D4.12

Embedded Transform TF Library Software Release Y2

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.12
Deliverable name	Embedded Transform TF Library – Software Release Y2
Date	December 2019
Dissemination level	public
Workpackage and task	4.4
Author	Bosch
Contributors	eProsima
Keywords	micro-ROS, robotics, ROS, microcontrollers
Abstract	Task 4.4 has been stopped during the project review on 10 September 2019 in Luxembourg. Instead, a new Task 7.5 on community demos has been introduced. Since the corresponding amendment has not been implemented yet, the present deliverable contains the deliverable for this new Task 7.5 as well as a new deliverable on the software release of runtime tracing tools.

1 Notice

In the Task 4.4, an agent-side filter to minimize the data communicated from the TF subsystem to the microcontroller has been developed in 2018 and beginning of 2019. With this filter, two of the four subtasks have been completed successfully. Our first experiences in the demos and use-cases revealed that the remaining two subtasks (real-time capable TF and efficient transformation computations) are of very low priority for most applications.

In the project review on 10 September 2019, we therefore proposed to the project officer and the reviewers to stop this task in favor of a new task on two easily reproducible demos for the ROS community. As a consequence, the deliverables D4.12 (D48) and D 4.13 (D49) shall be replaced by two new deliverables on open-source software releases of community demos. The project officer and reviewers agreed to this proposal. As the corresponding amendment has not been implemented yet as of December 2019, the documentation on the open-source releases of the community demos (preliminary deliverable number D7.16) will be published as the present deliverable.

During the project review, we furthermore proposed to introduce a new deliverable on the software release of runtime tracing tools (preliminary number D5.7), which have been implemented by Bosch in Task 5.3. Again, since the amendment has not been implemented yet, this deliverable in included here.

D7.16 (preliminary number)

Community Demos
Software Release Y2

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D7.16 (preliminary number)
Deliverable name	Community Demos – Software Release Y2
Date	December 2019
Dissemination level	public
Workpackage and task	7
Author	Ralph Lange (Bosch)
Contributors	Ingo Lütkebohle (Bosch), Borja Outerelo (eProsima)
Keywords	micro-ROS, robotics, ROS, microcontrollers, community, demo, Kobuki, Crazyflie
Abstract	This document provides links to the released software and documentation on the community demos based on a Kobuki (Turtlebot 2) robot and a Crazyflie quadcopter as part of the dissemination and communication activities in Work Package 7.

Contents

1 Overview to Results

This document provides links to the released software and documentation for the new deliverable *Community Demos - Software Release Y2* as part of the dissemination and communication activities in Work Package 7. The decision to introduce this deliverable for the second year (and an updated version for third year) of OFERA was made at the project review on 10 September 2019 in Luxembourg.

As an entry-point to all software and documentation, we created dedicated webpages on the micro-ROS website: micro-ros.github.io/kobuki_demo/ and micro-ros.github.io/crazyflie_demo/

The annex includes a copy of these webpages and a copy of the major documentation file of this software release.

2 Links to Software Repositories

2.1 Software for Kobuki Community Demo

The description of the demo setup and the ROS 2 code (launch files, URDF, etc.) for the laptop for controlling and visualizing the robot:

- Git repository: https://github.com/micro-ROS/micro-ROS_kobuki_demo
  Branch: master
  Main package: micro-ros_kobuki_demo_remote
  Latest commit as of December 2019: Commit d193989

The micro-ROS-based application code for the Kobuki robot, i.e. for the attached Olimex STM32-E407 board:

- Git repository: https://github.com/micro-ROS/apps
  Branch: demo/kobuki
  Subfolder of executable: apps/examples/kobuki/
  Latest commit as of December 2019: Commit ab1e5a5

2.2 Software for Crazyflie Demo and the Combined Demo

The micro-ROS-based application code (i.e. firmware based on FreeRTOS) for the Crazyflie quadcopter:

- Git repository: https://github.com/eProsima/crazyflie-firmware/
  Branch: crazyflie_microros
  Latest commit as of December 2019: Commit 54a5b8f

Extensions to micro-ROS for real-time operating system FreeRTOS:

- Git repository: https://github.com/micro-ROS/crazyflie_extensions/
  Branch: master
  Latest commit as of December 2019: Commit de1ce3a

Visualization of crazyflie quadcopter with rviz and ROS nodes for controlling the Kobuki robot by Crazyflie in combined demo:

- Git repository: https://github.com/micro-ROS/micro-ROS_kobuki_demo/
  Branch: crazyflie_demo
  Main package: micro-ros_crazyflie_demo_remote
  Latest commit as of December 2019: Commit 4640190

3 Annex 1: Webpage on Kobuki Demo

Content of micro-ros.github.io/kobuki_demo/ from 18th December 2019.

