D5.3

Micro-ROS benchmarks - Revised

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D5.3
Deliverable name	Micro-ROS benchmarks - Revised
Date	December 2019
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
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 second iteration of benchmarking of the micro-ROS.

This document starts with explanations of whole stack benchmarking procedures. This is followed by descriptions of hardware and software tools used. 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
MTB	Micro Trace Buffer
DWT	Debug, Watchpoint and Trace
TPIU	Trace Port Interface Unit
OS	Operating System
RTOS	Real Time Operating System
HW	HardWare
IP	Internet Protocol
TCP	Transmission Control 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

2 Methodology

Benchmarking will be splitted into two parts, one with measurements and result interpretation. In this section, the focus will be brought on measurement methodologies that provide data for later interpretation in a way that reduces the targeted benchmarked application's overhead.

The communication benchmarks methodologies are going to be introduced together with reason they were chosen.

2.1 General topology

The topology we are going to use is as follow:

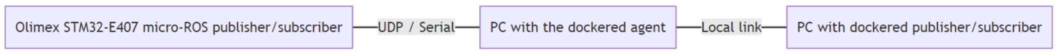

Olimex STM32-E407 micro-ROS publisher/subscriber
UDP / Serial
PC with the dockered agent
Local link
PC with dockered publisher/subscriber

Figure 1: General topology

The communication between the reference board and the PC is using the micro-ROS-agent, that is instantiated inside docker container.

2.2 Performance and communication benchmarks

The communication benchmarking will be done using both the Serial and the Ethernet connection.

2.2.1 Ethernet

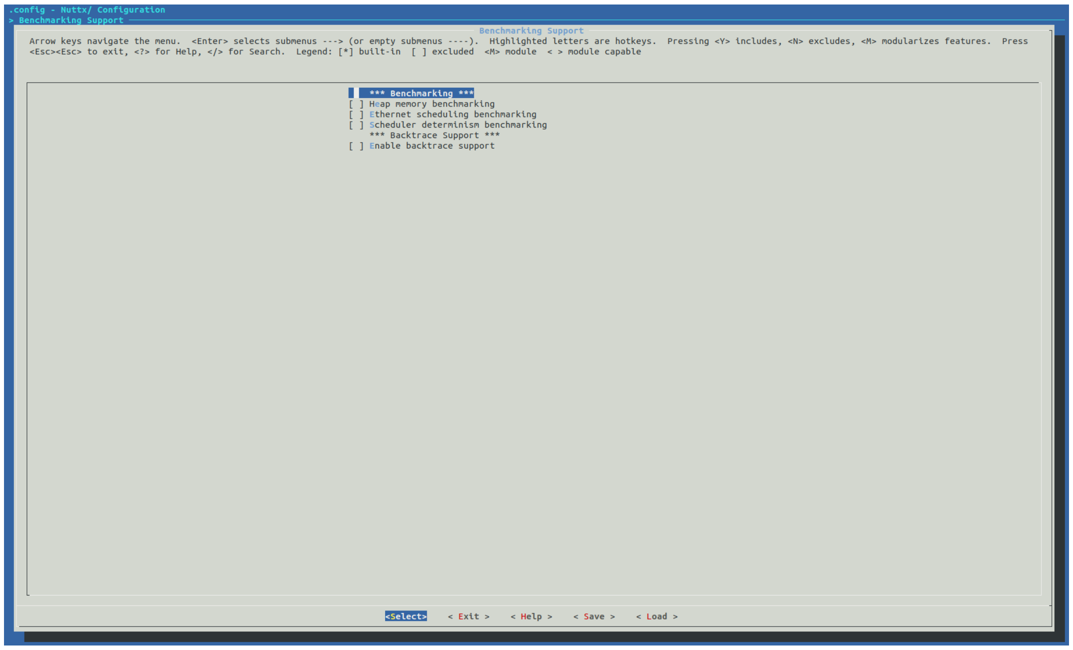
The reference board is hooked with an Ethernet 10/100 Mbps connection. To measure the board 100 Mbps Ethernet performance, theoretically, a timer with around 6.4us is required.

```
(84 Bytes) / ((100 / 8) * 1024 * 1024) ~= 6,4
```

Where 84 Bytes size corresponds to the smallest packet sent on the wire.

```
Preamble + SOF +  64 Ethernet packet + Interpacket Gap
```

A one micro second granularity can be achieved using the timer within the reference platform. Benchmarking functionality offers an abstraction in NuttX to make it easier to port it and to use it on every device.
Furthermore the benchmarking functionality is included in the Kconfig menu configuration to enable seamless integration.



```
.config - Nuttx/ Configuration
> Benchmarking Support
                                     Benchmarking Support
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                                        *** Benchmarking ***
                                    [ ] Heap memory benchmarking
                                    [ ] Ethernet scheduling benchmarking
                                    [ ] Scheduler determinism benchmarking
                                        *** Backtrace Support ***
                                    [ ] Enable backtrace support

                         <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 2: NuttX benchmarking options

2.2.2 Serial

The serial communication on the reference board is set to use 115200 bauds per seconds. The serial configuration is:

- 8 bits payload
- Not bit parity check,
- 1 stop bit.

Let's assume that the amount of data to send for an integer around 16bytes, the granularity needed is around one millisecond.

