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

Christophe Bédard

ROS 2 Hardware Acceleration Working Group meeting
May 31st, 2022

Polytechnique Montréal
DORSAL Laboratory
🇨🇦

POLYTECHNIQUE MONTRÉAL
TECHNOLOGICAL UNIVERSITY

Summary

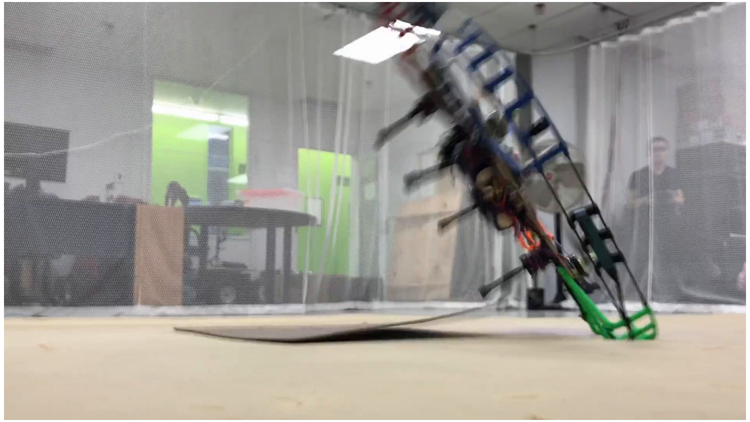
1. Introduction, context
2. Tracing
3. `ros2_tracing`
4. Tracing ROS 2
5. Message flow analysis
6. ROS 2 executor
7. Runtime overhead evaluation
8. Conclusion and future work
9. Questions

Introduction

- Robotics
  - Commercial or industrial applications
  - Safety-critical applications
  - Can be distributed and connected over a network
- Key elements
  - Publish-subscribe: modularity
  - Higher-level scheduling of tasks is challenging
  - Performance targets, real-time constraints

Context

- Debugging tools
  - Debugging: GDB
  - Logs: usually human-readable strings
- Distributed systems
  - How to analyze a distributed system as a whole?
- Real-time, production
- Observability problems
  - Observer effect
  - Have to avoid influencing or affecting the application
  - Especially when extracting timing-related data



Tracing

- Goal: gather runtime execution information
- Low-level information
  - Raw/binary data
  - Needs to be processed
- Useful when issues are hard to reproduce
- We want to minimize the overhead!
  - To use in production
  - Observer effect
- LTTng
  - High-performance, low-overhead tracer
- Workflow (static instrumentation)
  - Instrument an application with trace points
  - Configure tracer, run the application
  - Trace points generate events (information)
  - Events make up a trace

ros2_tracing

- gitlab.com/ros-tracing/ros2_tracing
- Collection of tracing instrumentation and tools
  - Closely integrated into ROS 2
- Tools to instrument the core of ROS 2 with LTTng
  - `rclcpp`, `rcl`, `rmw` (`rmw_cyclonedds`, `rmw_fastrtps`)
- Tools to configure tracing with LTTng
  - Command: `ros2 trace`
  - Action for ROS 2 launch: `Trace`
- Paper
  - doi.org/10.1109/LRA.2022.3174346

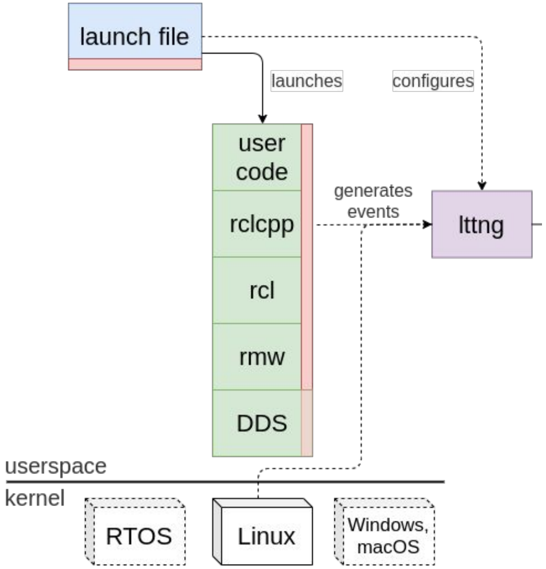


Figure 1. Instrumentation and general workflow.