The micro-ROS Kobuki Demo illustrates the use of micro-ROS on the Kobuki platform, which is the mobile base of the well-known Turtlebot 2 research robot.

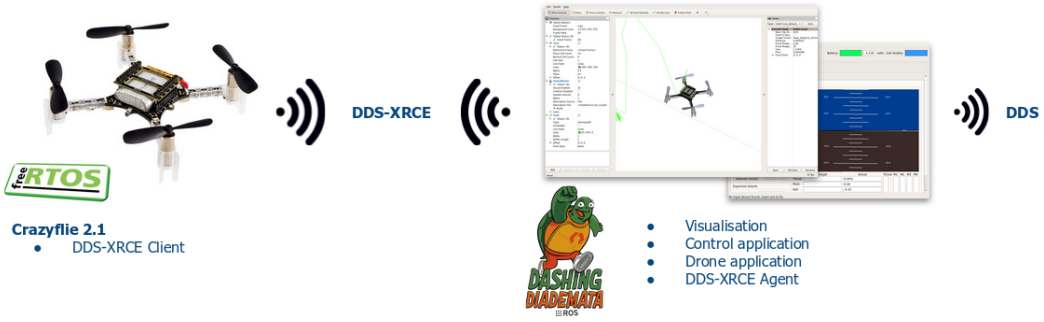
The basic idea and working principle of this demo is as follows: Instead of a laptop running ROS, the Kobuki is equipped with a STM32F4 microcontroller only. This STM32F4 runs the micro-ROS stack and a port of the thin_kobuki driver, which interacts with the robot's firmware (which runs on a built-in microcontroller). The STM32F4 communicates the sensor data via DDS-XRCE to a remote laptop running a standard ROS 2 stack, the micro-ROS agent and rviz. At the same time, using the other direction of communication, the Kobuki can be remote-controlled.

To run this demo yourself, follow the instructions given in https://github.com/micro-ROS/micro-ROS_kobuki_demo

4 Annex 2: Webpage on Crazyflie Demo

Content of micro-ros.github.io/crazyflie_demo/ from 18th December 2019.

This demo aims to expose a **micro-ROS** use case. It runs on a pair of embedded devices: a **Crazyflie 2.1** drone, used as a user controller, and a **Kobuki Turtlebot 2** as a mobile and controlled device.



Both of them rely on **micro-ROS** publication and subscription mechanisms and use an underlying **Micro XRCE-DSS client**.

This demo also includes conventional ROS 2 tooling as a demonstration of integration with **ROS 2**. We use Gazebo, RVIZ and simple ROS 2 nodes (aka **external nodes**) acting as data converters.

This demo was developed taking as base the Kobuki demo.

