward of time-sensitive frames [66]) to the Ethernet protocol to support time-sensitive communication. However, to the best of our knowledge, none of the implemented TSN protocol stack supports RDMA communication. This work is the first step to carry out RDMA communication for time-sensitive automotive applications. Our proposed communication stack is developed keeping in mind a possible future extension to integrate TSN with RDMA.

In the automotive domain, to the best of our knowledge, RDMA technology has only been used in Mobile Data lake (MDLake) to collect vehicle data from all loggers [14]. Again, hardware-enabled RoCE_v2 is used for high-bandwidth communication.However, we propose to use Soft-RoCE for inter-SoC communication in distributed AD/ADAS applications. We have performed real-world experiments to evaluate the performance of our proposed communication stack involving Soft-RoCE in terms of latency and determinism which are critical requirements of AD/ADAS applications.

2.7 Conclusion

The chapter delves into the automotive industry's shift from signal-oriented to service-oriented architectures, driven by the complex requirements of autonomous systems and their functionalities. This transition is supported by the adoption of Adaptive AUTOSAR and ROS2, which offer sophisticated frameworks for managing dynamic and complex electronic and electrical architectures. Adaptive AUTOSAR facilitates the deployment of flexible, high-performance automotive applications via a service-oriented approach, whereas ROS2 is conducive to the development of modular and scalable autonomous applications. The narrative further discusses timing models for autonomous systems, highlighting the necessity of deterministic programming and discrete-event semantics to ensure safety and reliability standards. Additionally, it addresses the real-time performance challenges and solutions for ROS2 applications, emphasizing the importance of determinism in system execution to guarantee reliability and safety in autonomous operations.

Building on this literature review, the objective was to assimilate the strengths of existing methodologies and frameworks, especially in understanding middleware timing models and managing real-time performance across distributed networks. This endeavor aimed to identify and bridge gaps in the automotive domain, focusing on the nuanced requirements of service-oriented architectures including Adaptive AUTOSAR and ROS2. Investigating advanced timing models and the real-time behavior of ROS2 applications was pivotal to securing deterministic execution crucial for the safety and certification of autonomous vehicles. The shift towards distributed autonomous architectures, facilitated by cutting-edge System on Chips and communication protocols like Automotive Ethernet and RDMA over Converged Ethernet, underscored the need for high-performance computing and real-time communication, essential for the next wave of autonomous vehicles. This comprehensive review set the stage for enhancing understanding and capability to model, analyze, and optimize the real-time performance of middleware, ensuring the development of robust, reliable, and safe automotive systems amidst increasing complexity and connectivity.

The thesis targets critical gaps in understanding and optimizing the real-time behavior of middleware in autonomous systems, focusing on extracting and modeling timing models to enhance performance, safety, and reliability. It proposes to validate novel methodologies addressing challenges related to managing latency and jitter in distributed networks, particularly for applications utilizing the Data Distribution Service communication frameworks. By exploring advanced techniques for timing analysis, system modeling, and real-time control strategies, including kernel bypassing and CPU offloading, the research aims to significantly improve the deterministic performance of next-generation autonomous applications. This focused effort endeavors to contribute valuable insights and practical solutions to the automotive industry, facilitating the development of safer, more reliable, and efficient autonomous vehicles.

Chapter 3

System Model Extraction

Autonomous Autos developed using ROS2 are expected to have enormous capabilities and driving functionalities. We can see some demos from Autoware Auto [48] and [105] where the car has full localization functionality against the environmental variable using LIDARs, cameras, and a global position system (GPS). As shown in Figure 3.1, these cars are also able to build a complete real-time 3-dimensional surrounded view to be able to detect other road participants. They are also able to classify these participants, their interference, velocities, and their predicted path of motion. Additionally, they can conduct motion-time maneuvers such as stop, start, and yield maneuvers based on the road semantics. All related autonomous-driving applications developed on ROS2, mainly consist of ROS2 *nodes*, each dedicated to a specific function such as localization or object detection. Communication among nodes is done through through *topics*. For example, a ROS-based lidar-related functionality can look like a connected chain of nodes and topics. The chain is activated upon receiving the new data from the LIDAR. When the new data from the LIDAR arrives, the nodes subscribed to this data are activated, triggering a function known as the *subscriber callback* that is responsible for performing some computations and filtering on the received data. For instance, this node can be a point cloud filter or sensor fusion node that applies a threshold filter on the quality and intensity of the received data from the LIDAR. Following certain computations, the node may publish data on the next topic which itself can trigger the execution of the following periodic nodes in the chain.

Fulfilling the timing requirements of such chains during the design phase is a must for safety considerations. Functionally, control loops of the chains must execute within approximately 10 milliseconds while maintaining microsecond-level jitter to ensure system stability and responsiveness. This rapid processing is essential for the autonomous nature of the chain to enable object detection and subsequent responses within a stringent 100-millisecond window, a timeframe that significantly surpasses human capabilities, enhancing safety and performance. From an adaptability standpoint, the software architecture must be agile, and capable of swiftly adjusting to different vehicle models and traffic scenarios. It necessitates the ability to perform real-time transitions between driving maneuvers, a process contingent on the quick evaluation of the Domain of Operation (DoO) changes, which are often unpredictable and varied. In terms of non-functional requirements, updates to the

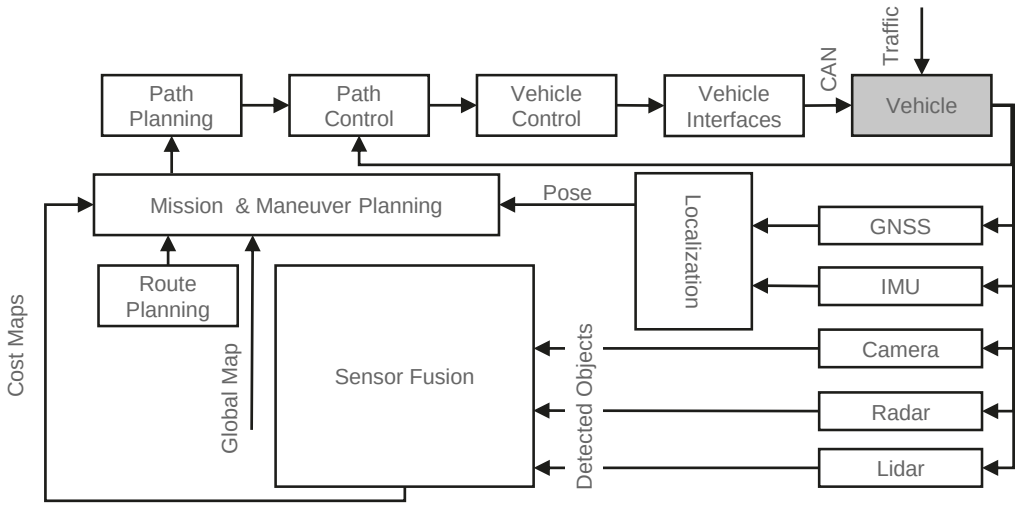


Figure 3.1: Function-plan of self-driving car architecture [105]

system, even at the modular level, must be executed in a manner that does not degrade the timing performance of the application. This ensures that enhancements or repairs to the system uphold the stringent timing standards essential for the reliable operation of ADAS, maintaining the system's usability and adaptability attributes without compromising performance [105].

The stringent timing requirements necessitate a comprehensive understanding of the system's timing architecture to ensure predictable performance. Through this chapter, we focus on extracting timing models for services and functionalities of applications that have been developed using ROS2. In ROS2 semantics, we consider extracting and modeling the architecture for various types of callbacks, including timer, subscriber, service, and client callbacks to study their timing properties. The main motivation behind our goal is to understand and model the timing architecture of the deployed application in a formal way that can later open the door for further optimizations and performance verification.

This chapter is organized as follows: It starts with an exploration of the timing properties of ROS2 applications, providing a foundation for understanding their temporal dynamics. The subsequent section introduces application tracing using eBPF, a technique pivotal for capturing the runtime behavior of these applications. We then progress to a detailed discussion on the development of a trace-based timing model extractor, which is dissected into its framework design, integration with ROS2, and utilization of eBPF with OS scheduler tracepoints. This section further extends into the intricacies of timing model extraction, execution time measurement, end-to-end latency assessment, and DAG generation for ROS2-based applications. The chapter advances with an evaluation of the framework, leading to insights on the integration and optimization of applications, and culminates with a concluding synthesis of the presented concepts.

3.1 Timing properties of ROS2 applications

In real-time systems literature, a task commonly refers to a schedulable entity in the system. In ROS2, callbacks are synonymous with tasks as they are scheduled by ROS2 executors. Note that for each service/action, multiple topics are created during initialization and then the related functions on the server and client sides are triggered similar to subscriber callbacks. In software developed following ROS2 semantics, we can find timer, subscriber, service, and client callbacks. For example, in Autoware's Autonomous valet parking (AVP) example, The ROS chain starts from LIDAR point cloud data followed by filter, fusion, downsampler, and localization until the current pose (position and orientation of the car) is generated for planning.

Definition 3.1.1. ROS2 Chain (ch)

$ch\{n_c, n_\tau\}$ where n_c is defined as set of callbacks $\mathbb{CB}\{cb_1, cb_2, \cdots, cb_{n_c}\}$ and n_τ is defined as set of topics $\mathbb{TP}\{tp_1, tp_2, \cdots, tp_{n_\tau}\}$.

Similarly, In the context of this chapter, we consider the application as a set of chains where each chain is defined as per Definition 3.1.1 as a set of callbacks n_c and topics n_c. For each callback cb_k, we measure its *execution time* $et_{k,u}$ at each invocation $u \in \mathbb{N}$. where $et_{k,u}$ is defined as the duration for which a callback runs on the processor between its start and end. The timing relations between callbacks are defined based on the topics. If a callback cb_k, for example, writes data on a topic tp_m and a callback $cb_{k'}$ reads data from tp_m then cb_k and $cb_{k'}$ have a precedence relation, i.e., cb_k precedes $cb_{k'}$ in time. Using such execution information, we provide a technique in Section 3.3 that is not only able to estimate the worst-case/best-case/average execution time of each functionality of the application but is also able to define their precedence and dependency relations. Since in cyber-physical systems (where software/hardware components interact with the physical world) the computation chains run periodically, we also assume that computation chains execute every T where T is the period of the chain. As shown in Figure 3.2, for such periodic chains, we denoted the time interval between the dispatch of a chain and the end of its execution (i.e., when the output of the chain is available) as the end-to-end latency of the chain (L). When the end-to-end latency of a chain varies then it means that the chain has a jitter. If we have variation in the availability of the output between [L-J, L+J], then J is the jitter of the chain.

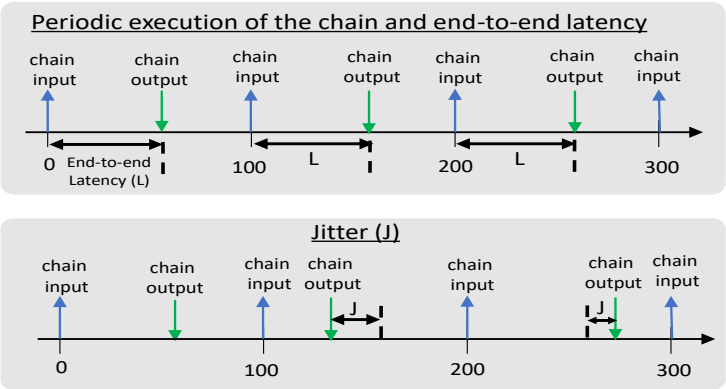


Figure 3.2: Latency and jitter definition for chains

3.2 Tracing

Tracing in operating systems is a critical diagnostic technique that involves monitoring and recording system activities and events, such as system calls, task scheduling, and hardware interactions. This process enables developers and system administrators to gain insights into the system's behavior and performance under various conditions. By providing a detailed record of events and their timings, tracing helps in identifying bottlenecks, understanding system latencies, and optimizing both application and kernel code. It is particularly vital for analyzing the timing properties of a system, ensuring that performance issues can be accurately diagnosed and resolved [36]. The importance of tracing extends to real-time systems as well, where meeting strict timing requirements is essential. In such environments, tracing allows for the verification of timing constraints and the detection of jitter and latency issues, thereby ensuring that the system adheres to its real-time specifications [98].

Within the Linux ecosystem, 'perf' and LTTng (Linux Trace Toolkit Next Generation) are two of the most prominent technologies used for tracing. LTTng is designed to provide low-overhead, high-precision tracing for both kernel and user-space applications. It utilizes tracepoints, kprobes, and uprobes to collect data, which is then logged efficiently to minimize performance impact. The design of LTTng, particularly its use of per-CPU buffers for storing trace data, ensures that the overhead remains low, making it suitable for production environments where maintaining performance is crucial. On the other hand, 'perf' leverages hardware performance counters and software events to offer a comprehensive analysis tool for performance monitoring and debugging. While 'perf' provides invaluable insights into system performance, including CPU utilization, cache behavior, and process execution, it can introduce a higher overhead compared to LTTng, especially when used for detailed sampling at high frequencies. Together, they offer a robust set of tools for diagnosing and enhancing Linux system performance, each with its strengths and use cases.

eBPF has emerged as a revolutionary technology in the Linux ecosystem, offering a highly flexible and powerful framework for monitoring and modifying system behavior in real time. Unlike traditional tracing tools, eBPF operates by allowing users to run sandboxed programs directly in the Linux kernel without changing kernel source code or loading kernel modules, providing unparalleled insights into system performance and security monitoring.

This mechanism enables a wide range of applications, from performance analysis and network traffic monitoring to security enforcement and system tracing. Compared to LTTng and perf, eBPF stands out for its versatility and depth of integration with the Linux kernel, allowing for more dynamic and customizable monitoring scenarios. While LTTng specializes in low-overhead tracing for both the kernel and user-space applications and perf excels in detailed performance analysis through hardware counters and software events, eBPF provides a broader toolkit that can intercept a wide array of system calls and events with minimal overhead. This makes eBPF a powerful complement to LTTng's efficient event logging and perf's performance-centric analysis, offering a more granular control and customization potential for system observability and troubleshooting.

One of the first works to utilize execution traces for model extraction is mentioned in [98], which laid the groundwork for a sophisticated analysis of real-time systems. This paper introduced the concept of leveraging a trace of scheduling events, including activations, dispatches, preemptions, deactivations, and thread exits, to unravel the intricacies of system behavior. By doing so, it allows for the construction of detailed timing models that reflect the real-time behavior of individual threads, ranging from non-self-suspending periodic tasks with jitter and offset to segmented and floating self-suspending task models, as well as request-bound functions. The outcome is a richly detailed system model that integrates a set of timing models for each thread, alongside essential system parameters, offering unprecedented insights into system performance. While [98] sets the stage for model extraction, our current work expands on this foundation, applying the approach to more sophisticated applications and architectures. We build upon the initial methodology to explore and analyze the behaviors of complex systems, pushing the boundaries of what can be understood from execution traces.

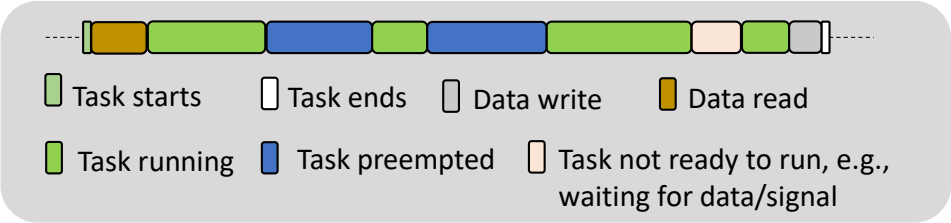


Figure 3.3: Task Execution Model

3.2.1 Application Tracing using eBPF

Execution tracing, spanning both userspace and the kernel, entails the meticulous observation and logging of events and activities unfolding during the execution of a program or operating system. This practice proves indispensable for debugging, performance analysis, and gaining insights into the intricate workings of software systems. In userspace, the process involves monitoring the operations of applications and processes operating beyond the kernel's realm. Developers deploy diverse tools and methodologies to trace the execution flow, pinpoint bottlenecks, and scrutinize the performance of running programs. Moreover, the enhancement of this tracing methodology involves strategically embedding tracepoints within the software platform. These tracepoints serve as markers designed to capture essential events as shown in Figure 3.3, such as task start and end, data read and write operations, as well as various scheduler events like task switches and wakeups. By incorporating these tracepoints, developers can attain a more comprehensive understanding of the software's behavior, facilitating effective debugging, performance optimization, and analysis of intricate system interactions. Conversely, kernel execution tracing focuses on monitoring and logging events transpiring within the operating system's core. This comprehensive tracing is achieved through the strategic placement of tracepoints or probes, acting as markers embedded within the application or kernel. These markers, when affixed to a program, facilitate the identification of predetermined behaviors during execution, providing a means to delve into the intricacies of the running program's behavior.

eBPF (extended Berkeley Packet Filter) stands out as a powerful and flexible technology that has gained prominence in the domain of systems and performance tracing. It is a a Linux kernel technology facilitating the execution of in-kernel virtual machine programs (eBPF programs) within the kernel context [107]. It offers a more dynamic and programmable approach for tracing and monitoring activities within the application and the kernel layers. One of the key advantages of eBPF is its ability to inject custom tracing code into the kernel without requiring modifications to the kernel code. This dynamic programmability allows for real-time analysis and modification of system behavior without the need for kernel recompilation which is crucial for its adoption in the industry.

As shown in Figure 3.4, eBPF programs are loaded as bytecode into the kernel by leveraging the bpf() system call. However, before being loaded eBPF programs undergo a validation process through the kernel static analyzer to assess their safety and adherence to predefined criteria, ensuring their integrity and security for the kernel environment. In prac-

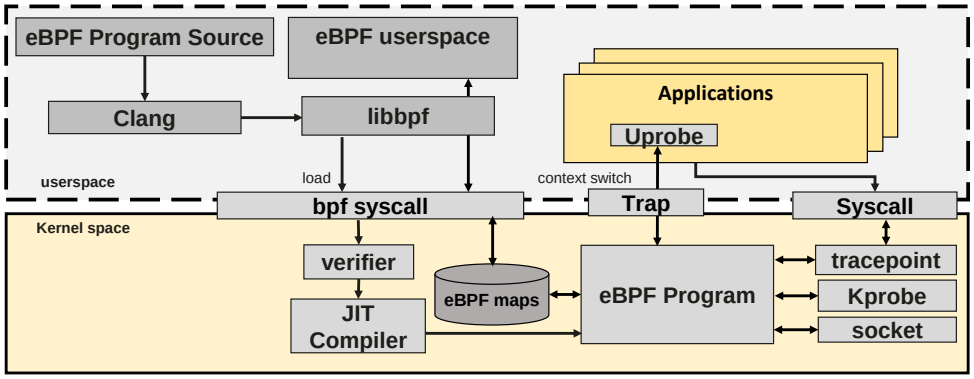


Figure 3.4: eBPF workflow from [142]

tice, a tool like bcc is used to write efficient ebpf programs in restricted C which are then converted to bytecode using LLVM Clang compiler [78]. eBPF programs can be hooked when various kernel and userspace functions are invoked, thereby enabling comprehensive system monitoring. Through the utilization of the kernel probes (kprobes) and the userspace probes (uprobes), eBPF programs can capture information regarding system calls, function entry and exit points, and network events. This collected data can not only be used for monitoring but can also be efficiently shared with userspace applications through the maps infrastructure of the eBPF.

The adoption of eBPF over LTTng for tracing in ROS2 applications is underpinned by several compelling advantages, as discussed in the previous section. The primary reasons to use eBPF instead of LTTng as in [17] is that utilizing eBPF allows the attachment of probes to the entry and exit points of ROS2 functions, enabling the retrieval of arguments and return values without necessitating modifications to the standard libraries. In contrast to LTTng, there is no requirement for direct instrumentation and recompilation of ROS2 libraries. While [76] proposes using `LD_PRELOAD()`to redirect function calls to a tracing library instead of ROS2 libraries, subsequently calling the original ROS2 functions, this approach involves multiple lines of code to update addresses for finding the original functions. This introduces significant tracing overhead without offering additional capabilities and is not feasible when a function is defined in the header. Additionally, eBPF exhibits more efficient trace filtering [120], a feature leveraged to capture kernel events specific to ROS2 nodes exclusively. Additionally, it facilitates the filtering of events related to one or more ROS2 nodes, enhancing the efficiency of debugging processes. Besides, eBPF is deemed safer and offers increased programmability [107]. This characteristic opens avenues for future exploration, such as dynamic scheduling of ROS2 nodes to enhance timing performance and the secure execution of ROS2 callbacks.

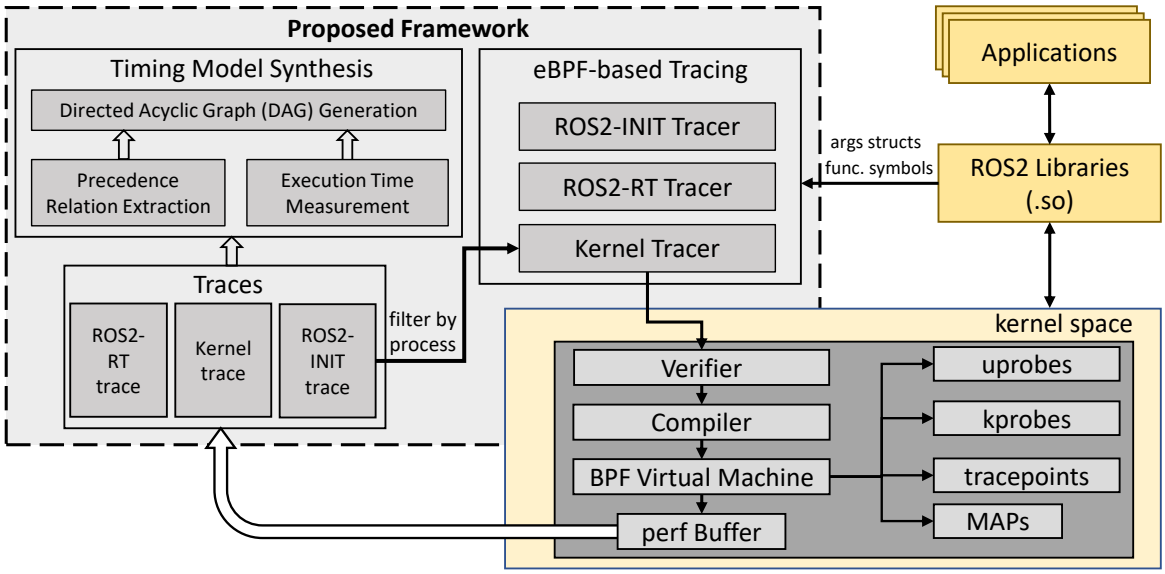


Figure 3.5: Proposed trace-enabled timing model synthesis framework.

3.3 Trace-based Timing Model Extractor

3.3.1 Framework Design

In this section, we will introduce our eBPF-based time model extractor framework, as depicted in Fig. 3.5, which serves to bridge the gap between tracing and timing analysis. By employing eBPF, our framework is designed to offer three distinct tracers. The first, referred to as the Initial tracer (**ROS2-INIT**), is tasked with collecting events that occur during the initialization of ROS2 nodes. This tracer is adept at relating nodes to their respective threads/processes, which aids in filtering events on a per-node basis, consequently reducing the memory footprint of traces. Following this, we have the Run-time tracer (**ROS2-RT**), which is responsible for monitoring the start and end of callbacks, in addition to tracking data read from and written to DDS topics. Lastly, the **Kernel Tracer** is designed to gather scheduler events within the operating system, such as `sched_switch`, `sched_wakeup`, and `sched_process_fork` all of which are related to ROS2 callbacks. This collection is facilitated by the trace filtering capabilities of eBPF, thereby enhancing the effectiveness of our framework in capturing essential timing data.

As illustrated in Fig. 3.5, our proposed framework uses the collected traces to synthesize timing models of applications as *directed acyclic graphs* (DAGs) defined as per Definition 3.3.1 We model ROS2 callbacks as vertices (V) and DDS communication between them as edges (E). Further, we propose to model a *service* using n tasks if it is invoked by n different clients. Otherwise, DAG will show one vertex with n incoming edges and n outgoing edges, i.e., $n \times n$ chains passing through the vertex, which is a wrong interpretation. Also, we propose to model a synchronization of m data, e.g., for sensor fusion, using $m+1$ tasks, i.e., m tasks reading data each and then these n tasks feed data to another task that outputs

the result. Further, we *combine* ROS2 and scheduler events to measure the execution time of a callback for each invocation. We get statistical information from these measurements, e.g., measured worst-case, best-case, and average values. We then annotate the DAG with the obtained timing information.

Definition 3.3.1. Data Acyclic Graph (DAG)

DAG{V,E} $\Leftrightarrow$ where V is a set, called the vertices, and E is a set, called Edges. A graph is formed by vertices and by edges connecting pairs of vertices, where the vertices can be any kind of object that is connected in pairs by edges[53].

The DAG we generate can serve as an input for analysis and optimization by, e.g., [31, 127, 33, 5, 21]. Also, our framework is *not tied* to any particular application because we probe functions in ROS2 middleware and not in applications directly. We employ our framework to synthesize the timing model of a *real-world benchmark* implementing LIDAR-based localization in AVP. The goal of the framework is to synthesize timing models of any set of applications running over ROS2 without modifying or tracing the application itself.

3.3.2 eBPF with ROS2

Using eBPF, we have attached *uprobes* and *uretprobes* to several functions in different ROS2 layers as shown in Table 3.1 and Figure 3.6. Using a probe, we not only know when the probed function is entered (or exited) but also we can read function arguments (or return values). Considering that a middleware function is called by all ROS2 nodes (or callbacks), we distinguish between them using the arguments passed on to the function. We distinguish between callbacks (or topics) during a ROS2 function call using the arguments passed on to the function. Here, the main challenge is to understand complex ROS2 data structures and identify functions to probe so that we get the required events with enough information to reason about what is happening at the application layer. All information we extract by parsing the function arguments and return values is provided in Table 3.1. Table 3.1 lists the information we extract using each probe. Due to limited space, we do not elaborate on how we traverse through the function arguments. We highlight that we do not want to attach probes to applications because we want to develop a framework that can generically help in understanding the timing behavior of any set of applications implemented over ROS2.

In our developed framework, we have incorporated several significant aspects and concepts to enhance the functionality of our tracer. By utilizing the Linux *perf* tool, we are able to acquire call graphs for ROS2 nodes during the execution of various callback types, which aids in identifying ROS2 functions that are invoked and can be probed. Each event generated by our probes includes a timestamp for chronological ordering, a process ID (PID) to associate the event with a specific ROS2 node, and a probe name that denotes the type of information obtainable from the event. We attach *uprobes* $\{\mathbb{P}_2, \mathbb{P}_5, \mathbb{P}_9, \mathbb{P}_{12}\}$ and *uretprobes* $\{\mathbb{P}_4, \mathbb{P}_8, \mathbb{P}_{11}, \mathbb{P}_{15}\}$ to `execute_*` {`timer, subscription,service, client`} functions to capture the start and end times of callbacks. A technique is devised to extract the source timestamp (*srcTS*) of specific data in a topic by probing functions in the rmw layer

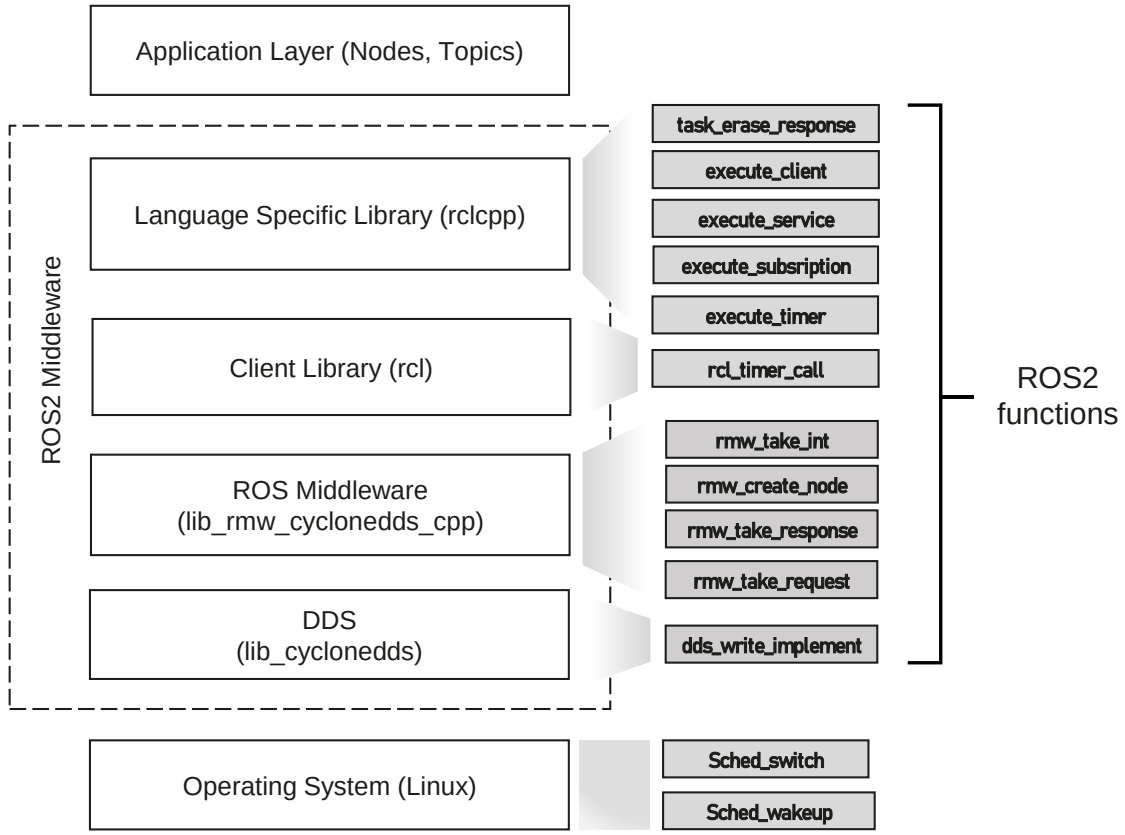


Figure 3.6: ROS2 architecture with our proposed tracepoints

as `rmw_take_*{int, request, response}` at both entry and exit points $\{\mathbb{P}_6, \mathbb{P}_{10}, \mathbb{P}_{13}\}$. This process involves storing the address of *srcTS* at the function entry in a BPF map and retrieving the value from the stored address at the function exit. In the context of ROS2, where a service can be invoked from multiple client nodes, events for $\mathbb{P}_{12}$, $\mathbb{P}_{13}$, and $\mathbb{P}_{15}$ occur for the response to a specific service request sent to each client node. To differentiate these events, we probe `take_type_erased_response()` at exit using $\mathbb{P}_{14}$ and examine its return value to determine if the client callback will be dispatched. Additionally, we observed a library named `message_filters` in AVP that provides APIs for synchronizing data arriving at a node from different topics, which is used for sensor fusion. To identify a subscriber callback used for data synchronization, we attach a probe $\mathbb{P}_7$, meaning the probed function runs every time data is read from the topic and requires synchronization.

