2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

AROSpect: A ROS 2 Timing Introspection Framework

Lukas Dust[1 2], Christopher Timperley[3] and Rong Gu[2]

Abstract— **This paper introduces AROSpect, a framework for timing introspection and controlled experimentation for ROS 2-based applications. AROSpect enables developers to model system components using standardized templates, inject synthetic delays, and measure end-to-end latencies across message paths. Through instrumentation, the framework supports iterative refinement of timing parameters and identification of misconfigurations. A case study using a multi-agent turtlesim system demonstrates how AROSpect can guide developers to understand the effects of adapting timing parameters, contributing toward more predictable robotic systems.**

I. Introduction

Modern robotic systems range from autonomous vehicles to industrial and collaborative robots, and they are increasingly distributed, complex, and time-critical. To manage this distributed complexity and time criticality at scale, many organizations adopt ROS 2 [1], which provides the communication plumbing, developer tooling, and rich ecosystem of reusable components and capabilities needed to build distributed robotic systems.

Correctness in these systems is defined not only by functional behavior but also by timing correctness, since computations and communications must meet deadlines to avoid degraded performance, unsafe behavior, or mission failure (e.g., missed collision avoidance or unstable control loops). A key challenge in designing such systems is achieving real-time capability: the ability to guarantee that all activated components meet their timing constraints predictably [2]. This is particularly critical in safety-critical domains, where timing violations can have catastrophic consequences. For instance, automotive safety mechanisms must deploy within an extremely restrictive time limit, e.g., airbags must be activated within 15–30 milliseconds and fully inflated within 60–80 milliseconds [3]. Missing these deadlines constitutes a timing violation that compromises system reliability.

While ROS 2 helps to facilitate prototyping and systems integration, ensuring timing correctness in ROS 2-based systems remains a significant challenge. Already at design time, developers lack tools to systematically experiment with and analyze timing behavior in a controlled and reproducible way. This reduces developers' ability to make informed design decisions, identify misconfigurations, and improve performance. Despite its importance, accurately estimating worst-case execution times (WCETs) and end-to-end latencies in ROS 2-based systems remains difficult [2]. Timing behavior is influenced by a multitude of interacting factors, including middleware communication (e.g., Data Distribution Service(DDS)), node scheduling, hardware variability, and runtime conditions [4]–[7]. Existing tools either provide offline analysis, which often yields overly pessimistic bounds, or runtime monitoring, which detects failures only after they occur [8], [9]. This leaves developers with limited insight into where timing bottlenecks, overruns, or error propagation originate. As a result, frameworks, such as the widely used Autoware framework [10], adopt a “fail-safe on timing basis” strategy, acknowledging that timing constraints may be violated as long as overruns are handled gracefully [2]. However, existing methods and tooling lack mechanisms for systematic timing experimentation and analysis prior to deployment, making timing behavior difficult to understand and improve.

In this paper, we present AROSpect, a ROS 2-based experimentation framework for timing analysis of robotic systems. AROSpect models the execution of the original ROS 2 system in a controlled environment, enabling comparative analysis with different sets of timing configurations, focusing on end-to-end latencies across message paths. AROSpect enables developers to observe system-wide effects of timing injection, where developers can introduce controlled delays, as well as the adaptation of execution rates.

Overall, AROSpect aims to contribute towards controlled experimentation and analysis of timing parameters for robotics developers, giving a better base for decisions when building and optimizing ROS 2 systems.

The main contributions of this paper are as follows:

- A systematic framework to explore and analyze timing behavior in ROS 2 systems.
- Executable ROS 2 model templates allowing developers to model real-world ROS 2 systems from a timing perspective and manipulate such models for timing analysis in a controlled environment.
- Inbuilt monitoring options, where execution times, latencies, and delay propagation are measured across nodes and topics.
- A demonstration of AROSpect on a multi-agent turtlesim example [11], where guided manipulation of timing parameters reveals delay propagation and informs design choices toward more predictable and perfomant ROS 2 systems.

*This work was supported by the Swedish Knowledge Foundation through the research profile DPAC – Dependable Platforms for Autonomous Systems and Control (Grant No. 20150022), as well as the Ericsson Research Foundation. The AI system ChatGPT developed by OpenAI was used to assist in drafting and structuring the source code implementation. All generated content was reviewed, manually analyzed, and revised by the authors.