`Trace` action for ROS 2 launch

- Starts tracing when launched
- Stops tracing when exiting
- Great for complex systems with multiple nodes

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from tracetools_launch.action import Trace
def generate_launch_description():
    return LaunchDescription([
        Trace(
            session_name='ros2-session',
            events_kernel=['sched_switch'],
            events_ust=['ros2:rclcpp_publish', 'ros2:*'],
        ),
        Node(
            package='pkg',
            executable='exe',
        ),
    ])
```

`Trace` action for ROS 2 launch (2)

- Also available in XML and YAML launch files

```xml
<launch>
  <trace
    session-name="ros2-session"
    events-kernel="sched_switch"
    events-ust="ros2:rclcpp_publish ros2:*"
  />
  <node pkg="pkg" exec="exe" />
</launch>
```

```yaml
launch:
- trace:
    session-name: ros2-session
    events-kernel: sched_switch
    events-ust: ros2:rclcpp_publish ros2:*
- node:
    pkg: pkg
    exec: exe
```

Instrumentation

- Design principles
  - Want information about each layer & the interaction between them
  - However, layers make it hard to get the full picture
  - Need to gather small bits of information here and there
  - Put it all together offline or externally
- Applies to most layers
  - `rclcpp`, `rcl`, `rmw`, DDS
