

Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems

Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, Michel Dagenais

Abstract—Distributed robotic systems rely heavily on the publish-subscribe communication paradigm and middleware frameworks that support it, such as the Robot Operating System (ROS), to efficiently implement modular computation graphs. The ROS 2 executor, a high-level task scheduler which handles ROS 2 messages, is a performance bottleneck. We extend `ros2_tracing`, a framework with instrumentation and tools for real-time tracing of ROS 2, with the analysis and visualization of the flow of messages across distributed ROS 2 systems. Our method detects one-to-many and many-to-many causal links between input and output messages, including indirect causal links through simple user-level annotations. We validate our method on both synthetic and real robotic systems, and demonstrate its low runtime overhead. Moreover, the underlying intermediate execution representation database can be further leveraged to extract additional metrics and high-level results. This can provide valuable timing and scheduling information to further study and improve the ROS 2 executor as well as optimize any ROS 2 system. The source code is available at: github.com/christophebedard/ros2-message-flow-analysis.

Index Terms—Software tools for robot programming, distributed robot systems, Robot Operating System (ROS), performance analysis, tracing.

I. INTRODUCTION

Modern robotic systems often leverage complex distributed processing: they use distributed perception [1], motion planning [2], and decision making [3]. They are built on software frameworks like ROS 2 [4], the successor to the Robot Operating System (ROS) [5]. Such middleware frameworks greatly simplify the development of modular computation graphs. However, high-level scheduling of tasks in ROS 2 (i.e., internal message handling, and subscription, service, or timer callback execution) brings a number of performance challenges. Several methods and tools have been proposed to study the default ROS 2 executor and compare its performance with other proposed executor designs [6], [7], [8], [9]. Furthermore, getting message latencies across a whole system is paramount to assessing the impact of these new approaches as well as evaluating the overall performance of a system, from one end to the other. Tools have been introduced to study message latencies in real systems [10], [11]. However, these techniques are either not applicable to existing ROS 2 systems, since they require non-trivial code modifications, or result in significant runtime overhead. Finally, existing techniques also cannot identify relationships

The financial support of Ericsson, NSERC, Prompt, and Vanier Canada Graduate Scholarship is gratefully acknowledged.

Department of Computer Engineering and Software Engineering, Polytechnique Montréal, Montreal, Quebec H3T 1J4, Canada, {christophe.bedard, pierre-yves.lajoie, giovanni.beltrame, michel.dagenais}@polymtl.ca

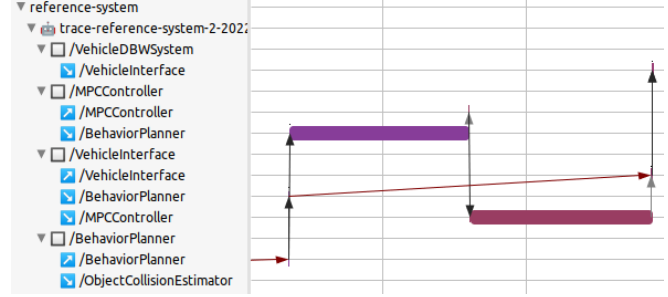


Fig. 1. Message flow visualization using our method.

between messages across possibly-distributed systems, which is necessary to extract the end-to-end latency.

Low-overhead tracing has been used as a way to extract execution information for performance analysis purposes without impacting or perturbing the system. Furthermore, various techniques allow combining and correlating kernel and userspace events from multiple traces (i.e., from distributed systems). In previous work, we proposed `ros2_tracing` [12], a framework with low-overhead instrumentation and orchestration tools for tracing ROS 2. This tracing framework allows extracting ROS 2 execution information, and can be extended with additional instrumentation for more advanced use-cases or performance analysis goals, such as providing information about the overall performance of the message-passing system.

Moreover, critical path analysis has been used to model the interactions inside parallel and distributed systems. For example, wait-related kernel events can be used to recursively compute wait dependencies across machines for requests spanning multiple hosts in a distributed system [13]. This technique helps to understand and explain the actual process execution, and to identify the prime target for performance optimization (i.e., the bottleneck). Likewise, in this paper, we present a message flow analysis method for ROS 2 distributed computation graphs. It can be useful for both end users of ROS 2, looking to analyze and optimize their system, and for developers, looking to do the same for the internals of ROS 2, although these two target audiences are not necessarily distinct.

Contributions. As shown in Fig. 1, our proposed technique can extract and visualize the paths of messages across distributed ROS 2 systems, providing information about the overall systems performance. Building a message flow graph in a low-overhead way, without modifying the applications themselves as the existing methods do, requires more com-

plex runtime execution data collection and analysis. With this novel approach, we bring the following contributions:

- An intermediate execution representation database of trace data obtained from a distributed system, providing information on ROS 2 objects (i.e., nodes, publishers, subscriptions, and timers) and events (i.e., message publication and reception instances, and subscription and timer callback instances). This database can be further leveraged to derive additional metrics and high-level results.
- Matching of published and received messages, compatible with distributed systems, without needing to modify the user code and without adding significant overhead.
- Inference of one-to-one and one-to-many causal links between received messages and published messages, both automatically for direct links, and using simple user-level annotations for more complex indirect causal links.
- Extraction and visualization of the flow of messages across distributed ROS 2 systems, as well as the state of the executor over time for each process.
- Experiments and validation of our proposed method on both synthetic and real robotic systems, demonstrating that it can be used to study and optimize existing systems, and that it has a low runtime overhead.

This paper is structured as follows: We first survey related work in Section II. We then summarize relevant background information and describe our intermediate execution representation in Section III. Thereafter, we present our analysis method in Section IV and our executor visualization in Section V. Next, Section VI presents experiments where we apply our method to two systems, and Section VII provides an evaluation of the runtime overhead. Future work is outlined in Section VIII. Finally, we conclude in Section IX.

II. RELATED WORK

Previous work has identified open problems relating to the communications latency of ROS 2 [4] and its executor. Relevant methods were proposed to observe and study those problems.

A. Communications

First, the general performance of ROS 2 was evaluated by Maruyama et al. [14], Gutiérrez et al. [15], and Puck et al. [16]. Other work focuses on more specific elements of the performance of ROS 2, including its overhead with relation to the underlying middleware, DDS [17]. Kronauer et al. [18] evaluated the overhead of ROS 2 using profiling, and showed that it can lead to a 50% latency overhead. Some of this overhead can be attributed to the serialization and deserialization of complex message structures. Jiang et al. [19] proposed an adaptive serialization technique to improve communication performance by up to 93%. Wang et al. [20] proposed a single-host inter-process communications (IPC) layer for ROS 1 [5] and ROS 2 which reduces the overhead of IPC for large messages. Finally, Puck et al. [21] noted in another performance evaluation of ROS 2 that the

use of dynamic memory allocations, when fetching new messages from the underlying middleware, accounts for a significant portion of the internal message processing time.

Various tools have been proposed to study message latency in ROS 2. The `performance_test` [22] benchmarking tool allows measuring the latency between publishers and subscriptions directly, while [23] allows defining a custom message graph topology. However, these benchmarking tools only evaluate the performance of a synthetic system or communication configuration. To measure the performance of real systems, observability tools are needed. Nishimura et al. [10] proposed RAPLET, which breaks down the latency between the publication of a message and the execution of the subscription callback function on the other end. It tracks messages using a sequence number in the message structure itself. Unfortunately, this sequence number field is not included in all messages. Similarly, Witte and Tichy [24] presented a tool to track messages in ROS 1 in order to interactively modify their content. These techniques cannot be applied to existing systems, since they require the addition of a custom message header, and their runtime overhead is significant, which can affect the validity of their results [25].

B. Executor

Moreover, other previous work has identified and studied open problems with the ROS executor, which is a high-level task scheduler [26]. It is responsible for fetching new messages from the underlying middleware and executing the corresponding subscription callbacks as well as timer callbacks, making the executor a clear performance bottleneck. Furthermore, scheduling tasks on top of the OS scheduler itself is challenging. Multiple methods model exchanges of ROS messages, from node to node, as event chains and pipelines in a directed acyclic graph (DAG). Peeck et al. [27] focused on online monitoring for reacting to latency violations in event chains. Casini et al. [28] proposed a formal scheduling model and a response-time analysis for ROS 2 to bound worst-case response times. Tang et al. [29] then proposed a more specific version that is, however, only valid for independent linear processing chains. Blass et al. [30] built on the work by Casini et al. [28] and proposed an online automatic latency manager for ROS 2. Their work also helped illustrate how the higher-level scheduling of tasks in ROS 2 does not interact well with classic OS-level scheduling techniques. Blass et al. [31] further extended this work, and stressed how the ROS 2 executor differs from normal schedulers in the literature, since it inherently prioritizes in order: timers, subscriptions, service servers, and service clients.

