Tracing ROS 2 with `ros2_tracing`

Christophe Bédard

ROS World 2021
October 20, 2021

DORSAL Laboratory
Polytechnique Montréal 🇨🇦

POLYTECHNIQUE MONTRÉAL
TECHNOLOGICAL UNIVERSITY

Plan

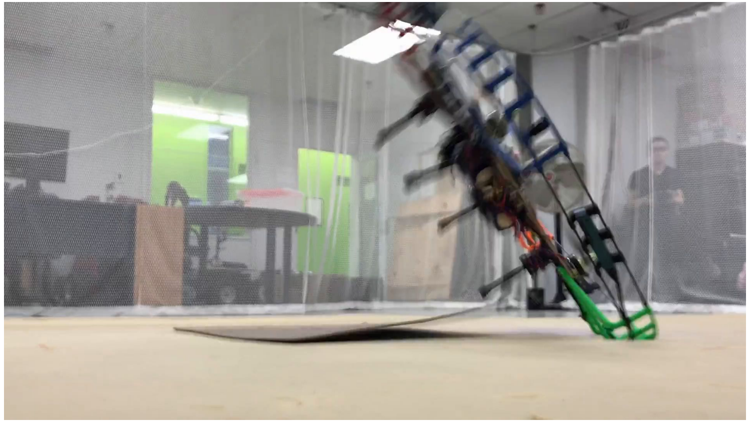
1. Introduction
2. Context
3. Tracing & LTTng
4. `ros2_tracing`
5. Analysis
6. Demo
7. Conclusion
8. Questions

Introduction

- Robotics
    - Many different types of applications
    - Toys, commercial applications, industrial applications
    - Safety-critical systems
- ROS 2
    - New capabilities
    - Distributed systems
    - Real-time constraints

Context

- Debugging and diagnostics tools
    - Debugging: GDB
    - Logs: ROS, `printf()`
    - Introspection: rqt_graph
    - Others: diagnostic_aggregator, libstatistics_collector
- Distributed systems
    - How to analyze a distributed system?
- Real-time, production
- Observability problems
    - Observer effect
    - Have to avoid influencing or affecting the application
- Observing an application's (lack of) determinism
    - See Ingo Lütkebohle's ROSCon 2017 talk about determinism in ROS: doi.org/10.36288/ROSCon2017-900789
    - See also his talk at the ROSCon 2019 real-time workshop: apex.ai/roscon2019



Tracing

- Goal: gather runtime execution information
  - Low-level information
  - OS and application
- Useful when issues are hard to reproduce
- Many different tracers with different features
  - LTTng, perf, Ftrace, eBPF, DTrace, SystemTap, Event Tracing for Windows, etc.
- Workflow (static instrumentation)
  - Instrument an application with trace points
  - Configure tracer, run the application
  - Trace points generate events (information)
  - Events make up a trace
- We want to minimize the overhead!
  - To use in production
  - Observer effect

LTTng

- lttng.org
- High-performance tracer
  - Low overhead
  - Userspace tracer + kernel tracer
- Linux only
- Instrumentation
  - Built into the Linux kernel (e.g., sched_switch, net_dev_queue)
  - Added statically to your application
  - Or by `LD_PRELOAD`ing libraries
- Trace data processing
  - Online (live)
  - Offline (more common & simpler)

LTTng - example

- Creating a tracing session, enabling trace events, tracing our application, and stopping

```
$ lttng create ros2-session
$ lttng enable-event --kernel sched_switch
$ lttng enable-event --userspace ros2:rclcpp_publish
$ lttng enable-event --userspace ros2:*
$ lttng start
$ ros2 run package executable
$ lttng stop && lttng destroy
```

LTTng - example (2)