- Object instances
  - Node, publisher, subscription, timer
- Events
  - Callback execution (subscription, timer)
  - Message publication
  - Message taking (for subscription callbacks)
  - Executor state (executing, waiting, etc.)
  - Etc.

Trace data processing

- Trace Compass
  - Trace data analysis framework with built-in analyses
- Distributed systems
  - Combine traces, usually after execution
  - Synchronize traces using NTP, PTP, or offline sync using Trace Compass
- Modeling ROS 2 objects and instances from trace data
  - Using pointers as unique IDs in payload of most tracepoints
  - Multiple processes: combine with PID
  - Multiple hosts: combine with host ID
- Can use this pre-processed data to extract further metrics or build other analyses
  - Simple queries

Message flow

- Graph of the path of a message across a distributed ROS system
  - From a root to a leaf of the computation graph (DAG)
  - Message → message → message → etc.
- Shows what happened and when it happened

- First prototype for ROS 1 back in 2019
  - christophebedard.com/ros-tracing-message-flow/
- Several limitations
  - Unreliable tracking of messages across the network (pub → sub)
  - Only considered direct 1-to-1 links between messages

Message flow analysis

- Graph of the path of a message across a distributed ROS 2 system
  - Build by combining multiple segments and links
- Subscription and timer callbacks
- Message publication instances
- Transport links
- Causal message links

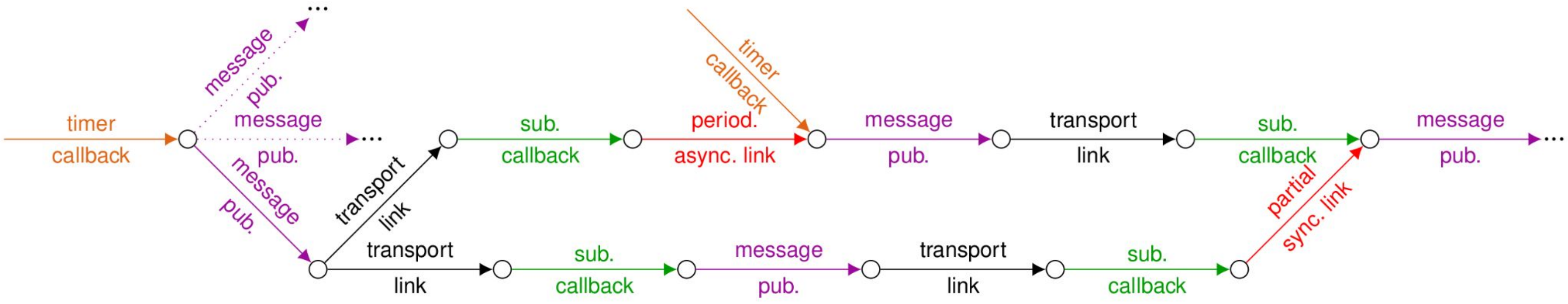


Figure 2. Simplified representation of a message flow graph.

Transport link

- Link between publication instance and corresponding subscription callback
- Includes more than just network time
  - Delay between message reception and callback execution
- One-to-many link
  - 1 publisher → N subscriptions

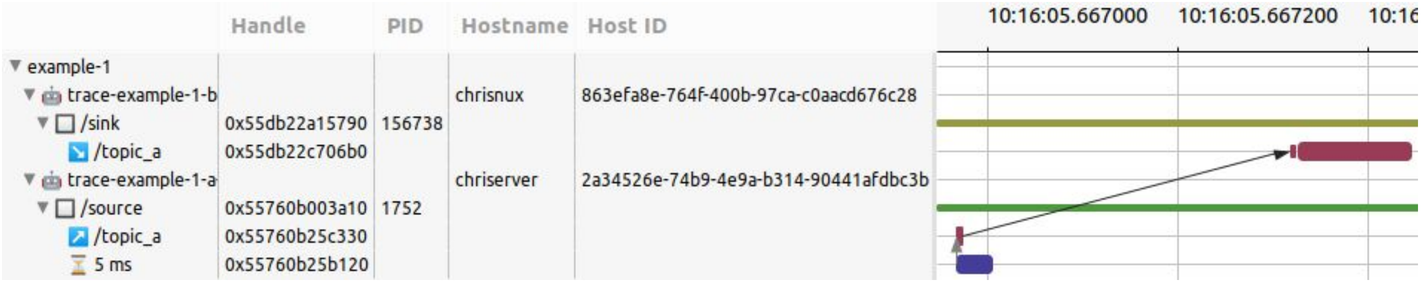


Figure 3. Transport link from one host to another host.

Causal message link

- Link between N input messages and M output messages
  - Subscription → … → publisher
- Causality is primarily based on message data
  - Data of input message is used to generate output message
  - No strict time constraint
- Direct or indirect

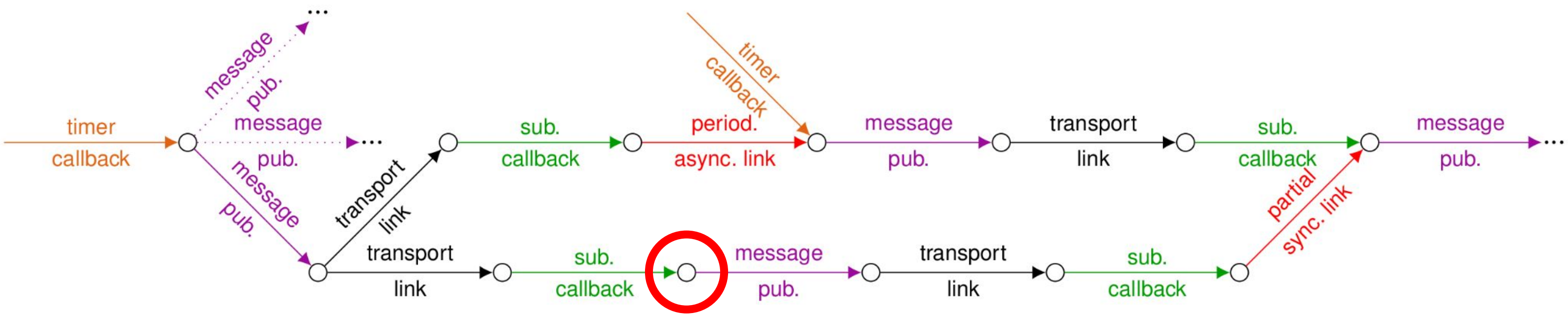


Figure 4. Message flow graph with causal message links in red.

Direct causal link

- Direct causal link
  - Message publication during subscription callback for message
- Can be inferred automatically
  - No need for additional information or instrumentation

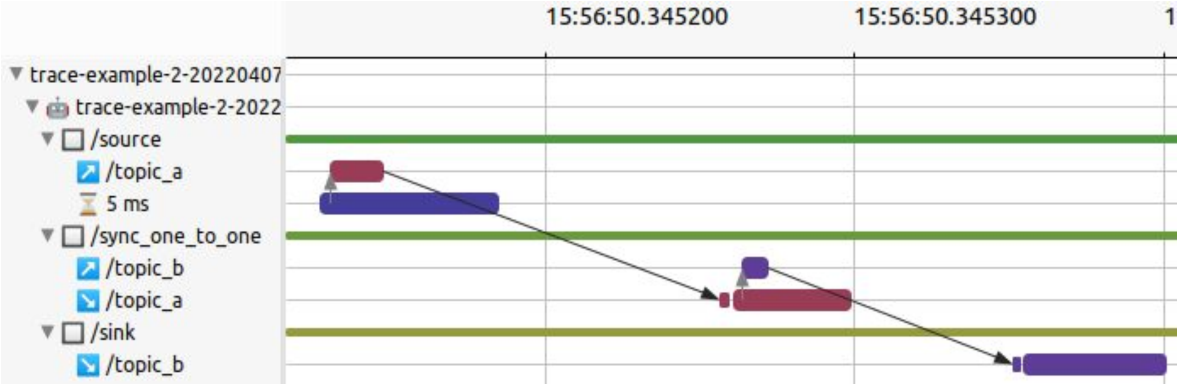


Figure 5. Direct causal link.

Indirect causal link