[1]Scandinavian AI Laboratory - SCAILAB AB. lukas@scailab.se
[2]School of Innovation, Design, and Engineering (IDT), Mälardalen University. {lukas.dust, rong.gu}@mdu.se
[3]School of Computer Science, Carnegie Mellon University. ctimperl@andrew.cmu.edu

II. Background and Related Work

A. ROS 2

ROS 2 [1] has become the de facto middleware for robotics research and development, providing modularity, extensibility, and wide adoption across academic and industrial domains. Contrary to what its name might suggest, ROS 2 is not an operating system in itself. It runs on top of a conventional OS, most commonly Linux, leveraging its services and infrastructure. ROS 2's modular nature enables developers to develop and integrate complex sensing, planning, and actuation pipelines. However, while ROS 2 provides flexibility, it was not originally designed with hard real-time guarantees in mind. Nevertheless, towards real-time capabilities, ROS 2 supports among other communication middlewares, the Data Distribution Service (DDS). However, the lack of native mechanisms for bounding worst-case execution times (WCETs) or guaranteeing end-to-end timing across distributed nodes makes it difficult to ensure predictable behavior in safety-critical systems such as autonomous driving and industrial robotics. Recent research has highlighted that the non-deterministic scheduling of executors, variations in message latencies, and complex causal links across processing chains pose significant challenges for real-time robotics [4], [5], [9].

ROS 2 systems are composed of modular units known as nodes, which execute within the host operating system. Nodes interact through defined communication channels. A node consists of functions, called callbacks, the smallest schedulable units in ROS 2. Callbacks are triggered by specific events, such as incoming messages or timer-based triggers. Timer-based triggers utilize system timers to initiate functions at predefined intervals, enabling time-sensitive operations within the robotic system. Figure 1 illustrates a simple ROS 2 setup involving two nodes and one communication channel.

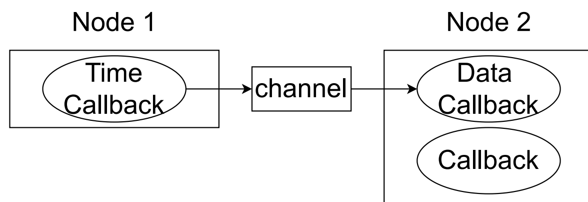

Fig. 1. An example ROS 2 system showing two nodes, callbacks, and one communication channel.

B. Timing Analysis and Modeling of ROS 2 Systems

Timing analysis has emerged for understanding and predicting system behavior in ROS 2-based applications. In particular, end-to-end latency analysis focuses on the delay experienced by data as it flows through a computation chain consisting of sensors, processing nodes, and actuators. Several works have analyzed latency behavior in ROS 2. Teper et al. [6] proposed a systematic framework for end-to-end timing analysis of ROS 2 systems, modeling computation chains across sensors, filters, fusion, and actuators. This work formalized the decomposition of ROS 2 applications into timing-critical components. In our work, we utilize the abstractions of the sensor, filter, fusion, and actuators to model a ROS 2 system.

- Sensor: A sensor is a ROS 2 node, which periodically outputs a message to one or more designated channels. Ideally, a sensor reading.
- Filter: A filter node receives data through one channel and forwards the information to another channel.
- Fusion: A fusion node receives data through multiple channels and fuses the information into one output channel.
- Actuator: An actuator node receives data through one channel and acts as a sink to a processing chain.

Other works have examined priority assignment and scheduling policies [12], [13], addressing how ROS 2 executors can be configured to reduce jitter and deadline misses. Response-time analyses exploiting starvation freedom and execution-time variance were introduced in [5], enabling tighter bounds for systems under non-preemptive scheduling. Latency and jitter measurement frameworks have been proposed for autonomous driving stacks [14], highlighting the difficulty of bounding delays in DAG-shaped pipelines. Additional contributions address multi-threaded executor behavior [15], message synchronization disparities [16], and causal message flow analysis [17].

C. Positioning of AROSpect

Collectively, related works have provided significant insights into the challenges of real-time ROS 2. However, most approaches either (i) rely on theoretical analysis under restrictive assumptions, or (ii) provide measurement frameworks without interactive experimentation.

AROSpect aligns with established timing analysis techniques while extending them into an experimental platform to act as an experimental playground to study timing behavior under controlled perturbations. Hereby, AROSpect builds upon prior work in end-to-end timing analysis [6] by extending the modeling approach that decomposes ROS 2 systems into standard components (sensor, filter, fusion, actuator) while enabling injection of synthetic delays and load.

III. Motivating Example

To illustrate the importance of timing analysis in robotic systems, we consider a representative architecture from autonomous driving: an Autoware-based self-driving stack [10]. This system comprises multiple interconnected components, each implemented as a ROS 2 node, forming a directed acyclic graph (DAG) of data flow and control.

Autonomous driving pipelines typically include the following stages:

- Perception: Sensor data (e.g., camera, LiDAR) is processed for object detection and tracking.
- Localization: Sensor fusion (SLAM, GNSS, IMU) estimates the vehicle's position.
- Planning: Path planning and obstacle avoidance generate safe trajectories.