To help tackle some of these challenges, new executor designs have been proposed for ROS 2. The callback-group-level executor [6], now available as an alternative in ROS 2, allows having multiple distinct executor instances on multiple threads without interference. This enables scheduling of the OS threads themselves, using different priorities depending on system requirements, instead of bundling all ROS 2 elements together, as the default executor does. This

results in lower latencies for higher-priority callback groups, as demonstrated by Yang and Azumi [32]. Similarly, Choi et al. [7] proposed a priority-driven chain-aware scheduler and showed that it helps lower end-to-end latencies as well. Staschulat et al. [8], [9] proposed a budget-based executor for real-time operating systems. To benchmark and compare executor designs, a reference system was proposed [33]. It is based on the computation graph of Autoware [34], [35], an autonomous driving system completely based on ROS.

C. Tracing and Data Analysis

To investigate performance issues, low-overhead tracing has been widely used for collecting execution information in a minimally-invasive way. In particular, the LTTng tracer [36], which has a low runtime overhead [37], was used by Lütkebohle [38] to investigate determinism and message timing issues in ROS 1. They proposed [39] as a generic tracing tool for ROS 1. As a follow-up to this for ROS 2 – and to improve on it – in previous work, we presented `ros2_tracing` [12], a framework with low-overhead instrumentation and tracing tools for ROS 2. The proposed instrumentation can be used to extract simple metrics, such as publishing rate and subscription or timer callback duration, as demonstrated in [12]. For other, more advanced performance analysis use-cases, the instrumentation can easily be extended.

To extract useful information from trace data, advanced trace analysis methods build models from the trace data. One interesting technique is the critical path method, where the critical path is defined as the longest path in a DAG. The critical path is therefore the overall program or end-to-end latency bottleneck; shortening it effectively reduces the total execution time. Yang and Barton [40] applied the method to compute the critical path of the execution of parallel and distributed programs. Trace data from multiple hosts in a distributed system can be combined and synchronized for analysis as a whole [41], [42], [43]. Giraldeau and Dagenais [13] used wait-related trace events from the kernel (e.g., scheduling, network, or interrupts) to recursively compute wait dependencies across machines. While such wait-related operating system primitives are used in many applications, application-level information is required for more specialized analyses [44]. For example, ROS-level information could be used to apply the critical path method to the computation graph of a ROS system.

Previous work has partially tackled this critical path analysis effort. Santos et al. [45] used static code analysis to extract a model of the system architecture for ROS 1. While it does not require executing the code, it only provides an overall view of the system architecture: it does not provide time-related information about the individual messages going through the system and the links between the messages. Therefore, to consider possibly complex dynamic timing interactions, runtime execution information is required. To this end, Li et al. [11] used `ros2_tracing` [12] and `tracetools_analysis` [46], a simple trace data processing library, to provide an end-to-end latency breakdown.

However, links between input subscriptions and output publishers need to be manually provided by the user; they are not automatically detected. [39] was used and extended by [47] to visualize the flow of messages. Unfortunately, [47] has many limitations and uses simplistic assumptions which do not always hold. For example, to track messages between nodes, it selects the first TCP packet that is queued after a message is sent by ROS 1. It then matches that network packet when it is received on the other end, and selects the next message reception event. This heuristic is simple and does not require adding additional fields to the messages themselves, but it is far from solid, since it could select packets from other applications. Furthermore, it only considers direct causal links inside subscription callbacks, i.e., where the message being processed by the callback instance is linked to the message that is published during that callback instance. However, many systems use custom message cache mechanisms that are independent from the ROS 2 API, which makes detecting and modeling those causal links far from trivial. Finally, it does not support one-to-many or many-to-many causal links, i.e., where one or more input messages are linked to more than one output message, and also does not work with more than one machine.

D. Summary

In summary, numerous latency- and executor-related open problems exist in ROS 2. Building a model and graph of the path of messages across a ROS 2 system would provide useful information to further study or work on resolving those open problems. Thus, we propose a low-overhead technique that can transparently and natively track messages while supporting complex application-dependent causal links between messages.

III. INTERMEDIATE EXECUTION REPRESENTATION

Before extracting the flow of ROS 2 messages across a distributed system, we first process the raw trace data to create a higher-level intermediate representation of the execution. The underlying database can then be queried to build the actual message flow analysis; this process is greatly simplified by the intermediate representation. The database can also be queried for other analysis purposes.

A. ROS 2 Architecture

As shown in Fig. 2, ROS 2 contains multiple abstraction layers. From top to bottom, i.e., from user-level to OS-level: `rclcpp` and `rclpy`, `rcl`, and `rmw`. The client libraries, `rclcpp` and `rclpy`, offer the actual user-facing ROS 2 C++ and Python APIs, respectively. They use a common underlying library, `rcl`; this architecture reduces duplicate code and thus makes adding new client libraries simpler. Then, `rcl` calls `rmw`, the middleware interface. This interface is implemented for each underlying middleware implementation, e.g., for each distinct DDS implementation. This allows ROS 2 to use any message-passing middleware with any transmission mechanism, as long as it is done through this interface. The flow of information, from the

user code on one end to the user code on the other end, is illustrated in Fig. 2.

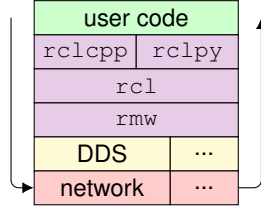


Fig. 2. ROS 2 architecture and interactions between layers. For example, a message is created by user code, goes through the layers of the ROS 2 architecture down to the network, and then goes back up to the user code on the other end (e.g., from one node to another node). While DDS is the default middleware, other middlewares and transmission mechanisms can be used.

B. Processing

Since the ROS 2 architecture contains multiple separate abstraction layers, the information collected by `ros2_tracing` [12] for analysis purposes is spread out over all layers. This also provides internal information about ROS 2. For example, the duration of the message publication call can be broken down into `rclcpp`, `rcl`, and DDS time. Hence, `ros2_tracing` instruments all layers in order to collect all relevant information. Furthermore, to minimize the runtime impact, the `ros2_tracing` instrumentation is split into two distinct groups: initialization and runtime. The initialization instrumentation points collect one-time information, in order to minimize the size of the data collected by the runtime instrumentation points, which are executed more often. Therefore, we need to combine data from multiple instrumentation points in order to get the high-level information we need. Moreover, `ros2_tracing` does not include instrumentation for the chosen DDS implementation; thus we instrumented it and combined all this information.

For example, when a new publisher object is created, 3 tracepoints are triggered: the first one is in `rcl`, the second one is in `rmw`, and the last one is in the DDS implementation. By combining the execution information collected at different levels of the ROS 2 architecture by the 3 tracepoints into one publisher entry in the database, we can attribute the `rcl`-, `rmw`-, or DDS-level data, of the subsequent publication instance trace events, to the corresponding publisher. To correlate and merge the information from multiple trace events, unique identifiers are required. In most cases, `ros2_tracing` uses the values of the pointers to the underlying internal data structures, i.e., memory addresses. To combine DDS-level information with the above ROS 2 information, we use the globally-unique identifier (GUID or GID) of the DDS data writer, which is the DDS-level object that actually sends messages from a ROS 2 publisher. This GUID is used internally in ROS 2 and is part of the `rmw` interface. We have instrumented both eProsima Fast DDS [48] and Eclipse Cyclone DDS [49]. As a result of the above design, either DDS implementation can be used without affecting the model.

Our intermediate representation needs to be valid when tracing multiple processes on one host computer, and when combining data from multiple hosts. Unfortunately, memory addresses are only valid for one process. To account for multiple processes on the same host, we combine the pointer value with the process ID (PID). Then, to account for multiple hosts, we combine the pointer value and PID with a unique host ID obtained from the trace data. This 3-tuple is thus unique across different processes and hosts.

In summary, as depicted in Fig. 3, the resulting intermediate representation database contains information about all ROS 2 objects: nodes, publishers, subscriptions, timers, and executors. It also contains all relevant instances: message publication instances, subscription and timer callback instances, and executor states over time. This database can then be queried for analysis purposes. As for services and actions, while they are not included in the model, the corresponding objects and request-response instances could easily be added, since they are similar to the current model.

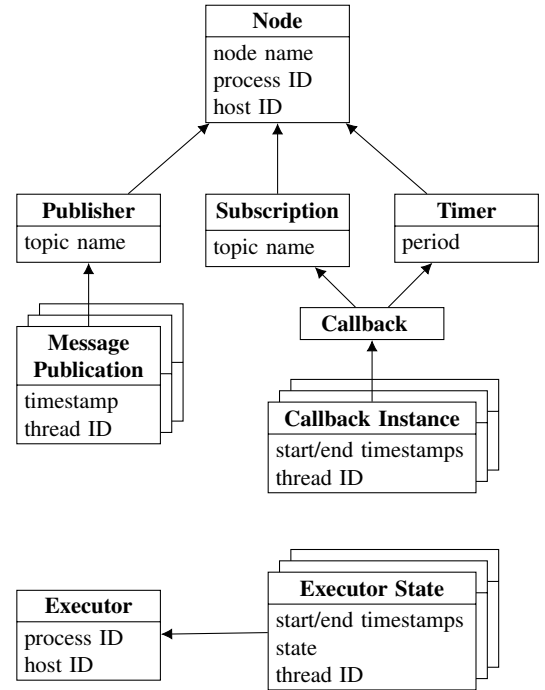


Fig. 3. Information in the intermediate representation database. Arrows represent relationships between objects and instances in the database. For example, each message publication instance is linked to its corresponding publisher object, which is itself linked to its corresponding node.

