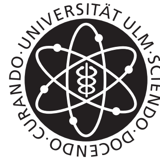


universität ulm

Faculty of Engineering, Computer Science and Psychology
Institute of Measurement, Control and Microtechnology

Enabling Reproducibility in ROS 2 by Ensuring Sequence Deterministic Callback Execution

Master's Thesis

by

Jonas Otto, B. Sc.

05.07.2023

Supervisor:	M. Sc. Matti Henning, M. Sc. Jan Strohbeck
Examiner:	Prof. Dr.-Ing. Klaus Dietmayer
Co-examiner:	Prof. Dr.-Ing. Christian Waldschmidt

I hereby declare that this thesis titled:

Enabling Reproducibility in ROS 2 by Ensuring Sequence Deterministic Callback Execution

is the product of my own independent work and that I have used no other sources and materials than those specified. The passages taken from other works, either verbatim or paraphrased in the spirit of the original quote, are identified in each individual case by indicating the source.
I further declare that all my academic work has been written in line with the principles of proper academic research according to the official "Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis" (University Statute for the Safeguarding of Proper Academic Practice).

Ulm, 05.07.2023

Jonas Otto

Contents

1 Introduction **1**

2 Background **3**
2.1 ROS . . . 3
2.1.1 Communication . . . 3
2.1.2 ROS Launch . . . 5
2.2 Dynamic Reconfiguration . . . 6
2.3 Software Testing . . . 6
2.3.1 Software Performance Metrics in Autonomous Driving . . . 7
2.3.2 Recorded Data . . . 8
2.3.3 Simulation . . . 9

3 Implementation **11**
3.1 Problem Description . . . 11
3.2 Sources of Nondeterministic Callback Sequences . . . 13
3.2.1 Lost or Reordered Messages . . . 13
3.2.2 Inputs From Parallel Processing Chains . . . 14
3.2.3 Multiple Publishers on the Same Topic . . . 15
3.2.4 Parallel Service Calls . . . 15
3.3 Design Goals . . . 16
3.4 Controlling Callback Invocations . . . 18
3.4.1 Callback Outputs . . . 20
3.4.2 Timer Callbacks . . . 21
3.4.3 Callbacks for Time-Synchronized Topics . . . 22
3.5 Ensuring Sequence Determinism Using Callback Graphs . . . 23
3.6 Node and System Description . . . 27
3.6.1 Node Configuration . . . 27
3.6.2 Launch Configuration . . . 28
3.7 Dynamic Reconfiguration . . . 28
3.8 Launch System . . . 29

4 Evaluation **33**
4.1 Verification of Functionality . . . 33
4.1.1 Lost or Reordered Messages . . . 33

4.1.2 Inputs From Parallel Processing Chains 34
4.1.3 Multiple Publishers on the Same Topic 36
4.1.4 Parallel Service Calls 37
4.1.5 Discussion 39
4.2 System Setup 40
4.3 System Integration 41
4.3.1 Simulator 42
4.3.2 ROS Bag Player 42
4.3.3 ROS Nodes 43
Planning Module 43
Tracking Module 44
Recorder Node and Ego-Motion Estimation 45
4.3.4 Discussion 45
4.4 Application to existing Scenario 46
4.4.1 Simulator 46
4.4.2 ROS Bag 47
4.4.3 Dynamic Reconfiguration 49
4.4.4 Discussion 51
4.5 Execution-Time impact 52
4.5.1 Analysis 52
4.5.2 Discussion 54

5 Conclusion **57**

Acronyms **59**

Bibliography **61**

List of Figures

2.1 The ROS 2 client library API stack. 4

3.1 An exemplary ROS node graph of an autonomous-driving test setup. 12
3.2 Node graph showing a data source S and processing node P, connected with topic M. 13
3.3 Node graph showing a data source S and node T connected by two parallel topics $D1$ and $D2$. 14
3.4 Node graph showing data source S and node T connected by two parallel paths, each path contains a processing node. 14
3.5 Node graph showing data source S and node T connected by two parallel paths, where the processing nodes on both paths use the same output topic. 15
3.6 Node graph showing three nodes, two of which concurrently call a service provided by the third node. 16
3.7 Considered architecture of running all components within a custom execution environment, without using ROS. 18
3.8 Considered architecture of integrating the orchestrator directly into the ROS client library stack. 19
3.9 Chosen architecture of an external orchestrator component, that intercepts all communication between nodes on a ROS topic level. 20
3.10 Visualization of the ROS topic interception of node inputs by the orchestrator. 21
3.11 Callback graph for two inputs into a ROS graph with two parallel processing paths as shown in Fig. 3.4. 26
3.12 Sequence diagram of communication between orchestrator and reconfigurator during the dynamic reconfiguration step. 30

4.1 Sequence diagram showing dropped messages due to subscriber queue overflow. 34
4.2 Sequence diagram showing a slowdown of the data source to prevent dropping messages by overflowing the subscriber queue. 34
4.3 Sequence diagram showing the execution of two parallel processing nodes with nondeterministic processing time. 35

4.4 Sequence diagram showing a deterministic callback order at T despite nondeterministic callback durations at $P1$ and $P2$. 35
4.5 Sequence diagram showing serialized callback executions of nodes $P1$ and $P2$, which is required to achieve a deterministic callback order. . 36
4.6 Sequence diagram showing the parallel execution of callbacks at $N1$ and $N2$, which both call the same service. 38
4.7 Sequence diagram showing the serialized callbacks from Fig. 4.6. . . . 38
4.8 Node graph of the system setup used within Chapter 4. 41
4.9 Evaluation of the MOTA and MOTP metrics using the experimental setup. 47
4.10 OSPA distance of tracks versus ground truth during multiple simulation runs. 50
4.11 Absolute difference in OSPA distances between the simulation runs. . 50
4.12 OSPA distance of tracks versus ground truth over time, comparison between simulation run with and without the orchestrator. 51
4.13 Comparison of execution time for one simulation run. 53

1 Introduction

The growing complexity of robotics software stacks necessitates the use of frameworks that allow the separation of individual components and explicit communication between them. In the research community, in particular, open-source frameworks have prevailed over proprietary middleware or custom full-stack implementations. Their ease of use and the ease of sharing and re-using loosely coupled components often outweigh potentially stronger guarantees about reliability and determinism or performance benefits of competing options. The Robot Operating System (ROS) is such an open-source framework that has gained popularity in both research and industry over the last decade ([MFG+22]). It provides abstractions for building modular robotics software and serves as a communication middleware between components. The event-triggered execution model of ROS combined with its weak guarantees for the message-passing implementation can however not guarantee the deterministic execution of an entire software stack. This lack of repeatability presents a problem for the testing and evaluation of software components, which has become an integral part of the development cycle of modern robotics applications. Especially in domains such as automated driving, continued verification of the expected system performance is critical for assuring the safety of the entire system. Without deterministic execution of test cases, observed changes in resulting metrics can not be traced back to specific software changes with certainty, since random variations may manifest from nondeterministic execution. This is especially critical when dynamically changing parameters and functionality by runtime reconfiguration of the software stack, which is desirable for example for adaptive, situation-aware environment perception methods as proposed in [HMG+23].

In this thesis, a method is presented to enable repeatable execution of ROS software stacks by ensuring a deterministic callback sequence at each individual ROS node. This is enabled by iteratively building a callback graph from pre-defined specifications of node behavior. Using the callback graph, data flow within the software stack can be specifically inhibited to ensure deterministic callback ordering at each node.

Chapter 2 provides the necessary background on ROS and introduces the relevant types of software testing and evaluation. In Chapter 3, first, the nondeterministic communication and callback behavior and its consequences are described. Then, four fundamental sources of nondeterministic callback ordering are identified. The rest of Chapter 3 describes in detail the method of ensuring deterministic callback execution

and the design and implementation of the orchestrator, which is the newly developed component implementing the proposed method. In Chapter 4, the functionality of the implementation is verified and discussed using the previously identified minimal examples, and usability is assessed by integrating existing components of an existing autonomous driving stack. Chapter 5 summarizes the findings and gives an outlook on future enhancements.

2 Background

This chapter introduces ROS and motivates the additional use case of dynamically reconfiguring ROS systems. Additionally, background on robotics software testing methodology is given, which will form the context and intended use of the method proposed in Chapter 3.

2.1 ROS

The Robot Operating System (ROS) [MFG+22] is a software framework used by robotics developers and the wider research community to implement reusable software components. It provides valuable abstractions and libraries in the areas of communication between components, system startup, configuration, and introspection. Since its first release in 2010, it has gained wide adoption in the robotics community, with thousands of packages released which provide common functionality in the form of ROS components that can be integrated into complete software stacks. In 2017, the first distribution of ROS 2 was released. This marked a complete redesign and introduced multiple new features explained in detail below, such as Quality of Service (QoS) and the reliance on the Data Distribution Service (DDS) standard for message transport. This work is entirely using ROS 2, and while core concepts have changed little, important details relied upon here do not apply to ROS 1.

This section serves as a brief introduction to the concepts of ROS nodes and topics, which are the basic primitives allowing communication between software modules, as well as the launch system.

2.1.1 Communication

Individual software components within a ROS stack are referred to as *nodes*. A node typically has one distinct functionality, and well-defined inputs and outputs. An example may be an object detector node, which has a camera image as input, and a list of object hypotheses as output. Usually, although not always, each ROS node runs in a dedicated process.

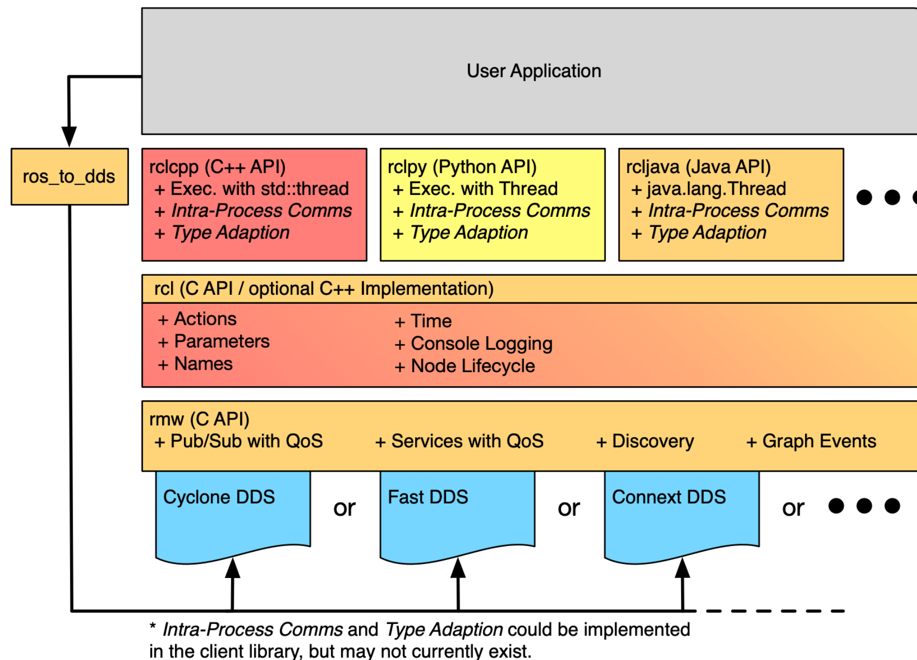


Figure 2.1: The ROS 2 client library API stack, showing the ability to both support multiple underlying communication middlewares (DDS implementations) and client library bindings in multiple languages. Diagram by the contributors of the ROS 2 documentation on docs.ros.org, licensed under CC BY 4.0.

Every ROS Node is a participant in the ROS communication graph, which represents the connections between the inputs and outputs of multiple nodes. Multiple such communication graphs (or "ROS node graphs") are shown in this work, such as in Fig. 3.1. Communication between nodes is message-based and happens via topics, which are multi-producer, multi-consumer message channels. Nodes can send messages to topics using publishers, and receive messages from topics using subscribers. For each publisher and subscriber, nodes can configure a number of QoS settings, notably, they can request best-effort or reliable transport and can configure queue sizes. For publishing, the application calls a simple Application Programming Interface (API) method of the publisher object, which may directly transfer the message or push the message to a queue for later, asynchronous transmission. Receiving messages from subscribers is realized by registering a callback function with the ROS API.

While ROS provides this functionality to the nodes by its API, the underlying functionality of message delivery and node discovery relies on existing implementations of the DDS standard. Figure 2.1 shows the API stack, with the user code at the very top and the DDS implementation at the bottom. The diagram illustrates that ROS forms a common layer that enables the use of multiple DDS implementations within

the middleware (visualized as the blue elements at the bottom of the diagram) and the use of different programming languages for application development (*ros client library* bindings, directly below the user application in the diagram).

An important aspect of the ROS node communication model is that there is no implicit or explicit back-channel or feedback to the publisher of a message about its (intended) reception. This implies that there exists no concept of back pressure or congestion: If a downstream node is not able to process messages at the rate at which they are published, they will be queued according to the nodes queuing policies (dropping messages if necessary), but upstream nodes producing the messages will not be throttled or otherwise notified.

Individual topics in ROS are identified by name and type, whereby the type is an externally defined structure of named elements which themselves are other ROS types or of a predefined type such as string, numeric, or array types. Topics are created as soon as a node creates a corresponding publisher and subscriber, and two nodes must use the matching name and type to communicate over a topic. In order to allow flexibility when using a node in different environments, the possibility of changing internally used names while starting a node is provided, and referred to as *name remapping*.

ROS does provide additional mechanisms for communication patterns that do not fit the publish-subscribe model: ROS *services* provide a method of one-way remote procedure calling between ROS nodes. A ROS node can provide a service by registering a service *server*, which can then be called by other nodes using a service *client*. Identically to topics, services are identified by name and type, where the type of a ROS service includes both the request and response type. In contrast to topics, a service call always has a response, which the caller can await.

An additional mechanism built on top of services is *actions*, which are a ROS way of controlling long-running, interruptible, tasks running within a ROS node. Since these are however fundamentally built atop of services and topics and are not used in the ROS software stack used for evaluation in this work, actions are not of special interest to this work.

2.1.2 ROS Launch

Since ROS systems usually consist of multiple nodes, a method for starting an entire robotics software stack is part of the ROS ecosystem. The ROS launch system allows developers to specify actions required to bring up a system in a launch file. Those launch files are typically Python scripts, and actions include launching node processes, including other launch files, or setting parameters.

The possibility to include other launch files allows developers to create subsystems

that themselves consist of multiple nodes in a specific configuration. Typically, name remapping is used within the launch file to connect multiple nodes, by setting their internal, generic names for subscribers and publishers to a common name.

In this work, the launch system will be utilized to redirect subscriptions of nodes under test by setting appropriate name remappings, and then including the original launch file as a subsystem, with those parameters applied (further details are provided in Section 3.8).

2.2 Dynamic Reconfiguration

The combination of a specific set of active components, their specific connections, and parameters is referred to as the *system configuration.* The above section describes how a static, or initial system configuration is specified by the launch file.

Recently, however, research has gone into finding the optimal system configuration depending on the current operating environment, in order to minimize processing requirements while maintaining sufficient system performance [HMG+23].

