Tracing ROS 2 with `ros2_tracing`

Christophe Bédard

ROSCon Fr 2021
June 22, 2021

DORSAL Laboratory
Polytechnique Montréal 🇨🇦

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

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
- Observability problems
  - Observer effect
  - Have to avoid influencing or affecting the application
- Distributed systems
  - How to analyze a distributed system?
- Real-time
- Observing an application's (lack of) determinism

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!
  - Observer effect
  - Use in production

LTTng

- lttng.org
- High-performance tracer
    - Low overhead
- Linux only
- Instrumentation
    - Built into the Linux kernel
    - Or added statically to your application
- Trace data processing
    - Online (live)
    - Offline (more common & simple)

LTTng - example

```
$ 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 pkg exe
$ lttng stop && lttng destroy
```

LTTng - example (2)

```
$ 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 }, { publisher_handle = 0x541A40, message = 0x5464F0 }
ros2:rcl_publish: { cpu_id = 1 }, { publisher_handle = 0x541A40, message = 0x5464F0 }
ros2:rmw_publish: { cpu_id = 1 }, { rmw_publisher_handle = 0x541AE0, 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 Dashing (2019)
- Tools to instrument the core of ROS 2 with LTTng
  - `rclcpp`, `rcl`, `rmw`
- Tools to configure tracing with LTTng
  - Command: `ros2 trace`
  - Action for ROS 2 launch: `Trace`

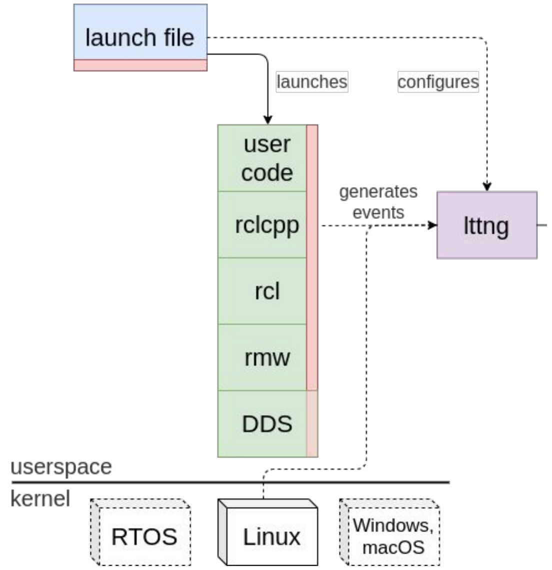


Figure 1. Instrumentation and general workflow.

Instrumentation

- Only on Linux, not included in the binaries
  - Install LTTng and (re)build ROS 2
- 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
- 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, pub, sub, timer
- Events
  - Callback execution (sub, timer)
  - Message publication
  - Lifecycle node state change
  - 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:callback_start: { callback =  0x🤖, is_intra_process = 0 }
      ros2:rclcpp_publish: { publisher_handle =  0x🇨🇦, message =  0x🇫🇷 }
      ros2:rcl_publish: { publisher_handle =  0x🇨🇦, message =  0x🇫🇷 }
      ros2:rmw_publish: { rmw_publisher_handle = 0x..., message =  0x🇫🇷 }
ros2:callback_end: { callback =  0x🤖 }
```

Tools - `ros2 trace` commande

```
$ 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
```