C. Implementation Details

To build a database of raw trace data as an intermediate representation, as described in the previous section, we used Eclipse Trace Compass [50], an open-source trace analysis framework. Since traces collected from multiple computers usually do not have the same clock reference, the traces need to be synchronized for the combined data to be valid, time-wise. Trace Compass can synchronize traces from distributed systems using network packet data collected from the kernel [43]. System clocks can also be synchronized

directly using NTP [51]. The time synchronization method and its precision should of course be taken into consideration when extracting time-related information from the database. We then use Trace Compass to perform our message flow analysis using information from the intermediate execution representation database: this is greatly simplified by using higher-level, preprocessed information (as shown in Fig. 3) instead of raw trace data.

IV. MESSAGE FLOW ANALYSIS

Our proposed message flow analysis builds a graph of the path of messages across a ROS 2 system using the information from the intermediate execution database, described in the previous section. However, to achieve this, we must add more information to the database and combine multiple elements. We must first track messages as they are sent over the network transport, to link a message being published by a publisher to the same message being received by one or more subscriptions (Section IV-A). Then we add causal links, between messages that act as an input to a node, to the messages that are published by that node as the output (Section IV-B). Finally, we put everything together to build the flow graph (Section IV-C).

A. Transport Links

Transport links associate a publication instance to the corresponding subscription callback instance on the other end. These links are always one-to-many, since messages always originate from a single publisher but can be received by any number of subscriptions.

ROS 2 internally provides metadata for all received messages, including the GID of the source publisher and the timestamp of the time right before DDS sent the message over the network. This information is collected on the subscription side using an instrumentation point in `rmw`. The publisher GID is collected during initialization; the source timestamp is collected on the publisher side using DDS instrumentation, since this information is not made available to ROS 2. To uniquely identify messages and thus avoid collisions in case multiple publishers emit messages at the same time, we should use a combination of the publisher GID and the source timestamp. Unfortunately, as of writing this, a bug in the implementation of the `rmw` interface for Cyclone DDS¹ prevents us from relying on the GID. We therefore instead reduce the probability of collisions by combining the source timestamp with the topic name, which is known on both sides of the transport link. Furthermore, unless the source clock has a higher granularity, collisions are unlikely, given that source timestamps have nanosecond-level precision. These elements are all available from the intermediate execution representation database.

Collecting this low-level execution information, to track messages, allows our method to work transparently, i.e., without needing to modify a system to add fields to messages or rely on high-level tracking logic, unlike what

is done by [10], [24]. More importantly, we expect the overhead to be much smaller, given the low overhead of the `ros2_tracing` instrumentation, as demonstrated in [12]. Fig. 4 shows an example of a transport link, with a subscription on one computer receiving a message from a publisher on another computer. Given our technique for tracking messages and uniquely identifying ROS 2 objects (see Section III-B), there is no difference between a transport link between two computers, and a transport link constrained to a single computer.

B. Causal Message Links

For causal links, we define the causality of messages based on both time and value. In direct cases, an output message is generated and published when a new input message is received and processed, thus linking the two messages. In indirect cases, an input message is linked to an output message if the content of the former is used to generate the content of the latter, without any strict requirements on time. Indeed, the link is not strictly time-related, since causal links can be asynchronous, as we will explain in the following.

1) *Direct Case:* For the direct case, new messages are published on any number of topics directly during the subscription callback for a received message. The input message is thus linked to all messages that are published between the start and end of the corresponding subscription callback instance on the same thread. For example, by collecting execution information during runtime using `ros2_tracing`, we obtain the following trace events:

- 1) At timestamp X : start of callback on thread T for message I received by subscription S
- 2) At timestamp Y : publication of message O with publisher P on thread T
- 3) At timestamp Z : end of callback on thread T for message I received by subscription S

With $X < Y < Z$, message I received by subscription S is directly linked to message O published by publisher P , since it was published during the callback for message I on the same thread T . This direct causal link can therefore be inferred using the execution information collected with `ros2_tracing`; no user-level annotation is necessary for this case. Since normal ROS 2 subscription callbacks only process a single message, the causal link for the direct case is strictly one-to-many. Another example is shown in Fig. 5, with a pipeline of three nodes and direct one-to-one causal links. In this case, the message flow graph generated from this exchange would be visually identical, since there are no additional links to be found.

2) *Indirect Case:* For the indirect case, causal links are the result of user-level code, i.e., above the ROS 2 API. We therefore cannot detect these causal links from the trace itself; users need to provide this application-specific information for the links to be detected. We achieve this using simple annotations in the form of one-time tracepoints during the initialization phase, after the subscriptions and publishers have been created. As shown in Listing 1, annotations simply contain the name of the message link type and the list of

¹github.com/ros2/rmw_cyclonedds/issues/377

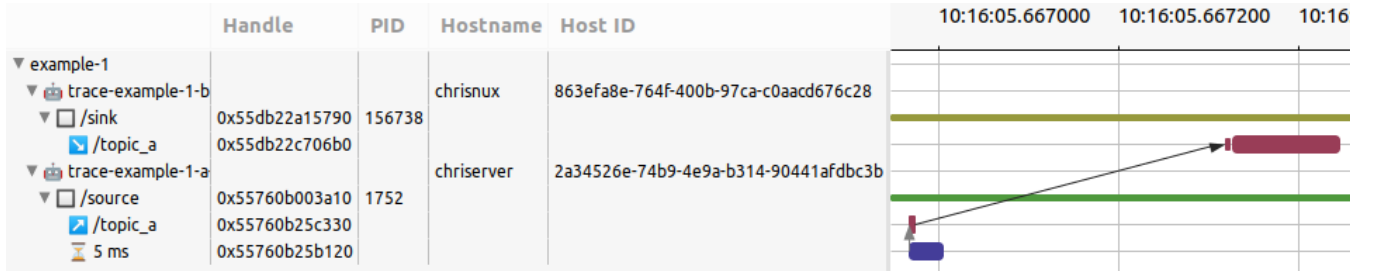


Fig. 4. Transport link example. The tree structure on the left represents traces, with publishers, subscriptions, and timers under the nodes of each trace. To the right of this, internal handles, PIDs, and host information are shown: this is the 3-tuple needed to uniquely identify ROS 2 objects (see Section III-B). Then, on the right is a time-based chart, which provides an abstract representation of the execution using time segments and arrows. In this example, a 5 ms timer triggers a callback which publishes a message on `/topic_a` under node `/source`. This message is received by the `/topic_a` subscription of node `/sink` on the other computer. Next to timers and subscriptions, segments represent the duration of a specific callback instance, from beginning to end. The smaller segment before the subscription callback segment represents the message being fetched (or *taken*) from the underlying middleware, before it is provided to the callback instance. For publishers, the segments represent the duration between the initial user-level publication call and the underlying DDS call. The longer arrow between the `/topic_a` publisher and subscription represents the transport link, i.e., the message going from the publisher to the subscription over the network. The shorter arrow between the 5 ms timer callback and the `/topic_a` publisher shows that the message was published during the timer callback.

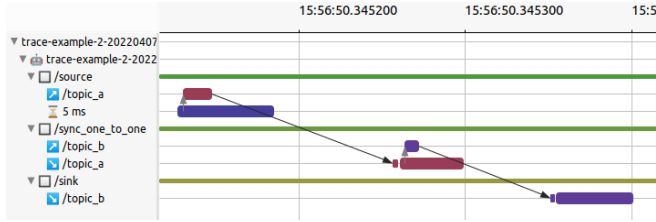


Fig. 5. Direct causal message link example. A message is published on `/topic_a` during a 5 ms timer callback by node `/source`. The message is then received by the corresponding subscription under node `/sync_one_to_one`. During the subscription callback for that message, a message is published on `/topic_b`, which is finally received by the corresponding subscription under node `/sink`. The link between the input message and the output message is therefore a direct one-to-one causal link.

input subscriptions and output publishers as arrays along with their length. By providing the connection between the input publishers and the corresponding output subscriptions and the message link type, indirect causal links between input messages and output messages can be inferred from the runtime execution data. Annotation can thus be easily added to an existing system, without needing to modify the existing application logic or message structures as is done by [10], [24].

