D5.6

Micro-ROS benchmarking and validation tools Release - initial

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D5.6
Deliverable name	Micro-ROS benchmarking and validation tools Release - initial
Date	December 2020
Dissemination level	public
Workpackage and task	5.6
Author	Alexandre Malki (Łukasiewicz-PIAP), Tomasz Kołcon (Łukasiewicz-PIAP)
Contributors	Agnieszka Sprońska (Łukasiewicz-PIAP), Mateusz Maciaś (Łukasiewicz-PIAP), Jan Staschulat (Bosch)
Keywords	Microcontroller, NuttX, Benchmarking
Abstract	This deliverable summarizes works in Task 5.3: Benchmark tooling for application developers.

Contents

1 Summary

This deliverable summarizes the works in Task 5.3: Benchmark tooling for application developers. It is a description of initial release of benchmarking tools that will be continued and described in D5.8. Previous developments described in D5.5 are continued here. The tools developed in the scope of task 5.3 are used internally as well as released as open source packages.

This document will first provide links to the released software and their respective documentation. In addition, A revised and general description of the created tool and it's architecture.

Term	Definition
ROS	Robot Operating System
MCB	Multi-connectors board
SWO	Single Wire Output
SWD	Serial Wire Debug
JTAG	Joint Test Action Group (also name of interface)
ITM	Instrumentation (or Instruction) Trace Macrocell
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
AST	Abstract Syntax Tree
TFA	Trace Framework Abstraction
SB	Shadow Builder
KPI	Key Performance Indicators

2 Overview

This document provides generic introduction to architecture and concepts. It also links all required source repositories.

Most of this content is already available for community at:

- https://micro-ros.github.io/docs/concepts/benchmarking [1],
- https://micro-ros.github.io/docs/tutorials/advanced/benchmarking/ [2],
- https://github.com/micro-ROS/benchmarking_shadow-builder/ [3].

This document uses version captured on the 21st of October 2020.

2.1 Introduction to Benchmarking

Developing and working on stabilizing an application from the scribbles to the final executing binary is a long and hard task. During this process developers may come across stability and performances issues. In addition to these issues, some specified QoS might be difficult to quantify. Solving these problems without the proper tools might be a frustrating and a tedious task leading to reduce the developers efficiency. An adapted benchmarking tool could overcome most those development's obstacles and reduce development time. There are different KPIs (Key Performance Indicators) that one might be interested into. In the framework of this micro-ROS project, the KPI can be freely chosen by the developer. In this way, the benchmarking tool will remain flexible and allow the community to constantly add some support for a lot of different KPIs.

The problems to tackle are:

- Out there, many benchmarking tools exist. Each of them targeting different KPIs
- Platform dependency (Linux/Nuttx/Baremetal etc.)
- Too few time/resources to code benchmarking tools for each platform and/or KPIs
- Avoid code overhead: keep code clarity,
- Avoid execution overhead: do not want to make the execution slower when benchmarking.

2.2 Our Benchmarking tool framework

The benchmarking tool under development is providing a framework to allow developers to create their own benchmarking tool. Each part a developer wants to benchmark can be added as a plugin using the provided framework. In this way plugins can be shared and this improves re-usability as much as possible.

2.3 Architecture

The picture below depicts the overall architecture of the Shadow-Builder benchmarking tool framework. All the relevant elements constituting this framework are described in the following sections.