Such a dynamic reconfiguration may be realized by a dedicated software component, which evaluates the current situation on the basis of available sensor data and environment information. This module may then decide to perform a system reconfiguration when appropriate, and as such may start and stop nodes, or change parameters for running nodes.

To enable this use case, it is necessary to allow changing the system configuration during runtime. ROS allows starting and stopping nodes at any time, and new publishers and subscribers can join existing topics. Parameters within ROS nodes may also be changed during runtime, although the specific node implementation may choose to only read parameters once during startup. While this is generally possible within ROS, the interaction of dynamic reconfiguration with the work presented in this thesis requires special attention (Section 3.7), due to the additional information about system behavior required by the proposed method.

2.3 Software Testing

While testing has long been considered an essential part of all software development, it is both especially important and uniquely challenging for robotics, and in particular automotive, software development. Research in autonomous driving aims to improve road safety, but this places the responsibility over the safety of occupants and especially

other traffic participants on the software, which makes testing and verification of correct behavior essential.

The type of testing relevant to this work can be classified as integration- or system testing. In the context of ROS software stacks, this amounts to testing one or multiple ROS nodes entirely, in contrast to more specific testing which would directly test an algorithm inside a node, without taking the ROS-specific code into consideration. This work considers performance testing, meaning testing that determines how well the application or system completes the desired task. Additionally, the focus lies explicitly on post-processing testing instead of determining system metrics during runtime. In an autonomous driving context, this amounts to testing using a simulator or recorded data, and not online performance testing during test drives. Other testing methods may verify attributes related to software quality and resilience, but those are not of particular interest in this work. Achieving reproducibility is especially difficult for those testing methods involving multiple components and their interaction and communication, which is what this work aims to address by ensuring deterministic execution.

Regression testing describes the practice of verifying that the performance of the system under test does not fall below previous test executions. As a special case of regression testing, one could verify that the output of the system *exactly* matches a previous output. This allows the developer to verify that presumably non-functional changes do indeed not modify the observable system behavior, which may have previously been quantitatively evaluated.

2.3.1 Software Performance Metrics in Autonomous Driving

A variety of metrics have been proposed for quantitative evaluation and comparison of both the whole-system performance of autonomous driving software stacks, as well as individual software components within such a stack.

One possibility for assessing the entire system performance of an autonomous driving stack is to measure criticality. Criticality is defined by [NWB+21] in Definition 1 as “the combined risk of the involved actors when the traffic situation is continued”. In [WNK+23], an overview and comparison are given of metrics that measure the criticality of a traffic scenario, many of which use models for driver behavior in order to predict dangerous situations by factors such as small distances or large relative speeds. Notably, the authors of [WNK+23] explicitly assume a deterministic testing environment, in which repeating the same inputs yields the same outputs. Since those metrics evaluate the resulting traffic situation, they require running the entire software stack, even when the influence of only a single module on the result is to be determined.

As an example for performance evaluation using application-specific metrics, multiple metrics for a multi-object tracking module are considered. Specifically, the Multiple Object Tracking Precision (MOTP) and Multiple Object Tracking Accuracy (MOTA) metrics as proposed in [BS08] are used in this work. MOTP is defined as the average distance error d over all matches i in each timestep t (with c_t the number of matches between detections and ground-truth objects in timestep t)

$$\text{MOTP} = \frac{\sum_i^t d_t^i}{\sum_t c_t}.$$

MOTA provides a measure for how well the tracking algorithm performs with respect to missed objects (m), false positives (fp), and track mismatches (mme, i.e. identity switches between identified objects) over the total number of objects g_t, as defined by

$$\text{MOTA} = 1 - \frac{\sum_t (m_t + fp_t + mme_t)}{\sum_t g_t}.$$

Both metrics are calculated over an entire sequence, instead of individual frames.

An additional metric for multi-object tracking applications is the Optimal Subpattern Assignment (OSPA) metric as defined in [SVV08]. This metric directly measures the distance between two sets of states with different cardinality, and can thus be calculated for each timestep instead of over an entire sequence. The OSPA metric of order p is defined for two sets $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_n\}$ and a distance measure $d^{(c)}(x, y)$ with cutoff at c as

$$\bar{d}_p^{(c)}(X, Y) = \left(\frac{1}{n} \left(\min_{\pi \in \Pi_n} \sum_{i=1}^{m} d^{(c)}(x_i, y_{\pi(i)})^p + c^p(n - m) \right) \right)^{1/p}.$$

In the context of multi-object tracking, the sets X and Y represent the estimated tracks at a specific time step and the corresponding ground truth states. The resulting distance may then be interpreted as the average distance between a track and its corresponding ground truth object, with unassigned tracks being assigned the cutoff value c. This metric will be used in Section 4.4.3 to visualize a change in the system performance during a single simulation run, which would not be visible using a metric that is averaged over the entire sequence.

2.3.2 Recorded Data

Evaluation and testing of robotics software is often not performed during runtime, but instead using pre-recorded input data. This enables fast iteration and comparison of approaches, methods, or versions thereof with the same inputs. Specific publically available datasets have evolved into de-facto standards, which allows comparison

and benchmarking within the entire research community. These datasets are usually accompanied by ground-truth annotations, which are often required to calculate application-specific metrics. Some benchmarks focus on comparing system-level benchmarks and evaluating multiple modules, such as the NuPlan benchmark ([CKT+22]) which aims to compare the resulting long-term driving behavior in a closed-loop simulation.

The nuScenes dataset ([CBL+19]) for example contains camera images as well as lidar and radar measurements from an autonomous vehicle, as well as annotations for class and bounding box of visible objects, and is used extensively to evaluate object detectors in the autonomous-driving context. In those benchmark datasets, input data is commonly available in a format specific to that benchmark. For use within ROS, these formats are often converted to ROS bags, which provide a standard method for storing message data within ROS at a topic level. For direct recording, the ROS bag recorder is available. It subscribes to specified topics, and stores every received message to disk in its serialized format, together with metadata required for replaying the messages. To replay a bag, the ROS bag player creates publishers for every topic recorded in the bag and publishes the messages in the same order as recorded.

Time handling during ROS bag replay differs from the normal execution of a ROS software stack: Since ROS messages may (and often do) contain timestamps of data acquisition or message creation, and nodes expect to compare them to the current time, a desired functionality is to replay not only the messages but also the time of recording. This is supported in ROS by delegating timekeeping to the ROS client library as well, which then subscribes to the well-known `/clock` topic to allow overriding the node's internal clock. The ROS bag player then periodically publishes this topic with the time of recording, setting all node clocks.

2.3.3 Simulation

Using a simulator is another method for off-robot software testing besides using recorded sensor data. A simulator allows for closed-loop execution of the software stack or module under test. This allows the evaluation of more modules, such as planning or control algorithms, which directly and immediately influence the robot's behavior.

A large number of robotics simulators have been developed, each with specific use cases and goals, even in the context of autonomous vehicles alone: General robotics simulators such as Gazebo ([KH04]) feature a general physics engine capable of simulating arbitrary robots with involved locomotion techniques and a large variety of sensors. Application-specific simulators such as CARLA ([DRC+17]) utilize existing rendering engines to simulate typical sensors such as cameras and LIDAR in high fidelity, and use specific models for simulation of relevant objects such as vehicles and

other traffic participants. Higher-level simulation tools do not simulate individual sensor measurements, but the output of detectors, greatly reducing the computational effort at the cost of not being able to use and test specific detection modules.

The simulator used for evaluation in this work is the DeepSIL framework introduced in [SMHB21]. While the specific deep-learning-based trajectory prediction features are not used here, it provides a representative baseline for a simulator in use for autonomous-driving development, in order to evaluate the integration effort of the proposed framework. In the configuration used for evaluation, DeepSIL generates detections from virtual sensors and detection algorithms and simulates vehicles either by using a driver model or using control inputs generated by external planning and control modules. The simulated detections, simulated vehicle state estimation as well as ground truth object states are published to the software under test via ROS topics.

3 Implementation

This chapter will first introduce the problem of nondeterministic behavior in ROS using minimal examples in Sections 3.1 and 3.2. Then, in Section 3.3, the design goals and intended use cases for the implemented software are determined. Lastly, Sections 3.4 to 3.8 describe the implementation in detail, covering the control of callback invocations, ordering of callbacks using dependency graphs, node and system configuration files, as well as details concerning dynamic reconfiguration.

3.1 Problem Description

An important property of all testing and evaluation approaches outlined in Section 2.3 is determinism. A nondeterministic simulation for example may result in substantially different scenarios leading to changing evaluation metrics over multiple test runs. If the software under test itself is not deterministic, a regression test might fail even if no functional changes to the software have been made.

It can be observed, however, that even with deterministic data sources such as simulators and recordings, and deterministic algorithms under test, resulting metrics in ROS systems may be nondeterministic. This is due to nondeterministic callback execution: Varying processing times of intermediate modules and nondeterministic behavior and latencies within the communication middleware can change the order in which callbacks are executed, which may alter a node's behavior, even if the content of the individual input messages is consistent.

Figure 3.1 displays an example ROS node graph for a testing setup in an autonomous-driving context. A simulator produces noisy object detections, which are published on multiple topics. The tracking module receives those detections and sends a combined list of tracked objects to the planning module, which generates a vehicle trajectory and appropriate control signals. Those are fed back to the simulator, updating the vehicle state. The evaluation module receives ground-truth data from the simulator and the results of the tracking and planning modules. It calculates metrics for evaluating the performance or safety of planned vehicle trajectories. The individual nodes are all assumed to be input/output deterministic, meaning that when repeating the exact same sequence of inputs, the same sequence of outputs is produced. Nondeterministic

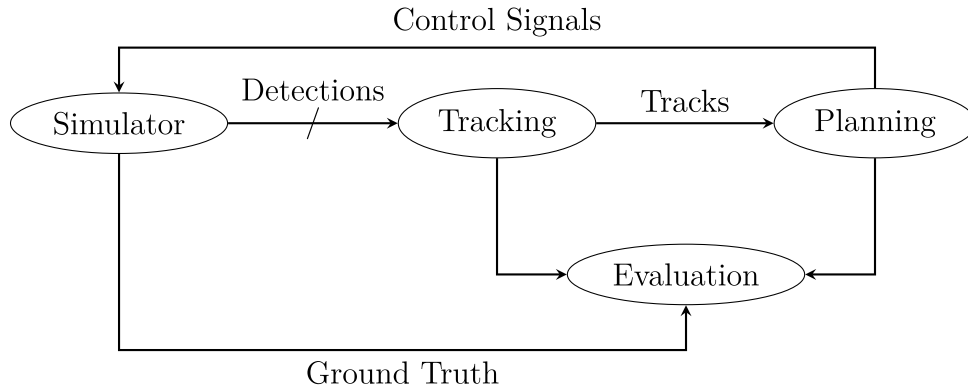


Figure 3.1: An exemplary ROS node graph of an autonomous-driving test setup.

behavior of the entire system may however occur in multiple situations, which all influence the order of callbacks at each node:

- The control feedback from the planning module to the simulator has variable delay, and may happen during an earlier or later simulation timestep.
- The order of received detections at the tracking module is nondeterministic, which may influence its result.
- If the planning module is triggered by a timer instead of by data input, it may happen before or after an input is received from the tracking module.

Those aspects may additionally be influenced by external factors such as system load and scheduling or choice of middleware implementation, leading to different evaluation results for each run.

While the system may be expected to be resilient toward such arguably small deviations in runtime behavior, such that those do not significantly degrade system performance, eliminating those is still desirable. If the system is fully deterministic with respect to callback execution, any change in performance can be unambiguously attributed to the changes made to algorithms and parameters, and performance evaluation is perfectly repeatable. Additionally, regression testing for non-functional changes would be possible by demanding exact equality of system output in response to simulated or recorded inputs.

3.2 Sources of Nondeterministic Callback Sequences

In this section, minimal examples of nodes and connecting topics will be presented, which introduce nondeterministic behavior even for deterministic nodes.

3.2.1 Lost or Reordered Messages

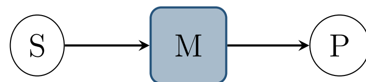


Figure 3.2: Node graph showing a data source S and processing node P, connected with topic M.

Figure 3.2 shows two ROS nodes communicating via one topic, without any additional publishers or subscribers connected to the topic. In this scenario, the sending node publishes messages at a high rate, while the receiving node processes messages slower than required to handle every message. This causes the subscriber queue to fill up, eventually dropping messages. Current ROS defaults use the `keep last N` queue handling strategy, which would cause the oldest message to get dropped from the queue when a new one arrives. Under varying system load, the number of processed messages changes, which leads to nondeterministic node behavior.

It should be noted that this can not be avoided by using the reliable QoS setting in ROS. A reliably delivered message may still cause another message to be dropped from the subscriber's queue if there is no space for the incoming message. Messages actually getting lost during delivery, which may happen using the best-effort QoS setting on a constrained transport medium, such as a low-bandwidth wireless network, are not handled here. A possible measure against this behavior is the `keep all` queuing mode, but this is often not feasible, since this may cause the queue size as well as the input-output latency of the node to grow without bounds.

Finally, message reordering might be of concern. The DDS standard allows ordering incoming data in the `BY_RECEPTION_TIMESTAMP` mode, which implies that the receive order might not match the order in which the messages were published. While ROS does not make any claims regarding message ordering, it is assumed that the reliable QoS setting eliminates message reordering. Nonetheless, message reordering, should it occur, is later also addressed by the same mechanism as possible queue overflow.

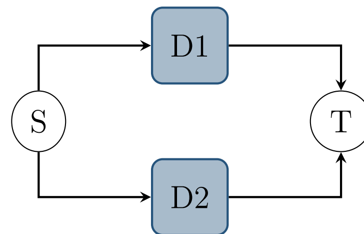


Figure 3.3: Node graph showing a data source S and node T connected by two parallel topics $D1$ and $D2$, on which messages are published simultaneously by S.

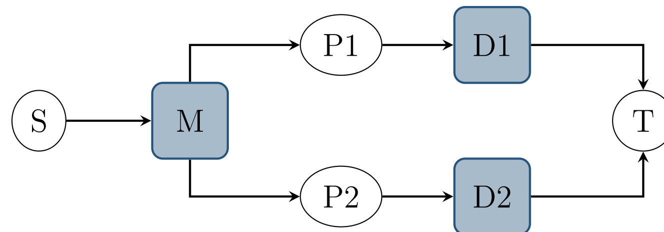


Figure 3.4: Node graph showing data source S and node T connected by two parallel paths. Each path contains a processing node with a dedicated output topic. Both paths share the same input topic M.

3.2.2 Inputs From Parallel Processing Chains

In this scenario, a node receives messages on multiple topics, which originate from the same event in no specified order. In Fig. 3.3, node S publishes a message to both topics $D1$ and $D2$ during the same callback. Usually, those messages would be regarded as published at the same time. This results in a nondeterministic receive order of both messages at node T, since transmission latency might differ.