```
(16 / (115200 / 8)) ~= 1.1
```

To measure such a variation, a timer of at least 1 ms resolution would be enough. Every timer on the reference platform can achieve such a resolution.

2.2.3 General measurement at low-level communication medium

The measurement method for Serial and Ethernet are similar. The idea would be to start a zero-initialized timer before sending a packet at a low-level. Then measure the started timer once the function returns from sending. In this way, it would be possible to measure the time spent to send a packet.

2.2.4 User application general measurement

The general user measurement will be performed at a low-level in a user application. Thus measurement at a high level can be done. And communication comparison (in key functions) can be achieved between Fast-RTPS and Micro XRCE-DDS:

Key functions that are going to be analyzed:

- measurement of the micro ros initialization
- measurement of message send

2.2.5 Bandwidth analysis measurement

The bandwidth analysis measurement will be performed using the Wireshark [1] open-source software to capture packets. The packet capture will start from the moment a micro-ROS publisher establishes the first connection with the micro-ROS agent. The capture will stop when the then publisher reports the end of communication. Finally, the measurement of the total bandwidth will be achieved with the pcap [2] tool suite to retrieve the throughput of the communication between the node and the agent.

2.3 Real-Time assessment

Real time assessment benchmarking relates to 2 key points:

- determinism of a task scheduler.
- latency of a task scheduler.

The real-time assessment will be performed using the Real-Time Executer (RTE) to ensure, as indicated by its name, its "real-timeness" when executing an callback.

2.3.1 Determinism

The time determinism of execution is the foundation of all RTOS for time critical application. This means that at all time it is be possible to know what is the state of the running application (operating system + task + drivers). To achieve such a measurement, the application being benchmarked will just provide information on task scheduling. The information about which task is being executed, how long does the task has access to the scheduler and how often is scheduled the tasks.

In order to have to keep measurement consistent and relevant for benchmarking, the measurements sampled would be stored in a place in the RAM for a certain period, and dump the data once the benchmarking is done or the buffer full. This will eliminate the time overhead that transferring data would add.

2.3.2 Latency

The latency benchmarking will focus on determining the scheduler's latency variance (min/max/average). For instance, the latency for a task to access the CPU. Thus the latency will be a set of calculation based on the results that were gotten while measuring determinism of an application.

2.4 Resource usage

Resource usage benchmarking will be based on work described in D5.2: Reference values for HW + OS and shows results of platform baseline benchmarking based on defined use cases. Now, additional information about memory usage will be provided to understand where are the most used. The tool described in D5.2 was improved to be able to use C++ program for benchmarking.

2.4.1 Heap allocation resources

For heap allocation benchmarking a new tool was created. This tool is tracking allocation on a specific stack belonging to a specific task.

In order to analyze the dynamic allocations, the software will parse all allocation and search for a initial stack allocation. The initial stack allocation is found when parsing the function call that create the task to which the stack will be assigned to. Once the initial stack allocation found, the software calculate the upper and lower addresses the stack is set to. Then parsing continue. All allocations done within the range [Lower Stack, Upper Stack] will be counted and function that are calling for allocation will be reported.

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.

The platform is assessed in terms of security-sensitive applications. Security features developed in tasks:

- 3.1 (micro-RTPS Additional Features)
- 4.1 (micro-ROS client library) of an application.

will be evaluated.

2.6 Communication resilience

Network tools are used to introduce configurable disturbances in the communication such as packet loss, packet delay and limited or varying throughput. This build on previous work by Łukasiewicz-PIAP, which is open sourced in https://github.com/piappl/ros2_benchmarking [3].

Network will be disrupted in a controlled manner for testing through the use of software network emulation tools such as Linux Netem [4]. Supplementary evaluation for non-emulated disturbances will be conducted as well.

3 Tools used

3.1 Hardware tools

During benchmarking a specialized board was designed by Łukasiewicz-PIAP called MCB (Multi Connectors Board) and used.

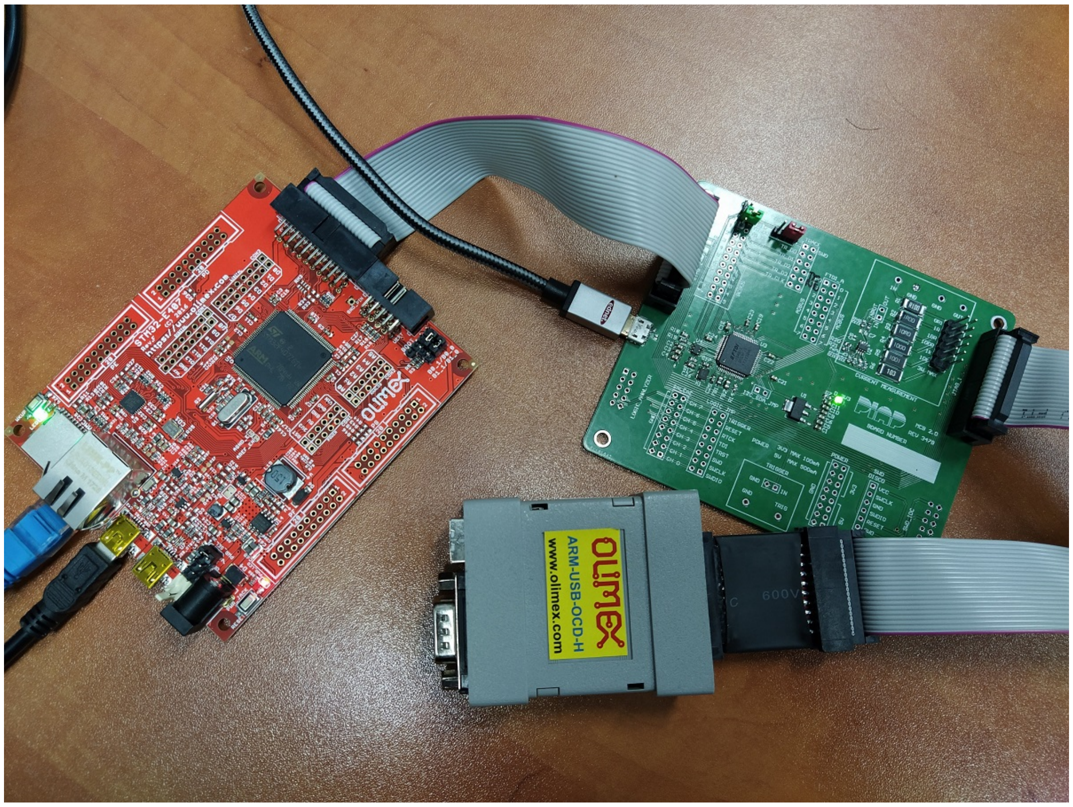


Figure 3: MCB and Olimex JTAG set

In the current work, the board was developed to provide data from the ARM debug facilities:

- SWO trace line fast reading
- JTAG interconnection

A ST-LINK debugger, but it can be any JTAG-SWD adapter supported by OpenOCD.

The basic platform on which we run and test micro-ROS is Olimex STM32-E407.

3.2 Software tools

To do all benchmarks tools described in D5.5 are used:

- pea (performance execution analysis)

- mfa (memory footprint analysis)

Tools are available on:

- https://github.com/microROS/benchmarking [5].

Comparing to previous benchmarks done deliverable software tools are more advanced. It is extended version based on Shadow Builder and Trace Framework Abstraction.

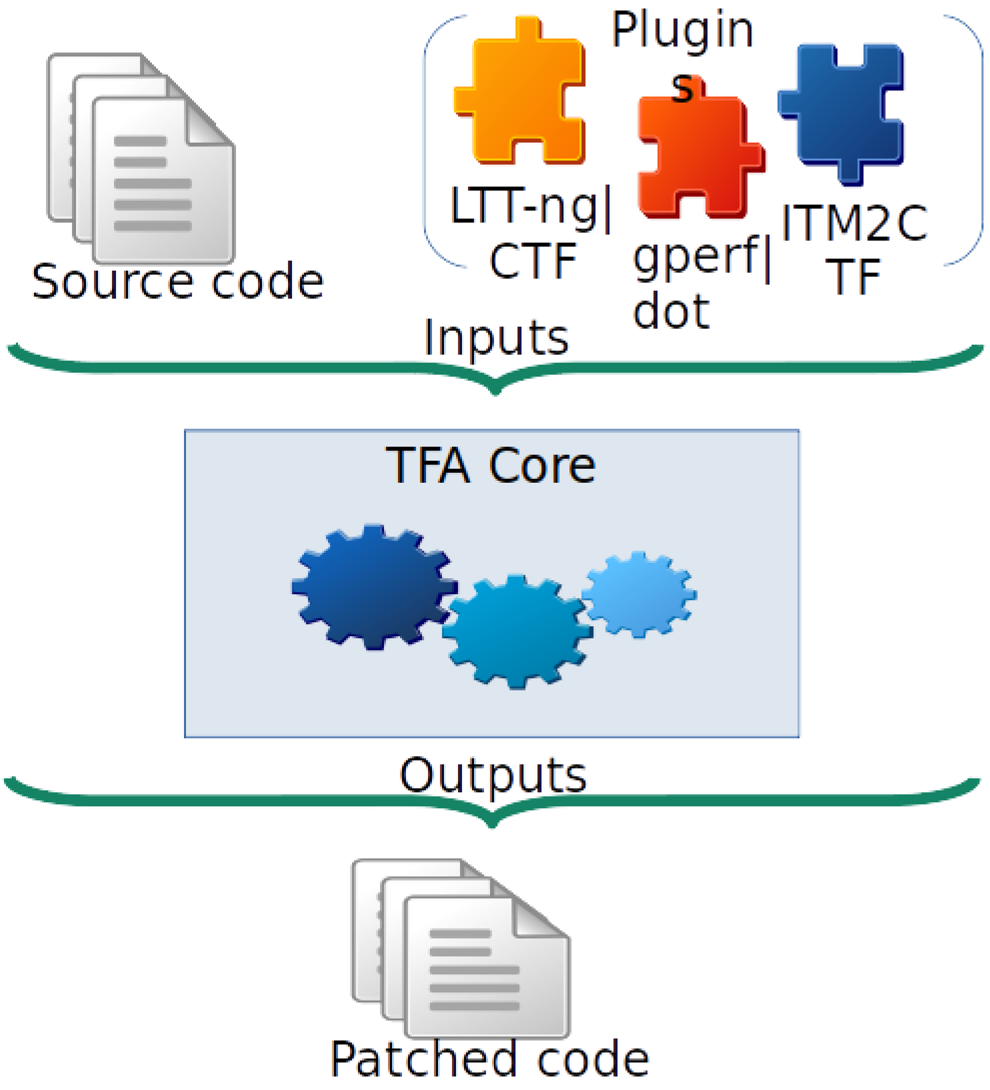


Figure 4: TFA and shadow building concept

4 Benchmarks setup

This section is described in the Deliverable D5.5[6] section Tool Setup.

5 Results

Results are the outcome of the applied methods discussed in the methodologies section.

This chapter gather results gotten from benchmarking according to the deliverable i.e. :

1. Resource usage: memory usage and execution performances.
2. Real-time assessment: being able to determine how deterministic is the micro-ROS stack.
3. Communication performance: measurement throughput.
4. Communication resilience: to see how behaves the application running given the corruption that happened.

Full results are available on Github: https://github.com/micro-ROS/benchmarking-results/tree/master/dec2019/ [7]

5.1 Current development status

The development was mostly focusing on abstraction tool allowing the end user to benchmark its application. Additional some visualisation tools and helper tools were developed to provide specific results.