Table 3.1: Inserted probes in ROS2 Foxy.

No.	ROS2 lib	Function	Params/Purpose
$\mathbb{P}_1$	cyclone_dds	`create_node`	shows the PID of the thread executing the callbacks
$\mathbb{P}_2$	rclcpp	`execute_timer`	notifies timer CB starts
$\mathbb{P}_3$	rcl	`rcl_timer_call`	shows timer CB ID
$\mathbb{P}_4$	rclcpp	`exec_timer`	notifies timer CB ends
$\mathbb{P}_5$	rclcpp	`exec_subscrip.`	notifies subscriber CB starts
$\mathbb{P}_6$	cyclone_dds	`rmw_take_int`	notifies a read event on a topic and shows subscriber CB ID, topic name and source timestamp of data
$\mathbb{P}_7$	message_ filters	`operator`	shows that a subscriber CB is used for data synchronization
$\mathbb{P}_8$	rclcpp	`exec_subscrip.`	notifies subscriber CB ends
$\mathbb{P}_9$	rclcpp	`exec_service`	notifies service CB starts
$\mathbb{P}_{10}$	cyclone_dds	`take_request`	notifies a service request received event and shows service CB ID, service name, and source timestamp of request
$\mathbb{P}_{11}$	rclcpp	`execute_service`	notifies service CB ends
$\mathbb{P}_{12}$	rclcpp	`execute_client`	notifies client CB starts
$\mathbb{P}_{13}$	cyclone_dds	`take_response`	notifies a service response received event and shows client CB ID, service name, and source timestamp of response
$\mathbb{P}_{14}$	rclcpp	`erase_response`	notifies if a client CB will be dispatched
$\mathbb{P}_{15}$	rclcpp	`execute_client`	notifies client CB ends
$\mathbb{P}_{16}$	cyclone_dds	`dds_write_impl`	notifies a write event on a topic and shows the topic name and source timestamp of data/request/response

3.3.3 eBPF with OS scheduler trace points

In our developed framework, we managed to attach trace points to different OS scheduler events, e.g. `sched_switch` and, `sched_wakeup`. `sched_switch` is an OS event that notifies when the scheduler gives a CPU to a new thread while `sched_wakeup` notifies when the scheduler switches execution from one thread to another. Nevertheless, we plan to extend our framework for trace-based debugging and optimization of ROS2 applications where other scheduler events also become crucial. From such an event, we can obtain the CPU where the switch is occurring, the process ID and the scheduling priority of both the previous and new threads, and The state of the previous thread at the time of the switching. These events are essential for measuring the exact execution time of a callback especially since it is easy to deduct preemption times when using them. Also, if a ROS2 node is pinned to a CPU or it has been assigned a real-time priority, we can get such information from `sched_switch` events. Using these events, the CPU affinity and scheduling priority of ROS2 nodes can be easily deducted. To reduce the memory footprint of the generated traced data, we added a filtering functionality based on the PIDs of ROS2 nodes. These PIDs are got using the probe $\mathbb{P}_1$ which is then shared with BPF maps of `sched_switch` event handler. This reduces the memory footprint by an order of three or more.

3.3.4 Tracing Challenges

Identifying functions to trace or probe within ROS2's complex architecture indeed poses a significant challenge. The modular and distributed nature of ROS2 results in a vast number of potential interaction points, each contributing to the system's overall behavior. Tools like perf [99] and flamegraph [46] offer powerful ways to visualize and understand these interactions through the generation of control flow graphs. The second challenge in tracing ROS2 applications using eBPF revolves around the intricate task of navigating data structures to accurately locate the correct offsets, which are essential for obtaining pointers to the pertinent parameters. This challenge is compounded by the sophisticated data models employed within ROS2, where entities such as messages, services, and actions utilize complex, nested data structures. The dynamic system and the polymorphism inherent in ROS2's design further complicate the extraction of reliable offsets, as the structure and memory layout of these objects can vary significantly. Pahole [93] is a powerful tool designed to explore and analyze the memory layout of C structures in binary files, leveraging DWARF debugging information to provide insights into how data structures are organized in memory. By revealing the detailed structure of data, including the size, alignment, and offset of each member, pahole provides understanding of the exact layout of the data structures their applications use.

```
1 struct dds_writer {
2     struct  dds_entity      m_entity;            /*    0    536 */
3     /* --- cacheline 8 boundary (256 bytes) was 24 bytes ago --- */
4     struct  dds_topic       m_topic;           /*    536     8 */
5     . . .
6 };
```

Listing 3.1: data structure layout using pahole

```
struct data {
    string  topic;
    .......
};
ret = bpf_probe_read(&pointer_to_dds_topic,
      sizeof(void*), (void *)PT_REGS_PARAM1(ctx) + 536);
ret = bpf_probe_read(&pointer_to_ddsi_sertopic,
      sizeof(void*), (void *)pointer_to_dds_topic + 536);
ret = bpf_probe_read(&buffer
      sizeof(void*), (void *)pointer_to_ddsi_sertopic + 24);
ret = bpf_probe_read(&data.topic, sizeof(data.topic), buff);
```

Listing 3.2: eBPF program for extracting topic names

Listing 3.1 and 3.2 showcase a compelling example of how we navigated through the intricacies of data structures to extract topic names, a critical component in modeling ROS applications. Through the meticulous analysis provided by the pahole tool in Listing 3.1, we determined that the `dds_topic` structure is nested within the `dds_writer` structure, precisely 536 bytes offset from its start. This information served as the cornerstone for our subsequent eBPF script, which leverages a series of `bpf_probe_read` operations to systematically traverse the memory structure as shown in Listing 3.2. This procedural memory traversal and the strategic use of safe memory reading techniques underscore our approach capabilities in enabling successful extraction of the correct address to the traced parameters.

Tracing parameters passed by reference to functions presents a challenge, due to its dynamic nature and the complex memory management practices involved. To effectively trace these parameters, attaching eBPF probes at the entry point of the target function becomes essential. This approach allows for the capture and reading of parameter addresses as they are passed into the function, providing a snapshot of their memory locations at the moment of invocation. By storing these addresses in a hash map, we establish a mechanism for tracking the lifespan and modifications of the referenced parameters throughout the function's execution. Upon the function's exit, another eBPF probe can then be utilized to read the values from the previously stored addresses, allowing for a comprehensive view of how the parameters were altered during execution. This method effectively tackles the complexity of tracing reference parameters by ensuring accurate monitoring from entry to exit.

To mitigate the memory footprint of kernel-level traces, a filtering approach is employed. Initially, the tracing mechanism focuses on capturing the initialization phase of ROS2 nodes, strategically pinpointing the moment these nodes are online. Upon the successful initialization of a node, its unique identifier process ID (PID) is extracted and preserved within a dedicated hash map, serving as a registry of active nodes. This registry then becomes the cornerstone for subsequent filtering processes, where only events directly associated with these registered PID are considered for logging kernel events. By adopting this selective logging strategy, we substantially reduce the volume of data captured during tracing, focusing exclusively on interactions involving the pre-identified nodes. Consequently, this approach not only diminishes the memory demands of tracing operations but also enhances the clarity and relevance of the collected data, facilitating efficient analysis of nodes' behavior.

3.4 Timing Model Extraction for ROS-2 Applications

In a ROS2 node, the single-threaded executor executes a callback from its initiation to completion before revisiting the ready queue of callbacks. Consequently, each event occurring between a *callback_start* and the subsequent *callback_end* associated with a specific ROS2 node, identified by its PID, supplies details about the execution of a unique callback (CB) instance. In the context of our work, we specifically denote this as a CB instance occurrence. Building upon this concept, we propose Alg. 2 to process information related to CB execution and extract both architectural and timing attributes.

The algorithm we implemented for tracing callback (CB) invocations within a ROS2 environment follows a structured approach, as outlined in the steps below. Initially, the process begins with the extraction of crucial information from a `CB_start` event, specifically the CB's type and its initiation time, with the type being determined based on the associated probe name. Following this, the identification of the CB's ID is achieved through analysis of either a `timer_call` event for timer-based callbacks or a `take` event for other types of callbacks. The algorithm then proceeds to ascertain the subscribed topic by examining a `take` event, applying different handling mechanisms for client and service callbacks. In the case of service callbacks, a critical step involves identifying the caller by establishing a connection between `dds_write` and `take` events. Furthermore, the topic to which data is published is identified from a `dds_write` event, incorporating special considerations for topics associated with service requests and responses. An important conditional aspect of the algorithm is the handling of client callbacks that are not dispatched, as signified by a `take_type_erased_response` event; in such instances, the callback instance's information is not recorded. Additionally, callbacks related to data synchronization for subscribers are distinctly marked through the detection of `sync_subscribe` events. The algorithm culminates in the capture of the CB instance's end time from a `CB_end` event, facilitating the computation of the execution time according to a predefined algorithm detailed in Alg. 3.

Following the extraction of timing attributes, input and output topics, ID, and type, the stored information at the end of a CB instance is added to *CBlist*, which serves as a list of callbacks. A new entry is created in *CBlist* only if none of the existing entries identifies the same CB as the current instance. Conversely, if a match is found, the algorithm records the execution time and updates the published topic list (if a new topic is encountered) in the corresponding entry. For all CBs except services, matching is based solely on the ID, while for services, both the ID and subscribed topic are used for matching. It is crucial to note that, in the case of a service CB in *CBlist*, the updated topic name serves to identify a service request by a particular caller. Subsequently, after updating *CBlist*, all information related to the CB instance is deleted.

3.4.1 Execution Time Measurement

The algorithm, detailed in Alg. 3, calculates the execution duration of a Callback (CB) instance in ROS2 by integrating both ROS2 and *sched_switch* events. Essential for this computation are the *start* and *end* times of the CB instance (derived from ROS2 events), and the *PID* of the ROS2 node, which helps identify the thread $\mathscr{T}$ executing the CB. Additionally, it uses all *sched_switch* events, labeled as *SchedEvents*. The algorithm operates by identifying and summing up the time lengths of all execution segments within the CB instance timeframe, found by sequentially examining the events in *SchedEvents*. The execution time starts with the *CB_start* event when thread $\mathscr{T}$ is running and concludes with the *CB_end* event. The transitions of thread $\mathscr{T}$ in the *sched_switch* events signal the ends and starts of execution segments: the end of a segment is marked if the previous thread in a *sched_switch* event is $\mathscr{T}$, and the start of a new segment is indicated if the next thread is $\mathscr{T}$.

3.4.2 End-to-end latency measurement

There have been a few works recently discussing the measurement of the end-to-end latency of ROS2 computation chains in different contexts. While [74] and [129] have proposed to directly modify the application code and record timestamps, [80] and [77] have added trace points in ROS2 middleware and introduced techniques to measure the end-to-end latency from collected traces. We follow the latter philosophy, however, we have not used any of the existing tools due to their limitations. On the one hand, [80] does not measure the end-to-end latency of a chain directly but estimates it using a convolution integral of the response times of the constituent callbacks. On the other hand, *CARET*, according to [77], cannot accurately measure the latency of a chain when there is more than one data item available in the receive buffer for a topic.

The operation of measuring end-to-end latency for all chain executions in a system involves sequences of steps designed to track the time taken from the initiation of a computation to data generation as in Algorithm 1. Initially, the process begins by identifying a "computation start" event generated by the first node in the chain. Subsequently, before the corresponding "computation end" event occurs, the process seeks out a "data write" event initiated by this node. If this "data write" event does not pertain to the chain's output data, the system records the data ID and sequence numberensuring that each piece of data is uniquely identifiable by its sequence number. The subsequent step involves locating a "data read" event that matches both the data ID and sequence number, carried out by the following node in the chain. The chain's progression is then marked by finding the next "computation start" event for this subsequent node, and the cycle repeats through steps two to four. Upon reaching a "data write" event for the chain's final output data, the timestamp is captured. The end-to-end latency for this particular execution is determined by subtracting the "computation start" event's timestamp of the first node from the "data write" event's timestamp of the chain's output data. This procedure is iterated through all chain executions to construct a comprehensive measurement of the end-to-end latency across the system.

Our measurement technique scans the collected traces and, for each execution of a chain ch_μ, it traverses the events related to each callback cb_k in ch. For a callback cb_k except the first one, we find the following events in order: *CB_start*, *data_read*, *data_write*, and *CB_end*. For the first callback cb_1, we do not get any *data_read* event because we assume that it is a timer callback that acquires the sensor data directly via device drivers. We move from one callback cb_k to the next cb_{k+1} in the chain ch by matching the topic name and the source timestamp of the data in the *data_write* event of cb_k with the ones in the *data_read* event of cb_{k+1}. Here, the source timestamp is used to uniquely identify a data item across the publisher and subscriber callbacks. We save the start time of cb_1 and while traversing the events related to the chain, we find the time when the output of the chain is published by cb_n. We can measure L_μ as the difference between the two times instants.

Algorithm 1 End-to-end Latency Measurement for Chain Executions ch_μ

Require: Chain of callbacks ch_μ, Events E, Callback cb_k, cb_1, and cb_n
Ensure: End-to-end latency L_μ

for each execution of a chain ch_μ **do**
  for each callback cb_k in ch_μ **do**
    if $cb_k = cb_1$ **then** ▷ First callback in the chain
      Find *CB.start* for cb_1
      Save start time of cb_1
    else
      Find *CB.start* for cb_k
      Find *data_read* event for cb_k
    Find *data_write* event for cb_k
    Find *CB.end* for cb_k
    Move to next callback cb_{k+1} in ch_μ by matching topic name and source timestamp
  Save the time when the output of ch_μ is published by cb_n
  Measure L_μ as the time difference between start of cb_1 and output of cb_n

3.5 DAG Generation for ROS2-based Applications

The process of modeling ROS2 applications as Directed Acyclic Graphs (DAGs) from execution traces is a methodical approach to mapping out task interactions. Initially, ROS2 system traces are analyzed to enumerate all active node, each identified by a unique name. This crucial step lays the groundwork for understanding the systems operational structure. Following this, the traces are scrutinized to extract unique data IDs, which signify the various messages or units of data exchanged between tasks, thereby delineating the application's communication. For every distinct node ID, we pinpoint the originating task (sender) and the destination tasks (receivers), clarifying the data flow direction, which is pivotal for comprehending task interdependences and interactions. Directed edges are then drawn from the sender to each receiver task for every data ID, visually representing data trajectories and task influences within the system. Subsequently, tasks are incorporated as vertices in the DAG, establishing a graphical representation of task dependencies and data pathways, with vertices symbolizing the tasks and directed edges indicating the flow of data. This step is meticulously repeated for each node ID to ensure all communication channels are captured in the DAG, thus achieving a holistic depiction of the systems communication framework. The culmination of this iterative process results in a unique DAG for each node ID, offering a transparent illustration of the system's functional workflow. These DAGs become essential tools for performance analysis, bottleneck identification, and the refinement of communication protocols to heighten system efficiency and dependability.

The timing model of the ROS2-based application is meticulously constructed as a DAG of ROS2 nodes based on the Callback Lists (CBlists) elicited from Algorithm 2. This intricate construction of the DAG is unfolded through several pivotal steps. Firstly, each Callback (CB) within a *CBlist* is conceptualized as a distinct vertex within the DAG. In instances where a service is invoked by multiple callers n_{cl}, an equivalent number of vertices are generated, each representing an entry in the *CBlist*. This approach is critical to accurately model the service as part of various distinct computational chains. The interconnection between these vertices is established based on the subscription and publication relationships among the CBs. When a CB cb_k subscribes to a topic to which another CB $cb_{k'}$ publishes, an edge is drawn from to cb_k to $cb_{k'}$ except in instances entailing data synchronization. This paradigm of edge creation is equivalently applied to service CBs, taking into account the distinct request and response topic names pertinent to the caller and client CBs. A divergence in the DAG occurs when a CB publishes on more than one topic or if it publishes on a topic subscribed by more than one CB. In such cases, an OR junction is marked at cb_k indicating that it has been triggered by the publication of either $cb_{k'}$ or $cb_{k''}$on the topic.

In scenarios where multiple CBs employed for data synchronization, identified as MS_α. in the *CBlist*, a vertex $cb_k^{\&}$ is added. This vertex, defined as an 'AND' junction, has incoming edges from all CBs in the message synchronize node. This 'AND' junction vertex, with incoming edges from all CBs in MS_α, becomes active only when new data is received on all subscribed topics of the CBs in the message synchronizer node. This methodical approach effectively delineates the complex interplay and dependencies among CBs in ROS2 nodes, thereby providing a detailed insight into the system's temporal structure.

Algorithm 2 Extract callback attributes for a ROS2 node

Require: *PID* of the ROS2 node, *ROSEvents*, *SchedEvents*
CBlist = [ ]
for *event* in *ROSEvents*(*PID*).***SortByTime***() **do**
  if *event.type* is *CB_start* ($\mathbb{P}_2$/$\mathbb{P}_5$/$\mathbb{P}_9$/$\mathbb{P}_{12}$) **then**
    record *CB.type* ▷ $\mathbb{P}_2$, $\mathbb{P}_5$, $\mathbb{P}_9$, $\mathbb{P}_{12}$ denotes different CB type
    CB.start = *event.time*
  else if *event.type* is *timer_call* ($\mathbb{P}_3$) **and** *CB.start* $\neq \emptyset$ **then**
    CB.ID = *event.ID*
  else if *event.type* is *take* ($\mathbb{P}_6$/$\mathbb{P}_{10}$/$\mathbb{P}_{13}$) **and** *CB.start* $\neq \emptyset$ **then**
    CB.ID = *event.ID*
    if *event.type* is *take_response* ($\mathbb{P}_{13}$) **then**
      CB.intopic = ***cat***(*event.topic*, *CB.ID*)
    else if *event.type* is *take_request* ($\mathbb{P}_{10}$) **then**
      CB.intopic = ***cat***(*event.topic*, ***FindCaller***(*event*, *ROSEvents*))
    else
      CB.intopic = *event.topic*
  else if *event.type* is *dds_write* ($\mathbb{P}_{16}$) **and** *CB.start* $\neq \emptyset$ **then**
    if *event.topic* is a *service_request* **then**
      top_out = ***cat***(*event.topic*, *CB.ID*)
    else if *event.topic* is a *service_response* **then**
      top_out = ***cat***(*event.topic*, ***FindClient***(*event*, *ROSEvents*))
    else
      top_out = *event.topic*
    append *top_out* to *CB.outtopic*
  else if *event* shows *will_not_dispatch_client* ($\mathbb{P}_{14}$) **then**
    *CB.** = $\emptyset$
  else if *event* shows *sync_subscribe* ($\mathbb{P}_7$) **and** *CB.start* $\neq \emptyset$ **then**
    CB.isSyncSubscriber = TRUE
  else if *event.type* is *CB_end* ($\mathbb{P}_4$/$\mathbb{P}_8$/$\mathbb{P}_{11}$/$\mathbb{P}_{15}$) **and** *CB.start* $\neq \emptyset$ **then**
    CB.end = *event.time*
    CB.ET = ***GetExecTime***(*CB.start*, *CB.end*, *PID*, *SchedEvents*)
    CBlist.***AddToCallback***(*CB*)
    *CB.** = $\emptyset$
return CBlist

Algorithm 3 Compute execution time—***GetExecTime***(...)

Require: *start*, *end*, *PID*, *SchedEvents*
ExecTime = 0
last_start = *start*
for *event* in *SchedEvents*.***SortByTime*** **do**
  if *start* < *event.time* < *end* **then**
    if *event.prev_pid* = *PID* **then**
      ExecTime = *ExecTime* + (*event.time* − *last_start*)
    else if *event.next_pid* = *PID* **then**
      last_start = *event.time*
  else if *event.time* > *end* **then**
    ExecTime = *ExecTime* + (*end* − *last_start*)
    return *ExecTime*

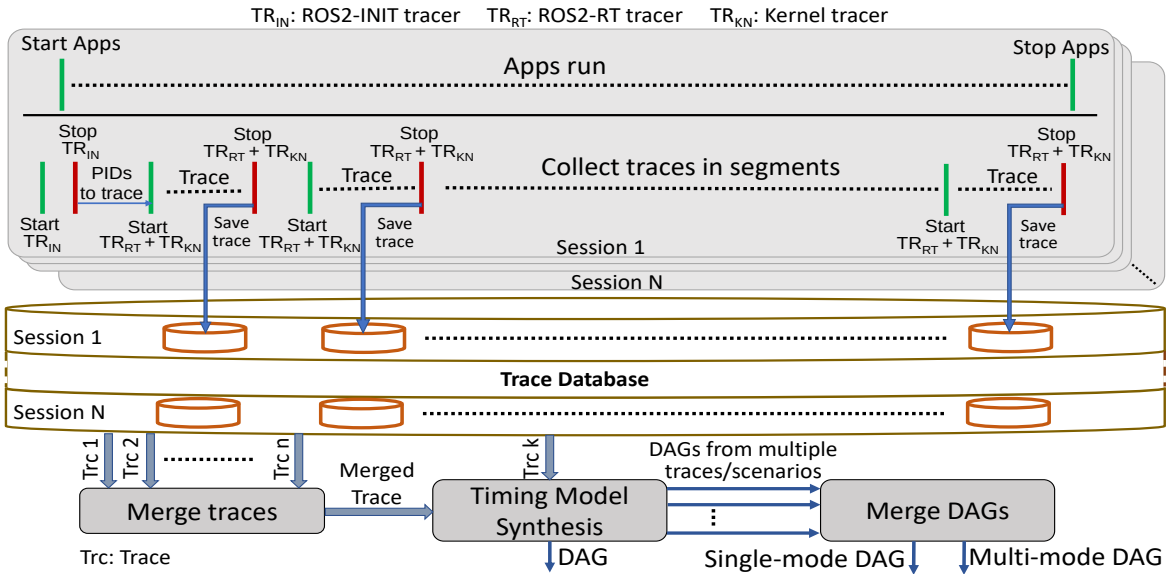
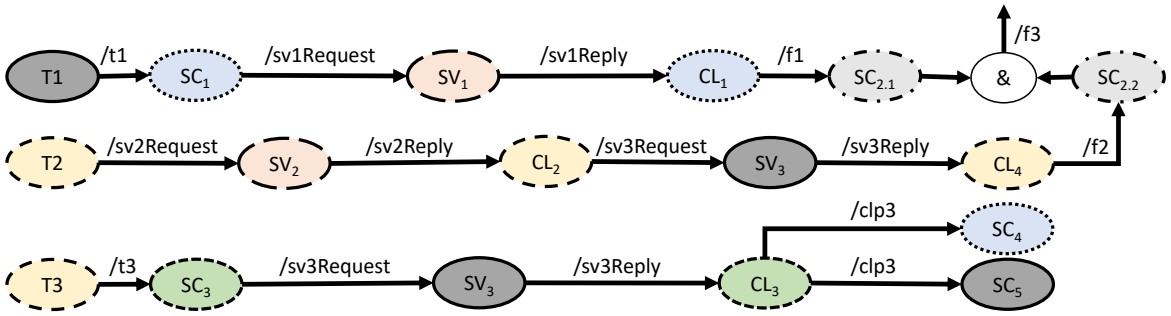


Figure 3.7: Deployment of tracing and model synthesis framework.

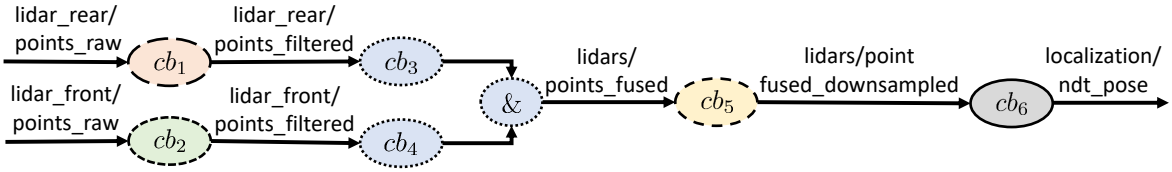
3.6 Framework Depolyment

Applications operate across varied Domains of Operation (DoO), experiencing significant changes in execution latency as a result. For instance, an autopilot application's response time can differ greatly between dense urban settings and highway, driven by the contrasting computation of the two scenarios. This diversity in operational conditions highlights the necessity of our deployment strategy, which emphasizes the collection of extensive trace data across multiple runs and different DoOs. Our methodology aims to construct accurate behavioral models of the system, ensuring that our timing models are closely aligned with the real-world performance of applications, thus effectively addressing the wide range of conditions they may encounter. We showed how to extract timing models based on measurements instead of formal worst-case execution time (WCET) analysis considering that the latter does not always scale for industry applications. Hence, for accurate modeling, we collect large amounts of traces across several runs of the applications and our framework is compatible with this. Even when one run is long and the trace buffers are limited in size, we can stop TR_{RT} and TR_{KN}, store the traces in a database server, and restart TR_{RT} and TR_{KN} with empty buffers. In the end, we might have a large number of traces in the server collected during multiple tracing sessions, as shown in Fig. 3.7.

There are several approaches to processing traces as shown in Figure 3.7. One method involves combining all traces and applying DAG generation algorithms on this unified trace. Alternatively, individual DAGs can be constructed for each trace, then merged where vertices and edges represent the aggregation of all individual DAGs, allowing for the computation of worst-case, best-case, and average execution times for callbacks by analyzing these aggregated DAGs. Another strategy might include combining traces by merging those from a single run to form individual DAGs, and then combining these DAGs across various runs. Additionally, the exploration of creating multi-mode DAGs is possible by merging traces according to specific modes or scenarios, such as urban versus highway conditions.



(a) Synthetic application.



(b) AVP localization.

Figure 3.8: Callbacks and the precedence relations between them.

3.7 Framework Evaluation

To evaluate the efficacy of our framework, we developed an experiment on a machine of 12 x CPUs and 46 GB of RAM, running Linux Kernel 5.4.1. The setup is composed of two applications running at the same time and bound to the same CPUs. The first application is the Autonomous valet parking (AVP) localization demo from Autoware.Auto project [12] which is an open-source project that provides a software stack for developing self-driving vehicles built on ROS2. The application runs for nearly 80 seconds and demonstrates an autonomous parking functionality during which the car starts from an initial position, drives autonomously to park in a target parking lot and then drives back to the initial position. The second application is a synthetic test that has been developed to mimic actual automotive chains with different driving functionalities. The second application is composed of six ROS2 nodes with different combinations of timer, subscriber, service, and client CBs. We run these applications 50 times, apply our DAG synthesis algorithms on traces collected for each run, and then merge these DAGs together, as explained in Sec. 3.6.

DAG Extraction and Timing Measurment

we ran our experimental setup and collected traces for 50 runs during which both application nodes were distributed to compute on different CPUs. Each CPU of the machine serves one node from the localization demo and another node from the synthetic test whose execution time varies with each run. The first goal of the experiment is to test the ability of the framework to generate multiple DAGS for multiple applications running at the same time which will always be the case in real-world scenarios. The second goal is to show the reliability of measurement-based approaches in extracting timing architecture that covers all the execution scenarios. From running the experiment, the framework was able to anticipate:

Table 3.2: Test Scenarios

Test Scenario	**Node Name**
Multiple timer callbacks in one node	n_2
Multiple subscriber callbacks in one node	n_6
Multiple service callbacks in one node	n_4
Multiple client callbacks in one node	n_2
Multiple clients using the same service	$\{cl_2, sc_3, sv_3\}$
Timer, subscriber, and service callback in one node	n_5
Synchronization on subscriber callback	sc_2
Divergence from callback	cl_3

- Synthetic-test DAG is generated as shown in Figure 3.8a. The DAG shows that the test is composed of six nodes $\{n_1, .., n_6\}$, each of which has multiple callbacks of different types. The ROS nodes of the test were configured as follows n_1= $\{sc_1, cl_1\}$, n_2= $\{T_2, T_3, cl_2, cl_4\}$, n_3= $\{sc_3, cl_3\}$, n_4= $\{sv_1, sv_2\}$, n_5= $\{T_1, sv_3, sv_4\}$, n_6= $\{sc_2, sc_4\}$. we can also find that the DAG was designed to cover the scenarios shown in Table 3.2.