In Fig. 3.4, a similar scenario is shown. Node T again has two input topics $D1$ and $D2$, and a message on both topics is triggered by a single callback at node S. Compared to the previous example however, S publishes a single message on topic M, that is then processed by both nodes $P1$ and $P2$, which then produce the outputs on $D1$ and $D2$. This exhibits the same problem of nondeterministic receive order of both messages at node T, and does so even if some assumptions about S and the transmission latency can be made. First, the nodes $P1$ and $P2$ add a nondeterministic processing latency to the total latency between S and T. This results in nondeterministic latency, even if the transmission latency of the ROS topic was constant. Second, the data source S publishes only a single message. In the previous example, deterministic behavior might be achieved if the middleware were to guarantee immediate and synchronous delivery of messages, and if the publish order within S was deterministic. Although these assumptions are not made about the ROS middleware, and generally do not

hold, this demonstrates that the problem persists even with stronger guarantees from the middleware.

3.2.3 Multiple Publishers on the Same Topic

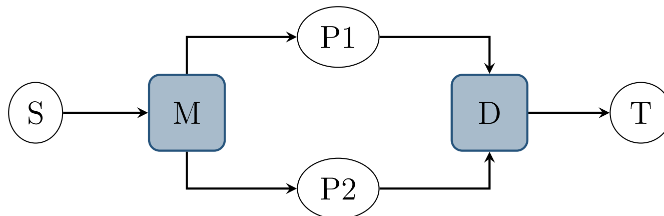


Figure 3.5: Node graph showing data source S and node T connected by two parallel paths, where the processing nodes on both paths use the same output topic D. Both paths also share the same input topic M.

This scenario again consists of a data source S, two processing nodes $P1$ and $P2$ and a node T which receives the outputs of $P1$ and $P2$, as shown in Fig. 3.5. Once S publishes a message, both processing callbacks at $P1$ and $P2$ run concurrently, eventually publishing an output. Distinct from the previous example, $P1$ and $P2$ use the same output topic D, which consequently is the only input of T. The communication middleware does not guarantee that the message delivery order at T matches the publish order at $P1$ and $P2$. This results in a nondeterministic arrival order of both messages at T. Note that while $P1$ and $P2$ run concurrently in this example, this would still be a concern if the processing nodes were triggered by separate inputs since callback duration and transmission latency would still be nondeterministic.

As with the scenario in Section 3.2.1, subscriber queue overflow is an additional concern here. If the subscriber queue of T is full already, a message from either publishing node may be dropped.

3.2.4 Parallel Service Calls

This example involves four nodes, as shown in Fig. 3.6: One node S publishes a message to topic M, which causes subscription callbacks at nodes $N1$, $N2$ and SP. SP provides a ROS service, which the nodes $N1$ and $N2$ call while executing the input callback. This causes three callbacks in total at the service provider node, the order of which is nondeterministic. In this case, this influences not only the future behavior of the service provider node but also the result of the callbacks at nodes

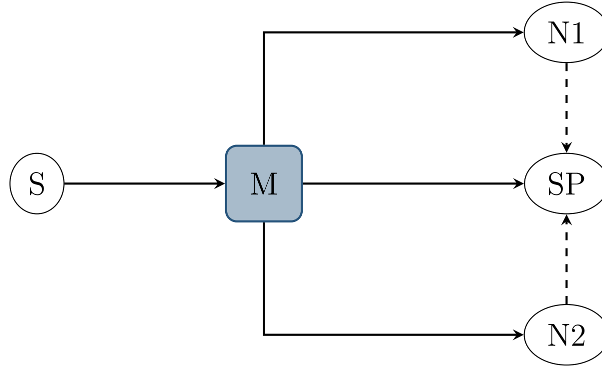


Figure 3.6: Node graph showing three nodes $N1$, $N2$ and SP all with topic M as an input. Nodes $N1$ and $N2$ call a service provided by SP during callback execution, as indicated by the dashed arrows.

$N1$ and $N2$, since each service response might depend on previous service calls and message inputs.

3.3 Design Goals

The goal of this thesis is to provide a framework for the repeatable execution of ROS systems, circumventing the nondeterminism caused by the communication middleware and varying callback execution duration.

In particular, the framework shall ensure that the sequence of callbacks executed at each node is deterministic and repeatable, even with nondeterministic callback durations of the entire system, arbitrary transmission delay of messages, and without guarantees of message delivery order in specific topics and between topics. This leads to fully deterministic system execution, provided the input data is deterministic, the system contains no hidden state beyond each node's state, and all ROS nodes have a deterministic input/output behavior. The component controlling callback execution in this way will in the following be referred to as the *orchestrator*.

The use case for this framework is that of a researcher or developer who is evaluating the entire software stack or a specific module within the stack by some application-specific metric. The researcher expects consistent results across multiple executions and expects that changes in the resulting measure only result from changes in software configuration. Using the orchestrator during live testing, such as when performing test drives of an autonomous driving system, is explicitly not intended, as the goal of ensuring deterministic callback ordering might stand in conflict with the goal of minimizing system latency during live execution.

It is anticipated that some or all nodes within the software stack under test will need to be modified in a way to make them compatible with the framework. These necessary modifications shall be kept to a minimum and should leave the node fully operational without changes in its behavior when the framework is not in use. Additionally, the ability to integrate nodes that are non-trivial to modify is desirable. This might be the case when using external ROS nodes, not only because the developer would usually be unfamiliar with that node's source code, but also because locally building that node might be considerably more effort compared to installing a binary distribution.

An additional design goal is to allow for runtime reconfiguration of the software stack. This includes starting and stopping nodes, changing the parameters of running nodes, or changing the communication topology. The use of the framework should not prohibit runtime reconfiguration, such as by requiring a static node graph, and the reconfiguration step itself should not cause any nondeterministic behavior.

The intended use case dictates the use of both recorded data (ROS bags) and simulators as sources of input data. For recorded data, existing ROS bags shall be usable, since re-recording data is costly and large repositories of recorded data often already exist. When using a simulation, the framework should work with existing simulators, and the integration effort shall be minimized. In the following, the specific data source used is referred to as the *data provider*.

Finally, the execution time impact of using the orchestrator shall be minimized. Ensuring a deterministic callback order will involve inhibiting callback execution for some time, and running callbacks serially that would otherwise run in parallel. Both this induced serialization overhead, as well as the runtime of the orchestrator itself, should be sufficiently small so as to not interfere with a rapid testing and development cycle.

3.4 Controlling Callback Invocations

In all the scenarios presented above, deterministic execution can be achieved by delaying the execution of specific callbacks in such a way, that the order of callback executions at each node is fixed. Multiple methods of controlling callback invocations have been considered, which also directly influence the general architecture of the framework:

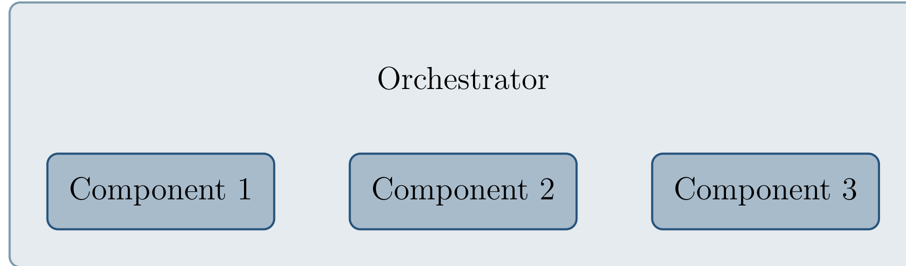


Figure 3.7: Considered architecture of running all components within a custom execution environment, without using ROS.

The first approach is to completely avoid the ROS communications middleware and directly invoke the component's functionality, without running the corresponding ROS callbacks. This would completely replace the ROS client library or corresponding language bindings, at least for the testing and evaluation use case, and provide a fully custom, and thus entirely controllable, execution environment. Figure 3.7 shows the individual components contained within the orchestrator, without the ROS-specific functionality. While this approach would provide the largest amount of flexibility, and no dependency on or assumptions about ROS, this has been considered not feasible.

While some ROS nodes cleanly separate algorithm implementation and ROS communication, and allow changing the communication framework easily, this is not the case for many of the ROS nodes considered here. If a ROS node includes functionality that is tightly coupled to the ROS interface, this would require a considerable re-implementation effort. This also introduces the possibility of diverging implementations between the ROS node and the code running for evaluation, which would reduce the significance of the results obtained from evaluation and testing. Additionally, this design represents a stark difference from the ROS design philosophy of independent and loosely coupled components.

The second possible approach is to modify the ROS client library in order to control callback execution on a granular level. Callback execution in ROS nodes is performed by the executor, and while multiple implementations exist, the most commonly used standard executor in the ROS Client Library for C++ (`rclcpp`) has previously been described in [CBLB19]. The executor is responsible for fetching messages from

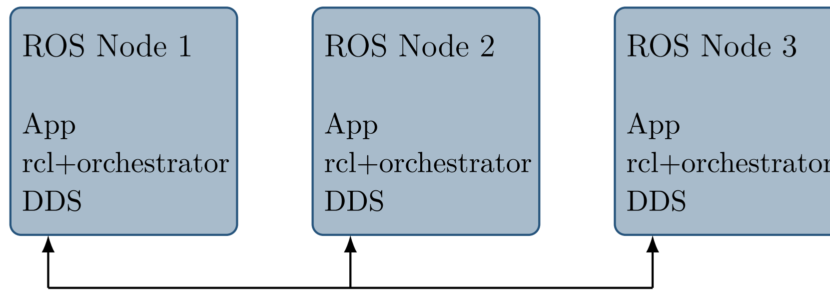


Figure 3.8: Considered architecture of integrating the orchestrator directly into the ROS client library stack to control callback invocations via the executor. The arrows represent ROS topics connecting the nodes, which would not be changed or modified using this approach.

the DDS implementation and executing corresponding subscriber callbacks. It also manages time, including external time overrides by the `/clock` topic, and timer execution. On this layer between the DDS implementation and the user application, it would be possible to insert functionality to inhibit callback execution and to inform the framework of callback completion, as shown in Fig. 3.8. Instrumenting the ROS node below the application layer is especially desirable since it would not require modification to the node's source code. This approach does however present other difficulties: While there is a method to introspect the ROS client libraries via the ros_tracing package, RCL does not offer a generic plugin interface or other methods to inject custom behavior. This leaves modifying the RCL implementation, and likely also the two most popular language bindings, the ROS Client Library for Python (`rclpy`) and `rclcpp` for C++, and building all nodes with those modified versions. Modifying and distributing those libraries as well as keeping them up to date with the upstream versions, however, present a considerable implementation overhead. Using custom `rclpy` and `rclcpp` versions additionally inconveniences library users, since the orchestrated version exhibits different behavior to the unmodified library, which can be unexpected and difficult to introspect.

The final approach taken is to intercept the inputs to each node on the ROS-topic level: The orchestrator exists as an external component and individual ROS node and ensures that all communication passes through it, with no direct connections remaining between nodes, as visualized in Fig. 3.9. With the knowledge of the intended node inputs (which are specified in description files, as described in Section 3.6.1), the orchestrator can now forward messages in the same way as with the originally intended topology. Additionally, however, the orchestrator can buffer inputs to one or multiple nodes, thereby delaying the corresponding callback execution. Since the orchestrator is not expected to execute additional callbacks (which would require generating or repeating messages), delaying callbacks is sufficient to control the node's behavior. By assigning every subscriber to a specific topic an individual connection (a

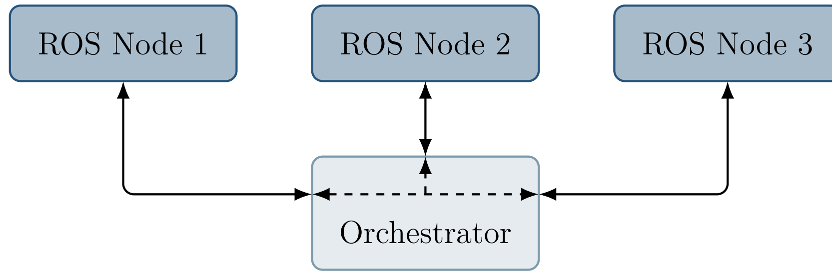


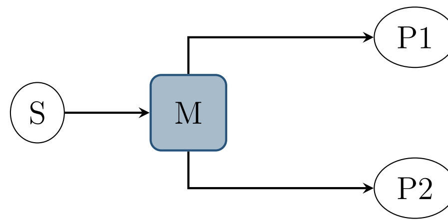
Figure 3.9: Chosen architecture of an external orchestrator component, that intercepts all communication between nodes on a ROS topic level.

distinct topic) to the orchestrator, it is also possible to separate callback executions for the same topic at different nodes. For inputs into the orchestrator, such separation is not required, since the orchestrator can ensure sequential execution of callbacks which publish a message on the corresponding topics. Figure 3.10 shows an example of a one-to-many connection between three nodes using one topic. When using the orchestrator, M is still used as an output of S, but each receiving node now subscribes to an individual input topic *P1/M* and *P2/M*.

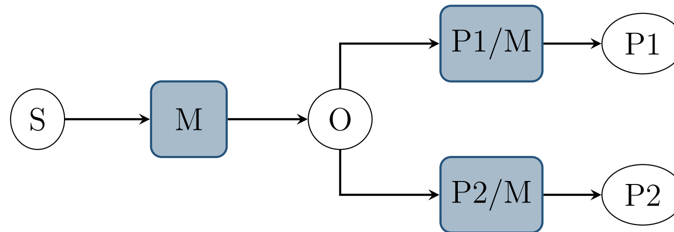
The orchestrator ROS node is typically located in the same process as the data provider, which would be a simulator or ROS bag player. This allows both components to interact directly via function calls, which greatly simplifies the interface compared to, for example, ROS service calls.

3.4.1 Callback Outputs

ROS callbacks may modify internal node state, but may also produce outputs on other ROS topics. The orchestrator needs to know which outputs a callback may have, and also when a callback is done, in order to allow new events to occur at the node. The possible outputs are configured statically, as detailed in Section 3.6.1. If a node omits one of the configured outputs dynamically, or if a node does not usually have any outputs which are visible to the orchestrator, a status message must be published, the definition of which is available in Listing 1. The `omitted_outputs` field optionally names one or multiple topics on which an output would usually be expected during this callback, but which are not published during this specific callback invocation.



(a) Before interception: Data source S publishes to topic M, which is an input to nodes $P1$ and $P2$.



(b) Interception using orchestrator O: The orchestrator subscribes to M and publishes to individual input topics for each node $P1$ and $P2$, allowing individual callback execution.

Figure 3.10: Visualization of the ROS topic interception of node inputs by the orchestrator.

```
1 string node_name
2 string[] omitted_outputs
```

Listing 1: ROS message definition of the status message, which informs the orchestrator that the specified node has completed its last callback. Optionally, a list of omitted outputs can be specified.

3.4.2 Timer Callbacks

Intercepting topic inputs also allows controlling timer callback invocations, although some limitations apply. Both in simulation and during ROS bag replay, node time is usually already controlled by a topic input through the `/clock` topic. This allows the node to run as expected during slower than real-time simulation and playback. Since the clock messages only contain the current time (and not information such as the playback rate), and ROS does not extrapolate this time, this forms a topic input that triggers timer callbacks. Like any other topic input, this topic name can be remapped to form a specific clock topic for each node, allowing triggering timer callbacks at each node individually.