5.2 Benchmark results for Olimex board

All the following benchmarking results were produced using the Olimex reference board. The software used for the benchmarking are:

- NuttX Realtime OS https://github.com/micro-ROS/NuttX .
- Publisher application from NuttX's applications set https://github.com/micro-ROS/apps/tree/dashing .
- micro-ROS middleware based on latest dashing version.

The agent will be running on the Docker connected to UART/Ethernet.

The memory analysis will be focused mostly on the HEAP. Only dynamic allocation will be analysed. As explained in the deliverable 5.2 [8], the sampling frequency will be around **~0,65625 MHz**.

5.2.1 Using Micro-XRCE-DDS over Serial Memory analysis

The JSON output file:

- [7]/bm_publisher_uart_mem/resultmemory.json

The number of allocations of blocks is reasonable, there is no excessive allocations of the same block from the same functions. So this section is going to focus on the size of allocation that are beyond 1024 bytes. It corresponds to a little less 1/100 of the total amount of the reference board's SRAM (196 KBytes):

Line	Function	File	Count	Size (bytes)
1	task_spawn	/uros_ws/firmware/NuttX/ sched/task/task_spawn.c	1	65016
2	emutls_alloc	/build/gcc-arm-none-eabi.../ ../src/libgcc/emutls.c	1	1048
3	exception_common	/uros_ws/firmware/ NuttX/...v7-m/gnu/up_exception.S	1	2072
4	work_lpstart	/uros_ws/firmware/NuttX/ ...wqueue/kwork_lpthread.c	1	2072

Observations (per line):

1. The stack_spawn is creating the stack needed by the application that is being ran (i.e. publisher). The stack holds all the initialised and un-initialised data that are used by the application. The chunk of memory is splitted between 65000 bytes + 16 bytes. The 65000 bytes are used for the application itself. This value is manually set by the developer. The value 16 bytes corresponds to "metadata" that are embedded in each chunk of memory.
2. The emutls_alloc allocate chunk of memory that is needed for the emutls object. This object will be used in the later TLS communication.
3. A stack is allocated to start a thread for the gnu exception. This is the first stack allocated to run the whole RTOS. The size corresponds to the default stack size.
4. The default kernel worker for the low priority queue. This low priority worker queue in charge of executing asynchronous functions in the kernel. Mostly to manage bottom/top halves.

Analysis:

Globally the the amount of allocation that are bigger than 1024 bytes is 68136. This corresponds to 35% of the total amount of memory available. Obviously, this amount does not takes into account the smaller chunks of memory that are allocated. But this give a good approximation.

5.2.2 Using Micro-XRCE-DDS over Ethernet Memory analysis

The JSON output file:

- [7]/bm_publisher_ethernet_mem/resultmemory.json

The results contain allocation that were made during the whole benchmarking session since platform boot-up. The publisher is sending 1000 messages to the micro-ROS-agent as fast as possible. The focus here in the table below is the amount of similar block that were dynamically allocated. Only allocation that are responsible by the middleware or above layer will be analysed.

Line	Function	File	Count	Size (bytes)
1	uxr_run_session_ until_confirm_de	/uros_ws/firmware/ mcu_ws.../core/session/session.c	1006	40

Observations (per line):

1. There are constants allocation of 40 bytes repeated more than 1000 times. According the the backtrace, the issue comes from the lower-level function. The culprit is the poll function. This event based file descriptor monitor allocate 40 bytes of memory on the heap after a call the function poll.

Analysis (per line):

1. Depending on the use case, poling the sockets using the poll posix is quite interesting as it offers good ressources usage. On the other hand, it could lead to fragmentation and over time it would reduce execution performances. Especially in the case when multiple messages are sent/received.

The focus in the next table are the number of blocks that are bigger than 1024 bytes. It corresponds to a little less 1/100 of the total amount of the reference board's RAM (196 KBytes):

Line	Function	File	Count	Size (bytes)
1	task_spawn	/uros_ws/firmware/NuttX/ sched/task/task_spawn.c	1	65016
2	nx_start_task	/uros_ws/firmware/NuttX/ sched/init/nx_bringup.c	1	2072
3	emutls_alloc	/build/gcc-arm-none-eabi.../ ../src/libgcc/emutls.c	1	1048
4	nsh_netinit	/uros_ws/firmware/apps/ nshlib/nsh_netinit.c	1	1592
5	nsh_netinit_thread	/uros_ws/firmware/ apps/n...s/ntpclient/ntpclient.c	1	2072
6	exception_common	/uros_ws/firmware/NuttX/ ...v7-m/gnu/up_exception.S	1	2072
7	work_lpstart	/uros_ws/firmware/NuttX/ ...wqueue/kwork_lpthread.c	1	2072
8	emutls_alloc	/build/gcc-arm-none-eabi.../ ../src/libgcc/emutls.c	1	1048

Observations and analysis (per line):

1. The stack_spawn is creating the stack needed by the application that is being ran (i.e. publisher). The stack holds all the initialised and un-initialised data that are used by the application. The chunk of memory is splitted between 65000 bytes + 16 bytes. The 65000 bytes are used for the application itself. This value is manually set by the developer. The value 16 bytes corresponds to "metadata" that are embedded in each chunk of memory.
2. nx_start_task is a temporary "proxy" stack that is needed to initialise the task that is created.
3. The emutls_alloc allocate chumk of memory that is needed for the emutls object. This object will be used in the later TLS communication.

4. ns_netinit is allocating a chunk of memory that is the size of Window's MTU.
5. As the network NTP(Network Protocol Time) is running as part of the stack, NuttX allocated a stack to start a thread runing the NTP service. The value is the default value of the allocation of thread.
6. A stack is allocated to start a thread for the gnu exception. This is the first stack allocated to run the whole RTOS. The size corresponds to the default stack size.
7. The default kernel worker for the low priority queue. This low priority queue worker in charge of executing asynchronous functions in the kernel. Mostly to manage bottom/top halves.
8. Another allocation of the emults, same observations and analysis fit for this table entry.

In total, from this table around 75400 bytes were dynamically allocated. This is around the 38% of the total amount of the reference platform. However, total amount of memory calculated is not taking into account smaller chunk of memory allocated. Meaning that the amount of memory allocated is beyond 75400 bytes.

5.2.3 Using Micro-XRCE-DDS over Serial Performance analysis