- AVP Localization demo DAG is generated as shown in Figure 3.8b. From the DAG, we can deduce that the autonomous driving functionalities that run during the demo are perception and localization. The perception is implemented through the callback in the **point_cloud_fusion** node that subscribes to the filtered data from the front and rear lidar and publishes the fused points in the */lidars/points_fused* topic. It is also clear that the point_cloud_fusion node is implemented as a message synchronizer for the front and rear lidar data which is denoted on the captured DAG model with the & sign. The fused data is then down-sampled in the **voxel_grid_cloud** node and used by the callback in the **p2d_ndt_localizer** node to publish the car position in the *localization/ndt_pose* topic.

- The timing behavior (mBCET, mACET and mWCET) of the localization demo callbacks is shown in Table 3.3. It reports the measured best-case, average, and worst-case execution times (mBCET, mACET, and mWCET, respectively) of each of the 6 CBs. They are measured over 50 runs. Here, the most computationally expensive CB, cb_2, has an average processor load of 27%—the LIDAR data arrives at 10 Hz. Such measurements are useful even for simple debugging and optimization, e.g., balancing load across processor cores or keeping the load below a certain threshold while determining core bindings of ROS2 nodes.

- For each CB in *SYN*, we have used a constant computational load for a single run. By comparing the measured with the designed execution times, we have validated our framework's ability to measure accurately. We change the load of each CB in *SYN* across runs to evaluate if the execution time profiles of *AVP*'s CB are sensitive to varying interfering loads. We note that for *SYN*, the measurement results over 50 runs

Table 3.3: Execution times (in ms) of callbacks in AVP localization.

CB	Node	mBCET	mACET	mWCET
cb_1	filter_transform_vlp16_rear	13.82	17.1	19.82
cb_2	filter_transform_vlp16_front	23.31	27.07	30.5
cb_3	point_cloud_fusion	0.41	3.1	3.97
cb_4	point_cloud_fusion	0.38	0.62	3.36
cb_5	voxel_grid_cloud_node	6.58	8.47	13.36
cb_6	p2d_ndt_localizer_node	2.78	25.64	60.93

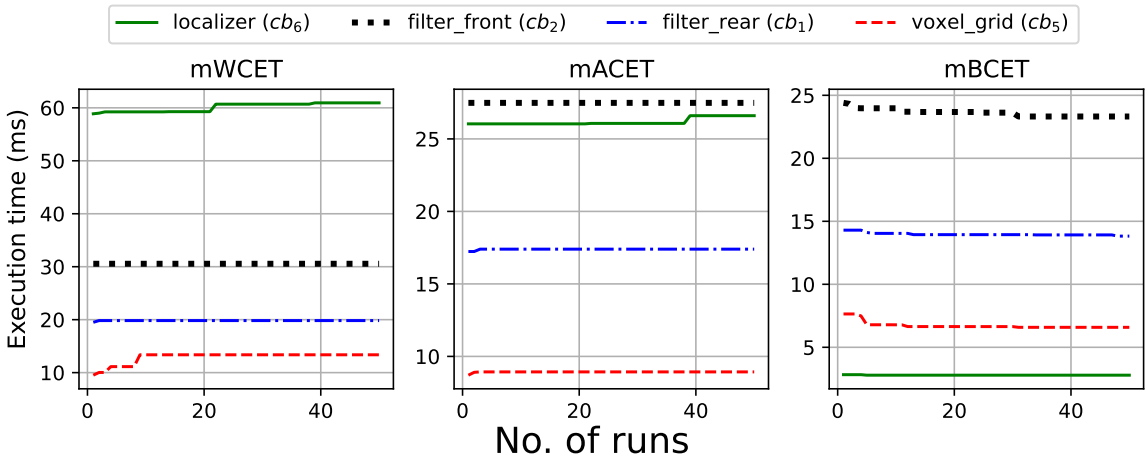


Figure 3.9: Estimation of timing attributes improve with more traces.

are not important so we do not report them. Further, Fig. 3.9 shows that mACET and mBCET change negligibly for cb_2 with increasing runs, while mWCET increases by 10% over 23 runs and thereafter remains unchanged. Such an evolution of mWCET shows that our modeling accuracy improves with more traces. We point out that, if we assume that test cases can be generated with a high coverage, our framework can support accurate model synthesis using tracing and measurement.

Tracing overheads:

In the course of an experimental setup where synthetic tests and Autonomous Valet Parking localization demo operate concurrently over 60 seconds, we observe two primary outcomes regarding tracing overheads. First, the experiment results in the generation of approximately 9MB of trace data, highlighting the volume of information collected during the process. Second, an analysis using bpftool [83] a tool for the manipulation and inspection of eBPF programs and mapsreveals that the eBPF probes utilized during the experiment average a computational demand of 0.008 CPU cores. This translates to a mere 0.3% of the total computational load attributed to the operations of the SYN and AVP applications. Such metrics are indicative of the efficiency and low overhead introduced by eBPF probes in monitoring and tracing system performance in complex, real-world applications.

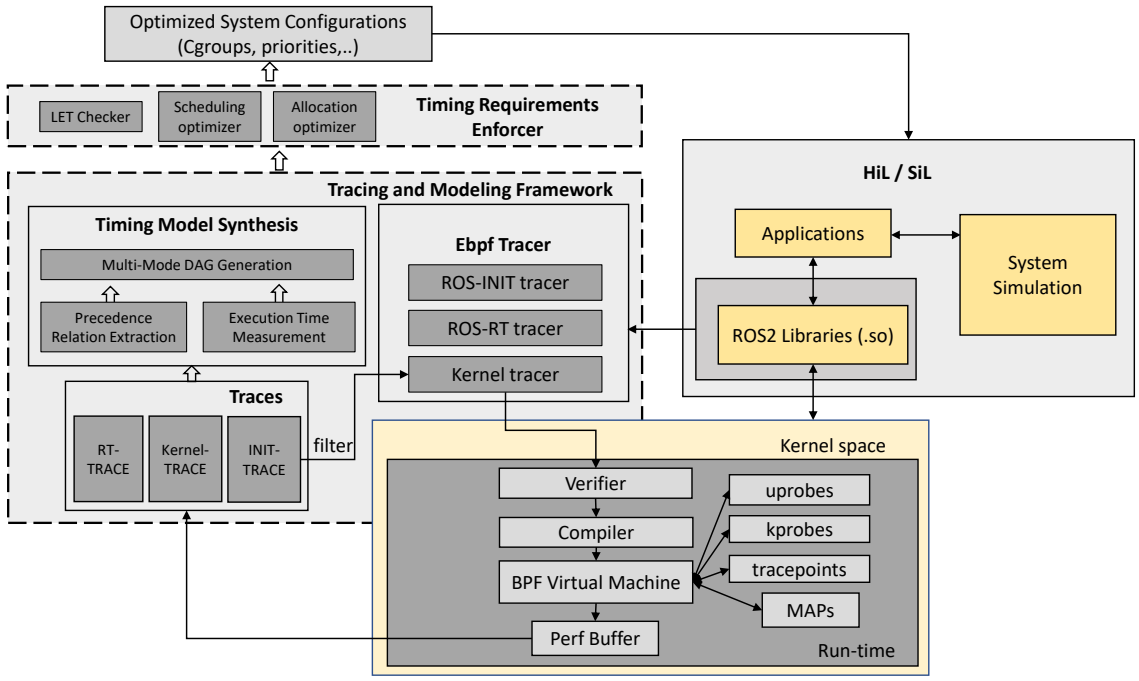


Figure 3.10: ROS2 Applications Integration Platform

3.8 Applications integration and optimization

The proposed framework in this chapter is not only able to perform time debugging and extract timing properties of the ROS2 applications but it is also beneficial when it comes to system integration. The framework as shown in Figure 3.10 can be a part of the Hardware in Loop (HiL) process where the generated timing models are checked against the agreed system requirements. These requirements can be for example checking if the running system is following some timing paradigms as Logical Execution Time(LET) [44] or achieving predefined Key Performance Indicators (KPIs). It proposes a flow for checking the timing properties for each of the application components while protecting the Intellectual Property (IP) of the third-party component developers. Additionally, one of the key strengths of the framework is its ability to deduct the intra-components performance effect which is the effect of the integrated components exerted on the timing performance when they share resources such as CPU and memory. Such reported information from the framework can be effectively used to propose further as C-group optimizations, scheduling priorities changes, or even optimized task-core allocation. Although the introduced profiling methodology is application-independent, It has to be considered that the processes have to be repeated when changing the target hardware. In the next chapter, we will show how we will use the extracted model to improve the real-time performance of the running applications.

3.9 Conclusion

In this chapter, we have explored the development of autonomous applications over ROS2, particularly within time-critical systems such as automotive, where recent years have witnessed a growing interest in model-based timing analysis and schedule optimization for ROS2-based applications. To complement these approaches, we introduced a tracing and measurement framework designed to obtain timing models of ROS2-based applications, leveraging the capabilities of the eBPF to probe various functions within the ROS2 middleware. This approach allows for the extraction of arguments or return values to elucidate the data flow within applications. By combining event traces from both ROS2 and the operating system, our framework generates a directed acyclic graph that illustrates ROS2 callbacks, delineates precedence relations among them, and elucidates their timing properties.

Moreover, our framework is not only compatible with existing analyses but also enhances the modeling of complex scenarios such as message synchronization in sensor fusion and service requests from multiple clients in motion planning. This capability is crucial, especially considering real-world scenarios where application code may be confidential and formal models are not readily available, yet our framework still facilitates the application of existing analysis and optimization techniques. Furthermore, we extended our investigation to include the tracing of ROS2-based autonomous applications using eBPF, implementing algorithms to process these traces and synthesize accurate timing models. Our comprehensive framework supports not only various ROS2 versions and DDS implementations but also offers scalability to other software architectures (e.g., AUTOSAR) and operating systems (e.g., QNX), thereby providing a versatile tool for debugging timing chains and optimizing them according to specific requirements. In alignment with the work of [22], our framework also opens avenues for enforcing runtime optimizations that can significantly enhance system performance, showcasing a holistic approach to the timing analysis and optimization of ROS2-based systems.

Chapter 4

Latency and Jitter Management

With the advancement in artificial intelligence and computer vision, there has been a strong drive towards enabling autonomy in many cyber-physical systems including cars, drones, and robots. ROS2 has become especially popular for this purpose due to a vast repertoire of open-source implementations of state-of-the-art artificial intelligence and computer vision algorithms available for it, its portability to different operating systems, and its support for collaborative development of complex software applications. As discussed in the previous chapter, software applications over ROS2 consist of a collection of modular functional components developed as nodes. nodes comprise one or more callbacks, each handling events generated by a timer or data availability. To communicate with each other, the nodes send and receive data using a publishing-subscribing mechanism through topics following Data Distribution Service (DDS) standard [92]. As a result, ROS2 applications contain computation chains as defined in 3.1.1 formed by a sequential invocation of callbacks connected via topics. Typically, such a chain reads and processes sensor data, performs planning, executes control logic, and applies actuation signals, e.g., [48].

In control systems literature, it has been shown that a long and unpredictable sensing-to-actuation delay (or end-to-end latency) in such chains can lead to a lower control performance or even lead to unsafe physical behavior [82, 112]. Two main factors affect the duration of this delay either the computation time of the callbacks or the interference by other workloads in the same processing unit. The latter factor has been long considered in real-time systems research, and techniques have been proposed even for ROS2 environment to minimize the interference to a high-criticality chain caused by best-effort workloads, e.g., [139, 33, 22, 5].

Computation chains in ROS2 are prone to significant end-to-end latency fluctuations, known as *jitters* defined in section 3.1. These variations often result from computational demands that vary with environmental factors, such as the number of objects in a scene. To

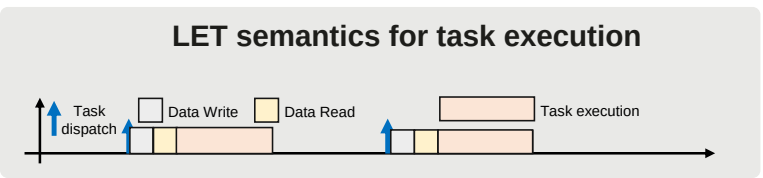


Figure 4.1: Logical execution time (LET)

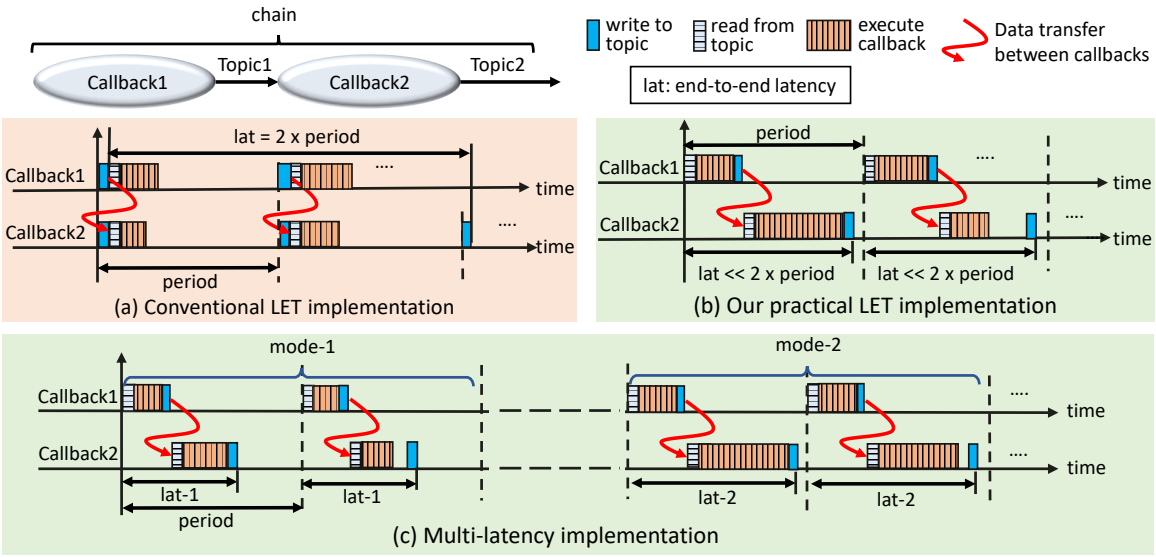


Figure 4.2: Latency shaping vs conventional logical execution time implementation.

address these jitters, the *Logical Execution Time* (LET) approach, a standard in the industry (referenced in AUTOSAR [8]), is widely used, as detailed in various studies [19, 95, 96, 43, 51, 52]. In a conventional LET implementation, at the beginning of a period, a task first writes its output from the previous execution, then reads the new input and starts processing it. Hence, a task spends a constant time between data read and write, which is equal to its period. This establishes a consistent time frame for each task, leading to a predictable end-to-end latency for the entire chain, illustrated in Figure 4.2a. However, integrating LET in ROS2 chains presents challenges. The ROS2-adopted DDS communication mechanism complicates controlling the timing of topic reads and writes. Each of the implementations of LET talks about specific instants for data read and write, however, it becomes challenging to control the timings of tasks using publish-subscribe mechanisms for communication. Additionally, LET paradigm ensures that each task has a fixed response time which also translates to a fixed end-to-end latency of task chains This means that the worst-case and average end-to-end latencies of task chains become too high for an LET implementation. The end-to-end latency of the task chain becomes equal to the summation of periods of the nodes forming the chain. Besides, the implementations of LET consider time-triggered execution of tasks enforcing a specific implementation method of task chains. However, this is not always valid in the case of ROS2 chains where we may have time-triggered and also event-triggered chains. Finally, the LET concept does not support multi-mode performance-oriented application design as there is no provision for multiple discrete end-to-end latencies of a task chain corresponding to different modes.

Towards a more practical LET implementation of a ROS2 chain, we propose to use *table-driven reservation servers*. Such a server reserves specific time slots on a processing unit in which it can run its assigned threads. Unlike conventional time-triggered scheduling where computation time is reserved for each task based on its worst-case execution time, we propose to assign chained callbacks to a server, capable of running multiple threads.

We take a practical approach and dimension the server based on the maximum end-to-end latency of the assigned callbacks that we measure during a priority-driven execution [3]. Our solution extends to include the two modes of data publication in ROS2: *asynchronous* and *synchronous*. Asynchronous publishing allows data to be sent independently of other processes, offering flexibility in transmission timing. Conversely, synchronous publishing involves sending data in alignment with specific system events or scheduled times [41]. To achieve comparable outcomes with the synchronous publish mechanism, we encounter a significant challenge. This arises when the final callback in the chain exhibits substantial execution time variability and simultaneously handles the chain's output publication. Our solution involves an architectural *extension* of the chain with an additional callback, specifically for republishing the chain's output. This additional callback executes within a subsequent, time-bound server. This server is specifically designed to exert precise control over the timing of the chain's output publication. We engineer a technique to implement the chain extension exploiting ROS2-supported modular software development, i.e., ***without any modification or recompilation*** of the application source code. We design a complete tool flow automating each step from profiling to running the chain using reservation servers.

In this chapter, We introduce the concept of latency shaping to minimize jitters in a ROS2 computation chain with a negligible impact on the worst-case end-to-end latency. We also demonstrate its versatility in supporting multi-mode chain operation without the need for dynamic schedule reconfiguration, which is crucial for next-generation autonomous systems. We additionally propose two chain-aware implementations of latency shaping that are compatible with ROS2 semantics. Besides, we develop tools that will automatically determine the optimal configuration for reservation servers and then create and configure them as well as assign chain components to run inside them. In essence, we enable design automation for latency shaping. We perform experiments to demonstrate that our proposed mechanism can be used to implement a ROS2 chain following the conventional LET concept. we also apply latency shaping on a real-world benchmark, i.e., for a chain from *Lidar* to *vehicle pose estimation* in Autoware's Autonomous Valet Parking (AVP) [48]. Finally we also perform experiments to demonstrate that our proposed concepts can be applied to implement a ROS2 chain following the conventional LET concept and a DAG comprising ROS2 callbacks.

4.1 Scheduling in operating systems

In computing, the transition from bare-metal applications to managing tasks by operating systems represents a pivotal advancement. Operating systems are a crucial intermediary between hardware and software, facilitating a more efficient, secure, and user-friendly computing environment. Bare-metal programming, where applications directly interact with hardware without an intermediary layer, severely limits multitasking capabilities and does not effectively provide the necessary abstractions for managing hardware resources. Operating systems overcome these limitations by introducing a layer of management that allows multiple applications to run simultaneously, sharing hardware resources without direct conflict. This layer enhances security by isolating applications from hardware and improves the overall efficiency and scalability of computing systems, enabling a more versatile and robust application development and execution environment.

Scheduling in an operating system is a fundamental function that manages the execution of multiple processes by allocating CPU time among them. This process ensures that system resources are utilized effectively, maintaining responsiveness and efficiency. Operating systems employ scheduling algorithms to decide which process runs at any given time based on priority, process execution time, and system resource availability. This mechanism allows for a balanced distribution of processor time among competing processes, optimizing performance and ensuring that no single process monopolizes the system resources. Through scheduling, operating systems achieve multitasking, providing the capability for processes to execute simultaneously, even on single-core CPUs. This is critical for maintaining system performance and responsiveness, especially in environments where multiple applications need to run concurrently.

Linux, a widely used operating system, implements several scheduling algorithms to manage processes efficiently. Among these, the Completely Fair Scheduler (CFS) and the O(1) scheduler are prominent examples, each designed to allocate CPU time among processes to maximize fairness and efficiency. However, despite the sophistication of these scheduling mechanisms, Linux traditionally does not guarantee real-time performance. This limitation stems from the nature of these schedulers, which are designed for general-purpose computing, where fairness and overall system throughput are prioritized over strict execution timing. Real-time performance requires deterministic response times, a feature that general-purpose schedulers in Linux are not designed to provide. These scheduling algorithms' non-real-time nature means that while they are highly effective for a broad range of computing tasks, they cannot guarantee the precise timing required for real-time applications, where even minor deviations in timing can lead to significant problems.

To cater to the stringent requirements of real-time applications, Linux incorporates specific scheduling policies designed to offer deterministic execution times. Among these, `SCHED_FIFO` (First In, First Out) is a real-time policy where tasks are scheduled according to their arrival time, with no task preemption once it starts running, unless a higher priority task arrives. SCHED_RR (Round Robin) is another real-time policy similar to `SCHED_FIFO`, but it adds time slicing, allowing tasks of the same priority to be executed in a round-robin fashion, ensuring that each task receives an equal share of CPU time within its priority level.

SCHED_DEADLINE is a more recent addition, providing deadline-based scheduling where tasks are run based on their ability to meet certain deadlines, which is crucial for applications where timing is paramount. This policy schedules tasks by considering their execution time, deadline, and period, ensuring that tasks are completed before their specified deadlines.

The challenge of achieving predictable performance in real-time schedulers, particularly in the context of cyber-physical systems (CPS), is significantly compounded by the intricate interactions between software and hardware, especially when interfacing with the external world and awaiting data from sensors. Cyber-physical systems, which integrate computation with physical processes, rely heavily on real-time data from sensors to make timely decisions and control mechanisms in a physical environment. These systems must process and respond to sensor inputs within stringent time constraints to maintain stability, safety, and efficiency. However, the variability in sensor data acquisition times, the processing of this data, and the execution of control commands can introduce unpredictability in system performance.

This unpredictability is largely due to the non-deterministic nature of software-hardware interactions within such systems. For example, delays may occur in reading sensor data due to hardware-level contention, signal noise, or fluctuations in environmental conditions affecting sensor responsiveness. Additionally, the process of transferring sensor data through the system's I/O interfaces can encounter variable latencies, further complicating the scheduling and timing of tasks. When these hardware-induced delays intersect with the software's need to process and react to sensor data in real time, it creates a challenge for even the most sophisticated real-time scheduling algorithms to maintain deterministic performance.

Moreover, the complexity of cyber-physical systems, which often operate in dynamic and unpredictable environments, means that software must continuously adapt to new data and conditions. This adaptability requirement can strain real-time schedulers, as they must constantly adjust task prioritisation and resource allocations to meet changing demands, all while striving to adhere to strict timing constraints. Thus, despite the advancements in real-time scheduling techniques, the inherent uncertainties in software-hardware interaction, particularly in sensor-dependent cyber-physical systems, pose a significant barrier to achieving jitter-free and latency-insensitive performance.

In the context of this thesis, we will primarily utilize SCHED_FIFO as the baseline for comparison with other scheduling mechanisms. SCHED_FIFO's straightforward scheduling approach provides a clear benchmark for evaluating the efficiency and responsiveness of other scheduling policies within the Linux operating system. By focusing on SCHED_FIFO, we aim to establish a foundational understanding of real-time scheduling performance, upon which we can compare the effectiveness of other algorithms that will be discussed in the next section. This comparative analysis will not only highlight the strengths and weaknesses of each scheduling policy in real-time contexts but also offer insights into optimizing Linux for real-time applications, thereby contributing to the broader field of operating systems research. Through our work, we endeavor to elucidate the nuances of Linux scheduling mechanisms, providing a comprehensive overview that aids in the development and deployment of real-time systems.

4.1.1 Table-driven reservation-based scheduling

Reservation-based schedulers in real-time systems, particularly in the context of global static scheduling of mixed-criticality systems, are designed to allocate computational resources (CPU time, memory, etc.) to tasks based on predefined reservations. These schedulers ensure that tasks receive a guaranteed amount of resources within a specific time frame, improving predictability and reliability in systems with varying levels of criticality. This approach is vital for mixed-criticality systems, where tasks of different importance levels coexist, ensuring that high-criticality tasks have the resources they need, even in the face of system changes or failures. Reservation-based schedulers are essential in real-time systems because they efficiently manage the execution of tasks with different criticality levels on a common platform. These schedulers can maintain system stability and performance by allocating resources through reservations, even when preempted or delayed by low-criticality tasks. This capability is crucial for critical real-time systems, such as those in the automotive, aerospace, and healthcare industries, where correctly executing high-criticality tasks is essential for safety and functionality.

In this chapter, we exploit table-driven reservation-based scheduling to enable deterministic execution of ROS2 computation chains. In such a scheduling environment, threads run inside reservation servers. A reservation server R_j is a sequence of time slots that are statically defined as defined in 4.1.1. A thread $\mathscr{T}_\alpha$ assigned to R_j can only run inside the time slots belonging to R_j. These reservation servers, on the one hand, enable to predict accurately when it will run the assigned threads, thereby offering timing determinism. On the other hand, they constrain the amount of time for which the assigned threads can run on the processor, thereby providing temporal isolation between different sets of threads.

Definition 4.1.1. Reservation Server (R_j)

R_j is defined using a set of time slots $\{sl_{j,1}, sl_{j,2}, \ldots, sl_{j,\eta_j}\}$ and a cycle time c_j where $sl_{j,l}$ start and end time are denoted as $st_{j,l}$ and $et_{j,l}$ repectively.

To perform our experiments, we use **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems (LITMUS^{RT}) [24]. It is a real-time extension of the Linux kernel providing real-time schedulers and synchronization mechanisms. It has a scheduler plugin *P-RES* implementing partitioned reservation-based scheduling that allows to define and use table-driven reservation servers. Considering that LITMUS^{RT} helps to control scheduling in a multi-core processor, it allows to assign a reservation server R_j to only one processor core which we denote by p_j. Note that a set of threads $\mathbb{T}_j$ can be mapped to a reservation server R_j. If multiple threads are ready-to-run in a reservation server, LITMUS^{RT} implements a round-robin scheduling policy to allocate processor time to these threads. Further, LITMUS^{RT} allows to assign a priority ρ_j to a reservation server R_j. This is a very useful feature to isolate high-critical applications from best-effort applications. That is, simultaneous to a reservation server R_j running threads from high-critical application, we can define another reservation server $R_{j'}$ to run best-effort threads where $\rho_j > \rho_{j'}$. In this case, $R_{j'}$ can run its threads only when there are no ready-to-run or running threads in R_j.

4.1.2 Timing behavior of computation chain

Table 4.1: Measured maximum end-to-end latency and jitters [ms]

	Interfering load per 100 ms				
	0	10	20	30	40
SCHED_FIFO latency.	57.35	58.1	58.78	59.8	62.8
SCHED_FIFO jitters.	16.2	16.72	17.14	17.73	18.13
Table-driven reservation latency.	$\approx$ 80 ms				
Table-driven reservation jitters.	$\approx$ 5 ms				

To understand the timing behavior of a ROS2 computation chain when implemented using different scheduling policies, we created five ROS2 nodes $\{N_1, N_2, N_3, N_4, N_5\}$ as shown in Figure 4.3. The chain starts with a timer callback cb_1 in N_1 that runs every 100 ms and publishes data in a topic T_1. Each of the other nodes runs a subscriber callback cb_i that reads data from topic T_{i-1}, runs some computation load, and then publishes data in T_i. The output of the chain is, hence, published in T_5. Each callback (timer or subscriber) in the chain has a WCET of 15 ms and a BCET of 5 ms and we consider a uniform distribution of the execution time between the BCET and the WCET. All nodes in the chain run on the same processor core. In addition to the chain, we have also created another ROS2 node N_{int} that runs on the same core and acts as an interfering workload for the chain. We vary the execution time of the timer callback cb_{int} inside N_{int} between $\bar{e}_{int}$ and $\frac{\bar{e}_{int}}{2}$ where $\bar{e}_{int} \in \{10, 20, 30, 40\}$ ms.

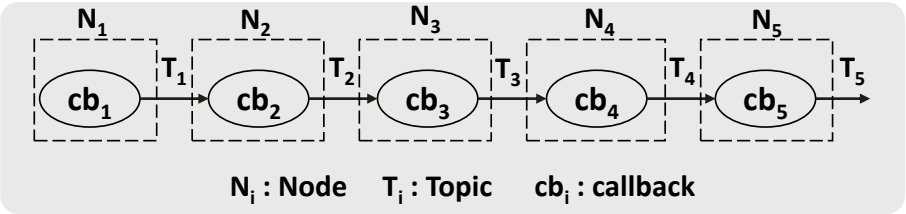


Figure 4.3: Motivational example

We use two different scheduling policies in Linux to run the ROS2 callbacks on a processor core.

- `SCHED_FIFO` ($\mathbb{S}_1$): Using `SCHED_FIFO`, we assign a high priority to the thread $\mathscr{T}_i$ executing cb_i in N_i where $N_i \in \{N_1, N_2, N_3, N_4, N_5\}$ and, a low priority to the thread $\mathscr{T}_{int}$ executing cb_{int} in N_{int}.

