Real-time Guarantee of Autonomous Driving Systems: State of the Art and Challenges

TIANZE WU, SKLP, Institute of Computing Technology, CAS & University of Chinese Academy of Sciences, China
WEISONG SHI, University of Delaware, USA

Autonomous Driving(AD) has garnered significant attention in recent years across multiple domains. Despite notable advancements in algorithms and hardware, large-scale deployment of autonomous vehicles remains constrained by the lack of adequate real-time guarantees. This paper comprehensively investigates real-time assurance challenges in AD systems from theoretical and practical perspectives. First, we introduce foundational concepts of real-time systems and analyze common modeling approaches, including the multi-rate DAG and processing chain DAG models. We then delve into the task scheduling and communication mechanisms of three representative middleware systems—ROS2, Cyber, and ERDOS—and the seL4 operating system. Our analysis reveals their underlying design philosophies and optimization strategies for real-time performance.

The findings highlight the critical difficulty of meeting stringent real-time requirements in autonomous systems. By offering insights into current limitations and opportunities for improvement, this paper aims to establish a deeper understanding of real-time assurance issues and encourage greater focus on system-level guarantees within the AD community.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: AD system, real-time guarantee, middleware, task scheduling

ACM Reference Format:
Tianze Wu and Weisong Shi. 2026. Real-time Guarantee of Autonomous Driving Systems: State of the Art and Challenges. *ACM Trans. Internet Things* 1, 1, Article 1 (February 2026), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

AD has garnered significant attention in recent years, leading to substantial investment in the field [2, 20, 44, 45]. Despite this, the large-scale replacement of human drivers by autonomous systems remains elusive. Even optimistic projections have pushed the timeline for widespread deployment to 2030 [45], indicating that we are still far from achieving fully autonomous vehicles in a true sense.

Upon closer examination of the technical hurdles hindering the widespread adoption of AD, two primary factors emerge: the immaturity of application-level algorithms and the inability of in-vehicle systems to effectively guarantee real-time performance for current AD applications. While there has been rapid progress in algorithms and hardware, research into the real-time capabilities

Authors' addresses: Tianze Wu, SKLP, Institute of Computing Technology, CAS & University of Chinese Academy of Sciences, Beijing, China, wutianze@ict.ac.cn; Weisong Shi, University of Delaware, Delaware, USA, weisong@udel.edu.

2577-6207/2026/2-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

of in-vehicle systems remains in its nascent stages [64], with limited dedicated investigation into this crucial aspect.

Despite the recognized importance of real-time guarantees for AD systems in academic literature, in-depth discussion and analysis are often lacking. For instance, while Liu *et al.* [41] included timeliness as a key metric for AD computing, their work did not elaborate on the specific system designs that ensure this performance. Similarly, Lin *et al.*[38] analyzed acceleration hardware's impact on latency and Alcon *et al.*[1] measured timing variations in Apollo, revealing challenges in real-time analysis, but they did not delve into system design. Although [24] highlighted real-time requirements in the context of operating system security for autonomous vehicles and introduced ROS, it did not analyze the potential impact of security threats on ROS's real-time performance. Liu *et al.* surveyed latency optimization, their work was limited to vehicular edge computing scenarios and did not extend to the topic of real-time guarantees within the vehicle [44].

Most practitioners may possess a general understanding of real-time guarantees in in-vehicle systems, such as the knowledge that end-to-end latency should be controlled within 100 milliseconds [38, 41]. However, the specific measures employed by real-world in-vehicle systems to ensure this, whether the 100-millisecond requirement applies universally, and the considerations for application development and system orchestration remain largely unanswered questions. This paper will provide a comprehensive and systematic introduction to this topic, spanning from theoretical foundations to practical implementations, to help enhance the understanding of real-time guarantees in in-vehicle systems.

The first part of this paper will focus on theoretical introductions. This paper will present several common real-time modeling approaches for AD systems, illustrating how real-time theory is applied to AD and the challenges encountered.

The second part will delve into practical systems. We will introduce real-world system case studies, analyzing their task scheduling and communication details to provide insights into the methods employed by current AD systems to ensure real-time performance. While access to proprietary commercial systems is limited, this paper will focus on the mechanism analysis of several open-source systems that are utilized in actual AD solutions. The selection criteria for these systems are: 1. Open-source availability; 2. Adoption by AD projects at or above L2 level; 3. Significant influence within the academic or industrial communities.

The selected systems can be categorized into two types: operating systems and middleware systems. An operating system is an essential component of all computing systems, directly controlling hardware resources and providing interfaces for applications to utilize these resources. In the conventional automotive industry, prominent commercial real-time systems include QNX and VxWorks [41]. However, these are primarily designed for conventional vehicle control applications based on ECUs (up to L1 capabilities) and differ significantly from L2 and higher-level AD solutions in terms of complexity and hardware architecture. In principle, AD systems could run directly on an operating system without the need for an additional layer of middleware abstraction. However, the presence of middleware offers several advantages. Firstly, it facilitates ecosystem building. For example, ROS provides an abstraction layer that effectively decouples applications from the operating system, enabling the rapid utilization of advancements within the robotics community and fostering overall field development. Secondly, middleware provides capabilities for managing the system in user space (e.g., resource management, OTA updates, security) [34], primarily because the management mechanisms provided by operating systems may not fully meet the requirements, so through the middleware to make up. This paper will focus on introducing the following three middleware systems and one operating system in detail(listed in Tab.1):

- ROS: ROS (Robot Operating System) is an open-source middleware system that holds a dominant position in the robotics field [47]. ROS has now evolved to its second generation, ROS 2. ROS 2 optimizes ROS' communication mechanisms, security policies, apis, etc. Renowned open-source AD systems such as Autoware [32], Apollo.Auto [5], and the AD system company Apex.AI [3] have either been or are still built upon ROS. Many newly developed systems for AD research also use ROS as a foundation.
- Cyber: Cyber is the middleware used by Apollo.Auto, designed by Baidu for AD systems. Initially, the Apollo project used ROS 1 as its middleware. However, as the project progressed, ROS 1 revealed several issues, leading Baidu to fully discontinue ROS 1 after version 3.5 and switch to Cyber.
- ERDOS: ERDOS [26] is a middleware system designed and developed by the Berkeley RISELab. It is an implementation of the D3 execution model, which stands for Dynamic Deadline-Driven. It centers around deadline management, allowing tasks to adjust their computations to meet dynamic deadlines. ERDOS is utilized in the open-source AD solution Pylot [27], a modular, vision-based AD solution that runs in the Carla simulator environment.
- seL4 [52]: seL4 is a high-assurance, high-performance operating system microkernel designed to provide the highest level of isolation guarantees for multiple software components running in a system through its unique capability-based access control model. Its kernel has undergone comprehensive formal verification, using rigorous mathematical reasoning to precisely demonstrate the correctness and security of the code. NIO's Sky.OS, launched in July 2024, is developed based on seL4.

Name	Type	Provider	AD Solution or System	Application Area	Deployed on Vehicles
ROS2	Middleware	OSRF	Autoware, Apex	Robotics	Yes
Cyber	Middleware	Baidu	Apollo	AD	Yes
ERDOS	Middleware	RISELab	Pylot	AD	No
seL4	OS	seL4	Sky.OS	Areas with extremely high security requirements	Yes

Table 1. Selected Systems for Real-Time Guarantees in AD

The remainder of this paper is organized as follows. Section 2 introduces the background and fundamental concepts of real-time theory. Section 3 reviews existing modeling approaches, analyzing their limitations in accurately representing the characteristics of complex AD systems. Section 4 and Section 5 present a comparative study of scheduling and communication mechanisms across representative AD software platforms. Section 6 discusses emerging trends and outlines potential research directions. Finally, Section 7 concludes the paper.

2 BACKGROUND: REAL-TIME THEORY

This section briefly introduces some key concepts and definitions in real-time theory.

2.1 Task and Scheduler

A real-time system can be represented as a set of real-time tasks. Each task is typically modeled as an infinitely looping process, with each iteration of the loop referred to as a job of the task. (Generally, the terms "job" and "instance" are equivalent.)

The amount of computation time required by a task is called its execution time or runtime. Execution time refers to the actual time a task spends running, excluding any waiting time caused by preemption or suspension. Completion time refers to the duration from the task's release time to its finish time. Completion time is also commonly referred to as response time.

The real-time requirement is that the duration from the release of a job to its completion must be less than or equal to a certain value (the task's deadline). If this duration exceeds the deadline, the job is said to have a deadline miss.

The scheduler's role is to select the most appropriate tasks to run based on the available computing resources. A task set is considered schedulable if it can be proven that every task in the set can meet its timing constraints (real-time requirements). If there exists at least one scheduling strategy under which the task set is schedulable, then the set is called feasible [12]. The process of determining whether a task set is schedulable under a specific scheduler or strategy is known as a schedulability test.

2.2 Dependency

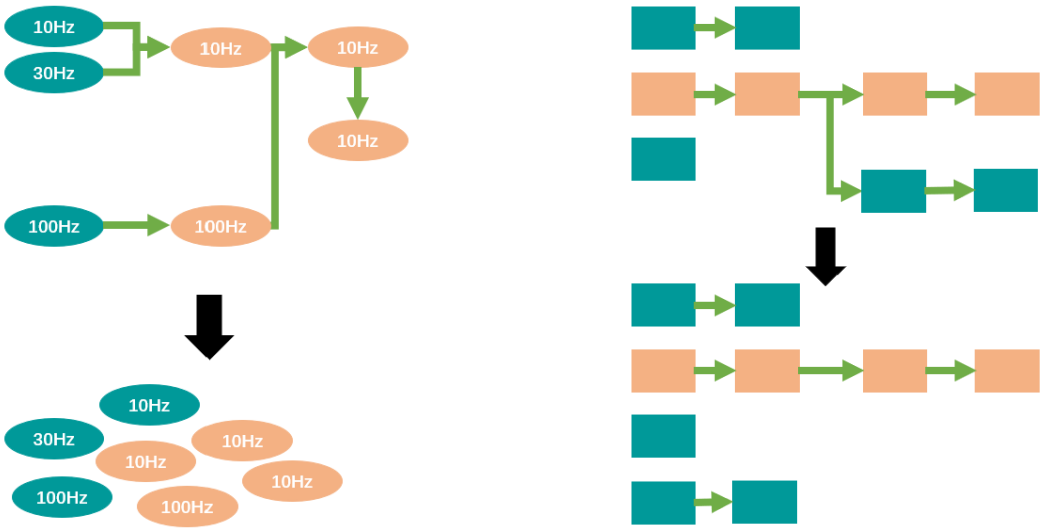
In many applications, tasks are not independent but have data dependencies (also known as job-level dependencies[7]). This means that the output of a preceding task (predecessor) serves as the input for a succeeding task (successor), following a read-execute-write semantic, similar to AUTOSAR's inherent communication model[50].

Task dependencies can be categorized into two types [33]: 1. Active (Event-Triggered) Dependency[25, 51]: A task is triggered for execution only after it has received all necessary input messages, potentially from different queues. In existing real-time analysis, event-triggered models typically assume sufficiently long queues to prevent message overwriting, ensuring that each output event from a predecessor triggers an execution of the successor. 2. Passive (Time-Triggered) Dependency[9, 17]: The execution of these tasks does not depend on the arrival of new messages. Instead, they follow periodic or sporadic execution patterns[39, 48], are triggered independently, and may have different periods (multi-rated systems)[8, 16, 19].