Listing 1
USER ANNOTATION TRACEPOINT EXAMPLE.

```
TRACEPOINT (
  message_link_type,
  (rcldcpp::Subscription *) subscriptions,
  subscriptions_length,
  (rcldcpp::Publisher *) publishers,
  publishers_length);
```

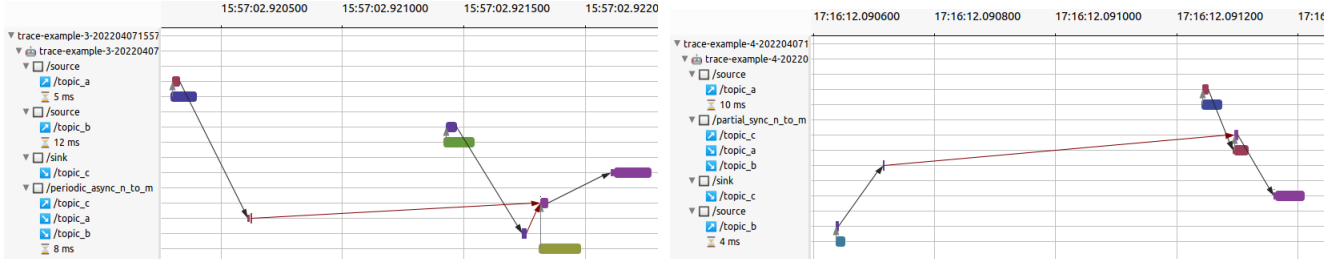
From studying real systems and the Autoware reference system [33], we define two types of causal links: periodic asynchronous link and partial synchronous link, both N-to-M, i.e., many-to-many. With the periodic asynchronous link, messages are received from N topics and are cached. A

timer periodically triggers a callback during which the last message from each of the N caches is used to compute a result and then publish messages on M topics. For the partial synchronous link, messages are received from N topics and are cached as well. However, the result is conditionally computed during the subscription callback itself: if all N caches contain a message, a result is computed and published on M topics. The caches are then reset so that at least one new message from each of the N input topics is received again before the next output. Without the link annotation, the partial synchronous link would be similar to the direct case; however, it would only link one out of N real input messages. Finally, while we only present two types of indirect causal links, more types could be identified. If other link types are defined, i.e., with valid heuristics, then they can be easily added to our method.

With the information from the annotations and the timer and subscription callback, and message publications instances from the intermediate execution representation database, we can thus automatically infer indirect causal links between specific input and output messages. Therefore, unlike [11], users do not need to provide these links after the fact, since they are already in the trace data. Furthermore, unlike the direct case, which is limited to a single input message per link, indirect causal links can be many-to-many, given their asynchronous nature. Fig. 6a shows an example for the periodic asynchronous link, while Fig. 6b shows an example for the partial synchronous link. In both cases, two input messages result in one output message. However, the mechanics of the two causal links are different. For the periodic asynchronous link, the output rate and delay between input and output depend on the period value for the timer. For the partial synchronous link, the output rate only depends on the rate of the inputs.

C. Building the Message Flow Graph

Information from the intermediate execution representation database, including transport links and direct causal



(a) Periodic asynchronous causal message link. All messages received by node `/periodic_async_n_to_m` are cached. The periodic callback triggered by the 8 ms timer then uses those cached messages to compute and publish an output message on `/topic_c`. Therefore, the subscription callback of the input messages (`/topic_a` and `/topic_b`) are linked to the output message publication (`/topic_c`). These two periodic asynchronous links are shown in red.

(b) Partial synchronous causal message link. Messages are received by node `/partial_sync_n_to_m`. The first message (`/topic_b`) is received and cached, since the other cache is empty. However, when the second message (`/topic_a`) is received, both messages are available, and are therefore used to compute and publish an output message on `/topic_c`. Therefore, the subscription callback of the first message (`/topic_b`) is linked to the output message publication (`/topic_c`) which happens during the subscription callback for the second message (`/topic_a`).

Fig. 6. Indirect causal message links examples: (a) periodic asynchronous and (b) partial synchronous. In both examples, messages are published periodically using timers on `/topic_a` and `/topic_b` by two `/source` nodes. These messages are then received by subscriptions under the `/periodic_async_n_to_m` and `/partial_sync_n_to_m` node, respectively. An output message linked to the input messages is eventually published on `/topic_c`.

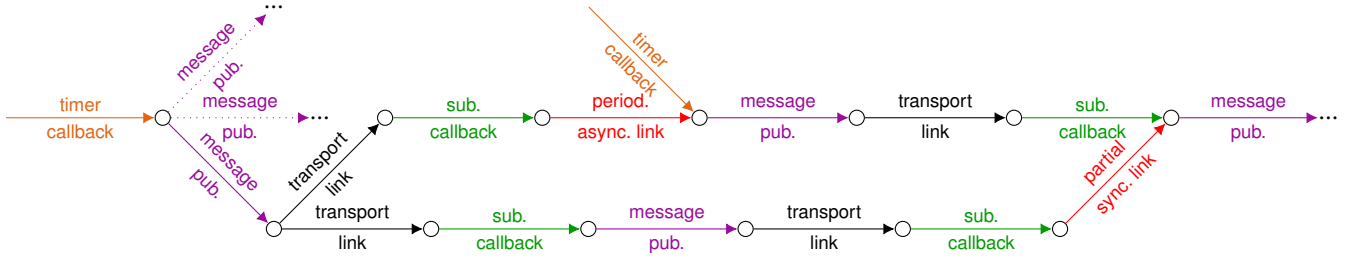


Fig. 7. Simplified representation of a typical message flow graph, showing all edge types, each type with a specific color. Edges are segments of the message flow, and their duration is their weight. Vertices link one or more input edges to one or more output edges. The third message (bottom) has two outgoing transport links, i.e., it is received by two subscriptions. The first message (top) is processed by a subscription callback and put into a message cache. This message has a periodic asynchronous causal link to an output message generated and published by a timer callback (above). This last message is then received and processed by a subscription callback, which uses it along with another message linked by a partial synchronous causal link (below) to generate and publish a final message.

message links, can be displayed directly to provide a visual representation, as shown in Fig. 4 and Fig. 5. We then need to use this information to build the message flow graph for a particular message, as selected by a user in the Trace Compass GUI.

As presented by Casini et al. [28] and used by [29], [30], [31], ROS computation graphs can be modeled as directed acyclic graphs (DAGs). The flow of a message can therefore also be modeled as a DAG. Fig. 7 shows a simplified version of a typical message flow graph. Message flow graphs have a limited set of edge types. Each edge type can be preceded and followed by a specific set of other edge types. For example, transport link edges are preceded by message publication edges and followed by subscription callback edges. Using this logic, and the different data sources presented in previous sections, the message flow graph can be constructed recursively, one edge at a time, going both forward and backward from the initial edge selected by the user. For example, an output message will be linked to all input messages that are detected, possibly combining both direct and indirect causal links. It is important to note that message flow graphs may be incomplete or invalid if links are not detected. This could

happen if the user does not correctly annotate indirect causal links, or if the traced system contains types of indirect causal links that are not supported.

Unlike [47], which assumed for simplification purposes that the message flow graph is a directed graph that only contains one-to-one links, our method supports one-to-many transport links and many-to-many causal message links. Furthermore, unlike [47] again, our method also builds the message flow graph both forward and backward from the initial element. This can be useful when analyzing traces from a ROS 2 system: building a message flow graph starting from one of the roots of the ROS 2 computation DAG will be different from a graph built starting from the leaf of a computation DAG, even if the resulting message flow graphs intersect. The initial segment from which to build the message flow graph can thus be chosen depending on the user needs.

Finally, there is one exception when modeling ROS computational graphs as DAGs. As explained by Blass et al. [30], the `/tf` topic is a special topic. It is used to communicate information about relationships between coordinate frames (*transforms*). All nodes that need and provide this infor-

mation both subscribe and publish to the same topic, thus seemingly creating a loop in the graph model. However, `/tf` messages have a field which identifies the two coordinate frames to which the transform message applies. Therefore, a node publishing a `/tf` message might also receive that same message; however, it will not be used. As Blass et al. [30] do, we can detect the transport links for `/tf` with the same node as the source and destination, and remove them, since we assume that these messages are not meant for the originating node itself, and will not be used by it. However, this does not actually cause loops in our message flow graph implementation. Indeed, the subscriptions to the `/tf` topic are special subscriptions that are managed by ROS 2: messages are received and put into a buffer, the content of which is used eventually by the user. This application-level link is not detected; therefore, our method does not detect any message flow segments after subscription callbacks for `/tf` messages.

V. EXECUTOR STATE VISUALIZATION

Using information from the intermediate representation database, we also build a visualization of the state of the executor over time. As explained in Section II-B, the ROS 2 executor is responsible for fetching new messages from the middleware and executing the user-provided subscription and timer callbacks. We split the executor runtime into three distinct states that repeat over time. In the first state, the executor waits for new events, such as new messages from the underlying middleware or timers that are ready to execute. Then, in the second state, the executor does internal processing to select the subscription or timer that will be executed, which represents pure executor overhead. This is an open problem in ROS 2, along with suboptimal scheduling policies. Finally, in the third state, the executor executes the corresponding user-provided subscription or timer callback. Fig. 8 shows an example of the resulting executor visualization over time. For a given executor instance, orange means that the executor has nothing to process, while green means that it is busy executing callbacks. This goes alongside the message flow analysis presented in Section IV, since it can help explain message processing delays: even if a message is received, the corresponding subscription callback might wait for some time until the executor instance executes it. This visualization can therefore be used to compare different executor designs and configurations to study and address the aforementioned open problems.