- User-level code is difficult to include in our execution model
  - Above ROS 2 API
  - For example: use of message caches
- Leads to indirect causal links
  - Asynchronous links
  - Many-to-many links
- Requires additional user-level annotation
  - Collected using simple tracepoints
  - Included in model of objects and instances

Indirect causal link - periodic asynchronous

- Messages are received & cached
- An output message is periodically published

Figure 6. Periodic asynchronous causal link.

Indirect causal link - partial synchronous

- Messages are received & cached
- An output message is published if all caches are full

Figure 7. Partial synchronous causal link.

Message flow graph

- Resulting message flow graph
  - Autoware reference system
  - Split over multiple processes
  - Distributed over 2 hosts
- We can extract
  - End-to-end latency
  - Intermediate latencies
- We can visually understand the execution
  - Find bottlenecks
- Compare with other execution data
  - Linux kernel (scheduling, I/O, etc.)
  - Other application data

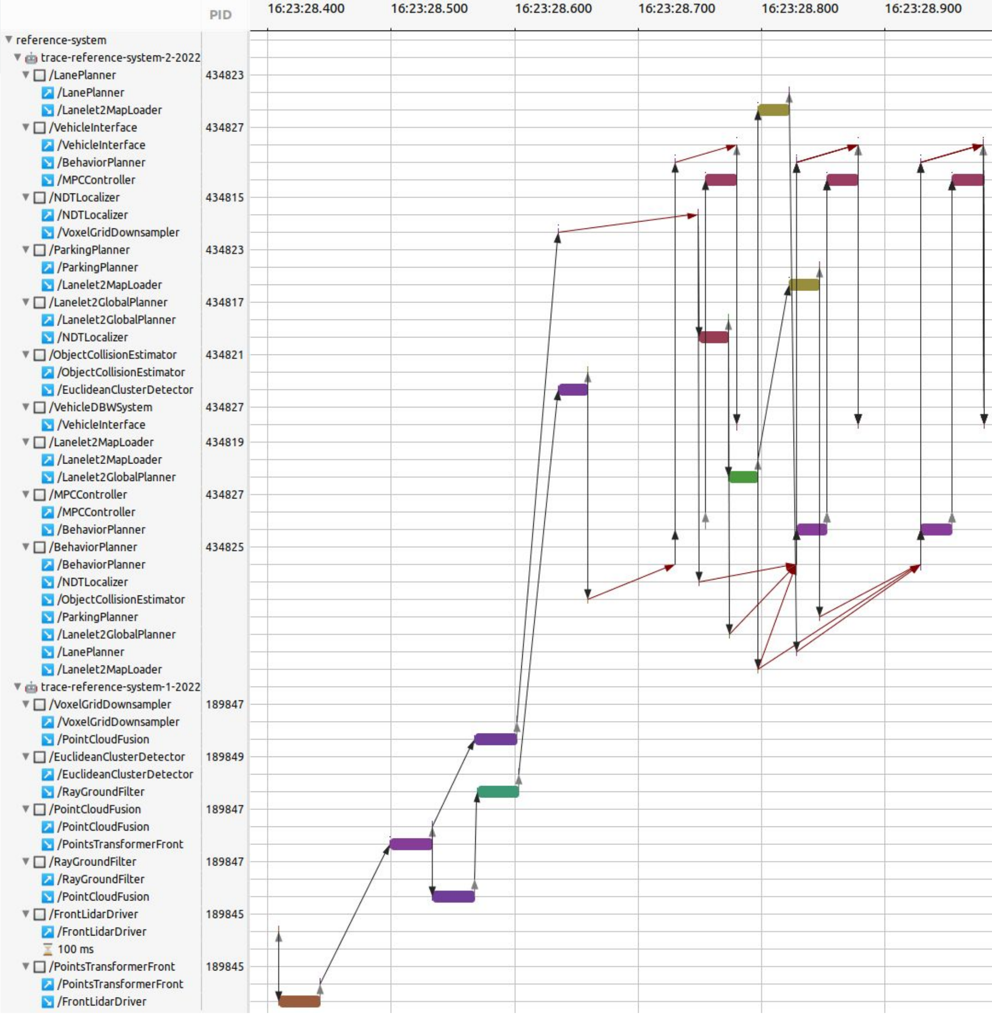


Figure 8. Message flow analysis result example.

ROS 2 executor & scheduling

- Executor
  - High-level task scheduler
  - Fetches new messages from underlying middleware
  - Executes user-provided timer and subscription callbacks