3 SYSTEM MODELS

3.1 Multi-rate DAG

3.1.1 Background. The multi-rate DAG model (shown in Fig.1a) was developed by researchers based on the electrified control systems in automotive chassis. The electrified control system in a car is a typical automated control system, similar to those used in aerospace and factory automation [22, 28, 29, 53]. Typical computational tasks running on it include engine control, braking control, and airbag control. Since these workloads do not inherently require significant processing power, their hardware platforms are typically characterized by multiple, less powerful Electronic Control Units (ECUs). Furthermore, due to hardware constraints, these ECUs generally only support operation modes triggered at fixed periods[21]. Different functionalities of the overall system are statically allocated to these ECUs, which are interconnected via a communication bus. Although high-level AD tasks are typically not executed on the vehicle's chassis now, the multi-rate DAG model remains a widely used model for AD systems in many research [30, 40, 56, 62], likely due to historical reasons.



(a) Multi-rate DAG Model (b) Processing Chains DAG Model

Fig. 1. Typical Models for AD system

3.1.2 Definition. We define a task model for AD systems as a multi-rate DAG model if it possesses the following characteristics:

(1) Each task in the task set is relatively independent, with a fixed execution period and deadline. The dependencies between tasks are passive.
(2) New messages will overwrite older ones (message queue of size 1).
(3) The entire system is structured as a Directed Acyclic Graph (DAG), where each computational task is represented as a node, and the dependencies between tasks form the directed edges.

3.1.3 Real-time Requirements. The real-time requirements for the multi-rate DAG model can be categorized into two aspects [56]:

- cyber part: Every instance of all tasks within the system must complete its execution before its specified deadline, ensuring the overall schedulability of the system.
- physical part: For data chains within the system, two newly defined constraints, reaction time and data age, are introduced [19]. In the context of AD, both metrics are crucial for the system's autonomous capabilities. Reaction time refers to the time elapsed from an external event occurring to the system outputting a control command; a longer reaction time implies a slower response to critical situations. Data age refers to the time from the generation of a source data point to the output of the last system control command influenced by it; a longer data age indicates a reduced awareness of the surrounding environment.

The AD system defined in an industry challenge[42] serves as a typical example of the multi-rate DAG model. The entire system is structured as a DAG, with each task operating at its specific frequency. It includes multiple distinct source nodes and a single sink node. Each source node represents a sensor mounted on the vehicle, while the sink node ultimately sends control commands to the vehicle's chassis. Sensors typically sample data at different frequencies. For instance, a camera might capture images at $30Hz$, while a LiDAR sensor might acquire point clouds at $10Hz$. These frequencies are often inherent to the sensor hardware. Some research efforts [43, 67] also explore

using unified signal triggering mechanisms to synchronize the sampling frequencies of different sensors. The industry challenge outlined three real-time requirements. Besides reaction time and data age, it also included a synchronization requirement at fusion nodes (such as the perception fusion node in [42]). This requirement specifies that the maximum difference in timestamps between messages from different channels during the fusion process must not exceed a certain threshold. Meeting these three requirements is crucial for ensuring driving safety.

3.2 Processing Chains DAG

3.2.1 Background. With the increasing level of intelligence and the rapid pace of technological advancements in vehicles, industrial control system hardware platforms are finding it challenging to meet the computational power, bandwidth, and development requirements of modern AD systems. Many AD solutions[5, 23, 32] now integrate modern computers directly into vehicles. The computer serves as the central processing unit (often called the Vehicle Control Unit) for handling AD and other intelligent tasks, and it does not have the operational constraints of traditional ECUs. The AD tasks are processed on VCU, and the results(control instructions) will then be sent to the car chassis for execution.

3.2.2 Definition. The processing chains DAG model is essentially a simplified multi-source DAG, as shown in Fig.1b. For some DAG systems with simple structures, a few minor modifications can transform them into the processing chains DAG model.

The main distinctions are as follows:

- Active Dependencies between Tasks: Without the fixed-time triggering constraints of ECUs, task execution is no longer strictly periodic. Instead, task dependencies are modeled as more intuitive active dependencies. Developers often prefer active dependencies because they are more resource-efficient (avoiding redundant processing of old inputs) and offer lower latency (processing can begin immediately upon receiving a new message without waiting for a periodic trigger). However, the default communication model for active dependencies assumes infinitely long message queues, preventing message overwriting. This assumption may not hold in real-world AD solutions, and some research [37] has addressed the topic of message queue lengths.
- Real-time Requirements: Unlike the multi-rate DAG model, the real-time requirements for the processing chains DAG model are simpler and more direct. The primary goal is to ensure that one or more critical processing chains can complete their execution within a specified time (response time or reaction time). Critical chains are typically those with the longest execution times or those responsible for the most crucial functionalities. The scheduling strategy should aim to minimize interference from other chains on these critical chains.

The system in [14] is a typical AD system modeled using the processing chains DAG approach. It consists of multiple trigger chains. The initial nodes of these chains are typically sensor nodes (often called source nodes), and these are the only tasks that execute periodically. All other tasks are triggered by active dependencies. Consequently, for an individual task, properties like its release time and deadline become somewhat redundant, as its execution depends solely on the completion of the jobs it relies upon. During analysis, tasks belonging to the same trigger chain are often assumed to share the same release time and deadline. We refer to one complete execution of such a chain as an instance of that chain.

3.3 Other Models

3.3.1 ROS-related Models. In recent years, there have been many real-time analysis and guarantee works for ROS2. Given the substantial influence of ROS in the AD field, this research has become

an integral part of real-time modeling for AD systems. Many evaluation scenarios in these works involve AD applications. This section introduces research related to real-time theory, and a more detailed discussion of ROS2's specific scheduling mechanisms and optimizations can be found in Sec.4.

The work by Casini et al.[13] was the first to conduct a real-time analysis of the ROS2 single-threaded executor, laying the groundwork for subsequent research in this area. Tang et al.[59] improved upon the analysis methods presented in [13] and revealed that the response time of a task chain in ROS is solely determined by the priority of the last callback in the chain. This finding suggests that the system's real-time performance can be enhanced through appropriate priority assignments. Blass et al.[10] built upon the work of [13] by considering the variability of actual execution times and a more detailed understanding of ROS2's scheduling mechanisms, leading to tighter response-time bounds. Tang et al.[60], addressing similar issues as in [10, 13], further enhanced the precision and efficiency of the analysis and compared the effects of different scheduling policies on the real-time performance of ROS2. The aforementioned research focuses on the ROS2 single-threaded executor. For the more intricate multi-threaded executor, Jiang et al.[31] provided the first response time analysis. Their findings suggest that improper use of the multi-threaded executor can increase the response times of chains, potentially even performing worse than employing multiple single-threaded executors. End-to-end real-time analysis in scenarios involving multiple single-threaded executors has also been explored in the work by Teper et al.[61]. Sobhani et al.[54] further extended the analysis by considering additional ROS2 features (mutually-exclusive callback groups) and a wider range of use cases, including chains with arbitrary deadlines and chains distributed across multiple executors.

3.3.2 Model Specifically Designed for AD. The work by Wu et al.[64] presents a system model specifically tailored for AD systems, although it is not strictly a task model within the real-time theory domain. The paper divides the AD system into four main modules: 1. A perception message preprocessing module running on the Body Control Modules(BCM); 2. A perception module running on the Central Computing Modules(CCM); 3. A decision-making and planning module running on the CCM; and 4. A safety protection module running on the BCM. The paper then connects these modules into a complete system using two execution paths. The main path is responsible for the core AD functionalities, providing soft real-time guarantees, while a secondary path acts as a safety fallback, offering hard real-time guarantees. Following the detailed modeling of the AD system, the paper uses perception and decision-making as examples to argue that the real-time requirements for AD systems cannot be defined in the same way as conventional real-time systems. Instead, they need to be tailored to the specific characteristics of each module or even each application. Furthermore, the authors suggest introducing a new metric, "result quality", to aid in defining the real-time requirements of AD systems.

3.4 The Gap between Theory and Practice

3.4.1 Over-simplification. To simplify analysis, both multi-rate DAG and processing chains DAG models make numerous simplifications—such as unified dependency types, idealized communication mechanisms, and simplified application logic. As a result, neither model can accurately capture the actual scenarios found in AD systems. The multi-rate DAG model describes more like a traditional ECU-based distributed automotive chassis system, while the processing chains DAG model even avoids addressing message fusion entirely, thus remaining far from the characteristics of real AD systems.

The fundamental reason for such simplifications lies in the overwhelming complexity of AD systems. Modeling and analyzing concurrency have long been challenging problems. Even for a single

concurrency paradigm, formal modeling and schedulability analysis are already computationally intensive [6, 35, 49, 57]. The AD system is a typical heterogeneous sensor system characterized by large scale and deep execution pipelines. Within such a system, multiple concurrency paradigms often coexist, leading to a rapid expansion of the state space and an exponential growth in system complexity. Without simplifying certain real-world factors, analysis would become intractable. However, these simplifications also cause a significant mismatch between the abstract model and the actual system, making it difficult to apply results derived from such models to practical AD systems.

3.4.2 Practical Dilemmas of Real-time Requirements. The models above primarily describe task dependencies and execution constraints from a computational or scheduling perspective, but they overlook the cyber–physical interactions fundamental to AD systems. Although real-time task models can formalize temporal dependencies and resource usage, they cannot capture how variations in control performance, perception accuracy, or actuator latency influence vehicle safety. This limitation often renders traditional real-time analyses overly conservative or misaligned with the actual safety and stability objectives of AD systems [15].

To bridge this gap, recent studies have advocated incorporating a CPS-oriented perspective into system modeling [46, 58, 63, 64, 66]. By integrating data ages, control errors, perception uncertainty, and physical constraints into task models, real-time analysis can more faithfully reflect the intrinsic characteristics and safety-critical behaviors of AD systems. However, these efforts are still in the early stage of exploration and are not yet capable of handling complex cases like AD system.

4 SCHEDULING

4.1 ROS2

In ROS2, the scheduling unit is "callback", which is triggered by external events such as the arrival of new input data or a timer reaching its set time. Callbacks belong to a Node, which is a user-level abstraction for managing related callbacks. Each callback is assigned to one callback group within its Node. Callback groups allow users to specify how their callbacks should be scheduled for execution. There are two types of callback groups: Mutually Exclusive Group and Reentrant Group. Callbacks within a mutually exclusive group execute sequentially. In contrast, callbacks within a reentrant group can execute concurrently. There are five types of callbacks in ROS2: timer callbacks (C^{tmr}), subscriber callbacks (C^{sub}), service callbacks (C^{srv}), client callbacks (C^{clt}), and waitable callbacks (C^{wat}).

An Executor manages the execution of callbacks, and callbacks are assigned to an executor via callback groups. At the operating system level, an executor can be thought of as a process. During the scheduling process, the executor maintains a dynamic ‘wait_set‘ that tracks all currently ready callbacks (in versions before Eloquent Elusor, timer callbacks were not included in the ‘wait_set‘). A callback is considered ready if it has pending messages to process and satisfies the requirements of its associated callback group.

4.1.1 Scheduling Process. The entity responsible for executing callbacks is the thread created by the executor. The execution flow of this thread is illustrated in Fig.2[13]:

(1) The point in time when the latest ‘wait_set‘ is obtained (thread safety is ensured through a mutex lock) is referred to as the polling point. The interval between two consecutive polling points is called the processing window. When the ‘wait_set‘ is empty, a polling action is initiated. The executor iterates through all its callback groups, checking if any callback within a group is ready to execute. If so, all callbacks in that group are marked for further inspection.