- *Table-driven reservation* ($\mathbb{S}_3$): We use Linux patched with LITMUS^{RT} for this case. We create 5 reservation servers R_1, R_2, R_3, R_4, and R_5 with the time slots (ms) [0,16], [16,32], [32,48], [48, 64], and [64,80], respectively, and a cycle time of 100 ms. $\mathscr{T}_i$ uses R_i and, hence, can run only in the respective time slots. We assign a high priority to these reservation servers. Further, we create a low-priority reservation server R_{int} with the time slot (in ms) [0, 100] and a cycle time of 100 ms. $\mathscr{T}_{int}$ runs using R_{int}.

In each of the above configurations, we try to prioritize the execution of the threads running the callbacks in the computation chain.

We measure the worst-case and the best-case end-to-end latency of the chain, denoted by L_{wc} and L_{bc}, respectively. Further, we define the jitters as $\frac{L_{wc}-L_{bc}}{2}$. Table 4.1 shows the variation in the maximum end-to-end latency and the jitters with varying interfering processor loads for the two configurations. We have the following main observations:

- By assigning a higher priority to the workload in the chain using `SCHED_FIFO`, we obtain the lowest worst-case end-to-end latency. In this case, the jitters in the chain are mainly due to the varying execution times of the callbacks in the chain—such a variation in execution times is common in real-world applications. Hence, with `SCHED_FIFO`, it is challenging to control jitters in a chain when it has time-varying workloads.

- Using table-driven reservations, the worst-case latency increases compared to `SCHED_FIFO`. This is mainly due to the over-provisioned length of the time slots in the reservation stations that are calculated based on the worst-case execution times of the callbacks in the chain. That is, until a callback's time slot comes, it cannot start even when the previous callback has finished execution. Nevertheless, due to such constrained progress in the chain's execution, the jitters reduce to reflect only the variation in the execution time of the last callback in the chain. Hence, with table-driven reservations, the jitters reduce at the expense of an increased worst-case end-to-end latency value.

Concerning the above observations, the goal of this work is to get the best of `SCHED_FIFO` and table-driven reservations. That is, we want to eliminate jitters in the chain while the worst-case end-to-end latency does not increase compared to what is possible using `SCHED_FIFO`.

4.2 The Logical Execution Time in Control Systems

In control systems, the propagation of delay and jitter from sensing to actuation time can lead to a range of challenges that impact both the performance and stability of the system. Delays increase the overall response time, making the system slower to adapt to changes or disturbances, which can significantly degrade its performance. These delays, along with jittera variability in timingcan introduce stability issues, particularly in systems that rely on precise timing for feedback loops, leading to oscillations or even divergent behavior. As a consequence, there's an increase in overshoot and settling time, where the output not only exceeds the desired setpoint but also takes longer to stabilize. To counter these effects, control systems may require more complex strategies, such as predictive or adaptive control, increasing the design and analysis complexity. Moreover, significant delays and jitter compromise the system's robustness and reliability, making it more prone to disturbances and less predictable over time [49] [29].

The Logical Execution Time (LET) model is used to add time determinism to periodic computations by eliminating output jitter. In essence, LET delays the program output of a task at the end of the task period, trading delay for output jitter. This model is particularly useful in real-time systems, where deterministic timing and predictable behavior are essential, especially for tacking the jittery behavior of control systems. In a system composed of periodic tasks and runnables, each task τ_i is characterized by a tuple $((T_i, C_i, O_i))$, where T_i represents the period of the task, indicating how frequently the task is activated. C_i is the worst-case execution time, defining the maximum time required for the task to complete its execution. O_i specifies the initial offset, which is the delay before the task's first activation within its period. Tasks release an infinite sequence of jobs, with the first job being released at time O_i and subsequent jobs at times $r_{i,k} = O_i + (k-1)T_i$, under the assumption that $O_i < T_i$ for all tasks. The scheduling framework within which these tasks operate is defined by the hyperperiod of the system, calculated as the least common multiple of the task periods. This ensures that the task set can be scheduled independently of the offset, meaning for any task τ_i, its worst-case response time R_i should not exceed its period T_i, for any given O_i. This model facilitates a deterministic approach to task scheduling to ensure predictability and reliability of real-time systemsy [84] [106].

In the LET paradigm, tasks are categorized as either writers or readers of labels, with a unique writer per label but potentially multiple readers. Task communications are synchronized to occur at deterministic times, aligned with task activation periods. LET mandates a fixed interval from when a task reads its input to when it writes its output, regardless of the execution time. Inputs and outputs are logically updated at the start and end of their LET period, respectively. This setup, where LET typically equals the task period, presumes that these updates incur no computation time—a condition emulated through buffering ensuring communication determinism and system stability [84].

The adaptation of LET for multicore autonomous systems using middleware as AUTOSAR presents several challenges as discussed in [20] [44]. It requires significant adjustments to task code generation and runnable execution, posing notable implementation hurdles. Establishing effective communication mechanisms, either through explicit models with LET runnables or through implicit models without impacting timing behavior, adds complexity. Additionally, the necessity for memory allocation for local variables to mirror inputs and outputs complicates LET implementation on multicore platforms. Ensuring precise synchronization of input and output operations, whether via hardware or software, is crucial and challenging. Furthermore, task and core allocation strategies significantly influence the LET model's efficiency and performance, making it imperative to optimize these aspects for multicore environments. These challenges highlight the intricate nature of embedding LET within AUTOSAR applications, necessitating advanced solutions and tailored implementation strategies to leverage LET in autonomous systems successfully.

In this chapter, we explore LET model for ROS2-based applications, addressing the inherent challenges such as controlling the timing of read and write for ROS2's DDS communication publishing-subscribing mechanism. The conventional LET approach, which ensures tasks maintain a constant time between data read and write equivalent to their period, leading to predictable end-to-end latencies, faces complications in ROS2 chains. These include difficulties in enforcing time-triggered execution and accommodating event-triggered chains, alongside the inability to support multi-mode, performance-oriented application design with varying end-to-end latencies. We aim to tackle these issues, ensuring that the integration of LET in ROS2 does not compromise the model's benefits of fixed response times and predictable latency.

4.3 Latency Shaping

Let us consider a chain *ch* with n callbacks. Here, *ch* is a critical chain and the goal is to schedule the computations in *ch* with minimal interference so that the worst-case end-to-end latency is minimized. Further, *ch* extends between sensing and actuation where we want to control the variation in its end-to-end latency, i.e., the jitters must be as short as possible. In this section, we demonstrate how our proposed mechanism uses table-driven reservation-based scheduling to achieve the above goals while considering ROS2 semantics and different characteristics of the chain. We note that table-driven reservations have proved to be very effective in implementing certifiable safety-critical systems due to their inherent determinism [132].

4.3.1 Controlling the worst-case end-to-end latency

For each callback cb_k in *ch*, we get a measured worst-case execution time (mWCET) that we denote by $\bar{e}_k$. The conventional method to schedule the callbacks in the chain would be to define a reservation server for each thread running a callback, as we have done in Section 4.1.2. In that case, we can map a thread $\mathscr{T}_k$ running cb_k to a reservation server R_k.

In order to control the worst-case end-to-end latency, we obtain a maximum measured end-to-end latency $\overline{L}_{PR}$ of *ch* when scheduled using a high priority and `SCHED_FIFO`. Following our goal to keep the maximum end-to-end latency close to $\overline{L}_{PR}$, we define a latency control server R_γ as defined in Definition 4.3.1. We assume that $\overline{L}_{PR} < c_\gamma$, i.e., one execution of the chain can be accomplished within a cycle. The maximum processor utilization contributed by the chain is given by $\overline{U} = \frac{\overline{L}_{PR}}{c_\gamma} < 1$. That is, one CPU core can run the whole chain without any overlapping executions. we then assign all threads in $\{\mathscr{T}_k | 1 \leq k \leq n\}$ (running the callbacks in *ch*) to R_γ. Considering that the callbacks run by these threads form a chain, they do not interfere and delay each other.

Definition 4.3.1. Latency Control Server (R_γ)

R_γ is defined as a server with a time slot $sl_\gamma := [0, \overline{L}_{PR} + \varepsilon\ ]$, cycle c_γ:= the period of the chain execution where ε is as a short time buffer to *ch* in case of overshooting.

At the beginning of sl_γ in a cycle, cb_1 runs. Thereafter, cb_k can start whenever it has received the data from cb_{k-1}—similar to using $\mathbb{S}_1$ and a `SCHED_FIFO`. Hence, the chain will finish execution as soon as possible and the maximum end-to-end latency $\overline{L}_{SH}$ using this technique will be approximately equal to $\overline{L}_{PR}$. We assign a high priority to R_γ, e.g., $\rho_\gamma = 1$ using LITMUS^{RT}. To improve the utilization of the processor core when *ch* does not run until $\overline{L}_{SH}$, we can define another server R_{BE} with $\rho_{1'} > \rho_1$ and map threads performing best-effort tasks to it. In that case, R_γ basically isolates *ch* from the interference by the best-effort tasks and we still get $\overline{L}_{SH} \approx \overline{L}_{PR}$.

Experimentally, we have also studied the DDS communication threads in a ROS2 node. We have observed that a few of them—depending on the DDS implementation we use—

influence the end-to-end latency by delaying the communication between the chain's callbacks. Hence, we must isolate these threads from interference by best-effort workloads. Further, we have observed that while they run during data send and receive, they also run at other instants performing protocol-related tasks (e.g., sending heartbeat signals and polling message queues). The reason for this is that they are also responsible for other protocol-related tasks, e.g., sending heartbeat signals and polling message queues. Hence, we cannot assign them to servers with specific time slots without delaying an important DDS-related task significantly (e.g., by tens of milliseconds). Also, they typically run for a short duration (several hundred microseconds) when they wake up. They mostly run in parallel to the ROS2 threads executing the callbacks. Considering the above observations related to the DDS threads, we bind them to a different processor core and use $\mathbb{S}_1$ to schedule them with a high priority. This ensures that the DDS communication in the chain will have a minimal latency.

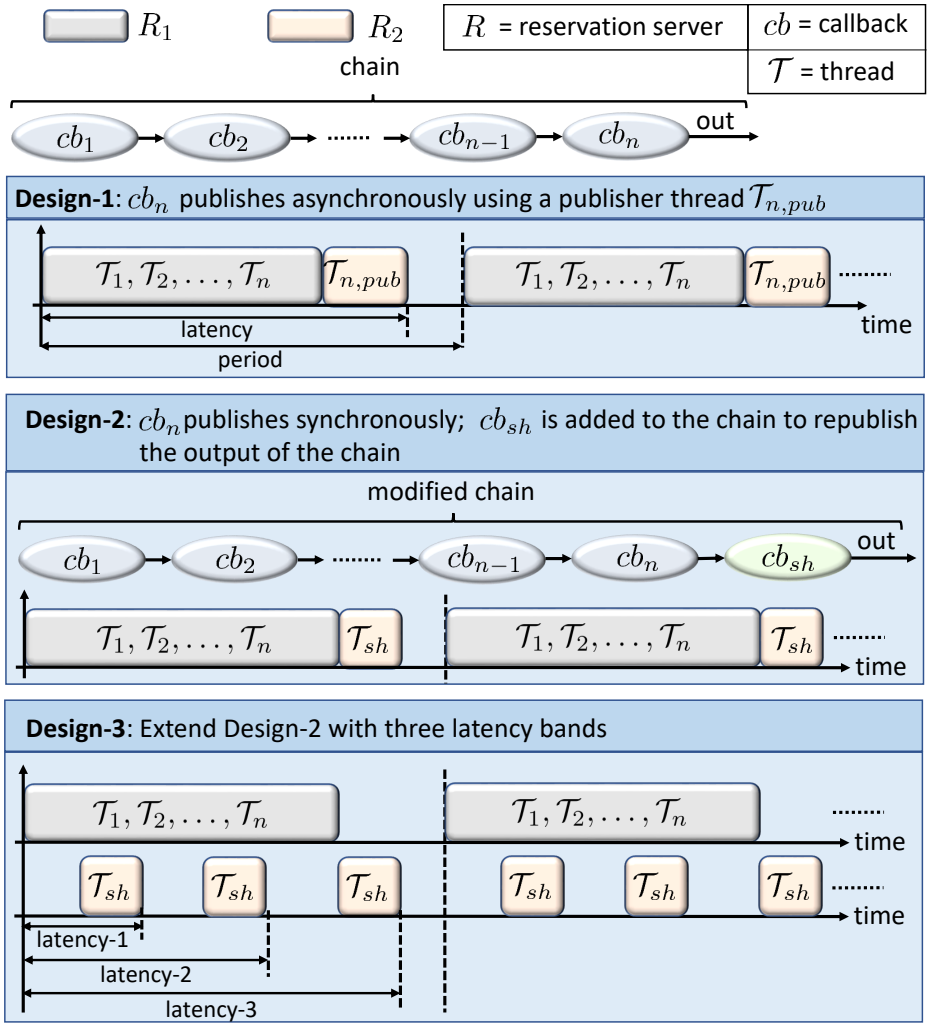


Figure 4.4: Chain-aware single- and multi-band latency shaping.

4.3.2 Controlling the variations in the end-to-end latency

When we put the entire chain in a high-priority reservation server R_γ as in the previous section, we can minimize the worst-case end-to-end latency. Now, to realize the second goal of minimizing the jitters, we study two different mechanisms for publishing the chain output, namely, asynchronous and synchronous publish [41].

Asynchronous publish

Asynchronous publishing in ROS2 DDS signifies a publication methodology that leverages an internal thread for data dispatch, enabling the write operation to conclude prior to the actual data transmission. It grants users enhanced control over their threads by ensuring a quicker and more deterministic return of the write operation, thereby obviating the need for the user thread to engage in network communications. This feature is particularly crucial when the user thread is involved in time-sensitive tasks. In terms of performance, although synchronous publication might excel in conserving context switching resources, asynchronous mode introduces the possibility of configuring flow controllers. This is instrumental in preventing network congestion, especially at high data transmission rates. Additionally, asynchronous publishing is adept at circumventing blocking behaviors, notably when the History is at full capacity. It necessitates the configuration of QoS policies related to history management to avert such scenarios. Consequently, asynchronous publishing in ROS2 DDS is invaluable for time-sensitive applications and contexts necessitating efficient data transfer, offering superior thread control, tailored performance optimization for substantial data flows, and the mitigation of blocking incidents [42].

Definition 4.3.2. Asynchronous Variation Control Server (R_δ)

R_δ is defined as a server with a time slot sl_δ : = [et_γ, et_γ + $\overline{e}_{n,pub}$], cycle c_δ:= c_γ and priority ρ_δ : =ρ_γ where et_γ marks the end of the time slot sl_γ in R_γ and $\overline{e}_{n,pub}$ is the maximum time required to publish the output of the chain.

When the last callback cb_n in the chain ch writes its output using this mechanism, a *publisher* thread $\mathscr{T}_{n,pub}$ is woken up that then publishes the data to its subscribers. For such a chain, we propose to execute this thread in a reservation server R_δ, as illustrated by Design-1 in Figure 4.4. We configure R_δ with a time slot sl_δ given by 4.3.2. Here, $\mathscr{T}_{n,pub}$ can be woken up any time in between the start and the end of R_γ but it cannot run until it gets the time slot sl_δ. Hence, the output of the chain is available to its subscribers only during sl_γ. That is, the end-to-end latency of the chain varies in between $\overline{L}_{PR} + \varepsilon$ and $\overline{L}_{PR} + \varepsilon + \overline{e}_{n,pub}$. In our experiments, we have observed that $\overline{e}_{n,pub} < 1$ ms which will result in a maximum variation of less than 1 ms. We term the above idea as *latency shaping*. Typically, in real-time systems theory, a task is either event-triggered or time-triggered. However, latency shaping implements a time- and event-triggered task to publish the output of the chain. That is, $\mathscr{T}_{n,pub}$ is triggered first when cb_n invokes an appropriate DDS API to publish data, and later, at the time instant when sl_δ starts, it is dispatched. Here, $\mathscr{T}_{n,pub}$ cannot run until both conditions are fulfilled.

Synchronous publish

Synchronous publishing offers precise control over the timing and flow of data transmission. It can enhance performance by obviating the need for asynchronous threads to be notified and awakened, which in turn saves on context switching resources. Additionally, it fosters a more deterministic behavior by ensuring that data transmission occurs directly within the context of the user thread, thereby allowing for more predictable timing and control. Moreover, synchronous publishing can lead to reduced latency, as it bypasses the overhead associated with asynchronous thread notification and the process of data transmission. Finally, it aids in avoiding blocking behavior, which is particularly advantageous when the history is fully occupied. This is achieved through improved management of QoS policies related to history management, ensuring smoother data handling and transmission [42].

Using this mechanism, cb_n can directly publish the output of the chain for its subscribers, i.e., the same thread $\mathscr{T}_n$ runs cb_n as well as publishes the data produced by cb_n. In this work, we study only static allocation of threads to reservation servers. Now, if we keep $\mathscr{T}_n$ in R_γ, we end up getting same jitters as with SCHED_FIFO and a high priority. Also, we can use two servers, R_γ and R_δ, as in Section 4.3.2, where $\mathscr{T}_1 - \mathscr{T}_{n-1}$ run using R_γ and $\mathscr{T}_n$ uses R_δ. Here, we can configure sl_γ based on the measured worst-case end-to-end latency of the sub-chain consisting $cb_1 - cb_{n-1}$ when run using SCHED_FIFO and a high priority, while sl_δ shall be longer than the measured WCET of cb_n. Also, sl_γ is immediately followed by sl_δ in time. We note that in this design, the variation in the execution time of cb_n still contributes to jitters in the chain, which may not be acceptable.

Definition 4.3.3. Synchronous Variation Control Server (R_δ)

R_δ is defined as a server with a time slot sl_δ : = [et_γ, et_$\gamma + \bar{e}_{sh}$], cycle c_δ:= c_γ and priority ρ_δ : =ρ_γ where et_γ marks the end of the time slot sl_γ in R_γ and $\bar{e}_{sh}$ is the maximum time required by cb_{sh} to execute the logic for publishing the output of cb_n.

Hence, we propose to add a *latency-shaping* subscriber callback cb_{sh} inside a ROS2 node N_{sh} at the end of the chain ch, as shown by Design-2 in Figure 4.4. We change the name of the output topic to which cb_n publishes. This is a trivial change which does not require modifying or recompiling the application code, which is a crucial consideration as the application sources are often not available to a systems (or timing) engineer in the industry. We can remap the published topic name for cb_n in the launch file. We denote the original topic name to which the chain publishes by T_n and the modified name by T_n^{mod}. We implement cb_{sh} to subscribe to and read data from T_n^{mod} and publish the same data to T_n, i.e., it republishes the output of the chain. This essentially extends the chain ch to $(cb_1, \ldots, cb_n, cb_{sh})$ considering that the end of a chain execution is identified when its output is available. Now, we run this extended chain using two reservation servers, R_γ and R_δ, that execute the sub-chain $(cb_1, \ldots, cb_n)$ and the callback cb_{sh}, respectively. While R_γ shall comprise a slot sl_γ as per Definition 4.3.1, R_δ shall be defined as per Definition 4.3.3. Considering that cb_{sh} only republishes a data item and does not perform any computations, $\bar{e}_{sh}$ should be short and impacts the worst-case end-to-end latency negligibly. Also, the time to republish should be fairly constant and, hence, the jitters are negligible.

4.3.3 Multiple latency bands

In our implementations of latency shaping in the previous section, we obtain one short band in which the end-to-end latency of the chain lies for each execution. However, we can easily produce more such bands by just re-configuring R_δ with multiple non-overlapping time slots, as shown by Design-3 in Figure 4.4. Let us denote the length of a time slot by Δ_δ which can be calculated based on the characteristics of the chain. For example, $\Delta_\delta = \overline{e}_{n,pub}$ if cb_n is publishing asynchronously. We can defind R_δ as per Definition 4.3.4 which means that we do not place a time slot where the chain will never publish its output. Also, depending on how cb_n publishes, we can configure sl_{δ,η_δ}. This ensures that the worst-case end-to-end latency does not change compared to our implementation with one latency band. With this modified definition of R_δ, the chain can publish its output only in one of the time slots, thereby producing at most η_δ latency bands. For example, in case of synchronous publish, if cb_{SH} is triggered (by a new available data) after $x_{\delta,\alpha}$ and before $x_{\delta,\alpha+1} - \Delta_\delta$ then the chain publishes its output in $sl_{\delta,\alpha+1}$.

Definition 4.3.4. Multi-latency Variation Control Server (R_δ)

R_δ is defined as a server with η_δ non-overlapping time slots $(sl_{\delta,1}, sl_{\delta,2}, \ldots, sl_{\delta,\eta_\delta})$, where $sl_{\delta,\alpha} = [x_{\delta,\alpha} - \Delta_\delta, x_{\delta,\alpha})$ considering $x_{\delta,1} - \Delta_\delta > \underline{L}_{PR}$, where $\underline{L}_{PR}$ is the measured best-case end-to-end latency of the chain ch when running using `SCHED_FIFO` and a high priority.

This is a very useful byproduct of our proposed mechanism, because it supports multi-mode implementation of ROS2 chains. That is, in each mode, the chain has a different end-to-end latency or sensing-to-actuation delay, and the control logic can be updated accordingly [111, 113]. This is particularly important in autonomous systems, where the timings of a computation chain are highly dependent on environment perception and in each scenario, a customized control algorithm can be used. For example, if there is a heavy traffic during city driving, the end-to-end latency of a chain from LIDAR/camera to steering and speed control might be long because of a long computation time in object detection and tracking. A control algorithm particularly optimized for such a long yet non-varying latency can be used in such an environment. We note that when a control model is developed to be robust to large variations in the sensing-to-actuation delay, it offers a lower performance. However, when the closed-loop model varies negligibly, a high-performing controller can be designed. If the modes can be switched safely, then the latency-shaped chain from sensing to actuation can offer a high overall control performance. To co-design a multi-mode controller and its implementation using latency shaping is a future work.

Further, in ROS2, a liveliness quality-of-service (QoS) [28] can be defined for a topic and when a data item in that topic becomes older than the specified duration (or QoS) it is not used any more for further processing. We note that our multi-band implementation of latency shaping is compatible with such QoS specification because we can place the bands in a way so that the QoS requirements are met. That is, the distance between two adjacent bands should be less than the maximum time for which a data stays alive.

4.4 Design Automation for Latency Shaping

4.4.1 Optimal placement of time slots

In our single-band implementation of latency shaping (in Section 4.3.2), the end-to-end latency in each execution is approximately equal to the worst-case value, thereby increasing the average end-to-end latency significantly compared to using *SCHED_FIFO* and a high priority. However, using our multi-band implementation of latency shaping, we can improve the average value compared to a single-band implementation. Let us assume that the design specification allows to have η_δ modes (or latency bands) in which the chain *ch* can operate. We can formulate a mathematical problem to place η_δ time slots in R_δ. In the process, our goal is to minimize the average end-to-end latency $\tilde{L}_{SH}$ of the chain using η_δ time slots. We choose to optimize $\tilde{L}_{SH}$ because it is often used as a performance measure by systems engineers in the industry.

Towards our goal, we need to profile the chain's execution with a high priority under *SCHED_FIFO* covering different scenarios. Let $L_{\mu,PR}$ be the end-to-end latency of a chain execution. Further, we consider a cumulative probability distribution function $\phi(\cdot)$ where $\phi(x)$ gives the measured probability that $L_{\mu,PR} \leq x$. We represent each latency band by just one value corresponding to the end of the time slot (e.g., $x_{\delta,\alpha}$ for $sl_{\delta,\alpha}$). The probability that $x_{\delta,\alpha-1} < L_{\mu,PR} \leq x_{\delta,\alpha}$ is $\phi(x_{\delta,\alpha}) - \phi(x_{\delta,\alpha-1})$, which is also approximately equal to the probability that the chain's output will be published in the time slot $sl_{\delta,\alpha}$. Hence, we derive an approximate expression to compute $\tilde{L}_{SH}$ as follows:

$$\tilde{L}_{SH} = \phi(x_{\delta,1}) \cdot x_{\delta,1} + \sum_{\alpha=2}^{\eta_\delta} [\phi(x_{\delta,\alpha}) - \phi(x_{\delta,\alpha-1})] \cdot x_{\delta,\alpha}. \tag{4.1}$$

We also consider constraints on placing the first slot (or $x_{\delta,1}$) and the last slot (x_{δ,η_δ}), as explained in Section 4.3.3. Further, we need to respect that any two consecutive slots are non-overlapping, i.e., $x_{\delta,\alpha-1} < x_{\delta,\alpha} - \Delta_\delta$. If necessary, we can also consider a liveliness constraint, e.g., $x_{\delta,\alpha} - x_{\delta,\alpha-1} \leq \Delta_{LV}$, where Δ_{LV} is the specified liveliness QoS for the output of the chain.

We study particle swarm optimization (PSO) [72] to solve the above constrained optimization problem and obtain the position of latency bands. We use PYSWARMS [87], a Python library, for PSO problem formulation and solution.

4.4.2 Automated implementation of latency shaping

We can also automate the implementation of latency shaping for a ROS2 computation chain using a tool flow, as shown in Figure 4.5. First of all, we have a *Chain Profiler* that collects traces related to the chain and measures its end-to-end latency—briefly explained in Section 3.4.2—when it is running with *SCHED_FIFO* and a high priority. Using the measured end-to-end latency values provided by the Profiler, *Optimizer* computes the position of the time slots for R_δ offline, as described in Section 4.4.1.

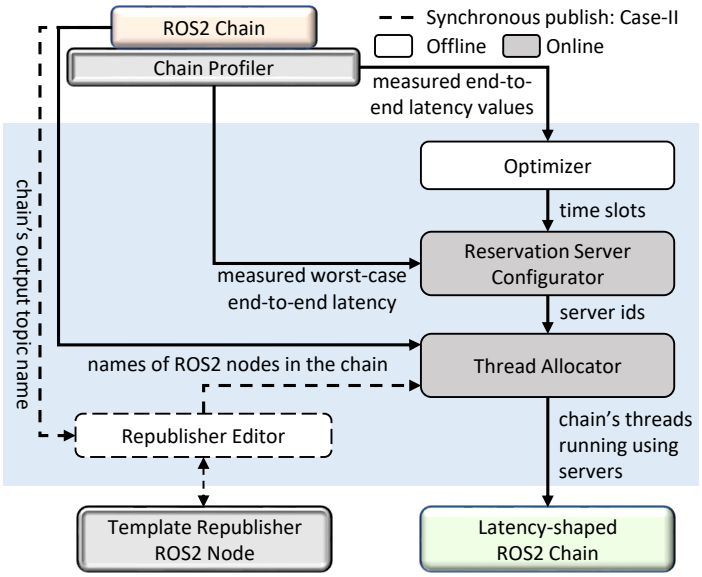


Figure 4.5: Tool flow for automated latency shaping.

For a chain that publishes its output synchronously, we need to create an additional ROS2 node with a latency shaping callback cb_{sh} extending the chain. For this, we have developed a template node N_{sh} with a callback cb_{sh} that subscribes to a topic and republishes a consumed data on another topic. We have developed a *Republisher Editor* that is implemented as a *bash* script to automatically make necessary modifications to extend the chain as follows: (i) It edits the launch file of the original application to remap the published topic name of cb_n, i.e., from T_n to T_n^{mod}. (ii) It also edits the launch file of N_{sh} and remaps subscribed and published topic names to T_n^{mod} and T_n, respectively, thereby architecturally extending the chain by cb_{sh}. (iii) It uses a ROS2 API—*ros2 topic type*—to get the data type typ_n of the output of the chain. From the obtained type, it can further derive the ROS2 package pkg_n and the header file $head_n$ containing the declaration of typ_n. (iv) It edits *CMakeLists.txt* and *package.xml* files in N_{sh} to link to pkg_n. (v) It edits the source code of N_{sh} to include $head_n$ and declare the type of subscribed and published topics as typ_n. (vi) It builds N_{sh} to generate the desired executable.

Further, we have written a *bash* script to implement a *Reservation Server Configurator* that creates and runs two reservation servers, R_γ and R_δ. The Configurator reads files to obtain (i) the measured worst-case end-to-end latency of the chain $\overline{L}_{PR}$ while creating R_γ as per Equation 4.4.1 and (ii) the time slots produced by the Optimizer while creating R_δ as described in Section 4.3.3. We have also implemented a *Thread Allocator* as a *bash* script that (i) first identifies the threads running the callbacks in the chain, i.e., $cb_1 - cb_n$, cb_{sh} or $\mathscr{T}_{n,pub}$ (if present) and (ii) then allocates them to R_γ and R_δ as per design. The Allocator uses Linux tools (i.e., *ps* with appropriate arguments) to identify a ROS2 node for a particular thread ID. It uses a LITMUS^{RT} API to allocate a thread to a reservation server. At the end, the components of the chain run using reservation servers following the latency shaping concept.