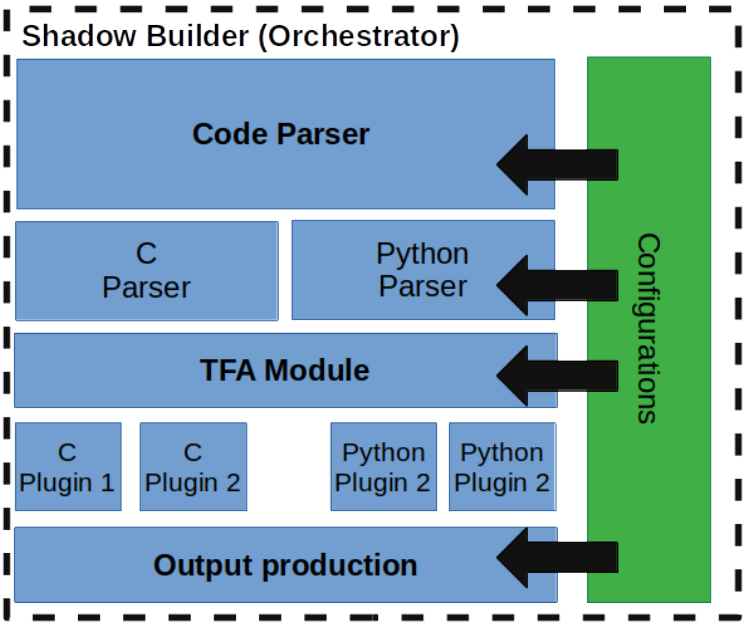

Figure 1: Illustration of general concept

2.4 Trace Framework Abstraction

The Shadow builder only parses comments from the application and passes them along to the Trace Framework Abstraction (TFA) Core. The TFA core is aware of the plugins that are available, all the plugins' capabilities and platform targeted. The process goes as explained below:

- The line containing the functionality Benchmarking::XX::YY is checked against all the available plugins
- Plugins that are capable of handling functionality respond and replace with a piece of code,
- Then the output file is added in a folder corresponding to the platform type and benchmarking type.

Being generic is the key for this benchmarking tool. However, the plugins, in contrary, brings the specific implementation needed to benchmark a specific platform. Every plugin provide information as requested by the parser:

- Provide a list of supported platforms
- Provide a list of functions that are handled
- Provide snippets codes that will be added for benchmarking
- Provide a list of patches and/or patch code,
- Optional provide an end script to run and execute the benchmarks

2.5 Shadow Builder

The Shadow Builder (SB) is a tool that will transparently instrument the code to benchmark. The tools will be able to output an "instrumented code" that will be compiled as a normal code. The following steps describe what the Shadow Builder process flow is:

- Get configuration file from the user (Benchmarking Configuration File)
- Get appropriate sources
- Execute Trace Framework Abstraction Configuration file
- Parse the sources file needed
- Injecting code
- Compile the targeted binary for different platform,
- If needed, depending on what type of benchmark is undertaken, compile another target binary benchmarked.

The SB (Shadow Builder) is meant to be as transparent as possible for the user. And if the benchmarking is not activated, it should be bypassed. # Source code

- Shadow Builder:
  - Git repository: https://github.com/micro-ROS/benchmarking_shadow-builder [4]
  - Path relative to the repository: ./shadow-builder
- Trace Framework Abstraction Core:
  - Git repository: https://github.com/micro-ROS/benchmarking_shadow-builder [4]
  - Path relative to the repository: ./tfa_core
- Trace Framework Abstraction Plugins:
  - Git repository: https://github.com/micro-ROS/benchmarking_shadow-builder [4]
  - Path relative to the repository: ./tfa-plugin
- Tutorial on how to write a Trace Framework Abstraction Plugin:
  - Git repository: https://github.com/micro-ROS/benchmarking_shadow-builder/tree/master/tutorial/01_create_plugin_time [3]
- Patched NuttX:
  - Git repository: https://github.com/micro-ROS/NuttX [5].
- Patched NuttX applications:
  - Git repository: https://github.com/micro-ROS/nuttx_apps [6].
- Multi-Connectors Board:
  - Git repository: https://github.com/micro-ROS/mcb [7].
- Other benchmark tools over Serial Wire Debug:
  - Git repository: https://github.com/microROS/benchmarking [8]. # Hardware used

This was covered in the deliverable 5.5 - Section 4.1 https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf[9]

2.6 Power measurement

The power measurement were achieved using the Multi-Connector Board (MCB). Using one of the shunt resistor available on the MCB, it's possible to measure the current. Measuring the current is done by using a voltmeter to measure the voltage at the shunt resistance's terminals.

Applying a simple formula:

```
Power_system = Power_whole - Resistor_shunt * (Current_whole ^ 2)
Current_whole = Voltage_shunt / Resistor_shunt
Power_whole = Voltage_source * Current_whole
```

Where the:

- Current_whole is the current taken from the power source.
- Voltage_shunt is the voltage at the shunt resistor's terminals
- Resistor_shunt is the shunt resistor value
- Power_whole is the power provided by the power source
- Voltage_source is the voltage provided by the power source,
- Power_system is the power that the system without the resistor needs.