The executor then calls ‘rcl_wait‘ from the ‘rcl‘ layer to determine if these selected callbacks have been triggered.

(2) After obtaining the current ‘wait_set‘ at the polling point, the executor selects the highest-priority callback among all ready callbacks for execution, following a predefined priority order based on callback type (timer > subscription > service > client > waitable). Within the same callback type, callbacks registered earlier have higher priority. During the selection process, the executor re-checks if the callback's associated callback group is currently eligible for execution. Although a preliminary filtering occurs when populating the ‘wait_set‘, this second check is necessary because the ‘wait_set‘ is not updated after every callback execution. Callbacks executed within a processing window might alter the runnable status of their respective groups. If the group is not eligible, the callback is skipped, but it remains in the ‘wait_set‘. It's important to note that in earlier ROS2 versions (before Eloquent), timer callbacks (C^{tmr}) were handled separately[31]. They were not part of the ‘wait_set‘. Instead, after each callback completion, the system would access the ‘rcl‘ layer to update the set of all ready timers (effectively performing a separate polling for timer callbacks). Later versions treat timers as regular callbacks, managed uniformly with other types.

(3) Once a callback is selected, the executor first retrieves its input message. After obtaining the message, the callback instance begins execution, and the callback is removed from the ‘wait_set‘. The executor does not interrupt a running callback; it continues execution until completion. This represents a non-preemptive process at the middleware level. Therefore, application developers should design callbacks to avoid excessively long execution times or potential blocking. After a callback finishes execution, the thread attempts to retrieve the next runnable callback from the ‘wait_set‘ and continues the process. When no runnable callbacks remain in the ‘wait_set‘, the thread returns to the polling point to begin updating the ‘wait_set‘ for the next iteration.

ROS2 provides two types of Executors: SingleThreadedExecutor and MultiThreadedExecutor. A SingleThreadedExecutor creates and uses only one thread for execution. A MultiThreadedExecutor, on the other hand, creates multiple threads to execute the process in parallel.

4.1.2 Optimization Work. As observed, callbacks triggered during a processing window must wait for all callbacks in the previous ‘wait_set‘ and all timer callbacks within that window to complete before they can be executed, leading to a clear case of priority inversion. Consequently, many optimization efforts for ROS2 focus on increasing the frequency of ‘wait_set‘ updates, for example, by having the executor perform a polling operation after each callback execution. Another straightforward optimization involves moving away from using callback type and registration order as the basis for priority. Instead, a dedicated priority field is added to each callback, allowing the executor to determine the execution order based on these priority values. Different systems can assign values to this field according to their specific requirements. Many research works[4, 11, 14, 18, 18, 54] generally follow these two optimization strategies for ROS2.

4.2 Cyber

A key distinguishing characteristic of Cyber compared to other middleware systems is its shift of scheduling functionality from the kernel space to the user space, with coroutines replacing threads as the fundamental scheduling unit. Tasks within Apollo are executed by coroutines. The execution flow of a coroutine involves an infinite loop with three steps: 1. Attempting to acquire data; 2. If data is acquired, execute the user-defined data processing handler; 3. If no data is acquired or processing is complete, continue to sleep. Processors in Cyber serve to decouple coroutines from threads. They are responsible for retrieving ready coroutines (managing tasks) based on predefined rules

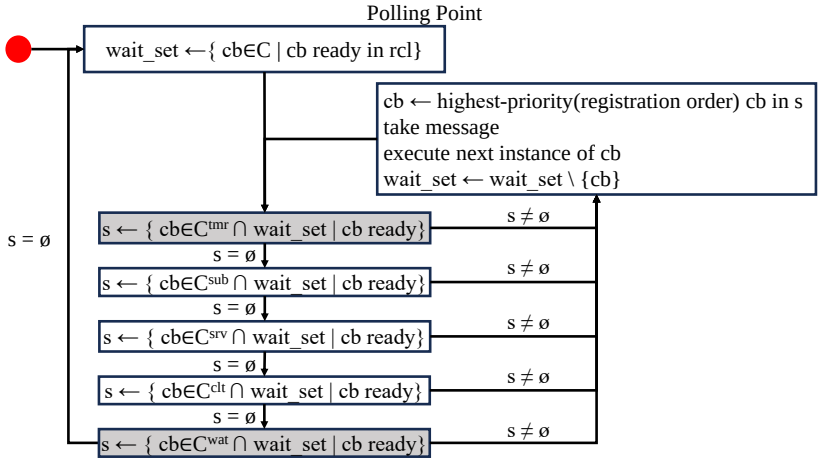


Fig. 2. Running Pattern of Executor Thread in ROS2.

and invoking resources to execute them (managing resources). Given the one-to-one relationship between processors and threads in Cyber, a processor can be simply understood as representing a thread.

Similar to the execution flow of executor threads in ROS2, processors in Cyber continuously attempt to acquire ready coroutines for execution. If no coroutines are ready, a processor will relinquish the CPU and wait until it is awakened. Task execution is also non-preemptive; for a given thread/processor, the current task must complete or voluntarily suspend before the next task can be acquired. Cyber implements assembly-level register switching code for coroutine switching within a processor. Since these operations occur in user space, they incur significantly less context switching overhead compared to operating system-level thread switching. Unlike ROS2, Cyber does not have a unified polling point for updating a 'wait_set'. Instead, whenever a new coroutine becomes ready, it is placed into a specific coroutine queue. The system then attempts to wake up an idle processor to execute the newly ready coroutine.

Cyber offers two scheduling schemes: the Classic scheme and the Choreography scheme.

- In the Classic scheme, tasks across the entire system are organized into multiple groups. As illustrated in Fig.3, coroutines, processors (and their associated threads), and system hardware (CPUs) are all assigned to specific groups. Within each group, it's possible to configure the number of processors (threads), CPU affinity (which CPUs are allocated), the scheduling policy of the threads (e.g., SCHED_OTHER, SCHED_FIFO in Linux), and the individual priorities of the coroutines. Importantly, within a group, there isn't a fixed mapping between processors and coroutines; instead, all processors within a group share a common array of coroutines. The coroutines within a group are structured as a two-level array, primarily ordered by priority, and secondarily by the order of addition for coroutines with the same priority. Processors always select the highest-priority ready coroutine for execution.
- The Choreography scheduling scheme categorizes all tasks into two types: critical tasks and ordinary tasks. For critical tasks, users must explicitly assign them to specific processors. Processors executing critical tasks maintain their own local priority queues. During operation, a processor will only look for the highest-priority ready coroutine within its local queue. Similarly, when a coroutine belonging to a critical task becomes ready, it will only wake up

the processor to which it is bound. For ordinary tasks, the Choreography scheme places them all into a default group. This default group functions similarly to the groups in the Classic scheme, where processors share a global two-level array of coroutines. Critical and ordinary tasks require separate configuration of their CPU sets, thread scheduling policies, and thread priorities, enabling resource isolation between these two types of tasks.

Cyber significantly reduces system scheduling overhead by employing coroutines. Given the substantial amount of I/O operations in AD tasks, performing most task switching in user space is a judicious design choice. Furthermore, because task switching is handled in user space, it allows for the configuration where each CPU core runs only a single, fixed thread/processor. This can further minimize the overhead associated with thread migration within the system.

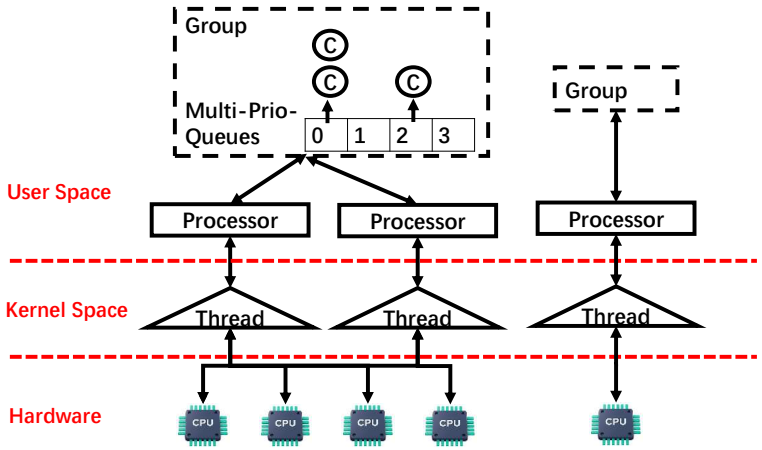


Fig. 3. Architecture of Cyber.

4.3 ERDOS

The core design principle of the ERDOS middleware is D3, which stands for Dynamic Deadline-Driven. The authors of ERDOS posit that AD systems must maximize the quality of their output within a specified timeframe (before a deadline), and system design must be centered around this novel requirement. Before delving into its scheduling mechanism, we will first provide a brief overview of ERDOS's design. ERDOS models an AD system as a Directed Acyclic Graph (DAG), where each node in the graph is termed an Operator, effectively representing a functional module within the system. To realize the D3 concept, ERDOS primarily focuses on the following aspects:

- Defining the system's end-to-end latency deadline based on environmental factors such as vehicle speed and scene complexity.
- Allocating individual deadlines to each stage (referred to as Operators in ERDOS) along the processing pipeline, derived from the overall end-to-end deadline.
- Monitoring the execution of each operator and dynamically adjusting its computational behavior based on its allocated deadline to maximize output quality (the paper proposes several techniques, including anytime algorithms, multi-version execution, and early termination).
- Implementing timeout handling when an operator exceeds its deadline, including dynamically readjusting the deadline allocations for downstream operators.

This paper will not delve into the intricate details of ERDOS. Our focus will be on how ERDOS handles system scheduling. As depicted in Fig.4, scheduling in ERDOS operates at the level of all operators within a Node. The Node is a concept in ERDOS's communication topology, which we can temporarily consider as a single machine (although technically it's a process, typically only one ERDOS process runs per machine). Upon node creation, a thread pool is established using Rust's Tokio library. The number of threads in this pool typically matches the number of available CPU cores to minimize thread switching overhead. The smallest schedulable unit in ERDOS is a coroutine, as defined by Tokio. Similar to the coroutines in Cyber, these are lightweight task units residing in user space.

Coroutines in ERDOS have two primary responsibilities:

(1) ‘operator_executor‘: There is a one-to-one correspondence between an ‘operator_executor‘ and an operator. It has two primary responsibilities: 1. Listening for incoming messages to its associated operator. Upon receiving a new message, the ‘operator_executor‘ encapsulates it into an event and inserts it into the DAG within the operator (ERDOS refers to this as a lattice; we will not delve into the specifics of the lattice, but it's important to know that events have dependencies and are stored in a specific order to ensure correct processing). Finally, it notifies the ‘event_runner‘ to process the event. 2. Managing the deadline constraints defined for its operator. This involves two main tasks: first, upon receiving a regular message, it records the arrival time and starts a countdown timer (implemented using Rust's ‘futures-delay-queue‘; further details are omitted here). Second, in the event of a timeout, the ‘operator_executor‘ invokes the designated timeout handling function.
(2) ‘event_runner‘: Responsible for monitoring and processing events generated by the ‘operator_executor‘. When an ‘event_runner‘ detects the arrival of a new event, it first identifies the corresponding operator. Then, it processes the events sequentially according to their order within the lattice (a lock mechanism is used to prevent race conditions when multiple ‘event_runners‘ attempt to retrieve events). Processing all events for a given operator sequentially is intended to prevent event starvation.