4.5 Case Studies

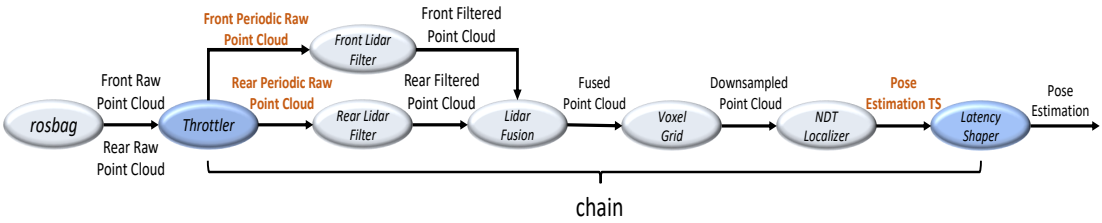


Figure 4.6: Lidar-enabled localization in Autonomous Valet Parking

4.5.1 Autonomous valet parking (AVP)

Experiment setup

We study a localization application in AVP. Here, we run the localization demo [13] provided by Autoware where the car starts from a predefined position, then drives to a specific parking spot, and finally exits the spot to return to the initial position. We run the demo in a workstation with AMD Ryzen Threadripper Pro 3955WX with 16 CPU cores at 3.7 GHz, 64 GB RAM, and an Nvidia RTX2080 GPU. As a software platform, we run ROS2 Foxy and Eclipse Cyclone DDS over Linux v5.4.1 patched with LITMUS^{RT}.

In this demo, a rosbag [73] plays back raw point cloud data recorded from two Lidars at the front and rear of a car. We observe that it does not publish data at regular intervals, but, data are available at almost 10 Hz. To ensure that the localization application runs once in every 100 ms, we have introduced a ROS2 node, Throttler, as shown in Figure 4.6. This node has two callbacks, cb_{rTh} and cb_{fTh}, where cb_{rTh} subscribes to the data from the rear Lidar and cb_{fTh} from the front Lidar. Here, cb_{rTh} (or cb_{fTH}) blocks a data item if it has not come after a certain time interval concerning the previous data, otherwise it republishes the data on a topic T_{rTh} (or T_{fTh}). We put the thread $\mathscr{T}_{Th}$ running these callbacks in a reservation server R_{Th} with a 2 ms time slot repeating every $p_{Th} = 100$ ms. Hence, cb_{rTh} (or cb_{fTh}) publishes data every 100 ms so we can consider it to be a timer callback that starts a chain following our Definition 3.1.1.

Table 4.2: Measured WCETs of callbacks in AVP [in ms]

cb_{rTh}	cb_{fTh}	cb_{rFl}	cb_{fFl}	cb_{fus}	cb_{vg}	cb_{loc}
0.6	0.6	19.9	30.7	3.3	9.2	50.8

Chain Definition

T_{rTh} (or T_{fTh}) is subscribed to by a callback cb_{rFl} (or cb_{fFl}) in a Rear Lidar Filter (or Front Lidar Filternode, as shown in Figure 4.6. Further, cb_{rFl} and cb_{fFl} publish the filtered point cloud data on topics, T_{rFl} and T_{fFl}, respectively. Lidar Fusion node runs a synchronization callback cb_{fus} that subscribes to both T_{rFl} and T_{fFl} and publishes the fused point cloud data on T_{fus} only when both subscribed data are available. Voxel Grid node runs a callback cb_{vg} that subscribes to T_{fus} and publishes the downsampled point cloud data on T_{ds}. Finally, the NDT Localizer node runs a callback cb_{loc} to estimate the car's position based on the subscribed data on T_{ds}. Here, cb_{loc} publishes the estimated pose on T_{pos}. In our experiments, we apply latency shaping to the chain ch^{rLoc} originating with the acquisition of the rear Lidar data, i.e., it is formed by cb_{rTh}, cb_{rFl}, cb_{fus}, cb_{vg}, and cb_{loc}. The output of the chain is published on T_{pos}.

Using SCHED_FIFO

First, we configure the threads running the callbacks cb_rFl, cb_{fus}, cb_{vg}, cb_{loc}, and cb_{fFl} to run with *SCHED_FIFO* and a high priority (i.e., 99). We bind $\mathscr{T}_{fFl}$ to run on Core-0 and $\mathscr{T}_{rFl}$, $\mathscr{T}_{fus}$, $\mathscr{T}_{vg}$, and $\mathscr{T}_{loc}$, on Core-1, which we maintain for all experiments with AVP. In both Core-0 and Core-1, we also run interfering load using timer callbacks, $cb_{int,0}$ and $cb_{int,1}$, in nodes, $N_{int,0}$ and $N_{int,1}$, respectively. These timer callbacks run every 10 ms and have a maximum execution time of 2 ms. We assign a lower priority (i.e., <99) to the threads $\mathscr{T}_{int,0}$ and $\mathscr{T}_{int,1}$. During the demo run, we measure—using the chain Profiler—the end-to-end latency for each execution of the chain ch and the maximum execution time (mWCET) of each callback. We have provided mWCETs over multiple runs in Table 4.2. We show the variations in the end-to-end latency in one run in Figure 4.9a. We note the average $\tilde{L}^{loc}_{PR}$ and the maximum end-to-end latency $\overline{L}^{loc}_{PR}$ as well as the jitters $\frac{\overline{L}^{loc}_{PR}-\underline{L}^{loc}_{PR}}{2}$ in Table 4.4 Row 1. We see that ch^{rLoc} experience large jitters, i.e., 25.9 ms, that we want to eliminate using latency shaping. These results help us to configure the reservation servers in the following experiments.

Table 4.3: Reserved time slots for running the callbacks in the chain [ms].

Row	Config.	$\mathscr{T}_{Th}$	$\mathscr{T}_{rFl}$	$\mathscr{T}_{fus}$	$\mathscr{T}_{vg}$	$\mathscr{T}_{loc}$
1	1 server per callback	[0,2]	[2,23]	[33,37]	[37,47]	[47,98]

		R_{Th}	R^{loc}_{γ}				R^{loc}_{δ}
Row	Config.	$\mathscr{T}_{Th}$	$\mathscr{T}_{rFl}$	$\mathscr{T}_{fus}$	$\mathscr{T}_{vg}$	$\mathscr{T}_{loc}$	$\mathscr{T}^{loc}_{sh}$
2	1 server per chain	[0,2]	[2,86]				—
3	1-band LS	[0,2)	[2,86]				[86,87]
4	Random 3-band LS	[0,2]	[2,86]				[50,51] [70,71] [86,87]
5	Optimized 3-band LS	[0,2]	[2,86]				[62,63] [77,78] [86,87]

Using conventional table-driven reservation

Next, we run the chain using table-driven reservation servers following the traditional time-triggered scheduling paradigm. We run each thread in $\{\mathscr{T}_{Th}, \mathscr{T}_{rFl}, \mathscr{T}_{fus}, \mathscr{T}_{vg}, \mathscr{T}_{loc}\}$ in a high-priority reservation server comprising one time slot per 100 ms, which are given in Table 4.3 Row 1. Here, we have configured the time slots based on the mWCETs of the callbacks in Table 4.2. We also run $\mathscr{T}_{fFl}$ using a high-priority reservation server R_{fFl} with a time slot [2, 33] ms. We note that the threads, $\mathscr{T}_{int,0}$ and $\mathscr{T}_{int,1}$, running the interfering callbacks, $cb_{int,0}$ and $cb_{int,1}$, are put in low-priority reservation servers, $R_{int,0}$ and $R_{int,1}$, respectively. We do not change the configuration R_{fFl}, $R_{int,0}$ and $R_{int,1}$ for the remaining experiments with AVP. Figure 4.9b shows the end-to-end latency variation for the chain using this configuration. In this experiment, we have obtained a maximum end-to-end latency of 95.7 ms (see Table 4.4 Rows 2), which is 13% longer than that obtained using SCHED_FIFO. This observation is similar to what we have in Section 4.1.2 for the synthetic chain. Further, we see that the jitters reduce from 25.9 ms with SCHED_FIFO to 24.3 ms with table-driven reservation, similar to our observation in Section 4.1.2. However, the reduction here is small because the last callback cb_{loc} contributes the most to the jitters in this chain, which cannot be eliminated when we follow the conventional use of table-driven reservation servers.

Table 4.4: Chain's timings performance with different schedule configurations [ms].

	End-to-end latency		
Config.	**Avg.**	**Max.**	**Jitters**
SCHED_FIFO	67.4	84.7	25.9
1 server per callback	72.8	95.7	24.3
1 server per chain	64.2	84.4	25.8
1-band LS	86.6	86.7	0.1
Random 3-band LS	76.4	86.6	≈0.1 per band
Optimized 3-band LS	71.3	86.5	≈0.1 per band

Using one reservation server (Section 4.3.1)

Now, we run the threads in $\{\mathscr{T}_{rFl}, \mathscr{T}_{fus}, \mathscr{T}_{vg}, \mathscr{T}_{loc}\}$ in one high-priority reservation server R_{γ}^{rloc}, while we still keep $\mathscr{T}_{Th}$ in R_{Th} so that it behaves as a timer callback. The slots in the reservation servers are given in Table 4.3 Row 2. We configure R_{γ}^{rloc} based on the measured maximum end-to-end latency of ch^{loc} with SCHED_FIFO. As expected, the measured maximum end-to-end latency (i.e., 84.4 ms) is similar to what we have obtained using SCHED_FIFO (i.e., 84.7 ms). Also, the jitters remain almost the same (i.e., 25.9 ms with SCHED_FIFO and 25.8 ms with this configuration). Further, Figure 4.9c shows the end-to-end latency variation for the chain.

Single-band latency shaping (LS) (Section 4.3.2)

In Autoware's implementation of NDT Localizer, Pose Estimate is published synchronously. Hence, we apply latency shaping by extending the chain with cb_{sh}^{loc} as shown in Figure 4.6. We allocate the threads running the extended chain using three reservation servers R_{Th}, R_{γ}^{loc}, and R_{δ}^{loc}, where R_{Th} runs $\mathscr{T}_{Th}$, R_{γ}^{loc} runs $\mathscr{T}_{rFl}$, $\mathscr{T}_{fus}$, $\mathscr{T}_{vg}$, and $\mathscr{T}_{loc}$, and R_{δ}^{loc} runs $\mathscr{T}_{sh}^{loc}$. The time slots in R_{Th}, R_{γ}^{loc}, and R_{δ}^{loc} are given in Table 4.3 Row 3. In Figure 4.9d, we see that the end-to-end latency of the chain under this configuration lies in a short band. We have a maximum end-to-end latency of 86.7 ms which is slightly longer than $\overline{L}_{PR}^{loc} = 84.7$ ms. This is because we have (i) added cb_{sh}^{loc} and (ii) set time slot boundaries in the order of ms. Also, we get jitters around 0.1 ms, which is significantly lower compared to what we got with SCHED_FIFO (i.e., 25.9 ms).

Multi-band latency shaping (Section 4.3.3 and 4.4.1)

Further, we reconfigure R_{δ}^{loc} with 2 additional time slots randomly placed, see Table 4.3 Row 4). Figure 4.9e shows that this reconfiguration produces three short bands where the end-to-end latency of the chain ch^{loc} lies. In Table 4.4 Row 5, we see that the average end-to-end latency has become 76.4 ms, which is 13% longer than what we got using SCHED_FIFO, while being 12% shorter than our single-band latency shaping. The maximum end-to-end latency and the jitters per band change negligibly with respect to our single-band implementation.

Now, we optimize the placement of two additional time slots, as explained in Section 4.4.1. The optimized time slots are provided in Table 4.3 Row 5 and the obtained latency bands are shown in Figure 4.9f. In Table 4.4, we see that average end-to-end latency reduces to 71.3 ms after optimization, i.e., we can improve it by 7% relative to random placements. Also, it is now only 6% longer than what we got using SCHED_FIFO. Even, if we do not consider multi-mode application design, the jitters become 12.5 ms using this optimized configuration, which is less than 50% of that obtained using SCHED_FIFO, while we increase the worst-case and the average end-to-end latency values by only 2% and 6%, respectively. This shows that *our proposed latency shaping has huge potential to improve the overall timing performance of real-world computation chains*. For the chain ch^{loc}, we further study how the average end-to-end latency reduces with more number of bands, as shown in Figure 4.7. In this particular case, we see that the average end-to-end latency changes negligibly beyond 4 bands.

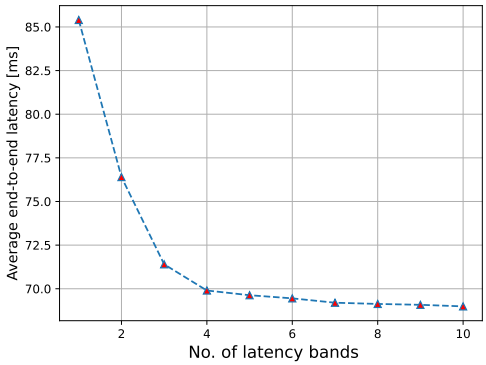


Figure 4.7: Optimized average latency vs number of bands.

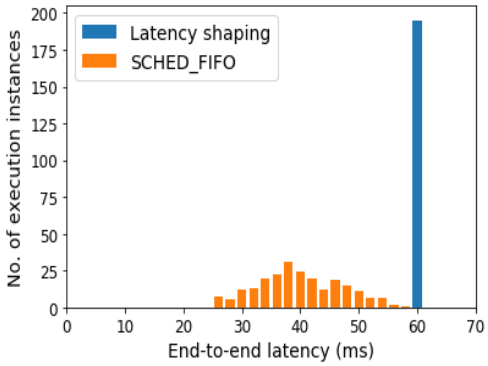


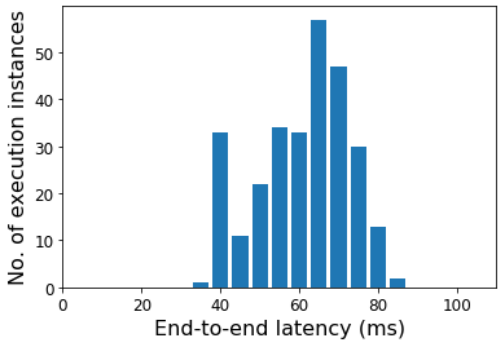
Figure 4.8: Latency variation for synthetic chain.

4.5.2 A synthetic case study

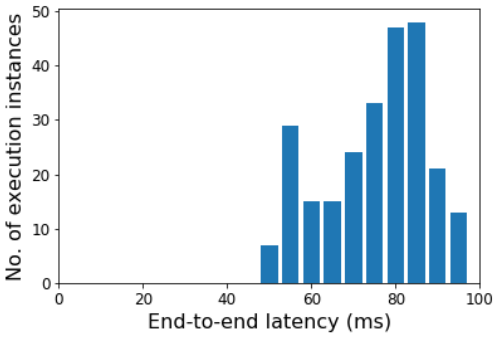
We have further developed a synthetic ROS2 chain ch^{syn} with five callbacks $cb_1 - cb_5$, similar to what is described in Section 4.1.2. However, in this experiment, we have implemented cb_5 to publish T_5 asynchronously. Here, we have used eProsima's Fast DDS instead of Eclipse Cyclone DDS as the former contains a full-fledged implementation enabling asynchronous publish. We allocate the threads running the callbacks in the chain, $\mathscr{T}_1 - \mathscr{T}_5$, as well as the publisher thread $\mathscr{T}_{5,pub}$ on the same processor Core-0. On the same core, we allocate an interfering timer callback cb_{int} in N_{int}, which is dispatched every 10 ms and has a maximum execution time of 2 ms. We create three reservation servers R_γ^{syn}, R_δ^{syn}, and R_{int} on Core-0 where R_γ^{syn} runs $\mathscr{T}_1 - \mathscr{T}_5$, R_δ^{syn} runs $\mathscr{T}_{5,pub}$, and R_{int} runs $\mathscr{T}_{int}$. Further, R_γ^{syn} and R_δ^{syn} are high-priority servers enabling latency shaping, while R_{int} is a low-priority server. Time slots for the reservation servers are defined as follows: $sl_\gamma^{syn} = [0,59]$ ms, $sl_\delta^{syn} = [59,60]$ ms, and $sl_{int} = [0,100]$ ms. Using this configuration, the end-to-end latency of ch^{syn} lies in a short band as shown in Figure 4.8 where the measured maximum value is 59.6 ms while the jitters are less than 0.1 ms. The figure also shows the variation in the end-to-end latency of ch^{syn} when it is scheduled using a high priority and SCHED_FIFO. In that case, we measure a maximum end-to-end latency of 58.5 ms and jitters of 16.5 ms. Again, latency shaping produces negligible jitters with a minimal increase ($\approx$2%) in the worst-case chain latency.

4.5.3 Comparison with LET

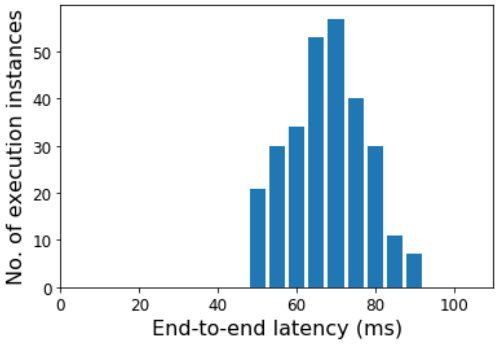
Using table-driven reservation servers, we can implement each callback following LET. (i) When each callback cb_i in the chain publishes asynchronously, we can put the corresponding publisher thread $\mathscr{T}_{i,pub}$ in a high-priority reservation server with a time slot placed at the beginning of each period. This will ensure that the DDS communication will take place at the beginning of the period and we can say that the response time of each callback is equal to its period. (ii) However, when a callback cb_i publishes its output synchronously, then we need to insert another callback $cb_{i,LET}$ in the chain similar to our latency-shaping callback cb_{sh} in Section 4.3.2. We can run $cb_{i,LET}$ using a reservation server with a time slot at the beginning of each period.



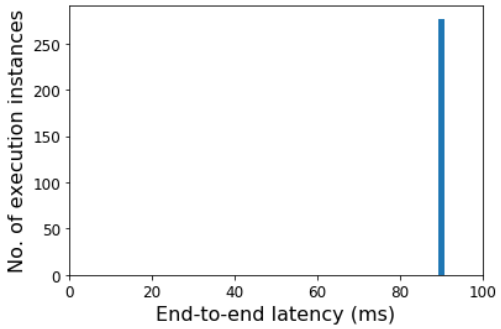
(a) SCHED_FIFO



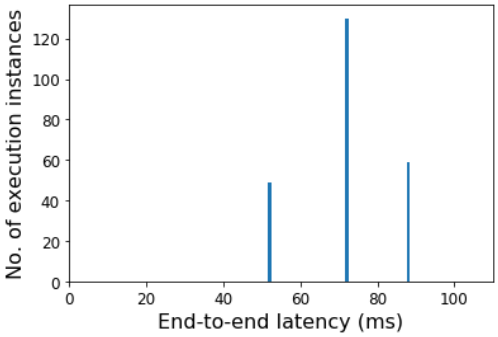
(b) One server per callback



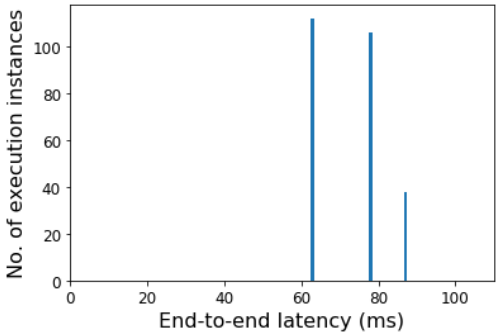
(c) One server per chain



(d) Single-band latency shaping



(e) Three random bands



(f) Three optimized bands

Figure 4.9: Latency variation in the chain under different schedule configurations.

Table 4.5: Comparison with LET.

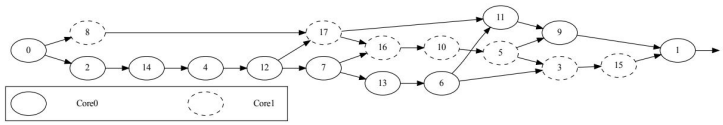
	Worst-case end-to-end latency [in ms]		
Chain	1-band shaping	LET per callback	LET per chain
Synthetic	59.2	500	100
AVP	86.7	simulation fails	100

We apply LET to each callback in a synthetic chain with five callbacks and a period of 100 ms—as described in Section 4.1.2. Here, each callback publishes synchronously. We put a maximum interfering load of 2 ms per 10 ms as described in Section 4.5.2. As expected, we get a fairly constant end-to-end latency around 500 ms.

When the end-to-end latency of a chain is less than the period of its execution, we can consider the whole chain as one task. Hence, using our latency shaping concept, we can apply LET for the whole chain by just configuring R_δ to comprise a time slot at the beginning of each period. In this case, we shall get a constant end-to-end latency of the chain that is equal to one period. Now, we apply LET to the aforementioned synthetic chain and get an end-to-end latency of 100 ms.

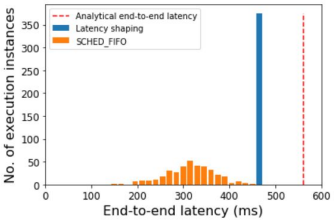
Further, we apply LET to the same chain in AVP as in Section 4.5.1 and get a fairly constant end-to-end latency around 100 ms, which is 15% longer than what we have obtained using latency shaping. We note that when we apply LET to each callback in this chain, the end-to-end latency becomes too long and the simulation does not run properly. In Table 4.5, we provide the end-to-end latency values obtained using our single-band latency shaping and the LET implementations for both synthetic and AVP chains. It demonstrates that we can flexibly keep the end-to-end latency shorter than a conventional LET implementation if the chained computations always finish within a time less than the period

Incorporated note to consider in the execution of autonomous systems is the potential for overshooting in executing chains, where the execution fails to complete before its deadline. This results in the remaining execution time propagating as a delay to subsequent executions, adversely impacting the overall system behavior by introducing unintended latencies and potential system instability. In our design, we mitigate the risk of overshooting by meticulously designing time servers based on the measured Worst-Case Execution Time of the callbacks, minimizing the likelihood of such scenarios. Nevertheless, to address any unexpected overshooting, one strategy involves discarding the execution of the overshooted callback to prevent the delay's propagation, utilizing functions like `ros::callbackQueue::removeByID()`. This method, however, necessitates the application being designed with an awareness of the underlying scheduler, incorporating feedback mechanisms from scheduler probes to dynamically adjust to execution anomalies. Future work could include additional experiments, employing synthetic cases to demonstrate effective management of overshooting scenarios, ensuring system robustness and reliability despite unforeseen delays.



Vertex	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
WCET [ms]	87	98	28	37	23	91	39	27	25	86	17	85	9	50	12	24	82	21

(a) DAG configuration obtained by applying [38].



(b) DAG's end-to-end latency.

Figure 4.10: Latency shaping of a DAG of ROS2 callbacks.

4.5.4 Latency shaping of DAGs

While we have explained our ideas concerning computation chains, they can be trivially extended for DAGs of callbacks. We consider partitioned scheduling for DAGs—each callback runs on a specific processor core—because LITMUS^{RT} offers only single-core servers. Towards latency shaping, we first apply a recent schedule optimization algorithm [126] on the input DAG. It is based on deep reinforcement learning and minimizes the number of cores required to meet the worst-case DAG timing requirement. In the process, it can compute the core assignment for callbacks while also adding more edges in the DAG (or precedence relations between callbacks).

Figure 4.10 shows a transitive reduction (for a clear visualization) of such an output DAG. Each of the 18 vertexes represents a callback. A vertex border gives the core assignment, i.e., we need two cores. This DAG executes every 764 ms. WCET of each callback is shown and we assume that the execution time varies uniformly between $0.5\times$WCET and WCET. Now, we have developed a synthetic ROS2 application with 18 nodes, each has one callback. Each callback publishes on a topic. Two outgoing edges from a vertex imply multiple subscribers of the topic. We implement vertex 0 as a timer callback (period 764 ms) while all other vertices are subscriber callbacks. We implement a vertex with two or more incoming edges as a synchronizing subscriber callback similar to cb_{fus} in Section 4.5.1. We put dummy computations on each callback based on the assumed execution time variation for it. We run this application following the core assignment in Figure 4.10a and using $\mathbb{S}_1$ with a high priority. We measure the DAG's end-to-end latency—the time between the start of callback 0 and when callback 1 publishes its output—and its variation is shown in Figure 4.10b. We observe a maximum value of 450 ms.

We configure two high-priority servers $R_{\gamma,0}$ and $R_{\gamma,1}$ to run on Core 0 and Core 1, respectively, while each comprises a slot $[0,450]$ ms. The threads running the callbacks in Core 0 (Core 1) are assigned to $R_{\gamma,0}$ ($R_{\gamma,1}$). The output topic name of callback 1 is remapped from T_1 to T_1^{mod}. We add a node N_{sh} with a callback cb_{sh} that subscribes to T_1^{mod} and publishes on T_1. We put another high-priority server R_δ on Core 1 with a time slot $[450,451)$. We map the thread running cb_{sh} to R_δ. Further, we run the latency-shaped DAG application and record an almost constant end-to-end latency, see Figure 4.10b. We note that its theoretic value is 560 ms (based on the WCETs), while our measurement-based approach produces ≈ 450 ms. This experiment shows that, we can perform latency shaping for multiple chains in a DAG while using existing algorithms to first co-optimize their theoretical latency values.

4.6 Conclusion

In conclusion, this chapter has presented a comprehensive discussion on the implementation of latency shaping in cyber-physical systems , particularly focusing on ROS2 computation chains. The proposed approach, characterized by the use of table-driven reservation servers, significantly mitigates timing jitters between sensing and actuation, thus ensuring negligible jitters and maintaining a constant end-to-end latency that surpasses traditional LET implementations. Notably, our latency shaping strategy does not necessitate modifications or recompilations of the application code or ROS2 libraries, upholding the industry's requirement for separation between application development and timing engineering. By leveraging real-world benchmarks, such as Lidar-based localization, our methodology not only demonstrates a 13% reduction in end-to-end latency compared to enhanced LET implementations but also supports multi-mode application design for high-performance operations in varied environments. Furthermore, the practicality of our approach is underscored by its reliance on profiling results rather than analytical frameworks, offering a versatile solution that enables co-optimization of jitters and average end-to-end latency. This paves the way for a more efficient and reliable implementation of autonomous CPS, such as self-driving cars, enhancing their performance in both city and highway driving scenarios.

Chapter 5

Deterministic Distributed Communication

The advancement of autonomous driving technologies highlights the indispensable role of distributed architectures. These systems are pivotal for harnessing the capabilities of AI-based applications, facilitating real-time decision-making, and ensuring the safety and efficiency of autonomous vehicles navigating through complex environments. A key feature of these architectures is their ability to distribute computational tasks across a network of interconnected platforms, each powered by System on Chips (SoCs). SoCs, notable for their compact, power-efficient design, are instrumental in this technological evolution, facilitating the execution of sophisticated computational tasks directly within the vehicle to minimize latency and improve response times to environmental stimuli.

A pivotal aspect that enhances the functionality of these distributed systems is the adaption of real-time deterministic communication. This type of communication is critical for ensuring the timely and reliable exchange of data among the distributed components of an autonomous vehicle, which is imperative for applications demanding high precision and timing. The deterministic nature of this communication means that data is transmitted and received with guaranteed timing, a necessity for the coordination and safety of autonomous vehicles, ensuring that decisions are made based on the most current and precise data, thereby allowing vehicles to adapt to their environment with enhanced agility.

Furthermore, the importance of real-time deterministic communication extends to distributed middleware frameworks, like ROS2, which play a critical role in managing communication and data exchange between distributed systems. Deterministic communication protocols are vital for these frameworks, enabling them to support the complex, time-sensitive tasks required by autonomous driving technologies and other sophisticated robotic applications. This is particularly relevant for ADAS applications, which rely heavily on deep neural networks (DNNs) for functions such as planning and control, environment perception, and sensor fusion. These DNNs, being computationally intensive, often necessitate custom hardware accelerators for expedited processing, crucial for meeting the strict timing requirements of ADAS applications, where end-to-end latency is constrained to human reaction times of approximately 390 – 600 ms [137].

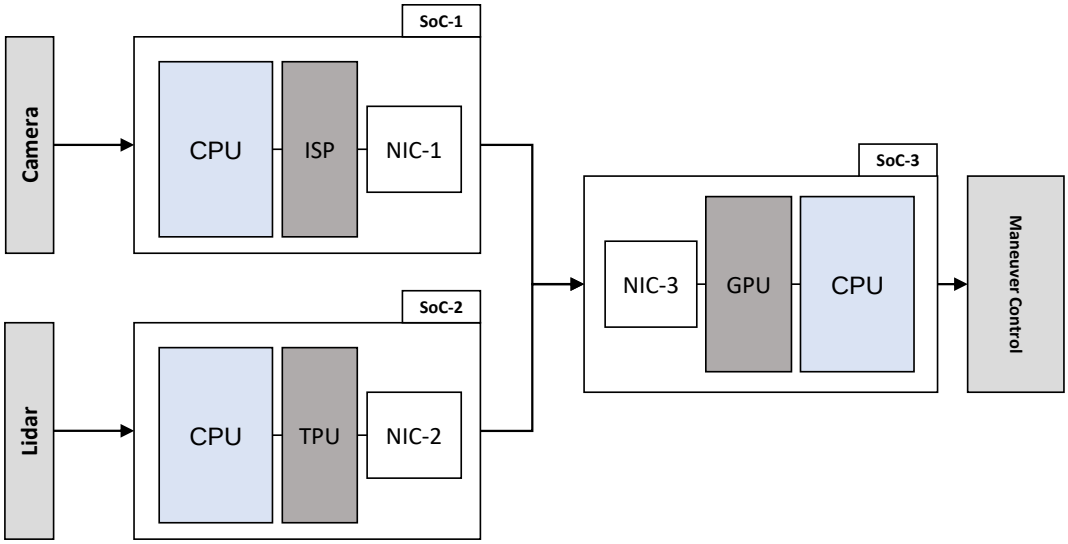


Figure 5.1: Distributed platform architecture of ADAS application

.