3 Configuration

There is some information to know before using Shadow Builder. The architecture section provides a glimpse of the tool's internals. But before starting, the terminology will be introduced below.

3.1 Terminology

3.1.1 Tags

A tag is a comment that is used by the Shadow Builder. A comment formatted as follow:

- /** Benchmarking::ModuleGroup::Function */ : without any parameters,
- /** Benchmarking::ModuleGroup::Function(param0,...,paramN) */ : with parameters

is considered a *SB comment* also known as a **Tag**.

3.1.2 Trace Framework Abstraction

This is a framework provided to aid developers to create various plugin.

3.1.3 Plugins

Plugins are library files provide replacement for a **Tag**.

3.1.4 Parser

The element (front-end) in charge of looking for comment within a string. If this string comment is **Tag** it is dispatched.

3.2 Software components

The Shadow Builder is made of 4 core components that are:

- **The parser element**: This is just a backend tool that indicates the **TFA** module that a **Tag** was found in the source code. It is also in charge of writing a piece of code. Currently, the LLVM and Clang are used to parse and adapt the code. The Clang libtooling is documented more in details on the official webpage: https://releases.llvm.org/10.0.0/tools/clang/docs/index.html .
- **The TFA module**: This software is in charge of manipulating the different plugins and dispatching the **Tags** found by the the parser element. At first, it filters the comments to make sure that comments are formatted correctly. In addition it also sanitises the commentaries (also called tags). The **TFA** module is quite intuitive but more details are provided within the tfa_core/README.md.
- **The TFA plugins**: These are elements that respond to a **TFA** module dispatch when a commentary matches the commentaries. The plugins are basically a shared library file that is opened on startup. Some more details on how to use and write them are available here: tfa-plugins/README.md.
- **Shadow Builder**: The umbrella core module that is in charge of orchestrating the aforementioned modules. More about it here shadow-builder/README.md.

3.2.1 Source tree

Below the tree view of the important folders, files and their descriptions.

```
shadow_builder
    ├── common            --> Toolbox source files
    ├── examples          --> Examples of instrumented code
    ├── ext               --> External download and libraries
    ├── prepare_build.sh  --> Script fetching and installing the dependencies
    ├── res               --> Configuration files folder
    ├── shadow-builder    --> Shadow Builder source files
    ├── tfa_core          --> TFA's source files,
    └── tfa-plugins       --> TFA plugin's folder
```

3.3 Installation

3.3.1 Dependencies

First of all, in order to start compiling the Shadow Builder it is needed to retrieve a list of dependencies. Note that this guide was tested on Ubuntu 18.04.

```
$ sudo apt install git libzmq3-dev binutils libssl-dev python3-distutils python3
 # To get a more recent version of cmake.
$ sudo snap install cmake --classic
```

In order to build clang and LLVM, GCC 9 is needed. Under the official PPA, it is not possible to find it. Therefore, it is recommended to install from a test ppa:

```
# Accept the key by pressing enter
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt update
$ sudo apt install gcc-9 g++-9 g++-9-multilib
```

Because it is possible to have several versions of GCC on a machine, the update-alternative will be used to set the correct GCC version. If the alternative for GCC is available and set up, the block below can be skipped.

```
$ sudo update-alternatives --remove-all gcc
$ sudo update-alternatives --remove-all cc
$ sudo update-alternatives --remove-all g++
$ sudo update-alternatives --remove-all c++

# This is current gcc compiler
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-xxx 10
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9 20

 # This is the current g++ compiler
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-xxx 10
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-9 20

$ sudo update-alternatives --install /usr/bin/cc cc /usr/bin/gcc 30
$ sudo update-alternatives --set cc /usr/bin/gcc

$ sudo update-alternatives --install /usr/bin/c++ c++ /usr/bin/g++ 30
$ sudo update-alternatives --set c++ /usr/bin/g++

$ sudo update-alternatives --install /usr/bin/c++ c++ /usr/bin/g++ 30
$ sudo update-alternatives --set c++ /usr/bin/g++
```

Then select the gcc9 and g++9 as following:

```
$ sudo update-alternatives --config gcc
There are 2 choices for the alternative gcc (providing /usr/bin/gcc).

  Selection    Path            Priority   Status
------------------------------------------------------------
* 0            /usr/bin/gcc-9   20        auto mode
  1            /usr/bin/gcc-7   10        manual mode
  2            /usr/bin/gcc-9   20        manual mode

Press <enter> to keep the current choice[*], or type selection number: 2
```