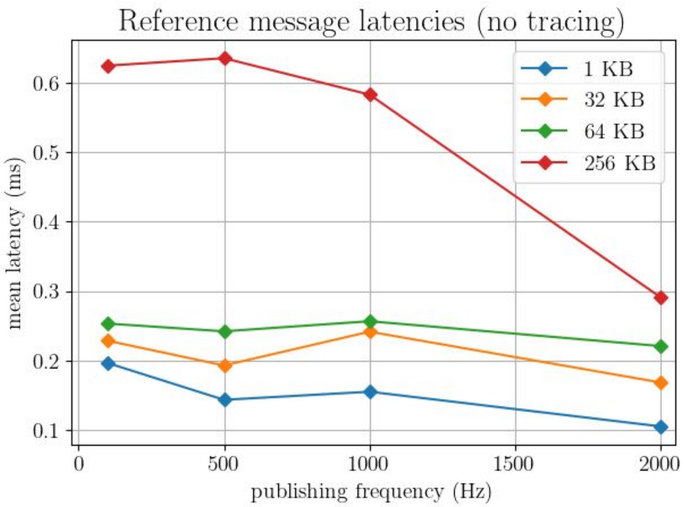
Tools - `Trace` action for ROS 2 launch

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

Overhead benchmark

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

Overhead benchmark - results



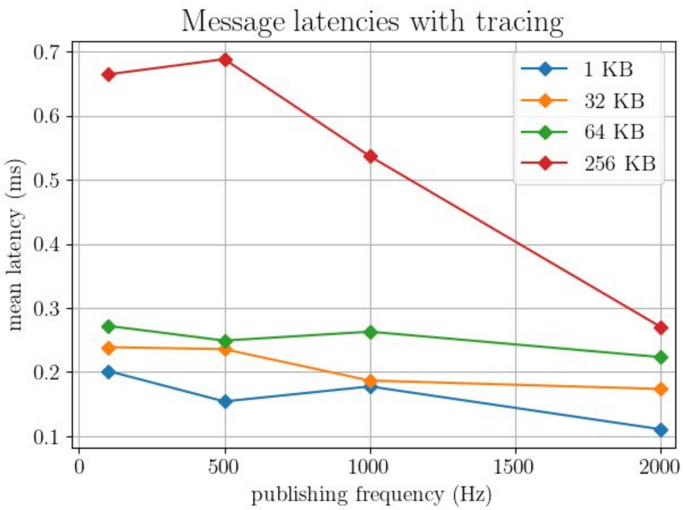


Figure 2. Individuals results.

Overhead benchmark - results (2)

- Hard to conclude, but encouraging
- There might be too much variability in the OS and networking layers

- Optimize real-time setup
- Other benchmarks
  - Distributed
  - Use median latency values

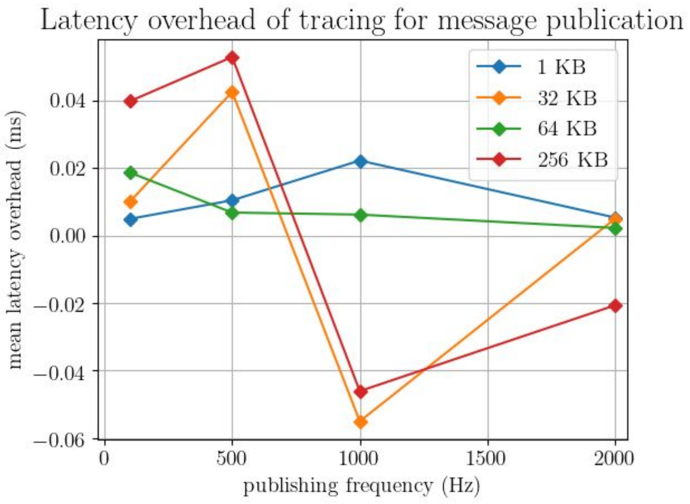


Figure 3. Overhead results.

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 data, provides pandas DataFrames
  - Offers simples functions to analyze those DataFrames
  - Use inside a Jupyter Notebook
- Advanced analyses
  - **Correlate** ROS 2 trace events with the Linux kernel trace events
  - **Analyze the aggregation** of traces from multiple systems