For each operator managed by a node, a corresponding ‘operator_executor‘ is created. The number of ‘event_runners‘ is determined empirically (currently, it's set to the maximum of the number of threads and the number of operators within the node). These created coroutines are then managed by Rust's Tokio library for scheduling. These coroutines themselves do not have any explicit priority or other scheduling attributes set. However, it's evident that within each operator in ERDOS, the event processing has an inherent priority due to the ordered lattice structure maintained for each operator. ERDOS primarily sorts events based on their timestamp and message type, suggesting that event processing within an operator largely follows a chronological order.

Tokio's multi-threaded scheduler employs the following strategies:

(1) Cooperative Scheduling: Tokio employs cooperative scheduling, meaning that coroutines must explicitly yield control of the CPU to allow other coroutines to run. This approach prevents any single coroutine from monopolizing the CPU for extended periods, ensuring fair execution opportunities for all tasks.
(2) Priority for Local Run Queues: Each thread maintains its own local run queue to store coroutines ready for execution. Threads prioritize executing coroutines from their local queue to minimize cross-thread communication overhead. In addition to local queues, Tokio also has a global queue for coroutines that are shared across all threads (these might be coroutines that couldn't fit in a local queue or were not woken up by a worker thread). When a local queue is empty or a thread has executed a predefined number of coroutines from its local queue, it will then retrieve coroutines from the global queue.

(3) Work Stealing: When both the local and global queues are empty, a thread will attempt to "steal" tasks from the task queues of other threads, typically taking about half of the coroutines from another thread's queue. This strategy effectively leverages the benefits of multi-core CPUs, enhancing overall concurrency performance.
(4) Monitoring I/O and Timer Events: If there are no runnable coroutines available, or if Tokio has continuously scheduled and executed a set number of coroutines, it will then check for pending I/O or timer events. If any are found, the corresponding coroutines are woken up and added to a run queue.
(5) LIFO Slot: When a coroutine wakes up another coroutine (e.g., by spawning a new one or sending a message through a channel), the newly awakened coroutine is added to a special "LIFO slot" associated with the worker thread, rather than directly to a queue. After the thread finishes executing its current coroutine, it will immediately execute the coroutine in the LIFO slot. To prevent starvation of coroutines in the local queue, Tokio ensures that after a certain number of consecutive executions from the LIFO slot, a coroutine from the regular queue must be executed. The LIFO slot mechanism exploits spatial locality, leading to performance gains, and also reduces message transmission latency by minimizing queuing delays during message passing.

As we can see, Tokio employs several different queues to strive for both fair scheduling and high performance, while also incorporating mechanisms to prevent potential starvation issues. However, it's important to note that Tokio is not inherently a real-time scheduler and therefore cannot provide strict real-time guarantees for the system. ERDOS's real-time optimizations are primarily focused at the application level and have demonstrated promising results, which can offer valuable insights for researchers dedicated to optimizing the real-time capabilities of the system itself.

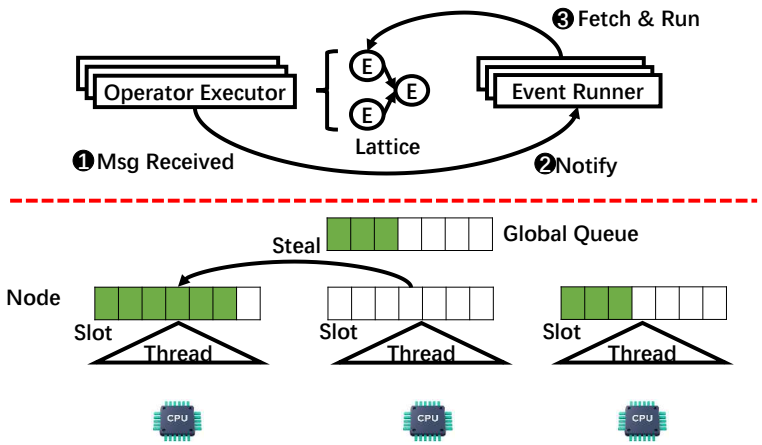


Fig. 4. Architecture of ERDOS.

4.4 seL4

In seL4, threads are the unit of scheduling. seL4 employs a preemptive round-robin scheduler that supports 256 priority levels. Each thread is assigned a Maximum Control Priority (MCP) and a normal priority. The normal priority represents the thread's actual runtime priority and can be modified, while the MCP defines the upper limit of priority that the thread is permitted to set. The

scheduler selects the highest-priority ready thread for execution. If multiple threads have the same priority, they are scheduled using a First-In, First-Out (FIFO) round-robin approach, where each thread runs for its allocated time slice before a context switch occurs.

To enhance system isolation, seL4 offers a hierarchical scheduling mechanism known as Domain Scheduling. A Domain represents a scheduling partition that encapsulates a set of threads and associated resources (such as memory and capabilities). Each domain operates within its own address space, ensuring isolation from other domains. A top-level scheduler manages the scheduling between different domains, while each domain can have its own internal scheduler responsible for scheduling the threads within it. The top-level scheduler employs a round-robin algorithm to cycle through the different domains. Each domain is allocated a fixed time slice, and when this time slice expires, the scheduler switches to the next domain in the round-robin order. The scheduling order and time slice allocation for domains are statically configured at compile time and cannot be dynamically modified during runtime. This static configuration guarantees the determinism and predictability of the scheduling. The switching between domains is non-preemptive, meaning that once a domain starts executing, it will not be interrupted by other domains until its allocated time slice is exhausted. If a domain initiates inter-process communication (IPC) with another domain during its execution, this communication will only be processed when the initiating domain's time slice ends and the scheduler switches to the target domain, potentially introducing some latency.

seL4's scheduling design adheres to the principles of simplicity and efficiency. It provides a high degree of determinism through fixed priorities, preemptive scheduling, and hierarchical domain management. Furthermore, its straightforward logic results in minimal scheduling overhead.

4.5 summary

Table 2. Comparison of Scheduling Mechanisms among ROS2, Cyber, ERDOS, and seL4

	ROS2	**Cyber**	**ERDOS**	**seL4**
Scheduling Unit	Callback	Coroutine	Coroutine	Thread
Core Scheduler	Executor (SingleThreaded / MultiThreaded) with polling "wait_set"	Processor–Coroutine model with Classic or Choreography scheme	Operator-level scheduling & Tokio runtime thread pool	Preemptive round-robin with 256 priorities; hierarchical domain scheduling
Priority Control	By callback registration order and callback types	Per-group coroutine priority and CPU affinity; configurable scheduling policy	Deadline-driven (D3 model)	Static priority per thread; MCP limits dynamic changes
Scheduling Level	User space	User space & Kernel space	"Application" space & User space	Kernel space
Allow Preemption	No	No	No	Preemptive within domain, non-preemptive between domains

Here, we summarize two observations on the current scheduling mechanism.

4.5.1 middleware-centric system architecture. The boundary between OS and middleware has become increasingly blurred, driven by two key factors: (1) the ongoing "micro-kernelization" trend, which aligns with the safety and real-time requirements of AD systems; and (2) the overlapping responsibilities of both layers in system management, which introduce unnecessary coordination overhead [36].

We foresee a future trend toward tighter integration. Real-time OS will continue shedding non-essential functionalities to enhance determinism—aiming for full preemption, higher-resolution timers, and minimal kernel complexity. Their core responsibility will be limited to hardware abstraction, with little or no additional functionality such as scheduling or memory management policies.

Middleware, on the other hand, will increasingly take charge of system optimization in conjunction with application-layer solutions. Operating in user space allows middleware to access rich runtime context with minimal cost, enabling dynamic, application-aware optimization. This middleware–OS integration represents a promising direction for future AD system platforms, combining real-time determinism with flexible, high-level adaptivity.

4.5.2 underutilized scheduling capabilities. While this section detailed how these systems provide real-time guarantees by addressing issues like priority inversion and minimizing task switching overhead, the crucial factor in determining the safety and reliability of an AD solution lies in how effectively these capabilities are leveraged. This includes configuring task priorities and designing scheduling algorithms appropriately. In our assessment, the AD solutions—Autoware, Apollo, and Pylot—have not fully exploited these capabilities:

- **Autoware** explicitly states that its current focus is on implementing functionalities, with real-time considerations being secondary.
- **Apollo** employing two intuitive coarse-grained scheduling strategies: 1. Assigning higher priorities to tasks later in the execution pipeline based on task dependencies to mitigate priority inversion. 2. Allocating dedicated executors to critical tasks to isolate them from potential interference.
- **Pylot** does not fully utilize the trade-off capabilities provided by ERDOS. Only a small portion of the system can change its execution logic with deadlines. We also noticed that although ERDOS proposed that the deadline should change with the scenario, only two scenario options are provided. How to identify scenario types and set appropriate deadlines remains unsolved.

5 COMMUNICATION

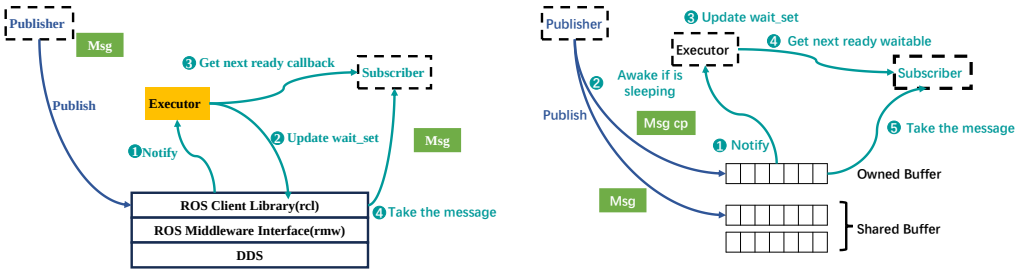
5.1 ROS2

ROS2 provides two communication pathways for pub-sub communication pattern: one leveraging the underlying DDS message middleware, and the other being ROS2's native intra-process communication.

DDS Path: ROS2 essentially delegates the communication functionalities to the underlying DDS middleware. The process of a message being sent and received via the DDS path is(refer to Fig.5a):

(1) The user creates a message and invokes the 'publish' function to send it. This process involves adding the user-created message to the send queue of the corresponding publisher within the underlying DDS middleware. This operation is asynchronous, meaning the 'publish' function returns immediately, and the remaining message transmission is handled entirely by the DDS middleware based on the configured QoS policies.
(2) Assuming the DDS middleware successfully delivers the message to the receive queue on the subscriber side (the receive queue managed by the DDS middleware), the DDS middleware then sets the state of the receiving subscriber to "ready" and notifies any waiting threads in the upper layer.
(3) A thread within the executor, after completing all tasks currently in its 'wait_set', enters a polling point to repopulate the 'wait_set'. This involves actively checking for any callbacks that are ready to be executed, adding them to the 'wait_set', and then executing these callbacks sequentially. Assuming the subscriber is found to have received data at the polling point of this cycle, it is placed in 'wait_set'.
(4) The executor continuously processes callbacks in the order they appear in the 'wait_set'. The next callback to be executed is the subscriber's callback. The executor first marks the state of

the subscriber's callback group (if it's a mutually exclusive type, other callbacks within the same group cannot run concurrently). Then, it retrieves the received message from the DDS middleware and finally executes the user-defined callback function to process the message.