```
file: /uros_ws/firmware/NuttX/arch/arm/src/armv7-m/gnu/up_exception.S
function: exception_common
hits: 19
details: [
    { hits: 2, line: 176, address: 8000724},
    { hits: 2, line: 180, address: 8000728},
    { hits: 1, line: 222, address: 800073e},
    { hits: 1, line: 238, address: 8000740},
    { hits: 2, line: 239, address: 8000744},
    { hits: 4, line: 241, address: 8000748},
    { hits: 1, line: 249, address: 800075a},
    { hits: 1, line: 250, address: 800075e},
    { hits: 1, line: 252, address: 8000762},
    { hits: 1, line: 262, address: 8000768},
    { hits: 1, line: 301, address: 800077e},
    { hits: 2, line: 308, address: 8000782},

]
file: /uros_ws/firmware/NuttX/arch/arm/src/common/up_mdelay.c
function: up_mdelay
hits: 325
details: [
    { hits: 57, line: 66, address: 8000f0e},
    { hits: 33, line: 66, address: 8000f10},
    { hits: 56, line: 66, address: 8000f12},
    { hits: 62, line: 66, address: 8000f1c},
    { hits: 29, line: 66, address: 8000f1e},
    { hits: 24, line: 66, address: 8000f20},
    { hits: 64, line: 66, address: 8000f22},
]
file: /uros_ws/firmware/NuttX/sched/sched/sched_unlock.c
function: sched_unlock
hits: 1
details: [
    { hits: 1, line: 250, address: 8001828},

]
file: /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_eth.c
function: stm32_phyread
hits: 615
details: [
    { hits: 1, line: 2995, address: 80055d6},
    { hits: 1, line: 2997, address: 80055da},
    { hits: 1, line: 3001, address: 80055de},
    { hits: 66, line: 3001, address: 80055e0},
    { hits: 31, line: 3001, address: 80055e2},
    { hits: 56, line: 3001, address: 80055e4},
    { hits: 166, line: 3003, address: 80055ec},
    { hits: 26, line: 3003, address: 80055ee},
    { hits: 65, line: 3003, address: 80055f2},
    { hits: 1, line: 3005, address: 80055f4},
    { hits: 2, line: 3005, address: 80055f6},
    { hits: 70, line: 3001, address: 80055fe},
    { hits: 29, line: 3001, address: 8005600},
    { hits: 32, line: 3001, address: 8005602},
    { hits: 68, line: 3001, address: 8005604},

]
file: /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_eth.c
function: stm32_phyinit
hits: 1
details: [
    { hits: 1, line: 3218, address: 80059c0},

]
file: /uros_ws/firmware/NuttX/sched/semaphore/sem_tickwait.c
function: nxsem_tickwait
hits: 8
details: [
    { hits: 2, line: 88, address: 802939c},
    { hits: 1, line: 102, address: 80293aa},
    { hits: 1, line: 142, address: 80293ce},
    { hits: 1, line: 152, address: 80293e2},
    { hits: 1, line: 157, address: 80293ea},
    { hits: 1, line: 158, address: 80293ee},
    { hits: 1, line: 180, address: 802940c},

]
file: /uros_ws/firmware/NuttX/drivers/serial/serial.c
function: uart_write
hits: 1
details: [
    { hits: 1, line: 1067, address: 802a408},

