Concurrency in ROS 1 and ROS 2

Nicolo Valigi, SWE at Cruise nicolovaligi.com

ROSCon 2019

Testimonials 1/2

Latency on camera in ROS - ROS Answers: Open Source Q&A Forum

https://answers.ros.org › question › latency-on-camera-in-ros ▾

May 11, 2018 - I measure the **latency** by pointing the camera at a stopwatch on my screen ... Outside **ROS**, just by using a simple **C++** program and openCV I ...

Ros publish() taking more time than expected	5 answers	Apr 19, 2017
Image subscriber lag (despite queue = 1)	9 answers	Nov 5, 2015
Plotting lag, jitter, bandwidth and frequency of topics ...	9 answers	Mar 26, 2013
ros::time corresponding to message sent	1 answer	Jun 2, 2018

More results from answers.ros.org

Testimonials 2/2

Latency between nodelets

nodelet | latency. | node | ROS | stereo_image_proc

asked Apr 12 '17
mkreddy477
58 ●8 ●11 ●17

Dear all,

I have implemented stereo_image_proc with customized algorithms. Now I am trying to measure the latency in the pipeline.

I have observed that each nodelet has inconsistency latency ranging form 0 to 60ms. I have measured the latency as follows Ros_latency = (nodelet1_publishing_time - nodelet2_subscribing_time)

How can I minimize these delays so that making the whole system consistent.

Thanks in advance

This presentation covers

A journey from ROS down to the hardware.

1. From ROS abstractions to Linux threads (user-level)
2. From Linux threads to CPU hardware (kernel-level)
3. Profiling and analysis tools

If there's one thing you learn from this presentation

please disable the Nagle algorithm on your TCP sockets

```
ros::Subscriber sub = nh.subscribe(
    "my_topic",
    1,
    callback,
    ros::TransportHints().tcpNoDelay()
);
```

and [1]:

```
rosbag record --tcpnodelay
```

[1] https://github.com/ros/ros_comm/pull/1295/files

User-level concurrency

The basic ROS execution model

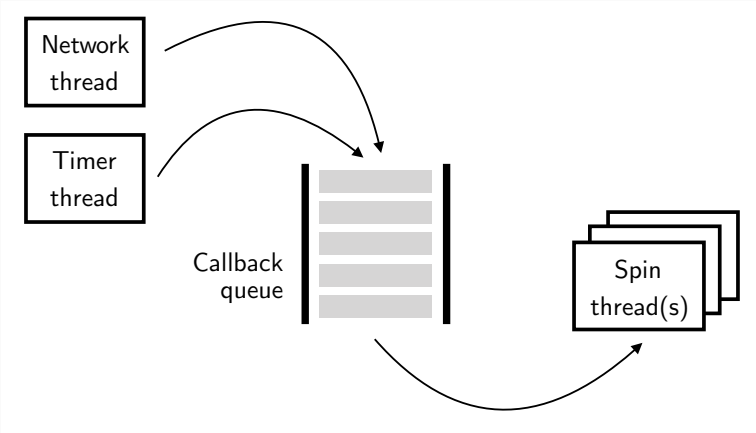
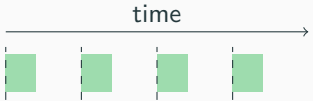


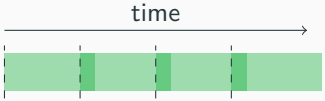
Figure 1: Network and spin threads interact through the callback queue

A single node

With a single node and a single CPU core, everything works fine:



But as the computation time increases, there's not enough time to keep up, and executions would start to overlap:



Enter the `AsyncSpinner`

If we don't want to drop messages, the hope is to throw more hardware at the problem.

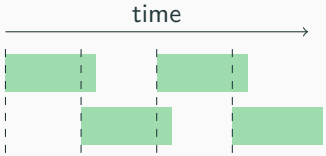
ROS 1 has `AsyncSpinner`, which spawns multiple threads that works on callbacks in parallel.

Except it doesn't work here, as each subscription is protected by a mutex. You can opt-in on a per-subscription basis by writing:

```
ros::SubscribeOptions ops;
ops.template init<std_msgs::String>(
    "chatter", 1000, chatterCallback);
ops.allow_concurrent_callbacks = true;
ros::Subscriber sub = nh.subscribe(ops);
```

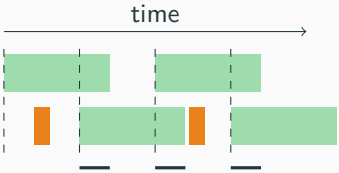
`AsyncSpinner` rocks

With this change, incoming messages are handed off to multiple CPU cores that can work on them in parallel:



Priority inversion

If the same node subscribes to a second (higher-priority) topic, then simply having multiple threads does not guarantee that the higher priority messages are processed first.