(a) DDS path (b) Intra-Process Path

Fig. 5. Data Paths of ROS2

Intra-Process Path: Intra-Process communication is an optimization in ROS2 specifically designed for communication between nodes residing within the same process (and therefore managed by the same executor). This approach allows publishers and subscribers in the same process to exchange messages without incurring the overhead of data copying, thus enhancing communication efficiency. Unlike the DDS path, intra-process communication primarily utilizes waitable callbacks for message delivery. When a publisher and subscriber are created, enabling intra-process communication is specified. To minimize data copying, ROS2 distinguishes between two types of subscribers when establishing this connection: 'Take_shared' and 'Take_ownership'. The specific type of subscriber depends on whether the message input type of its defined callback function is a 'shared_ptr' (a C++ smart pointer indicating shared ownership of the data). While we won't delve into the intricacies here, it's important to understand that users can decide whether the callback needs to take ownership of the message based on their specific requirements (e.g., if the message needs to be modified or further passed along). This decision impacts the level of optimization achievable with intra-process communication (zero-copy can be achieved if ownership is not required). The process of a message being sent and received via the intra-process path is(refer to Fig.5b):

(1) The user creates a message and calls the 'publish' function to send it. In contrast to the DDS path, the intra-process path strives to minimize data copying. Therefore, during sending, the publisher will copy the message based on whether any of the subscribers to that topic require ownership of the message. Ideally, if none of the subscribers need ownership, this message transmission can achieve zero-copy (not considering potential inter-process communication).
(2) Each subscriber maintains its own message storage queue (represented as "Owned Buffer" and "Shared Buffer" in Figure 5b). Once a message is placed into this queue, the executor is notified of the new message arrival. The subsequent processing is similar to the DDS path, with the key difference being that the callback function type used for intra-process communication is waitable.

5.1.1 Message Fusion. In systems organized as a Directed Acyclic Graph (DAG), message fusion scenarios, often referred to as fusion nodes, are common. ROS2 provides a message filter mechanism to handle these message fusion tasks. As illustrated in Fig.6, this mechanism is built upon the foundation of subscriber callback functions. Initially, all subscribers within the message filter

(corresponding to the topics that need to be fused) are assigned a default callback function. Within these callbacks, the received messages are placed into a temporary queue specific to each subscriber (the data structure used here can be customized based on the synchronization policy). Subsequently, the system checks if the current set of messages satisfies the fusion criteria defined by the chosen fusion strategy. If the criteria are met, the user-registered callback function is invoked, and the fused message is passed as an argument. If the conditions are not met, the system returns and waits for one of the subscribers to receive a new message, at which point the process repeats. It's important to note that the actual user-defined callback function for processing the fused message is not directly managed by the executor. Instead, it exists as part of the callback chain of these special subscribers. Its execution is contingent on when the callbacks of those subscribers are scheduled and whether the fusion conditions are met.

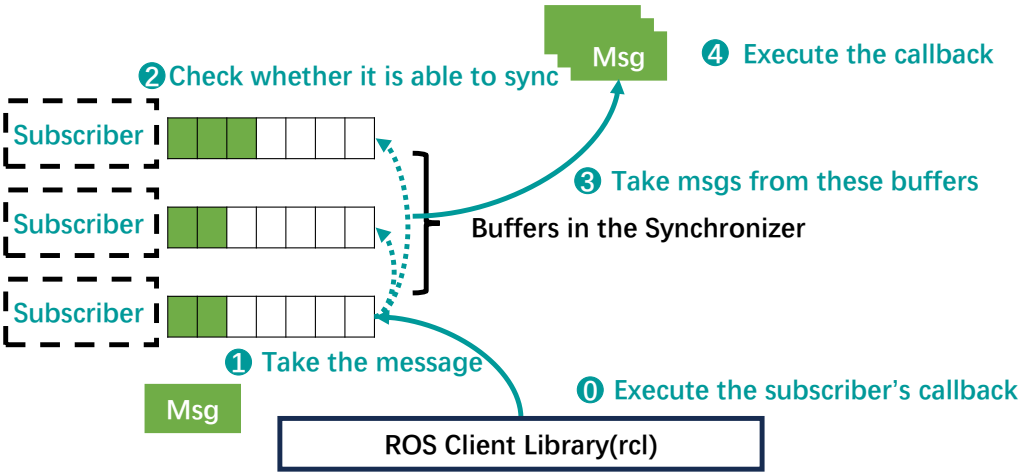


Fig. 6. Data Fusion Mechanism of ROS2

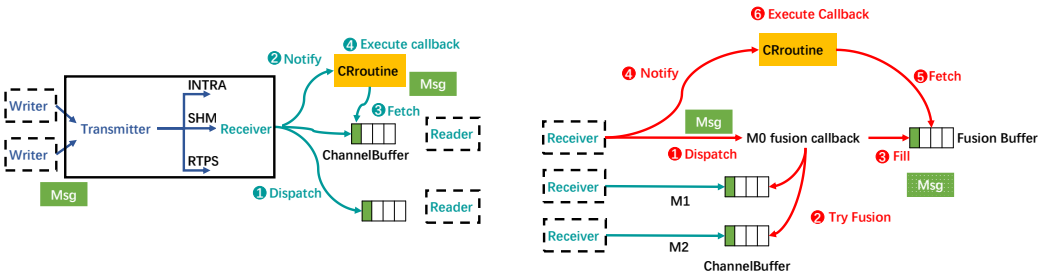
5.1.2 QoS Options. The QoS options for communication empower users to customize the behavior of their communication. This functionality is a standard feature in virtually all systems that support message-based communication. In ROS2, for instance, there are currently eight standard QoS policy types available for configuration: History (how messages are kept), Depth (the size of the message queue), Reliability (guarantee of message delivery), Durability (persistence of messages), Deadline (expected maximum delay), Lifespan (maximum message validity period), Liveliness (mechanism to detect if a publisher is still active), and Lease Duration (maximum time a publisher can be considered alive). Beyond these eight standard options, DDS vendors often provide additional QoS policies specific to their implementation, which users can configure directly by bypassing the ROS 2 layer. However, doing so can lead to tighter coupling between the application and the specific DDS middleware being used. When utilizing the QoS mechanism, users can not only configure communication behavior but also monitor QoS-related events. They can provide callback functions that are executed in response to these events, allowing for custom handling to meet specific application requirements. In ROS2, callbacks associated with QoS events are of

the C^{wat} type, and their triggering process is similar to the intra-process communication path described earlier. When configuring QoS settings, it's crucial to ensure compatibility between the communicating endpoints (e.g., a publisher and a subscriber). For example, you cannot set a publisher's reliability to "Best Effort" while configuring the subscriber's reliability to "Reliable." Given that the fundamental principles of QoS mechanisms are largely consistent across different systems, we will avoid redundant explanations in subsequent sections.

5.2 Cyber

Cyber's communication mechanism shares numerous similarities with ROS2. It also supports both publish-subscribe (pub-sub) and service-client communication patterns, with this section focusing on the pub-sub model. In Cyber, the counterparts to ROS2's Publisher, Subscriber, and Topic are termed Writer, Reader, and Channel, respectively. Given the intricate hierarchical structure of Cyber's codebase, which involves many new terms and implementation specifics, we will present a streamlined overview, and the subsequent content will not directly map one-to-one with the Cyber source code.

As illustrated in Fig.7a, the underlying entities in Cyber responsible for sending and receiving messages are the Transmitter and Receiver. A single Transmitter or Receiver corresponds to a Channel, establishing a many-to-one relationship with Writers and Readers. There are three communication pathways: intra-process (INTRA), which is similar to ROS2; DDS (RTPS), also analogous to ROS2; and an additional shared memory (SHM) path unique to Cyber. When establishing the connection between a Receiver and a Transmitter, Cyber automatically selects the appropriate path. If they reside within the same process, the INTRA path is used. If they are on the same machine but in different processes, the SHM path is utilized. For communication across different machines, the RTPS path is employed. Cyber also provides configuration options allowing users to explicitly specify a communication path, although user-specified paths must adhere to logical constraints (e.g., RTPS can be used for intra-process communication, but INTRA cannot be used for cross-machine communication). We will first describe the general communication flow and then detail the distinctions between these different pathways.



(a) Hybrid Path (b) Data Fusion

Fig. 7. Communication Mechanism of Cyber

5.2.1 *Communication Flow.*

(1) Cyber employs three Transmitters, each corresponding to one of the three communication paths. When a Writer and a Reader are created, the necessary path is automatically enabled. For instance, if a Reader on the same machine but in a different process subscribes to a

Channel, the Writer's SHM communication path is activated. When a message is published, Cyber sends it through all the enabled paths.

(2) Upon the arrival of a message at the receiving end (the arrival method varies depending on the path), the Receiver is notified of the new message (the notification mechanism also differs across communication paths). The Receiver then performs three key actions: 1. It identifies the Channel associated with the new message and subsequently finds all Readers subscribed to that Channel. 2. It places a pointer to the new message into the ChannelBuffer of each of these Readers. 3. It notifies the coroutine associated with each Reader. It's crucial to note that to achieve zero-copy, Cyber does not differentiate based on whether a Reader needs ownership of the message. Consequently, users should refrain from directly modifying the message within their callback functions to prevent data inconsistencies.

(3) Once the Reader's coroutine is awakened, it waits to be executed by the Cyber executor. Upon execution, the coroutine retrieves the new message from the ChannelBuffer and then invokes the user-defined callback function to process it.

The following outlines how the Dispatcher for each communication path detects the arrival of a new message:

- INTRA: As the Transmitter and Receiver reside within the same process, this communication path is the most straightforward. When the Transmitter on the INTRA path sends a message, it directly identifies the corresponding Receiver based on the channel information and then invokes the appropriate callback function.
- SHM: In SHM mode, the Transmitter needs to convey the message to the Receiver via shared memory. When the 'Transmit' function is called, the SHM mode writes the message into a pre-allocated shared memory segment (Cyber optimizes this by allowing messages to be created directly within the shared memory, further reducing copy operations). Subsequently, the Receiver is notified of the new message arrival (Cyber provides two notification mechanisms, which will be elaborated upon later). Unlike the INTRA mode, SHM utilizes a dedicated thread (a separate thread, not a coroutine; Cyber's scheduling policy allows for independent configuration of SHM worker threads) to monitor for incoming messages. When this monitoring thread detects a new message, it executes the corresponding callback function (the Dispatch process).
- RTPS: The RTPS path is the simplest in terms of message sending, as the Transmitter directly calls the interfaces of the underlying DDS middleware. On the Receiver side, the dispatch function is also directly invoked by the callback function of the Subscriber within the DDS middleware. Therefore, Cyber does not need to manage any additional listening threads for this path. The configurable QoS options for RTPS can also be specified directly within Cyber, and since they are quite similar to those in ROS2, we will not provide a separate explanation here.

5.2.2 Message Fusion. Cyber employs a fixed strategy for message fusion. It designates one channel as the main channel. Whenever the main channel receives a new message, a fusion operation is triggered. This operation retrieves the most recent messages from all other channels involved in the fusion (messages from these other channels can be reused, while the main channel's new message initiates the fusion and is thus not reused). These retrieved messages are then combined with the main channel's message to create the final fused message.

