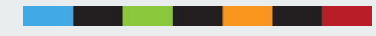


Tracing ROS 2

Christophe Bourque Bédard

Progress Report Meeting
June 4, 2021

Polytechnique Montréal
DORSAL Laboratory

POLYTECHNIQUE
MONTRÉAL

TECHNOLOGICAL
UNIVERSITY

Summary

1. Introduction
2. ROS 2
3. Tracing ROS 2
4. Instrumentation
5. Overhead benchmark
6. Upcoming work and conclusion
7. Questions

Introduction

- Robotics
  - Commercial or industrial applications
  - Safety-critical applications
  - Can be connected over a network (e.g. 5G)
- Key elements
  - Message passing and Remote Procedure Call (RPC)
  - Real-time constraints
- Robotics software development can greatly benefit from tracing

ROS 2

- Robot Operating System 2
    - docs.ros.org/en/galactic
- Open source framework and set of tools for robotics software development
    - Well-known in robotics
    - Used for NASA's 2023 Moon rover, VIPER!
- Message passing between "nodes"
    - Publish/subscribe
    - Service/action calls (~RPCs)
- Modular
    - Each node generally accomplishes a very specific task
    - Nodes are put together to perform complex tasks
- Uses Data Distribution Service (DDS) as the middleware
    - OMG standard
- Intra-process, inter-process, and distributed

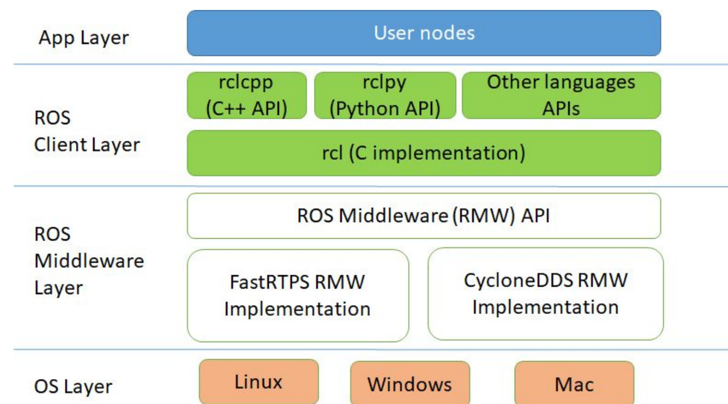


Figure 1. ROS 2 architecture and abstraction layers.

Tracing ROS 2

- LTTng instrumentation part of the ROS 2 core
  - gitlab.com/ros-tracing/ros2_tracing
- Instrumentation not part of the distributed binaries
  - Want to change that
  - Current work should help
- Closely integrated with ROS 2
  - To encourage use/adoption
  - ROS 2 CLI tools
  - ROS 2 launch/deployment system

Instrumentation

- Some design principles
  - Most likely similar to instrumentation for other applications
- Multiple layers of abstraction
  - Want information about each layer & the interaction between them
  - However, layers make it hard to get the full picture
- Real-time
  - Applications usually 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
- Publishing
  - Constant number of trace events, constant overhead (?)
- Not using DDS instrumentation here

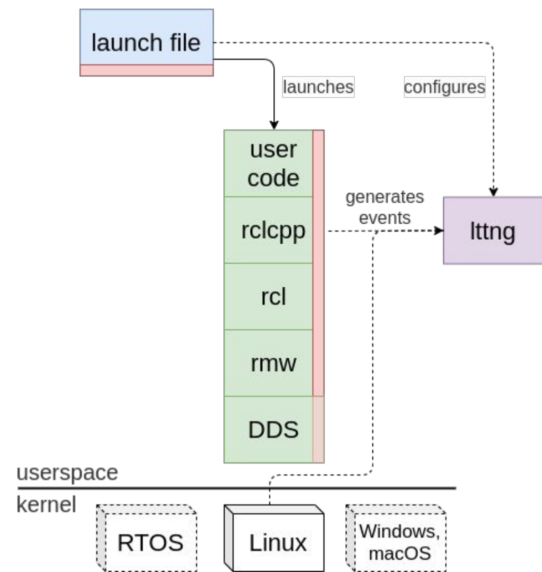
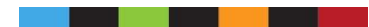