The communication delays between accelerators, alongside the computational demands of DNNs, significantly influence the end-to-end latency of these applications. Given the high data transfer volumes between accelerators, which can surpass tens of megabytes to accurately represent the three-dimensional environment [27], and the potential for network interference and contention, this paper underscores the significance of communication technologies that fulfill the low latency and deterministic timing requirements of ADAS applications.

5.1 Data flow in current automotive platforms

The guarantee of meeting timing requirements for applications on modern architectures is complex and challenging as it requires demonstrating timing predictability, correctness, and determinism. A system is considered timing predictable if it can be proven during design time to satisfy the specified timing requirements during execution. These requirements include end-to-end latency and execution time constraints on the application flow.

Figure 5.1 shows an example ADAS application for pedestrian detection, inspired from [58]. It uses a Lidar point cloud data stream for occupancy grid generation. The camera frames are used for object classification, e.g., to distinguish a pedestrian from other objects. Here, the camera data is processed on SoC-1 using an Image Signal Processor (), while the point cloud data from Lidar is processed in a Tensor Processing Unit () on SoC-2. The planning and control decisions are then taken on SoC-3 with the aid of a Graphics Processing Unit (GPU). Hence, the data produced by and must be sent to the GPU in SoC-3.

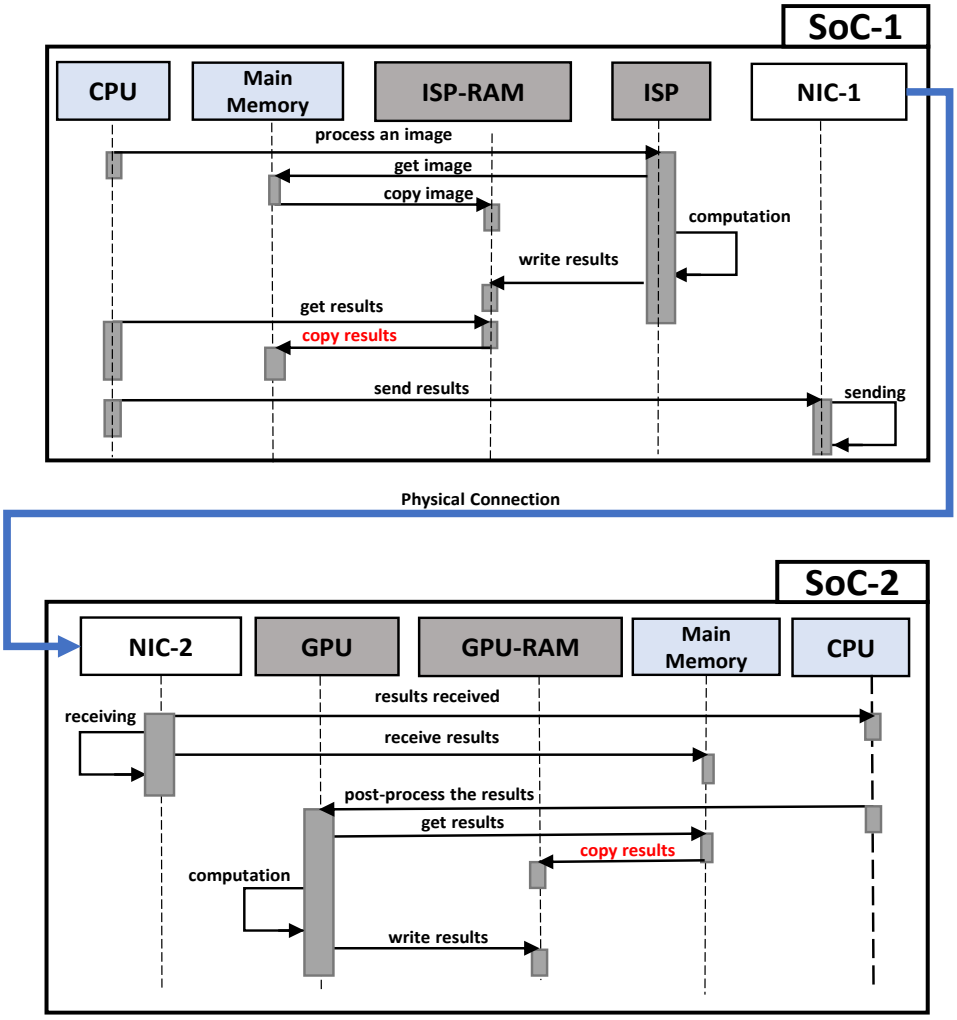


Figure 5.2: Sequence diagram showing RDMA-supported data transfer between accelerators

Two major factors increase the latency of such data communication between different SoCs [121]:

- The network access requests from an accelerator to the network interface card (NIC) is performed via the CPU on the same SoC as shown by the sequence diagram in Figure 5.1b. Processing of such requests consumes millions of CPU cycles for megabytes of data transfer. Also, other workloads running on the CPU can interfere with such requests potentially causing additional delays in the data transfer.
- Transferring data between accelerators via CPUs involve unnecessary memory copies as illustrated in Figure 5.2, which again delays the communication.

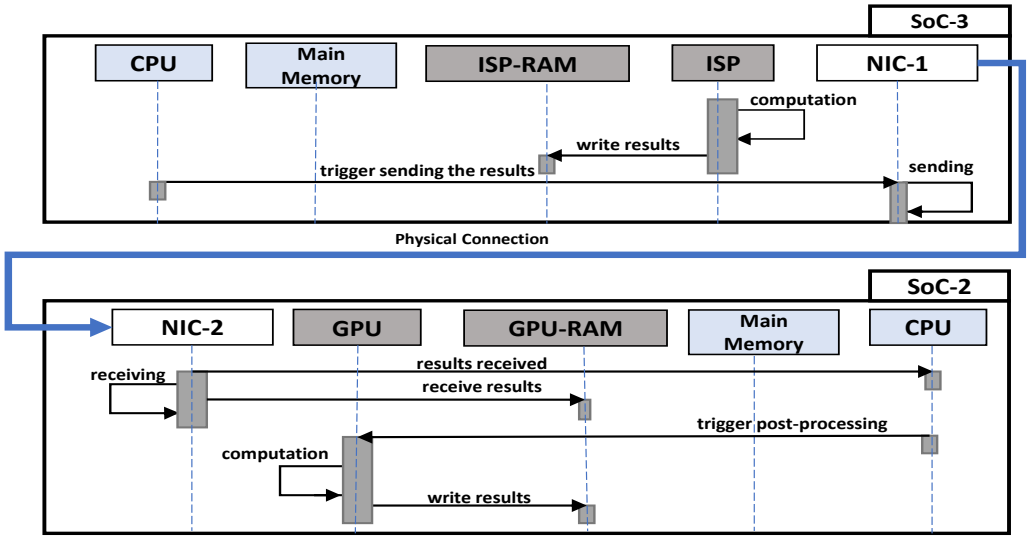


Figure 5.3: Sequence diagram for RDMA-supported data transfer between accelerators

To improve the inter-SoC communication latency, CPU involvement, and memory copies need to be reduced as much as possible. Remote Direct Memory Access (RDMA) [102] is a technology that can enable direct access of the memory in one SoC (e.g., ISP's RAM) by a processing unit in another SoC (e.g., GPU in SoC-3). RDMA is normally implemented at the hardware level (e.g., integrated into a NIC) with limited extension possibilities. Instead, in this chapter, we focus on Soft-RoCE, a software implementation of RDMA over a converged Ethernet (RoCE) network [130]. Soft-RoCE is compatible with any Ethernet NIC since it does not use hardware acceleration. Hence, it allows RDMA technology to be integrated in a scalable, portable, and hardware-independent manner. With Soft-RoCE, the CPU can execute other workloads, while, on the same SoC, data is being read by a remote accelerator. Thus, the CPU is free to manage other computations while the data transfer is achieved with low latency, low CPU load, and high bandwidth. Also, note that it significantly reduces memory copies as it allows the Network Interface Card (NIC) to transfer the data directly to and from the application memory, eliminating the need to copy the data between the network stack and the application memory. This chapter advocates the use of RDMA technology for automotive applications, in particular, data-intensive AD/ADAS applications, because it reduces the latency of transmitting a large amount of data between distributed SoCs compared to communication protocols like Transmission Control Protocol (TCP) [71]. Figure 5.3 depicts the updated sequence diagram for inter-SoC data transfer in the pedestrian detection application (from Figure 5.1). Data produced by the ISP on SoC-1 (or TPU on SoC-2) is directly read by the GPU on SoC-3 with minimal involvement of the CPU on SoC-1 (or the CPU on SoC-2). In Figure 5.3, it is clear that data in the RAM of the producer accelerators can be directly read by the consumer accelerator bypassing the CPUs in the SoCs. The data is copied directly between the accelerators and the NIC, unloading the CPU and reducing its utilization.

5.1.1 Effect of CPU Centric Architecture

In the contemporary landscape of SoC design, architectures often prioritize CPU as the primary processing entity, with accelerators and other processing units playing secondary roles. This CPU-centric approach inherently shapes the computation and communication flows within the system, potentially influencing the overall performance and efficiency, especially under conditions of high demand or contention. The objective of this study was to methodically explore the ramifications of such a CPU-centric architecture on the overall system dynamics, focusing on how CPU performance and contention impact both CPU-bound tasks and those executed by secondary accelerators.

To dissect the influence of CPU-centric architectures, a targeted experiment was conducted using "isolbench," [131] a benchmark specifically designed to induce memory contention by generating direct memory access (DRAM) requests that lead to cache misses. This setup aimed to simulate realistic scenarios of high contention, reflecting the challenges faced in actual system operation. The experiment revolved around pedestrian detection functionality, a critical component in autonomous driving systems, which relies on a synchronized operation among various SoC coresincluding control cores, data cores, an AI Core, and ISP. The pedestrian detection function was initially allocated to Core 0, with subsequent iterative mappings of isolbench across all SoC CPU cores, from Core 0 to Core 11. This approach allowed for a systematic increase in load and an observation of its consequent effects on both CPU-bound and accelerator-driven tasks.

The experiment yielded insightful data on the impact of CPU-centric architecture on system performance:

- Dramatic Increase in CPU-bound Task Latency: The post-processing functions, reliant on CPU performance, exhibited a latency increase of nearly 600%. This marked degradation directly corresponded with the elevated levels of memory contention induced by the benchmark, highlighting the vulnerability of CPU-centric tasks to system stressors.

- Significant Slowdown in Accelerator-Driven Tasks: Contrary to expectations that CPU-independent accelerators might remain unaffected, the AI Core and ISP demonstrated a notable slowdown, with execution times doubling (approximately 100% increase). This outcome challenges the premise of full independence of these accelerators from CPU-induced contention, indicating a more intertwined relationship between the CPU and other system components than previously acknowledged.

- The study also unveiled a high degree of variability in system performance in response to different levels of isolbench contention. This variability, manifesting as non-deterministic execution times, suggests complex interdependencies within the SoC that are affected by CPU load and memory contention.

.

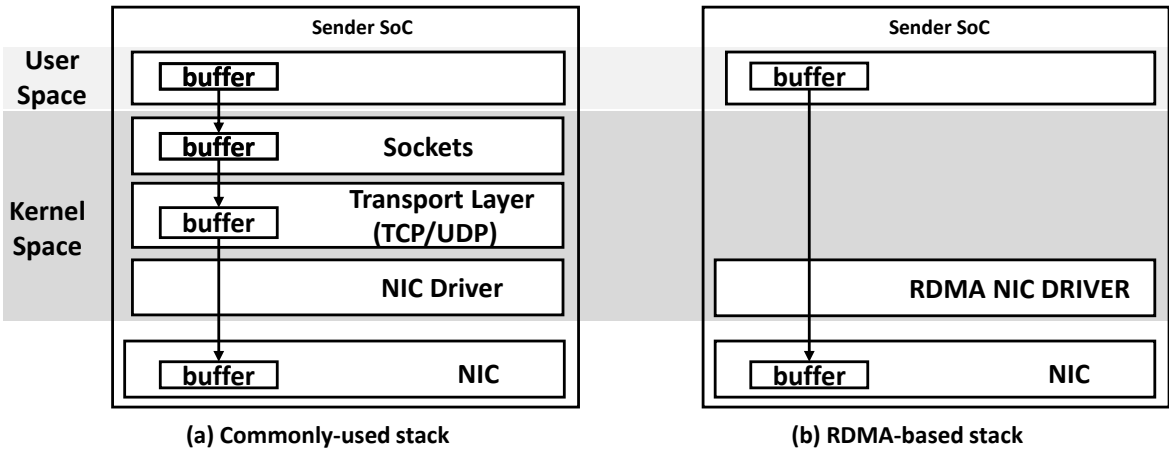


Figure 5.4: Data transfer with commonly-used stack and RDMA-based stack.

5.2 Remote Direct Memory Access (RDMA)

RDMA can reduce communication latency for distributed software applications implemented over multiple heterogeneous SoCs. Figure 5.4 illustrates the main reasons for reduced latency by comparing RDMA against commonly-used communication stacks involving TCP/UDP. In Figure 5.4a, we see that multiple interventions of the operating system in the CPU are necessary to transfer data through different layers of the conventional communication stack. Also, in the process, data is copied in the buffers of different layers (e.g., in sockets and transport layer) using system calls (e.g., *memcpy()*). Conversely, Figure 5.4b shows that RDMA sending operation (same will be on the receiving side) bypasses the kernel almost entirely and efficiently transfers the data between the application buffer and the RDMA-supported NIC [90]. Note that this transfer does not involve any system call and is accomplished using direct memory access (DMA) channels. Therefore, we can say that RDMA allows direct access to the application memory (i.e., without involving the operating system and the CPUs on both sending and receiving sides) by remote devices connected in the same network.

RDMA can provide significant benefits to automotive applications that use distributed SoCs. It enables the transfer of data between devices with minimal CPU involvement, reducing latency and improving overall system performance. The most important feature offered by RDMA is that it enables direct access to the application memory for the devices connected by the same network. As shown in Figure 5.4a, the RDMA stack is different from the conventional network stacks. The latter depends on the intervention of the operating system to transfer data from an application's virtual buffer (user space) to the NIC. Through the transfer, data is copied from the application to the sockets layer and then to the transport layer before finally being sent to the NIC. On the other side, the RDMA sending and receiving operations bypass the kernel and efficiently transfer the data between the application buffers and the RDMA-supported NICs [90].

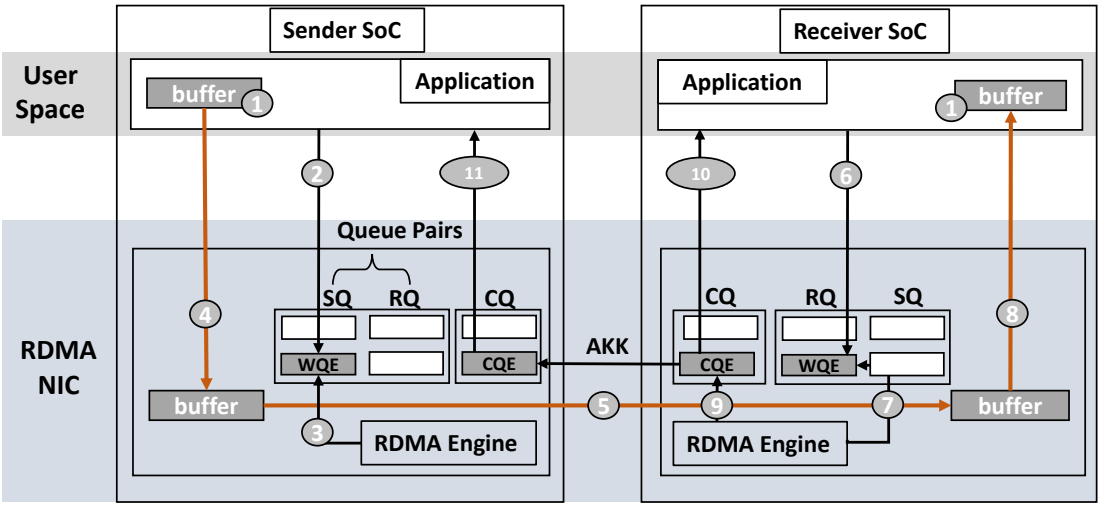


Figure 5.5: Different components in an RDMA connection.

5.2.1 Communication using an RDMA connection

Figure 5.5 shows the different components in an RDMA connection and how communication is carried out using them. RDMA communication typically requires a NIC that implements RDMA engine, also called Host Channel Adapter (HCA). In particular, HCA includes all the necessary logic to implement the RDMA protocol. The HCA is placed on a Peripheral Component Interconnect express (PCIe) slot on the SoC and, hence, it can use DMA. To establish an RDMA connection between two SoCs, it is necessary to first reserve and map (or *pin*) memory buffers on the sender and receiver sides and inform the kernel that the registered memory will be used for RDMA communication (1). During the initialization of an RDMA connection, HCA registers are mapped on the memory using which application can directly invoke RDMA transfers, i.e., a fast path is created between the application and the HCA bypassing the kernel. That is, a pair of work queues, called a *Queue Pair* (QP), are generated for communication scheduling on the HCAs at both sending and receiving sides. A QP consists of a *Send Queue* (SQ) and a *Receive Queue* (RQ). Besides the QP, a *Completion Queue* (CQ) is generated to track the completion of a scheduling instruction, also called a *Work Queue Element* (WQE), residing on either of the work queues. The primary content of a WQE is a pointer to the target buffer. In SQ, a WQE contains a pointer to the data that needs to be sent, while in RQ, the pointer in a WQE addresses the buffer where the incoming data has to be placed.

When an application initiates an RDMA send operation, a WQE is created and placed on the SQ in the HCA (2). The HCA polls the QP and, hence, gets the WQE (3). Once the HCA gets a WQE, it processes the WQE and fetches the data from the memory region specified in the WQE to the HCA buffer using DMA (4). The HCA then creates and sends a data packet comprising the data, the SoC address, the RDMA connection identifier, among other information (5).

Simultaneously, at the receiving side, the application creates and places a WQE on the RQ in the HCA (6). Now, when the receiver HCA receives a data packet and identifies the RDMA connection, it checks the corresponding RQ for a WQE (7). If a WQE is available, the HCA puts the data in the memory region specified in the WQE using DMA, otherwise, it rejects the packet (8). If the data transfer is completed successfully, the HCA will put a completion queue element (CQE) on the CQ (9). The application polls the CQ to check if the data is received so that it can continue processing the data (10).

On successful completion, the receiver HCA also sends an acknowledgement to the sender HCA. Once the sender HCA receives the acknowledgement, it also puts a CQE on the CQ at the sending side. As the application polls the CQ also at the sender side, it gets notified that the data is sent successfully (11).

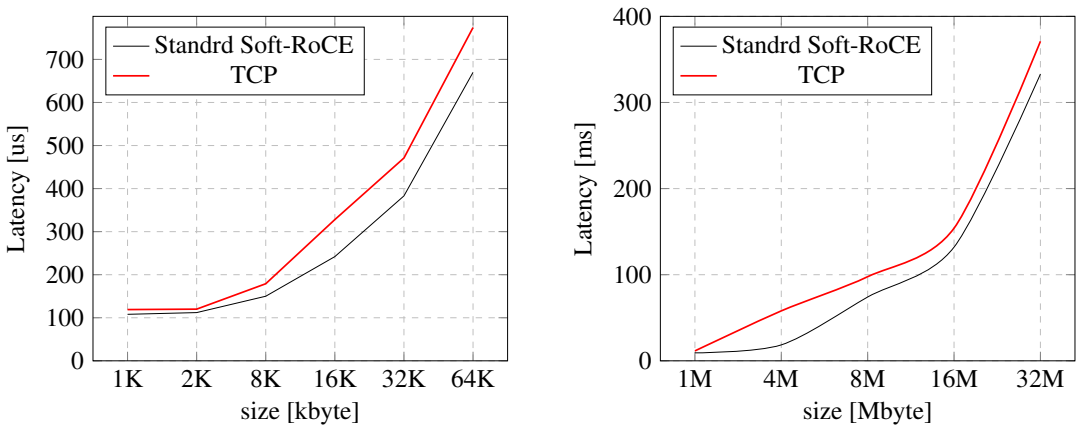
5.2.2 RDMA over Converged Ethernet (RoCE)

RDMA semantics of InfiniBand was adapted to run over Ethernet and the corresponding specification (RoCE version 1 or RoCE_v1) was released by InfiniBand Trade Association (IBTA) in April 2010 [67]. RoCE_v1 uses standard Ethernet-based services at the data link layer. RoCE_v1 uses Layer 2 (L2) information and supports packet routing only within an L2 subnet. Later, in 2014, IBTA revised RoCE_v1 and released RoCE version 2 (RoCE_-v2) that supports routing of data packets on the network layer [68]. Packet routing across different sub-networks is possible because RoCE_v2 uses Layer 3 (L3) information. A global routing header (GRH) is used by the network layer to route RoCE_v2 data packets, which is similar to IPv6 addressing.

Typically, RDMA transfer are carried out using a dedicated hardware RDMA engine, as explained in Section 5.2.1. This increases the dependencies on external hardware and the associated proprietary software. However, these dependencies can be avoided by using Soft-RoCE. Soft-RoCE is a complete software implementation of the RDMA principles that makes RoCE_v2 protocol available for any Ethernet-based network interconnect [130]. It is an open-source Github community project, with contributions from IBM, Mellanox and System Fabric Works and its implementation is available as a Linux kernel module. Soft-RoCE avoids system calls and enables zero copy on the sending side, while it needs only one copy on the receiving side. The reason for this one copy is that the RDMA connection has to be identified for the received data before it can be copied to the corresponding pinned memory buffer. The performance of Soft-RoCE is comparable to RoCE_v2 [130] while it offers more flexibility and allows a complete RDMA implementation over any NIC. Soft-RoCE enables more efficient data transfers compared to the default Ethernet protocol stack, as shown in Section 5.2.3.

5.2.3 Low latency communication using Soft-RoCE

To quantitatively assess the benefits of using RDMA with respect to the standard communication stack, we perform experiments where we transfer data between two Intel x86_64



(a) smaller data sizes

(b) larger data sizes

Figure 5.6: Soft-RoCE vs TCP in terms of communication latency.

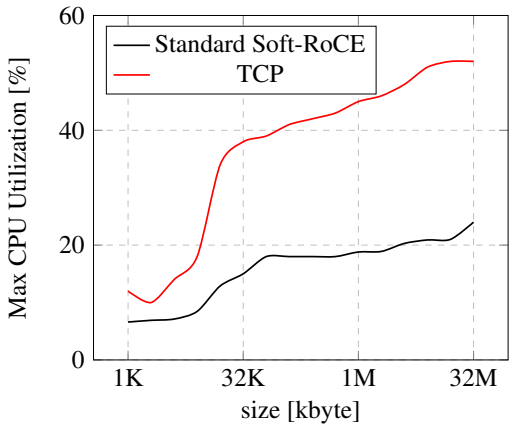


Figure 5.7: Maximum CPU utilization with Soft-RoCE and TCP.

processors running Linux version 5.13.0-51-generic as the operating system. We compare communication latency for the default communication stack (which uses TCP to transfer data) and Soft-RoCE. We use qperf [99] to measure the communication latency. We vary exponentially the data size from 1 Kbyte to 32 MB and for each data size, we perform 150 data transfers. Figure 5.6 shows the average communication latency with TCP and Soft-RoCE for different data sizes. We can clearly see that Soft-RoCE performs better than TCP. In specific cases the communication latency can be reduced by 36%, e.g., for 4 MB data, Soft-RoCE offers a latency of 37 ms while it is 58 ms with TCP.

We also measure the maximum CPU utilization on the sender side for each data size across all runs both with TCP and Soft-RoCE. The results are shown in Figure 5.7. We can see that the maximum CPU utilization is much lower with Soft-RoCE in comparison to TCP. For more than 1 MB data, Soft-RoCE can reduce the absolute value by more than 25%, e.g., the max CPU utilization with Soft-RoCE is 20.3% for 4 MB data while it is 48.1% with TCP. These results emphasizes the fact the Soft-RoCE can effectively reduce the CPU load and the CPU can use this time to execute other workloads.

5.3 Nondeterministic behavior of Soft-RoCE

Considering that multiple applications are running simultaneously on a many-SoC automotive platform, several of them may want to send data over the same physical network link. For such scenarios, multiple RDMA connections can be created between two SoCs [145]. We have experimentally verified that such implementations can also be accomplished using Soft-RoCE. In such implementations, each RDMA connection can be used by an application to transfer specific data (or a series of data produced by the same periodic/sporadic task). For example, in the pedestrian detection application , if we have both ISP and TPU on SoC-1 and the GPU on SoC-2, the results of ISP and TPU can be sent to the GPU using two different RDMA connections created between SoC-1 and SoC-2. Each RDMA connection is associated with its own pinned memory buffer, data channels, and queues (discussed in Section 5.2).

While using multiple RDMA connections between two SoCs, we have identified the following major drawbacks that prevent the usage of Soft-RoCE for AD/ADAS applications: (i) When two time-critical applications use different RDMA connections to send their data, we have observed that data packets are transferred in an arbitrary order to the receiver SoC. In certain cases, one application might have to wait for an arbitrarily long time before its data is transferred. In AD/ADAS applications, a deterministic ordering of packets, e.g., first-in first out (FIFO), enable performing a worst-case analysis to determine an upper bound on the communication delay [50], which helps to provide performance guarantees. (ii) When a time-critical AD/ADAS application sends data in parallel with a best-effort application, there is no guarantee that the former will be prioritized for RDMA communication. In particular, we have observed that Soft-RoCE does not have any notion of priority for communication scheduling. This means that a critical data packet might be delayed by a nondeterministic amount of time while a non-critical data packet is transferred by Soft-RoCE, which is again not desirable.

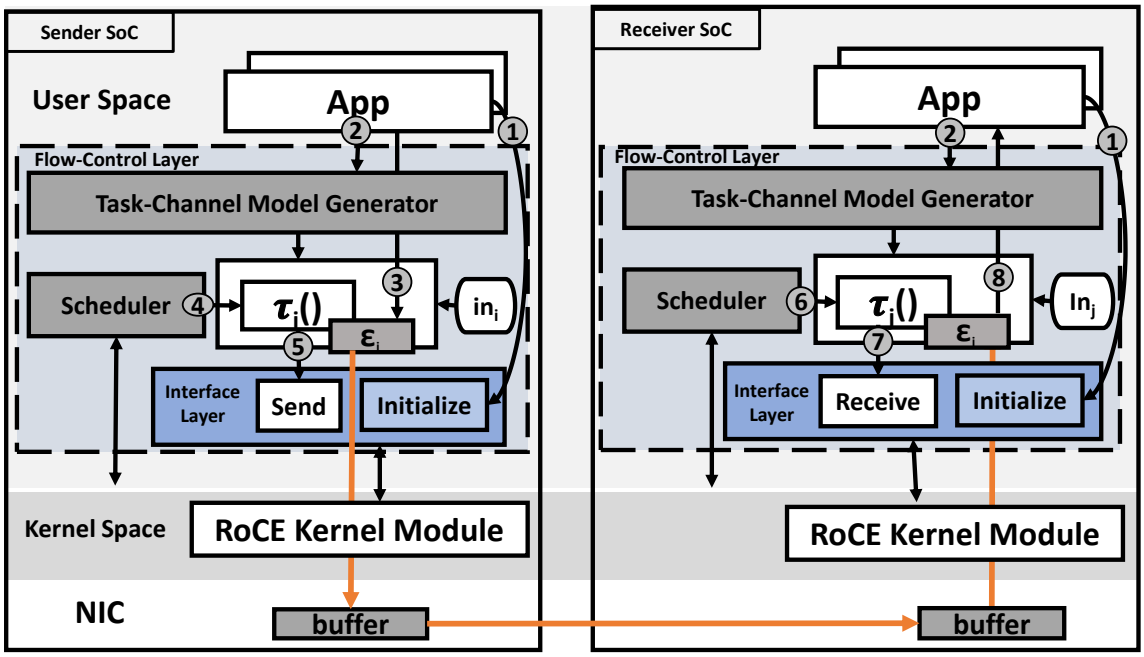


Figure 5.8: Deterministic Soft-RoCE communication stack.

5.4 RDMA-Based ADAS Communication Stack

In this section, we describe the proposed multi-layer communication architecture and its initial implementation, as shown in Figure 5.8. In particular, we add two upper layers over a default Soft-RoCE implementation.

