D5.4

Micro-ROS benchmarks - Final

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D5.4
Deliverable name	Micro-ROS benchmarks - Final
Date	August 2020
Dissemination level	public
Workpackage and task	5.2
Author	Tomasz Kołcon (Łukasiewicz-PIAP), Alexandre Malki (Łukasiewicz-PIAP)
Contributors	Mateusz Maciaś (Łukasiewicz-PIAP)
Keywords	Microcontroller, NuttX, Benchmarking, Micro-ROS
Abstract	This deliverable summarizes works in Task 5.2: Benchmarking of whole stack.

Contents

1 Summary

This deliverable summarizes the works in Task 5.2: Benchmarking of whole stack. It is the third and last iteration of benchmarking of the Micro-ROS.

This document starts with explanations of whole stack benchmarking methodology. Next, benchmarking setup will be presented. Subsequently, results are presented for each benchmark along with comments. At the end, conclusions are presented.

Term	Definition
ROS	Robot Operating System
MCB	Multi-connectors board
SWO	Single Wire Output
SWD	Serial Wire Debug
SWV	Serial Wire Viewer
JTAG	Joint Test Action Group (also name of interface)
ITM	Instrumentation (or Instruction) Trace Macrocell
ETB	Embedded Trace Buffer
OS	Operating System
RTOS	Real Time Operating System
HW	HardWare
IP	Internet Protocol
UDP	User Datagram Protocol
RAM	Random Access Memory
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks.
I/O	Input/Output
ETM	Embedded Trace Macrocell
JSON	JavaScript Object Notation
UART	Universal Asynchronous Receiver-Transmitter
DDS-XRCE	DDS For Extremely Resource Constrained Environments
DDS	Data Distribution Service
KPI	Key Performance Indicators
CPU	Central Processing Unit
GCC	GNU Compiler Collection
PID	Process Identifier
CTF	Common Trace Format
SWOT	Strengths, Weaknesses, Opportunities, and Threats

2 Methodology

2.1 Introduction

Continuous integration, nowadays, are considered as one of the most important foundation of the software development cycle. Indeed, on a giant project, each developers, manager and etc... are willing to contribute asynchronously at a fast rate. Needless to add that each of theses contributors are striving to make the software to reach it's best version of itself, would eventually break something unintentionally that they do not own. The CI (Continuous Integration) answered to this issue by letting those hands provide their code in a controlled way without requiring many more resources. Very often the CI is based on a tool like Jenkins, Travis, and focus on unit tests informing whether or not one's contribution did not break something that is not related (or thought was not related) to a part the coder own. As a complement to the CI, BMs (BenchMarkings) would provide additional information such as amount of memory used by an application, time of execution, latency, real bandwidth and so on. All of the information are named KPI (Key Performance Indicators) and are here to monitor performances rather than showing what works what does not. Those performances KPI are measurable for a defined set of hardware as each hardware component has its limitations. Furthermore, comparing two different CPU would not make much sense if the goal is to improve the software performance on a chosen platform. That is why this document is going to stick with only one set of hardware devices.

Comparing to previous deliverble D5.3 [1] this time Shadow Builder was used to perfrom the most of measurements. More about Shadow Builder is on Micro-ROS website: https://micro-ros.github.io/docs/concepts/benchmarking/.

For this deliverable lastest *dashing* release of Micro-ROS was used. However, please note that a new Foxy release has recently been announced. In the upcoming deliverable D5.6 we will use the Foxy (or newer) version benachmarking results as examples of how to use the benchmarking tools.

2.2 Benchmarking

In the current document the benchmarking KPIs are:

- Execution time of function in the application and the RTOS
- Memory Usage of an application,
- And communication performances: latency, bitrate.

Nonetheless, it is worth to note that they are 2 types of benchmarks:

- Externally assisted benchmarks : Using external devices to measure the performance of device,
- Self BM: Using internal kernel or hardware functions.

2.3 Workflow

The workflow is just the methodology applied for

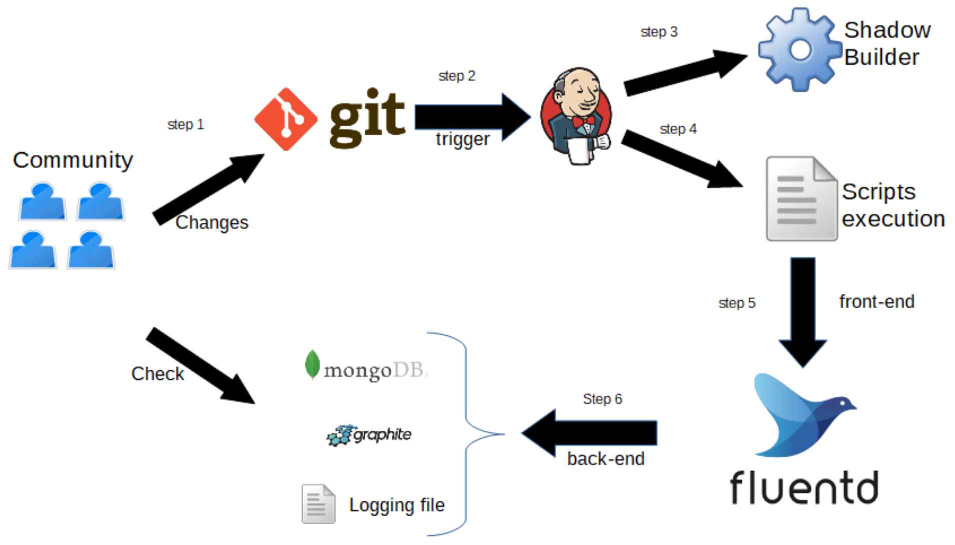

Figure 1: Workflow

All the steps are going to be described as below:

- Step 1. Getting a change from the GIT
- Step 2. & Step 3. Run the Shadow Builder
- Step 4. Run the instrumented code
- Step 5. Retrieve data (data collection),
- Step 6. Publish the results.

2.4 General methodology

Depending on what type of communication medium Micro-ROS is set up with, the methodology may vary. Of course, the variations are small and related to the topology.

In order to achieve benchmarking, the RTOS and the application were instrumented. Depending on the point of interest, different probes were placed in different part of the RTOS. The data format was following a standard called Common Trace Format (V1.8)[2]. This standard is even used in Zephyr (competitor of NuttX). As a matter of fact, the CTF core was ported from Zephyr to NuttX.

Data are retrieved and analysed with babeltrace and the babeltrace python API [3].

Every events are timed using an internal free-running timer (in the case of NuttX running on the Olimex STM32-E407 TIM2). Thanks to this the device can have time clock with the resolution of nearly 10 nanoseconds. The current configuration of the resolution is 100 nanoseconds, which is more than enough to measure perforances of the communication, considering that the minimal Ethernet (64bytes) frame at 100Mbps.

Using only the timestamped measurements allows to make delta calculation offline.

The software configuration is likely to change. However the software role will be kept the same:

- Agent running on a PC
- Subscriber running on one Olimex STM32-E407,
- Publisher running on one Olimex STM32-E407.

At a hardware level the USB - CDC/ACM console is going to be used on the Olimex STM32-E407 boards for Ethernet and Serial benchmarkings. For 6LoWPAN the serial USART6 will be used as the console to reduce the memory footprint and execution impact. Hence the USB OTG1 of both Olimex STM32-E407 boards shall be connected to a computer. Additional hardware setup must performed but will depend on the type of topology (Ethernet / Serial / 6LoWPAN):

2.4.1 Measurement repeatability and relevance

In order to make the measurement consistent, two new events were added. Those events delimit the moment when the measurement shall start and when the measurement shall stop.

This addition was necessary in order to remove some delay that may be take between the time when the device is being run and when the application under benchmark is actually running.

The CTF events that were added are:

- start_measurement: This timestamp the start of the measurement,
- stop_measurement: This timestamp the end of the measurement.

Using this method, it becomes possible to measure the length of the session and even establish a metric that would determine how performs an application with a specific hardware/software. This metric would allow one to determine what is the hardware for a specific application.