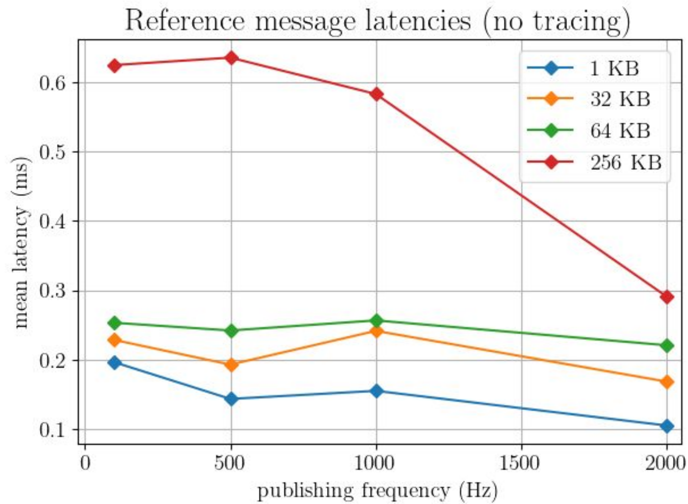
Figure 2. Workflow and instrumentation.



Overhead benchmark

- Goal: measure tracing overhead in a ROS 2 context
  - Mainly interested in latency overhead
  - Expecting it to be very small
  - Tool: gitlab.com/ApexAI/performance_test
- Parameters
  - Intra-process, inter-process, distributed
  - Publisher frequency
  - Message payload size
  - Number of nodes (publishers/subscribers) & graph setup
  - Quality of service settings
  - DDS implementation
- 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



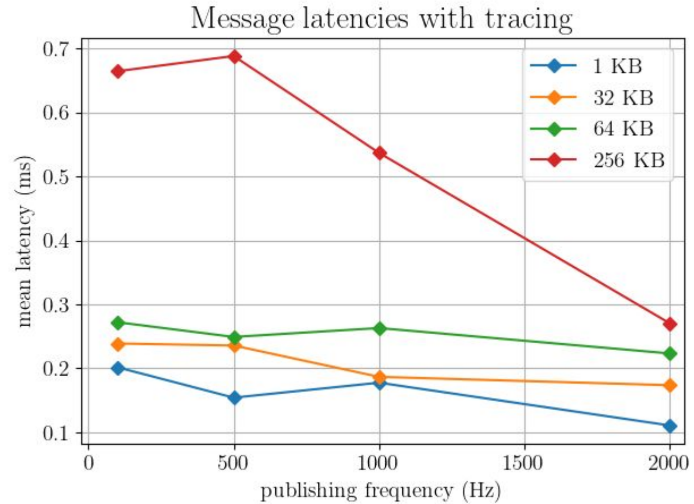


Figure 3. Individual results.

Overhead benchmark - results (2)

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

- Further benchmarks/experiments
  - Distributed
  - Use median latency values

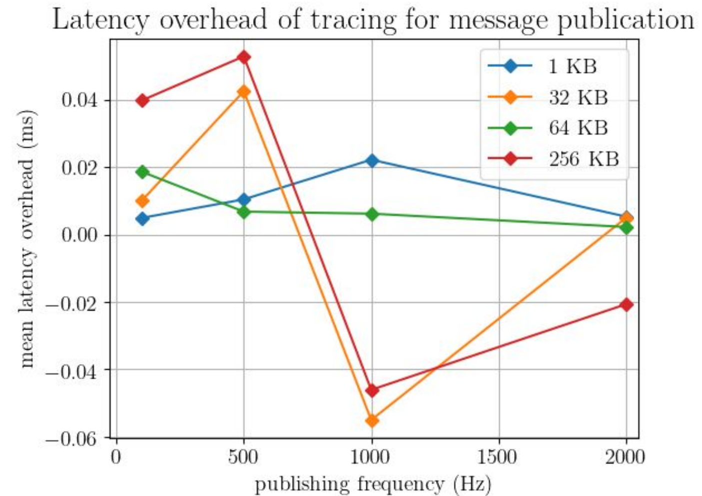


Figure 4. Overhead results.

Upcoming work and conclusion

- Instrumentation
  - Internal message handling
  - DDS level
- Critical path analysis for ROS 2 messages

Questions?

- christophe.bedard@polymtl.ca

- Links (bis)
  - docs.ros.org/en/galactic
  - nasa.gov/viper/lunar-operations#software
  - gitlab.com/ros-tracing/ros2_tracing
  - gitlab.com/ApexAI/performance_test