This approach is limited, however, when multiple timers exist at the same node: Even if the timers are configured to different frequencies, the timer invocations will inevitably occur at the same time at some point. In that instant, the `/clock` input

triggers both (all) timers, without the ability to specifically target the callback of an individual timer. With two callbacks running simultaneously (and depending on the executor, possibly in parallel), nondeterministic message ordering may occur if, for example, both timer callbacks publish a message to the same topic. Thus, using multiple timers at the same node is only acceptable if the corresponding outputs are separate. Additionally, simultaneous execution must not change the internal node state nondeterministically, which may be ensured by using a single-threaded executor that executes the timer callbacks sequentially in a consistent order.

Using only one timer per node eliminates this problem as well, although there remains one instance where multiple timers fire at once: When each node receives the first clock input, the internal clock jumps from zero to the initial simulation or recorded time. This results in the execution of at most one "missed" timer callback, and, if the clock input is a multiple of the timer period, one "current" timer callback. The latter case is immediately observed with a simulation timer starting at a large multiple of one second, and timers running at a fraction of one second. This is an especially challenging situation since both callback invocations correspond to the *same* timer, compared to *separate* timers above. This implies that both callbacks have exactly the same outputs, making it impossible for the orchestrator to differentiate the outputs of both callback executions. A desirable property of a ROS node may be that the node itself only sets up timers when the node-local time has been initialized, which may be possible using ROS 2 "lifecycle nodes", which have the notion of an initialization phase at node startup. In this work, however, it was considered acceptable to discard the outputs of initial timer invocations in that case, since nodes can not usually be expected to perform such initialization.

3.4.3 Callbacks for Time-Synchronized Topics

The `message_filters` package is not part of the ROS client library, but its popularity and interaction with message callback execution make it a relevant component to consider: This package provides convenient utilities for handling the use case in which messages on two or more subscriptions are expected to arrive (approximately) at the same time and need to be processed together. Specifically, it provides the `ApproximateTimeSynchronizer` class which wraps multiple subscribers and calls a single callback with all messages, as soon as messages have arrived on all topics within a sufficiently small time window.

While this makes the node robust against variations in message reception time and order, it complicates reasoning about the node's behavior from the outside. The time synchronizer introduces an additional state to the node in the form of cached messages, which then influences whether a callback is executed or not for subsequent incoming messages. Additionally, the callback behavior is now dependent on the

message contents, since by default the messages are not synchronized by reception time but by timestamp embedded inside the message (which might for example be the acquisition time of contained measurement data).

For handling such callbacks using the orchestrator, the following approach has been taken: For each input of the time synchronizer, it is initially assumed that the combined callback will be invoked. An instance of `ApproximateTimeSynchronizer` is additionally held at the orchestrator, which is then used to determine if the callback is expected to execute or not for a particular input message. Since the message needs to be forwarded even when no callback is expected, a pathological error case emerges. Consider the case in which a `ApproximateTimeSynchronizer` is used to synchronize messages on topics A and B, where A is published at a significantly higher rate than B. The synchronizer may be parameterized in a way such that a message on B might be correctly combined with any of the last few messages on A. This could lead to a scenario where many messages are published on A, without receiving any confirmation, before publishing a message on B, which causes the combined callback. The message B might be combined nondeterministically with any message A, since for example, the latest message on A might not even be received by the node yet.

3.5 Ensuring Sequence Determinism Using Callback Graphs

Once the orchestrator has the ability to individually control callbacks at ROS nodes, it can ensure a deterministic order of callback execution at each node, leading to deterministic system execution. In order to avoid the sources of nondeterminism presented in Section 3.2, the orchestrator constantly maintains a graph of all callbacks which are able to execute in the near future. By introducing ordering constraints between callbacks as edges in the graph, and only executing callbacks when those constraints are met, the possibly nondeterministic situations presented above are sufficiently serialized to guarantee a deterministic callback order. In the following, the elements of the callback graph are discussed in detail:

A callback graph contains nodes for events that occur in the ROS system, the data provider, and the orchestrator itself. Callback graph nodes, which each represent a callback invocation, will be referred to as *actions* in the following, in order to clearly distinguish them from ROS nodes, which represent individual software components (that might execute actions at specific points in time). The orchestrator contains one callback graph, which gets extended every time the next data input is requested. A data input is any ROS message that is not published by a node within the system under test, but originates from an external source, such as data generated by a simulator or messages from a ROS bag. Completed actions are removed from the

graph. Edges between actions represent dependencies in execution order: An edge (u, v) from action u to action v implies that the action u must be executed after the action v has run to completion. All outgoing edges from an action are created with the action itself. Additional edges are not added at a later time, and edges are only removed once one of the connected actions is removed. It should be noted that time inputs on the `/clock` topic for triggering timer callbacks as described in Section 3.4.2 are not represented as actions, as they do not contain any message data that needs to be buffered. Instead, the appropriate timer callback actions are created as soon as the clock input is offered by the data provider. Once the actions are ready to execute, a corresponding clock message is sent to the node to trigger the callback.

There are four distinct types of edges: `CAUSALITY` edges exist between actions that have an intrinsic data dependency, which for ROS means one action is triggered by an incoming ROS message, which the other action publishes. The ordering of two actions connected by such an edge is guaranteed implicitly since one action is directly triggered by the other. This means the orchestrator does not have to explicitly serialize those callbacks.

`SAME_NODE` edges are inserted between actions that occur at the same ROS node. This guarantees that multiple actions at the same node, such as the callbacks for multiple different subscriptions, occur in the same order for every data input.

`SAME_TOPIC` edges are inserted from an action that publishes a specific topic, to existing actions that are triggered by messages on that topic. This dependency prevents message reordering and subscriber queue overflow, by ensuring that actions that publish on a topic only run after all the actions which are triggered by a previous message on that topic.

`SERVICE_GROUP` edges ensure deterministic execution involving service calls. The result of a service call is considered to be dependent on the state of the service-providing node, and all service calls are assumed to possibly alter that state. Similarly, all other actions occurring directly at the service-providing node are also considered to alter that node's state. A service group for a particular service contains all actions which may call the service and all actions which occur directly at the service provider node. The `SERVICE_GROUP` edge is then added to all nodes in all service groups of the services that a particular action may call. This ensures a deterministic execution order of all actions which can modify the service-providers state.

To illustrate the effects of specific edge types, the scenario from Fig. 3.4 is considered for two subsequent inputs. The resulting callback graph is shown in Fig. 3.11. Actions corresponding to the first input are shown in the left half of the graph. `CAUSALITY` connections drawn in blue show connections directly corresponding to the ROS node graph: They connect each callback to the previous callback publishing the required input data. `SAME_NODE` edges connect the corresponding callbacks between timesteps,

and the two callbacks of node T within each timestep. This ensures that the callback order at T is deterministic even if the processing times of $P1$ and $P2$ are variable. The `SAME_TOPIC` edges in this example might seem redundant to the `SAME_NODE` connections, the outgoing edge from the second data input, however, is required to ensure that both inputs are not reordered before they arrive at the orchestrator. This graph also shows additional nodes which do not directly correspond to callbacks within the software stack under test: The input nodes represent data inputs that may come from a ROS bag or the simulator. *Buffer nodes* represent the action of storing a message at the orchestrator, and allow parallel execution by allowing `SAME_TOPIC` dependencies to be made to specific outputs of callbacks instead of entire callbacks. Some elements have been excluded from this graph for brevity: The callbacks at node T do not have any output, which requires them to publish a status message. The reception of this status message is usually represented in the graph analogous to the buffer nodes.

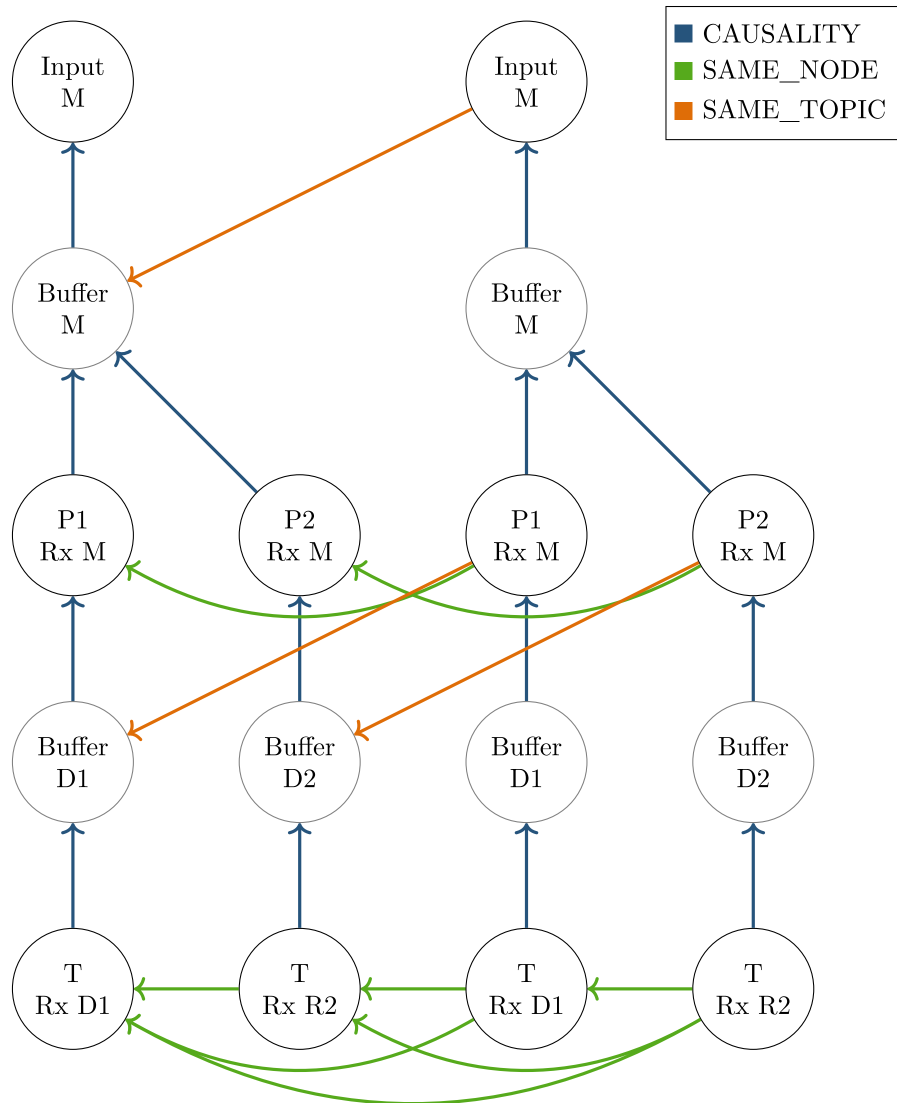


Figure 3.11: Callback graph for two inputs into a ROS graph with two parallel processing paths as shown in Fig. 3.4. "Input" actions represent the publishing of a topic by the data source. "Buffer" actions represent the input of an intercepted topic at the orchestrator, potentially for forwarding to downstream nodes. Message callbacks at ROS nodes are represented as "`<node name>` Rx `<topic>`".

3.6 Node and System Description

In order to build the callback graph, information about the node behavior and system configuration has to be available to the orchestrator. While some aspects of system configuration, such as connections between nodes could be inferred during runtime by using available introspection functionality in ROS, this is not possible for node behavior. Also, since buffering of some topics is necessary, some connections between nodes need to be redirected via the orchestrator, changing the system configuration. This type of system configuration is usually made before starting the nodes and is generally not possible during runtime.

To enable the reuse of node configuration information, the configuration is split into node configuration and launch configuration. Both of those are implemented as static configuration files in JSON format and are described in detail in the following.

3.6.1 Node Configuration

Each node requires a description of its behavior, in particular, which callbacks occur at the node and what the effects of those callbacks are. A node configuration consists of a list of callbacks and a list of provided services:

```
1 {
2   "name": "Trajectory Planning Node",
3   "callbacks": [ ... ],
4   "services": [ ... ]
5 }
```

Each callback specifies its trigger, possible service calls made during execution, its outputs, and flags regarding closed-loop simulation and online reconfiguration (which is described in detail in Section 3.7):

```
1 {
2   "trigger": { ... },
3   "outputs": [ Names of output topics ],
4   "service_calls": [ Names of services which may be called ],
5   "changes_dataprovider_state": false,
6   "may_cause_reconfiguration": false
7 }
```

The trigger specifies a timer, an input topic, or multiple input topics in the case of a message-filter callback:

```
1 { "type": "timer", "period": 40000000 }
```

```
1 { "type": "topic", "name": "imu" }
```

```
{
  "type": "approximate_time_sync",
  "input_topics": ["camera_info", "image"],
  "slop": 0.1,
  "queue_size": 4
}
```

3.6.2 Launch Configuration

The launch configuration describes the entire software stack under test. More specifically, it describes specific instances of nodes and connections between them. Each node is identified by a unique name, and the type of node is specified by reference to the corresponding node configuration file. Connections between nodes are specified using name remappings, which assign a globally unique topic name to the internal names used in the node configuration. In this example, an ego-motion estimation node is instanced for the simulated "vhcl1800" vehicle, receiving the proper sensor data input and providing the "/sil_vhcl1800/ego_motion_service" service:

```
"sil_vhcl1800_ego_motion_service": {
  "config_file": ["orchestrator", "ego_motion_node_config.json"],
  "remappings": {
    "imu": "/sil_vhcl1800/imu",
    "ego_motion_service": "/sil_vhcl1800/ego_motion_service"
  }
}
```

With the corresponding node configuration:

```
{
  "name": "Ego-Motion Service",
  "callbacks": [{
      "trigger": {"type": "topic", "name": "imu"},
      "outputs": []
    }],
  "services": ["ego_motion_service"]
}
```

3.7 Dynamic Reconfiguration

Dynamically reconfiguring components during runtime (see Section 2.2) presents a challenge to the orchestrator, as the software setup is usually specified in advance in the launch configuration file.

To support this use case in combination with the orchestrator, the following assumptions are made with respect to the reconfiguration process:

- The reconfiguration process is initiated by a ROS node during the execution of a callback. It is configured beforehand which callback may cause a reconfiguration.
- Reconfiguration is instant and happens between two data inputs.

In the following, the ROS node which decides when to reconfigure the system is referred to as the "reconfigurator". To ensure deterministic execution before, during, and after reconfiguration, coordination between the reconfigurator and orchestrator is necessary: The orchestrator provides a "reconfiguration announcement" ROS service, which the reconfigurator must call if a reconfiguration is to be performed. The orchestrator then completes the processing of all in-progress and waiting callbacks, without requesting the next data- or time input from the data provider. Once all callbacks are complete, the orchestrator then calls the reconfigurator to execute the reconfiguration. Once complete, the reconfigurator returns the new system configuration to the orchestrator. This process is illustrated in Fig. 3.12.

After loading the new configuration, the orchestrator needs to restart execution. The ROS communication topology might however change significantly during reconfiguration. To ensure that all topics from every node are intercepted and subscribed by the orchestrator, it performs the same initialization as on startup.