]
file: /uros_ws/firmware/NuttX/drivers/serial/serial_io.c
function: uart_xmitchars
hits: 7
details: [
    { hits: 1, line: 70, address: 802a770},
    { hits: 2, line: 70, address: 802a778},
    { hits: 1, line: 70, address: 802a7a4},
    { hits: 1, line: 74, address: 802a7aa},
    { hits: 1, line: 74, address: 802a7b6},
    { hits: 1, line: 81, address: 802a7ce},

]
file: /uros_ws/firmware/NuttX/libs/libc/stdio/lib_libfwrite.c
function: lib_fwrite
hits: 1
details: [
    { hits: 1, line: 110, address: 802d6fe},

]
file: /uros_ws/firmware/NuttX/drivers/usbdev/cdcacm.c
function: cdcacm_fillrequest
hits: 4
details: [
    { hits: 1, line: 333, address: 80309cc},
    { hits: 2, line: 333, address: 80309d2},
    { hits: 1, line: 333, address: 80309d4},

]
```

Function	Hits	Percentage of the total time
exception_common	19	~1%
up_mdelay	325	~33%
sched_unlock	1	~0.1%
stm32_phyread	615	62%
stm32_phyinit	1	~0.1%
nxsem_tickwait	8	~0.8%

Function	Hits	Percentage of the total time
uart_write	1	~0.1%
uart_xmitchars	7	~0.7%
lib_fwrite	1	~0.1%
cdcacm_fillrequest	4	~0.4

During this executing, the up_mdelay function is using 33% of the cpu time. Waiting for packets to be transmited. Of course, even if the ethernet is not used, but plugged to the network, 62% of the time is used by the phyread. This should be avoid at all cost. The ethernet was necessary for the NTP client.

If the application does not include the network stack, then the execution could be faster. Therefore there would be more room for more processing.

5.2.4 Using Micro-XRCE-DDS over Ethernet Performance analysis

The output of the performance benchmarking tools is shown below:

```
file: /uros_ws/firmware/NuttX/sched/init/nx_start.c
function: nx_start
hits: 618
details: [
    { hits: 311, line: 876, address: 8001120},
    { hits: 307, line: 876, address: 8001124},

]
file: /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_idle.c
function: up_idle
hits: 311
details: [
    { hits: 311, line: 433, address: 80039fc},

]
file: /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_eth.c
function: stm32_phyread
hits: 618
details: [
    { hits: 2, line: 2980, address: 80056a8},
    { hits: 1, line: 2986, address: 80056aa},
    { hits: 1, line: 2987, address: 80056b4},
    { hits: 1, line: 2993, address: 80056c0},
    { hits: 1, line: 2997, address: 80056c6},
    { hits: 1, line: 3001, address: 80056ca},
    { hits: 58, line: 3001, address: 80056cc},
    { hits: 34, line: 3001, address: 80056ce},
    { hits: 68, line: 3001, address: 80056d0},
    { hits: 163, line: 3003, address: 80056d8},
    { hits: 31, line: 3003, address: 80056da},
    { hits: 68, line: 3003, address: 80056de},
    { hits: 1, line: 3005, address: 80056e2},
    { hits: 1, line: 3005, address: 80056e4},
    { hits: 1, line: 3014, address: 80056e8},
    { hits: 57, line: 3001, address: 80056ea},
    { hits: 33, line: 3001, address: 80056ec},
    { hits: 32, line: 3001, address: 80056ee},
    { hits: 64, line: 3001, address: 80056f0},

]
```

Observations and analysis:

Below the code's snippets that are refered by the tool:

- /uros_ws/firmware/NuttX/sched/init/nx_start.c:876:

```
842   /* The IDLE Loop **********************************************************/
843   /* When control is return to this point, the system is idle. */
844
845   sinfo("CPU0: Beginning Idle Loop\n");
846   for (; ; )
847     {
848       /* Perform garbage collection (if it is not being done by the worker
849        * thread).  This cleans-up memory de-allocations that were queued
850        * because they could not be freed in that execution context (for
851        * example, if the memory was freed from an interrupt handler).
852        */
853
854 #ifndef CONFIG_SCHED_WORKQUEUE
855       /* We must have exclusive access to the memory manager to do this
856        * BUT the idle task cannot wait on a semaphore.  So we only do
857        * the cleanup now if we can get the semaphore -- this should be
858        * possible because if the IDLE thread is running, no other task is!
859        *
860        * WARNING: This logic could have undesirable side-effects if priority
861        * inheritance is enabled.  Imaginee the possible issues if the
862        * priority of the IDLE thread were to get boosted!  Moral: If you
863        * use priority inheritance, then you should also enable the work
864        * queue so that is done in a safer context.
865        */
866
867       if (sched_have_garbage() && kmm_trysemaphore() == 0)
868         {
```

```
869           sched_garbage_collection();
870           kmm_givesemaphore();
871         }
872 #endif
873 
874       /* Perform any processor-specific idle state operations */
875 
876       up_idle();
877     }
```

- /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_idle.c +433

```
432 void up_idle(void)
433 {
434 #if defined(CONFIG_SUPPRESS_INTERRUPTS) || defined(CONFIG_SUPPRESS_TIMER_INTS)
435   /* If the system is idle and there are no timer interrupts, then process
436    * "fake" timer interrupts. Hopefully, something will wake up.
437    */
438 
439   nxsched_process_timer();
440 #else
441 
442   /* Perform IDLE mode power management */
443 
444   BEGIN_IDLE();
445   stm32_idlepm();
446   END_IDLE();
447 #endif
448 }
```

- /uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_eth.c:3005

```
2979 static int stm32_phyread(uint16_t phydevaddr, uint16_t phyregaddr, uint16_t *value)
2980 {
2981   volatile uint32_t timeout;
2982   uint32_t regval;
2983 
2984   /* Configure the MACMIIAR register, preserving CSR Clock Range CR[2:0] bits */
2985 
2986   regval  = stm32_getreg(STM32_ETH_MACMIIAR);
2987   regval &= ETH_MACMIIAR_CR_MASK;
2988 
2989   /* Set the PHY device address, PHY register address, and set the buy bit.
2990    * the  ETH_MACMIIAR_MW is clear, indicating a read operation.
2991    */
2992 
```

```
  regval |= (((uint32_t)phydevaddr << ETH_MACMIIAR_PA_SHIFT) & ETH_MACMIIAR_PA_MASK);
  regval |= (((uint32_t)phyregaddr << ETH_MACMIIAR_MR_SHIFT) & ETH_MACMIIAR_MR_MASK);
  regval |= ETH_MACMIIAR_MB;

  stm32_putreg(regval, STM32_ETH_MACMIIAR);