- Control: Commands are issued to actuators based on planned trajectories.

Each stage operates under strict timing constraints. For example, perception must deliver updates within tens of milliseconds to ensure timely reactions, while control loops must execute at high frequency to maintain stability. An extract of such a pipeline from the Autoware system is shown in Figure 2.

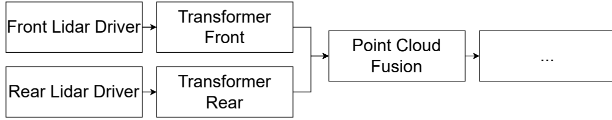

Fig. 2. An extract showing components of the Autoware lidar pipeline.

Timing violations can lead to critical failures in such systems:

- Perception Delay: A LiDAR point cloud is delayed due to processing overload. The planner acts on outdated obstacle data, potentially failing to avoid a pedestrian.
- End-to-End Latency Accumulation: Individual components meet their deadlines, but cumulative latency exceeds the system's 100 ms reaction budget. At highway speeds, this results in delayed steering corrections and unsafe drift.
- Execution Jitter: Localization updates arrive irregularly, causing the planner to generate unstable paths.
- Hardware-Induced Variability: Porting the system from a high-performance GPU to an embedded platform doubles execution time, violating timing assumptions.

These failure modes are difficult to detect and analyze due to the complexity of the DAG, non-deterministic scheduling, and limited tooling for end-to-end timing experimentation. For demonstration purposes and sake of simplicity, in the remainder of this paper, we utilize a simplified version of a multi-agent system with general path planning and collision avoidance mechanisms for experimentation.

IV. AROSPECT DESIGN

AROSpect is a modular framework for timing introspection in ROS 2 systems.[1] Its design provides a structured way to model message-passing architectures, inject synthetic delays, and perform reproducible timing experiments. The framework enables researchers and developers to observe how timing variations propagate through a system and to identify potential sources of latency violations or jitter.

A. Components and Logic

AROSpect is organized around four node templates. Namely, sensors, filters, fusions, and actuators. The templates are linked by standardized message abstractions and supported by an offline analysis pipeline.

[1]Published online: https://github.com/ljdust/AROSpect/tree/ICRA

Node Templates: AROSpect defines reusable node templates that represent common architectural roles in robotic software.

Sensor Nodes: ROS 2 nodes that are time-triggered sources at the beginning of a processing chain. Each sensor publishes data at a configurable rate and with an optional execution delay. Sensors provide reproducible entry points for timing analysis.

Filter Nodes: ROS 2 nodes that process incoming messages before republishing them. Filters may subscribe to one or more topics, apply configurable execution delays, and chain multiple processing paths.

Fusion Nodes: ROS 2 nodes that merge multiple streams. Fusion can be performed through subscription callbacks or periodic timer callbacks, modeling common synchronization patterns in robotic systems.

Actuator Nodes: ROS 2 nodes that are terminal sinks of the computation graph. Actuators timestamp and log received messages into structured CSV files, providing ground truth for analysis.

Generally, each node consists of callbacks executing the following steps:

1) Take timestamp
2) Append timestamp to message
3) Perform execution delay
4) Send message

Message Abstraction: Instead of sending the messages used in the real system, all communication in the AROSpect system model uses a standardized message type. Each record contains:

1) Timestamps (time of message creation and handling of the message in preceding callbacks).
2) Identifiers (denoting the nodes the message has passed through by their original ID).
3) A counter (tracking how many processing steps have occurred).

This abstraction makes it possible to trace messages across arbitrary system paths while preserving their timing lineage. By carrying a full history of processing events, messages serve as self-contained timing records that can later be correlated in analysis.

Paths and Execution Semantics: AROSpect explicitly models paths from sensors to actuators. Along each path, messages may pass through filters or fusions, each of which can introduce controlled variability in delay or scheduling. By parameterizing delays at each stage, AROSpect allows "what-if" experimentation (e.g., slowing down a sensor driver or increasing a fusion workload) while retaining reproducibility.

Analysis Framework: The final component of AROSpect is the analysis and visualization. The analysis script aligns actuator events with their upstream sensor contributions, reconstructing end-to-end latencies for each processing path. Subplots are generated per sensor, showing the time spent in each segment of the path, the aggregate end-to-end duration, and the message instance of each sensor contributing to an actuator output.

The alignment of graphs allows direct comparison of alternative processing paths (e.g., a sensor message routed through one filter versus two). As a result, users can quantify both the critical paths and the variance in execution times.