4.1 Index

4.2 Setup

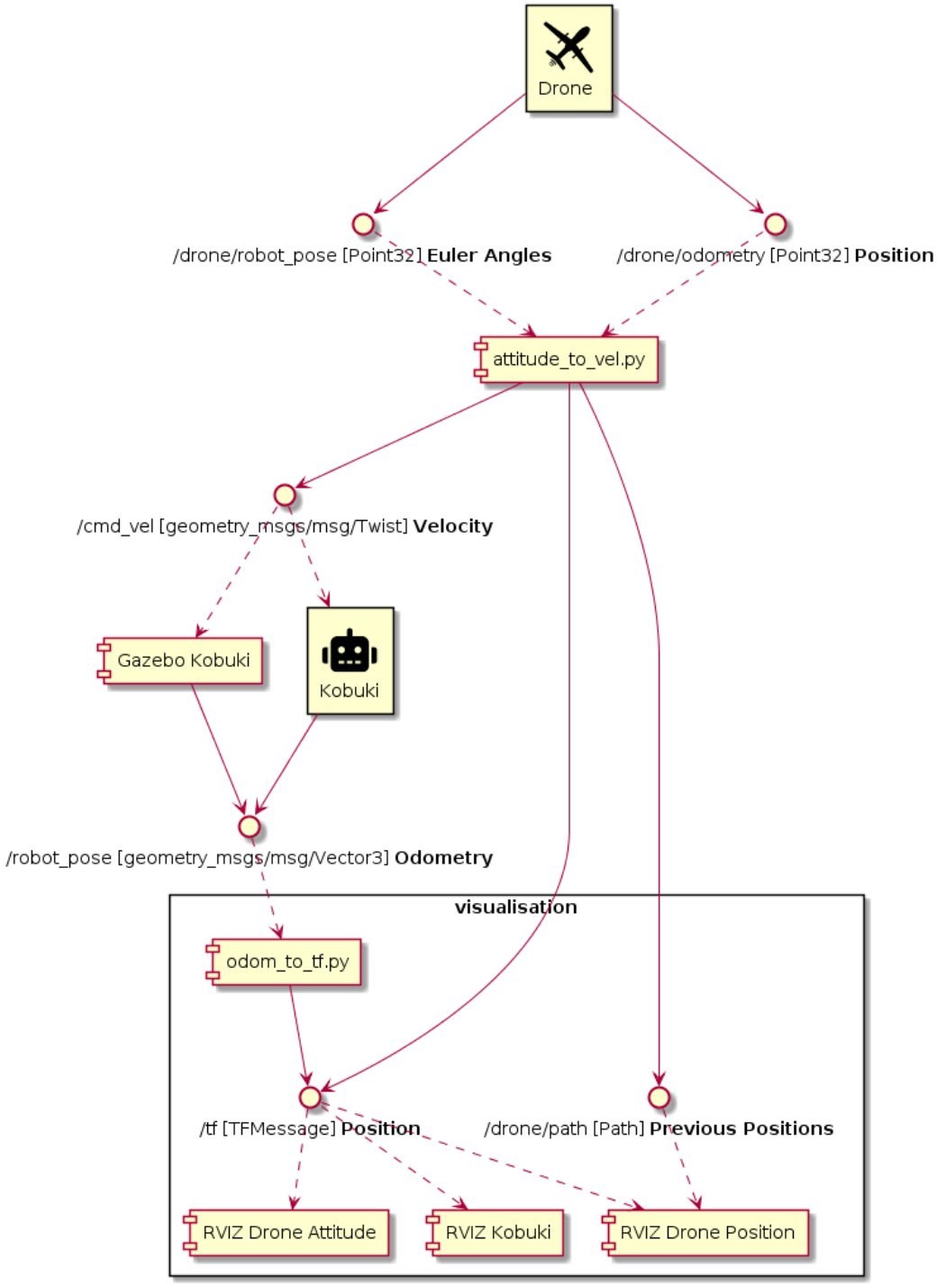
The proposed demo is composed of different kind of messages and topics.

The **Crazyflie 2.1** drone relies on ST STM32F405 MCU running **FreeRTOS**. Using the RTOS capabilities and the integrated radio communication device, the drone can run a node that publishes: - its own relative position as a 3D vector (X, Y and Z) using a *geometry_msg/Point32* message type on */drone/odometry* topic. - its own attitude as a 3D vector (pitch, roll and yaw) using a *geometry_msg/Point32* message type on */drone/attitude* topic.

The **Kobuki Turtlebot 2** robot is controlled using a UART protocol through a custom DB25 connector. The micro-ROS node runs on an Olimex STM32-E407 board attached to that UART port. This hardware features a ST STM32F407 MCU running **Nuttx** RTOS. In the same way, this node can communicate with the robot (UART) and with the ROS2 world (integrated Ethernet). Its used topics are: - a subscription on */cmd_vel* topic (*geometry_msg/Twist* message type) to receive the controlling angular and linear velocity. - a publication on */robot_pose* topic (*geometry_msg/Vector3* message type) which includes X position, Y position and robot yaw.

The **external ROS 2 nodes** are rclpy tools with some different functionalities: - *attitude_to_vel.py* - Converts Crazyflie */drone/attitude* to Kobuki Turtlebot 2 */cmd_vel* so that drone pitch is mapped to robot linear velocity and drone roll to angular valocity. - Converts Crazyflie publications on */drone/attitude* and */drone/attitude* topics to *tf2_msgs/TFMessage* messages (required by RVIZ visualizer) - *odom_to_tf.py* - Converts Kobuki Turtlebot 2 publications on */robot_pose* topic to *tf2_msgs/TFMessage* messages (required by RVIZ visualizer).

The following image shows the described setup.



4.3 Required Hardware

This setup uses the following hardware:

Item	
Kobuki Turtlebot 2	Link
Olimex STM32-E407	Link

Item	
Olimex ARM-USB-TINY-H	Link
Crazyflie 2.1	Link
Flow Desk v2	Link
Debug adapter	Link
Crazyradio PA 2.4 GHz USB dongle	Link
Additional battery + charger (optional)	Link

4.4 Installation

4.4.1 Install external ROS 2 nodes

Install Micro XCRE-DDS. Recommended procedure:

```
git clone https://github.com/eProsima/Micro-XRCE-DDS.git -b v1.1.0
cd Micro-XRCE-DDS
mkdir build && cd build
cmake ..
make
sudo make install
```

Create a workspace folder for this demo:

```
mkdir -p crazyflie_demo/src
cd crazyflie_demo
```