  /* Wait for the transfer to complete */

  for (timeout = 0; timeout < PHY_READ_TIMEOUT; timeout++)
    {
      if ((stm32_getreg(STM32_ETH_MACMIIAR) & ETH_MACMIIAR_MB) == 0)
        {
          *value = (uint16_t)stm32_getreg(STM32_ETH_MACMIIDR);
          return OK;
        }
    }

  nerr("ERROR: MII transfer timed out: phydevaddr: %04x phyregaddr: %04x\n",
       phydevaddr, phyregaddr);

  return -ETIMEDOUT;
}
```

As shown in the results, the most time is spent in the up_idle:

`/uros_ws/firmware/NuttX/arch/arm/src/chip/stm32_idle.c`

and in the idle loop:

`/uros_ws/firmware/NuttX/sched/init/nx_start.c`

So concretly, it means that the number of hits in up_idle and idle_task can be added together as they are both part of the idle loop.

Function	Hits	Percentage of the total time
up_ilde + nx_start(idle_loop)	929	~60%
phy_read	618	~40%

The application spent most of its time in idle or waiting for transfers complete.

5.2.5 Using Micro-XRCE-DDS over Ethernet Network analysis

The network analysis will be separeted into 3 different criterions:

- Latency: native, 1000ms, 5000ms
- Packet lost: 10%, 60%,
- Packet error: 1%, 10%.

Native performance

The pcap output file:

- [7]/bm_publisher_ethernet_net/publisher_native.pcapng

Output data form pcapinfo tool:

```
File name:            publisher_native.pcapng
File type:            Wireshark/... - pcapng
File encapsulation:   Ethernet
File timestamp precision:  nanoseconds (9)
Packet size limit:    file hdr: (not set)
Number of packets:    2 064
File size:            188 kB
Data size:            121 kB
Capture duration:     4,160115193 seconds
First packet time:    2019-12-19 11:47:56,705651820
Last packet time:     2019-12-19 11:48:00,865767013
Data byte rate:       29 kBps
Data bit rate:        233 kbps
Average packet size: 58,81 bytes
Average packet rate: 496 packets/s
SHA256:               5d114e2f4261d511dcff9d12da0e7024cd02e51f1130fb0131e66dc48a235bde
RIPEMD160:            cb6f483c5b6258f4328c3a3e5f263015d4c0aa07
SHA1:                 30cfb4cd189a9ed88462414dc4195b336c86f498
Strict time order:    True
Capture hardware:     Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz (with SSE4.2)
Capture oper-sys:     Linux 5.3.15-050315-generic
Capture application: Dumpcap (Wireshark) 2.6.10 (Git v2.6.10
 packaged as 2.6.10-1~ubuntu18.04.0)
Number of interfaces in file: 1
Interface #0 info:
                      Name = enp2s0
                      Encapsulation = Ethernet (1 - ether)
                      Capture length = 262144
                      Time precision = nanoseconds (9)
                      Time ticks per second = 1000000000
                      Time resolution = 0x09
                      Operating system = Linux 5.3.15-050315-generic
                      Number of stat entries = 1
                      Number of packets = 2064
```

1000ms latency

The pcap output file:

- [7]/bm_publisher_ethernet_net/publisher_delay_1000ms.pcapng

Due to the very slow transmition rate, the number of packet sent was reduce to 100. Output data form pcapinfo tool:

```
File name:           publisher_delay_1000ms.pcapng
File type:           Wireshark/... - pcapng
File encapsulation:  Ethernet
File timestamp precision:  nanoseconds (9)
Packet size limit:   file hdr: (not set)
Number of packets:   4 706
File size:           431 kB
Data size:           278 kB
Capture duration:    54,773729984 seconds
First packet time:   2019-12-19 12:09:56,481678379
Last packet time:    2019-12-19 12:10:51,255408363
Data byte rate:      5 082 bytes/s
Data bit rate:       40 kbps
Average packet size: 59,16 bytes
Average packet rate: 85 packets/s
SHA256:              b0b9669c793486b4d799944e591bdf72d69e2f1abfa5a69302b8df4cf030b746
RIPEMD160:           6bd22de309545bf726c2e1b724bb887029e8afb7
SHA1:                e7897f8d8548fe031219b31a1fa47c4b3b5e8ceb
Strict time order:   True
Capture hardware:    Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz (with SSE4.2)
Capture oper-sys:    Linux 5.3.15-050315-generic
Capture application: Dumpcap (Wireshark) 2.6.10
(Git v2.6.10 packaged as 2.6.10-1~ubuntu18.04.0)
Number of interfaces in file: 1
Interface #0 info:
                     Name = enp2s0
                     Encapsulation = Ethernet (1 - ether)
                     Capture length = 262144
                     Time precision = nanoseconds (9)
                     Time ticks per second = 1000000000
                     Time resolution = 0x09
                     Operating system = Linux 5.3.15-050315-generic
                     Number of stat entries = 1
                     Number of packets = 4706
```

5000 ms latency

The pcap output file:

- [7]/bm_publisher_ethernet_net/publisher_delay_5000ms.pcapng

Using this latency the micro-ROS did not connect to the software application. The output error is as display below:

```
UDP mode => ip: 192.168.8.1 - port: 8888

>>> [rcutils|error_handling.c:106] rcutils_set_error_state()
This error state is being overwritten:

  'failed to create node session on Micro ROS Agent.,
  at /uros_ws/firmware/mcu_ws/uros/rmw_microxrcedds/
  rmw_microxrcedds_c/src/rmw_node.c:216, at /uros_ws/firmware/
  mcu_ws/ros2/rcl/rcl/src/rcl/node.c:326'

with this new error message:

  'rcl node's rmw handle is invalid, at /uros_ws/firmware/mcu_ws/ros2/rcl/rcl/
  src/rcl/node.c:464'

rcutils_reset_error() should be called after error handling to avoid this.
<<<
[ERROR] [rcl]: Failed to fini publisher for node: 1
Node initialization error: rcl node's rmw handle is invalid, at /uros_ws/
firmware/mcu_ws/ros2/rcl/rcl/src/rcl/node.c:464
```

10% lost packets

The pcap output file:

- [7]/bm_publisher_ethernet_net/publisher_10_pourcent_lost.pcapng