VI. EXPERIMENTS

We first apply our proposed method to a synthetic system and then apply it to a real system. These experiments demonstrate how our technique can be used to analyze ROS 2 itself, as well as application-level logic, for performance optimization purposes. The code and instructions for these experiments are available in our repository.

A. Autoware Reference System

For our first experiment, we use a reference system [33] with a synthetic computation graph based on Autoware [34],

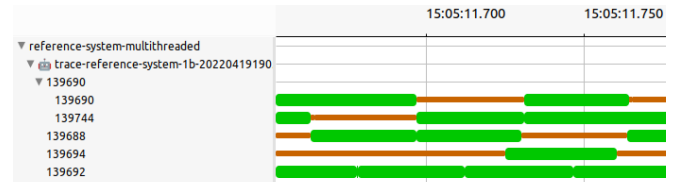


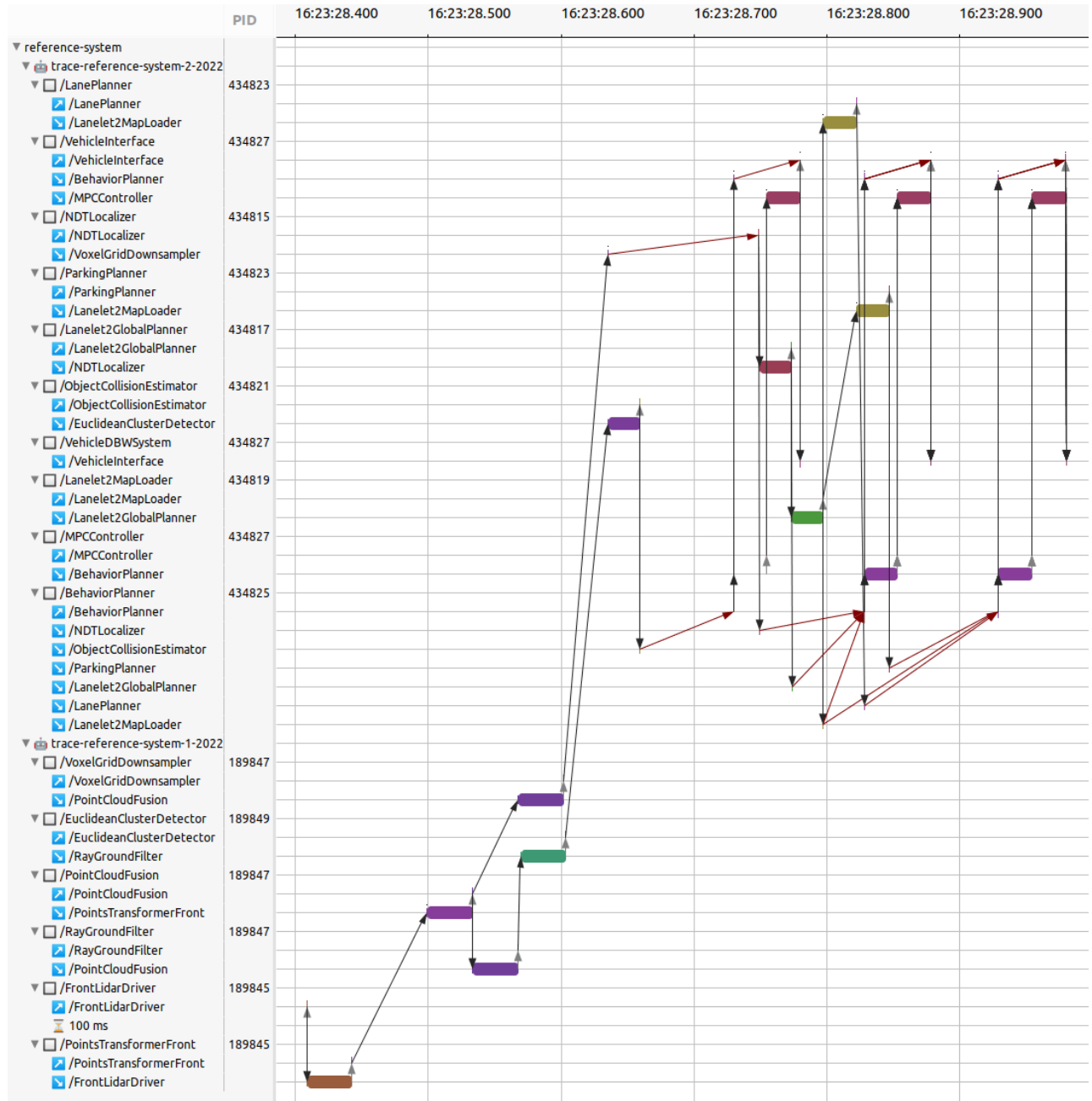
Fig. 8. Visualization of the state of executor instances for each process over time. The tree structure on the left lists IDs of processes under each trace. For multi-threaded executor instances, individual thread IDs are listed under the process ID. Green segments represent execution instances (e.g., timer callback, subscription callback). Orange segments indicate that the executor is waiting for new events. Red segments represent internal executor processing, although they are too small to be visible for this time range.

[35], a ROS-based autonomous driving stack. The nodes and topics are based the Autoware system; however, all messages have the same type, and computation is replaced with a processing-intensive task, in order to consume CPU time. The computation graph has multiple inputs and outputs, i.e., sensor data and vehicle commands or secondary visualization outputs. It uses all types of causal message links, as defined in Section IV-B. However, it specifically uses the periodic asynchronous link in a many-to-one configuration, and the partial synchronous link in a two-to-one configuration.

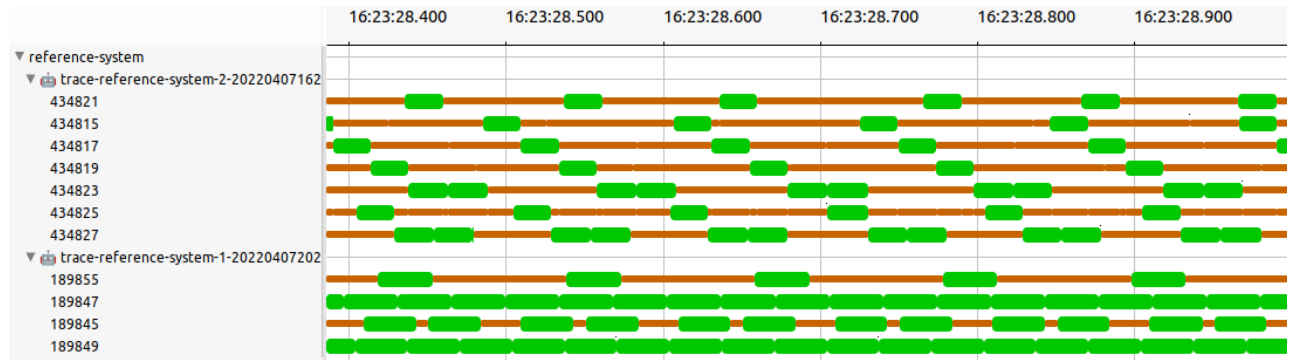
We split the nodes defined in the reference system into multiple executables and split those executables into two launch files. Each launch file is run on a specific host, with the two hosts being on the same network. We set the launch files to configure the LTTng tracer using `ros2_tracing` and enable all ROS 2 and DDS tracepoints. Furthermore, we enable network-related events, in order to synchronize the traces using [43].

Fig. 9 shows the entire message flow graph starting from one of the lidar drivers, which is one of the roots of the computation graph, along with the state of the executors over time. As displayed in Fig. 9a, the initial `/FrontLidarDriver` message results in three separate `/VehicleInterface` messages to the `/VehicleDBWSysyem` node. This is due to the caching and asynchronous nature of the indirect causal links, which results in one-to-many or many-to-many links, as explained in Section IV-B.2. The end-to-end latency ranges from 370 ms for the first message to 571 ms for the third message.