And repeat those steps for:

- c++ –> selecting /usr/bin/g++
- cc –> selecting /usr/bin/gcc
- gcc –> selecting /usr/bin/gcc-9
- g++ –> selecting /usr/bin/g++-9

It is possible to check if the right version of gcc and g++ in the command line:

```
$ gcc -v
$ g++ -v
$ cc -v
$ c++ -v
```

3.3.2 Compilation

Before starting, some internal dependencies must be retrieved. To do so, the script at the root of folder should be called as follow:

```
$ ./prepare_build.sh # retrieves/compiles and installs dependencies
```

Once this step is done, the ext/dl subdirectory can be removed. This will save a lot of space. Note that compilation might take some time.

Then once all dependencies are built, the Shadow Builder is ready to be compiled:

```
$ mkdir -p build; cd build
$ cmake ..
$ make -j4
```

3.3.3 First run

To run the the Shadow Builder against the examples (located ina examples/example_stupid_main), the following steps to perform are:

```
$ cd build # previously created during the compilation step.
$ ./shadow-program -s ../res/sb-res/bcf.xml -t ../res/tfa-res/tfa.xmlb
```

This command will create an instrumented code that will be generated in the folder /tmp/output/test_DATE.

Once in the folder, the following commands will compile and execute the code:

```
$ cd /tmp/output/test_DATE/
$ mkdir -p build; cd build
$ cmake .. ; make
$ ./simple_stupid_example
```

The output of this command should be:

```
Monitor var i: 0
Monitor var i: 4294967295
Exe time: 8 sec : 758 ms : 534352 ns
```

To understand in details, it is recommended to go through all configurations file's examples, the examples code and their *READMEs* files.

For more details, referring to all *READMEs* files are a good start. The code is documented as well and can provide additional information about behaviors at a very low level.

3.3.4 Configuration files

The Shadow Builder is using XML configuration files to locate the **TFA** plugins folder **currently only one folder can be provided**. By default the plugin folder is located in the folder /tfa-plugins/

More information regarding the different attributes and node fo the XML configuration can be found here: res/README.md .

3.3.5 Plugins

Plugins are the "Responder" to *Tags* that are found in the code. This *Tags* are used to tell the Shadow Builder and more specifically its plugins that a some benchmarking request are made in the code.

The response is up to the implementation in the plugin. More details, can be found here tfa-plugins/README.md .

3.3.6 Unit tests

Unit tests are available in the folder /tests/xxx_<xx>.cpp Each test shall be added in the CMake-File.txt in the module under tests.

In addition the option -DENABLE_TESTS=ON shall be passed to the cmake to activate tests creation.

3.4 Tested Platforms

The Shadow Builder was successfully compiled with ubuntu 18.04 LTS with the following configuration:

- 2 Core / 4 Threads Intel Processor
- 4GB DDR4 RAM and 8 GB of swapping memory,
- 60 GB of disk space

4 Example

4.1 Introduction

This section is dealing with one specific benchmarking tool called the **Shadow Builder**. More specifically, this tutorial aims to create a plugin from A to Z and how to instrument the code to benchmark.

For the sake of ease of understanding, this tutorial is proposing to benchmark the time spent on a simple looping function.

4.1.1 Get appropriate sources

The SB is in charge of getting the path/git repository to the source code that needs to be benchmarking.

- The benchmarking:
  - The sources are specified by the user in the benchmarking configuration file.
- Injecting code:
  - In order to inject code, there are many existing for such a task. The Clang AST tool was chosen to inject code as it gains attraction in the developer community.

4.2 Binary generation for instrumented code

The binary generation is the process of compiling the source code. In order to benchmark, it is necessary to instrument the code. The code will be instrumented in a transparent way for the programmer. To limit and configure the instrumentation scope, a configuration file is provided by the programmer. This configuration is parsed and the code is injected as described in a configuration file.

4.2.1 Receiving inputs

The binary generation's pipeline receives two inputs to work with:

- Configuration benchmarking file,
- Source code to benchmark.