```
File name:             publisher_10_pourcent_lost.pcapng
File type:             Wireshark/... - pcapng
File encapsulation:    Ethernet
File timestamp precision:  nanoseconds (9)
Packet size limit:     file hdr: (not set)
Number of packets:     2 181
File size:             199 kB
Data size:             128 kB
Capture duration:      6,332854869 seconds
First packet time:     2019-12-19 12:29:02,852914242
Last packet time:      2019-12-19 12:29:09,185769111
Data byte rate:        20 kBps
Data bit rate:         162 kbps
Average packet size: 59,00 bytes
Average packet rate: 344 packets/s
SHA256:              78a8892351d137c559de4b08425d824815d0da52a24d94a9fc337860b68e2c10
RIPEMD160:           9eee93cdc43e69acdaf5d8bc474535fc8976f025
SHA1:                0a69d0a2cfdb07b9bcb73bf87b75e65bf4364b5c
Strict time order:   True
Capture hardware:    Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz (with SSE4.2)
Capture oper-sys:    Linux 5.3.15-050315-generic
Capture application: Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged
as 2.6.10-1~ubuntu18.04.0)
Number of interfaces in file: 1
Interface #0 info:
                     Name = enp2s0
                     Encapsulation = Ethernet (1 - ether)
                     Capture length = 262144
                     Time precision = nanoseconds (9)
                     Time ticks per second = 1000000000
                     Time resolution = 0x09
                     Operating system = Linux 5.3.15-050315-generic
                     Number of stat entries = 1
                     Number of packets = 2181
```

60% lost packets

The pcap output file:

- [7]/bm_publisher_ethernet_net/publisher_10_pourcent_lost.pcapng

Unfortunately, the connection and registration in the multi-cast group did not happen. Moreover the software generated an error:

```
>>> [rcutils|error_handling.c:106] rcutils_set_error_state()
This error state is being overwritten:

  'Issues creating micro XRCE-DDS entities, at /uros_ws/firmware/mcu_ws/
  uros/rmw_microxrcedds/rmw_microxrcedds_c/src/rmw_publisher.c:190'

with this new error message:

  'unable to remove publisher from the server, at /uros_ws/firmware/mcu_ws/
  uros/rmw_microxrcedds/rmw_microxrcedds_c/src/rmw_microxrcedds_topic.c:155'

rcutils_reset_error() shouPublisher initialization error: unable to remove
publisher from the server, at /uros_ws/firmware/mcu_ws/uros/rmw_microxrcedds/
rmw_microxrcedds_c/src/rmw_microxrcedds_topic.c:155, at /uros_ws/firmware/
mcu_ws/ros2/rcl/rcl/src/rcl/publisher.c:171
ld be called after error handling to avoid this.
<<<
```

1% lost packets

The pcap output file:

- [7]/bm_publisher_ethernet_net/publisher_1_pourcent_lost.pcapng

Around 658 packets were send according to the publisher. Moreover the software generated an error:

```
Sent: '658'
TOTAL sent: 1000

>>> [rcutils|error_handling.c:106] rcutils_set_error_state()
This error state is being overwritten:

  'error publishing message, at /uros_ws/firmware/mcu_ws/uros/rmw_microxrcedds/
  rmw_microxrcedds_c/src/rmw_publish.c:60, at /uros_ws/firmware/mcu_ws/ros2/
  rcl/rcl/src/rcl/publisher.c:257'

with this new error message:

  'unable to remove publisher from the server, at /uros_ws/firmware/mcu_ws/
  uros/rmw_microxrcedds/rmw_microxrcedds_c/src/rmw_publisher.c:275'

rcutils_reset_error() should be called after error handling to avoid this.
<<<

>>> [rcutils|error_handling.c:106] rcutils_set_error_state()
This error state is being overwritten:

  'unable to remove publisher from the server, at /uros_ws/firmware/
  mcu_ws/uros/rmw_microxrcedds/rmw_microxrcedds_c/src/rmw_publisher.c:275,
   at /uros_ws/firmware/mcu_ws/ros2/rcl/rcl/src/rcl/publisher.c:226'

with this new error message:

  'Unable to fini publisher for node., at /uros_ws/firmware/mcu_ws/
  ros2/rcl/rcl/src/rcl/node.c:429'

rcutils_reset_error() should be called after error handling to avoid this.
<<<
```

10% of the packets are corrupted

The pcap output file:

- [7]/bm_publisher_ethernet_net/publisher_10_pourcent_corrupted.pcapng

The first 3 packets could manage the connection and an error was shown on the micro-ROS client:

```
Sent: '0'
Sent: '1'
Sent: '2'
TOTAL sent: 1000

>>> [rcutils|error_handling.c:106] rcutils_set_error_state()
This error state is being overwritten:

  'error publishing message, at /uros_ws/firmware/mcu_ws/uros/
  rmw_microxrcedds/rmw_microxrcedds_c/src/rmw_publish.c:60,
  at /uros_ws/firmware/mcu_ws/ros2/rcl/rcl/src/rcl/publisher.c:257'

with this new error message:

  'unable to remove publisher from the server, at /uros_ws/
  firmware/mcu_ws/uros/rmw_microxrcedds/rmw_microxrcedds_c/src/rmw_publisher.c:275'

rcutils_reset_error() should be called after error handling to avoid this.
<<<

>>> [rcutils|error_handling.c:106] rcutils_set_error_state()
This error state is being overwritten:

  'unable to remove publisher from the server, at /uros_ws/
  firmware/mcu_ws/uros/rmw_microxrcedds/rmw_microxrcedds_c/
  src/rmw_publisher.c:275, at /uros_ws/firmware/mcu_ws/
  ros2/rcl/rcl/src/rcl/publisher.c:226'

with this new error message:

  'Unable to fini publisher for node., at /uros_ws/firmware/
  mcu_ws/ros2/rcl/rcl/src/rcl/node.c:429'

rcutils_reset_error() should be called after error handling to avoid this.
<<<
```

Observations and analysis:

According to the benchmarking done, the micro-ROS communication over Ethernet shows a good immunity regarding packet losses and high latency below 5s. However regarding packets corruption, even at low percentage (1%), the micro-ROS does not behave well. Additionally, in most cases where the conditions were very bad, the micro-ROS did not show any failsafe mechanism (even restarting the application was not working). To make it work as expected it was needed to restart both the agent and the publisher (hard reset).

5.2.6 UDP and micro-ROS

UDP Raw one message results. The pcap output file:

- [7]/bm_publisher_ethernet_perf_vs_raw_udp/udp_raw_1_message.pcapng