B. Implementation

The framework is implemented in ROS 2 Jazzy, leveraging its parameterization and launch capabilities. Each node template is implemented in C++ for performance, while analysis and visualization are handled in Python. The repository provides:

- YAML configuration files for modeling ROS 2 systems and setting node parameters (topic names, execution delays, IDs, etc.),
- Launch scripts for reproducible experiments, including reference setups such as a simplified Autoware pipeline,
- Logging infrastructure that stores per-node timing data in CSV format for offline analysis.

The modularity of the templates allows rapid prototyping. A new experiment can be constructed simply by editing configuration files and relaunching the system, without modifying the source code of the nodes.

C. Workflow using AROSpect

Users start with a real system design. As a first step, users need to determine essential timing parameters such as execution times and publishing rates. Such parameters can be determined using tools such as *ROS2_tracing* [18]. Now, users start modeling the system components using the different node templates. After modeling the first version of the system, they can run the model and analyze the results. Based on the output of the first iterations of running the mirrored system in AROSpect, the users can decide what parameters to change. If there are missed instances in some message flows, they can adapt the publishing rate or the delays in specific components and analyze the effect compared to the initial version. In this way, iteratively, better fit parameters can be determined.

After adapting the mirrored system, the users can take the parameters to the real system's implementation. It is important to note that the AROSpect model is an abstraction focusing on simplifying experimentation. Hence, there might be a deviation between the obtained changes in the model vs. the real system. Hence, monitoring of changes to the real system is required.

D. Example

Figure 3 shows an example system with three different paths leading to an actuator. The information sent by *Sensor1* can either travel through *Filter1*, *Fusion1*, *Fusion3*, to the actuator (Path 1.0, marked with blue arrows) or travels through *Filter2*, *Fusion2*, *Fusion3*, to the actuator (Path 1.1, marked with purple arrows). The information sent by *Sensor2* travels through *Filter2*, *Fusion2*, *Fusion3* (Path 2, marked with red arrows), to the actuator. Due to the fusion of information in the Fusion node, each actuator output is based on two sensor readings. Each instance of the actuator output is either based on one individual reading from *Sensor1* and *Sensor2*, or two instances of *Sensor1*.

An example for a graph generated by the AROSpect analysis can be found in Figure 4. Three bar graphs are positioned above each other, each graph representing one path. Each graph consists of bars showing the end-to-end delays of each actuator instance. The bar graphs display the components' contributions to the delay in colors. Above each bar, the instance of the Sensor reading is displayed. It can be seen that the first instance of the actuator execution has been based on two instances of information from *Sensor1* (Path1.0 and Path1.1). In contrast, the second and third actuator executions have been based on one instance of *Sensor1* (Path1.0) and *Sensor2* (Path2). It can also be seen that all three execution instances are based on the same sensor reading of *Sensor1*. This also explains the rise in delay over the three instances.

V. DEMONSTRATION

In this section, we show how to detect different misconfigurations with static delays and potential mismatches in publishing rates. Secondly, we demonstrate how AROSpect can be utilized to improve timing parameters iteratively. Therefore, we present a ROS 2 system that is used to demonstrate the application of AROSpect and how the framework can be used to improve timing configurations in ROS 2 systems[2]. As a base for the experiment systems, we use ROS 2 turtlesim [11]. For demonstration purposes, we modify the ROS 2 turtlesim to simulate a multi-agent system, showing how AROSpect can be used to identify real-world problems in components such as collision avoidance, and path planning.

Therefore, we run different configurations in turtlesim, demonstrating potential problems in such systems. Then, we utilize AROSpect to modify parameters to improve end-to-end latency.

The demonstration aims to answer the following research questions:

1) How can AROSpect help to identify different types of misconfigurations leading to potential timing errors?
2) How can AROSpect help to understand timing parameters and assist in finding better-suited timing parameters to improve timing performance?

A. Methodology

To answer the research question, we apply system implementation with experimental implementation. We demonstrate the application of AROSpect on a controlled ROS 2 system, illustrating its use in identifying potential misconfigurations. ROS 2 turtlesim will act as a simplified, controllable version of a multi-agent robotic system with different potential mismatches. Firstly, a misconfiguration in update rates, where specific sensors publish information at a slower rate than the subsequent computing nodes can

[2]The source-code for the experiment system and and AROSpect setup are published online: https://github.com/ljdust/AROSpect/tree/ICRA

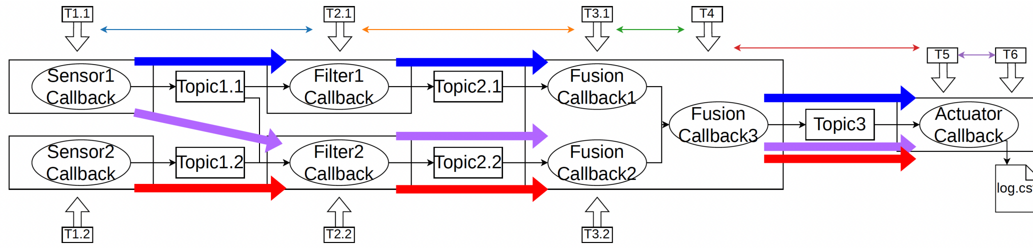

Fig. 3. Example of a system modeled with AROSpect.