- *Flow control layer* that manages RDMA operations performed by the applications. In particular, we can add different scheduling policies to send and receive data in this layer.
- *Interface layer* that introduces C++ wrapper APIs to facilitate the integration of Soft-RoCE with distributed computation applications across heterogeneous architectures. The flow control layer uses easy-to-use APIs provided by this layer to carry out RDMA communication. This layer wraps the APIs in Libibverbs which is a user-space library comprising 36 or more ibverbs (Infiniband verbs) to interact with the Soft-RoCE kernel module.

The design goal of our proposed solution is to provide determinism and guarantees without modifying the SoftRoCE underlying implementation. The design manages RDMA operation requests from the application layer and converts them into *discrete tasks* that can be controlled and prioritized based on the computation flow end-to-end timing requirements. By transforming the operations into controllable tasks, Deterministic SoftRoCE enhances the predictability of RDMA performance, and enables control over the timing and sequencing of RDMA operations, leading to improved predictability and reduced latency for tasks in the computation flow. This makes it well-suited for distributed computation real-time applications that require deterministic performance and timely response.

5.4.1 Interface layer

This layer comprises a user space C++ library that provides three simple APIs, namely *Initialize*, *Send*, and *Recieve*.

1. `Initialize` API is invoked to set up an RDMA connection. It initializes the state of different RDMA components, e.g., QP and CQ states. It registers the memory to be used for communication. It defines the connection parameters (e.g., local and remote IP addresses) to identify the remote side and establish a connection with it. The arguments to this API are a pointer to the memory area to be registered and the IP address of the remote SoC with the associated port number. Note that `Initialize` API has to be invoked on both the sending and receiving sides.

2. `Send` API is invoked on an SoC to send data. It creates and posts a work request on the SQ of the QP. it defines an error object using which details of the error can be propagated to the application level on an unsuccessful completion of the work request. The arguments to *Send* API are (i) the IP address of the receiver SoC and (ii) the size of and the pointer to the data to be transferred.

3. For a successful communication, the receiver SoC also needs to invoke the `Receive` API to receive the data. Similar to the `Send` API, it creates and posts a work request on the SQ of the QP and defines an error object to notify the application of any error in the communication. The arguments to `Receive` API are the IP address of the sender SoC, the size of the data to be received and the pointer to the memory where the data will be stored.

The aforementioned APIs help the application developer to integrate Soft-RoCE in their application without the need to be acquainted with the ibverbs and the associated challenges to use them appropriately for efficient communication, e.g., maintaining QP states, creating work requests, and defining memory protection domains. Each API in this layer encapsulates several ibverbs[1] and provides simple interfaces that any application developer can interpret correctly, e.g., pointers to data, data sizes, and IP addresses.

[1] ibverb details ommitted for the sake of readability and space constraint

5.4.2 Flow control layer

The main function of this layer is to schedule work requests across different RDMA connections according to a desired policy. To realize this functionality, we have used Tasking Framework [57] which is an open-source multi-threading platform based on the Task/Channel model introduced in [47]. It provides abstract classes with virtual methods to create applications as DAGs of *tasks* (or functionalities) and *channels* (or message queues between communicating tasks).

In the context of RDMA communication, we use (i) channels to implement RDMA buffers and (ii) tasks to implement RDMA send and receive operations. Further, the activation of a task (τ_i) can be controlled by configuring the task input(s) (in_i). The flow control layer offers *time-driven* and *event-driven* activation of tasks. In a time-driven activation, the task invoking an RDMA send/receive operation is triggered by a periodical signal generated a timer. Hence, we can control the rate at which data is sent/received irrespective of the rate at which data is produced. On the other hand, RDMA operation can also be carried out (or the corresponding task can be activated) in response to an event, e.g., data is pushed into the channel by the application. Using such an event-driven activation, data can also be sent/received sporadically. Both time- and event-driven activation are useful in AD/ADAS applications. Also, when multiple inputs are defined for a task, they can be combined, e.g., by *AND* or *OR* operation, to activate the task. That is, a task can be triggered when either of the input signals is available or it can be triggered only when all input signals have arrived. The above task activation options in the flow control layer allow our proposed stack to be used for applications with different timing characteristics and requirements.

Using our proposed stack, a number of threads are created to carry out different RDMA operations. These threads can be scheduled according to a policy. In the flow control layer, we have implemented a static fixed-priority scheduler. Note that when tasks have equal priority, the scheduler dispatches them using a FIFO policy. Unlike the default implementation of Soft-RoCE, using a real-time scheduling policy (as described above) our proposed stack can guarantee (i) in-order packet delivery for applications with equal priority and (ii) lower worst-case and average latency for high-priority data packets.

Figure 5.8 shows how (i) applications interact with the flow control layer and (ii) the flow control layer uses the APIs in the interface layer. After an RDMA connection is established using the *initialize* API of the interface layer (1), the application have to invoke the task/channel generator of the flow control layer both at sending and receiving sides to define theirs channel(s) (ε) and task(s) (τ) (2). Now when the data to be sent is available at the sending application, it is directly pushed to its associated channel using the push() functionality that is implemented in the flow control layer (3). push() starts a chain of function calls: activate() and queue() that queues the task in the ready queue of the priority-based scheduler (4). Once the sender task is scheduled for execution, the scheduler calls the perform() function which calls the *Send* API of the interface layer (5). Simultaneously, on the receiver side, the RDMA operations are managed in the same way except that the receiver task is activated and queued by the scheduler with an empty channel (6). When the receiver task is scheduled for execution, the scheduler calls the perform() function that calls the *Receive*

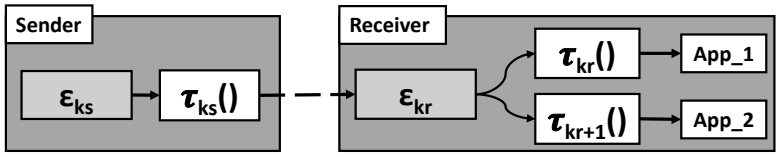


Figure 5.9: One channel feeding data to multiple consumer tasks.

API of the interface layer (7). If the receive operation is successful, the data is then pulled by the application, and the operation is completed (8).

5.4.3 Real-time extensions

Multi-rate DAGs

In automotive systems, we can find applications that comprise tasks running at different frequencies. Such applications are often modeled using multi-rate DAGs [133]. In such DAGs, we can easily find a producer task that feeds data to two or more consumer tasks that run with different frequencies. Consider an example where frames captured by the camera are usually available at 30 Hz. They can be fed to pedestrian detection tasks also at 30 Hz. At the same time, for lane departure detection, they can be fed at 100 Hz. Now, when the consumer tasks are in the same SoC, multiple memory regions are pinned typically to receive the same data at different rates. However, our communication stack allows us to implement a set of tasks $\{\tau_K, \tau_{K+1}, .., \tau_{Kn}\}$ that consume data from the same channel ε_{cam} without introducing additional memory copies. Hence, with our proposed communication stack, it is very easy to implement realistic distributed automotive systems where applications run at different rates.

The conventional RDMA operation to serve these flows will require the user to pin another buffer on SOC-1 and copy the data between the buffers before creating parallel flows of data. However, as shown in Figure 5.9, we can use our stack, to implement a set of tasks on the reciever side $\{\tau_{Kr}, \tau_{Kr+1}, .., \tau_{Krn}\}$ that consume the same data channel ε_{ks} on the sender side without introducing extra copy in the driving functionality flow. through the task-channel model, our proposal can support DAGs that have different firing conditions and even different priorities in order to meet high-level driving application end-to-end timing requirements.

Fixed-point preemption

For mixed-criticality automotive applications sharing the same physical network link for communication, we may encounter a case where a best-effort application is blocking the transmission of data packets of a time-sensitive application. That is, a large amount of data may be sent by a best-effort application while a time-sensitive application waits to send its data. This is because Tasking Framework does not support preemption. Also, such a scenario will lead to a substantially longer communication latency for a critical application which might cause unacceptable performance degradation.

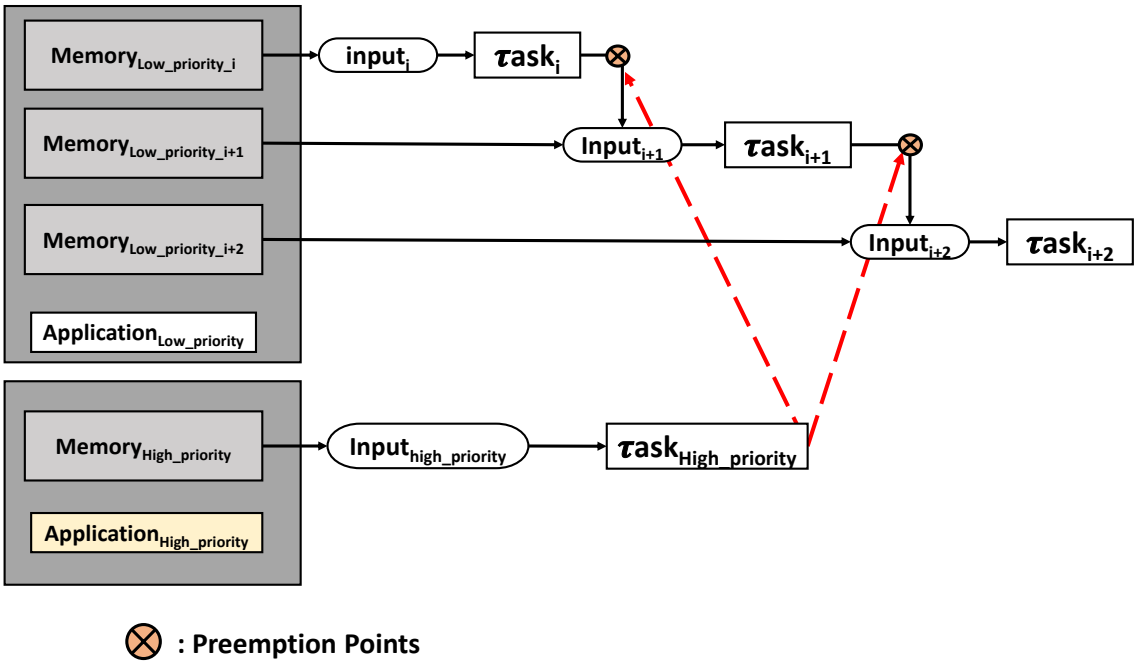


Figure 5.10: Fixed-point preemption in an RDMA communication.

To address this problem, we introduced a preemption feature in RDMA communication. This preemption policy is particularly useful where multiple applications use the same RDMA port to send/receive data and have different priorities. In essence, our flow control layer supports fixed-point preemption [25], i.e., scheduling decisions can be taken at one or more fixed points while sending the whole data. To allow such a fixed-point preemption, we use a multi-channel and multi-task implementation for sending low-priority and large-sized data. In this implementation, τ_{LP}—the task implementing the RDMA sending functionality for a low-priority appplication—is split into a set of tasks τ_{LPi}, τ_{LPi+1}, .., $\tau_{LPn}\}$. Each task takes data from one channel and there is a fixed set of channels $\{\varepsilon_{LPi}$, ε_{LPi+1}, .., $\varepsilon_{LPn}\}$ instead of one channel ε_{LP}. Figure 5.10 shows an example of how we can implement preemption-enabled RDMA communication. Here, the sum of the sizes of these partitioned channels is equal to the size of the preemption-unaware channel that can hold the whole data to be sent. Each of these tasks are non-preemptively scheduled in a certain order given by the design. At each preemption point, the scheduler checks the availability of other high-priority RDMA functionalities (or communication tasks), reducing its blocking time due to low-priority RDMA functionalities. Due to the fixed preemption points, τ_{HP} (in Figure 5.10) can be executed before τ_0, after τ_0 or after τ_1 based on when it is ready to run. This ensures reduced and deterministic latency for high-priority RDMA communication. For such a preemption policy one can apply worst-case latency analysis similar to the worst-case response time analysis known in real-time system literature [25]. To apply such a technique to estimate the communication latency is a future work.

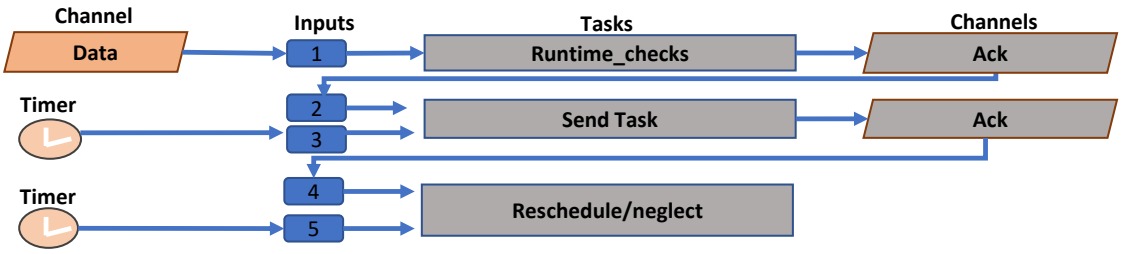
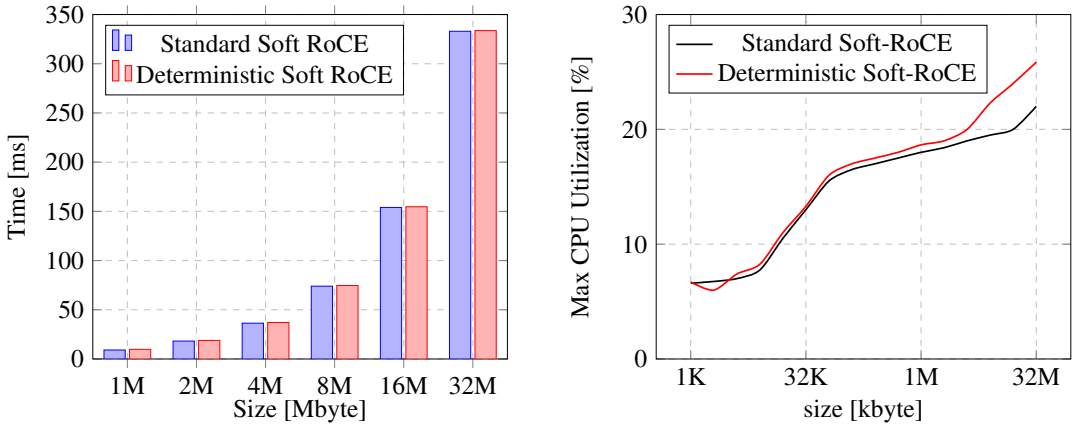


Figure 5.11: Sending data via the flow control layer.

5.4.4 Support for more QoS metrics

Figure 5.11 illustrates how a more robust RDMA send operation can be carried out using our proposed stack. Here, an RDMA send operation involves three steps as follows: (i) to perform runtime checks before starting data transfer, (ii) to send data (i.e., calling the *Send* API in the interface layer), and (iii) to receive the acknowledgment for successful completion or reschedule the operation if acknowledge is not received.

Once data is pushed by the application to the channel on the sending side, it can trigger a task to perform runtime checks. These checks can be related to different QoS requirements, e.g., data freshness or security-related. However, the more checks we perform, the more the communication overhead, i.e., the communication latency will increase. Once these checks are finished, a task is triggered to send data. This task runs based on the scheduling policy configured in the flow control layer. Note that we can also configure this task to wait for a timer signal activated periodically. Such a periodic activation will allow us to implement time-triggered communication over Soft-RoCE, which may be desirable in many safety-critical applications. Further, as mentioned in Section 5.2.1, once an RDMA communication is completed successfully, there is a WQE in the CQ. We can also use this information from Soft-RoCE layer to activate a task in the flow control layer. Simultaneously, we can configure the same task to get triggered by a timer signal generated after a pre-configured amount of time from the activation of the data send task. If the task is first triggered because of the notification for successful completion, then the timer is inactivated. However, if the task is triggered by the timer signal then the whole flow can be restarted or a warning can be generated based on the design requirements.



(a) Communication latency

(b) Maximum CPU Utilization

Figure 5.12: Overheads of Deterministic Soft-RoCE versus Standard Soft-RoCE

5.5 Evaluation

Hardware setup:

We emulate a MPSoC platform with internal RDMA connections by creating a two-node setup where each node resembles an SoC. Each node comprises a x86_64 Intel CPU and a RTX 2080 NVIDIA GPU. Both of them are equipped with an RTL8111 NIC and are connected to the same 8-port Gigabit Ethernet switch. Linux 5.13.0-51-generic kernel runs on both nodes.

Different communication stacks:

To transfer data between the two computation nodes, we use three different communication stacks. (i) *TCP*: We can use the default Ethernet protocol stack with TCP/IP. (ii) *Standard Soft-RoCE*: We can use the native Soft-RoCE implementation extended with our simple-to-use APIs (for initialization, handshake, send and receive). (iii) *Deterministic Soft-RoCE*: We can use the full multi-layer communication stack (including the scheduler layer) that we have implemented. While we have compared (i) and (ii) in Section 5.2.3, in this section, we mainly compare (ii) and (iii).

Communication overheads:

We assess the overheads added by Deterministic Soft-RoCE compared to Standard Soft-RoCE in terms of communication latency and maximum CPU utilization. We measure the overheads for sending data packets of varying sizes from one node to another. (i) We have observed that the increase in the communication latency stays constant around 670 us. We have seen a similar overhead for smaller packets where this increase is significant, e.g., Deterministic Soft-RoCE increases the latency of sending 64 KB data by approximately 100%. Nevertheless, as shown in Figure 5.12a, for larger packets with more than 1 MB data, the overhead is less than 7.5%. Later, in this section, we will see that this is an acceptable cost to pay for sending large-sized safety-critical data in AD/ADAS applications considering that Deterministic Soft-RoCE will reduce the worst-case communication latency significantly in the presence of interfering best-effort data packets. (ii) We have measured the maximum

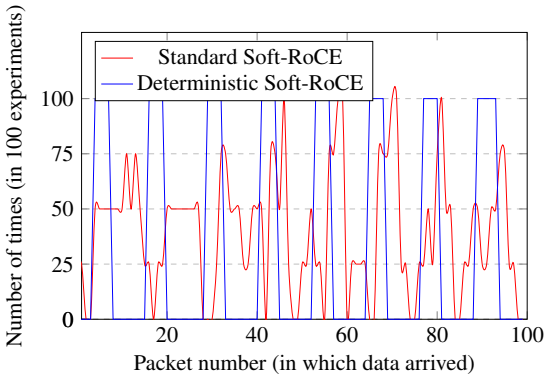


Figure 5.13: Packets delivered in order with Deterministic Soft-RoCE and out of order with Standard Soft-RoCE.

CPU utilization on the sender side while sending each data packet. For each data size, we have considered 150 different data transfers and have noted the maximum CPU utilization among them. Figure 5.12b shows the maximum CPU utilization for sending data of different sizes both with Deterministic Soft-RoCE and Standard Soft-RoCE. It is clear that with Deterministic Soft-RoCE CPU utilization is higher, i.e., more computation is required by the CPU. For data size up to 4 MB, the difference in the CPU utilization is less than 1% in absolute value, which is negligible. For data size more than 4 MB, we see around 5 – 6% absolute increase in CPU utilization. Nevertheless, the CPU utilization is still significantly lower than when using the standard TCP stack, as observed in Figure 5.7.

Packet order:

We have created three tasks on one node that are sending data in a round-robin fashion. In each turn, a task sends 4 data packets consecutively. For each sender task, there is a task on another node receiving the data. We repeat the experiment 100 times. Let us assume here that the data transfers have equal priority, i.e., a high priority. For such an assumption, we expect the data to be sent as per the first-in-first-out (FIFO) policy. We use Wireshark [**wireshark**] to observe when packets are transferred on the network for one task and note down the order. Figure 5.13 shows the number of times (in 100 experiments) data sent by the observed task appeared on the network as the $n-$th packet, where $0 \leq n \leq 100$ (we only consider the first 100 packets in each experiment). With Deterministic Soft-RoCE, we have seen that the packets always appear in the network in the same order as they are sent. This is shown by the solid blue line in the figure where 4 consecutive packets are sent at regular interval by the task under study across all experiments. The line drops to 0 when packets from other tasks are being sent. Conversely, with Standard Soft-RoCE, packets appear on the network out of order, as shown in the figure. Overall with Standard Soft-RoCE, for the observed task, only 30% of the packets are received in the same order as they are sent. This nondeterministic behavior of Standard Soft-RoCE prevents it from being used for time-sensitive applications because a packet can be delayed for an arbitrary amount of time, especially when multiple tasks are communicating over the same network link.

Communication latency:

In this experiment, we have three tasks sending data with different priorities (i.e., high,

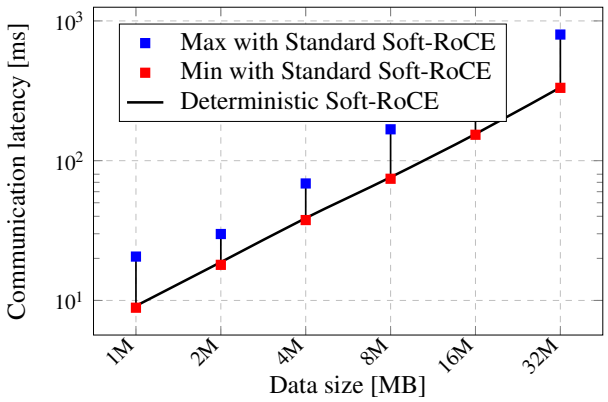


Figure 5.14: Latency for a high-priority data packet.

medium, and low) on one node. On the receiving side, we have a consumer task for each data. We vary the data size exponentially from 1 MB to 32 MB and carry out 100 runs for each data size. In Figure 5.14, we show that when Standard Soft-RoCE is used for the data transfers, we see that there is a large variation in the communication latency of the high-priority packet. The maximum latency can easily be more than two times the minimum latency, e.g., for 1 MB, we observe a maximum latency of 20.6 ms and a minimum latency of 8.9 ms. This clearly shows that Standard Soft-RoCE is not suitable for sending latency-sensitive (or high-priority) data packets. Conversely, Deterministic Soft-RoCE respects the QoS requirement of a packet (e.g., priority in this case). Hence, in Figure 5.14, we can see that the communication latency of the high-priority packet remains constant for each data size when Deterministic Soft-RoCE is used. Also, the communication latency for the high-priority packet is nearly equal to the minimum latency observed when Standard Soft-RoCE is used. These results show that our proposed communication stack has the potential to be used for AD/ADAS applications with firm real-time requirements.

Distributed automotive application:

With the increasing complexity of automotive systems, their payloads are evolving to include machine learning and neural network applications that require high computation needs with specific computation accelerators. The payload of the automotive system-on-chip (SoC) refers to the set of software and hardware components responsible for carrying out the specific tasks required by the automotive system, such as processing sensor data, controlling actuators, and running decision-making algorithms. In this experiment, we want to quantitatively assess the benefits of using Soft-RoCE towards reducing the end-to-end latency of ADAS applications. In this context, the most important challenge is that, to the best of our knowledge, there is no available automotive benchmark that runs on accelerators distributed over multiple SoCs. Nevertheless, it is not difficult to imagine that such implementations will be common in autonomous vehicles where, for example, object detection and tracking is followed by motion planning that is further followed by vehicle control and each of these algorithms can be accelerated using specialized processors, e.g., [79]. Hence, we have developed a benchmark to resemble an ADAS application, which is also an engineering contribution of this chapter.

We started with an object detection algorithm based on YOLO [103], the state-of-the-art family of DNN architectures and models, pre-trained on the COCO dataset [81]. The DNN has 106 layers with fully convolutional underlying architecture and provides a very high accuracy in object detection. Considering that we were looking for at least two neural networks where the result of one is fed into the second, we partitioned YOLO into two neural networks. Here, we have modified YOLO so that it can be easily partitioned through parametrization. It is also possible to obtain more than two neural networks from YOLO. An example partitioned YOLO is shown in Figure 5.15 where the input of the first neural network is a scaled image (or a camera frame). Note that splitting DNNs for accelerated training is known [62] and RDMA can even be used in such setups to improve the training throughput. However, here, we have used partitioned YOLO for inference.

We run two neural networks obtained from YOLO in two different GPUs attached to different nodes. The output of the first neural network will be used as input by the second neural network which we transfer using Standard Soft-RoCE. For the partitions shown in Figure 5.15, 5.64 MB data is transferred using Standard Soft-RoCE. We measure the computation and communication time for object detection using distributed YOLO. The computation times in the two nodes add up to 100 ms and the communication latency is 50 ms, i.e., the end-to-end latency for object detection is 150 ms. Further, we transfer the same amount of data using TCP where we get a communication latency of 65 ms. Thus, with TCP, we can say that the end-to-end latency of object detection is 165 ms. That is, with Standard Soft-RoCE, we can reduce the end-to-end latency by 9.1%. If we consider Deterministic Soft-RoCE that has an overhead of 0.67 ms, the reduction in the end-to-end latency is 8.7%. These results emphasizes the fact that reduction in the communication latency can improve the end-to-end latency of ADAS applications significantly. Note that here we have not considered the impact of Deterministic Soft-RoCE in reducing the worst-case latency in comparison to TCP and Standard Soft-RoCE.

Choosing YOLO for this experimental framework was a deliberate decision, aimed at closely replicating the operational dynamics of an actual payload within middleware environments, as facilitated through its integration with ROS via the darknet_ros repository [45]. This decision underscores our intention to align our experimental setup with real-world application scenarios, leveraging the advanced capabilities of YOLO within the structured and versatile ROS ecosystem. Although the initial phase of our experiment focused on deploying YOLO in a bare-metal configurationwithout the direct involvement of middleware solutions (ROS2). The potential for extending this work to fully incorporate such environments is clear and compelling. Integrating YOLO with ROS2 as a subsequent phase would not only enhance the simulation's fidelity to the complexities and challenges of live deployment scenarios but also provide a rigorous framework for testing and evaluating the effectiveness of our proposed RDMA stack. This extension would enable us to assess our system's ability to handle real-world demands, ensuring that our research contributes valuable insights into the scalability, performance, and reliability of object detection systems in robotic applications.

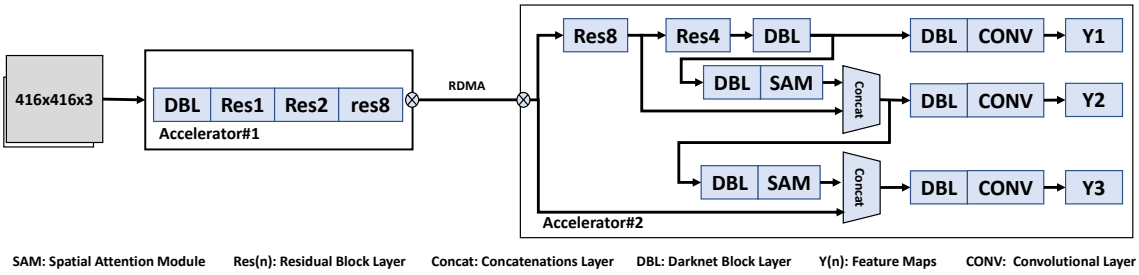


Figure 5.15: Partitioned YOLO running on multiple accelerators.

5.6 Conclusion

In this chapter, we have explored the utilization of Soft-RoCE within the domain of Autonomous Driving and Advanced Driver-Assistance Systems , advocating for its adoption to meet the stringent low-latency and deterministic communication requirements essential for distributed automotive platforms. Our proposition leans on the integration of Soft-RoCE to significantly diminish communication latency, surpassing the performance of conventional Ethernet communication over TCP, while addressing the challenge of nondeterministic timing behavior inherent in standard Soft-RoCE implementations. Through the development and experimental validation of a multi-layer communication stack, we have demonstrated the feasibility of achieving deterministic and low-latency data transfer for high-priority tasks, essential for the real-time performance demands of AD/ADAS applications.

Looking ahead, we aim to augment this communication stack with support for Time-Sensitive Networking (TSN) to cater to diverse traffic classes, thereby enhancing the robustness and reliability of communication for autonomous functionalities. This involves providing seamless interfaces for integrating Data Distribution Service (DDS) over Soft-RoCE, thus facilitating easier adoption and implementation across different System on Chips (SoCs). Our experimental results underscore the potential of RDMA technology, traditionally employed in data centers, in revolutionizing automotive platforms by offering low latency communication with minimal CPU involvement. The proposed multi-layer stack, complemented by a C++ library for straightforward application interfaces, not only demonstrates a reduction in end-to-end latency for distributed object detection tasks but also showcases minimal implementation overhead, underscoring the viability of our approach in mitigating the impact of interfering data traffic on critical communication pathways. Further, we have developed a C++ library that offers easy-to-use communication interfaces for distributed applications while implementing the proposed architecture. Experiments show that our proposal reduces the end-to-end latency of distributed object detection by nearly 9% while having an implementation overhead of less than 1.5% and it also minimizes the effects of other data traffic on the delay in high-priority communication. Moving forward, we plan to validate the real-time performance of our proposed stacks across various SoCs, further cementing the foundation for next-generation vehicles powered by computationally intensive deep neural networks (DNNs) and distributed software with strict timing requisites.