The underlying implementation of message fusion in Cyber follows a straightforward principle. In the standard message reception process, the Dispatch process involves placing the received message into the ChannelBuffer associated with the Reader. However, in fusion scenarios, specifically for the designated main channel (M0 in Fig.7b), the Dispatch process is modified. Instead of placing

the new message from the M0 channel into M0's ChannelBuffer, the fusion procedure is executed. This procedure attempts to retrieve the latest messages from the other participating Channels. If successful, these messages are then combined into a single fused message and stored in a dedicated fusion buffer. For clarity in the preceding explanation, we assumed that each Reader has a user-registered callback function for processing messages. However, in reality, user-registered callback functions are associated with a Component rather than directly with a Reader (while other Readers might have callbacks, they are less relevant to the main fusion flow and have been omitted for brevity; interested readers can consult the source code of Cyber's Blocker component). A Component can be viewed as a functional module that can subscribe to 1 to 4 Channels. Its callback function is designed to process the messages fused from these Channels. Once the fused message is placed into the fusion buffer, the coroutine responsible for processing this message is awakened (this coroutine belongs to the Component, not to any specific Reader). This coroutine then attempts to retrieve the fused message from the fusion buffer and proceeds with its processing.

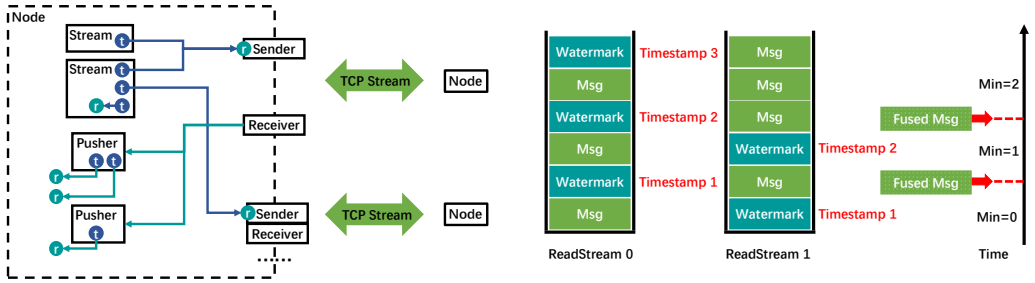
5.3 ERDOS

In ERDOS, communication between operators is established through Streams (referred to as OperatorStream in the source code, but simplified to Stream for brevity). Unlike a normal topic, a Stream in ERDOS is a one-to-many structure, which aids in tracking and managing data flow. Stream is a logical construct, and its underlying implementation relies on the mpsc (multi-producer single-consumer) asynchronous communication channel mechanism provided by the Tokio library, as depicted in Fig.8a. The dark blue circles represent the producers (tx) in the mpsc channels, while the light blue circles represent the consumers (rx). During the initial setup of ERDOS, the data pathways are initialized based on the location of the operators. We will now describe the different scenarios:

- Inter-thread: When two operators reside within the same node (equivalent to the same process), ERDOS employs zero-copy to optimize communication. During the establishment of a connection, the transmit (tx) and receive (rx) ends of the mpsc channel are directly associated with the stream. When a message is sent, only the memory address of the message is passed, avoiding any data copying.
- Inter-node: If two operators are located in different nodes, communication is facilitated through TCPStream. TCPStream is a structure provided by Tokio for asynchronous TCP connections, supporting asynchronous read and write operations. Upon node initialization, it begins establishing point-to-point TCPStream connections with other nodes. For each TCPStream connection, the node creates a corresponding Sender, responsible for sending messages to other nodes, and a Receiver, responsible for receiving messages. When a sender is created, it also creates an mpsc channel, retaining the receive end (rx) for itself and registering the transmit end (tx) with all streams that require inter-node communication. ERDOS launches a dedicated coroutine for each sender. This coroutine continuously monitors its rx end for new messages and, upon receiving one, transmits it via the TCPStream. The receiver operates in reverse. Each node maintains a map of streams to Pushers for all its receivers. A Pusher stores all the rx ends of the mpsc channels that subscribe to a particular stream on that node. ERDOS also starts an independent coroutine for each receiver. This coroutine listens for incoming messages from the TCPStream. Upon receiving a message from another node, it uses the message's destination stream information to find the corresponding pusher and then utilizes all the tx ends within that pusher to forward the message to the appropriate operator.

When an operator wants to write a message to a stream, it invokes all the txs associated with that stream to send the message to each subscriber. In the case of an inter-thread path, the message's

memory address is directly passed through the tx-rx channel. For an inter-node path, the message is first sent to the sender responsible for the current node. The sender then transmits the message to the destination node via TCPStream. Upon receiving the message, the receiver on the destination node uses the stream information embedded in the message to locate the corresponding pusher. The receiver then utilizes each tx within the pusher to forward the message to the appropriate rx of the subscribing operator. The operator's execution coroutine listens to its rx end and performs the necessary processing upon receiving a message.



(a) Inter-thread & Inter-node Path (b) Data Fusion

Fig. 8. Communication Mechanism of ERDOS

5.3.1 Message Fusion. ERDOS utilizes the concept of watermarks, which is common in stream processing systems, to assist in tracking the progress of data flow. A watermark is a special type of message that signifies that all regular messages with timestamps preceding a certain point in time have arrived. Message fusion is essentially synchronizing the state of multiple data streams, making watermarks an ideal trigger for fusion operations. ERDOS currently provides an example demonstrating the fusion of two input streams(depicted in Fig.8b). An operator with two input streams receives both regular messages and watermarks from upstream. Upon receiving a watermark, it indicates that all regular messages with timestamps earlier than or equal to the watermark's timestamp have been processed by the upstream operator and received by this stream. ERDOS maintains a 'MinTimestamp' for such fusion operators, representing the minimum timestamp among all received watermarks across the input streams. In the example, the initial 'MinTimestamp' is 0. When ReadStream 1 receives a watermark with a Timestamp of 1, 'MinTimestamp' remains 0 until ReadStream 2 also receives a watermark with a Timestamp of 1, at which point 'MinTimestamp' is updated to 1. ERDOS identifies opportunities for message fusion by detecting changes in the 'MinTimestamp'. Implementation-wise, when the 'MinTimestamp' changes, the 'Operator_Executor' generates an event. The processing of this event, which involves invoking a callback function, constitutes the message fusion process. In ERDOS' example, the callback function for a ReadStream might update an internal state, while the callback function for the watermark (which is essentially a callback triggered by the 'MinTimestamp' update, presented to the user as a watermark callback) generates the final fused result based on the states associated with the two ReadStreams. This illustrates that ERDOS delegates the specific logic of message fusion to the user, allowing for the implementation of diverse fusion strategies by customizing the callback functions for both regular messages and watermarks.

5.4 seL4

The initial development of the seL4 system (whose predecessor was L4) was driven by the need to optimize Inter-Process Communication (IPC), which in the context of seL4 is more accurately described as inter-thread communication. Communication in seL4 is facilitated through endpoints, as illustrated in Fig.9. An endpoint maintains two queues: one for threads waiting to receive messages (receiver threads) and another for threads waiting to send messages (sender threads). If the number of threads on both sides matches, each sender thread is paired with a receiver thread to complete message transmission. If there are surplus threads on either side, they are placed in the respective queue, awaiting a corresponding thread from the other side. Each thread has an associated buffer for message storage. When a sender thread sends a message, it informs the kernel of the message's length, and the kernel then copies that portion of the message from the sender thread's buffer to the receiver thread's buffer. Similar to its scheduling mechanism, seL4's IPC communication is designed to be highly simple and concise, providing a foundation upon which users can build more elaborate communication patterns, such as publish-subscribe.

seL4 primarily employs the following two techniques to optimize the performance of IPC communication:

(1) Utilizing Physical Registers: Standard IPC communication in seL4 involves using thread buffers for message transfer. These buffers reside in a fixed memory region, and message transmission typically requires copying. However, for scenarios involving small messages, seL4 can optimize this by using physical registers directly for transmission. In register-based message passing, the kernel only needs to restore the relevant values from the sender thread's TCB to the physical registers during a thread context switch. The receiver thread can then directly read the message data from these registers, effectively achieving zero-copy communication.
(2) Fast Path: The Fast Path is a highly optimized IPC pathway offered by seL4. When a message is being transmitted, the kernel performs several prerequisite checks. If these conditions are met, the communication proceeds through the Fast Path. This pathway incorporates a range of optimizations to enhance communication efficiency, including minimizing kernel operations, streamlining code paths to reduce CPU branch instructions, employing code optimizations, and leveraging compiler optimizations.

5.5 summary

Table 3. Comparison of Communication Mechanisms among ROS2, Cyber, ERDOS, and seL4

	ROS2	Cyber	ERDOS	seL4
Communication Paths	Intra-process & DDS	Intra-process & Shared-memory & DDS	Intra-process & TCP	Intra-process
Message Fusion / Synchronization	Synchronizer mechanism; user-defined callback fusion	Main-channel-based fusion	Watermark mechanism; timestamp-based fusion triggering	Not applicable (low-level IPC only)
QoS Configuration	QoS for DDS	QoS for DDS	None	None
Zero-copy Support	Intra-process	Intra-process & Shared-memory	Inter-thread	Register-based

Here, we summarize the common technical characteristics across these systems and offer supplementary details on these technologies.

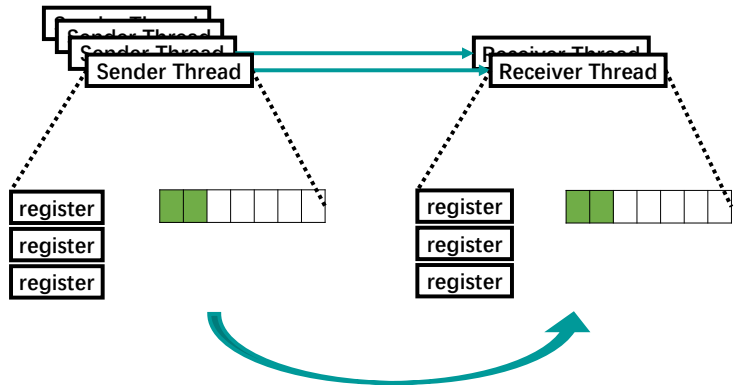


Fig. 9. Communication Mechanism of seL4.

5.5.1 Zero-Copy. Traditional IO copying involves data interaction between user space, kernel space, and hardware. A message is copied from user space to kernel space, and then from kernel space to the network card. This includes two context switches and two data copies. If the subscriber receiving process is also considered, there are four context switches and four data copies.

Zero-copy technology generally reduces CPU involvement through Direct Memory Access (DMA) technology, and it has been widely used in various mainstream hardware platforms. There are many ways to reduce context switching and data copying. Common methods include: 1. By memory mapping (mmap in Linux), the address of the read buffer in the kernel is mapped to the user space buffer, thereby achieving sharing of the kernel buffer and application memory. This eliminates the process of copying data from the kernel read buffer to the user buffer. However, the kernel read buffer still needs to copy the data to the kernel write buffer (socket buffer). The mmap method can reduce one data copy. 2. Through the sendfile system call (introduced in Linux 2.1), data can be directly transmitted I/O within the kernel space, thus eliminating the round-trip copying of data between user space and kernel space (reducing two context switches and one data copy). After Linux 2.4, through DMA gather technology (requires DMA hardware support), one CPU data copy operation in the kernel space can be further reduced. However, this approach means that the data does not pass through user space, so users cannot process this data, making it less useful than mmap in many cases. 3. Through the splice system call (introduced in Linux 2.6), a pipeline can be established between the read buffer in the kernel space and the network buffer (socket buffer), thereby avoiding CPU copy operations between the two. Splice technology no longer requires dedicated DMA hardware and still only requires two DMA copies and two context switches. However, it also has the problem that users cannot modify the data, and because it uses Linux pipes, one of the two file descriptors used to transmit data must be a pipe device.