- Viewing the trace: each trace event has a name, timestamp, payload

```
$ babeltrace ros2-session/
sched_switch: { cpu_id = 1 }, { prev_comm = "swapper/1", prev_tid = 0, prev_prio = 20, prev_state = (
   "TASK_RUNNING" : container = 0 ), next_comm = "test_ping", next_tid = 416160, next_prio = 20 }
ros2:callback_start: { cpu_id = 1 }, { callback = 0x541190, is_intra_process = 0 }
ros2:rclcpp_publish: { cpu_id = 1 }, { message = 0x5464F0 }
ros2:rcl_publish: { cpu_id = 1 }, { publisher_handle = 0x541A40, message = 0x5464F0 }
ros2:rmw_publish: { cpu_id = 1 }, { message = 0x5464F0 }
ros2:callback_end: { cpu_id = 1 }, { callback = 0x541190 }
```

ros2_tracing

- gitlab.com/ros-tracing/ros2_tracing
- Collection of tools
- Closely integrated into ROS 2
  - To promote use and adoption
  - Since ROS 2 Eloquent (2019)
  - Many improvements and additions since then
- Tools to instrument the core of ROS 2 with LTTng
  - `rclcpp`, `rcl`, `rmw` (`rmw_cyclonedds*`)
- Tools to configure tracing with LTTng
  - Command: `ros2 trace`
  - Action for ROS 2 launch: `Trace`

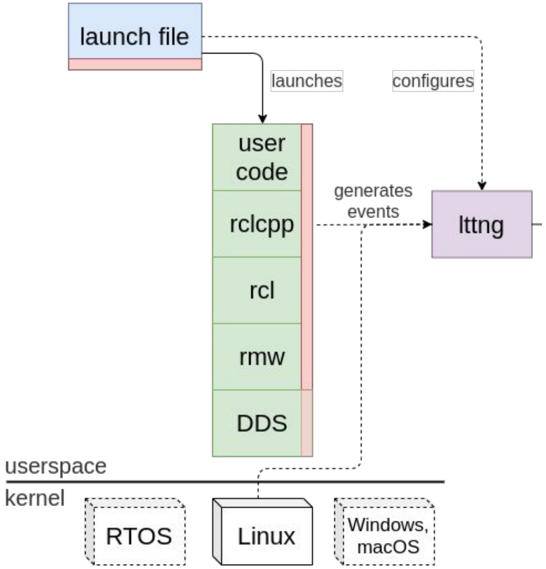


Figure 1. Instrumentation and general workflow.

Tools - `ros2 trace` command

- To easily start tracing
- Starting & stopping is done manually

```
$ ros2 trace \
    --session-name ros2-session \
    --kernel sched_switch \
    --ust ros2:rclcpp_publish ros2:*
writing tracing session to: /home/chris/.ros/tracing/ros2-session
press enter to start...
press enter to stop...
stopping & destroying tracing session
```

Tools - `Trace` action for ROS 2 launch

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

```
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',
        ),
    ])
```

Tools - `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

- Only on Linux, not included in the binaries
  - At least for now
  - Install LTTng and (re)build the `tracetools` package
- Instrumentation was designed to support multiple tracers
  - Other tracers and/or OSes, eventually
  - `rclcpp`, `rcl`, `rmw`, etc. → `tracetools` → LTTng
- 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
- Real-time
  - Applications generally have a non-real-time initialization phase
  - We take advantage of this to collect as much information up front
  - It lowers overhead in the real-time “steady state” phase

Instrumentation (2)

- Object instances
  - Node, publisher, subscription, timer
- Events
  - Callback execution (subscription, timer)
  - Message publication
  - Message taking (for subscription callbacks)
  - Lifecycle node state change
  - Internal executor phases
  - Etc.
- Applies to most layers
  - `rclcpp`, `rcl`, `rmw`
  - DDS (work in progress with Eclipse Cyclone DDS)

Instrumentation - example

- Ping node: a timer is used to publish a message periodically

```
ros2:rcl_node_init: { node_handle =  0x🚀, rmw_handle = 0x..., node_name = "test_ping" }
ros2:rcl_publisher_init: { publisher_handle =  0x🇩🇪, node_handle =  0x🚀, topic_name = "/ping", queue_depth = 10}

ros2:rcl_timer_init: { timer_handle =  0x⏲️, period = 500000000 }
ros2:rclcpp_timer_callback_added: { timer_handle =  0x⏲️, callback =  0x🤖 }
ros2:rclcpp_callback_register: { callback =  0x🤖, symbol = "std::_Bind<void (PingNode::*(PingNode*))()>" }
ros2:rclcpp_timer_link_node: { timer_handle =  0x⏲️, node_handle =  0x🚀 }

ros2:callback_start: { callback =  0x🤖, is_intra_process = 0 }
      ros2:rclcpp_publish: { message =  0x🇨🇦 }
      ros2:rcl_publish: { message =  0x🇨🇦, publisher_handle =  0x🇩🇪 }
      ros2:rmw_publish: { message =  0x🇨🇦 }
ros2:callback_end: { callback =  0x🤖 }
```