Clone this repo:

```
git clone --single-branch --branch crazyflie_demo https://github.com/micro-ROS/micro-ROS_kobuki
```

Install Gazebo. Recommended procedure:

```
curl -sSL http://get.gazebosim.org | sh
```

Install gazebo_ros_pkgs (ROS 2). Recommended procedure:

```
source /opt/ros/dashing/setup.bash
wget https://bitbucket.org/api/2.0/snippets/chapulina/geRKyA/f02dcd15c2c3b83b2d6aac00afe2811628
vcs import src < ros2.yaml
rosdep update && rosdep install --from-paths src --ignore-src -r -y
rm ros2.yaml
```

Build the project:

```
source /opt/ros/dashing/setup.bash
rosdep update && rosdep install --from-paths src --ignore-src -r -y
colcon build --symlink-install
```

4.4.2 Build and flash Crazyflie 2.1 firmware

Install the toolchain:

```
sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
sudo apt-get update
sudo apt install gcc-arm-embedded dfu-util
```

Download and build the **Crazyflie 2.1** firmware repository:

```
mkdir crazyflie_firmware
git clone https://github.com/eProsima/crazyflie-firmware -b crazyflie_demo
cd crazyflie_firmware
git submodule init
git submodule update
make PLATFORM=cf2
```

Unplug the **Crazyflie 2.1** battery

Push the reset button while connecting the USB power supply.

The top-left blue LED blinks, first slowly and after 4 seconds sightly faster, now it is in DFU programming mode. Check it with `lsusb`:

```
Bus 001 Device 051: ID 0483:df11 STMicroelectronics STM Device in DFU Mode
```

Flash the device:

```
sudo dfu-util -d 0483:df11 -a 0 -s 0x08000000 -D cf2.bin
```

Unplug and plug the **Crazyflie 2.1** power to exit DFU mode.

4.4.3 Install Crazyflie Client + Bridge

Install dependencies:

```
sudo apt-get install libusb-1.0-0-dev
sudo apt-get install python3 python3-pip python3-pyqt5 python3-pyqt5.qtsvg
```

Fix permissions for the Crazyradio PA 2.4 GHz USB dongle (restart required for apply changes):

```
sudo groupadd plugdev
sudo usermod -a -G plugdev $USER
sudo echo SUBSYSTEM==\"usb\", ATTRS{idVendor}==\"1915\", ATTRS{idProduct}==\"7777\", \
MODE=\"0664\", GROUP=\"plugdev\" > /etc/udev/rules.d/99-crazyradio.rules
sudo echo SUBSYSTEM==\"usb\", ATTRS{idVendor}==\"0483\", ATTRS{idProduct}==\"5740\", \
MODE=\"0664\", GROUP=\"plugdev\" > /etc/udev/rules.d/99-crazyflie.rules
```

Clone the repo dependencies:

```
git clone -b Micro-XRCE-DDS_Bridge https://github.com/eProsima/crazyflie-clients-python
```

4.4.4 Build and flash Kobuki Turtlebot 2 firmware

Create a workspace for building **micro-ROS**:

```
source /opt/ros/crystal/setup.bash
sudo apt install python-rosdep curl flex ed gperf openocd automake ed bison libncurses5-dev gcc
mkdir -p kobuki-firmware/src
cd kobuki-firmware
git clone --recursive -b crazyflie_demo https://github.com/micro-ROS/micro-ros-build.git src/mi
colcon build --packages-select micro_ros_setup
source install/local_setup.bash
```

Build **micro-ROS Agent**:

```
ros2 run micro_ros_setup create_agent_ws.sh
colcon build
source install/local_setup.sh
```

Install tools:

```
git clone https://bitbucket.org/nuttx/tools.git ~/tools
pushd ~/tools/kconfig-frontends >/dev/null
./configure --enable-mconf --disable-nconf --disable-gconf --disable-qconf
LD_RUN_PATH=/usr/local/lib && make && make install && ldconfig
popd >/dev/null
```

Build Olimex STM32-E407 firmware:

```
ros2 run micro_ros_setup create_firmware_ws.sh
cd firmware/NuttX
tools/configure.sh configs/olimex-stm32-e407/uros
cd ../..

#Put here your agent IP and port
find ./firmware/mcu_ws/ -name rmw_microxrcedds.config -exec sed -i "s/CONFIG_IP=127.0.0.1/CONFI
find ./firmware/mcu_ws/ -name rmw_microxrcedds.config -exec sed -i "s/CONFIG_PORT=8888/CONFIG_P

ros2 run micro_ros_setup build_firmware.sh
```

Connect Olimex ARM-USB-TINY-H JTAG debugger to Olimex STM32-E407 and flash the board:

```
cd firmware/NuttX
scripts/flash.sh olimex-stm32-e407
```

4.5 Usage

After installation, the following packages should be present in your system:

```
.
+-- Micro-XRCE-DDS # used for installing Micro-XRCE-DDS
+-- crazyflie_demo
+-- crazyflie-firmware # used for building and flashing Crazyflie 2.1 firmware
+-- kobuki-firmware # used for building and flashing Kobuki Turtlebot 2 firmware
+-- crazyflie-clients-python
```

Make sure that all ROS 2 or micro-ROS nodes created along with the following steps **can reach each other using its network interfaces**.