In short, the configuration describes:

- What is benchmarked (sources)
- Where to benchmark, or in other words, where to retrieve the sources.
- What type of benchmark what KPI is targeted.
- Optionally against what base line to compare to (base line source)

4.2.2 Parse and Check

Once the input received, the shadow builder parses the configuration file. From the configuration file, the shadow builder obtains:

- The different benchmarkings to be performed.
- The targeted platforms.

In addition to parsing, the Shadow Builder is in charge of checking capabilities and consistency within the configuration file and the different TFA's plugins registered in the TFA module.

4.2.3 TFA Execution

Once parsed and checked against the TFA module capabilities, the shadow builder is in charge of translating instrumented code into source code. The source is translated in cooperation with the TFA module. At the end of this step, the TFA generates the new forged source code ready for compilation. In addition to the forged source code, the TFA will generate scripts that will the execute the benchmarks.

4.2.4 Compilation

The compilation can be executed for every type of benchmarks and platforms targeted. Depending on the type of benchmark that is being executed, there is one or more binaries per benchmarks session. The number of binary generated also depends on what plugins are provided by the user to the shadow builder. The shadow builder retrieves capabilities of the plugins and request from the developer, matches them and generates software according to the matches.

4.2.5 Steps to start benchmarking

The shadow builder executes the following steps:

- Software sources are passed to the shadow builder.
- The source is parsed and upon comments containing */Benchmarking::XX::YY/* (a tag) the code line is passed to the Trace Framework Abstraction module. Using comments is preferable → No includes header needed.
- All plugins that are registered to the TFA with the Benchmarking::XX::YY functionality will return a piece of code is to be added to the source.
- Once all parsed, the shadow builder compiles the target for all the different platforms requested either according to the plugins or according to the user configuration.

4.3 Create a plugin

In order to create a plugin, the information that is crucial to figure out are:

- What is to benchmark? –> The time spent in a a function.
- How to do so? –> Is there a plugin already supporting it? Yes, then to do. And the code to profile can be instrumented.

If no plugin supports it, then a plugin has to be created. Then, another set of questions arises, which are (according to the context):

1. How could it benefit to others?
2. What piece of code would be used to measure the time? (In C or C++?)
3. What platform can it support? (OS, CPU, etc.)
4. How should the code be instrumented?

The answers to these questions would be:

1. Create a generic plugin and write a documentation that would be understandable for a normal user and an expert user.
2. Using the `timespec` and `clock_gettime` Linux syscall.
3. From previous answer –> OS: Linux on any CPU as long as it has the same Linux API.
4. Using a simple way: A comment could be created as follow `/** Benchmarking::plugin_name::function */` . The choice for the current tutorial would be `/** Benchmarking::TimeBenchmarking::Timer */` These answers provide us with the minimum necessary for the creation of a plugin.

4.3.1 Create a TFA-Plugin

4.3.1.1 File tree structure

The final code shall be located in `src_root_sb/tfa-plugin/TimeBenchmarking` with the following structure:

```
TimeBenchmarking
    ├── CMakeLists.txt
    ├── inc
    │   └──TimeBenchmarking
    │       └── TimeBenchmarking.h
    └── src
        └── TimeBenchmarking.cpp
```

4.3.1.2 Register a new plugin

The **SB** is relying on TFA's plugins to be executed in response to the parser dispatch. Therefore, some interoperability is needed.

All new plugins are written by implementing the IPlugin interface as shown in the file `src_root/tfa_core/inc/tfa/IPlugin.h`. The interface needs to implement the pure virtual function. A simple example would be as in the plugin_test:

In the plugin header:

```
class TimeBenchmarking: public IPlugin {
public:
    TimeBenchmarking();
    ~TimeBenchmarking();

    TFAInfoPlugin& getInfoPlugins();
    bool initializePlugin();
};

extern "C" IPlugin* create() {
    return static_cast<IPlugin *>(new TimeBenchmarking);
}
extern "C" void destroy(IPlugin* p) {
    delete p;
}
```

In the plugin source code:

```
TimeBenchmarking::TimeBenchmarking() {}

TimeBenchmarking::~TimeBenchmarking()
{
    if(mInfos) {
        delete mInfos;
    }
```

```
}
```

4.4 Create a listener

Now the plugin is ready to be registered within the TFA's core. So when a session is running, the plugin will be found. However nothing happen at this stage. Indeed the plugin written is not listening to a specific tag.