At the time of writing, some restrictions exist on the type of reconfiguration actions that may be performed. In particular, creating or changing timers at an existing node, and starting new nodes containing timers is not supported. This is not inherently impossible and would be recommended as a useful extension for dynamic reconfiguration support. Implementation of this feature was omitted however due to the lack of an immediate requirement combined with the high implementation effort due to the implicit nature of triggering timer callbacks by clock inputs and the timer behavior when receiving the first clock input.

3.8 Launch System

The ROS 2 launch system is utilized to perform the initial topic interception via the orchestrator by remapping the corresponding topic names. The orchestrator provides the functionality to automatically generate the list of required remappings from the launch and node configuration files. These remappings map directly from the node-internal name to the intercepted topic name of the format `/intercepted/{node_name}/sub/{topic_name}`. By using node-specific remapping rules of the form `nodename:from:=to`, all remappings can be generated in the same place and then be applied at once, which allows wrapping an existing launch file

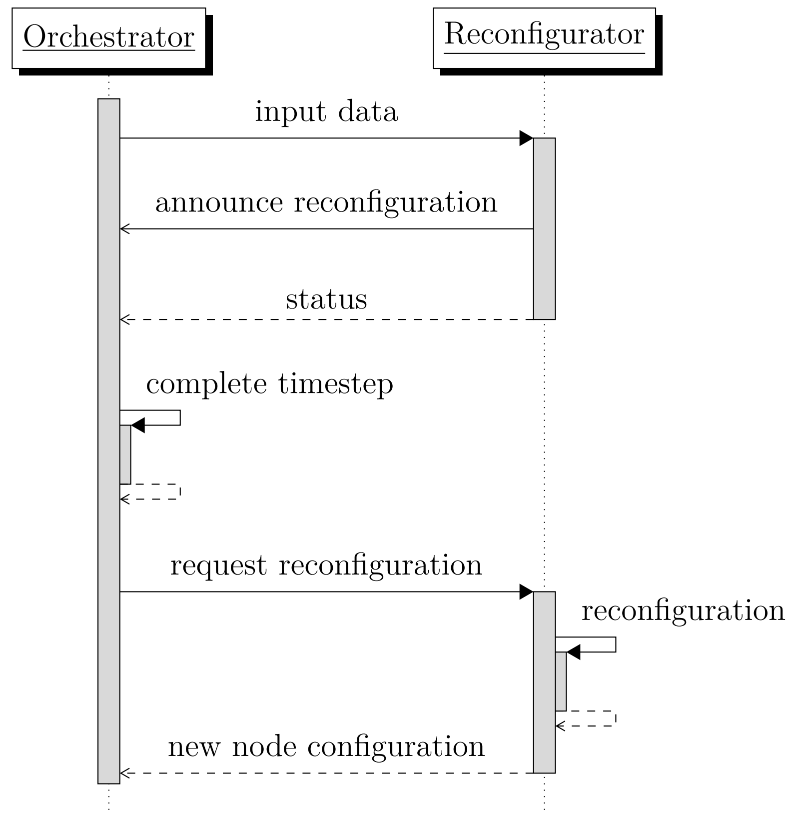


Figure 3.12: Communication between orchestrator and reconfigurator during the dynamic reconfiguration step. The first callback at the reconfigurator is a message callback with the `may_cause_reconfiguration` flag set. The second callback is the execution of the reconfiguration service call.

without making any modifications to it. The following shows an example launch file that starts the software stack under test by first generating the required remappings in line 3 and then including the original launch file below.

```
def generate_launch_description():
    return LaunchDescription([
        *generate_remappings_from_config(
            "orchestrator",
            "sil_reconfig_launch_config.json"
        ),
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource([
                PathJoinSubstitution([
                    FindPackageShare('platform_sil'),
                    'launch/sil.py'
                ])
            ])
        )
    ])
```

A limitation exists with respect to the already existing launch file due to the capabilities of the node-specific remapping in ROS: The `nodename:` prefix which is used to restrict the remapping rule to one specific node, does not accept namespaces in the node name. This might necessitate changing the use of ROS namespaces to prefixes (without a forward slash separator) for node names in the existing launch files. Note that this limitation only applies to node names, and not to topic names.

4 Evaluation

In this chapter, the functionality and applicability of the proposed framework will be evaluated. In Section 4.1, the behavior with and without the orchestrator in the minimal examples presented in Section 3.2 is verified. Section 4.2 then introduces the experimental setup used for further evaluation, which represents a real use case utilizing an existing autonomous-driving software stack. The process of integrating the existing components with the orchestrator is covered in Section 4.3, followed by an evaluation and discussion of using the orchestrator in the presented use case in Section 4.4. In Section 4.5, the impact of using the orchestrator on execution time is explicitly assessed, and approaches for improvement are discussed.

4.1 Verification of Functionality

In order to verify the functionality of the orchestrator, without depending on existing ROS node implementations, individual test cases for specific sources of nondeterminism as well as a combined mockup of an autonomous driving software stack were developed. In the following, each of the examples containing sources of nondeterministic callback sequences identified in Section 3.2 is individually evaluated.

4.1.1 Lost or Reordered Messages

To verify that the problem of lost messages due to overflowing subscriber queues, as introduced in Section 3.2.1, is solved by the orchestrator, a test case was set up: A data source S publishes messages at a fixed frequency. The messages are received by the node under test P, which has a fixed queue size (of three messages in this example), and a varying processing time that on average is significantly slower than the period of message publishing. After processing, P publishes the result on a different topic. This behavior might correspond to a simulator running at a fixed frequency, and a computationally expensive processing component such as a perception module, running on a resource-constrained system.

Figure 4.1 shows the sequence of events when running this test: The first timeline shows the periodic publishing of input messages by S. The second timeline shows the callback duration of node P. It can be seen that once the processing of the

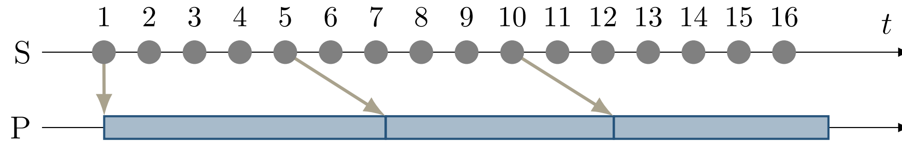


Figure 4.1: Sequence diagram showing dropped messages due to subscriber queue overflow, with a subscriber queue size of 3 at P. The corresponding ROS graph is shown in Fig. 3.2.

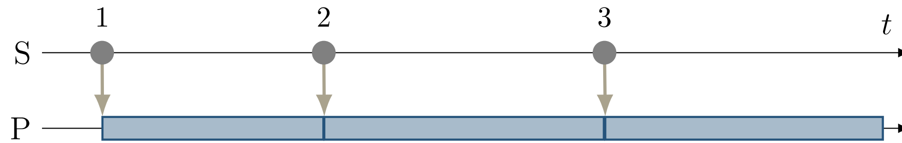


Figure 4.2: Sequence diagram showing a slowdown of the data source to prevent dropping messages by overflowing the subscriber queue.

first message finishes, processing immediately continues for message 5, which is the third-recent message published at that point in time, skipping messages 2, 3, and 4 which were published during processing. During the processing of message 5, four further messages are discarded. The exact number of skipped messages depends on the callback duration, which in this case is deliberately randomized but is usually highly dependent on external factors such as system load.

When using the orchestrator, the message publisher is still configured to the same publishing rate, but waits for the orchestrator before publishing each message. Figure 4.2 shows that each message is now processed, regardless of callback duration. This necessarily slows down the data source, which can not be avoided without risking dropping messages from the subscription queue at the receiving node.

By only sending messages to a node once the processing of the previous message is completed, reordering of messages by the middleware is also prevented. This is not explicitly demonstrated here but follows immediately from the fact that only one message per topic is being transmitted at any point in time.

4.1.2 Inputs From Parallel Processing Chains

To verify deterministic callback execution at a node with multiple parallel inputs, the example introduced in Section 3.2.2 with the ROS graph shown in Fig. 3.4 is realized. Figure 4.3 shows all callback invocations resulting from two inputs from S. Without the orchestrator, the combination of nondeterministic transmission latency and variable duration of callback execution at $P1$ and $P2$ results in a nondeterministic execution order of both callbacks at T resulting from one input from S.

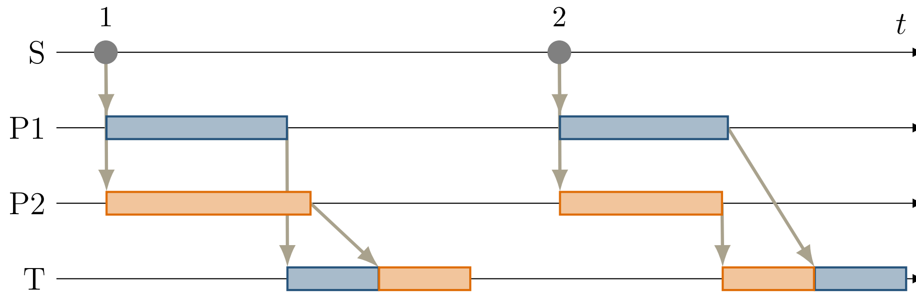


Figure 4.3: Sequence diagram showing the execution of two parallel processing nodes $P1$ and $P2$ with nondeterministic processing time. This results in a nondeterministic callback order at T, which subscribes to the outputs of both chains. The corresponding ROS graph is shown in Fig. 3.4.

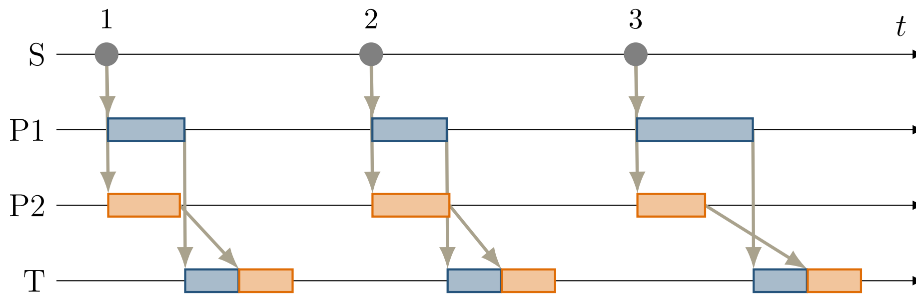


Figure 4.4: Sequence diagram showing a deterministic callback order at T despite nondeterministic callback durations at $P1$ and $P2$ as an effect of the orchestrator on the behavior shown in Fig. 4.3.

For input 1, $P1$ finishes processing before $P2$, and no significant transmission latency occurs, which causes T to process the message on $D1$ before $D2$. Following input 2, $P2$ is slightly faster than $P1$ resulting in a different callback order compared to the first input.

Using the orchestrator, the callback order changes, as visualized in Fig. 4.4. For the first and third data input, $P1$ requires more processing time than $P2$. This would ordinarily allow the $D2$ callback at T to execute before the $D1$ callback. The orchestrator however ensures a deterministic callback order at T for every data input from S, by buffering the $D2$ message until T finishes processing $D1$. Note that the orchestrator does not implement a specific callback order defined by the node or externally. It only ensures that the order is consistent over multiple executions. The actual order results from the order in which nodes and callbacks are listed in configuration files, but this is not intended to be adjusted by the user. If a node requires a distinct receive order, it must implement appropriate ordering internally, to

ensure correct operation without the orchestrator. From the point of the orchestrator, consistently ordering $P2$ before $P1$ would have also been a valid solution.

4.1.3 Multiple Publishers on the Same Topic

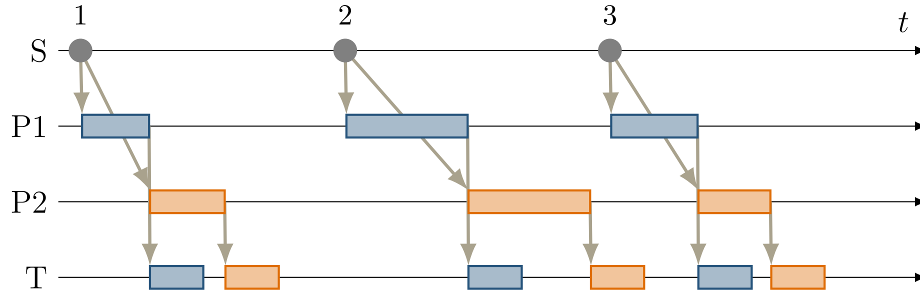


Figure 4.5: Sequence diagram showing serialized callback executions of nodes $P1$ and $P2$, which is required to achieve a deterministic callback order at T in this example, since $P1$ and $P2$ use the same output topic. The corresponding ROS graph is shown in Fig. 3.5.

This example extends the previous scenario from Section 4.1.2 such that both processing nodes publish their result on the same topic, corresponding to the example introduced in Section 3.2.3, with the ROS graph shown in Fig. 3.5. Again, this results in nondeterministic callback order at T, with a callback order identical to the previous case shown in Fig. 4.3. In this case, both callback executions at T are of the same callback, while previously two distinct callbacks were executed once each.

Because only node *inputs* are intercepted, this scenario requires serializing the callbacks at $P1$ and $P2$. Figure 4.5 shows the resulting callback sequence when using the orchestrator. By ensuring that processing at $P2$ only starts after the output from $P1$ is received, reordering of the messages on D is prevented. Note that while the different colors of the callbacks at T correspond to the sources of the corresponding input, both inputs cause the same subscription callback to be executed at the node. Generally, the node would not be able to determine the source of the input message.

Since the processing time of $P2$ is longer than the processing time of the first callback at T in this example, the orchestrator causes a larger overhead for this node graph compared to the previous one. $P2$ starts processing simultaneously to the first T callback, causing T to be idle between the completion of the first callback and the completion of processing at $P2$. It should be noted, however, that even though the total processing time exceeds the input frequency of S for input 2, the data source was not required to slow down. Figure 4.5 shows that T is still running while $P1$ processes input 3. This kind of "pipelining" happens implicitly because the callback execution

at $P1$ has no dependency on the callback at T, and by eagerly allowing inputs from S. In the current implementation, the orchestrator requests the publishing of the next message by the data provider as soon as the processing of the last input on the same topic has started. In the case of a time input, the input is requested as soon as no actions remain which are still waiting on an input of a previous time update. Both kinds of input may additionally be delayed if the system is pending dynamic reconfiguration, or if a callback is still running that may cause a reconfiguration at the end of the current timestep.

4.1.4 Parallel Service Calls

Figure 3.6 shows the node setup for this example, which has been identified in Section 3.2.4. A single message triggers a callback at three nodes, one of which (SP) also provides a ROS service. The two other nodes $N1$ and $N2$ call the provided service during callback execution. The resulting order of all three callbacks at SP in response to a single message input is nondeterministic, as shown in Fig. 4.6. Since the orchestrator only controls service calls by controlling the callback they originate from, it is necessary to serialize all callbacks interacting with the service, which in this case are the message callbacks at $N1$, $N2$, and SP.

The resulting callback sequence is shown in Fig. 4.7. By serializing the callbacks at $N1$ and $N2$, the order of service callbacks at SP is now fixed. In this example, it is again apparent that parallel execution of the $N1$ and $N2$ callbacks might be possible while still maintaining a deterministic callback order at SP. This limitation is discussed in detail in Section 4.1.5.