2.4.2 Communication medium methodology

Because Micro-ROS offers different communication mediums, each of them need a different hardware and software configuration as explained below:

2.4.2.1 Ethernet To start benchmarks using Ethernet communication the following setup was used:

- Five ports Gigabit switch A
- PC (agent) connect to the switch A,
- Two Olimex STM32-E407 boards connected to the switch A.

The Agent application running on the PC should be the following command line:

```
$ docker run -it --net=host microros/micro-ros-agent:dashing udp4 --port 8888 -v6
```

Note: Due to the fact that the subscriber works partially, benchmarks data will retrieved only using the publisher application

To run the benchmarks, under NuttX the **uros_XXXX_eth** where **XXXX** is a specific configuration. Once each time the environment is configured (in the docker) the whole software needs to be compiled and flashed as explained bellow:

```
> ros2 run micro_ros_setup configure_firmware.sh uros_XXXX_eth
> ros2 run micro_ros_setup build_firmware.sh
> ros2 run micro_ros_setup flash_firmware.sh
```

And the application will be executed form the nsh as following:

```
nsh> publisher
```

The Olimex STM32-E407 has to be connected to the USART3 to retrieve the benchmarking data. The sequence of action to take when starting the software.

2.4.2.2 Serial

For other medium such as Serial the methodology is similar as the above expect for the topology and the configuration.

The topology:

- The Olimex STM32-E407 should be connected over UART using the USART6 (located on the UEXT port). The USART has to be connected to the PC where the agent is running

The configuration to be used for NuttX is **uros_XXXX_serial**

The agent command the PC should be running is the following:

```
$  docker run -it --net=host microros/micro-ros-agent:dashing serial \
> --dev /dev/ttyUSBX -v6

$ # where ttyUSB X correspond to the usb / serial interface.
```

2.4.2.3 6LoWPAN

For other medium such as 6LoWPAN the methodology is similar as the above expect for the topology and the configuration.

The topology:

- The Olimex STM32-E407 should be connected to MRF24J40 module,
- The PC where the agent is running should be connected with a 6LoWPAN device (ATUSB dongle).

The configuration to be used for NuttX is **uros_XXXX_sixlowpan**

The agent command the PC should be the following:

```
$  docker run -it --net=host microros/micro-ros-agent:dashing udp6 --port 9999 -v6
```

2.4.3 Communication performance benchmarks

The way the communication is measured, is dependent on the specific low-level driver. However the methodology is the same. Two communication CTF events are sent com_start and com_finish:

- com_start: generate information about the moment when the driver received the frame from the upper layers,
- com_finish: generate information about the moment when the data was transmitted.

The following driver were patched to allow this measurement:

- stm32 Ethernet driver: using the interface name "eth"
- stm32 serial driver: using the interface name "serial"
- mrf24j40 driver: using the interface name "radio"

The configuration to run are : The configuration for NuttX is **uros_com_eth** / **uros_com_serial** / **uros_com_sixlowpan** shall be enabled built and flashed onto the Olimex STM32-E407 under test.

2.4.4 Real-time and determinism assessment and execution time per thread

Measurements in regards to the real-timeness and determinism were done using the scheduler instrumentation:

- thread_suspend: Each time a thread is being preempted this CTF event is generated,
- thread_resume: Conversely, when a thread was handed to the CPU this CTF event is generated.

The configuration for NuttX is **uros_sched_eth** / **uros_sched_serial** / **uros_shed_sixlowpan** shall be enabled built and flashed onto the Olimex STM3-E407 under test.

The interesting features for our analysis are the following:

- Scheduler set as Round Robin
- Scheduler using priority
- Scheduler frequency 100Hz,
- CTF timestamp set with an accuracy around 100 nanoseconds.

In addition, the execution methodology was changed from the previous deliverable. Previously, the tool was making use of the internal ARM debug facilities. This approach was dropped for several reasons:

- The ITM facility is not available on all microcontrollers as it's up to the manufacturer to implement it or not

- The ITM facility was sending a PC sample at deterministic frequency. However, it was not possible to determine which context (hence thread) the CPU was executing at the moment the sample was taken. So, for a baremetal, single threaded application, that would be a good solution. But on a RTOS, this is not really interesting,
- As the thread_suspend and thread_resume event had to be implemented in order to benchmark the determinism, using the ITM would have been redundant.

Once the data are retrieved over the UART, various calculation can be made to establish different metrics.

2.4.5 Static memory usage and function usage

The static memory usage and the function usage are both using the same underlying methodology. This common methodology is using the GCC instrumentation built-in function to instrument function calls and function returns. In this way it each time a function is called, a message can be logged. In the case of the static memory, the GCC instrumentation will log a event with the stack usage per PID. In the case of the function usage, each function call will be logged and each function return will be logged as well. However this method is not perfect as it introduces an overhead that can affect the execution performances. To mitigate this, only the application is instrumented.

For the stack usage only one element is generated:

- stack_usage: a per PID stack information.

For the function usage:

- function_usage: a per pid the function calls and function returns.

In order to trace the stack, the following NuttX configuration must be provided: **uros_mem_static_eth / uros_mem_static_serial / uros_mem_static_sixlowpan**

In order to trace the functions, the following NuttX configuration must be provided: **uros_func_usage_eth / uros_func_usage_serial / uros_func_usage_sixlowpan**

Once all the data are taken, accurate measurement can be done for examples, time spent in a function, amount of memory use statically etc…

2.4.6 Dynamic memory usage

The dynamic memory analysis has slightly changed its approach towards the use of CTF probing. Indeed instead of using a custom protocol and for the need to abstract the tracing, the dynamic allocation algorithm was changed. Additionally, the trace was enhanced by more information such as the PID and the free call tracing as explained below;

- heap_alloc: provides information regarding the backtrace, real size, size requested, and the pointer to the block allocated,

- heap_free: provides information regarding the pointer to the block allocated, and the real size allocated.

To trace those allocations the configuration for NuttX should be one of the following according to which communication medium you want to use with: **uros_mem_dyn_eth** / **uros_mem_dyn_serial** / **uros_mem_dyn_sixlowpan**

2.4.7 Interrupts and interrupt service routines

Interrupts may be critical for an RTOS application. Thus analysing them would be a great way to solve some performance issue that could be cause by a faulty sensor's driver configuration/code. Here the approach taken was to measure the time from the moment the RTOS is entering into the IRQs upper half dispatcher to the moment when the RTOS is leaving the upper half dispatcher. For this purpose, two ctf events were created:

- isr_enter: which is in charge of timestamping when the RTOS is entering the dispatcher. The interrupt number is provided,
- isr_exit: which is is charge of timestamping when the RTOS is leaving the dispatcher. The interrupt number is provided as well, this can be useful for preemptive interrupts.

2.5 Security assessment

Security assessment will cover most of the security flaws, possibilities those flaws are breached and how severely would the breach impact/compromise the whole system.

So far there are no implemented authentication and encryption capabilities into Micro XRCE-DDS, so it is difficult to provide complete a security assessment. As result of this point, A SWOT analysis of Micro-ROS will be conducted with some conclusions.

2.6 Communication resilience

Comparing to D5.3 [1] 6LoWPAN communication was taken into account. As before network will be disrupted in a controlled manner for testing through the use of software network emulation tools such as Linux Netem [4].

The 6LoWPAN measurements are the results of live test in conjunction with the smart warehouse use-case performed by Łukasiewicz-PIAP.

3 Results

As in deliverable D5.3 [1] all results of measurements are collected on Github repository. It was organized in this way due to the large amount of data. In addition the most of this data is difficult to analyze. For this reason, only selected data are presented below along with an analysis and with the link to specific repository.

Complete results are available here:

https://github.com/micro-ROS/benchmarking-results/tree/master/aug2020/ [5]

3.1 Ethernet

3.1.1 General measurement at low-level communication medium and latency

Here results of the communication measurement are presented.

Complete results are available here:

- [5]/ethernet_publisher/com/20200826_17_20_1598455246com_analysis.data [5].