Just as a reminder, the listener is an object derived from the interface **ITFACommentListener**. It is listening to as specific *Tag* which will be replace by a piece of code.

The declaration of the object shall be as display below:

```
class Timer: public ITFACommentListener
{
public:
    Timer();
    Status runnableComments(const TFACommentInfo& cleanComment,
                std::string& replacement);
};
```

As shown above, the class is inheriting from the ITFACommentListenner class. The ITFACommentListener has one pure-virtual method called runnableComments. This means that the plugin has to implement the method runnableComments(...).

```
Timer::Timer()
 :
    ITFACommentListener("Benchmarking::User::Timer")
{
}

Status Timer::runnableComments(const TFACommentInfo& cleanComment,
        std::string& replacement)
{
    return Status::returnStatusError();
}
```

Now, the functions are correctly implemented. The timer needs several things to measure the time spent in a function:

1. Start the timer before the function, get an initial timestamp
2. Stop the timer after the function has returned, get another timestamp
3. Measure the delta between the two timestamp measured above.
4. Print the delta in a human readable.

The plugin will need a way to get the timestamps, by using the clock_gettime, and print it to the user by using printf.

A tag can be provided by several parameters. These are useful for the sake of the timer:

- A parameter to identify what's is the timer's status (i.e. start or stop),

- A parameter to identify the timer itself in a unique way by the dev,
- A parameter that is needed for header declaration.

The concrete example below:

```
#include <time.h> // Needed to access the clock_gettime() function
#include <stdio.h> // Needed to access the printf function.

void func2benchmark(...)
{

    /** declare and measure the starting timestamp */
    struct timestamp timer_start, timer_stop;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &timer_start);
    // do something very slow

    /** Measure the timesamp now, process and show the results */
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &timer_start);
    {
        struct timespec *start = &timer_start;
        struct timespec *stop = &timer_stop;
        struct timespec result;

        if ((stop->tv_nsec - start->tv_nsec) < 0) {
            result.tv_sec = stop->tv_sec - start->tv_sec - 1;
            result.tv_nsec = stop->tv_nsec - start->tv_nsec + 1000000000;
        } else {
            result.tv_sec = stop->tv_sec - start->tv_sec;
            result.tv_nsec = stop->tv_nsec - start->tv_nsec;
        }

        printf("Exe time: %ld sec : %ld ms : %ld ns\\n",
            result.tv_sec, result.tv_nsec / 1000000, result.tv_nsec % 1000000);
    }
}
```

Using the TFA, the code would be as follow:

```
/** Benchmarking::TimeBenchmarking::Timer(declare) */

void func2benchmark(...)
{
/** Benchmarking::TimeBenchmarking::Timer(start, timer1) */
    //do something very slow
/** Benchmarking::TimeBenchmarking::Timer(stop, timer1) */
}
```

When comparing the differences, the amount of overhead code clarity introduced by the method is very low.

4.4.1 Declare replacement

In order to declare the includes needed to benchmark, the way to do it would be to get the parameter 0 to be a string that matches "declare". The replacement would be simple headers.

```
#include <time.h>
#include <stdio.h>
```

It is needed to append \n to the end of a line, as this piece of code is going to be appended to the code.

```
Status Timer::runnableComments(const TFACommentInfo& cleanComment,
        std::string& replacement)
{
    const std::vector<std::string> params = comment.getParams();

    if (params.size() == 1 && params[0] == "declare") {
        replacement = "#include <time.h>\n";
        replacement += "#include <stdio.h>\n";
        return Status::returnStatusOkay();
    }

    return Status::returnStatusError();
}
```

It is mandatory to return Status::returnStatusOkay() to tell the tfa-core that the *Tag* was handled and therefore that no other plugin will be using it.

4.4.2 Start tag

The starting element will basically record a timestamp in the memory. The following actions are required to do it in C programming on a Linux system:

```
Status Timer::runnableComments(const TFACommentInfo& cleanComment,
        std::string& replacement)
{
    const std::vector<std::string> params = comment.getParams();


    if (params[0] == "start" && params.size() == 2) {
        std::string start = "timer_start_" + params[1];
        std::string stop = "timer_stop_" + params[1];

        replacement = "struct timespec " + start + ", " + stop + ";\n";
        replacement += "\tclock_gettime(CLOCK_PROCESS_CPUTIME_ID,
                &timer_start_" + params[1] + ");\n";
        return Status::returnStatusOkay();
    }
```

```
    return Status::returnStatusError();
}
```

It is mandatory to return Status::returnStatusOkay() to tell the tfa-core that the *Tag* was handled and therefore that no other plugin will be using it.