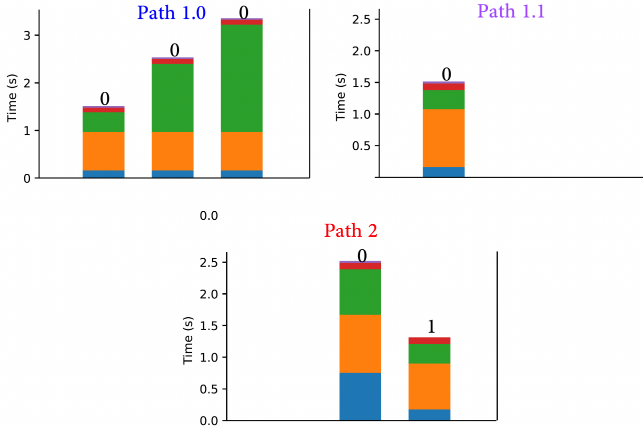

Fig. 4. Example of an analysis output. The total bar shows the end-to-end latency, with the colored areas showing which components of the system, highlighted in Figure 3, contributed to the end-to-end latency. E.g. blue is the time from a sensor callback to a filter callback.

process it. A second type of timing issue is delays that are introduced in specific components. We will utilize AROSpect over two paths of the same system and demonstrate how to use AROSpect to improve the timing parameters.

B. A Case Study System

The system designed for a case study is a simple version of a multi-agent system with general path planning and collision avoidance mechanisms. Turtlesim allows the user to spawn multiple agents and control them via inputs. Each agent periodically publishes its position and rotation to a *position* topic. The agents are controlled via velocity commands. We modify the turtlesim by adding components that control the agents and compose a multi-agent robotic system. An overview of the added components is shown in Figure 5. Generally, there are two crucial paths of information. The top path manages the goal and trajectory planning, while the bottom path manages collision avoidance. To simulate the stack of a multi-agent system, we implement the following nodes (ellipses in the figure):

1. Spawning (not shown in the figure, as it only executes once): The spawner receives the number of turtles as a parameter input and spawns the turtles on fixed locations.

2. Target Generator: The target generator periodically checks if the turtles have reached their defined goal position. When they reach their goal, the target generator updates the goal with a new location.

3. Path planner: Calculates periodically the vector from the position and orientation of the turtles to the target position and outputs the vector for each turtle individually in a topic. When the deviation in rotation is large, the path planner first outputs a rotation only. When the rotation error is smaller, a movement is added. For the sake of simplicity and demonstration, rotation errors are corrected by rotating the turtle in one direction only. When a turtle is more than 45 degrees from the target rotation, the turtle rotates towards the target orientation. When the angle is smaller, a forward movement is initiated.

4. Collision avoidance: The collision avoidance node contains a simple mechanism to avoid collisions by scaling the velocity of the turtles. It subscribes to the position topic of each turtle and checks periodically if turtles are close to each other. If the distance is under a specific threshold, the avoidance mechanism scales down the speed of the turtles that are too close to each other. The velocity scaling factors are published to designated topics.

5. Controller: The controller subscribes to the collision avoidance speed scale and the desired velocity topic from the path planner and outputs the control commands to each of the turtles periodically.

As shown in Figure 5, the nodes exist on two paths. The first path contains the target generator and the path planner, while the second path contains the collision monitor, before the information gets fused in the controller. We assume the update rate for the pose information of each turtle to be fixed at 10 ms. In the following, we create a scenario containing three turtles that are moving on intersecting paths. The paths are shown in Figure 6. In the first step, we set the parameters for update rates such that the turtles follow their goal and do not collide. Each update rates are set to 1 ms, and no delay is introduced to the components. In the next step, we introduce a 20 ms delay to each component to show the effects of timing errors. The turtles start colliding and overshooting their target rotation. This causes the turtle to spin in circles when attempting to rotate into the correct goal pose.