After further analysis from the extracted data are as follow:

```
***************** Bandwidth TX *****************

Avg for eth is 18012.976894100913 kbps
Max for eth is 68437.5 kbps
Min for eth is 3575.038896423193 kbps

***************** Bandwidth RX *****************

Avg for eth is 14518.11474672164 kbps
Max for eth is 15575.677022761256 kbps
Min for eth is 14266.36637550598 kbps

***************** Latency TX *****************

Avg for eth is 49.220 micro-seconds
Max for eth is 120.191 micro-seconds
Min for eth is 19.523 micro-seconds

***************** Latency RX *****************

Avg for eth is 32.3269 micro-seconds
Max for eth is 33.333 micro-seconds
Min for eth is 30.095 micro-seconds
```

According to these results the average bandwidth is around 18 Mbps on transmission and 14 Mbps on reception.

The latency represent the time in be ween when a packet was asked to be sent and when the packet was actually sent. According to the analysis, the latency average on the transmission is 50 microseconds and 33 microseconds on reception.

According to some test done on the side using a bare NuttX, the software is using at max 89 Mbps and around 15 Mbps on the TX. The RX bandwidth is very similar on the bare NuttX. Also similar latency results were observed on the transmission and reception lines.

In conclusion, it is noticeable that the transmission is quite efficient, but not as efficient as a raw UDP application on the transmission. This might due to the fact that the bare NuttX was transmitting bigger packets on average than the publisher application. However, regarding the packet latency on transmission and reception, and on reception bandwidth latency, the publisher application shows performance that are very close to the bare NuttX.

3.1.2 Real-Time and Determinism

At this point measurements of realtimeness are presented.

Complete results are available here:

- [5]/ethernet_publisher/sched/20200827_09_42_1598514179_sched.json [5],
- [5]/ethernet_publisher/sched/hreadable.data [5].

The data extracted from the babeltrace show the following about the NuttX scheduler. The additional information are:

- The thread_id 0 is the idle thread
- The thread_id 3 is the low priority work queue,
- The thread_id 7 is the publisher.

```
[01:00:21.445833238] (+0.000009524) 0 thread_resume: { thread_id = 7 }
[01:00:21.445993047] (+0.000159809) 0 thread_suspend: { thread_id = 7 }
[01:00:21.446002761] (+0.000009714) 0 thread_resume: { thread_id = 3 }
[01:00:21.446051904] (+0.000049143) 0 thread_suspend: { thread_id = 3 }
[01:00:21.446061428] (+0.000009524) 0 thread_resume: { thread_id = 0 }
[01:00:21.446085428] (+0.000024000) 0 thread_suspend: { thread_id = 0 }
[01:00:21.446095428] (+0.000010000) 0 thread_resume: { thread_id = 3 }
[01:00:21.446133047] (+0.000037619) 0 thread_suspend: { thread_id = 3 }
[01:00:21.446142571] (+0.000009524) 0 thread_resume: { thread_id = 0 }
[01:00:21.446273523] (+0.000130952) 0 thread_suspend: { thread_id = 0 }
[01:00:21.446283523] (+0.000010000) 0 thread_resume: { thread_id = 3 }
[01:00:21.446335809] (+0.000052286) 0 thread_suspend: { thread_id = 3 }
[01:00:21.446345333] (+0.000009524) 0 thread_resume: { thread_id = 7 }
[01:00:21.446505333] (+0.000160000) 0 thread_suspend: { thread_id = 7 }
[01:00:21.446514952] (+0.000009619) 0 thread_resume: { thread_id = 3 }
[01:00:21.446564190] (+0.000049238) 0 thread_suspend: { thread_id = 3 }
[01:00:21.446573714] (+0.000009524) 0 thread_resume: { thread_id = 0 }
[01:00:21.446597714] (+0.000024000) 0 thread_suspend: { thread_id = 0 }
[01:00:21.446607714] (+0.000010000) 0 thread_resume: { thread_id = 3 }
[01:00:21.446645333] (+0.000037619) 0 thread_suspend: { thread_id = 3 }
[01:00:21.446654857] (+0.000009524) 0 thread_resume: { thread_id = 0 }
```

```
[01:00:21.446779047] (+0.000124190) 0 thread_suspend: { thread_id = 0 }
[01:00:21.446789142] (+0.000010095) 0 thread_resume: { thread_id = 3 }
[01:00:21.446841333] (+0.000052191) 0 thread_suspend: { thread_id = 3 }
[01:00:21.446850857] (+0.000009524) 0 thread_resume: { thread_id = 7 }
[01:00:21.447010571] (+0.000159714) 0 thread_suspend: { thread_id = 7 }
[01:00:21.447020285] (+0.000009714) 0 thread_resume: { thread_id = 3 }
```

According to the results taken above, the software is running a deterministic way. Indeed, by looking closer, it is noticeable that the running sequence is the same.

Additionally, timing deltas between correlated events have a really low variation when switch (consecutive thread_suspend/thread_resume).

Moreover, the scheduler performs fast context switches, on average the are around 10 microseconds.

3.1.3 Execution

Based on the same result taken from the scheduler the metrics were extracted. The software showed that most of the time the application is waiting for the packet to be sent away to the the application.

Complete results are available here:

- [5]/ethernet_publisher/sched/20200827_09_42_1598514179_sched.json [5],
- [5]/ethernet_publisher/sched/hreadable.data [5].

In general the result we've gotten are showing that the application was running for around 2.25s. During this time, most of the time was take by the idle thread ~81% of the time and reset was split between the lpwork ~10 ms (task in charge of collecting packet and sending them to the Ethernet). The publisher itself was running around 11ms as well. Those two tasks are co-dependent, therefore the application was executing for around 20ms.

This does not mean that the application was only using less than 1% of the CPU. The application was waiting most of it's time ~19% of the time. The issue is mostly coming from IO operations.

3.1.4 Functions usage

Here the functions usage is presented.

Complete results are available here:

- [5]/ethernet_publisher/fusage/20200828_13_29_1598614159_fusage_analysis_hits.json [5],
- [5]/ethernet_publisher/fusage/hreadable.data [5].