4.5.1 Run Kobuki Turtlebot 2 Node

TODO: Explain data and power connections between Kobuki Turtlebot 2, Olimex STM32-E407 and MiniRouter.

Run the **micro-ROS Agent**:

```
cd kobuki-firmware
source /opt/ros/crystal/setup.bash && source install/local_setup.bash
ros2 run micro_ros_agent micro_ros_agent udp 9999
```

micro-ROS Agent should receive an incoming client connection and */robot_pose* topic should be published. Check it with `ros2 topic echo /robot_pose`

4.5.2 Run Crazyflie 2.1 Node

Connect Crazyradio PA 2.4 GHz USB dongle and turn on Crazyflie 2.1 drone.

Run the Crazyflie Client + Bridge:

```
cd crazyflie-clients-python
python3 bin/cfclient
```

This command should open the Crazyflie Client and print a serial device path in the terminal (something like /dev/pts/0).

Run (in another prompt) a **Micro XRCE-DDS Agent**:

```
MicroXRCEAgent serial --dev [serial device]
```

Micro XRCE-DDS Agent should receive an incoming client connection and */drone/attitude* and */drone/position* topics should be published. Check it with `ros2 topic echo /drone/attitude` and `ros2 topic echo /drone/position`

4.5.3 Run external ROS 2 nodes

Run commands:

```
cd crazyflie_demo
source /opt/ros/crystal/setup.bash && source install/local_setup.bash
ros2 run micro-ros_crazyflie_demo_remote attitude_to_vel
```

Topic */cmd_vel* should be published, and the **Kobuki Turtlebot 2** should start moving. Check it with `ros2 topic echo /cmd_vel`

4.5.4 Run RVIZ visualizers

Run complete visualizer:

```
cd crazyflie_demo
source /opt/ros/crystal/setup.bash && source install/local_setup.bash
ros2 launch micro-ros_crazyflie_demo_remote launch_drone_position.launch.py
```

RVIZ windows should be open, and a Crazyflie 2.1 drone model should represent the drone attitude and position along with a historic path.

Run attitude visualizer:

```
cd crazyflie_demo
source /opt/ros/crystal/setup.bash && source install/local_setup.bash
ros2 launch micro-ros_crazyflie_demo_remote launch_drone_attitude.launch.py
```

RVIZ windows should be open and a Crazyflie 2.1 drone model should represent **only** the drone attitude.

D5.7 (preliminary number)

Tracing Tools
Software Release

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D5.7 (preliminary number)
Deliverable name	Tracing Tools – Software Release
Date	December 2019
Dissemination level	public
Workpackage and task	5.3
Author	Ingo Lütkebohle (Bosch)
Contributors	
Keywords	micro-ROS, robotics, ROS, microcontrollers, tracing
Abstract	This document provides links to the released software and documentation for deliverable D5.7 *Tracing Tools - Software Release* of the works on tracing for micro-ROS in WP 5, as discussed in the project review 10 September 2019 and defined in the amendment to the Grant Agreement from December 2019.

Contents

1 Overview to Results

This document provides links to the released software and documentation for new deliverable *Tracing Tools - Software Release* of the works on tracing for micro-ROS (and ROS 2 in general) in WP 5, as discussed in the project review 10 September 2019.

As an entry-point to all software and documentation, we created a dedicated webpage on the micro-ROS website: https://micro-ros.github.io/tracing/

2 Links to Software Repositories

The repositories for tracing are found at https://gitlab.com/micro-ROS/ros_tracing. These include a number of release-management related repositories as well. The main technical repositories are:

- https://gitlab.com/micro-ROS/ros_tracing/ros2_tracing: The tracing abstraction API and related tools for testing and launching.
- https://gitlab.com/micro-ROS/ros_tracing/tracetools_analysis: The data-reading and aggregation modules for trace data analysis, a "verb" for the ros2 CLI tool to invoke them, and example Jupyter notebooks for visualization.