Furthermore, as displayed in Fig. 9b, a secondary visualization shows the state of all executor instances over time. In this experiment, we only use the default single-threaded executor, which means that timer and subscription callbacks within a single process can only be processed one at a time. We can see that some executor instances are busier than others; if executors are too busy, there can be a greater delay between message reception and processing. Depending on subscription options, old messages could be dropped, which wastes the CPU time used for publishing those dropped messages, thus resulting in a generally poor performance optimization. For example, unlike all executor instances under `trace-reference-system-2`, the executor instance

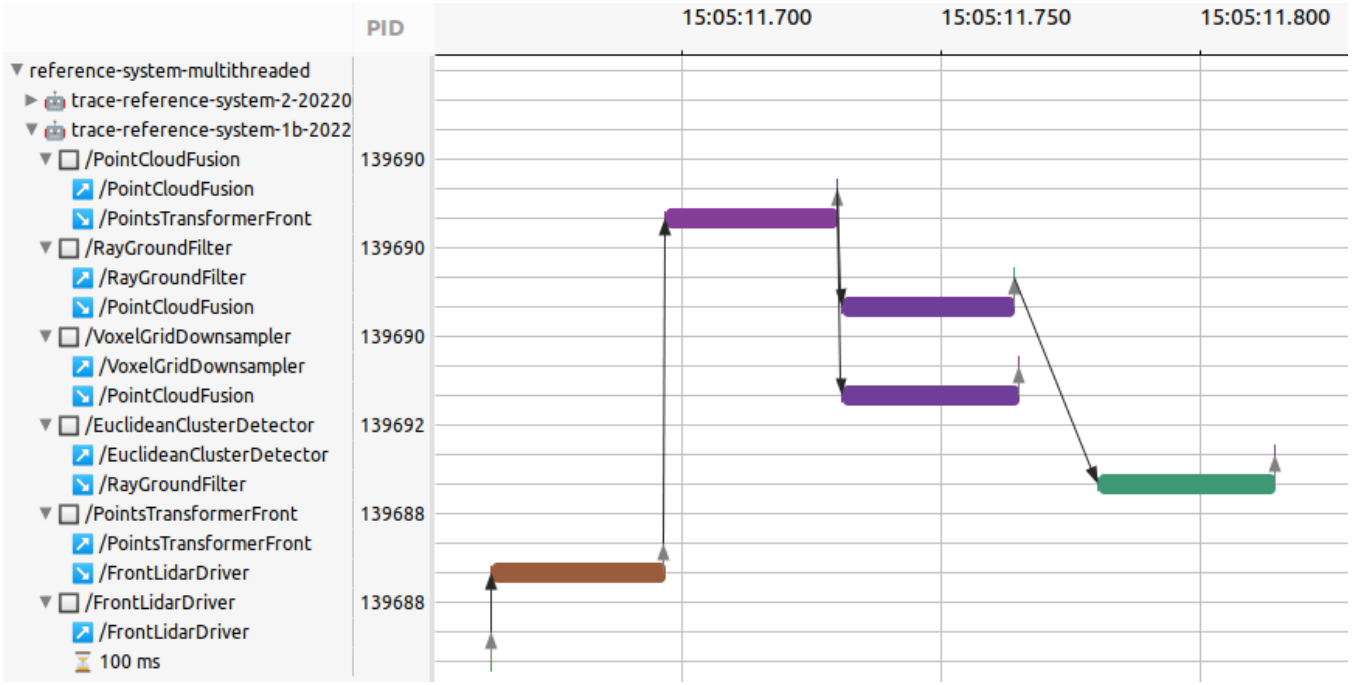


(a) End-to-end message flow graph. The initial root of the message flow graph is a 100 ms timer callback instance which publishes a /FrontLidarDriver message, while the main leaves of the graph are /VehicleInterface messages received by the VehicleDBWSystem node. The message flow graph includes direct and indirect causal links, including both periodic asynchronous and partial synchronous links.

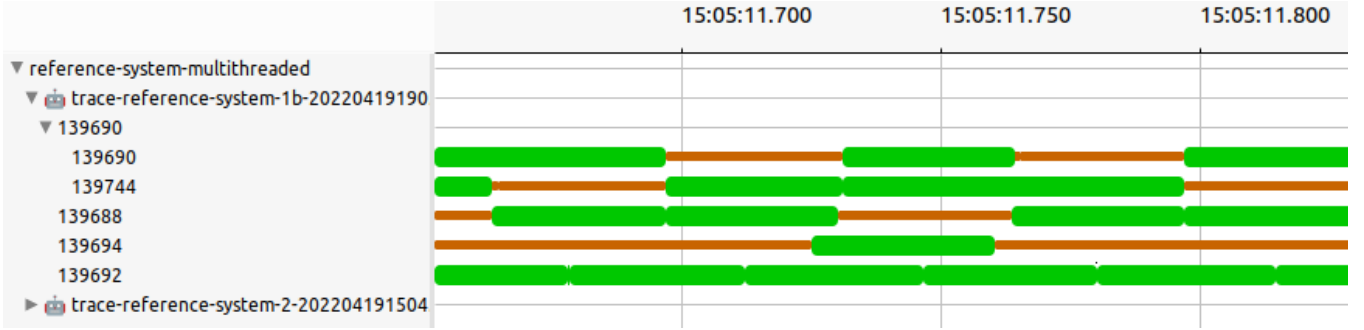


(b) State of all executor instances over time for the same time range as (a). Executor instances of the first host (lower half) are busier than the executor instances of the second host (upper half).

Fig. 9. Autware reference system (a) message flow result example and (b) executor state for the same time range.



(a) Partial message flow graph; segments for the second host are hidden. Callbacks for the same `/PointCloudFusion` message received by the `/RayGroundFilter` and `/VoxelGridDownsampler` nodes are executed concurrently.



(b) State of executor instances for the first host over time for the same time range as (a). Process 139690 has two executor threads (139690 and 139744), while other processes use single-threaded executors.

Fig. 10. Autoware reference system (a) message flow and (b) executor state for the same time range, showing the impact of a multi-threaded executor instance.

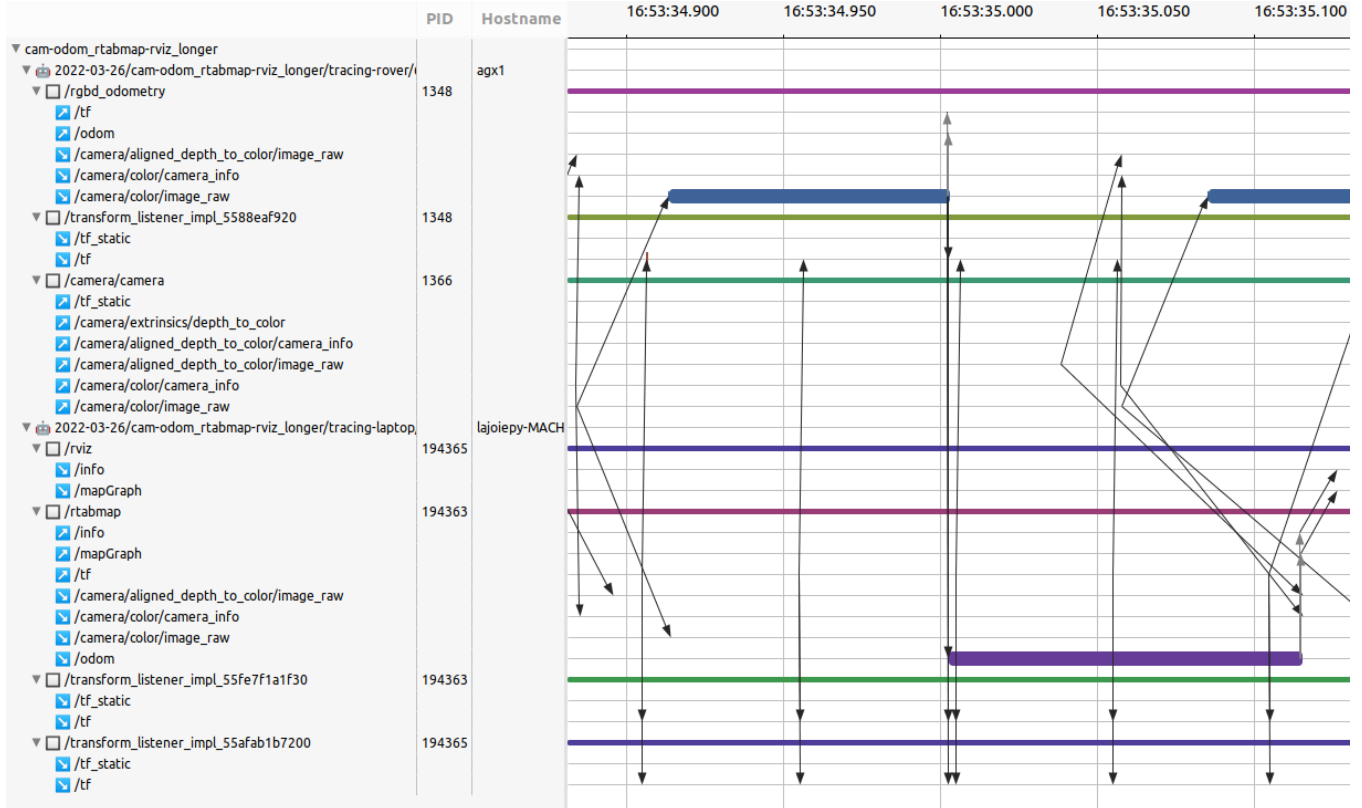
for process 189847 is always busy, which could explain why the callback for the `/PointsTransformerFront` message under the `/PointCloudFusion` node happens long after the message was published. Also, the `/VoxelGridDownsampler`, `/PointCloudFusion`, and `/RayGroundFilter` nodes are all on the same process and thus share the same single-threaded executor instance. The callbacks for the same `/PointCloudFusion` message under two different nodes thus cannot be processed at the same time. This directly affects the end-to-end latency, as our method helps highlight.