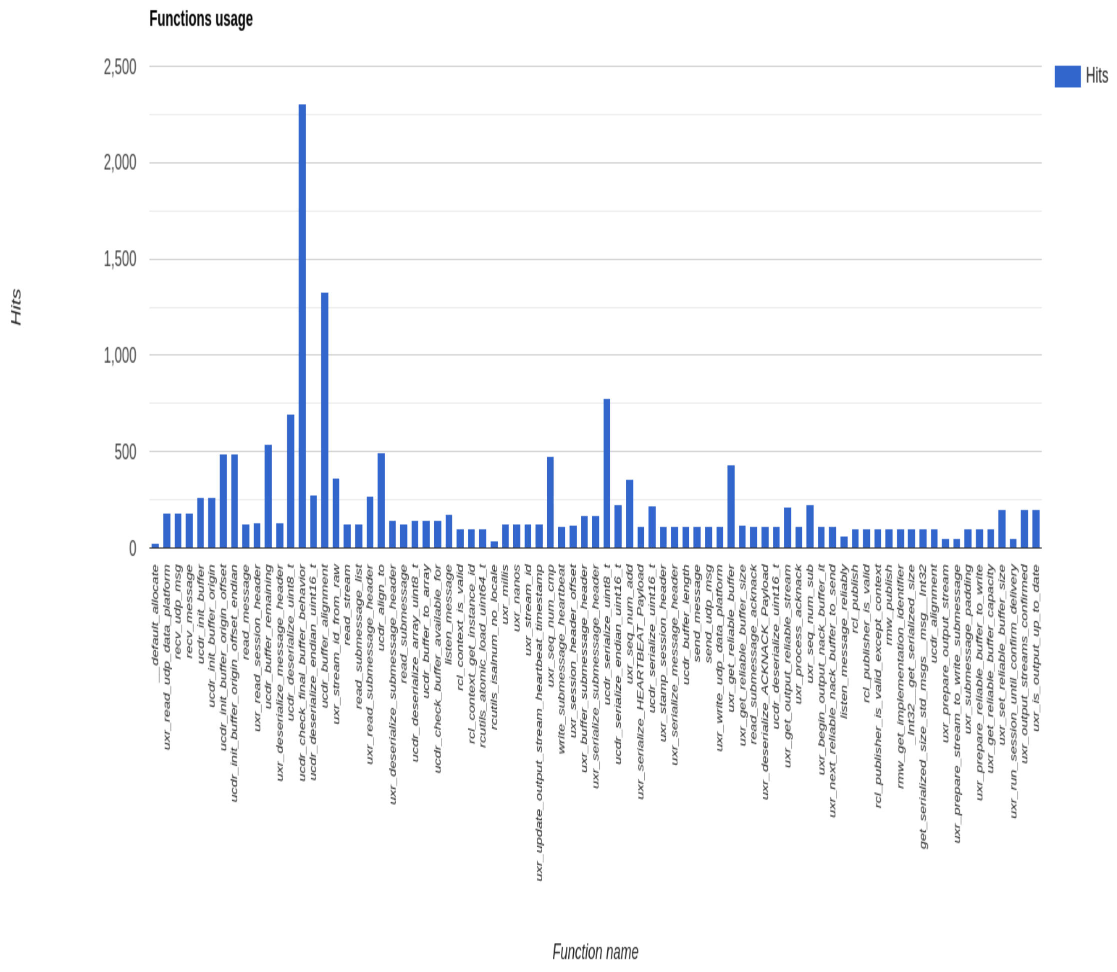

Figure 2: Function usage for Ethernet (shown only hits >20)

The functions that are called the most often (> 500 times) are:

- *ucdr_check_final_buffer_behavior* around 1154 times,
- *ucdr_buffer_allignement* around 664 times.

3.1.5 Static usage

At this point static memory usage is presented.

Complete results are available here:

- [5]/ethernet_publisher/mem-static/20200828_13_53_1598615615_mem_static.json [5],
- [5]/ethernet_publisher/mem-static/hdreadble.data [5].

Below the static memory usage results:

```
{"publisher": {"stack_used_bytes": 12568, "delim": 0}}
```

According to the results the amount of memory that the simple publisher is occupying is around 12568 byte of stack memory.

3.1.6 Heap allocation resources

Below are the results of dynamic memory usage measurements.

Complete results are available here:

- [5]/ethernet_publisher/mem-dyn/20200828_14_08_1598616535_mem_dyn.json [5],
- [5]/ethernet_publisher/mem-dyn/hreadable.data [5].

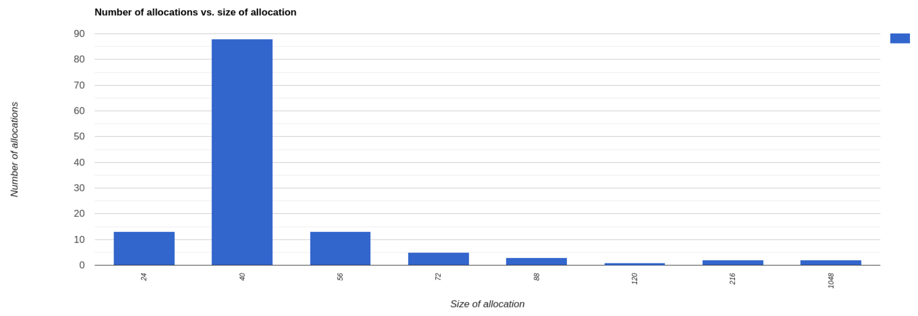

Figure 3: Dynamic memory allocations for Ethernet

Regarding the heap allocation results, it's noticeable that:

The biggest size that is allocated twice by the publisher is 208 bytes allocated by *rcutils_hash_map_unset*

The most of the allocations are done by the application to print the result and the rest is very small in total less that 2KB around. The Micro-ROS is mostly using stack memory.

3.1.7 Power usage

The results are available here:

- [5]/ethernet_publisher/pwr/measure_info.txt [5],
- [5]//ethernet_publisher/pwr/pwr_oscilloscope.PNG [5].

According to the measurements made, the Ethernet the publisher is using around 750mW of energy.

3.1.8 Interrupts and interrupt service routines

At this point interrupts measurements are presented.

The results are available here:

- [5]/ethernet_publisher/isr/20200831_07_32_1598938336_isr.json [5],
- [5]/ethernet_publisher/isr/hreadable.data [5].

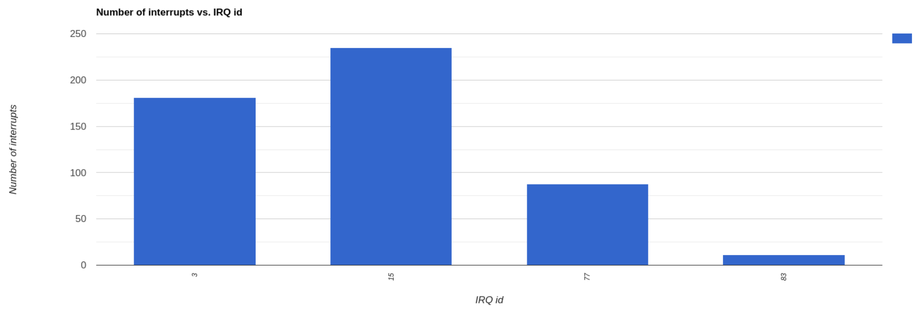

Figure 4: Interrupt number for Ethernet

As shown on the chart above, different interrupts are:

- Interrupt 3: Corresponds to the hard fault. Of course, this is not a hard fault. In NuttX, it's used for context switching
- Interrupt 15: Corresponds to the System Tick
- Interrupt 77: Corresponds to the Ethernet interrupt,
- Interrupt 83: Corresponds to the OTG USB. This is to the USB uart console.

3.1.9 Communication resilience

After running the different tests, the communication resilience results of Micro-ROS bring similar results as the previous deliverable. In a nuttshell, when the connection is failing or is unstable, the Micro-ROS on NuttX does not provide a failsafe solution.

More details regarding the result can be found here with the output of the agent and the output of the client.

- [5]/ethernet_publisher/com_resilience/publisher_delay_1000ms.txt [5]
- [5]/ethernet_publisher/com_resilience/publisher_delay_5000ms.txt [5]
- [5]/ethernet_publisher/com_resilience/publisher_lost_10_pourcents.txt [5]
- [5]/ethernet_publisher/com_resilience/publisher_lost_60_pourcents.txt [5]
- [5]/ethernet_publisher/com_resilience/publisher_currupted_1_pourcent.txt [5],
- [5]/ethernet_publisher/com_resilience/publisher_currupted_10_pourcents.txt [5].

Additionally the packet captures are also available:

- [5]/ethernet_publisher/com_resilience/publisher_delay_1000ms.pcapng [5]
- [5]/ethernet_publisher/com_resilience/publisher_delay_5000ms.pcapng [5]
- [5]/ethernet_publisher/com_resilience/publisher_lost_10_pourcents.pcapng [5]
- [5]/ethernet_publisher/com_resilience/publisher_lost_60_pourcents.pcapng [5]
- [5]/ethernet_publisher/com_resilience/publisher_currupted_1_pourcent.pcapng [5],
- [5]/ethernet_publisher/com_resilience/publisher_currupted_10_pourcents.pcapng [5].

3.2 Serial

3.2.1 General measurement at low-level communication medium and latency

Here results of communication measurement are presented.

Complete results are available here:

- [5]/serial_publisher/com/20200827_12_30_1598524236_com_analysis.data [5]

After further analysis from the relevant extracted data are as follow:

```
***********  TX Bandwith in kbps     **********
Avg for uart is 573.5415218805457
Max for uart is 603.1886967263744
Min for uart is 0.0

***********  RX Bandwith in kbps      ***********
Avg for uart is 583.1950581671621
Max for uart is 590.1571234325427
Min for uart is 569.6317900109369

***********  TX Latency  in micro-sec ***********
Avg for uart is 13.531629411764706
Max for uart is 21.715
Min for uart is 12.952

***********  RX Latency in micro-sec  ***********
Avg for uart is 13.39688
Max for uart is 13.715
Min for uart is 13.238
```

We can see that on average the baudrate is reaching:

- TX an average of 573 kbits/seconds which translate to around 75000 bauds.
- RX an average of 580 kbits/seconds which translate to around 75000 bauds.