The ROS 2 framework instrumentation has been merged into ROS 2 mainline with the following two pull requests:

- https://github.com/ros2/rclcpp/pull/789: PR for rclcpp
- https://github.com/ros2/rcl/pull/473: PR for rcl

3 Annex 1: Webpage on Tracing

Content of https://micro-ros.github.io/tracing/ from 19th December 2019.

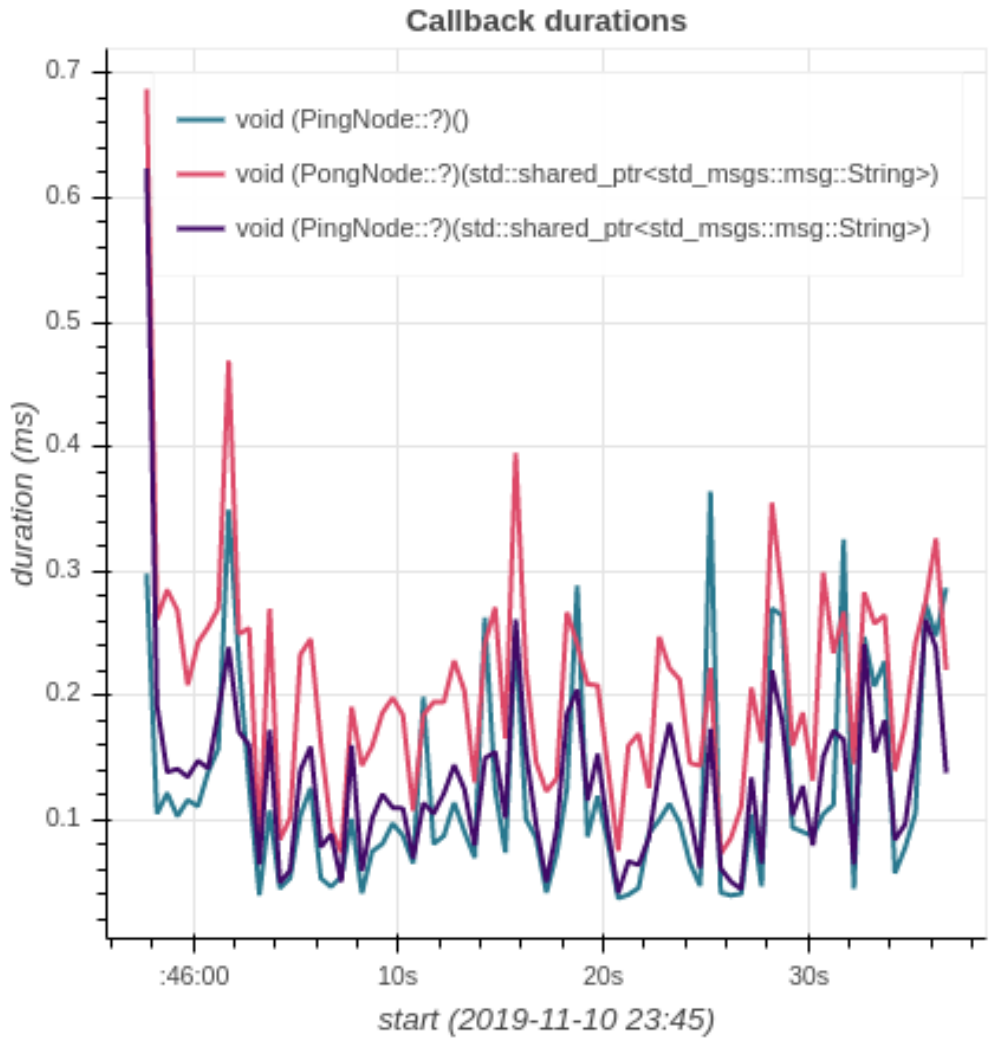
3.1 Introduction

Robotic systems can be hard to analyze and debug, and one big reason is that internal processing is always changing in response to sensory input. Therefore, the ability to continuously monitor and record data about the robotic software is important, to make sure it behaves deterministically, stays within resource limits, and also for later analysis.

On modern systems, the operating system and other running software has a big influence on the exact execution of the software. Therefore, we also need information about these aspects.

Tracing is a well-established method that allows to record run-time data, which is already well integrated with operating systems. For example, we can trace when a process is being scheduled, or when I/O occurs. Current tracing systems have minimal overhead and are very configurable to reduce overhead (and data size) even further.

This post aims to introduce our ongoing effort to instrument ROS 2 and provide trace analysis tools. I'll show how we can use the instrumentation and the current analysis tools to plot callback durations, like the plot shown below.



3.2 Setup

We'll assume you're using Ubuntu 18.04 bionic.