Overhead benchmark

- Goal: measure tracing overhead in a ROS 2 context
  - Mainly interested in a latency overhead
  - Tool: gitlab.com/ApexAI/performance_test
- Parameters
  - Inter-process: 1 pub → 1 sub
  - Publishing: 100 - 2000 Hz
  - Messages: 1 - 256 KB
  - Quality of service: reliable
  - Eclipse Cyclone DDS
- Setup
  - Ubuntu Server 20.04.2 with PREEMPT_RT (5.4.3-rt1)
  - Intel i7-3770 @ 3.40 GHz, 8 GB RAM
  - SMT/Hyper-threading disabled (4 cores, 1 thread/core)
  - SCHED_FIFO, RT priority 99, and other tuning
  - Run for 20 minutes, discard the first 5 seconds

Overhead benchmark - results

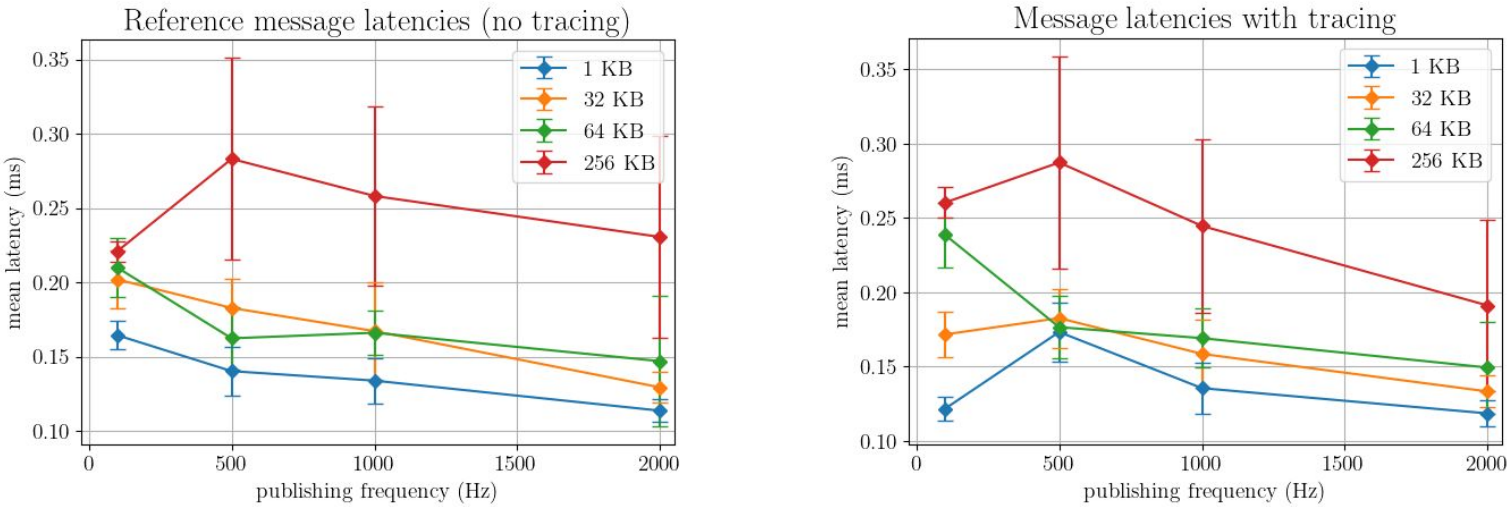
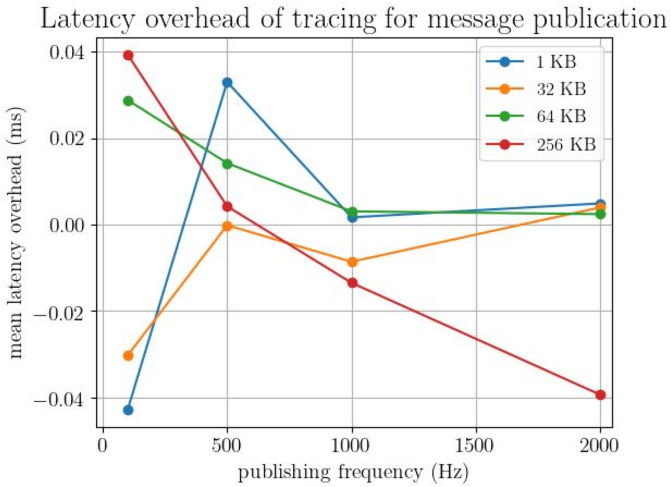