C. Modeling and Setup in AROSpect

For the sake of simplicity, in this demonstration, we focus on the analysis of the path from the sensor readings to the command velocity output of the controller for Turtle1 only. We model the different components of the system using AROSpect node templates. An overview of the modeled system can be found in Figure 7. In the first path (path

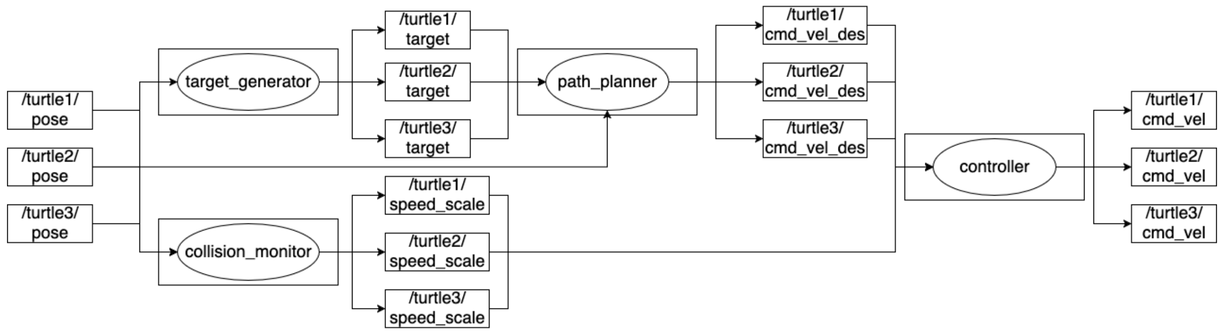

Fig. 5. Nodes modeling a multi-agent ROS 2 system interfacing with ROS 2 turtlesim.

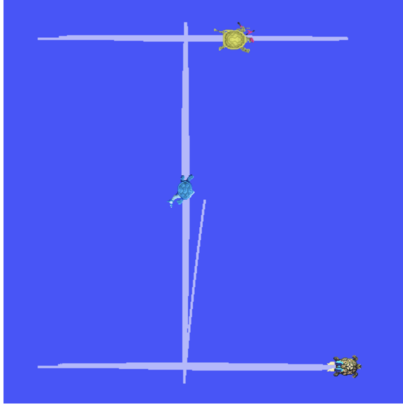

Fig. 6. Turtles simulating robots moving on their respective path.

planning), only the pose information from turtle1 is relevant for the command velocity output. Hence, we model the path containing the pose of turtle1 only. In the second path, (collision avoidance), the poses of all three turtles are fused to determine the speed scale. Hence, all pose inputs are modeled.

Given the modeled system in Figure 7, the following paths and subpaths can be obtained. The format of path indices is: *path.subpath*, e.g., *1.0* means subpath 0 of path 1.

1.0 *Sensor1*, target generator, path planner, controller
1.1 *Sensor1*, path planner, controller
2.0 *Sensor1*, collision monitor, controller
2.1 *Sensor2*, collision monitor, controller
2.2 *Sensor3*, collision monitor, controller

To allow the comparison of different timing configurations, we run the implemented model in AROSpect using the following timing configurations in five scenarios (SC):

SC1 Sensors 10 ms period, all other nodes 1 ms period, 0 ms artificial delay
SC2 Controller 1 ms period, all other nodes 10 ms period, 0 ms artificial delay
SC3 all nodes 10 ms period, 0 ms artificial delay
SC4 Controller 20 ms period, all other nodes 10 ms period, 0 ms artificial delay
SC5 All nodes 10 ms period, 20 ms artificial delay in controller

VI. Results

Figures 8 and 9 contain the results from running all five scenarios for *Path 1* (Purple, Blue) and *Path 2* (Red, Green, Pink) with each color respective a possible subpath. Each block in each bar is associated with a component in the path shown in Figure 7. In SC1 and SC2, it can be seen that multiple instances of the output are based on the same instances of sensor readings. That leads to a higher end-to-end latency. In SC3, the publishing rates are matching. Hence, each actuation output is based on an individual instance of sensor reading. One exception in SC3 in Path 2.0, where the sensor reading 21 is utilized twice. In SC4, the latency is generally low, but instances of sensor readings are missed.

Figure 8 shows a generated trace for the end-to-end analysis of the command velocity of the turtle 1, with the input information traveling through the collision avoidance path with the three different subpaths. Each graph shows the result of all five scenarios for each subpath. Based on the graphs and analysis from AROSpect, the user can conduct two types of improvements:

1) Publishing periods: mismatches in update rates are detected through many iterations of outputs based on the same instances of input information or large gaps in the input instances. Additionally, low frequencies in the sensors lead to higher jitters in a few cases, leading to longer end-to-end delays.
2) Delays: static delays can be identified as constant blocks with lower jitter. As seen in SC5, the extreme delay leads to other delay effects in further system components. The static delay leads to a higher end-to-end latency in addition to missed sensor values.

A. Answer to the RQs

RQ1: Through controlled experimentation, modular architecture, and the provision of templates, AROSpect can help the user to conduct timing experiments with different sets of parameters. The analysis helps to identify mismatches in publishing rates and the effects of delays.