First, let's install LTTng.

```
$ sudo apt-add-repository ppa:lttng/stable-2.10
```

```
$ sudo apt-get update
$ sudo apt-get install lttng-tools lttng-modules-dkms liblttng-ust-dev
```

We'll also need these Python packages to read traces and setup a tracing session through ROS.

```
$ sudo apt-get install python3-babeltrace python3-lttng
```

If the ROS 2 development tools and dependencies are not installed on your machine, install them by following the *System setup* section here.

Now we'll download all the necessary packages. First, create your workspace.

```
$ mkdir -p ~/ros2_ws/src
$ cd ros2_ws/
```

The `rcl` and `rclcpp` instrumentation has been integrated into Eloquent, so we simply need to recompile `ros2_tracing` & compile `tracetools_analysis`.

```
$ wget https://gitlab.com/micro-ROS/ros_tracing/ros2_tracing/raw/master/tracing.repos
$ vcs import src < tracing.repos
```

Now let's build and source.

```
$ colcon build --symlink-install
$ source install/local_setup.bash
```

3.3 Simple tracing example

Let's try tracing with a simple ping-pong example.

The `tracetools_test` package contains two nodes we can use. The first node, `test_ping`, publishes messages on the `ping` topic and waits for a message on the `pong` topic before shutting down. The second node, `test_pong`, waits for a message on the `ping` topic, then sends a message on the `pong` topic and shuts down.

To trace these nodes, we can use the `example.launch.py` launch file in the `tracetools_launch` package.

```
$ ros2 launch tracetools_launch example.launch.py
```

```
~/ros2_ws $ source install/local_setup.bash
~/ros2_ws $ ros2 launch tracetools_launch example.launch.py
[INFO] [launch]: All log files can be found below /home/example/.ros/log/2019-07-1
0 - 1 0 - 1 6 - 0 1 - 2 1 5 2 1 9
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [test_ping-1]: process started with pid [19995]
[INFO] [test_pong-2]: process started with pid [19996]
[test_ping-1] spinning
[test_ping-1] [INFO] [test_ping]: [output] some random pong string
[test_pong-2] spinning
[test_pong-2] [INFO] [test_pong]: [output] some random ping string
[INFO] [test_ping-1]: process has finished cleanly [pid 19995]
[INFO] [test_pong-2]: process has finished cleanly [pid 19996]
~/ros2_ws $
```

As shown above, you should see a few output lines, and that's it.

By default, traces are written in the `~/.ros/tracing/` directory. You can take a look at the trace's events using `babeltrace`.

```
$ cd ~/.ros/tracing/
$ babeltrace my-tracing-session/
```

If you only want to see the ROS events, you can instead do:

```
$ babeltrace my-tracing-session/ust/
```

```
_init: { cpu_id = 2 }, { procname = "test_pong", vpid = 20420, vtid = 20420 }, { p
ublisher_handle = 0x55C141B3BAE8, node_handle = 0x55C1416ABB00, rmw_publisher_hand
le = 0x55C141C20260, topic_name = "/pong", queue_depth = 10 }
[10:23:13.362567313] (+0.495375393)                              ros2:callback_star
t: { cpu_id = 3 }, { procname = "test_ping", vpid = 20419, vtid = 20419 }, { callb
ack = 0x55BA577F3FE8, is_intra_process = 0 }
[10:23:13.362682319] (+0.000115006)                              ros2:callback_end:
 { cpu_id = 3 }, { procname = "test_ping", vpid = 20419, vtid = 20419 }, { callbac
k = 0x55BA577F3FE8 }
[10:23:13.363166546] (+0.000484227)                              ros2:callback_star
t: { cpu_id = 3 }, { procname = "test_pong", vpid = 20420, vtid = 20420 }, { callb
ack = 0x55C141B3B9A0, is_intra_process = 0 }
[10:23:13.363902477] (+0.000735931)                              ros2:callback_star
t: { cpu_id = 0 }, { procname = "test_ping", vpid = 20419, vtid = 20419 }, { callb
ack = 0x55BA57755A20, is_intra_process = 0 }
[10:23:13.364346765] (+0.000444288)                              ros2:callback_end:
 { cpu_id = 3 }, { procname = "test_pong", vpid = 20420, vtid = 20420 }, { callbac
k = 0x55C141B3B9A0 }
[10:23:13.365130032] (+0.000783267)                              ros2:callback_end:
 { cpu_id = 0 }, { procname = "test_ping", vpid = 20419, vtid = 20419 }, { callbac
k = 0x55BA57755A20 }
~/.ros/tracing $
```

The last part of the `babeltrace` output is shown above. This is a human-readable version of the raw Common Trace Format (CTF) data, which is a list of events. Each event has a timestamp, an event

type, some information on the process that generated the event, and the fields corresponding to the event type. The last events of our trace are pairs of `ros2:callback_start` and `ros2:callback_end` events. Each one contains a reference to its corresponding callback.