4.4.3 Stop tag

The stopping element, which will be in charge of getting a timestamp, compute the delta time spent between the stop and the start and finally print in a human-readable way.

```
Status Timer::runnableComments(const TFACommentInfo& cleanComment,
        std::string& replacement)
{
    const std::vector<std::string> params = comment.getParams();
    const char difftime_func[] =
    "{\n\
        struct timespec *start = &%s;\n\
        struct timespec *stop = &%s;\n\
        struct timespec result;\n\
        if ((stop->tv_nsec - start->tv_nsec) < 0) {\n\
            result.tv_sec = stop->tv_sec - start->tv_sec - 1;\n\
                result.tv_nsec = stop->tv_nsec - start->tv_nsec + 1000000000;\n\
        } else {\n\
            result.tv_sec = stop->tv_sec - start->tv_sec;\n\
                result.tv_nsec = stop->tv_nsec - start->tv_nsec;\n\
        }\n\
        printf(\"Exe time: %%ld sec : %%ld ms : %%ld ns\\n\",\n\
                result.tv_sec, result.tv_nsec / 1000000, result.tv_nsec\n\
                %% 1000000);\n\
    }\n";

    if (params[0] == "stop" && params.size() == 2) {
        std::string start = "timer_start_" + params[1];
        std::string stop = "timer_stop_" + params[1];
        char buf[sizeof(difftime_func) + start.length() +
            stop.length()];

        sprintf(buf, difftime_func, start.c_str(), stop.c_str());
        replacement += "clock_gettime(CLOCK_PROCESS_CPUTIME_ID,
                &timer_stop_" + params[1] + ");\n";
        replacement += string(buf);

        return Status::returnStatusOkay();
    }

    return Status::returnStatusError();
```

}

It is mandatory to return Status::returnStatusOkay() to tell the tfa-core that the *Tag* was handled and therefore that no other plugin will be using it.

It is worth keeping in mind that the replacement code is actual C code that is going to be compiled. Therefore, one should be careful about the string formatting (e.g line return, escaping characters).

4.4.4 Summary

The whole runnableComment method would look like what follows:

```
Status Timer::runnableComments(const TFACommentInfo& cleanComment,
        std::string& replacement)
{
    const char difftime_func[] =
    "{\n\
        struct timespec *start = &%s;\n\
        struct timespec *stop = &%s;\n\
        struct timespec result;\n\
        if ((stop->tv_nsec - start->tv_nsec) < 0) {\n\
            result.tv_sec = stop->tv_sec - start->tv_sec - 1;\n\
            result.tv_nsec = stop->tv_nsec - start->tv_nsec + 1000000000;\n\
        } else {\n\
            result.tv_sec = stop->tv_sec - start->tv_sec;\n\
                result.tv_nsec = stop->tv_nsec - start->tv_nsec;\n\
        }\n\
        printf(\"Exe time: %%ld sec : %%ld ms : %%ld ns\\n\",\n\
                result.tv_sec, result.tv_nsec / 1000000, result.tv_nsec \n\
                %% 1000000);\n\
    }\n";

    const std::vector<std::string> params = comment.getParams();

    if (!params.size())
    {
        return Status::returnStatusError();
    }

    if (params[0] == "declare" && params.size() == 1) {
        replacement = "#include <time.h>\n";
        replacement += "#include <stdio.h>\n";
        return Status::returnStatusOkay();
    } else if (params[0] == "start" && params.size() == 2) {
        std::string start = "timer_start_" + params[1];
        std::string stop = "timer_stop_" + params[1];
```

```
        replacement = "struct timespec " + start + ", " + stop + ";\n";
        replacement += "\tclock_gettime(CLOCK_PROCESS_CPUTIME_ID, \
                &timer_start_" + params[1] + ");\n";
        return Status::returnStatusOkay();
    } else if (params[0] == "stop" && params.size() == 2) {
        std::string start = "timer_start_" + params[1];
        std::string stop = "timer_stop_" + params[1];
        char buf[sizeof(difftime_func) + start.length() +
            stop.length()];

        sprintf(buf, difftime_func, start.c_str(), stop.c_str());
        replacement += "clock_gettime(CLOCK_PROCESS_CPUTIME_ID, \
                    &timer_stop_" + params[1] + ");\n";
        replacement += string(buf);

        return Status::returnStatusOkay();
    }

    return Status::returnStatusError();
}
```