- Challenges
  - Scheduling on top of the OS scheduler can be inefficient & non-deterministic
- Possible solutions
  - Other executor designs, depending on the application/requirements
  - Optimize scheduling policies and priorities
- Want to study and compare executors
  - And optimize overall application performance

Executor state

- Green/orange: executing/waiting for new messages or timer trigger
- Some executor instances are busier than others
- Causes message processing delays, leads to bottlenecks

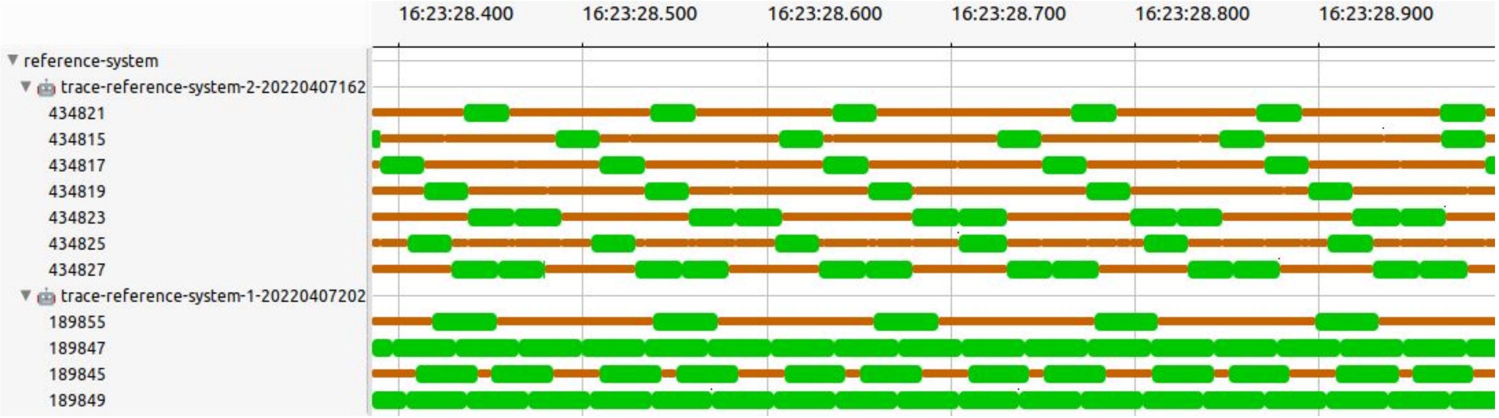


Figure 9. View showing state of executor instances (threads) over time.

Example: multi-threaded executor

- Use multi-threaded executor for busier process
- Could also use other executor designs

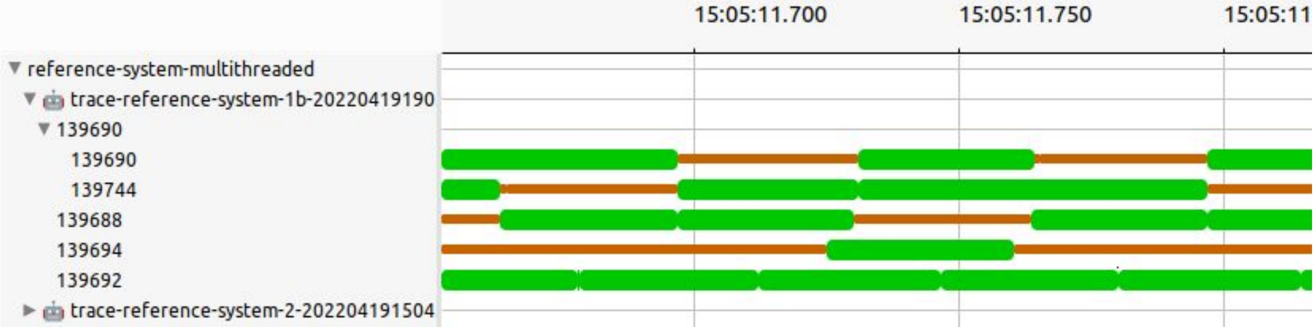


Figure 10. Executor state over time, with multi-threaded executor instance.

Example: multi-threaded executor (2)

- Callbacks sharing the same executor are then less of a bottleneck

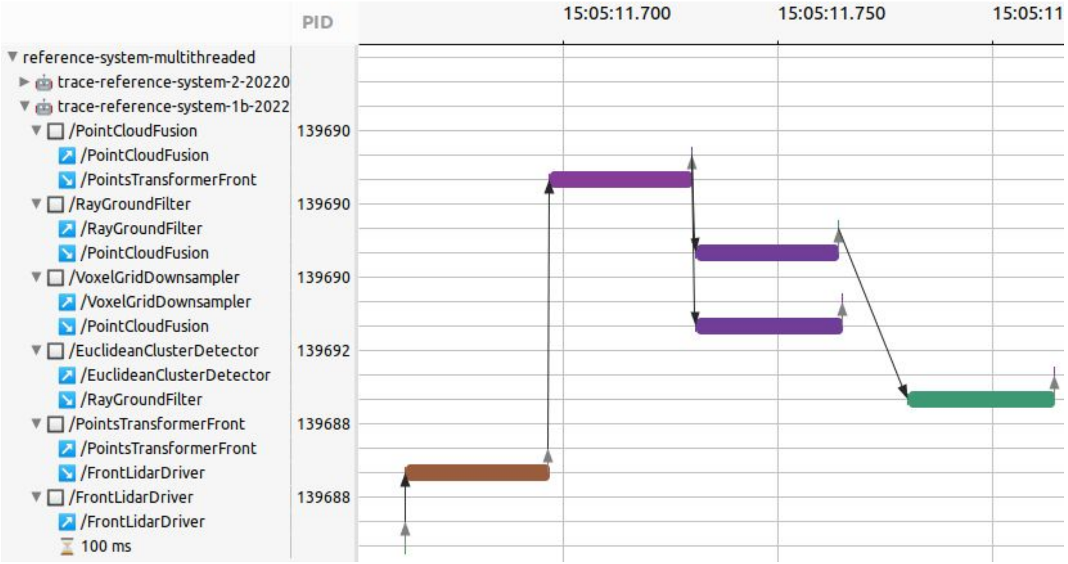


Figure 11. Message flow sub-graph, with multi-threaded executor instance.

Runtime overhead evaluation

- Extrapolating from previous overhead results
  - Should be very small
- Execute pipeline of nodes, without & with tracing
  - Total end-to-end latency of ~260 ms
- Overhead is the difference