In this case, adding more worker threads is just a bandaid solution.

Multiple callback queues

Instead, we can assign callbacks by priority into different callback queues, create multiple spinners, and assign a callback queue to each spinner.

```
// Create a second NodeHandle
ros::NodeHandle secondNh;
ros::CallbackQueue secondQueue;
secondNh.setCallbackQueue(&secondQueue);
secondNh.subscribe("/high_priority_topic", 1,
                   highPriorityCallback);

// Spawn a new thread for high-priority callbacks.
std::thread prioritySpinThread([&secondQueue]() {
    ros::SingleThreadedSpinner spinner;
    spinner.spin(&secondQueue);
});
prioritySpinThread.join();
```

Multiple queues in ROS 2

The idea is the same in ROS 2:

```
// Create a Node and an Executor.
rclcpp::executors::SingleThreadedExecutor executor1;
auto node1 = rclcpp::Node::make_shared("node1");
executor1.add_node(node1);

// Create another.
rclcpp::executors::SingleThreadedExecutor executor2;
auto node2 = rclcpp::Node::make_shared("node2");
executor2.add_node(node2);

// Spin the Executor in a separate thread.
std::thread spinThread([&executor2]() {
    executor2.spin();
});
```

ROS 2 Executor algorithm

The Executor has a peculiar execution model (see [2]). Callbacks are scheduled starting from a "snapshot" of ready callbacks reported by the IPC layer. Timers always go first, then subscriptions, then services. Once all callbacks in the snapshot have run, the IPC layer is queried again.

This is unlike ROS 1, which was plain FIFO, and is not great for predictability and leads to *priority inversion*.

There's work underway to optimize the `Executor` API and make it more flexible.

[2] Daniel Casini and Tobias Blaß and Ingo Lütkebohle and Björn B. Brandenburg, *Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling*

Kernel-level concurrency

From threads to CPU cores

`CallbackQueues` and `Spinners` map topics onto threads. But how do we map threads to CPU cores? This is the job of the scheduler in the Linux kernel.

For CPU intensive tasks, the best option is to have one thread for each CPU core. I have never seen a ROS stack like that, which means that we *also* need to use OS-level tools.

There are 4 main tools that can be useful for Robotics systems:

- thread priorities
- the deadline scheduler
- real-time patched kernel
- CPU affinities and `cpusets`

1/4 Thread priority

Tools like `nice` (or the `pthread` API) can be used to set the priority of threads. The default scheduler in Linux (CFS) gives more CPU time to threads with higher priority.

But this is only guaranteed over a sufficiently large time slice. For shorter periods of time, low-priority tasks might steal the CPU away from higher-priority tasks.

2/4 Deadline scheduling

The deadline scheduler is an entirely different scheduling algorithm, part of a broader effort to make the Linux kernel more *real-time* capable.

Threads can opt-in to this scheduling class and get a slice of n microseconds CPU time every time period of m microseconds.

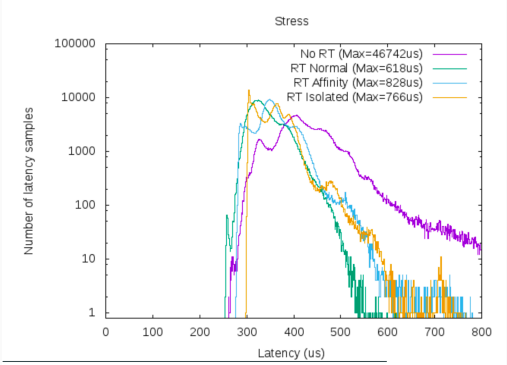
Deadline-scheduled tasks have priority over all other tasks in the system, and are guaranteed not to interfere with each other.

2/4 SCHED_DEADLINE code example

```
struct sched_attr attr;

attr.size = sizeof(attr);
attr.sched_flags = 0;
attr.sched_nice = 0;
attr.sched_priority = 0;

/* This creates a 10ms/30ms reservation */
attr.sched_policy = SCHED_DEADLINE;
attr.sched_runtime = 10 * 1000 * 1000;
attr.sched_period = attr.sched_deadline = 30 * 1000 * 1000;

int ret = sched_setattr(0, &attr, flags);
```

3/4 SCHED_DEADLINE and real-time kernels

Even with deadline scheduling, the kernel can still steal the CPU away from user code. The RT_PREEMPT patch to the kernel improves the situation quite a bit, and improves the latency distribution as reported in [3]:



[3] Carlos San Vicente Gutiérrez, Lander Usategui San Juan, Irati Zamalloa Ugarte, Víctor Mayoral Vilches, *Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications*

4/4 sched_getaffinity and cpuset

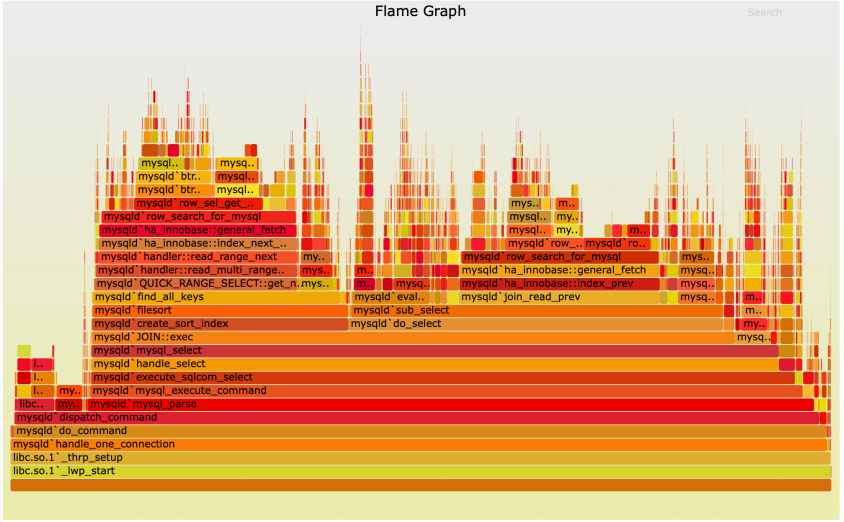
On multi-core machines, the Linux scheduler has heuristics to choose which CPU core a thread should run on. Sometimes, we want more control.

`sched_setaffinity` and `cpusets` can be used to limit which CPU cores a task can be scheduled on.

This improves cache performance and further limits interference between different components.

Profiling and analysis tools

perf is today's shiniest tool for Linux profiling, but is based on sampling.



Application-level tracing

There's a plethora of other tracing tools on the market:

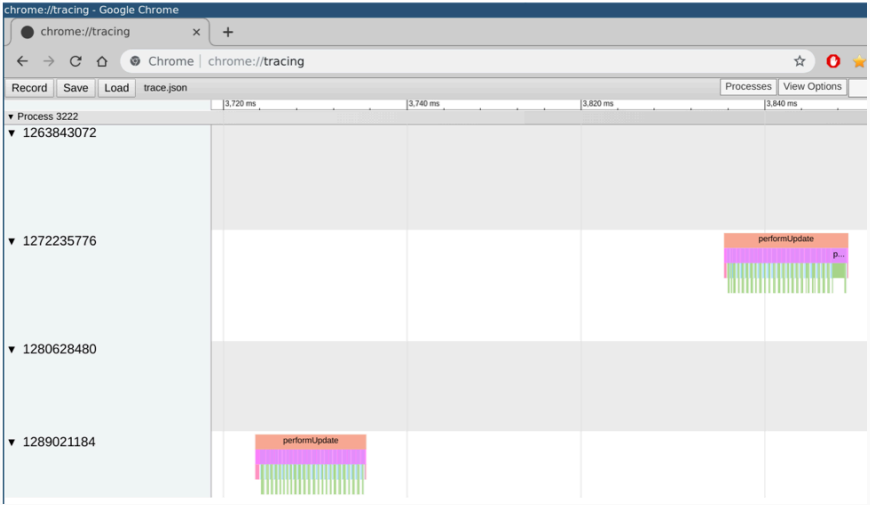
- github.com/bosch-robotics-cr/tracetools
- LTTng

To keep things simple, you can just add custom profiling code. Often implemented as RTTI objects in C++ to measure the runtime of (nested) scopes:

- github.com/swri-robotics/swri_profiler
- github.com/hrydgard/minitrace

Chrome tracing format

The Chromium project has a nice frontend to visualize tracing data.



Common issues with profiling ROS stacks

- Callback queues and worker thread pools remove all the context between related computations, thus making it harder to debug concurrency issues.
- Adding metadata to the profiler (timestamps, function arguments, etc..) helps trace the flow of execution during analysis.

10.000 feet view

10.000 feet view

- Concurrency is hard: the more threads you have on the system, the harder it is to ensure that the system is doing what you want.
- ROS encourages you to have lots of threads, and treats them like cattle. Please treat your threads like pets.
- Libraries should never call `std::thread` directly: nodes should have control over parallelism to avoid overloading or underloading the system.

Thank you / Questions