Figure 2. Individuals results: no tracing (left) vs. tracing (right).

Overhead benchmark - results (2)

- Still some variability: negative overhead?!
- But overall it does looks very good!



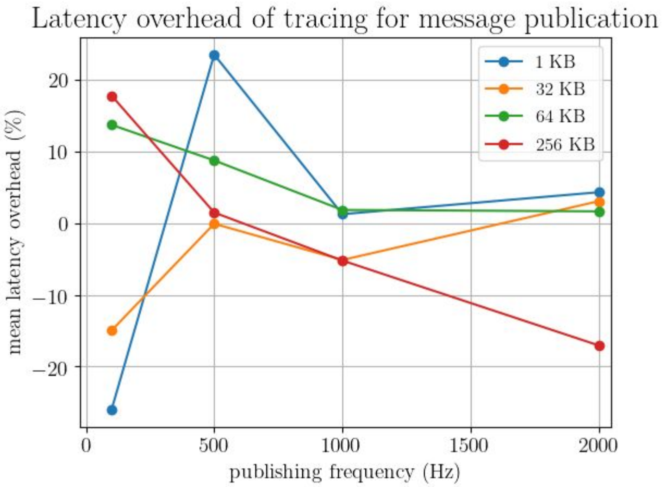


Figure 3. Overhead results: absolute (left) vs. relative (right).

Analysis

- Many tools to analyze traces generated by LTTng
  - babeltrace: babeltrace.org
  - Trace Compass: tracecompass.org
- `tracetools_analysis`
  - gitlab.com/ros-tracing/tracetools_analysis
  - Goal: quick trace analysis
  - Simple Python tool
  - Pre-processes raw trace data, provides multiple 2D tables as pandas DataFrames
  - Offers simple functions to analyze those DataFrames
  - Use inside a Jupyter Notebook, or in a simple Python file
- Advanced analyses
  - **Correlate** ROS 2 trace events with events from the Linux kernel or other applications
  - **Analyze the aggregation** of traces from multiple systems

Babeltrace

TRACE COMPASS

Analysis - example

- Plot callback durations

```
import tracetools_analysis; import bokeh
events = load_file('~/.ros/tracing/pingpong')                    # Read the trace
handler = Ros2Handler.process(events)                            # (Pre-)process the data
data_util = Ros2DataModelUtil(handler.data)
callback_symbols = data_util.get_callback_symbols()              # Extract callback functions
duration = bokeh.plotting.figure(...)
for callback, symbol in callback_symbols.items():                # For each callback...
   owner_info = data_util.get_callback_owner_info(callback)
   if not owner_info or '/parameter_events' in owner_info:       #   Filter out internal subscriptions
       continue
   duration_df = data_util.get_callback_durations(callback)      #   Get duration data for this callback
   duration.line(x='timestamp', y='duration', legend=str(symbol), #   Add to plot
                 source=bokeh.models.ColumnDataSource(duration_df))
bokeh.io.show(duration)                                          # Display final plot
```

Analysis - example (2)