Chapter 6

Conclusion Remarks

6.1 Summary

This thesis advocates the comprehensive modeling of middleware-based autonomous systems to foster a deeper understanding of component interactions and overall system dynamics for both computing and communicating components as the simple workflow shown in 6.1 Such methodologies are invaluable in safety-critical contexts, where adhering to the timing requirements and controlling behavior is paramount. Nonetheless, understanding the computation-communication behavior of such systems is not straightforward and faces obstacles, including system complexity, interactions between systems and their environments, and the heterogeneity of the systems. Through our work, we demonstrate that these challenges can be surmounted by adopting a systematic strategy that leverages the understanding of the system computation-communication behavior to extract, model, and analyze the application timing behaviors. In this section, we briefly revisit our research questions and give the intuitions behind the methodologies through which we approached these questions.

The thesis investigated the following principle research question:

Research Question 1 (Extract): How can timing properties and architectures of autonomous middlewares be effectively extracted and analyzed to optimize system performance and minimize inter-application interference?

This question's goal is to understand the application's time behavior, aiming to reveal the complex interactions between applications, identify bottlenecks, and assess the impact on system performance. To answer this question, we designed an approach for extracting the timing properties of middle-ware-based autonomous systems leveraging the operating system infrastructure. We have designed plug and plug-and-play mechanism that can extract and model autonomous applications' timing properties tackling the confidentiality and the availability of the application source code. We introduced a methodology to model the applications as DAGs using time-based precedes relationships besides offering mathematical algorithms for calculating the timing properties of such applications systematically, avoiding the need for heuristic approaches commonly found in existing literature. Furthermore, our approach exhibits scalability, enabling modeling system behaviors when running across various physical environments, and demonstrates its adaptability in real-world contexts.

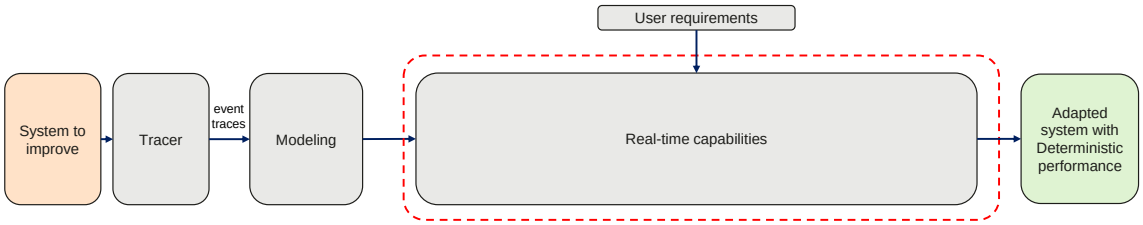


Figure 6.1: General workflow

Research Question 2 (Manage): How can jitter-free execution with controlled latency be guaranteed for autonomous applications using extracted timing information?

The question aims to leverage the extracted timing architecture for the systems to ensure deterministic execution and minimize jitter, focusing on managing latency effectively. It seeks to develop adaptive methodologies that can accommodate the varying nature of autonomous workloads, ensuring critical tasks are completed within their timing constraints, thus enabling reliable performance of essential functions under unpredictable conditions. To tackle this question, we propose implementing a reservation-based scheduler, complemented by a middleware-based solution, to guarantee determinism and ensure output availability within a bounded time frame. This strategy is inspired by the Logical Execution Time (LET) paradigm, a proven approach for managing latency and jitter in safety-critical real-time applications. We showed how to architecturally adapt the concept non-intrusively we also adapted an automated implementation, featuring a co-designed system optimizer to improve the system's overall timing behavior. By applying our proposed mechanism on a real-world benchmark implementing Lidar-based localization, we maintain a *constant* end-to-end latency that is even *13% shorter* than an improved LET implementation.

Research Question 3 (Exploration): How can network architectures be optimized to support the real-time requirements of distributed autonomous applications, mitigating the CPU centrism effect in a network-connected distributed architecture? With the increasing heterogeneity in the system architecture, the question shows the need for a network infrastructure for distributed autonomous systems, with a focus on real-time behavior while reducing CPU load from excessive data copying. In this context, we tested the applicability of RDMA-based communication for real-time applications, where we identified its weakness based on the extracted timing properties. We designed a multi-layer communication stack comprising a deterministic scheduler on top of the RDMA communication protocol. Further, we have developed an interface that offers easy-to-use communication interfaces for distributed applications while implementing the proposed architecture. Experiments show that our communication stack reduces the end-to-end latency of distributed real-world object detection applications by nearly 10% while having an implementation overhead of less than 1.5%. We also should minimizes the effects of other data traffic on the delay in high-priority communication

6.2 Future Research Directions

The explorations and findings presented in this thesis pave the way for several exciting future research opportunities. We have laid a foundation for understanding and optimizing the timing behaviors and architectures of autonomous middlewares. Moving forward, we aim to delve deeper into dynamic resource management, advanced latency management techniques, and the integration of cutting-edge network technologies. Here are the refined future research directions:

Leveraging eBPF-Based Tracing for Closed-Loop Scheduling Algorithms: The exploration into dynamic allocation and advanced latency management techniques sets the stage for integrating our eBPF-based tracing platform to design and implement closed-loop scheduling algorithms. eBPF's powerful capabilities in monitoring and tracing system behavior in real-time without significant overhead provide an unparalleled opportunity to gain deep insights into system operations. Utilizing these insights, future work will focus on developing scheduling algorithms that adapt dynamically to the system's state, ensuring optimal performance and responsiveness. This approach will enable autonomous systems to adjust their computational and communication resources in real-time, based on the actual workload and performance metrics gathered via eBPF tracing. By offering a closed-loop mechanism, these algorithms will not only react to changes in system dynamics but also proactively optimize resource allocation and scheduling decisions to meet the stringent requirements of real-time operations. The goal is to create a more intelligent, adaptive, and efficient system that can maintain high levels of performance and reliability under varying conditions. This novel approach is expected to significantly reduce jitter and latency, enhance system throughput, and improve the predictability and reliability of autonomous applications. By providing a more granular and accurate understanding of system behavior, this research direction will empower developers and systems architects to fine-tune their systems with unprecedented precision.

Dynamic Resource Management for Enhanced Performance: Building on the static allocation of threads to reservation servers, future work will explore dynamic allocation strategies, particularly focusing on optimizing the allocation for the last callback in the computation chain. By dynamically splitting the computation and data publishing tasks across separate servers, we aim to eliminate the need for additional latency-shaping callbacks, thereby streamlining processing efficiency and reducing system overhead.

Extension to Various Middlewares and Operating Systems: While the thesis has focused on ROS2, DDS, and Linux, the principles and methodologies developed are broadly applicable. Future work will expand the applicability of these concepts to other middlewares and operating systems, exploring their potential and adaptability in different contexts and architectures. This expansion will validate the universality of the proposed approaches and their effectiveness across a wider range of platforms.

Integration of Time-Sensitive Networking (TSN) and Advanced Networking Protocols: Recognizing the importance of real-time communication in distributed autonomous systems, future research will incorporate support for Time-Sensitive Networking (TSN) and different traffic classes. This also involves developing easy-to-use interfaces for Data Distribution Service (DDS) over Soft-RoCE and extending the communication stack to support these advanced networking protocols. By doing so, we aim to enhance the real-time performance and reliability of distributed systems, ensuring that they meet stringent timing requirements. To confirm the real-time performance and adaptability of the proposed communication stacks, future studies will implement these solutions across different SoCs. This validation process will assess the real-world applicability of the communication strategies in diverse hardware environments, ensuring that the proposed approaches can effectively minimize latency and improve overall system performance in various operational contexts

These future research directions aim to address some of the critical challenges in middleware based heterogeneous distributed autonomous systems, focusing on dynamic resource management, latency management, the extension to various platforms, and the integration of advanced networking protocols. By pursuing these avenues, we hope to further enhance the capabilities of autonomous systems, ensuring they are more efficient, reliable, and adaptable to the demands of real-world applications.

Bibliography

[1] Luca Abeni, Alessandro Biondi, and Enrico Bini. “Partitioning real-time workloads on multi-core virtual machines”. In: *Journal of Systems Architecture* (2022).

[2] Aeronautical Radio, Incorporated. *ARINC 664 P7 – AIRCRAFT DATA NETWORK PART 7 AVIONICS FULL-DUPLEX SWITCHED ETHERNET NETWORK*. 2009.

[3] Benny Akesson et al. “A Comprehensive Survey of Industry Practice in Real-Time Systems”. In: *Real-Time Syst.* 58.3 (2022), pp. 358–398.

[4] Maria Merin Antony and Ruban Whenish. “Advanced driver assistance systems (ADAS)”. In: *Automotive Embedded Systems: Key Technologies, Innovations, and Applications*. 2021.

[5] Abdullah Al Arafat et al. “Response Time Analysis for Dynamic Priority Scheduling in ROS2”. In: *ACM/IEEE Design Automation Conference (DAC)*. 2022.

[6] IEEE Standards Association. *IEEE Standard for Local and Metropolitan Area Networks–Virtual Bridged Local Area Networks Amendment 13: Congestion Notification (IEEE. 802.1Qau*. 2010.

[7] IEEE Standards Association. *IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control (IEEE. 802.11Qbb*. 2011.

[8] AUTOSAR Consortium. *Requirements on Timing Extensions*. 2018. URL: https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_RS_TimingExtensions.pdf (visited on 10/30/2023).

[9] AUTOSAR.org. *Adaptive AUTOSAR Architecture Documentation*. URL: https://www.autosar.org/fileadmin/standards/R21-11/AP/AUTOSAR_EXP_SWArchitecture.pdf.

[10] AUTOSAR.org. *AUTOSAR Adaptive Platform*. URL: https://www.autosar.org/standards/adaptive-platform.

[11] AUTOSAR.org. *AUTOSAR Classic Platform*. URL: https://www.autosar.org/standards/classic-platform.

[12] Autoware. *Autonomous Valet Parking Demonstration*. URL: https://autowarefoundation/autoware.auto/AutowareAuto/avpdemo.html.

[13] Autoware Foundation. *Localization Demo using rosbag*. 2020. URL: https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/rosbag-localization-howto.html (visited on 10/30/2023).

[14] b-plus Group. *MDLake 100G data sheet*. 2022. URL: https://www.b-plus.com/fileadmin/data%5C_storage/100G%5C_EN%5C_v1.0.pdf.

[15] Pavan Balaji, Hemal V Shah, and Dhabaleswar K Panda. "Sockets vs RDMA interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck". In: *Workshop on RDMA: Applications, Implementations, and Technologies (RAIT)*. 2004.

[16] Motti Beck and Michael Kagan. "Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure". In: *Workshop on Data Center-Converged and Virtual Ethernet Switching*. 2011.

[17] Christophe Bédard, Ingo Lütkebohle, and Michel Dagenais. "ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2". In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 6511–6518.

[18] Mohamed Benazouz and Alix Munier-Kordon. "Cyclo-Static DataFlow Phases Scheduling Optimization for Buffer Sizes Minimization". In: *ACM International Workshop on Software and Compilers for Embedded Systems*. 2013.

[19] Alessandro Biondi and Marco Di Natale. "Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm". In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2018.

[20] Alessandro Biondi et al. "Logical execution time implementation and memory optimization issues in autosar applications for multicores". In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2017.

[21] Tobias BlaSS et al. "A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance". In: *IEEE Real-Time Systems Symposium (RTSS)*. 2021.

[22] Tobias Blass et al. "Automatic latency management for ROS 2: Benefits, challenges, and open problems". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2021.

[23] Tobias Blass et al. "Automatic latency management for ros 2: Benefits, challenges, and open problems". In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2021, pp. 264–277.

[24] B. B. Brandenburg, M. Gül, and M. Vanga. *A tour of LITMUS^{RT}*. 2017. URL: https://litmus-rt.org/tutorial/manual.html.

[25] Reinder J Bril, Johan J Lukkien, and Wim FJ Verhaegh. "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption". In: *Real-Time Systems* 42 (2009).

[26] Broadcom Corporation. *BroadR-Reach R Physical Layer Transceiver Specification for Automotive Applications V3.0*. 2014.

[27] M. Bui et al. "Comparative Study of 3D Point Cloud Compression Methods". In: *IEEE International Conference on Big Data (Big Data)*. 2021.

[28] Nick Burek. *ROS QoS - Deadline, Liveliness, and Lifespan*. 2019. URL: https://design.ros2.org/articles/qos_deadline_liveliness_lifespan.html (visited on 10/30/2023).

[29] Roland Burns. *Advanced control engineering*. Elsevier, 2001.

[30] G. C. Buttazzo and A. Cervin. "Comparative Assessment and Evaluation of Jitter Control Methods". In: *International Conference on Real-Time and Network Systems (RTNS)*. 2007.

[31] Daniel Casini et al. "Response-time analysis of ROS 2 processing chains under reservation-based scheduling". In: *Euromicro Conference on Real-Time Systems (ECRTS)*. 2019.

[32] Anton Cervin et al. "Using JitterTime to Analyze Transient Performance in Adaptive and Reconfigurable Control Systems". In: *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019.

[33] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. "PiCAS: New design of priority-driven chain-aware scheduling for ROS2". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2021.

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

[35] Seunghyuk Choi et al. "Advanced driver-assistance systems: Challenges and opportunities ahead". In: *McKinsey & Company* (2016).

[36] The kernel development community. *Linux Kernel Tracing Technologies*. URL: https://docs.kernel.org/trace/index.html.

[37] Gongpei Cui. "A Driving Assistance System with Hardware Acceleration". In: (2015).

[38] Dakshina Dasari et al. "End-to-end analysis of event chains under the qnx adaptive partitioning scheduler". In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2022.

[39] Saeid Dehnavi et al. "CompROS: A composable ROS2 based architecture for real-time embedded robotic development". In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2021.

[40] eclipse.org. *Eclipse Cyclon DDS*. URL: https://cyclonedds.io/.

[41] eProsima. *DDS: Asynchronous vs synchronous publishing*. 2023. URL: https://www.eprosima.com/index.php/resources-all/performance/dds-asynchronous-vs-synchronous-publishing (visited on 10/30/2023).

[42] eprosima. *asynchronous and synchronous publishing*. URL: https://www.eprosima.com/idds-asynchronous-vs-synchronous-publishing.

[43] Rolf Ernst, Leonie Ahrendts, and Kai-Björn Gemlau. "System Level LET: Mastering Cause-Effect Chains in Distributed Systems". In: *Conference of the IEEE Industrial Electronics Society (IECON)*. 2018.

[44] Rolf Ernst et al. "The logical execution time paradigm: New perspectives for multicore systems (dagstuhl seminar 18092)". In: *Dagstuhl Reports*. Vol. 8. 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[45] Robotic Systems Lab - Legged Robotics at ETH Zürich. *YOLO ROS: Real-Time Object Detection for ROS*. URL: https://github.com/leggedrobotics/darknet_ros/.

[46] *Flame Graph*. URL: https://www.brendangregg.com/flamegraphs.html.

[47] Ian Foster. *Designing and building parallel programs: Concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[48] The Autoware Foundation. *Autonomous Valet Parking*. URL: https://autoware.auto/AutowareAuto/avpdemo.html.

[49] Gene F Franklin et al. *Feedback control of dynamic systems*. Prentice hall Upper Saddle River, 2002.

[50] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[51] Kai-Björn Gemlau, Leonie Köhler, and Rolf Ernst. "Efficient Run-Time Environments for System-Level LET Programming". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021.

[52] Kai-Björn Gemlau et al. "System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software". In: *ACM Trans. Cyber-Phys. Syst.* (2021).

[53] Alan Gibbons. *Algorithmic graph theory*. Cambridge university press, 1985.

[54] GL Gopu, KV Kavitha, and James Joy. "Service oriented architecture based connectivity of automotive ecus". In: *2016 international conference on circuit, power and computing technologies (iccpct)*. IEEE. 2016.

[55] Chuanxiong Guo. "RDMA in data centers: Looking back and looking forward". In: *Keynote at APNet* (2017).

[56] Chuanxiong Guo et al. "RDMA over commodity Ethernet at scale". In: *ACM SIGCOMM Conference*. 2016.

[57] Zain Alabedin Haj Hammadeh et al. "Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems". In: *Proceedings of the 15th OSPERT*. 2019.

[58] X. Han et al. “A real-time LIDAR and vision based pedestrian detection system for unmanned ground vehicles”. In: *Asian Conference on Pattern Recognition (ACPR)*. 2015.

[59] Jacqueline Henle et al. “Architecture platforms for future vehicles: a comparison of ros2 and adaptive autosar”. In: *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2022.

[60] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. “Giotto: a time-triggered language for embedded programming”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 84–99.

[61] Torsten Hoefler et al. “Datacenter Ethernet and RDMA: Issues at Hyperscale”. In: *arXiv preprint arXiv:2302.03337* (2023).

[62] Yanping Huang et al. “GPipe: Efficient training of giant neural networks Using pipeline parallelism”. In: *International Conference on Neural Information Processing Systems*. 2019.

[63] IEEE Standards Association. *IEEE Standard for Ethernet Amendment 5: Specification and Management Parameters for Interspersing Express Traffic (IEEE 802.3br-2016)*. 2016.

[64] IEEE Standards Association. *IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks - Amendment: 9: Stream Reservation Protocol (IEEE 802.1Qat-2010)*. 2010.

[65] IEEE Standards Association. *IEEE Standard for Local and Metropolitan Area Networks – Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks (IEEE 802.1AS2010)*. 2010.

[66] IEEE Standards Association. *IEEE Standard for Local and Metropolitan Area NetworksVirtual Bridged Local Area Networks - Amendment: Forwarding and Queuing Enhancements for Time-Sensitive Streams (IEEE 802.1Qav-2009)*. 2009.

[67] InfiniBand Trade Association et al. *Supplement to InfiniBand Architecture Specification 1.2.1 Annex A16*. 2010.

[68] InfiniBand Trade Association et al. *Supplement to InfiniBand Architecture Specification 1.2.1 Annex A17*. 2014.

[69] Karl Henrik Johansson, Martin Törngren, and Lars Nielsen. “Vehicle applications of controller area network”. In: *Handbook of networked and embedded control systems* (2005).

[70] Robert Kaiser and Stephan Wagner. “Evolution of the PikeOS microkernel”. In: *First International Workshop on Microkernels for Embedded Systems*. Vol. 50. 2007.

[71] G. Kaur, M. Kumar, and M. Bala. “Performance Evaluation of Soft-RoCE over 1 Gigabit Ethernet”. In: *IOSR Journal of Computer Engineering* 15 (2013).

[72] J. Kennedy and R. Eberhart. “Particle swarm optimization”. In: *International Conference on Neural Networks (ICNN)*. 1995.

[73] Karsten Knese. *ROS 2.0 rosbags*. 2019. URL: https://github.com/ros2/design/blob/ros2bags/articles/rosbags.md (visited on 10/30/2023).

[74] Tobias Kronauer et al. "Latency Analysis of ROS2 Multi-Node Systems". In: *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. 2021.

[75] Tobias Kronauer et al. "Latency analysis of ROS2 multi-node systems". In: *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. IEEE. 2021.

[76] Takahisa Kuboichi et al. "CARET: Chain-aware ROS 2 evaluation tool". In: *IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. 2022.

[77] Takahisa Kuboichi et al. "CARET: Chain-Aware ROS 2 Evaluation Tool". In: *IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. 2022.

[78] Chris Lattner and Vikram Adve. "The llvm compiler framework and infrastructure tutorial". In: *Languages and Compilers for High Performance Computing: 17th International Workshop, LCPC 2004*. Springer. 2005.

[79] Yunfei Li et al. "FPGA accelerated model predictive control for autonomous driving". In: *Journal of Intelligent and Connected Vehicles* (2022).

[80] Z. Li, A. Hasegawa, and T. Azumi. "Autoware_Perf: A tracing and performance analysis framework for ROS 2 applications". In: *Journal of Systems Architecture* 123 (2022).

[81] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. 2014.

[82] B. Lincoln and A. Cervin. "JITTERBUG: A tool for analysis of real-time control performance". In: *IEEE Conference on Decision and Control (CDC)*. 2002.

[83] Linux.org. *bpftool Manuel*. URL: https://man.archlinux.org/man/bpftool.8.en.

[84] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. "Analytical characterization of end-to-end communication delays with logical execution time". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2244–2254.

[85] Christian Menard et al. "Achieving determinism in adaptive AUTOSAR". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020.

[86] Qingkai Meng and Fengyuan Ren. "Lightning: A Practical Building Block for RDMA Transport Control". In: *IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. 2021.

[87] Lester James V. Miranda. *PYSWARMS: A research toolkit for particle swarm optimization in Python*. 2017. URL: https://pyswarms.readthedocs.io/en/latest/ (visited on 10/30/2023).

[88] Radhika Mittal et al. “Revisiting network support for RDMA”. In: *ACM Special Interest Group on Data Communication*. 2018.

[89] Keisuke Nishimura et al. “RAPLET: Demystifying publish/subscribe latency for ROS applications”. In: *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2021.

[90] *NividaDocs*. URL: https://docs.nvidia.com.

[91] Philipp Obergfell, Stefan Kugele, and Eric Sax. “Model-based resource analysis and synthesis of service-oriented automotive software architectures”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2019, pp. 128–138.

[92] OMG org. *DDS Standard V1.4*. URL: https://www.omg.org/spec/DDS/1.4/.

[93] *pahole tool*. URL: https://linux.die.net/man/1/pahole.

[94] Gerardo Pardo-Castellote. “OMG data-distribution service: Architectural overview”. In: *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* 2003, pp. 200–206.

[95] Paolo Pazzaglia et al. “Optimal Memory Allocation and Scheduling for DMA Data Transfers under the LET Paradigm”. In: *ACM/IEEE Design Automation Conference (DAC)*. 2021.

[96] Paolo Pazzaglia et al. “Optimizing Inter-Core Communications Under the LET Paradigm using DMA Engines”. In: *IEEE Transactions on Computers* 72.1 (2023), pp. 127–139.

[97] Joshué Pérez, David Gonzalez, and Vicente Milanés. “Vehicle control in ADAS applications: State of the art”. In: *Intelligent Transport Systems: Technologies and Applications* (2015).

[98] Marco Perronet et al. “Work in Progress: Automatic Response-Time Analysis for Arbitrary Real-Time Linux Workloads”. In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2022.

[99] *qperf*. URL: https://linux.die.net/man/1/qperf.

[100] Thinal Raj et al. “A survey on LiDAR scanning mechanisms”. In: *Electronics* (2020).

[101] Ragunathan Rajkumar et al. “Resource kernels: A resource-centric approach to real-time and multimedia systems”. In: *Multimedia Computing and Networking 1998*. Vol. 3310. SPIE. 1997.

[102] R. Recio et al. “RFC 5040: A Remote Direct Memory Access Protocol Specification”. In: *Internet Standards (IETF)* (2007).

[103] J. Redmon and A. Farhadi. “YOLOv3: An Incremental Improvement”. In: *arXiv* (2018).

[104] Günter Reichart and Rinat Asmus. “Progress on the AUTOSAR Adaptive Platform for Intelligent Vehicles”. In: *Automatisiertes Fahren 2020: Von der Fahrerassistenz zum autonomen Fahren 6. Internationale ATZ-Fachtagung*. 2021.

[105] Michael Reke et al. “A self-driving car architecture in ROS2”. In: *2020 International SAUPEC/RobMech/PRASA Conference*. IEEE. 2020.

[106] Stefan Resmerita, Andreas Naderlinger, and Stefan Lukesch. “Efficient realization of logical execution times in legacy embedded software”. In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 2017.

[107] Liz Rice. *Learning EBPF: Programming the linux kernel for Enhanced Observability, networking, and security*. OReilly Media, 2023.

[108] ROS.org. *FAST RTPS DDS*. URL: https://index.ros.org/r/fastrtps/.

[109] ROS.org. *ROS2 FOXY*. URL: https://docs.ros.org/en/foxy/index.html.

[110] Lorenzo Rosa et al. “DerechoDDS: Strongly consistent data distribution for mission-critical applications”. In: *IEEE Military Communications Conference*. 2021.

[111] Debayan Roy et al. “GoodSpread: Criticality-Aware Static Scheduling of CPS with Multi-QoS Resources”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2020.

[112] Debayan Roy et al. “Semantics-Preserving Cosynthesis of Cyber-Physical Systems”. In: *Proceedings of the IEEE* 106 (2018), pp. 171–200.

[113] Debayan Roy et al. “Tighter Dimensioning of Heterogeneous Multi-Resource Autonomous CPS with Control Performance Guarantees”. In: *Design Automation Conference (DAC)*. 2019.

[114] Matthew Ruff. “Evolution of local interconnect network (LIN) solutions”. In: *2003 IEEE 58th Vehicular Technology Conference*. IEEE. 2003.

[115] Changhan Ryu and Sungryong Do. “A Method for Managing Software Assets in the Automotive Industry (Focusing on the Case of Hyundai Motor Company and Parts Makers)”. In: *Applied Sciences* (2023).

[116] Selma Saidi et al. “Autonomous systems design: Charting a new discipline”. In: *IEEE Design & Test* 39.1 (2021), pp. 8–23.

[117] Yukihiro Saito et al. “ROSCH: Real-time scheduling framework for ROS”. In: *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2018.

[118] Sergio Saponara. “Hardware accelerator IP cores for real time Radar and camera-based ADAS”. In: *Journal of Real-Time Image Processing* (2019).

[119] Sergio M Savaresi and Mara Tanelli. *Active braking control systems design for vehicles*. Springer Science & Business Media, 2010.

[120] Suchakrapani Datt Sharma and Michel Dagenais. “Enhanced userspace and in-kernel trace filtering for production systems”. In: *Journal of Computer Science and Technology* (2016), pp. 1161–1178.

[121] Mark Silberstein. “OmniX: An Accelerator-Centric OS for Omni-Programmable Systems”. In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2017.

[122] Safety Standard. “ISO-26262: Road Vehicles Functional safety.(2016)”. In: *Retrieved Oct* (2016).

[123] Miroslaw Staron. *Automotive software architectures*. Springer, 2021.

[124] Jan Staschulat, Ingo Lütkebohle, and Ralph Lange. “The rclc Executor: Domain-specific deterministic scheduling mechanisms for ROS applications on microcontrollers: work-in-progress”. In: *2020 International Conference on Embedded Software (EMSOFT)*. IEEE. 2020.

[125] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. “The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments”. In: *IEEE Transactions on Computers* (1995).

[126] Binqi Sun et al. “Edge Generation Scheduling for DAG Tasks using Deep Reinforcement Learning”. In: *IEEE Transactions on Computers* (2024).

[127] Yue Tang et al. “Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2020.

[128] Yue Tang et al. “Response time analysis and priority assignment of processing chains on ros2 executors”. In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 231–243.

[129] Harun Teper et al. “End-To-End Timing Analysis in ROS2”. In: *IEEE Real-Time Systems Symposium (RTSS)*. 2022.

[130] The RoCE Initiative of the InfiniBand Trade Association (IBTA). *Soft-RoCE: RDMA transport in a software implementation*. 2015.

[131] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. “Taming non-blocking caches to improve isolation in multicore real-time systems”. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2016.

[132] Manohar Vanga et al. “Supporting Low-Latency, Low-Criticality Tasks in a Certified Mixed-Criticality OS”. In: *International Conference on Real-Time Networks and Systems (RTNS)*. 2017.

[133] Micaela Verucchi et al. “Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020.

[134] Yong Wan et al. “An In-Depth Analysis of TCP and RDMA Performance on Modern Server Platform”. In: *IEEE International Conference on Networking, Architecture, and Storage*. 2012.

[135] Tianshi Wang et al. “Congestion Detection and Link Control via Feedback in RDMA Transmission”. In: *IEEE International Conference on Service Science (ICSS)*. 2022.

[136] Yi Wang et al. “Error Recovery of RDMA Packets in Data Center Networks”. In: *International Conference on Computer Communication and Networks (ICCCN)*. 2019.

[137] B. Wolfe et al. “Rapid holistic perception and evasion of road hazards”. In: *Journal of Experimental Psychology: General* 149 (3 2020).

[138] Yuqing Yang and Takuya Azumi. “Exploring real-time executor on ROS 2”. In: *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE. 2020, pp. 1–8.

[139] Yuqing Yang and Takuya Azumi. “Exploring real-time executor on ROS 2”. In: *IEEE International Conference on Embedded Software and Systems (ICESS)*. 2020.

[140] Yifan Yuan et al. “Rambda: RDMA-driven Acceleration Framework for Memory-intensive ţs-scale Datacenter Applications”. In: *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2023.

[141] Xinhai Zhang et al. “Architecture exploration for distributed embedded systems: a gap analysis in automotive domain”. In: *2017 12th IEEE international symposium on industrial embedded systems (SIES)*. IEEE. 2017, pp. 1–10.

[142] Yusheng Zheng et al. “bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions”. In: *arXiv preprint arXiv:2311.07923* (2023).

[143] Yibo Zhu et al. “Congestion control for large-scale RDMA deployments”. In: *ACM SIGCOMM Computer Communication Review* 45 (2015).

[144] Adam Ziebinski et al. “A survey of ADAS technologies for the future perspective of sensor fusion”. In: *Computational Collective Intelligence: 8th International Conference, ICCCI 2016, Halkidiki, Greece, September 28-30, 2016. Proceedings, Part II 8*. 2016.

[145] Tobias Ziegler, Viktor Leis, and Carsten Binnig. “RDMA communciation patterns”. In: *Datenbank-Spektrum* 20 (2020).