Analysis - example

```python
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 obj, symbol in callback_symbols.items():                         # For each callback...
   owner_info = data_util.get_callback_owner_info(obj)
   if not owner_info or '/parameter_events' in owner_info:           #   Filter out internal subscriptions
       continue
   duration_df = data_util.get_callback_durations(obj)               #   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)

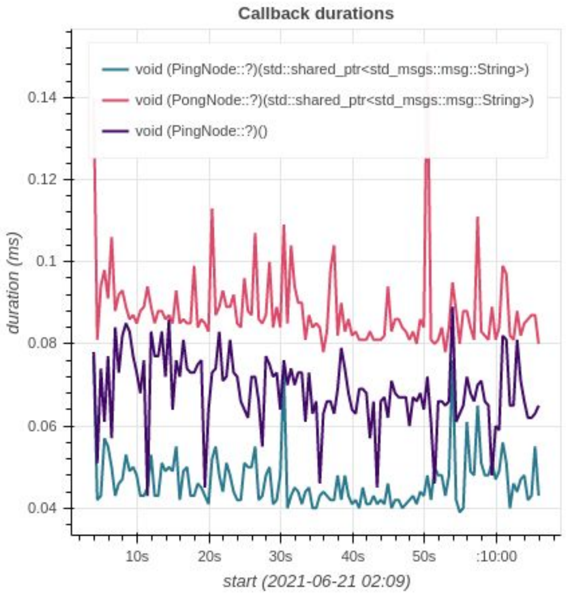


Figure 4. Callback duration plot.

Analysis - example (3)

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

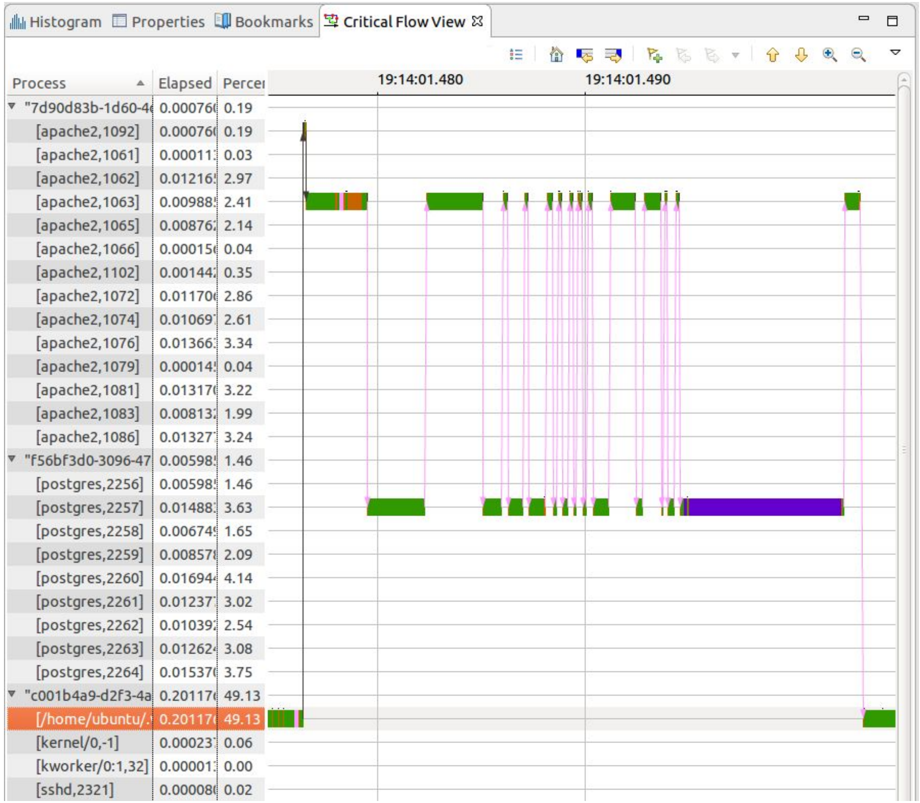


Figure 5. Critical path analysis using Trace Compass.

Demo

- ...with ros2_control!
  - Instrument the controller manager
  - Extract controller information: `init()` and `update()`
  - Compare with messages published on `/dynamic_joint_states`
- Instructions and Python code in a Jupyter Notebook
  - gitlab.com/ros-tracing/tracetools_analysis/-/blob/add-basic-ros2-control-demo/tracetools_analysis/analysis/ros2_control_demo.ipynb

Demo - results

- Simple demo
- A lot of information, many possibilities!

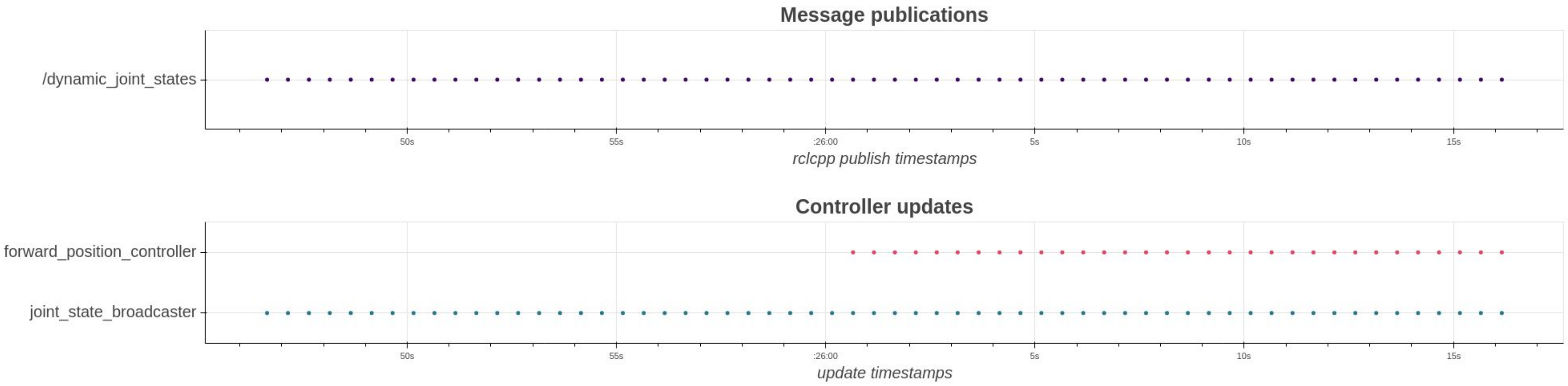


Figure 6. Demo results.

Conclusion

- Tracing
  - Gather low-level runtime execution information
  - Low overhead
- `ros2_tracing`
  - Tools to instrument the core of ROS 2
  - Tools to setup tracing with LTTng
- Analysis
  - Correlate OS events with ROS 2 events
  - Analyze the aggregation of traces from multiple systems
- Future
  - Include the instrumentation in the Linux binaries?
  - Instrumentation
    - Executor, internal handling of messages
    - DDS
  - What would you like to see?!

Questions?

- github.com/christophebedard

- Important links
  - lttng.org
  - gitlab.com/ros-tracing/ros2_tracing
  - gitlab.com/ros-tracing/tracetools_analysis

Demo (2)

- Instrumentation for ros2_control : `init()` and `update()`