To reduce communication overhead, modern systems use zero-copy technology to optimize communication on a single machine. Intra-process communication in ROS2 is a zero-copy implementation achieved by directly sharing data within a process (similar to Cyber's INTRA communication and ERDOS's Inter-thread communication). ROS2 only needs to pass the address of the data to achieve communication. Data copying only occurs when the subscriber needs ownership of the message. For inter-process communication in ROS2, it is outsourced to different DDS middleware,

which will use various technologies to optimize the communication process. These technologies are usually invisible to users. In recent versions, ROS2 has provided a new mechanism called Loaned Message, where users can "borrow" memory from the underlying DDS middleware to avoid unnecessary copies. However, this mechanism is currently immature and requires the DDS middleware to support this feature. Cyber also introduced a similar mechanism in version 10.0. It added the AcquireMessage function in SHMTransmitter, which is to retrieve a block of pre-allocated memory from the Arena to use as a message. The essence of these two new mechanisms is still to minimize the number of message copies.

5.5.2 Node Composition. As previously discussed, ROS2's intra-process communication offers lower overhead and higher performance compared to the DDS communication path, particularly when dealing with large data volumes, where this difference becomes even more pronounced [65]. Consequently, a straightforward strategy for scenarios with significant communication load or stringent latency requirements is to place publishers and subscribers within the same process whenever feasible, enabling their communication to occur via the intra-process path. The Node Composition feature provided by ROS2 accomplishes precisely this. It empowers developers to construct and manage robot software components more flexibly and efficiently. Node composition allows multiple nodes to be loaded into a single process. While each node retains its independent logic and functionality, they can share data and resources much more efficiently.

In practical applications within Autoware, particularly in source data processing scenarios, numerous instances of node composition can be observed. Given the large size and high update frequency of raw data, employing intra-process communication significantly reduces communication overhead in such situations. It's fair to say that intra-process communication and node composition are a standard combination for real-world system deployments.

Apollo also employs a similar approach. Nodes involved in processing raw sensor data are typically placed within the same process and utilize Cyber's INTRA communication to minimize communication overhead. This optimization suggests that some concerns regarding the performance of DDS communication on a single machine might be unwarranted, as scenarios with genuinely high communication loads tend to leverage intra-process communication methods whenever possible.

5.5.3 Shared Memory Communication. As discussed in the preceding sections, the overall communication flow in Cyber and ROS2 is quite similar. The most significant difference is that Cyber has specifically implemented a shared memory communication path. Cyber offers two primary shared memory mechanisms:

(1) POSIX Shared Memory: The memory mapping mechanism ‘mmap‘ is a POSIX standard system call that allows processes to share memory by mapping the same regular file. The underlying principle is that when a file is opened, if the corresponding pages are already present in the page cache, the kernel directly returns that address for reading and writing, enabling multiple processes to operate on the same memory region. Once a regular file is mapped into a process's address space, the process can access it like regular memory, without needing to call ‘read()‘ or ‘write()‘ operations explicitly. POSIX shared memory offers a simple and highly flexible interface but relies on the file system. It is well-suited for scenarios where the size of the shared memory needs to be dynamically adjusted.
(2) XSI Shared Memory: This is a shared memory technology based on the System V IPC (Inter-Process Communication) mechanism. It enables inter-process shared memory communication by mapping files within a special file system called ‘shm‘. Essentially, each shared memory segment corresponds to a file in the ‘shm‘ file system. Data intended for sharing between processes is placed in an IPC shared memory region. All processes that need to access this

shared region must map it into their own address space. System V shared memory uses the ‘shmget‘ system call to obtain or create an IPC shared memory region and returns a corresponding identifier. Subsequently, the ‘shmat‘ system call maps this shared memory region into the process’s address space. Because a file with the same name in the ‘shm‘ file system is created and associated with the shared memory region when ‘shmget()‘ is called, the process of calling ‘shmat()‘ is similar in principle to mapping the identically named file in the ‘shm‘ file system, much like ‘mmap()‘. System V shared memory enjoys broad support across most Unix-like systems. It does not depend on the file system but has a more complex interface and lower flexibility compared to POSIX shared memory. It is suitable for scenarios requiring cross-platform compatibility.

Furthermore, as mentioned earlier, Cyber provides two methods for notifying receivers about the arrival of new messages in the shared memory communication path:

(1) ConditionNotifier: This method employs an additional shared memory segment, referred to as the Indicator, to facilitate notification. When a new message is written to the shared memory, the Indicator is updated with details such as the message’s location within the shared memory and the associated channel. A dedicated thread created by the SHM mechanism continuously polls the Indicator (Cyber configures this polling interval to 50 microseconds). Upon detecting an update, the thread retrieves the corresponding message and proceeds with further processing.
(2) MulticastNotifier: MulticastNotifier is a notification mechanism designed to signal the availability of data in inter-process communication. It is typically implemented using multicast sockets, allowing a single notification to be received by multiple receivers. Once a new message is written to the shared memory, Cyber sends a notification message via a Multicast Socket. A thread created by the SHM mechanism uses the Linux ‘poll‘ system call to listen for these notifications. Upon receiving a notification, the thread proceeds with the subsequent processing steps.

Cyber’s shared memory communication mechanism serves as a valuable reference, and many other communication middleware implementations also utilize the techniques discussed in this section. Positioned as an intermediary between Intra-Process and RTPS communication, shared memory offers a balance of high communication performance without significantly compromising the system’s modular architecture. This has made it a popular choice for on-machine communication methods. Several studies, such as [65], even suggest that shared memory communication might be the optimal solution when considering both performance and isolation.

5.5.4 Specifying Memory Allocators. To guarantee real-time performance, ensuring the determinism of every aspect of the system’s runtime behavior is paramount. Memory allocation, being a potentially time-consuming system operation, can introduce significant unpredictability if not managed carefully. For instance, some containers in the C++ standard library (like ‘std::vector‘) have automatic resizing capabilities, which might trigger substantial memory allocation operations during program execution. Real-time systems strive to avoid dynamic memory allocation during runtime. Taking ROS2 as an example, it provides a memory allocator interface, allowing users to define custom memory allocators for ROS2 objects. Subsequently, when new memory needs to be allocated, the system will bypass the default system allocator and utilize the user-specified one, thereby achieving more predictable behavior.

Regarding how users should design memory allocators suitable for real-time systems, the simplest approach is to implement a memory pool. This involves pre-allocating a large block of memory and then allocating and deallocating smaller chunks from it as required. ROS2 officially recommends the

TLSF (Two Level Segregate Fit) memory allocator. TLSF achieves fast allocation and deallocation by organizing free memory blocks into multiple lists based on their sizes. Furthermore, the TLSF allocator exhibits good temporal determinism, with a maximum of 168 processor instructions executed on x86 architecture CPUs, ensuring a deterministic worst-case execution time.

5.5.5 Pull-based Communication. In the execution logic of a callback function, a subscriber's message reception triggers the callback's execution. This process is inherently uncontrolled, with each execution consuming computational resources and incurring scheduling overhead (thread wake-up, data contention, etc.). For real-time systems, enhancing determinism often involves transforming this reactive, push-based message reception into a proactive, pull-based approach.

ROS2 offers the $Subscription- > take()$ interface to facilitate pull-based communication. The underlying principle is as follows: upon creating a subscription, it is placed into a special "orphan" callback group (details about callback groups can be found in Sec.4). This callback group is not associated with any executor, meaning that upon receiving a message, no thread will automatically execute its callback function. When it becomes necessary to process messages from this topic (typically within the callback function of another regular subscriber), the program can use $Subscription- > take()$ to explicitly retrieve the message. Similar mechanisms exist for both inter-process communication via DDS middleware and ROS2's native intra-process communication, although the specific interfaces differ (intra-process communication uses $take_data()$). Autoware utilizes this interface to implement 'InterProcessPollingSubscriber', which is employed in various functional modules, including planning and control.

Beyond the basic usage described above, users can directly leverage the 'wait_set' interface provided by ROS2 to gain finer control over the message reception, transmission, and execution process. The idea is to take direct control of the polling mechanism. The user program periodically updates and checks the status of the 'wait_set', and then extracts and processes messages as required. Similar to the 'WaitSet', the 'rclc executor' mechanism [55] exists, which is a custom executor designed for micro-ROS. The 'rclc executor' allows for the explicit definition of callback execution order and triggering conditions, thereby enhancing the real-time performance of the system. We believe that utilizing these methods to take over ROS2's task scheduling is essential for building real-time or near-real-time systems on top of ROS2.

Cyber and ERDOS also implement pull-based message retrieval mechanisms. In Cyber, messages can be retrieved directly from message queues by calling read message interfaces within coroutines. In ERDOS, users can override the 'run' method of an operator to gain complete control over its execution. In this scenario, the operator's message processing is no longer dependent on callback functions but is entirely managed by the user actively fetching messages.

6 RESEARCH DIRECTIONS

Based on our understanding of the real-time guarantee issue of AD systems, we summarize the following three open research directions:

- **Abstraction of Autonomous Driving System Models:** As summarized at the end of Sec.3, a significant gap remains between real-time systems theory and practical AD implementations. This gap arises primarily from two factors: the extreme complexity of AD system, and the inherent limitations of real-time models in describing Cyber-Physical Systems (CPS). Researchers in the real-time community must reconsider how to abstract AD systems in a way that balances the difficulty of analysis with fidelity to real-world behavior. Research in the CPS domain may provide useful insights, yet it also needs to confront the challenge of overly complex AD systems.

The various scheduling and communication techniques discussed in this paper largely serve to provide developers with more efficient tools—offering lower overhead, stronger determinism, and greater configurability. However, without theoretical breakthroughs at the modeling level, these techniques can only yield incremental improvements. True progress will depend on establishing new theoretical foundations that bridge the gap between model abstraction and real-world system behavior.

- **Intelligent Use of Application Info:** The various scheduling and communication techniques discussed in this paper largely serve to provide developers with more efficient tools—offering lower overhead, stronger determinism, and greater configurability. However, without theoretical breakthroughs at the modeling level, these techniques can only yield incremental improvements. True progress will depend on establishing new theoretical foundations that bridge the gap between model abstraction and real-world system behavior. Future research should move beyond providing real-time mechanisms to exploring how effectively AD frameworks utilize them. While current systems mitigate issues like priority inversion, few exploit scheduling capabilities to their full potential.
  We foresee growing use of automated configuration and learning-based scheduling, where middleware dynamically infers task priorities and resource mappings from runtime data. Integrating perception and decision semantics into scheduling policies could further enable adaptive timing behavior under varying driving contexts. Bridging theoretical schedulability and practical adaptability remains a key challenge for next-generation AD systems.
- **Full-Stack Deterministic Optimization:** Achieving robust end-to-end guarantees requires deterministic coordination across the entire software–hardware stack. While prior studies focus on individual layers—schedulers, communication, or perception—comprehensive full-stack optimization remains an open challenge.
  Future research should explore cross-layer co-design that jointly considers computation, communication, and memory determinism, supported by predictable dataflow pipelines and bounded-latency networking. Advancing such integration will demand both theoretical innovation and large-scale engineering effort, making full-stack real-time design one of the most promising research frontiers.