RQ2: Through an iterative process and comparisons, the user can test different sets of parameters and identify areas for improvement in the real system.

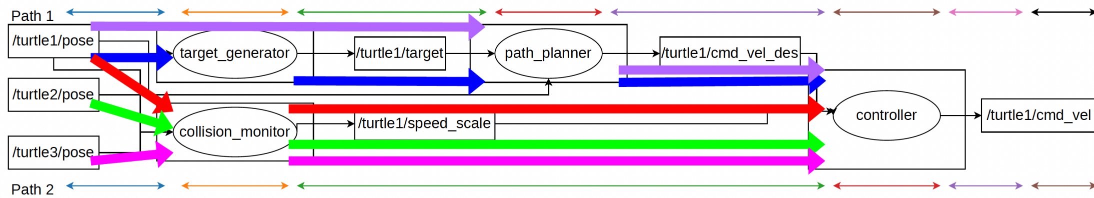

Fig. 7. AROSpect Model of the control loop of turtle1. The colored arrows highlight the different measured latencies.

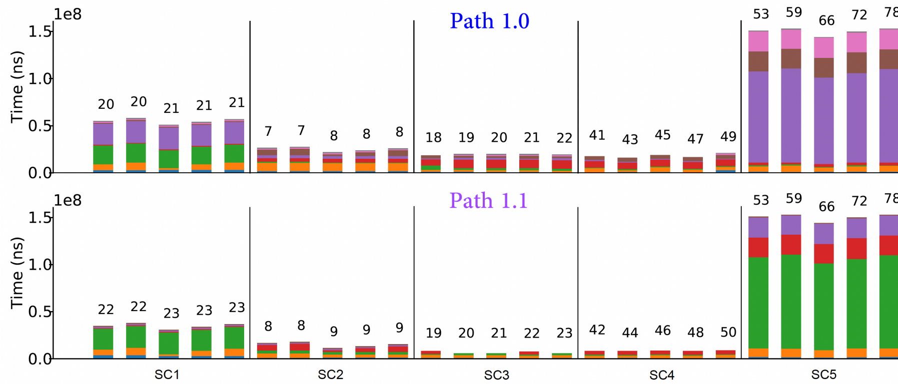

Fig. 8. Results for End-to-End latency for path 1 over the five different scenarios.

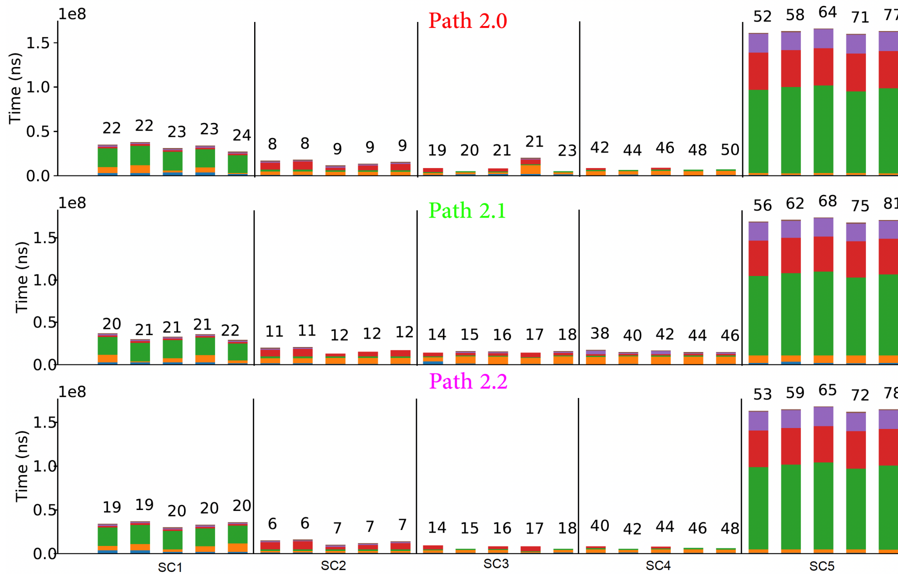

Fig. 9. Results for End-to-End latency for path 2 over the five different scenarios.