The UART was configured to run 115200 bauds per seconds. To compare with bare RTOS (NuttX), the number on RX and on TX are really similar.

3.2.2 Real-Time and Determinism

At this point measurements of realtimeness are presented.

Complete results are available here:

- [5]/serial_publisher/sched/20200829_11_21_1598692884_sched.json [5],
- [5]/serial_publisher/sched/hreadable.data [5].

The data extracted from the babeltrace show the following on the NuttX scheduler. The additional information are:

- The thread_id 0 is the idle thread
- The thread_id 3 is the low priority work queue,
- The thread_id 6 is the publisher.

```
[01:00:11.544279047] (+0.000010762) 0 thread_resume: { thread_id = 3 }
[01:00:11.544300857] (+0.000021810) 0 thread_suspend: { thread_id = 3 }
[01:00:11.544310285] (+0.000009428) 0 thread_resume: { thread_id = 6 }
[01:00:11.544394476] (+0.000084191) 0 thread_suspend: { thread_id = 6 }
[01:00:11.544404000] (+0.000009524) 0 thread_resume: { thread_id = 0 }
[01:00:11.563101428] (+0.018697428) 0 thread_suspend: { thread_id = 0 }
[01:00:11.563111333] (+0.000009905) 0 thread_resume: { thread_id = 6 }
[01:00:11.563276190] (+0.000164857) 0 thread_suspend: { thread_id = 6 }
[01:00:11.563285809] (+0.000009619) 0 thread_resume: { thread_id = 0 }
[01:00:11.564952952] (+0.001667143) 0 thread_suspend: { thread_id = 0 }
[01:00:11.564962761] (+0.000009809) 0 thread_resume: { thread_id = 6 }
[01:00:11.565003428] (+0.000040667) 0 thread_suspend: { thread_id = 6 }
[01:00:11.565013047] (+0.000009619) 0 thread_resume: { thread_id = 0 }
[01:00:11.565043142] (+0.000030095) 0 thread_suspend: { thread_id = 0 }
[01:00:11.565053142] (+0.000010000) 0 thread_resume: { thread_id = 6 }
[01:00:11.565094476] (+0.000041334) 0 thread_suspend: { thread_id = 6 }
[01:00:11.565104000] (+0.000009524) 0 thread_resume: { thread_id = 0 }
[01:00:11.565129428] (+0.000025428) 0 thread_suspend: { thread_id = 0 }
[01:00:11.565139333] (+0.000009905) 0 thread_resume: { thread_id = 6 }
[01:00:11.565176285] (+0.000036952) 0 thread_suspend: { thread_id = 6 }
[01:00:11.565185904] (+0.000009619) 0 thread_resume: { thread_id = 0 }
[01:00:11.565212380] (+0.000026476) 0 thread_suspend: { thread_id = 0 }
[01:00:11.565222285] (+0.000009905) 0 thread_resume: { thread_id = 6 }
[01:00:11.565259238] (+0.000036953) 0 thread_suspend: { thread_id = 6 }
[01:00:11.565268761] (+0.000009523) 0 thread_resume: { thread_id = 0 }
[01:00:11.565302380] (+0.000033619) 0 thread_suspend: { thread_id = 0 }
```

The analysis is the same as the one for the Ethernet. Showing that there are no differences regarding the RTOS, when the communication medium is different.

3.2.3 Execution

Here are the results of program execution time during communication cycle.

Complete results are available here:

- [5]/serial_publisher/sched/20200829_11_21_1598692884_sched.json [5],
- [5]/serial_publisher/sched/hreadable.data [5].

According to the analysis taken from the data, the total time of execution is around 13.04 seconds. The time of execution is 3 times slower than the time of execution over Ethernet.

The application was idle during 36% of the time meaning that the CPU was very busy. According to the numbers, the application is only running for about 30ms. The lpwork queue thread is executing around 10ms. The rest of the time the publisher application is waiting for IO completions and low-level kernel function that are taking about 60% of the time.

3.2.4 Functions usage

Here the functions usage is presented.

Complete results are available here:

- [5]/serial_publisher/fusage/20200828_16_22_1598624538_fusage_analysis_hits.json [5],
- [5]/serial_publisher/fusage/hreadable.data [5].

After extraction of the information from the generated trace the data are plotted as follow:

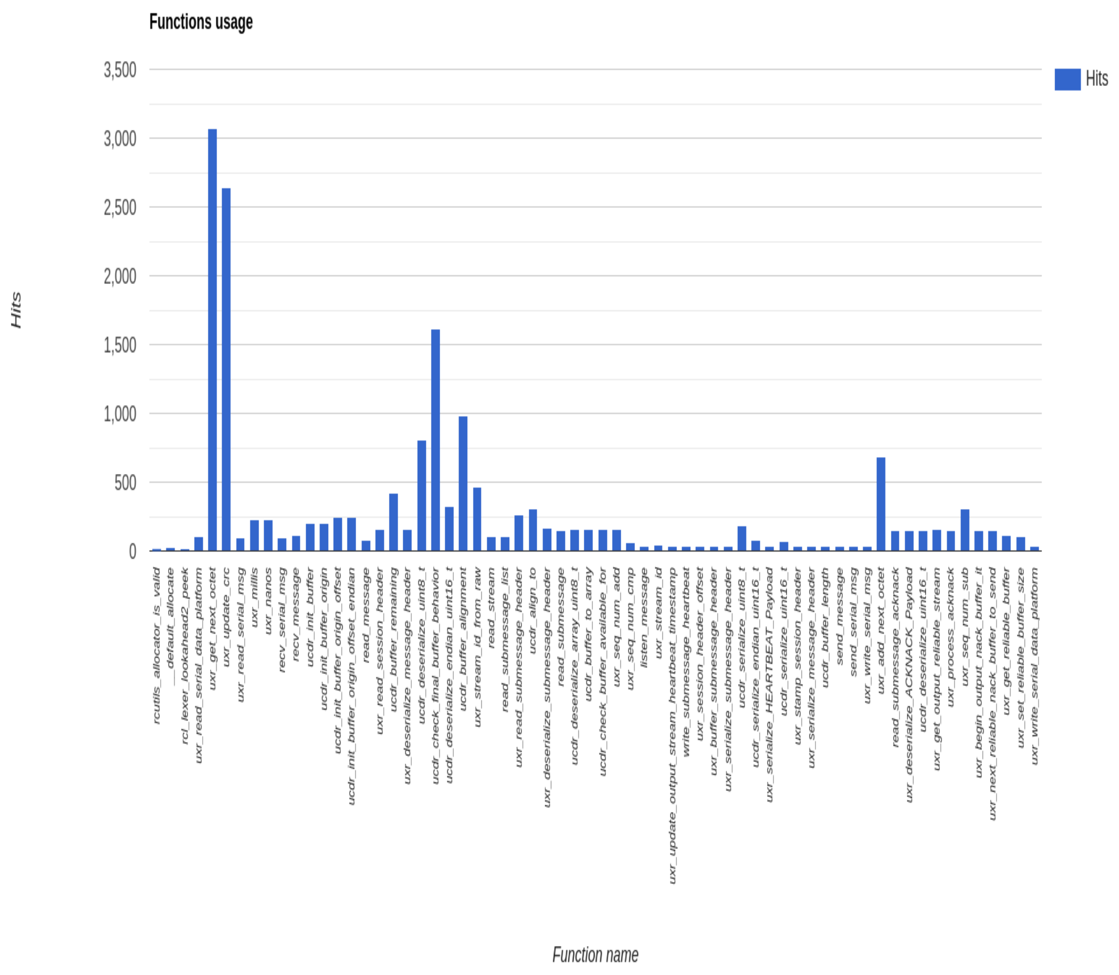

Figure 5: Function usage for Serial (shown only hits >20)

This picture is showing that the most of the time is spent in the functions:

- *uxr_get_next_octect* over 3000 times
- *update_next_crc* around 2600 times
- *ucdr_check_final_buffer_behavior* over 1600 times,
- *ucdr_buffer_allignement* a little lis 1000 times.