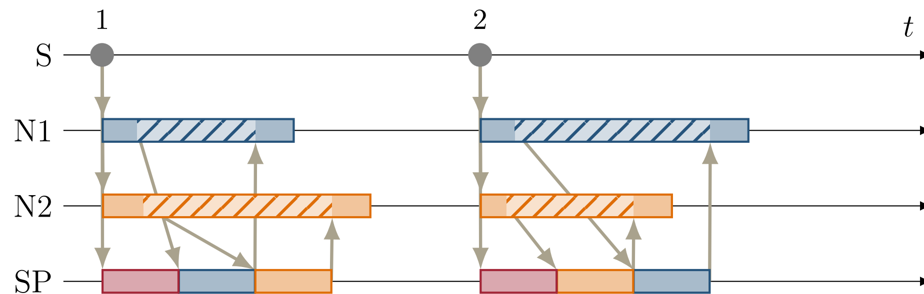


Figure 4.6: Sequence diagram showing the parallel execution of callbacks at $N1$ and $N2$. The hatched area within the callback shows the duration of service calls, which are made to a service provided by SP, upwards arrows represent responses to service calls. The variable timing of the service calls results in a nondeterministic callback order at SP. The corresponding ROS graph is shown in Fig. 3.6.

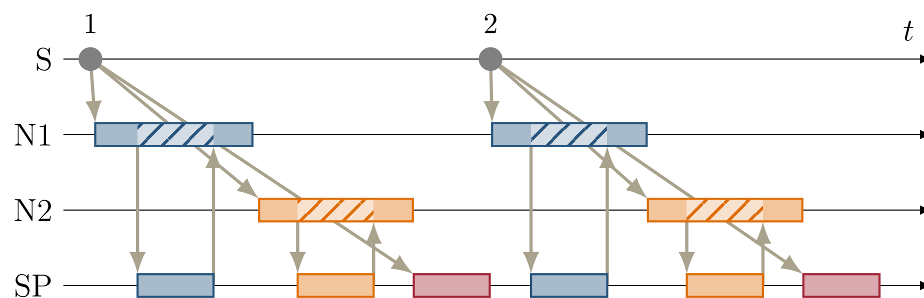


Figure 4.7: Sequence diagram showing the serialized callbacks from Fig. 4.6. Serialization of the callbacks at $N1$ and $N2$ leads to a deterministic callback order at SP.

4.1.5 Discussion

The ability of the orchestrator to ensure a deterministic callback sequence at all nodes has been shown for the minimal nondeterministic examples which were identified in Section 3.2. While all examples show successful deterministic execution, some limitations and possible improvements in parallel callback execution and thereby execution time are apparent and will be discussed in the following.

In the case of concurrent callbacks which publish on the same topic, parallelism could further be improved by extending the topic interception strategy. Currently, only the input topics of each node are intercepted by the orchestrator, the output topics are not changed. If the output topics of nodes were also remapped to individual topics, all `SAME_TOPIC` dependencies would be eliminated. In the example from Fig. 4.4, this would again allow the concurrent callbacks $P1$ and $P2$ to execute in parallel, with each output being individually buffered at the orchestrator. The individually and uniquely buffered outputs could then be forwarded to T in a deterministic order, effectively resulting in a callback execution behavior as in Section 4.1.2.

The last example of concurrent service calls (Section 4.1.4) also shows how this method of ensuring deterministic execution comes with a significant runtime penalty. Here, the orchestrator now requires all callbacks to execute sequentially, while previously all callbacks started executing in parallel, with the only point of synchronization being the service provider, depending on available parallel callback execution within the node. An important factor determining the impact of this is the proportion of service-call duration to total callback duration for the calling nodes. If the service call is expected to take only a small fraction of the entire callback duration, a large improvement in execution time could be gained by allowing parallel execution of the callbacks $N1$ and $N2$, which both call the service. This might be possible by explicitly controlling service calls directly instead of controlling the entire callback executing that call. In the example shown in Fig. 4.7, serializing only the service calls would allow the portion of the $N2$ callback before the service call to execute concurrently to $N1$, and the portion after the service call to overlap with the message callback at SP.

Another possible extension to improve parallelism in scenarios involving service calls is to allow specifying that some actions might interact with the service provider without modifying its state. Currently, all actions interacting with the service (by running at the same node, or calling the service) are assumed to modify the service provider state. To ensure deterministic execution, synchronization between non-modifying actions is however not required. If an action only inspects the service providers' state without modifying it, the order with respect to other such actions would not influence its result. Thus, it would suffice to synchronize non-modifying actions with previous modifying actions, instead of all previous actions.

In Section 4.1.2, it was identified that although the callback order at each node is not deterministic, a different order of callbacks in response to a single input might be expected during normal operation. This does not reduce the applicability of the orchestrator, since nodes that explicitly require a specific callback order must implement measures to ensure that anyways. It is however still desirable to keep the system behavior when using the orchestrator as close as possible to the expected or usual system behavior without the orchestrator. One proposed future addition is thus allowing nodes to optionally specify an expected callback duration in the corresponding configuration file. This information may then be used by the orchestrator to establish a more realistic callback ordering.

4.2 System Setup

In the following, the integration of the orchestrator with parts of an already existing autonomous driving software stack is evaluated. This section introduces the system setup and example use case, which will be utilized in Sections 4.3 and 4.4.

In this use case, the aim is to calculate metrics on the performance of a multi-object tracking module, which tracks vehicles that pass an intersection using infrastructure-mounted sensors. The ROS graph of the setup is shown in Fig. 4.8. The software stack consists of this tracking module, as well as components required to autonomously control one of the vehicles passing the intersection in the test scenario. A simulator provides measurements in the form of (possibly incomplete) bounding boxes and object class estimations, simulating both the sensor itself as well as an object detection algorithm. Alternatively, the same measurements are played back from a ROS bag. The tracking module receives measurements on a total of 12 individual topics for each sensor. Outputs from the tracking module, as well as ground truth object states provided by the simulator, are recorded by dedicated recorder nodes. This allows later post-processing and evaluation.

The part of the software stack controlling the autonomous vehicle consists of a second instance of the tracking module, a component estimating the vehicle's ego-motion as well as a trajectory planning and control module. The vehicle-local tracking module receives measurements from five simulated on-vehicle sensors similar to the infrastructure tracking module. The planning module receives information about the vehicle state from the simulator and produces acceleration and steering angle commands which are fed back to the simulator. Both the planning and local tracking modules may call the ego-motion service provided by the corresponding node while executing any callback. The other vehicles present in the scenario are fully controlled by the simulator.

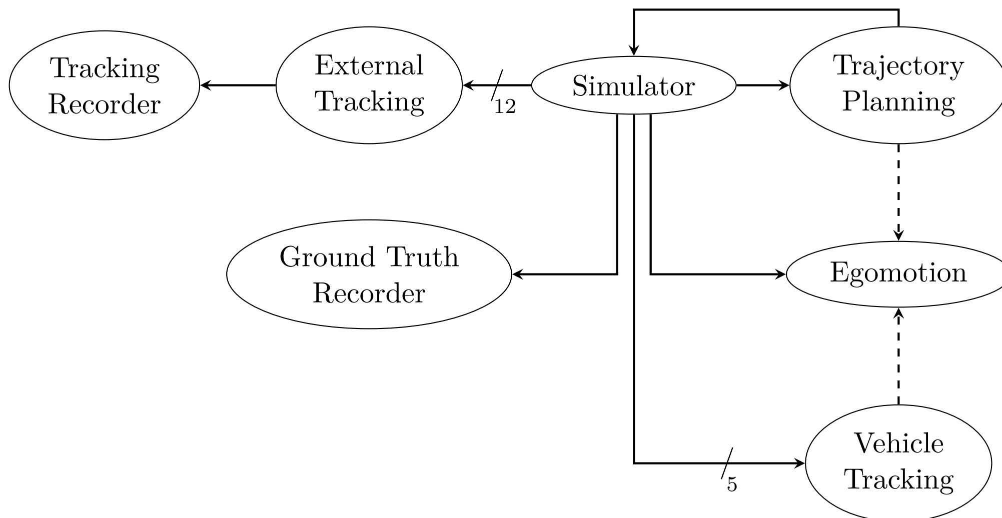


Figure 4.8: Node graph of the system setup used within this chapter. The connections between the simulator and both tracking nodes represent multiple parallel ROS topics. Dashed arrows show potential service calls.

The simulation is run until the controlled vehicle reaches a predefined area. When using recorded measurement data from a ROS bag, the scenario ends once every recorded measurement has been processed. The recorded results of the tracking module and the recorded ground truth data are then used to calculate application-specific metrics to assess the performance of the multi-object tracking algorithm.

4.3 System Integration

To determine the feasibility of integrating the proposed framework into existing software, the framework was applied to the scenario for testing a multi-object tracking module introduced in Section 4.2. In this section, the necessary modifications to each existing component are discussed. Sections 4.3.1 and 4.3.2 will cover the integration of both "data provider" components, a simulator, and the ROS bag player, which will contain the orchestrator. Section 4.3.3 covers the integration of the ROS nodes present in the test scenario.

4.3.1 Simulator

The orchestrator represents an individual component (see Section 3.4), but is located within the same process as the data provider, which in this case is the simulator.

The orchestrator component is instantiated within the simulator and then provides an API that the simulator must call at specific points to ensure deterministic execution. To instantiate and start the orchestrator, the simulator must also provide the orchestrator with the appropriate launch configuration. All API calls are of the form `wait_until_<condition>` and usually return a `Future` object that must be awaited before executing the corresponding actions. The `wait_until_publish_allowed` function must be inserted before publishing any ROS message on any topic. Before publishing a `/clock` message, the new time must be provided to the orchestrator using the dedicated `wait_until_time_publish_allowed` API call, which is required for the orchestrator to prepare for eventual timer callbacks. Before changing the internal simulation state, the `wait_until_dataprovider_state_update_allowed` method must be called. This usually happens by performing a simulation timestep, and this method ensures synchronizing this timestep with expected inputs present in a closed-loop simulation, such as vehicle control inputs. The `wait_until_pending_actions_complete` method is used to ensure all callbacks finish cleanly once the simulation is done.

To enable closed-loop simulation, the simulator must accept some input from the software under test, such as a control signal for an autonomous vehicle in this case. This implies a subscription callback, which must be described in a node configuration file. If this callback does not publish any further messages, a status message must be published instead.

4.3.2 ROS Bag Player

ROS already provides a ROS bag player, which could be modified to include the orchestrator. Modifying the official ROS bag player would have the advantage of keeping access to the large set of features already implemented, and preserving the known user interface. Some aspects of the official player increase the integration effort considerably, however. Specifically, publishing of the `/clock` topic is asynchronous to message playback and at a fixed rate. While this has some advantages for interactive use, it interferes with deterministic execution and would require a significant change in design to accommodate the orchestrator. Furthermore, as with the initial architecture considerations of the orchestrator, it is undesirable to fork existing ROS components and maintain alternative versions, as this creates an additional maintenance burden and might prevent the easy adoption of new upstream features.

Thus, a dedicated ROS bag player is implemented for use with the orchestrator instead of modifying the existing player. This does not have the same feature set as the official

player but allows for evaluation of this use case with a reasonable implementation effort. To integrate the orchestrator, the ROS bag player requires the same adaptation as the simulator, except for the `wait_until_dataprovider_state_update_allowed` call which is not applicable without closed-loop execution. Besides deterministic execution, a new feature is reliable faster-than-realtime execution, details of which are discussed in Section 4.5.

4.3.3 ROS Nodes

The individual ROS nodes of the software stack under test are the primary concern regarding implementation effort, as there is usually a large number of ROS nodes, and new ROS nodes may be created or integrated regularly.

The integration effort of a ROS node depends on how well the node already matches the assumptions made and required by the orchestrator: The orchestrator assumes that all processing in a node happens in a subscription or timer callback, and that each callback publishes at most one message on each configured output topic. For callbacks without any outputs or callbacks that sporadically omit outputs, a status message must be published instead (see Section 3.4.1).

Planning Module

The integration effort of the trajectory planning and control module is significant because the module violates the assumption that all processing happens in timer and subscription callbacks.

The planning module contains two planning loops: A high-level planning step runs in a dedicated thread as often as possible. A low-level planner runs separately at a fixed frequency. Handling incoming ROS messages happens asynchronously with the planning steps in a third thread.

While this architecture may have some advantages for runtime performance, it prevents external control via the orchestrator. This represents an inherent limitation for the orchestrator. Publishing of messages from outside a ROS callback is not able to be supported in any way, since it can not be anticipated in advance, making it impossible to integrate into the callback graph and synchronize it with other callbacks (see Section 3.5). In order to ensure compatibility with the orchestrator, an optional mode has been introduced in which both planning loops are replaced with ROS timers.

This does make the planning module compatible with the orchestrator, but introduces a problem that should have explicitly been avoided by the specific software architecture chosen: It runs the planning module in a completely different mode when using the orchestrator than without using the orchestrator. This reduces the relevance of testing

inside the orchestrator framework since specific problems and behaviors might only occur with the manual planning loop.

It might be possible in some cases to change the node in a way such that the usual mode of execution is compatible with the orchestrator, and thus avoids the problem of two discrete modes, but this is not possible in general. In the case of the trajectory planning module, for example, this is not desirable due to the integration of the planning loop with a graphical user interface that is used to interactively change planner parameters and to introspect the current planner state.

Tracking Module

While the tracking module does only process data within ROS subscription callbacks, the input-output behavior is still not straightforward: The tracking module employs a sophisticated queueing system, which aims to form batches of inputs from both synchronized and unsynchronized sensors, while also supporting dynamic addition and removal of sensors. Additionally, while processing is always triggered by an incoming message, the processing itself happens in a dedicated thread in order to allow the simultaneous processing of ROS messages.

The input-output behavior itself is configurable such that only the reception of specific sensor inputs cause the processing and publishing of a "`tracks`" output message. This is done to limit the output rate and reduce processing requirements. Due to the queueing, this does however not imply that reception of the configured input immediately causes an output to appear. It may be the case that additional inputs are required to produce the expected output.

This behavior can however still be handled by the node configuration without requiring major modification to the tracking module: The node configuration was modified such that any input may cause an output to be published. Then, the processing method was adapted such that a status message is published that explicitly excludes the `tracks` output using the `omitted_outputs` field when no tracks will be published. In some circumstances, specifically following dropped messages, the queueing additionally results in multiple outputs in a single callback. This behavior is described in detail in Section 4.4.2 and is not currently supported by the orchestrator.

While this is a pragmatic solution for describing the otherwise hard to statically describe input-output behavior of the tracking module, declaring more output topics than necessary for a callback is usually undesired: Subsequent callbacks which actually publish a message on the specified topic need to wait for this callback to complete due to a false `SAME_TOPIC` dependency. Additionally, the callback graph will contain possibly many actions resulting from the anticipated output. Those actions are then again false dependencies for subsequent actions, not only as `SAME_TOPIC` dependencies

but also SAME_NODE and SERVICE_GROUP edges. These false dependencies might reduce the number of callbacks able to execute in parallel and might force callback executions to be delayed more than necessary to ensure deterministic execution. Once a status message is received which specifies that the output message will not be published, the additional actions are removed, which then allows the execution of dependent actions.