In this case, the computation graph distribution could be improved: nodes could be split over more processes, and better executor designs could be used. For instance, as demonstrated in Fig. 10, to allow the callbacks of the `/VoxelGridDownsampler` and `/RayGroundFilter` nodes to run simultaneously, we can use a multi-threaded

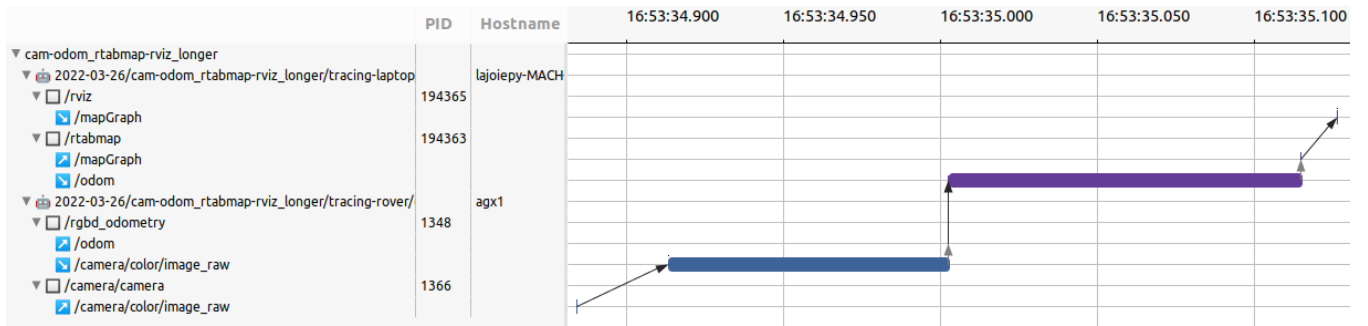
executor with 2 threads. As shown in Fig. 10b, there are two executor threads for the corresponding process (139690 and 139744). The subscription callbacks can then run concurrently, as shown in Fig. 10a. The end-to-end latency is hence reduced from 370 ms in the previous example to 298 ms in this example. Our method can therefore be used to compare or study the impact of proposed executor designs in order to address the open problems summarized in Section II.

B. RTAB-Map

For our second experiment, we distribute the RTAB-Map [52] simultaneous localization and mapping (SLAM) system over two computers and trace it. One host, an AgileX Scout Mini equipped with an NVIDIA Jetson AGX Xavier and an Intel RealSense D435i camera, runs the camera driver node and odometry node. The other host, a laptop on the same wireless network, runs the main SLAM node and rviz



(a) Display of intermediate execution representation data: subscription callbacks, message publications, transport links (black arrows), and direct causal links (gray arrows). A camera driver node (`/camera/camera`) and odometry node (`/rgbd_odometry`) run on the the first host (top half) along with a transform listener node (see Section IV-C). The RTAB-Map (`/rtabmap`) and rviz (`/rviz`) nodes run on the second host (bottom half) along with two transform listener nodes.



(b) End-to-end message flow graph for the main RTAB-Map computation pipeline, which goes through, in order: camera driver node (`/camera/camera`), odometry node (`/rgbd_odometry`), RTAB-Map node (`/rtabmap`), and rviz node (`/rviz`). The graph was generated starting from the subscription callback for a `/mapGraph` message received by the `/rviz` node, which is the last segment of the computation pipeline. Therefore, the message flow only goes backward from there to the root `/camera/color/image_raw` message published by the `/camera/camera` node on the other host. Looking at Fig. 11a, we know that generating the message flow graph starting from the initial `/camera/color/image_raw` message would result in a message flow graph with one-to-many links, similar to Fig. 9a.

Fig. 11. RTAB-Map (a) callback instances and message publications along with transport and direct links, and (b) message flow result for the main computation pipeline for the same time range.

to visualize the resulting map. To obtain synchronized traces, we synchronize the clocks of the two hosts using NTP [51].

A section of the trace is represented in Fig. 11. Fig. 11a shows all callback instances, message publications, transport links, and direct links from the intermediate execution representation database for this time range. Fig. 11b shows an end-to-end message flow graph for the main computation pipeline. Using Trace Compass, we find that the end-to-end latency is 242.3 ms; the duration of the first subscription callback (odometry computation) is 89.5 ms, and the duration of the second subscription callback (RTAB-Map) is 112.5 ms. As seen shortly after the 16:53:35.000 time mark under the `/rgb_d_odometry` node, a `/tf` message is published during a subscription callback instance. As mentioned in Section IV-C, the message is received by a special subscription, a transform listener, under the same process (PID 1348). However, this `/tf` message is actually only intended for the two transform listeners on the other hosts (PIDs 194363 and 194365), and does not cause a loop since there are no further segments after these `/tf` messages are received. Our method could be improved to model and detect indirect causal links after `/tf` messages.

VII. RUNTIME OVERHEAD EVALUATION

Since runtime overhead should be minimal to avoid perturbing an application [25], we also evaluate the overhead of execution data collection. In previous work, we demonstrated that `ros2_tracing` [12] introduces a mean end-to-end latency overhead of 0.0033 ms for a single message publication (i.e., publisher to subscription). Since the instrumentation proposed in [12] includes 10 tracepoints in the publish-subscribe hot path, this is comparable to a runtime cost per LTTng userspace tracepoint of 158 ns, as measured by [37]. Depending on the DDS implementation, our proposed method adds either 2 or 3 additional tracepoints to the hot path. However, the systems presented in Section VI include between 3 and 8 message transport instances, and have end-to-end latencies ranging from 240 ms to 370 ms. Furthermore, as discussed in [12], the combination of a high-level ROS 2 scheduler on top of the OS scheduler and networking stack introduces a lot of variability. Therefore, we expect the end-to-end latency overhead for real applications to be small, especially when compared to the absolute latency.

We create a computation graph similar to the experiments in Section VI, with 5 one-to-one message transport instances and an expected end-to-end latency of approximately 260 ms. We use an Intel i7-3770 (3.40 GHz) 4-core CPU, 8 GB RAM system with Ubuntu 20.04.2, and disable power-saving features. We run the computation graph at 10 Hz for 20 minutes first without and then with tracing to compare the runtime impact of tracing on the end-to-end latency. By comparing the latencies for each case, shown in Fig. 12, we obtain a difference of means of 0.1597 ms and a difference of medians of 0.0521 ms. This end-to-end latency overhead is small compared to a total latency of 260 ms; we therefore consider it suitable for real applications. Furthermore, this value is

within an order of magnitude of overhead values extrapolated from results by [12] and [37], respectively 0.0215 ms and 0.0103 ms. Finally, as mentioned previously, we expect this overhead to be less noticeable – and challenging to actually measure – on more complex ROS 2 systems.

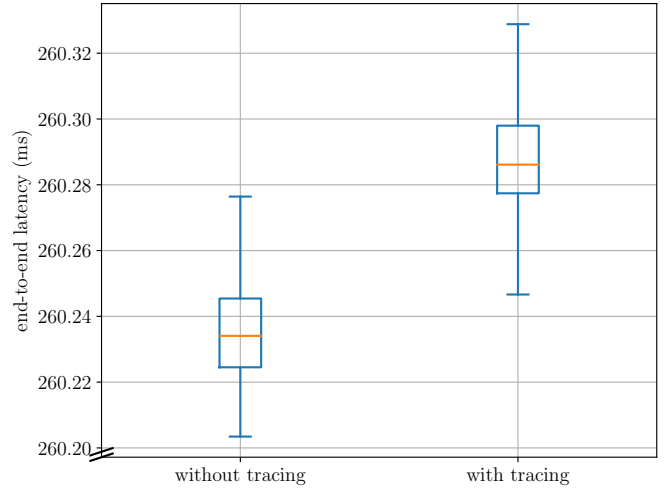


Fig. 12. End-to-end latency comparison, without tracing (left) and with tracing (right).

VIII. FUTURE WORK

Many improvements and additions could be made to our proposed message flow analysis method and executor state visualization.

First, our method can easily be extended with other indirect causal links. Moreover, transport links could be split into actual network transport time and a time delay between message reception by DDS and processing by the executor. As mentioned in Section VI-A, it would help highlight delays in callback executions when the executor is busy processing other callbacks. This would require additional instrumentation in the underlying DDS middleware. Other executor types, such as the multi-threaded executor or other executors presented in previous work [6], [7], [8], [9], could also be instrumented and supported for the executor state visualization. Additionally, as mentioned in Section IV-C and shown in Section VI-B, special subscriptions like the transform listener could be supported to be able to detect valid message flow segments resulting from received `/tf` messages. Finally, it would also be interesting to extend this work to include ROS 2 services and actions.