3.2.5 Static usage

At this point static memory usage is presented.

Complete results are available here:

- [5]/serial_publisher/mem-static/20200828_15_44_1598622287_mem_static.json [5],

- [5]/serial_publisher/mem-static/hreadable.data [5].

Below the static memory usage results:

```
{"publisher": {"stack_used_bytes": 12504, "delim": 0}}
```

According to the results, the amount of memory that the simple publisher is occupying is around 12504 bytes of stack memory. In comparison to the Ethernet, the stack usage is lower but the difference is very small.

3.2.6 Heap allocation resources

Below are the results of dynamic memory usage measurements.

Complete results are available here:

- [5]/serial_publisher/mem-dyn/20200828_15_20_1598620837_mem_dyn.json [5],
- [5]/serial_publisher/mem-dyn/hreadable.data [5].

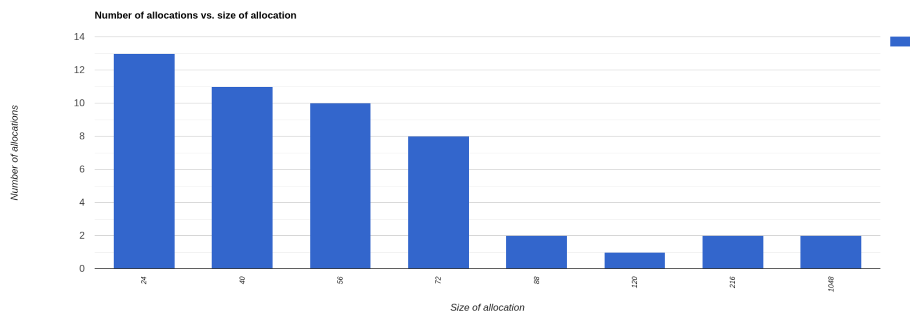

Figure 6: Dynamic memory allocations for Serial

According to the data the number of dynamic allocation is relatively low. The size of allocations vary between 24 and 216 bytes of memory. The total of simultaneously allocated memory does not go above 1KByte of data. Reducing dynamic allocation leads to better performances and reduce the potential memory leaks and/or security flaws.

3.2.7 Power usage

The results are available here:

- [5]/serial_publisher/pwr/measure_info.txt [5],
- [5]/serial_publisher/pwr/pwr_oscilloscope.PNG [5].

According to the measurements made, the Serial publisher is using around 500mW of energy.

3.2.8 Interrupts and interrupt service routines

At this point interrupts measurements are presented.

The results are available here:

- [5]/serial_publisher/isr/20200831_07_33_1598938415_isr.json [5],
- [5]/serial_publisher/isr/hreadable.data [5].

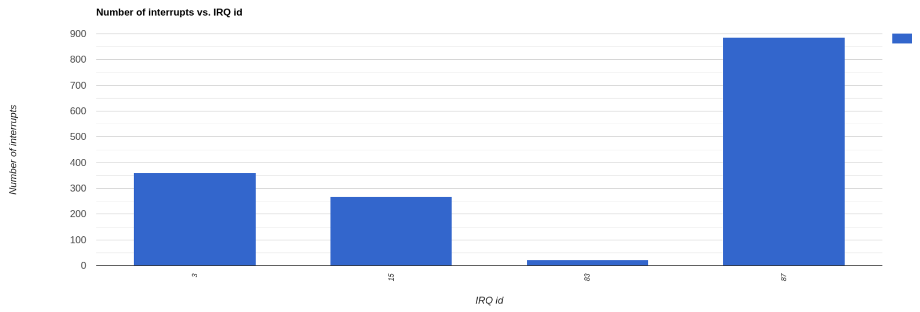

Figure 7: Interrupts number for Serial

As shown on the chart above, different interrupts are:

- Interrupt 3: Corresponds to the hard fault. Of course, this is not a hard fault. In NuttX it's used for context switching
- Interrupt 15: Corresponds to the system tick
- Interrupt 83: Corresponds to the OTG USB. This correspond to the USB uart console,
- Interrupt 87: Corresponds to the USART6 interrupt. It has the highest number because of the transmission happening over UART on each byte transmitted and received.

3.3 6LoWPAN

It should be noted that communication measurement results may be unpredictable due to the nature of radio communication.

3.3.1 General measurement at low-level communication medium and latency

Here results of communication measurements are presented. The results for the 6LoWPAN are different the measurements only show the SPI transfer speed between the CPU and the MRF24J40.

Complete results are available here:

```
***********  TX Bandwith in kbps     **********
Avg for radio is 373.46072769792994
Max for radio is 549.911121436277
```

```
Min for radio is 17.28179009482449

***********  RX Bandwith in kbps      ***********
Avg for radio is 758.5569422017933
Max for radio is 773.6098860696125
Min for radio is 746.8706121351537

***********  TX Latency  in micro-sec ***********
Avg for radio is 2184.4498952380955
Max for radio is 7233.047
Min for radio is 250.286

***********  RX Latency in micro-sec   ***********
Avg for radio is 499.22377500000005
Max for radio is 545.333
Min for radio is 494.571
```

The results show very fast transfers between the CPU and the MRF24J40. Those transfer does not only concerns the data itself, it also consider different configuration element.

The measured time based on the capture from wireshark shows a bit rate of around 31kpbs for the duration of the total application execution.

After analysing each packet independently when the connection is established and the publisher is sending its data, the instant bit rate is around 130kbps.

3.3.2 Real-Time and Determinism

At this point measurements of realtimeness are presented.

Complete results are available here:

- [5]/sixlowpan_publisher/sched/20200831_08_01_1598940069_sched.json [5],
- [5]/sixlowpan_publisher/sched/hreadable.data [5].

The data extracted from the babeltrace show the following on the NuttX scheduler. The additional information are:

- The thread_id 0 is the idle thread
- The thread_id 3 is the low priority work queue,
- The thread_id 6 is the publisher.

```
[01:00:07.076325142] (+0.000010095) 0 thread_resume: { thread_id = 3 }
[01:00:07.076346285] (+0.000021143) 0 thread_suspend: { thread_id = 3 }
[01:00:07.076355333] (+0.000009048) 0 thread_resume: { thread_id = 0 }
[01:00:07.999528285] (+0.923172952) 0 thread_suspend: { thread_id = 0 }
[01:00:07.999537809] (+0.000009524) 0 thread_resume: { thread_id = 3 }
```

```
[01:00:07.999567238] (+0.000029429) 0 thread_suspend: { thread_id = 3 }
[01:00:07.999576190] (+0.000008952) 0 thread_resume: { thread_id = 0 }
[01:00:08.079523523] (+0.079947333) 0 thread_suspend: { thread_id = 0 }
[01:00:08.079533142] (+0.000009619) 0 thread_resume: { thread_id = 3 }
[01:00:08.080424476] (+0.000891334) 0 thread_suspend: { thread_id = 1 }
[01:00:08.080434571] (+0.000010095) 0 thread_resume: { thread_id = 3 }
[01:00:08.080455619] (+0.000021048) 0 thread_suspend: { thread_id = 3 }
[01:00:08.080464666] (+0.000009047) 0 thread_resume: { thread_id = 0 }
[01:00:08.539523523] (+0.459058857) 0 thread_suspend: { thread_id = 0 }
[01:00:08.539533142] (+0.000009619) 0 thread_resume: { thread_id = 6 }
[01:00:08.539591047] (+0.000057905) 0 thread_suspend: { thread_id = 6 }
[01:00:08.539600285] (+0.000009238) 0 thread_resume: { thread_id = 3 }
[01:00:08.539661047] (+0.000060762) 0 thread_suspend: { thread_id = 3 }
[01:00:08.539670095] (+0.000009048) 0 thread_resume: { thread_id = 2 }
[01:00:08.540434190] (+0.000764095) 0 thread_suspend: { thread_id = 2 }
[01:00:08.540443238] (+0.000009048) 0 thread_resume: { thread_id = 3 }
[01:00:08.540469142] (+0.000025904) 0 thread_suspend: { thread_id = 3 }
[01:00:08.540478190] (+0.000009048) 0 thread_resume: { thread_id = 6 }
[01:00:08.540590857] (+0.000112667) 0 thread_suspend: { thread_id = 6 }
[01:00:08.540600000] (+0.000009143) 0 thread_resume: { thread_id = 3 }
[01:00:08.540664857] (+0.000064857) 0 thread_suspend: { thread_id = 3 }
[01:00:08.540674190] (+0.000009333) 0 thread_resume: { thread_id = 2 }
[01:00:08.540696761] (+0.000022571) 0 thread_suspend: { thread_id = 2 }
[01:00:08.540705809] (+0.000009048) 0 thread_resume: { thread_id = 3 }
[01:00:08.540734857] (+0.000029048) 0 thread_suspend: { thread_id = 3 }
[01:00:08.540743809] (+0.000008952) 0 thread_resume: { thread_id = 6 }
[01:00:08.540853523] (+0.000109714) 0 thread_suspend: { thread_id = 6 }
[01:00:08.540862666] (+0.000009143) 0 thread_resume: { thread_id = 3 }
[01:00:08.540927333] (+0.000064667) 0 thread_suspend: { thread_id = 3 }
[01:00:08.540936380] (+0.000009047) 0 thread_resume: { thread_id = 2 }
[01:00:08.540958952] (+0.000022572) 0 thread_suspend: { thread_id = 2 }
[01:00:08.540968000] (+0.000009048) 0 thread_resume: { thread_id = 3 }
[01:00:08.540997047] (+0.000029047) 0 thread_suspend: { thread_id = 3 }
[01:00:08.541006000] (+0.000008953) 0 thread_resume: { thread_id = 6 }
[01:00:08.541115619] (+0.000109619) 0 thread_suspend: { thread_id = 6 }
```