7 CONCLUSION

This paper reviewed the evolution of system architectures for autonomous driving, focusing on how scheduling and communication mechanisms shape real-time performance. We analyzed the limitations of existing real-time models in capturing the complexity of modern AD systems, highlighting the mismatch between theoretical abstractions and practical implementations. Through comparative study, we showed that while current middleware and operating systems provide the foundation for predictable execution, their separation often leads to fragmented timing control and inefficient resource coordination.

Looking ahead, we identified several promising directions for future research. Middleware is expected to take a central role in runtime management, while operating systems evolve toward lightweight, deterministic micro-kernels. Achieving end-to-end real-time guarantees will require cross-layer co-design that unifies computation, communication, and control under a CPS-oriented framework. Ultimately, bridging the gap between theory and deployment will be key to realizing autonomous driving platforms that are not only functionally capable but also provably safe, predictable, and verifiable in real-world operation.

REFERENCES

[1] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. 2020. Timing of autonomous driving software: Problem analysis and prospects for future solutions. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 267–280.

[2] Julio CS Dos Anjos, Kassiano J Matteussi, Fernanda C Orlandi, Jorge LV Barbosa, Jorge Sá Silva, Luiz F Bittencourt, and Cláudio FR Geyer. 2023. A survey on collaborative learning for intelligent autonomous systems. *Comput. Surveys* 56, 4 (2023), 1–37.

[3] Apex.AI. 2025. *Apex.AI.* https://www.apex.ai/ [Online].

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

[5] Baidu. 2025. *Apollo.* http://apollo.auto/

[6] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. 2012. A generalized parallel task model for recurrent real-time processes. In *2012 IEEE 33rd Real-Time Systems Symposium*. IEEE, 63–72.

[7] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. 2016. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 159–169.

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

[9] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. 2002. A protocol for loosely time-triggered architectures. In *International Workshop on Embedded Software*. Springer, 252–265.

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

[11] Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn B Brandenburg. 2021. 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, 264–277.

[12] Bjorn B Brandenburg. 2011. *Scheduling and locking in multiprocessor real-time operating systems*. Ph. D. Dissertation. The University of North Carolina at Chapel Hill.

[13] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn Brandenburg. 2019. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl, 1–23.

[14] Hyunjong Choi, Yecheng Xiang, and Hyoseung Kim. 2021. 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, 251–263.

[15] Hoon Sung Chwa, Kang G Shin, and Jinkyu Lee. 2018. Closing the gap between stability and schedulability: A new task model for cyber-physical systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 327–337.

[16] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. 2007. Period optimization for hard real-time distributed automotive systems. In *Proceedings of the 44th annual Design Automation Conference*. 278–283.

[17] Marco Dürr, Georg Von Der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. 2019. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–24.

[18] Daniel Enright, Yecheng Xiang, Hyunjong Choi, and Hyoseung Kim. 2024. PAAM: A Framework for Coordinated and Priority-Driven Accelerator Management in ROS 2. *arXiv preprint arXiv:2404.06452* (2024).

[19] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. 2009. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society.

[20] Shuo Feng, Haowei Sun, Xintao Yan, Haojie Zhu, Zhengxia Zou, Shengyin Shen, and Henry X Liu. 2023. Dense reinforcement learning for safety validation of autonomous vehicles. *Nature* 615, 7953 (2023), 620–627.

[21] Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. 2010. Scheduling dependent periodic tasks without synchronization mechanisms. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 301–310.

[22] Julien Forget, Frédéric Boniol, and Claire Pagetti. 2017. Verifying end-to-end real-time constraints on multi-periodic models. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 1–8.

[23] The Autoware Foundation. 2025. *AUTOWARE.AUTO Website.* https://www.autoware.auto/ [Online].

[24] Cong Gao, Geng Wang, Weisong Shi, Zhongmin Wang, and Yanping Chen. 2021. Autonomous driving security: State of the art and challenges. *IEEE Internet of Things Journal* 9, 10 (2021), 7572–7595.

[25] Alain Girault, Christophe Prévot, Sophie Quinton, Rafik Henia, and Nicolas Sordon. 2018. Improving and estimating the precision of bounds on the worst-case latency of task chains. *IEEE transactions on computer-aided design of integrated circuits and systems* 37, 11 (2018), 2578–2589.

[26] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E Gonzalez, and Ion Stoica. 2022. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 453–471.

[27] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A Wright, Joseph E Gonzalez, and Ion Stoica. 2021. Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 8806–8813.

[28] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. 2017. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[29] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. 2017. Waters industrial challenge 2017. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.

[30] Taeho Han and Kanghee Kim. 2023. Minimizing Probabilistic End-to-end Latencies of Autonomous Driving Systems. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 27–39.

[31] Xu Jiang, Dong Ji, Nan Guan, Ruoxiang Li, Yue Tang, and Yi Wang. 2022. Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 27–39.

[32] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. 2015. An open approach to autonomous vehicles. *IEEE Micro* 35, 6 (2015), 60–68.

[33] Tomasz Kloda, Antoine Bertout, and Yves Sorel. 2018. Latency analysis for data chains of real-time periodic tasks. In *2018 IEEE 23rd international conference on emerging technologies and factory automation (ETFA)*, Vol. 1. IEEE, 360–367.

[34] David Philipp Klüner, Marius Molz, Alexandru Kampmann, Stefan Kowalewski, and Bassam Alrifaee. 2024. Modern Middlewares for Automated Vehicles: A Tutorial. *arXiv preprint arXiv:2412.07817* (2024).

[35] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. 2010. Scheduling parallel real-time tasks on multi-core processors. In *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 259–268.

[36] Ao Li and Ning Zhang. 2024. Data-flow Availability: Achieving Timing Assurance in Autonomous Systems. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 445–463.

[37] Ruoxiang Li, Nan Guan, Xu Jiang, Zhishan Guo, Zheng Dong, and Mingsong Lv. 2022. Worst-case time disparity analysis of message synchronization in ros. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 40–52.

[38] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the twenty-third international conference on architectural support for programming languages and operating systems*. 751–766.

[39] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[40] Liangkai Liu, Zheng Dong, Yanzhi Wang, and Weisong Shi. 2022. Prophet: Realizing a predictable real-time perception pipeline for autonomous vehicles. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 305–317.

[41] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. 2020. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal* 8, 8 (2020), 6469–6486.

[42] S Liu, B Yu, N Guan, Z Dong, and B Akesson. 2021. Real-time scheduling and analysis of an autonomous driving system. *Proc. RTSS Ind. Challenge Problem* (2021), 1–47.

[43] Shaoshan Liu, Bo Yu, Yahui Liu, Kunai Zhang, Yisong Qiao, Thomas Yuang Li, Jie Tang, and Yuhao Zhu. 2021. The Matter of Time–A General and Efficient System for Precise Sensor Synchronization in Robotic Computing. *arXiv preprint arXiv:2103.16045* (2021).

[44] Tianen Liu, Shuai Wang, Zheng Dong, Borui Li, and Tian He. 2025. From Perception to Computation: Revisiting Delay Optimization for Connected Autonomous Vehicles. *Comput. Surveys* (2025).

[45] Sidi Lu and Weisong Shi. 2021. The emergence of vehicle computing. *IEEE Internet Computing* 25, 3 (2021), 18–22.

[46] Yehan Ma, Ruijie Fu, An Zou, Jing Li, Cailian Chen, Chenyang Lu, and Xinping Guan. 2024. Performance Optimization and Stability Guarantees for Multi-tier Real-Time Control Systems. In *2024 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 187–200.

[47] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. 2020. How do you architect your robots? State of the practice and guidelines for ROS-based systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 31–40.

[48] Aloysius Ka-Lau Mok. 1983. *Fundamental design problems of distributed systems for the hard-real-time environment*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[49] John K Ousterhout et al. 1982. Scheduling Techniques for Concurrent Systems.. In *ICDCS*, Vol. 82. 22–30.

[50] Oliver Scheickl, Christoph Ainhauser, and Peter Gliwa. 2012. Tool support for seamless system development based on AUTOSAR timing extensions. In *Embedded Real Time Software and Systems (ERTS2012)*.

[51] Johannes Schlatow and Rolf Ernst. 2016. Response-time analysis for task chains in communicating threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–10.

[52] The seL4 Foundation. 2025. *seL4 Microkernel*. https://sel4.systems/ [Online].

[53] Helder Silva, José Sousa, Daniel Freitas, Sergio Faustino, Alexandre Constantino, and Manuel Coutinho. 2009. RTEMS improvement–space qualification of RTEMS executive. *1st Simpósio de Informática-INFORUM, University of Lisbon* (2009).

[54] Hoora Sobhani, Hyunjong Choi, and Hyoseung Kim. 2023. Timing Analysis and Priority-driven Enhancements of ROS 2 Multi-threaded Executors. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 106–118.

[55] Jan Staschulat, Ralph Lange, and Dakshina Narahari Dasari. 2021. Budget-based real-time executor for micro-ros. *arXiv preprint arXiv:2105.05590* (2021).

[56] Jinghao Sun, Kailu Duan, Xisheng Li, Nan Guan, Zhishan Guo, Qingxu Deng, and Guozhen Tan. 2023. Real-Time Scheduling of Autonomous Driving System with Guaranteed Timing Correctness. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 185–197.

[57] Jinghao Sun, Xisheng Li, Mingyang Gong, Nan Guan, Zhishan Guo, Mingsong Chen, Jun Zhao, and Qingxu Deng. 2025. Jointly Ensuring Timing Disparity and End-to-End Latency Constraints in Hybrid DAGs. In *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 190–201.

[58] Chen Tang, Tuo Shi, Qian Xu, Kui Wu, Nan Guan, Jen-Ming Wu, and Jianping Wang. 2025. Designing and Implementing AoI-Optimized Scheduling for Autonomous Driving Systems. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*. IEEE, 1–10.

[59] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. 2020. Response time analysis and priority assignment of processing chains on ros2 executors. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 231–243.

[60] Yue Tang, Nan Guan, Xu Jiang, Xiantong Luo, and Wang Yi. 2023. Real-Time Performance Analysis of Processing Systems on ROS 2 Executors. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 80–92.

[61] Harun Teper, Tobias Betz, Mario Günzel, Dominic Ebner, Georg Von Der Brüggen, Johannes Betz, and Jian-Jia Chen. 2024. End-to-end timing analysis and optimization of multi-executor ROS 2 systems. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 212–224.

[62] Micaela Verucchi, Mirco Theile, Marco Caccamo, and Marko Bertogna. 2020. Latency-aware generation of single-rate DAGs from multi-rate task sets. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 226–238.

[63] Nils Vreman, Anton Cervin, and Martina Maggio. 2021. Stability and performance analysis of control systems subject to bursts of deadline misses. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[64] Tianze Wu and Weisong Shi. 2024. Timeliness in Autonomous Driving: Hype or Reality? *IEEE Internet Computing* 28, 5 (2024), 75–84.

[65] Tianze Wu, Baofu Wu, Sa Wang, Liangkai Liu, Shaoshan Liu, Yungang Bao, and Weisong Shi. 2021. Oops! it's too late. your autonomous driving system needs a faster middleware. *IEEE Robotics and Automation Letters* 6, 4 (2021), 7301–7308.

[66] Chengyuan Xu, Qian Xu, Jianping Wang, Kui Wu, Kejie Lu, and Chunming Qiao. 2022. AoI-centric task scheduling for autonomous driving systems. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 1019–1028.

[67] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. 2020. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1067–1081.