  - Difference of means   : 0.1597 ms
  - Difference of medians: 0.0521 ms
  - < 0.06%
- Likely challenging to measure on more complex systems

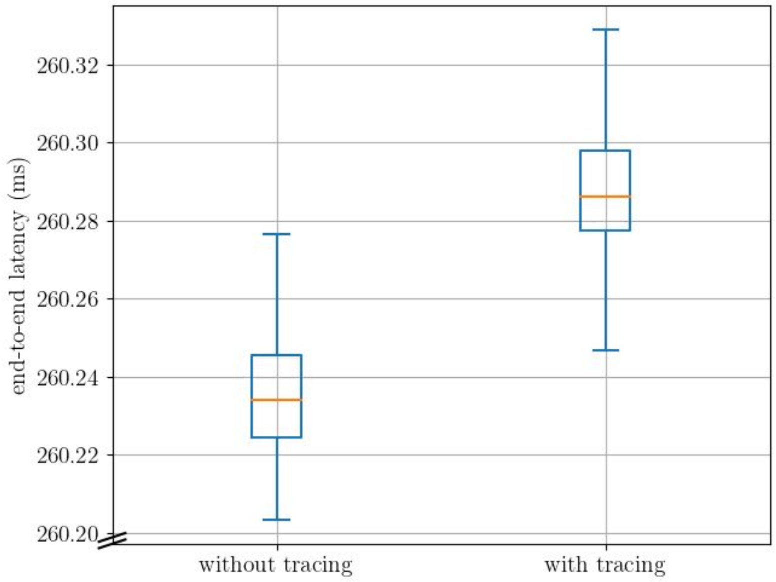


Figure 12. End-to-end latency comparison (note Y axis scale).

Conclusion and future work

- Tracking messages across nodes and hosts
    - Building a message flow graph using this information
    - Using user-level annotation to find more complex indirect causal links
- Computing end-to-end latency
- Study and improve performance of an application and ROS 2 itself

- Future work
    - Resolve wait dependencies resulting from asynchronous causal links
    - Critical path analysis at the ROS 2 level
    - Augment graph with other information: application-level or kernel-level

Questions?

- Papers
  - Message Flow Analysis with Complex Causal Links for Distributed ROS 2 Systems
    - Christophe Bédard, Pierre-Yves Lajoie, Giovanni Beltrame, Michel Dagenais
    - arxiv.org/abs/2204.10208
  - ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2
    - Christophe Bédard, Ingo Lütkebohle, Michel Dagenais
    - ieeexplore.ieee.org/document/9772997
    - arxiv.org/abs/2201.00393
- Links
  - gitlab.com/ros-tracing/ros2_tracing
  - github.com/christophebedard/ros2-message-flow-analysis

Message flow paper

`ros2_tracing` paper