```cpp
if (controller.c->init(controller.info.name) == controller_interface::return_type::ERROR) {
  to.clear();
  RCLCPP_ERROR(
    get_logger(),
    "Could not initialize the controller named '%s'",
    controller.info.name.c_str());
  return nullptr;
}
executor_->add_node(controller.c->get_node());
to.emplace_back(controller);
TRACEPOINT(
  control_controller_init,
  static_cast<const void *>(controller.c.get()),
  controller.info.name.c_str());
```

```cpp
controller_interface::return_type ControllerManager::update()
{
  std::vector<ControllerSpec> & rt_controller_list =
    rt_controllers_wrapper_.update_and_get_used_by_rt_list();

  auto ret = controller_interface::return_type::OK;
  for (auto loaded_controller : rt_controller_list) {
    // TODO(v-lopez) we could cache this information
    // https://github.com/ros-controls/ros2_control/issues/153
    if (is_controller_running(*loaded_controller.c)) {
      auto controller_ret = loaded_controller.c->update();
      TRACEPOINT(control_controller_update, static_cast<const void *>(loaded_controller.c.get()));
      if (controller_ret != controller_interface::return_type::OK) {
        ret = controller_ret;
      }
    }
  }

  // there are controllers to start/stop
  if (switch_params_.do_switch) {
    manage_switch();
  }

  return ret;
}
```

Demo (3)

- Launch file

```python
def generate_launch_description():
    launchfile_path = os.path.join(
        get_package_share_directory('ros2_control_demo_bringup'),
        'launch',
        'rrbot_system_position_only.launch.py',
    )
    base_ros2_control_launch = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([launchfile_path]),
        launch_arguments={ 'start_rviz': 'True' }.items(),
    )
    return LaunchDescription([
        Trace(
            session_name='ros2-control-demo',
            events_ust=[
                'ros2:*',
                'ros2:control_controller_init',
                'ros2:control_controller_update',
            ],
        ),
        base_ros2_control_launch,
    ])
```