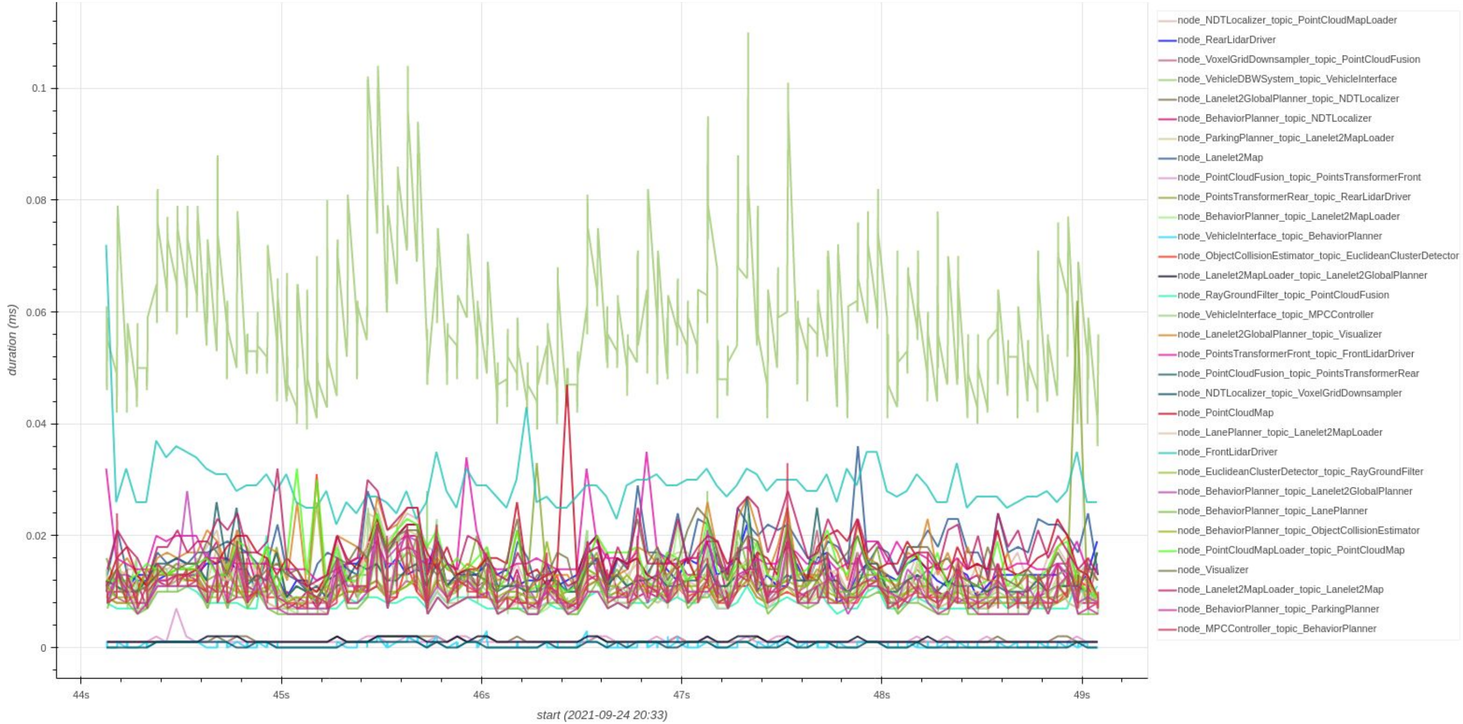


Figure 4. Callback durations plot.

Analysis - example (3)

- Using Trace Compass
- Critical path analysis of a wget request
- Computes dependencies between threads
- Only using data from the Linux kernel
  - Blocking system calls