Recorder Node and Ego-Motion Estimation

Both the nodes for recording the output of the tracking module and the ego-motion estimation match the assumptions made by the orchestrator and require very little integration effort, although some modification was necessary. Both nodes only have topic input callbacks that would usually not cause any message to be published, requiring the publishing of a status message to inform the orchestrator of callback completion.

The ego-motion module is the only node in the experimental setup offering a service used during the evaluation. This does however not require any modification within the node, as service calls are controlled by controlling the originating callbacks. It is required however to list the service in the node configuration, to ensure a deterministic order between service calls and topic-input callbacks at the node.

4.3.4 Discussion

In Section 3.3, the design goals towards the integration of existing nodes were established as minimizing the required modification to nodes, maintaining functionality without the orchestrator, and allowing for external nodes to be integrated without modifying their source code.

The implemented approach meets these goals to varying degrees. The integration of existing components with the orchestrator requires a varying amount of effort, depending primarily on how well the component matches assumptions made by the orchestrator. ROS nodes that fully comply with the assumptions made by the orchestrator and always publish every configured output require only a configuration file describing the node's behavior, which also works for external nodes without access to or modification of their source code. Nodes that have callbacks without any output and nodes that may omit some or all configured outputs in some callback executions require publishing a status output as described in Section 3.4.1 after a callback is complete. Since this only entails publishing an additional message, this modification does not impede the node's functionality in any way when not using the orchestrator. Nodes that fully deviate from the assumed callback behavior require appropriate

modification before being suitable for use with the orchestrator, as was illustrated with the tracking and planning modules in Section 4.3.3.

Creating the node configuration file does not present a significant effort for initial integration, but maintaining the configuration to match the actual node behavior is essential. Although the orchestrator can detect some mismatches between node behavior and description, omitted outputs and services can not be controlled by the orchestrator and might lead to nondeterministic system behavior.

While the model of ROS nodes that only execute ROS callbacks, which then publish at most one message on each configured output topic, is clearly not sufficient for all existing ROS nodes, it does apply to a wide class of nodes in use. Nodes such as detection modules and control algorithms often operate in a simple “one output for each input” way or are completely time triggered, executing the same callback at a fixed frequency. Such nodes are not part of this experimental setup, since the specific simulator in use already integrates the detection modules.

4.4 Application to existing Scenario

In this section, the effect of using the orchestrator in the use case introduced in Section 4.2 is evaluated. In the following, the ability of the orchestrator to ensure deterministic execution up to the metric-calculation step is demonstrated using both the simulator and recorded input data from a ROS bag, as well as combined with dynamic reconfiguration during test execution.

4.4.1 Simulator

When evaluating the tracking module in the previously introduced scenario, the MOTA and MOTP metrics introduced in Section 2.3.1 are calculated. To calculate these metrics, the tracking outputs are recorded together with ground truth data from the simulator during a simulation run. Those recordings are then loaded and processed offline. When running the evaluation procedure multiple times, it can be observed that the resulting values differ for each run, as shown in Fig. 4.9. This is due to nondeterministic callback execution during evaluation: Both the simulator and the trajectory planning module run independently of each other, and the callback sequence of the multiple inputs to the tracking module is not fixed.

When running the simulation using the orchestrator, the variance in the calculated metrics is eliminated. This shows that in this example the orchestrator successfully enabled the use case of repeatable execution of test cases for evaluating a software module inside a more complex system.

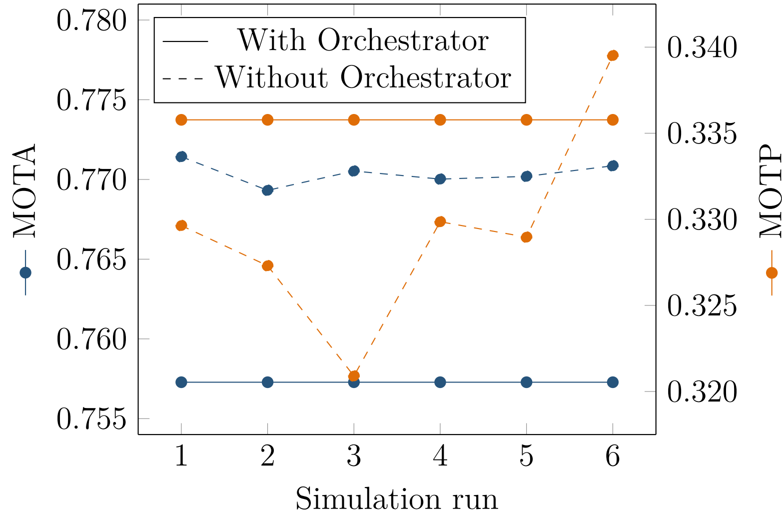


Figure 4.9: Evaluation of the MOTA and MOTP metrics in the scenario introduced in Section 4.2 over multiple simulation runs, both with and without the orchestrator.

Not only are the calculated metrics consistent, the deterministic execution as ensured by the orchestrator results in bit-identical outputs of the tracking module for every simulation run, and thus exact equality of the recordings generated. This enables additional use cases for testing such as easily comparing the output of the module before and after presumably non-functional changes are made to the source code. Previously, such a comparison would require parsing the recorded results, calculating some similarity measure or distance between the expected and actual results, and applying some threshold to determine equality. Now, simply comparing the files without any semantic understanding of the contents is possible.

4.4.2 ROS Bag

In order to test the use case of ROS bag replay, the player implemented in Section 4.3.2 is used. Although the ROS bag player provides inputs in deterministic order, the characteristics of the input data are different from the simulator. During the recording of the ROS bag, the sensor input topics and pre-processing nodes are subject to nondeterministic ROS communication and callback behavior. This results in a ROS bag with missing sensor samples (due to dropped messages as well as unexpected behavior of real sensors) and reordered messages (due to nondeterministic transmission of the messages to the ROS bag recorder). All those effects would usually not be expected from a simulator, which produces predictable and periodic inputs.

This does not present a problem for the orchestrator: Since the callback graph

construction is incremental for each input, the only a priori knowledge the orchestrator requires is the API call from the data provider informing the orchestrator of the next input, and the node and launch configurations to determine the resulting callbacks. Specifically, the orchestrator does not require information such as expected publishing frequencies or periodically repeating inputs at all.

In order to reuse the existing test setup, a ROS bag was recorded from the outputs of the simulator. To simulate the effects described above, the ROS bag is manually modified by randomly dropping messages and randomly reordering recorded messages.

Using the multi-object tracking module was not possible, however, since the high rate of dropped messages causes a callback behavior that can not be modeled by the node configuration as introduced in Section 3.6.1. In addition to the behavior described in Section 4.3.3 of zero or one output for each measurement input, certain combinations of inputs may cause multiple outputs from one input callback. This is due to a sophisticated input queueing approach, that forms batches of inputs with small deviations in measurement time, that only get processed once a batch contains measurements of all sensors. In case of missing measurements, a newer batch might be complete while older, incomplete batches still exist. The queueing algorithm assumes in that case that the missing measurements of the old batches will not arrive anymore (ruling out message reordering, but allowing dropping messages), and processes the old batches, producing multiple outputs in one callback. Handling more outputs than expected is not possible for the orchestrator since the orchestrator must determine when a callback is completed to allow the next input for the corresponding node. If a callback publishes additional outputs after it is assumed to have been completed already, the orchestrator can not identify the source of the additional output or wrongly assigns the output to the next callback expected to publish on the corresponding topic.

This queueing also makes the tracking module robust against any message reordering between the ROS bag player and the module itself, resulting in deterministic execution even without the orchestrator and at high playback speed. When using a ROS bag with reordered, but without dropped messages, the experimental setup can be verified and performs as expected with a ROS bag as the data source instead of a simulator, which also shows that the orchestrator can successfully be used in combination with existing node-specific measures to ensure deterministic input ordering. The further behavior of the orchestrator remains unchanged, meaning nondeterminism in larger systems under test such as the cases demonstrated in Section 4.1 is prevented.

Furthermore, when using ROS bags as the data source it may be possible to easily maximize the playback speed without manually choosing a rate that does not overwhelm the processing components causing dropped messages. More details on this specific use case will be given in Section 4.5.

```
{
  "name": "sil_reconfigurator",
  "callbacks": [
    {
      "trigger": {
        "type": "timer",
        "period": 1000000000
      },
      "outputs": [],
      "may_cause_reconfiguration": true
    }
  ]
}
```

Listing 2: Node configuration for the reconfiguration node mockup.

4.4.3 Dynamic Reconfiguration

To test the orchestrator in a scenario including dynamic reconfiguration, the previous setup was extended by such a component. Since a module for dynamic reconfiguration of components or the communication structure was not readily available, a minimal functional mockup was created: A "reconfigurator" component with a periodic timer callback decides within this callback if the system needs to be reconfigured, and then executes that reconfiguration. The node description for the reconfiguration node is given in Listing 2. In this example, the reconfiguration reduces simulated measurement noise, which could simulate switching to a more accurate, but also more computationally demanding perception module. The mock reconfigurator always chooses to reconfigure after a set time. A real working counterpart would require additional inputs such as the current vehicle environment, which are omitted here.

Figure 4.10 shows the OSPA distance (see Section 2.3.1) between the tracking result and the ground truth object data from the simulator over multiple simulation runs. The OSPA distance was chosen as a metric in this case since it is calculated for every time step instead of as an average over the entire simulation run, as is the case with the MOTA and MOTP metrics used above. This allows evaluation of how the metric changes during the simulation run and clearly shows the reconfiguration step. It is apparent that the reconfiguration module successfully switched to a lower measurement noise at $t = 7s$. Importantly, however, the evaluation results of the multiple runs do not completely overlap. This is again due to nondeterministic callback execution within the tracking, planning, and simulator modules. The differences between the runs, plotted in Fig. 4.11, show that all runs deviate from the first run, with two runs showing the largest difference at the exact time of reconfiguration.

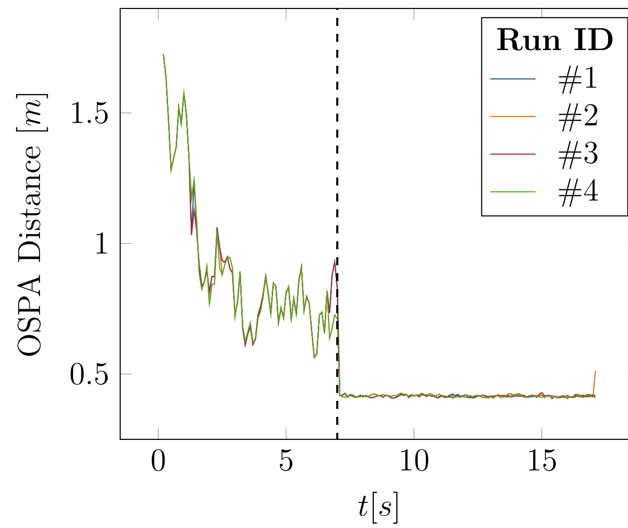


Figure 4.10: OSPA distance of tracks versus ground truth during multiple simulation runs. The dashed vertical line marks the timestep in which the runtime reconfiguration occurs.

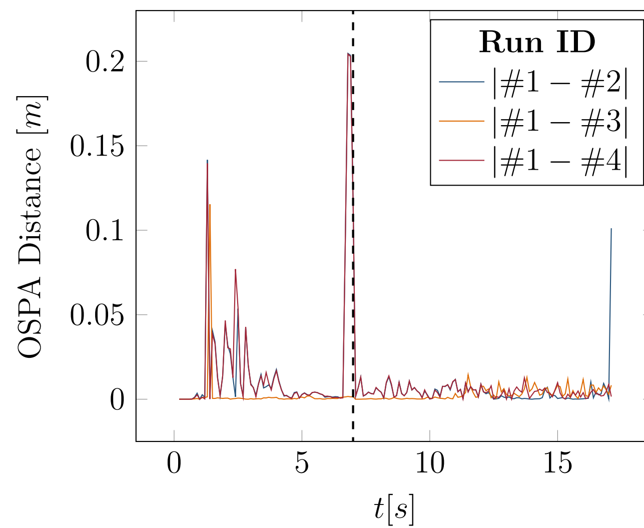


Figure 4.11: Absolute difference in OSPA distances between the simulation runs. The dashed vertical line marks the timestep in which the runtime reconfiguration occurs.

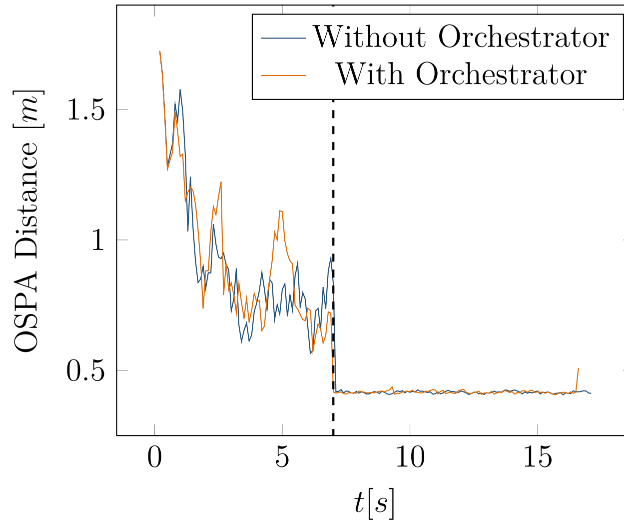


Figure 4.12: OSPA distance of tracks versus ground truth over time, comparison between initial simulation run and simulation while using the orchestrator.

Using the orchestrator, the measured tracking result does differ from the previous simulation runs, as shown in Fig. 4.12. The output is however deterministic and repeatable, even if a reconfiguration occurs during the simulation. Again, this demonstrates the successful application of the orchestrator framework, even in the presence of dynamic reconfiguration at runtime.

4.4.4 Discussion

In Section 4.4, the successful implementation of two design goals was verified: First, Sections 4.4.1 and 4.4.2 demonstrate successful use of the orchestrator with both a simulator and ROS bag as data sources. Notably, no additional requirements are placed on the specific ROS bag used, allowing the use of the orchestrator with already existing recorded data. Secondly, Section 4.4.3 shows that the guarantees of the orchestrator hold when the system is dynamically reconfigured at runtime. These tests represent exactly the use case of evaluation of a component within a larger software stack that motivated this work, that is able to run repeatedly and deterministically using the orchestrator.

In Section 4.4.2, a limitation of the orchestrator in terms of modeling a node's output behavior was reached. In order to use such nodes with the orchestrator in the future, an extension to the current callback handling might be required and is proposed here: A solution to this problem might be to allow the node to publish a status message after every callback, which specifies the number of outputs that have actually been

published in this specific callback invocation. This would allow the orchestrator to ensure the reception of every callback output, and prevent wrong associations of outputs to callbacks. As additional messages on the corresponding topics would also cause additional downstream callbacks for subscribers of those topics, this approach might however introduce additional points of synchronization across the callback graph.