It's now time to process the trace data! The `tracetools_analysis` package provides tools to import a trace and process it. Since reading a CTF trace is slow, it first converts it to a file which we can read much faster later on. Then we can process it to get `pandas` dataframes and use those to run analyses.

```
$ ros2 trace-analysis process ~/.ros/tracing/my-tracing-session/ust/
```

```
94834912015328  1571591844142313253  94834911844752
94834912602624  1571591844144610962  94834912603264
94342555941376  1571591844146303565  94342555942016
94834912611936  1571591844147216388  94834912612208
94342555950720  1571591844147522667  94342555950992
94834913214080  1571591844147781683  94834913214232

Callback symbols:
                        timestamp                                                symbol
callback_object
94834912603264   1571591844144764856  std::_Bind<void (rclcpp::TimeSource::*(rclcpp:...
94342555942016   1571591844146338856  std::_Bind<void (rclcpp::TimeSource::*(rclcpp:...
94834912612208   1571591844147229360  std::_Bind<void (PingNode::*(PingNode*, std::_...
94342555950992   1571591844147535327  std::_Bind<void (PongNode::*(PongNode*, std::_...
94834913214232   1571591844147787038        std::_Bind<void (PingNode::*(PingNode*))()>

Callback instances:
  callback_object            timestamp duration intra_process
0  94834913214232  1571591844647985905    81280         False
1  94834912612208  1571591844648979425   735737         False
2  94342555950992  1571591844648441867  1312481         False
================================================
processed 56 events in 274 ms
~/ros2_ws $
```

The output of the `process` command is shown above. In the last dataframe, named "Callback instances," you should see three rows. The first one is the timer callback that triggered the ping-pong sequence. The second one is the ping callback, and the third one is the pong callback! Callback function symbols are shown in the previous dataframe.

This is simple, but it isn't really nice visually. We can use a Jupyter notebook to analyze the data and display the results.

3.4 Callback duration analysis

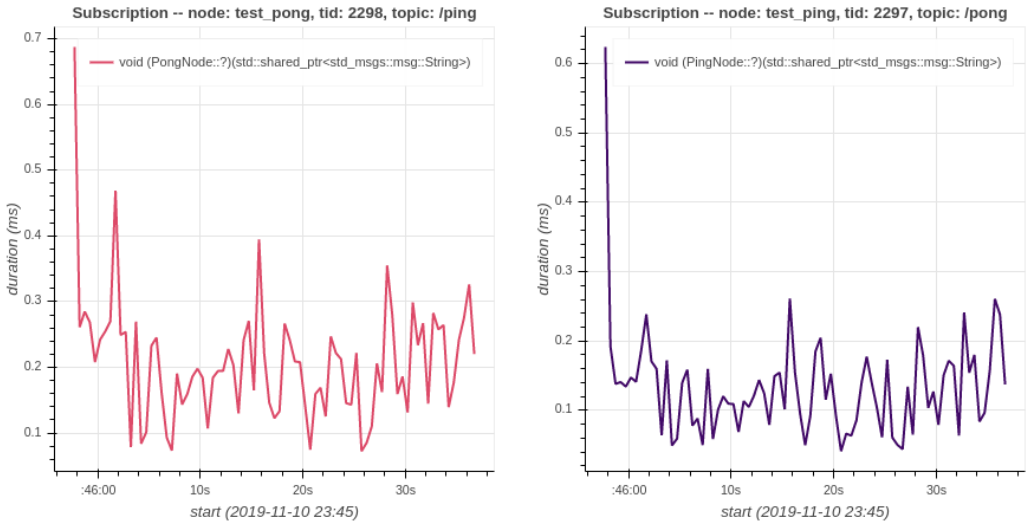
Add the following line to the arguments of each of the two `Node` objects in your launch file, which should be under `ros2_ws/src/ros2/tracing/tracetools_launch/launch/`. It will stop the nodes from shutting down after 1 exchange.

```
arguments=['do_more']
```

Delete the previous trace directory, and execute the launch file again. Let it run for some time (e.g. 10-20 seconds), then kill it with `Ctrl+C`.

To run an analysis that displays durations of callbacks over time, use this Jupyter notebook, which should be under `ros2_ws/src/tracetools_analysis/tracetools_analysis/analysis/`.

The resulting plots for the `/ping` and `/pong` subscriptions are shown below. We can see that the durations vary greatly.



3.5 Relevant links

The tracing packages can be found in the `ros2_tracing` repo. The analysis tools can be found in the `tracetools_analysis` repo.