VII. Discussion & Conclusion

In this paper, we present AROSpect. A framework that enables modeling of a ROS 2 system from a timing perspective, facilitating controlled experimentation, analysis, and optimization of timing parameters. In a first step, we present how the model presented in [6] can be used and extended to model a ROS 2 system. We design ROS 2 node templates that can be initialized through configuration to model ROS 2 systems. The components include embedded analysis functionality to inject controlled delays and manipulate update rates of callbacks. The user can gain an understanding of timing parameters through controlled experimentation, focusing on extracted processing chains. While real behavior might not be observable, it can be used to work with timing parameters to determine optimization areas and understand the effects of specific parameters on the timing constraints. The analysis allows tracing outputs back to the sensor instances and the information on which a decision is based. On a high level, the framework can be used by robotics systems developers to gain a better understanding of their systems from a timing behavior perspective. The system can also be used to experiment with different timing

parameters to see the effect on end-to-end delays in different paths. While the results abstract some components, such as jitter, the model still gives an approximation of what parameters of a system potentially can be adapted to improve timing behavior. Hence, AROSpect can help the developer to make strategic decisions on optimizations rather than exact parameters for the real system. We demonstrate this through experimentation on a real-world system that represents a ROS 2 multi-agent system. While the demonstration and implementation focus on static parameters, such as delays and jitters introduced through mismatches in update rates, there are open possibilities to extend the framework to include parameter fuzzying, delay bounds, conditional paths and conditional delays, such as non-static jitters in callbacks. The AROSpect model of a system is a simplified approach to modeling ROS 2 systems from an execution perspective. Hence, transferring parameters back to a real system might lead to differences in execution from the observed ones in the mirrored system. Future work can be devoted to evaluation on accuracy and making the models more accurately represent the real system. Furthermore, formal methods can be applied, such as model checking, to obtain potential paths that have not been covered in the model. Static analysis can help identify conditional paths and automatically determine system compositions. Model-based engineering methods can be applied to automate the transformation of real code into an AROSpect model. Additionally, future work can focus on extending the framework to help monitor ROS 2 applications during runtime without needing to model the system individually.

REFERENCES

[1] OpenRobotics, "Ros 2: Documentation," `https://docs.ros.org/en/jazzy`, Accessed: 05-03-2026.

[2] *Autoware concepts - Autoware Documentation — autowarefoundation.github.io*, `https://autowarefoundation.github.io/autoware-documentation/release-v1.0_beta/design/autoware-concepts/`, [Accessed 14-09-2025].

[3] "A test procedure for airbags," *CITA International Motor Vehicle Inspection Committee*, 2016. [Online]. Available: `https://citainsp.org/wp-content/uploads/2016/01/ECS-RSP-Study-2-TP-airbags.pdf`.

[4] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg, "Response-time analysis of ros 2 processing chains under reservation-based scheduling," in *31st Euromicro Conference on Real-Time Systems*, 2019, pp. 1–23.

[5] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ros 2 response-time analysis exploiting starvation freedom and execution-time variance," in *IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2021.

[6] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-to-end timing analysis in ros2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 53–65.

[7] L. Dust, E. Persson, M. Ekström, S. Mubeen, C. Seceleanu, and R. Gu, "Experimental evaluation of callback behavior in ros 2 executors," in *28th International Conf. on Emerging Technologies and Factory Automation*, 2023.

[8] S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "A formal toolchain for offline and run-time verification of robotic systems," *Robotics and Autonomous Systems*, vol. 159, p. 104 301, Jan. 2023.

[9] H. Teper, T. Betz, M. Gunzel, *et al.*, " 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)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 212–224.

[10] Autoware Foundation, "Autoware: Open-source software stack for autonomous driving," `https://github.com/autowarefoundation/autoware`, `https://github.com/autowarefoundation/autoware`, Accessed: 2025-05-26.

[11] *Introducing turtlesim and rqt; ROS 2 Documentation: Jazzy documentation — docs.ros.org*, `https://docs.ros.org/en/jazzy/Tutorials/Turtlesim/Introducing-Turtlesim.html`, [Accessed 05-03-2026], 2026.

[12] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," in *IEEE 27th Real-Time and Embedded Technology and Applications Symposium*, IEEE, 2021, pp. 251–263.

[13] Y. Tang, Z. Feng, N. Guan, *et al.*, "Response time analysis and priority assignment of processing chains on ros2 executors," in *IEEE Real-Time Systems Symposium*, 2020, pp. 231–243.

[14] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, and G. Fettweis, "Latency analysis of ros2 multi-node systems," in *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, 2021.

[15] X. Jiang, D. Ji, N. Guan, R. Li, Y. Tang, and Y. Wang, "Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 27–39.

[16] R. Li, N. Guan, X. Jiang, Z. Guo, Z. Dong, and M. Lv, "Worst-case time disparity analysis of message synchronization in ros," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 40–52.

[17] C. Bédard, P.-Y. Lajoie, G. Beltrame, and M. Dagenais, "Message flow analysis with complex causal links for distributed ros 2 systems," *Robotics and Autonomous Systems*, vol. 161, p. 104 361, 2023.

[18] C. Bédard, I. Lütkebohle, and M. Dagenais, "Ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ros 2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.