4.5 Execution-Time impact

Due to the required serialization of callbacks and buffering of messages, a general increase in execution time is to be expected when using the orchestrator. In the following, this impact is measured for a simulation use case and the individual sources of increased execution time, as well as possible future improvements, are discussed.

4.5.1 Analysis

To measure the impact of topic interception, the induced delay of forwarding a message via a ROS node is measured. In order to compensate for latency in the measuring node, the difference in latency for directly sending and receiving a message in the same node versus the latency of sending a message and receiving a forwarded message is measured. When using a measuring and forwarding node implemented in Python and using the "eProsima Fast DDS" middleware, the latency from publishing to receiving increases from a mean of 0.64 ms to 0.99 ms. This induced latency of 0.35 ms on average is considered acceptable and justifies the design choice of controlling callbacks by intercepting the corresponding message inputs.

Figure 4.13 shows a comparison of execution time for one simulation run of the scenario introduced in Section 4.2. The first bar shows the runtime without using the orchestrator, the bottom two bars show the time when using the orchestrator.

The simulator currently offers two modes of execution: `fast` executes the simulation as fast as possible, while `real_time` slows down the simulation to run at real-time speed if the simulation itself would be able to run faster than real-time. Using the `fast` mode is only appropriate combined with the orchestrator or some other method of synchronization between the simulator and software under test. If the simulator is not able to run in real-time, deliberate delays to ensure real-time execution should already be zero. Since Fig. 4.13 still shows an increase in runtime for using the `real_time` mode compared to the `fast` mode, the orchestrator is considered with the `fast` execution mode in the following. Nonetheless, it is apparent that the orchestrator

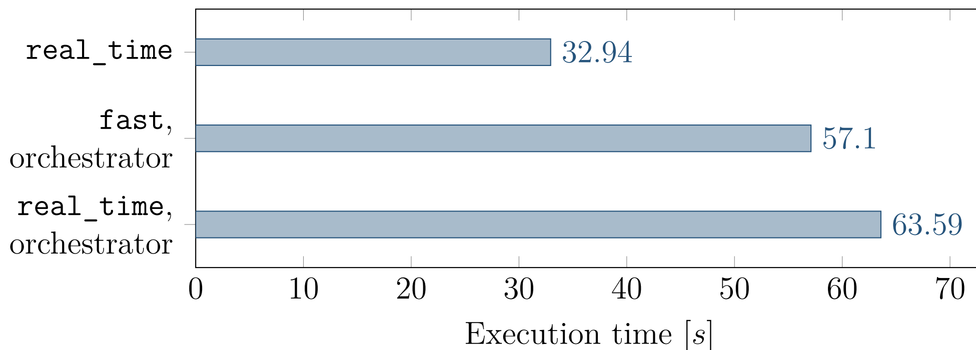


Figure 4.13: Comparison of execution time for one simulation run between not using the orchestrator, using the orchestrator with faster than real-time execution, and using the orchestrator with real-time execution.

causes a significant runtime impact as the execution time is increased by about 73% in the `fast` case.

Evaluating the orchestrator itself for execution time, it can be found that during a simulation run, the callback for intercepted message inputs runs on average 0.6 ms, and the callback for status messages runs 0.9 ms. The API functions for waiting until publishing a time or data input execute in 0.9 ms and 0.5 ms. This sums up to more than 12.3 seconds spent executing interception and status callbacks, which in this scenario happens within the simulator. The simulator furthermore spends about 5 seconds executing orchestrator API calls.

The remaining increase in execution time is explained by serializing the execution of dependent callbacks. The vehicle tracking and planning components may both call the ego-motion service, which prevents parallel execution. The speed of publishing inputs by the simulator is greatly reduced especially for nodes like the tracking module, which has a relatively large number of inputs (12, in the evaluated examples) that are published sequentially. This would usually happen without waiting, but the orchestrator requires confirmation from the tracking module that an input has been processed before forwarding the next input to ensure a deterministic processing order.

Finally, the orchestrator requires the simulator to receive and process the output from the planning module before advancing the simulation. This is realized by the `changes_dataprovider_state` flag for the corresponding callback in the node configuration file, which causes the `wait_until_dataprovider_state_update_allowed` API call to block until the callback has finished. For any simulator, the "dataprovider state update" corresponds to executing a simulation timestep, which results in an effective slowdown of each simulation timestep to the execution time of the longest

path resulting in some input to the simulator.

The other available flag for callbacks, `may_cause_reconfiguration`, presents a similar point of global synchronization: This flag is applied to callbacks of a component that may decide dynamically reconfigure the ROS system, as described in Section 2.2, based on the current system state (such as vehicle environment, in the autonomous driving use case). To ensure that the reconfiguration always occurs at the same point in time with respect to other callback executions at each node, any subsequent data inputs and dataprovider state updates must wait until either the reconfiguration is complete or the callback has finished without requesting reconfiguration. This presents an even more severe point of synchronization, since it immediately blocks the next data inputs from the simulator, and not only the start of the next timestep, while still allowing to publish the remaining inputs from the current timestep.

4.5.2 Discussion

Using the orchestrator significantly increased execution time in the simulation scenario. To reduce the runtime overhead caused by the orchestrator, multiple approaches are viable. As significant time is spent executing orchestrator callbacks and API calls, improving the performance of the orchestrator itself would be beneficial. A possible approach worth investigating could be parallelizing the execution of orchestrator callbacks. Both parallelizing multiple orchestrator callbacks and running those callbacks in parallel to the host node (the simulator or ROS bag player) could be viable. In addition to a more efficient implementation of the orchestrator itself, the overhead of serializing callback executions is significant. While some of that overhead is inherently required by the serialization to ensure deterministic execution, it has already been shown in Sections 4.1.3 and 4.1.4 that parallelism of callback executions can be improved with more granular control over callbacks, their outputs, and service calls made from within those callbacks.

When using a ROS bag instead of a simulator as the data source, some of the identified problems are less concerning. Since a ROS bag player does not have to perform any computation and reading recorded data is not usually a bottleneck for performance, the overhead of the orchestrator API calls is less problematic. Furthermore, without closed-loop simulation, the `wait_until_dataprovider_state_update_allowed` API call is not necessary which has been identified as a factor that reduces the potential for parallel callback execution. In some scenarios, the use of the orchestrator is even able to improve execution time: When replaying a ROS bag, the speed of playback is often adjusted. Use cases for playing back a recording at equal to or slower than real-time occur when the developer intends to use interactive tools for introspection and visualization such as for debugging the behavior of a software component in a specific scenario. Often, however, the user is just interested in processing all messages

in the bag, preferably as fast as possible. The playback speed is thus adjusted to be as fast as possible while the software under test is still able to perform all processing without dropping messages from subscriber queue overflow. This overflow however is usually not apparent immediately, and processing speed may depend on external factors such as system load, which makes this process difficult. When using the orchestrator, however, the processing of all messages is guaranteed and queue overflow is not possible. This allows the ROS bag player to publish messages as soon as the orchestrator allows, without specifying any constant playback rate. Playing a ROS bag is necessarily an open-loop configuration without any synchronization for dataprovider state update, and the player itself is expected to have a fast execution time when compared to the ROS nodes under test. If a speedup is achieved in the end depends on if the remaining overhead from serializing callback invocations outweighs the increased playback rate or not.

The design goal of minimizing the execution time impact is thus only partially achieved. As measured in this section and detailed in Section 4.1.5, the serialization of callbacks and thus the induced latency of executing callbacks is not minimal. The runtime of the orchestrator component itself has been shown to be significant as well, although this was not the bottleneck in this test scenario.

5 Conclusion

In this thesis, a method for repeatable execution of system tests for software stacks built using ROS was developed, implemented, and tested. The orchestrator achieves this without modifying lower levels of the ROS client library stack or middleware, by controlling callback invocations at the ROS topic level which are the source of observed nondeterminism in the execution of ROS software stacks. This is done while setting minimum requirements for the message-passing implementation, allowing in particular arbitrary transmission delay and reordering of messages. An ordering for callbacks at each node which also synchronizes service calls and concurrent access to output topics between multiple nodes is ensured using incrementally constructed callback graphs. The functionality of the implementation has been demonstrated using test cases for the distinct sources of nondeterminism in callback execution as well as with a real use case of running performance evaluation of a multi-object tracking module. This example represents the intended use case of post-processing evaluation, using the same software stack as when conducting real-world tests (which is explicitly not an area of application for the orchestrator).

This use case has also been utilized to evaluate the continued additional effort in integrating the orchestrator with new and existing ROS components. The integration of the orchestrator into an existing simulator and the implementation of a ROS bag player supporting the orchestrator has successfully been performed within this thesis. The ongoing effort of modifying new nodes for use with the orchestrator has been found to be strongly dependent on the complexity of the input/output behavior of the node. It ranged from no changes at all for simple nodes to larger modifications of the node's callback behavior such as changing from entirely ROS-independent execution to utilizing ROS timers.

Some limitations of the implemented approach have been identified. The orchestrator may currently execute callbacks in a deterministic, but unexpected order, as seen in Section 4.1.2. This is an effect of not requiring detailed information on the expected timing of callbacks and service calls, resulting in the implicit assumption that every callback has the same duration and makes service calls at the same point. In Section 4.4.2, a node was not able to be fully utilized with the orchestrator due to the specific input/output behavior. In particular, the behavior of publishing more than one message on a specified output topic in some callback invocations is currently not supported. Section 4.5 showed a significant increase in execution time when

using the orchestrator, resulting in part from non-optimal callback serialization. Both concurrent service calls (see Section 4.1.4) and concurrent callbacks which publish to the same topic (Section 4.1.3) currently serialize the entire originating callback, even if the concurrent access occurs only during a short fraction of the callback or while publishing an output.

Outlook Although the orchestrator is already useful in its current form, and using it to ensure the repeatability of automated testing of ROS components is planned, improvements in multiple areas are proposed.

As became apparent during the integration of the multi-object tracking module, it would be desirable to allow the configuration of a more complex input/output behavior than initially anticipated. The implementation for combined callbacks using the message filters package supports one complex, stateful model for callback execution, but a more general solution might exist, which would allow the integration of more nodes with fewer modifications. A possible solution for the specific problem encountered was proposed in Section 4.4.4. To further improve usability and ease the integration and maintenance of ROS nodes, automating some aspects of node and launch configuration files would be desirable.

Using static inspection or dynamic observation of a node during runtime could, for example, provide an initial version of a node description, or could detect divergence between an existing description and the observed behavior. Such analysis is possible within ROS. For instance, a method for inferring causal links between node inputs and outputs was recently proposed by Bédard et al. in [BLBD23]. Launch configurations and existing ROS launch files currently duplicate a lot of information, with unexpected behavior if configurations such as topic remappings differ between both. Reducing this redundancy would not only simplify the creation of the configuration file but also significantly reduce the potential for error while maintaining and changing both files.

A possible improvement to align the system behavior while using the orchestrator better to the behavior without the orchestrator could be to allow specifying an expected callback duration (see Section 4.1.5). This would allow the orchestrator to order the callbacks not only deterministically, but also in the order one would generally expect without the orchestrator.

Reducing the execution time impact of using the orchestrator is considered to be important for adoption. Approaches for improving the orchestrator's performance such as by multithreaded execution of orchestrator callbacks have been proposed in Section 4.5.2. In Section 4.1.5, methods for improving parallel callback execution by explicitly intercepting service calls and node outputs have been identified.

Acronyms

`rclcpp` ROS Client Library for C++ 18
`rclpy` ROS Client Library for Python 19

API Application Programming Interface 4

DDS Data Distribution Service 3

MOTA Multiple Object Tracking Accuracy 8
MOTP Multiple Object Tracking Precision 8

OSPA Optimal Subpattern Assignment 8

QoS Quality of Service 3

ROS Robot Operating System 1

Bibliography

[BLBD23] Bédard, Christophe; Lajoie, Pierre-Yves; Beltrame, Giovanni; and Dagenais, Michel: *Message flow analysis with complex causal links for distributed ROS 2 systems*. In: *Robotics and Autonomous Systems*, volume 161, 2023.

[BS08] Bernardin, Keni and Stiefelhagen, Rainer: *Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics*. In: *EURASIP Journal on Image and Video Processing*, volume 2008, no. 1, page 246309, 2008.

[CBL+19] Caesar, Holger; Bankiti, Varun; Lang, Alex H.; et al.: *nuScenes: A multimodal dataset for autonomous driving*. In: *arXiv preprint arXiv:1903.11027*, 2019.

[CBLB19] Casini, Daniel; Blaß, Tobias; Lütkebohle, Ingo; and Brandenburg, Björn B.: *Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling*. In: Quinton, Sophie (editor): *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Volume 133, 6:1–6:23. *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019.

[CKT+22] Caesar, Holger; Kabzan, Juraj; Tan, Kok Seang; et al.: *NuPlan: A closed-loop ML-based planning benchmark for autonomous vehicles*. 2022.

[DRC+17] Dosovitskiy, Alexey; Ros, German; Codevilla, Felipe; Lopez, Antonio; and Koltun, Vladlen: *CARLA: An Open Urban Driving Simulator*. In: Levine, Sergey; Vanhoucke, Vincent; and Goldberg, Ken (editors): *Proceedings of the 1st Annual Conference on Robot Learning*. Volume 78, pages 1–16. *Proceedings of Machine Learning Research*, PMLR, 2017.

[HMG+23] Henning, Matti; Müller, Johannes; Gies, Fabian; Buchholz, Michael; and Dietmayer, Klaus: *Situation-Aware Environment Perception Using a Multi-Layer Attention Map*. In: *IEEE Transactions on Intelligent Vehicles*, volume 8, no. 1, pages 481–491, 2023.

[KH04] Koenig, N. and Howard, A.: *Design and use paradigms for Gazebo, an open-source multi-robot simulator*. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Volume 3, 2149–2154 vol.3, 2004.

[MFG+22] Macenski, Steven; Foote, Tully; Gerkey, Brian; Lalancette, Chris; and Woodall, William: *Robot Operating System 2: Design, architecture, and uses in the wild*. In: *Science Robotics*, volume 7, no. 66, eabm6074, 2022.

[NWB+21] Neurohr, Christian; Westhofen, Lukas; Butz, Martin; et al.: *Criticality Analysis for the Verification and Validation of Automated Vehicles*. In: *IEEE Access*, volume 9, pages 18016–18041, 2021.

[SMHB21] Strohbeck, Jan; Müller, Johannes; Holzbock, Adrian; and Buchholz, Michael: *DeepSIL: A Software-in-the-Loop Framework for Evaluating Motion Planning Schemes Using Multiple Trajectory Prediction Networks*. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7075–7081, 2021.

[SVV08] Schuhmacher, Dominic; Vo, Ba-Tuong; and Vo, Ba-Ngu: *A Consistent Metric for Performance Evaluation of Multi-Object Filters*. In: *IEEE Transactions on Signal Processing*, volume 56, pages 3447–3457, 2008.

[WNK+23] Westhofen, Lukas; Neurohr, Christian; Koopmann, Tjark; et al.: *Criticality Metrics for Automated Driving: A Review and Suitability Analysis of the State of the Art*. In: *Archives of Computational Methods in Engineering*, volume 30, no. 1, pages 1–35, 2023.