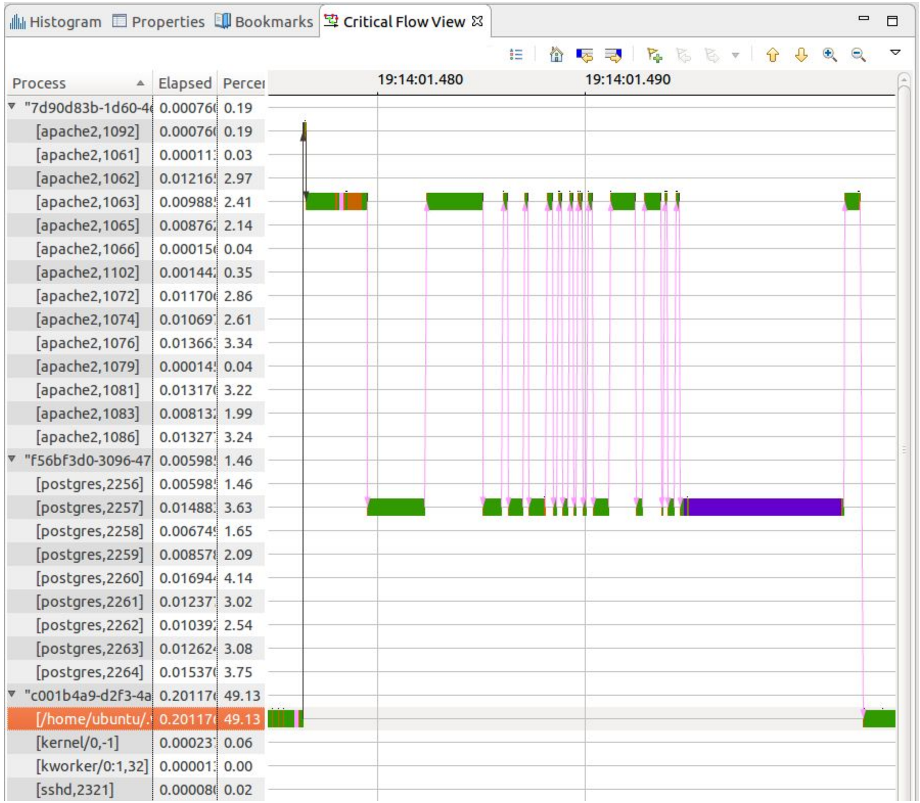
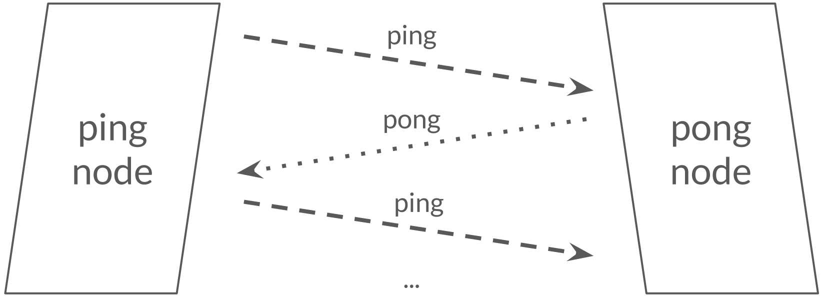


Figure 5. Critical path analysis using Trace Compass.

Demo

- Ping & pong nodes
  - Nodes exchange N messages every M milliseconds T times, then exit
- Link to instructions and Python code in a Jupyter Notebook
  - github.com/christophebedard/ros-world-2021-demo



