Agenda

1. Problems in performance analysis and execution monitoring
2. What is tracing?
3. ros2_tracing
   1. Installation
   2. Getting Started
   3. Built-in tracepoint recording and analysis
   4. Custom tracepoint recording
4. Outlook
ROS2 Tracing

Problems in performance analysis and execution monitoring

▲ Typical questions
  ▶ How long does my system take to react?
    – Corollary: Is my system always reacting fast enough?
  ▶ How much resources is my system consuming → sizing compute hardware
    – Corollary: Where does the resource use come from?
  ▶ Is my system still within its expected resource corridor?

▲ Complicating factors
  ▶ Distributed systems
    – Many nodes running concurrently
  ▶ Repetitive periodic processing
    – Not all of which are equal
  ▶ Performance Analysis: Low overhead important for correct data
  ▶ Execution Monitoring: Low overhead paramount
ROS2 Tracing
What is tracing?

- Tracing: „record information about system execution“
ROS2 Tracing

What kind of information can we record?

Linux Events & BPF Support

Dynamic Tracing

Tracepoints
- Linux 4.7
  - ext4:
    - Operating System
  - sched:
    - syscall:
    - task:
    - signal:
    - timer:
    - workqueue:
- sock:
- CPU Interconnect

uprobes
- Linux 4.3
  - VFS
  - Sockets
  - TCP/UDP
  - Volume Manager
  - IP
  - Block Device Interface
  - Ethernet
  - Device Drivers

kprobes
- Linux 4.1
  - jbd2:
  - block:
  - scsi:
  - net:
  - skb:

BPF output
- Linux 4.4

BPF stacks
- Linux 4.9

Software Events
- cpu-clock
- cs migrations
- page-faults
- minor-faults
- major-faults

PMC's
- Linux 4.9
  - cycles
  - instructions
  - branch-\*
  - L1-\*
  - LLC-\*

http://www.brendangregg.com/ebpf.html 2017
ROS2 Tracing
Example analyses: Brendan Gregg’s famous flame graphs

Source: http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
## ROS2 Tracing

### Tracepoints in ROS 2 (as of 12/2019)

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Invocation</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>User code</th>
<th>Executor</th>
</tr>
</thead>
<tbody>
<tr>
<td>rclcpp</td>
<td>Node</td>
</tr>
<tr>
<td></td>
<td>Subscription</td>
</tr>
<tr>
<td>rcl</td>
<td>rcl_node</td>
</tr>
<tr>
<td>rmw</td>
<td>rmw_node</td>
</tr>
<tr>
<td>DDS</td>
<td>Participant</td>
</tr>
<tr>
<td>Linux</td>
<td>OS Kernel</td>
</tr>
</tbody>
</table>

- RMW-Layer is not very interesting, but could be added easily.
- DDS has many implementations – not sure how to proceed here.
ROS2 Tracing

Static tracing

- Static tracing: *Compile-time defined tracepoints in the source code, inserted by developers*
  - Pro: Encodes developer knowledge about what is important
  - Pro: Has direct access to all the data
  - Con: Takes effort to add for each tracepoint
  - Con: Possibly dependent on specific tracing framework

```c
TRACEPOINT_EVENT(
    TRACEPOINT_PROVIDER,
    callback_start,
    TP_ARGS(
        const void *, callback_arg,
        int, is_intra_process_arg
    ),
    TP_FIELDS(
        ctf_integer_hex(const void *, callback, callback_arg)
        ctf_integer(int, is_intra_process, is_intra_process_arg)
    )
)
```

```c
void dispatch(
    std::shared_ptr<Message> message, const rmw_message_info_t & message_info)
{
    TRACEPOINT(callback_start, (const void *)this, false);
}
```
ROS2 Tracing
Aside: Dynamic tracing

- Dynamic tracing: *Run-time defined tracepoints, configured by analyst*
  - Pro: Can be attached to any of the event sources with relatively little effort
  - Con: For uprobes, need to know the symbol you’re attaching to → requires in-depth knowledge of code
  - Con: Currently only supported by kernel-based tracers → context-switching overhead

- Hint: You can often use dynamic tracing for your own code to add some extra info
ROS2 Tracing

Installation

- This is an excerpt from https://micro-ros.github.io/docs/tutorials/advanced/tracing/

- Pre-Requisites
  - Linux-Trace-Toolkit ng
  - Babeltrace

- We have a repo-list for use with vcs
- Main thing: Build with `-DWITH_LTTNG=ON`
- By the magic of dynamic linking, now you can trace every ROS 2 application ;-)
ROS2 Tracing
Tracing your system

- Option 1: „ros2 trace“
- Option 2: „Trace“ action in launch file
ROS2 Tracing

Analyzing the trace part 1: CLI

```
lui3s1@RNGX7819:/src/tracing_ws$ ros2 run tracetools_analysis cb_durations -./ros/tracing/my-tracing-session/converted

<table>
<thead>
<tr>
<th>Count</th>
<th>Sum</th>
<th>Mean</th>
<th>Std</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>17</td>
<td>0.101837</td>
<td>0.039629</td>
<td>PongNode::?(std_msgs::String)</td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>0.064122</td>
<td>0.033667</td>
<td>PingNode::?(std_msgs::String)</td>
</tr>
<tr>
<td>0</td>
<td>17</td>
<td>0.051924</td>
<td>0.007078</td>
<td>PingNode::?()</td>
</tr>
<tr>
<td>1</td>
<td>57</td>
<td>0.002521</td>
<td>0.004859</td>
<td>TimeSource::?(rcl_interfaces::ParameterEvent)</td>
</tr>
<tr>
<td>2</td>
<td>58</td>
<td>0.002322</td>
<td>0.002937</td>
<td>TimeSource::?(rcl_interfaces::ParameterEvent)</td>
</tr>
</tbody>
</table>
```
ROS2 Tracing
Analyzing the trace part 2: Jupyter Notebook

Callback durations

- PingNode::?()
- TimeSource::?(rcl_interfaces::ParameterEvent)
- TimeSource::?(rcl_interfaces::ParameterEvent)
- PongNode::?(std_msgs::String)
- PingNode::?(std_msgs::String)
ROS2 Tracing
Custom traces: Function instrumentation

▶ Sometimes, a more detailed view is needed → custom tracepoints
▶ A simple approach is function-instrumentation with –finstrument-functions
  ▶ By default, this has too much overhead
  ▶ The “instrument-attribute-gcc-plugin” by Christophe Bourque Bedard addresses this
▶ Usage:
  ▶ Add instrumentation attribute
  ▶ Compile with -fplugin=./instrument_attribute.so
▶ This is essentially a selective form of profiling
ROS2 Tracing
Custom Trace Example: Executor profiling

- This summer, several people noticed that the ROS 2 SingleThreadedExecutor can have significant CPU overhead. See https://discourse.ros.org/t/singlethreadedexecutor-creates-a-high-cpu-overhead-in-ros-2/10077/10
- We traced this using custom tracepoints in the executor
- Result:
ROS2 Tracing

Outlook

▶ Tracepoints for services etc.
▶ More analyses provided out-of-the-box
▶ Performance improvement for analysis
▶ Live tracing
▶ Capturing data from ebpf-tracing
ROS2 Tracing

Conclusion

- Tracing is an excellent infrastructure for system-level analysis
  - Both for kernel and user-space
- Bosch has contributed initial tracing support for ROS 2
  - Tracepoints in framework
  - Integration with tooling
  - Basic analysis scripts

- We have many interesting internship / thesis projects in this area