```
File name:           ./udp_raw_1_message.pcapng
File type:           Wireshark/... - pcapng
File encapsulation:  Ethernet
File timestamp precision:  nanoseconds (9)
Packet size limit:   file hdr: (not set)
Number of packets:   1
File size:           508 bytes
Data size:           86 bytes
Capture duration:    0,000000000 seconds
First packet time:   2019-12-24 10:28:31,861424477
Last packet time:    2019-12-24 10:28:31,861424477
Data byte rate:      0 bytes/s
Data bit rate:       0 bits/s
Average packet size: 86,00 bytes
Average packet rate: 0 packets/s
SHA256:
a090d54573ec982dbd0e18589fdf94fdc84e96b72bf498dec47ba27d1fa7cf32
RIPEMD160:           4634c563effba8a7e7a86b0e2f233ecccee41d9a
SHA1:                10714b6616dc68166a3df8b1c06e7acbcfc6ecb5
Strict time order:   True
Capture hardware:    Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz (with SSE4.2)
Capture oper-sys:    Linux 5.3.15-050315-generic
Capture application: Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as
2.6.10-1~ubuntu18.04.0)
Number of interfaces in file: 1
Interface #0 info:
                     Name = enp2s0
                     Encapsulation = Ethernet (1 - ether)
                     Capture length = 262144
                     Time precision = nanoseconds (9)
                     Time ticks per second = 1000000000
                     Time resolution = 0x09
                     Operating system = Linux 5.3.15-050315-generic
                     Number of stat entries = 1
                     Number of packets = 1
```

Micro-ROS publisher one message results. The pcap output file:

- [7]/bm_publisher_ethernet_perf_vs_raw_udp/publisher_1_message.pcapng

```
File name:           ./publisher_1_message.pcapng
File type:           Wireshark/... - pcapng
File encapsulation:  Ethernet
File timestamp precision:  nanoseconds (9)
Packet size limit:   file hdr: (not set)
Number of packets:   67
File size:           9 164 bytes
Data size:           6 577 bytes
Capture duration:    4,333756831 seconds
First packet time:   2019-12-23 12:00:11,850366164
Last packet time:    2019-12-23 12:00:16,184122995
Data byte rate:      1 517 bytes/s
Data bit rate:       12 kbps
Average packet size: 98,16 bytes
Average packet rate: 15 packets/s
SHA256:
3a9d2d80894a9dee05de0c51c5bbb787d5a537dd00f27bb4f922f92e8c0fc6b3
RIPEMD160:           75be2350994262c40fa98622a6ad3952eca8cc75
SHA1:                d0de8222cb136af4e12f01a119413f88d29062da
Strict time order:   True
Capture hardware:    Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz (with SSE4.2)
Capture oper-sys:    Linux 5.3.15-050315-generic
Capture application: Dumpcap (Wireshark) 2.6.10 (Git v2.6.10 packaged as
2.6.10-1~ubuntu18.04.0)
Number of interfaces in file: 1
Interface #0 info:
                     Name = enp2s0
                     Encapsulation = Ethernet (1 - ether)
                     Capture length = 262144
                     Time precision = nanoseconds (9)
                     Time ticks per second = 1000000000
                     Time resolution = 0x09
                     Operating system = Linux 5.3.15-050315-generic
                     Number of stat entries = 1
                     Number of packets = 67
```

It was observed that some improvement on the memory footprint of the static is 1/5 smaller (from 500Kbytes to ~113KBytes).

Also the Ethernet results showed a big difference. Sending one message (integer) over udp, is is using around **86 Bytes** whereas sending one message using a publisher would use around **6577 Bytes**. There is a difference of almost 2 order of magnitude.

5.3 Benchmarking successes

During those benchmarkings undergone, some issues were raised. Indeed, especially when performing network benchmarking, the application showed some weakness regarding the application reliability. The consortium is already aware of those weakness and is already working on solving them.

Benchmarking, beyond providing metrics to evaluate performances, is also providing a set of tests that helps finding potential weaknesses.

5.3.1 Benchmarking comparisons

Benchmarkings showed results that are provide informations against previous iteration of itself. A huge improvement was observed regarding memory re-allocation, performances in general.

However, comparison need to be pushed a little further toward external potential competitor or equivalent solution. In this regards, a simple udp client was created to communicate with a PC host.

5.4 Future general work

In order to facilitate benchmarking and create one unified tool a lot of work still on-going on the aspect below:

- Creating a abstraction compiler to multi-platforms to be benchmarked,
- Creating a tool to analyze the Stack (bss and data sections).

5.4.1 Future works planned for the memory tool

Currently the memory tool only analyze the dynamic allocation. However still more work is needed to complete the whole memory analysis:

- Analyzing the Stack BSS and Data section to know what is the application currently doing.

5.4.2 Future works planned for communication performances

The tools is currently only measuring the time spent in read/write function at the lowest level (Ethernet driver in NuttX). In the future, the measurement of the communication performance shall be done at higher level in the user application and/or the middleware.

5.4.3 Future works planned for communication resilience

Communication resilience with the use of Netem [4] allow any Linux box to emulate some packets losses/error/latency. This imply that the micro-ROS is using the Ethernet communication medium. Currently there are not tool allowing to do such an emulation. Therefore the effort to add are:

- Create a software/driver that bridge the communication between two UART devices to alter the packet Quality of Service.

6 Conclusions

In conclusion, the results show that micro-ROS:

- Into certain kernel configurations (using network stack + network driver) the binary usage around (512KB of flash is needed) and at least 95KB of ram corresponding to 30 KB of static kernel variables + around 65KB regarding the publisher application stack.
- Still there are repeated memory allocations than can could lead to memory fragmentation.
- Communication mechanisms showed some weaknesses for packets corruption and high latency network, those are already being taken care of.
- As the micro-ROS stack is still failsafe functionalities which, in mission critical projects, would be place micro-ROS in the no-go list.
- Raw UDP vs micro-ROS Stack message showed a big difference regarding the amount of byte sent.

Generally, micro-ROS offers great performance as:

- In comparison to a ROS2 node, a micro-ROS offers basic ROS2 functionalities, with a smaller memory footprint.
- Micro-ROS might have a heavier wire usage over Ethernet. However, this overhead is due to the provided security by ciphering the communication (TLS) and added robustness.
- In general the execution performance are not bottlenecked by the application, but the limitation comes from the underlying hardware. Meaning that the software is very efficient.

References

[1] 'Wireshark website'. https://www.wireshark.org/.

[2] 'Pcap tool'. [Online]. Available: https://en.wikipedia.org/wiki/Pcap

[3] Łukasiewicz-PIAP, 'ROS2 benchmarking'. [Online]. Available: https://github.com/piappl/ros2_benchmarking

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

[5] A. Malki, 'Micro-ROS benchmarking tools'. [Online]. Available: https://github.com/micro-ROS/benchmarking

[6] M. M. Tomasz Kołcon Alexandre Malki, 'D5.5'. [Online]. Available: http://ofera.eu/storage/deliverables/M16/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf

[7] PIAP, 'D5.3 results'. [Online]. Available: https://github.com/micro-ROS/benchmarking-results/tree/master/dec2019

[8] M. M. Tomasz Kołcon Alexandre Malki, 'D5.2'. [Online]. Available: http://ofera.eu/storage/deliverables/M16/OFERA_53_D52_Micro-ROS_benchmarks_-_Initial.pdf