Demo - results

- Simple demo, ROS 2-level information only

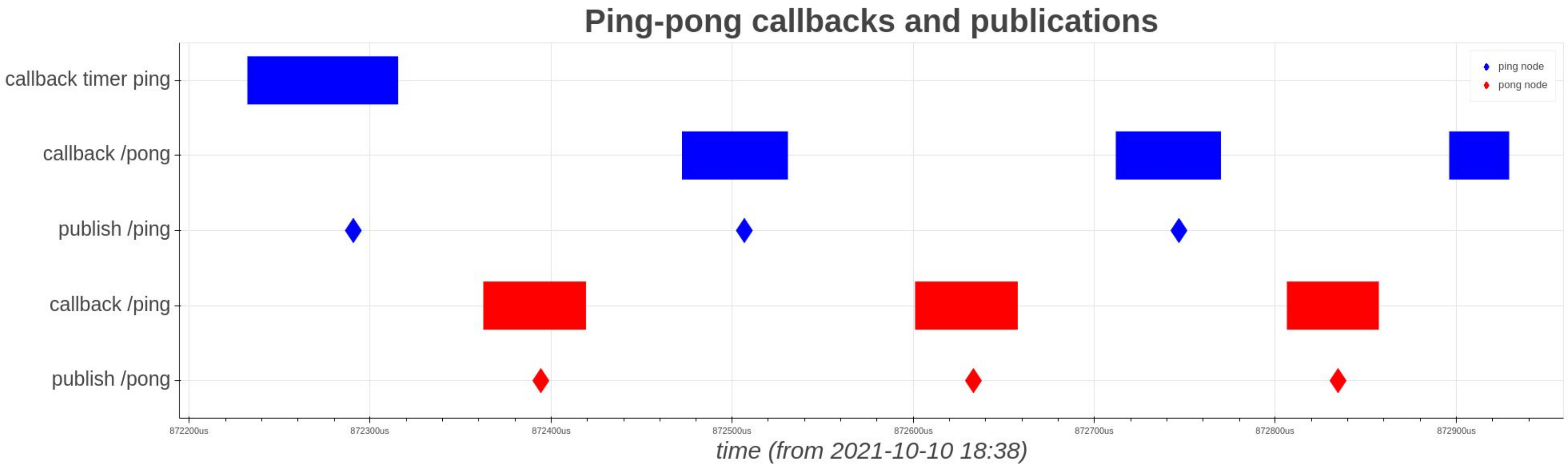


Figure 6. Results for 1 sequence.

Demo - results (2)

- Still a lot of information & many possibilities, especially if we add DDS/middleware instrumentation!

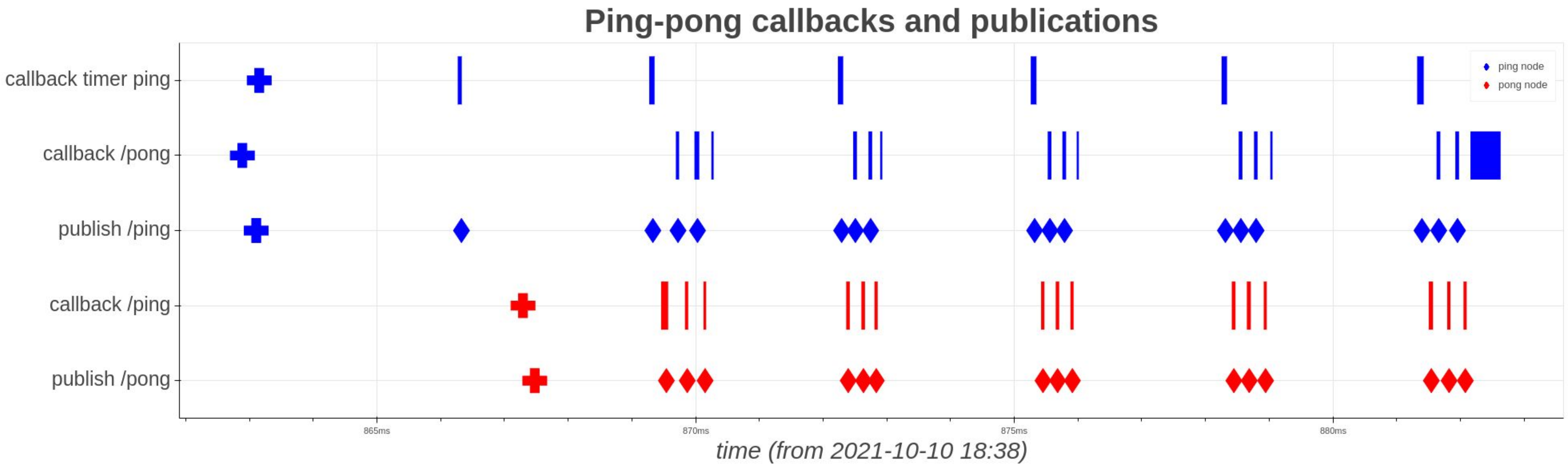


Figure 7. Overall demo results.

Conclusion

- Tracing
  - Gather low-level runtime execution information
  - Use a low overhead tracer
- `ros2_tracing`
  - Instrumentation for the core of ROS 2
  - Tools to trace with LTTng
- Analysis
  - Correlate OS events with ROS 2 events
  - Analyze the aggregation of traces from multiple systems
- Future
  - Including the LTTng tracepoints in the Linux binaries
  - Instrumentation
    - Internal handling of messages, tracking messages across nodes
    - DDS
  - What would you like to see?!

Questions?

- github.com/christophebedard

- Important links
  - gitlab.com/ros-tracing/ros2_tracing
  - gitlab.com/ros-tracing/tracetools_analysis
  - lttng.org
  - `ros2_tracing` tutorial in RTWG docs: bit.ly/RTWG_tracing_tutorial