Furthermore, our message flow analysis method could be further extended into a critical path analysis [40]. Fundamentally, indirect causal links (shown in red in Fig. 6, Fig. 7, and Fig. 9a) are wait intervals. Indeed, as mentioned in Section IV-B.2, the duration of a periodic asynchronous link depends on the period of the timer, and the duration of a partial synchronous link depends on the other input messages. These wait segments could thus be recursively replaced with the actual cause of the wait, as is done by Giraldeau and Dagenais [13] using kernel-level wait primitives for wait dependencies across a distributed system.

Similarly, the message flow graph could be augmented with other information. For example, as mentioned in Section III-B, the duration of a message publication call can be broken down into `rclecpp`, `rcl`, and DDS time. The required information is already collected using `ros2_tracing` and available in the intermediate execution representation database. Finally, other metrics, such as message publication or reception rate and executor usage over time, could be extracted from the database and displayed alongside our proposed visualizations.

IX. CONCLUSION

In conclusion, modern robotic systems are built as distributed computation graphs, using the publish-subscribe paradigm and frameworks such as ROS 2. However, there are open problems with the higher-level scheduling of tasks performed by the ROS 2 executor, which can affect performance.

We presented a low-overhead method for extracting and visualizing the flow of a message across a distributed ROS 2 system. Our novel approach can detect exchanges of messages across distributed systems, and also introduces simple annotations for indirect causal message links. This is achieved without needing to modify the applications themselves. While we only define two types of indirect causal links, our work can be extended to consider other types of message links as well. This would help ensure the validity of the message flow graphs generated for more complex systems. Combined with a visualization of the state of the executor instances over time, our work is useful for optimizing both application layers and ROS 2 itself.

Finally, the underlying intermediate execution representation data can be leveraged for further analyses. Furthermore, the message flow graph can also be extended with more information, and can be expanded into a critical path analysis, by recursively resolving wait dependencies resulting from indirect causal links.

REFERENCES

- [1] P.-Y. Lajoie, B. Ramtoul, F. Wu, and G. Beltrame, "Towards collaborative simultaneous localization and mapping: a survey of the current research landscape," *Field Robotics*, vol. 2, no. 1, pp. 971–1000, 2022.
- [2] R. K. Dewangan, A. Shukla, and W. W. Godfrey, "Survey on prioritized multi robot path planning," in *2017 IEEE International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*, 2017, pp. 423–428.
- [3] Z. Yan, N. Jouandeau, and A. A. Cherif, "A survey and analysis of multi-robot coordination," *International Journal of Advanced Robotic Systems*, vol. 10, no. 12, p. 399, 2013. [Online]. Available: <https://doi.org/10.5772/57313>
- [4] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.
- [5] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.
- [6] R. Lange, "Callback-group-level executor for ros 2," in *ROSCon Madrid 2018*. Open Robotics, September 2018. [Online]. Available: <https://vimeo.com/292707644>
- [7] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 251–263.
- [8] J. Staschulat, I. Lütkebohle, and R. Lange, "The rclc executor: Domain-specific deterministic scheduling mechanisms for ros applications on microcontrollers: work-in-progress," in *2020 International Conference on Embedded Software (EMSOFT)*. IEEE, 2020, pp. 18–19.
- [9] J. Staschulat, R. Lange, and D. N. Dasari, "Budget-based real-time executor for micro-ros," *arXiv preprint arXiv:2105.05590*, 2021.
- [10] K. Nishimura, T. Ishikawa, H. Sasaki, and S. Kato, "Raplet: Demystifying publish/subscribe latency for ros applications," in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2021, pp. 41–50.
- [11] Z. Li, A. Hasegawa, and T. Azumi, "Autoware.perf: A tracing and performance analysis framework for ros 2 applications," *Journal of Systems Architecture*, vol. 123, p. 102341, 2022.
- [12] C. Bédard, I. Lütkebohle, and M. Dagenais, "ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ros 2," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6511–6518, 2022.
- [13] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, 2015.
- [14] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.
- [15] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications," *arXiv preprint arXiv:1809.02595*, 2018.
- [16] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Rönnau, and R. Dillmann, "Distributed and synchronized setup towards real-time robotic control using ros2 on linux," in *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2020, pp. 1287–1293.
- [17] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings*. IEEE, 2003, pp. 200–206.
- [18] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, and G. Fettweis, "Latency analysis of ros2 multi-node systems," in *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. IEEE, 2021, pp. 1–7.
- [19] Z. Jiang, Y. Gong, J. Zhai, Y.-P. Wang, W. Liu, H. Wu, and J. Jin, "Message passing optimization in robot operating system," *International Journal of Parallel Programming*, vol. 48, no. 1, pp. 119–136, 2020.
- [20] Y.-P. Wang, W. Tan, X.-Q. Hu, D. Manocha, and S.-M. Hu, "Tzc: Efficient inter-process communication for robotics middleware with partial serialization," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 7805–7812.
- [21] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Rönnau, and R. Dillmann, "Performance evaluation of real-time ros2 robotic control in a time-synchronized distributed network," in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2021, pp. 1670–1676.
- [22] Apex.AI, "performance.test." [Online]. Available: <https://gitlab.com/ApexAI/performance.test>
- [23] iRobot, "irobot ros 2 performance evaluation framework." [Online]. Available: <https://github.com/irobot-ros/ros2-performance>
- [24] T. Witte and M. Tichy, "Inferred interactive controls through provenance tracking of ros message data," in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*. IEEE, 2021, pp. 67–74.
- [25] B. Gregg, *Systems Performance: Enterprise and the Cloud*, 2nd ed. Pearson, 2020.
- [26] "Executors." [Online]. Available: <https://docs.ros.org/en/rolling/Concepts/About-Executors.html>
- [27] J. Peeck, J. Schlato, and R. Ernst, "Online latency monitoring of time-sensitive event chains in safety-critical applications," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 539–542.
- [28] D. Casini, T. Bläß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ros 2 processing chains under reservation-based

- scheduling,” in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- [29] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, “Response time analysis and priority assignment of processing chains on ros2 executors,” in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 231–243.
 - [30] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, “Automatic latency management for ros 2: Benefits, challenges, and open problems,” in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 264–277.
 - [31] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, “A ros 2 response-time analysis exploiting starvation freedom and execution-time variance,” in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 41–53.
 - [32] Y. Yang and T. Azumi, “Exploring real-time executor on ros 2,” in *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2020, pp. 1–8.
 - [33] ROS 2 Real-Time Working Group, “Reference system.” [Online]. Available: <https://github.com/ros-realtime/reference-system>
 - [34] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monroy, T. Ando, Y. Fujii, and T. Azumi, “Autoware on board: Enabling autonomous vehicles with embedded systems,” in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.
 - [35] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, “An open approach to autonomous vehicles,” *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.
 - [36] M. Desnoyers and M. R. Dagenais, “The ltng tracer: A low impact performance and behavior monitor for gnu/linux,” in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.
 - [37] M. Gebai and M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–33, 2018.
 - [38] I. Lütkebohle, “Determinism in ros – or when things break /sometimes/ and how to fix it...,” in *ROSCon Vancouver 2017*. Open Robotics, September 2017. [Online]. Available: <https://doi.org/10.36288/ROSCon2017-900789>
 - [39] Bosch Corporate Research, “Ros 1 tracertools.” [Online]. Available: https://github.com/boschresearch/ros1_tracertools
 - [40] C.-Q. Yang and B. P. Miller, “Critical path analysis for the execution of parallel and distributed programs,” in *The 8th International Conference on Distributed*. IEEE Computer Society, 1988, pp. 366–367.
 - [41] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, “Estimating global time in distributed systems,” in *ICDCS*, vol. 87, 1987, pp. 299–306.
 - [42] B. Poirier, R. Roy, and M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, 2010.
 - [43] M. Jabbarifar and M. Dagenais, “Liana: Live incremental time synchronization of traces for distributed systems analysis,” *Journal of network and computer applications*, vol. 45, pp. 203–214, 2014.
 - [44] L. Gelle, N. Ezzati-Jivan, and M. R. Dagenais, “Combining distributed and kernel tracing for performance analysis of cloud applications,” *Electronics*, vol. 10, no. 21, p. 2610, 2021.
 - [45] A. Santos, A. Cunha, and N. Macedo, “Static-time extraction and analysis of the ros computation graph,” in *2019 Third IEEE international conference on robotic computing (IRC)*. IEEE, 2019, pp. 62–69.
 - [46] “tracertools.analysis.” [Online]. Available: <https://gitlab.com/ros-tracing/tracertools.analysis>
 - [47] C. Bédard, “Message flow analysis for ros through tracing,” 2019. [Online]. Available: <https://christophebedard.com/ros-tracing-message-flow/>
 - [48] eProsima, “Fast dds.” [Online]. Available: <https://github.com/eProsima/Fast-DDS>
 - [49] “Eclipse cyclone dds.” [Online]. Available: <https://github.com/eclipse-cyclonedds/cyclonedds>
 - [50] “Eclipse trace compass.” [Online]. Available: <https://www.eclipse.org/tracecompass/>
 - [51] D. L. Mills, “Internet time synchronization: the network time protocol,” *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.
 - [52] M. Labbé and F. Michaud, “Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation,” *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.