The analysis is the same as the one for the Ethernet. Showing that there are no differences when the communication medium is different.

3.3.3 Execution

Based on the same result taken from the scheduler the metrics were extracted. The software showed that most of the time the application is waiting for the packet to be sent away to the the application.

Complete results are available here:

- [5]/sixlowpan_publisher/sched/20200831_08_01_1598940069_sched.json [5],

- [5]/sixlowpan_publisher/sched/hreadable.data [5].

In general the results are showing that the application was running for around 3.50s. During this time, most of the time was take by the idle thread ~77% of the time and reset was split between the lpwork ~26 ms (task in charge of collecting packet and sending them to the ether net) and the hpwork which takes as well around 13ms the The publisher itself was running around 11ms as well. The previously mentioned tasks (hpwork and lpwork) are dependent on the publisher task. This means that the application is running around 50ms.

This does not mean that the application was only using less than 1% of the CPU. The application was waiting most of it's time ~22% of the time. The issue is mostly coming from IO operations.

3.3.4 Functions usage

Here the functions usage is presented. Complete results are available here:

- [5]/sixlowpan_publisher/fusage/20200831_11_30_1598952626_fusage_analysis_hits.json [5]
- [5]/sixlowpan_publisher/fusage/hreadable.data [5]

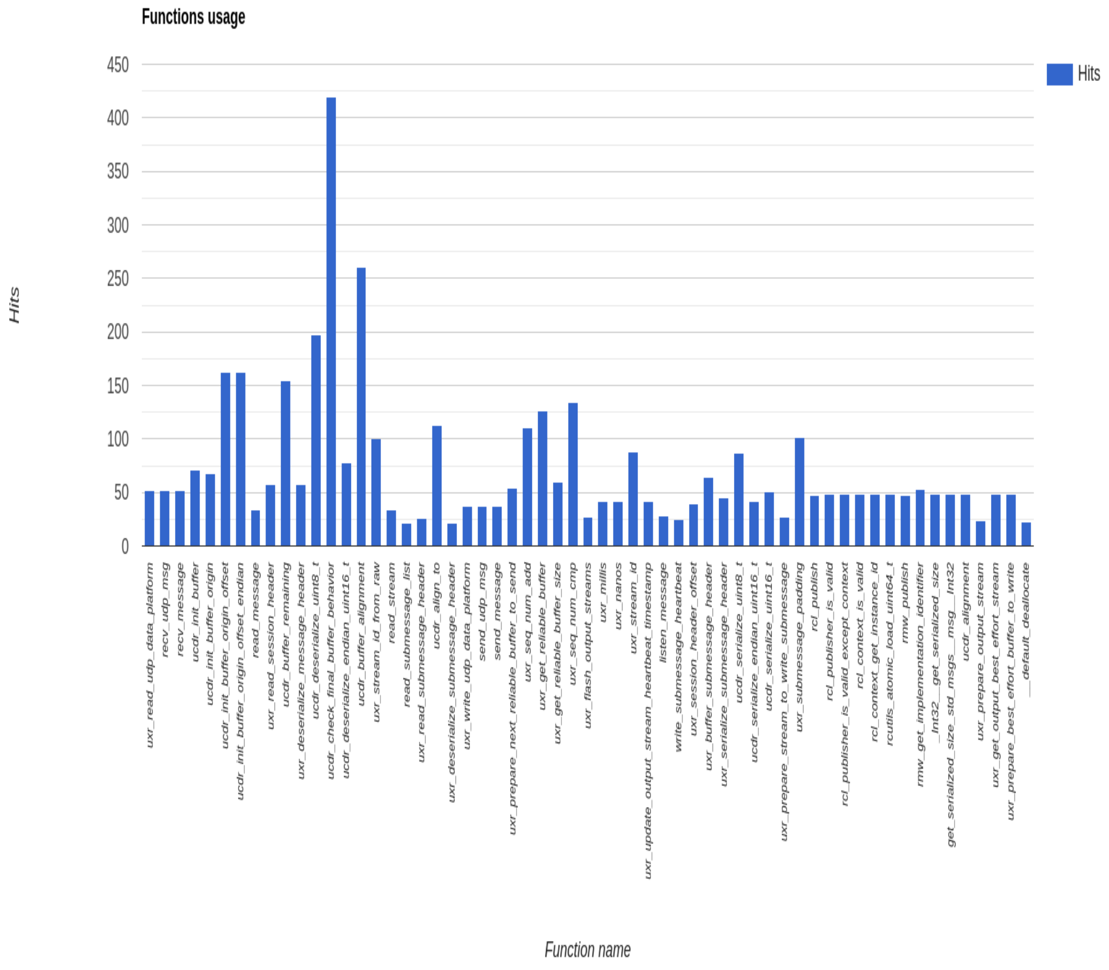

Figure 8: Function usage for Ethernet (shown only hits >20)

The functions that are called the most often (> 500 times) are:

- *ucdr_check_final_buffer_behavior* around 1154 times
- *ucdr_buffer_allignement* around 664 times

3.3.5 Static usage

At this point static memory usage is presented.

Complete results are available here:

- [5]/sixlowpan_publisher/mem-static/20200830_10_30_1598949024_mem_static.data [5],
- [5]/sixlowpan_publisher/mem-static/hreadable.data [5].

Below the static memory usage results:

```
{"publisher": {"stack_used_bytes": 13840, "delim": 0}}
```

According to the result, the amount of memory that the simple publisher is occupying is around 13840 bytes of stack memory. It's around 1.5Kbytes more than the Ethernet/Serial.

3.3.6 Heap allocation resources

Below are the results of dynamic memory usage measurements.

Complete results are available here:

- [5]/sixlowpan_publisher/mem-dyn/20200830_09_53_1598946833_mem_dyn.json [5]
- [5]/sixlowpan_publisher/mem-dyn/hreadable.data [5]

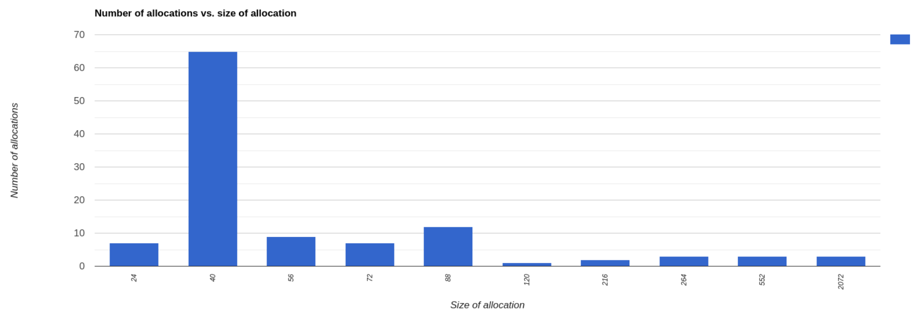

Figure 9: Dynamic memory allocations for 6LoWPAN

Regarding the heap allocation results, it's noticeable that:

The maximum size that is allocated twice by the publisher is 208 bytes allocated by *rcl_publisher_init*.

The most of the allocations are done by the application to print the result and the rest is very small, in total less that 2KB around. The Micro-ROS is mostly using stack memory.

3.3.7 Power usage

The results are available here:

- [5]/sixlowpan_publisher/pwr/measure_info.txt [5],
- [5]/sixlowpan_publisher/pwr/pwr_oscilloscope.PNG. [5]

According to the measurement made, the 6LoWPAN publisher is using around 750mW of energy.

3.3.8 Interrupts and interrupt service routines

At this point interrupts measurements are presented.

The results are available here:

- [5]/sixlowpan_publisher/isr/20200831_07_33_1598938415_isr.json [5],
- [5]/sixlowpan_publisher/isr/hreadable.data [5].

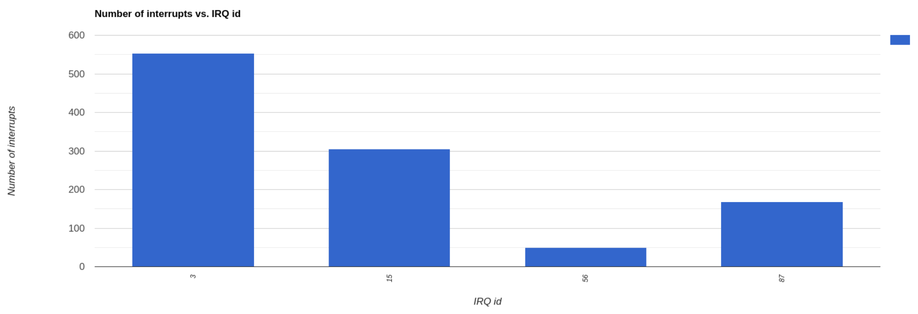

Figure 10: Interrupts number for Serial

As shown on the chart above, different interrupts are:

- Interrupt 3: Corresponds to the hard fault. Of course, this is not a hard fault. In NuttX it's used for context switching
- Interrupt 15: Corresponds to the system tick
- Interrupt 87: Corresponds to the USART6. This correspond to the uart console,
- Interrupt 56: Corresponds to the external interrupt line 10 to 15. This correspond to the MRF24J40 interrupt located on the PIN 12 of the Olimex STM32-E407 board.

3.3.9 Communication resilience

On the other hand many practical communication tests were performed during smart warehouse use-case preparation. The results of communication resilience are presented below.

Testing 6LoWPAN with 4 devices connected to Micro-ROS Agent it was noticed that:

- the 6LoWPAN devices are sensitive to the presence of too many devices operating at the same frequency leading to connection drops and delays
- data integrity in the event of transmission disruptions is assured
- in case of single device, no significant transmission delays were found,
- two-way communications possible (publisher/subscriber).

3.4 Security assessment

Micro-ROS uses the resource-optimized DDS for Extremely Resource Constrained Environments (DDS-XRCE) standard, implemented by eProsima's Micro-XRCE-DDS. As was mentioned before there are no implemented authentication and encryption capabilities into Micro XRCE-DDS. This feature is difficult to implement in microcontrollers, as authentication and encryption are computationally expensive. On the other side, Fast DDS and its RTPS wire protocol, mediating the interaction between the agent and the DDS world do implement security. It is a part of the protocol external to the Micro-ROS project. So, it is not planned to make any specific benchmarking regarding security in Fast DDS.

Despite these difficulties, a short analysis of Micro-ROS security will be presented. There are many methodoligies to do security assesment. For this this chapter, methodology described in [6] was used.

3.4.1 SWOT

SWOT analysis is a tool that help to identify strengths, weakness, opportunities and threats. This type of analysis is mainly used in related to business competition or project planning [7], but can be also used in other fields.

3.4.1.1 Strengths

- DDS implementation
- maintained by open-source community
- assured messages integrity,
- fully configurable QoS (Quality od Service) settings.

3.4.1.2 Weaknesses

- no implemented authentication capabilities,
- no implemented encryption capabilities.

3.4.1.3 Opportunities

- due to open-source license security can be developed / maintained by community
- better communication quality than in the case of custom protocol,
- authentification and encryption can be imlemented on microcontrollers with built-in security features like hardware crypto module.

3.4.1.4 Threats

- easy fake data injection (especially with radio connection)
- possibility to sniff the packets,
- sensitivity to overload due to limited resources of microcontrollers.

3.4.2 Conclusions and recommendations

The issue of security in communication with microcontrollers is specific. For the direct cable connections (Ethernet and Serial) authentification and encryption is not as important as for wireless connections (6LoWPAN). Therefore, when using radio communication, special attention must be paid to the above-mentioned risks. Despite the lack of encryption, the transmission can be secured at higher software levels. For example, using FIWARE Context Broker token for the operation can be generated, so it is impossible to make operation without the supervisor's permission. If encryption is required, a more efficient processor can be used, for example STM32 with security features like hardware crypto module [8]. Because XRCE-DDS is based on DDS it takes all its advantages, including QoS mechanisms.

4 Conclusions

In general we can see that the application is performing good:

- The amount of RAM used by the application itself is low ~15KBytes (not including) the RTOS.
- The communication performance are not to far from the performance of a bare application max -20% of performances (the Ethernet showed bigger differences)
- The RTOS on which the publisher application is running has really earned its name of Real-Time Operating System because the scheduler performs in an deterministic way that is predictable.
- The scheduler latency is below 10 microseconds and very repetitive.

Also, those benchmarks target the application itself which means that the lower layer are not taken into consideration. For instance the application is running different medium that can take a certain amount of memory (drivers are dynamically allocating some buffers and other needed structures).

The choice of the medium will be depending on the performance and what is possible to achieve in the environement where the micro-ROS based device will be running in. For instance:

- For low power two solution are available: UART/6LoWPAN
- For performance: Ethernet,
- For EMC constrained environement: UART.

The obtained results show that micro-ROS can be successfully used at the present stage of development.

References

[1] M. M. Tomasz Kołcon Alexandre Malki, 'D5.3'. [Online]. Available: http://ofera.eu/storage/deliverables/M16/OFERA_54_D53_Micro-ROS_benchmarks_-_Revised.pdf

[2] L. Foundation, 'Common Trace Format'. [Online]. Available: https://diamon.org/ctf/

[3] 'Babeltrace'. [Online]. Available: https://babeltrace.org/

[4] S. Hemminger, 'Netem tool'. [Online]. Available: https://wiki.linuxfoundation.org/networking/netem

[5] PIAP, 'D5.4 results'. [Online]. Available: https://github.com/micro-ROS/benchmarking-results/tree/master/aug2020

[6] D. Baghdasarin, 'MRO cybersecurity swot', *International Journal of Aviation, Aeronautics, and Aerospace*, vol. 6, no. 1, p. 9, 2019.

[7] Wikipedia, 'SWOT analysis'. [Online]. Available: https://en.wikipedia.org/wiki/SWOT_analysis

[8] 'Introduction to stm32 microcontrollers security - an5156'. https://www.st.com/content/ccc/resource/technical/document/application_note/group1/9f/0b/e4/b6/75/15/4f/e2/DM00493651/files/DM00493651.pdf/jcr:content/translations/en.DM00493651.pdf.