4.5 Register the listener

Once the listener is implemented, then it needs to be registered within the TFA plugin manager:

```
bool TimeBenchmarking::initializePlugin()
{
    /* This is a plugin compatibility platform */
    tbp = new TFABenchMarkingPlatform("Linux", "*", "*", "*");

    /** Here register the Timer listener */
    iclVect.emplace_back(static_cast<ITFACommentListener *>(new
                    Timer));

    /* This is the infoPlugin that holds the plugin name and the
        platform information */
`    mInfos = new TFAInfoPlugin("Test Plugin", *tbp);

    // Will be explained later how to mock up this part.
    return Status::returnStatusOkay();
}
```

A protected vector, inherited from the IPlugin class, needs to be appended for each listener which this plugin will be supporting and implementing.

4.6 Build

To compile the plugin this must be done from the build folder created beforehand in the **Shadow Builder** section.

4.6.1 Build configuration

The build configuration file will be the CMakeLists.txt at the root of the plugin

It shall look like the follows:

```
project(TimeBenchmarking VERSION 0.1 DESCRIPTION "MY Plugin")
set(CMAKE_CXX_STANDARD 14)

set(PLUGIN_NAME "TimeBenchmarking")

# Needed to get the function to do tests
include(../../CMakeMacros/CMakeTesting.txt)


# Plugins include folders
include_directories(inc/)

# Plugins source files
list(APPEND TEST_PLUGIN_SRC
        src/TimeBenchmarking.cpp
)

# Needed to create a shareable library
add_library(${PLUGIN_NAME} SHARED ${TEST_PLUGIN_SRC})

# Target library that link against
target_link_libraries(tfa)

# Needed to create a version of the shared library
set_target_properties(${PLUGIN_NAME} PROPERTIES SOVERSION ${PROJECT_VERSION})
```

Finally, it is needed to add into the parent's folder (i.e. src_root/tfa-plugins/CMakeLists.txt) CMakeLists.txt the subdirectory of the plugin:

```
cmake_minimum_required(VERSION 3.10)  # CMake version check
set(CMAKE_CXX_STANDARD 14)

add_subdirectory(plugin_test)
add_subdirectory(myplugin)

# Add project configuration here:
add_subdirectory(TimeBenchmarking)
```

The example is available here

4.6.2 TFA configuration

An example of the configuration file is in the source tree at `src_root/res/tfa-res/tfa.xml`.

This file only keeps track of the path where to look for plugins. It is important to note that this file is a template and renewed at each compilation.

In the current context, the path is the default one (i.e. src_root/build):

```
$ cd src_root/build/
$ make -j4
```

4.6.3 The SB configuration

The **SB** configuration is providing some hints where the source files to benchmark can be found and where the output folder should be set.

A detailed explanation can be found at the following link https://github.com/micro-ROS/benchmarking_shadow-builder/blob/master/res/README.md#shadow-builder-configuration.

4.7 Running the Shadow Builder

Once all the above steps are done and the plugin is compiled the command to run the code's instrumentation would be:

```
$ cd src_root/build/
$ ./shadow-program -s ../res/sb-res/bcf.xml -t ../res/tfa-res/tfa.xml
```

The output should be pushed into the folder that was provided as a parameter of the cbf configuration file (by default should /tmp/output/**session_name**). The **session_name** is the session's name (test by default) to which is appended the date and time when the benchmarking was started.

5 Other concepts and tools

5.1 Common trace format in Nuttx

In order to retrieve data/trace, an open-source standard targeting low resource impact and great flexibility was integrated in the RTOS (NuttX/Zephyr).

More about this format https://diamon.org/ctf/ [10]

In NuttX, the CTF library has been added in order to get the RTOS or application running on top of the RTOS to provide information about specific data.

To run the babletrace the following command need to be called from within the application:

```
sys_trace_ctf_meas_start(); // To start the measurements
//...
sys_trace_ctf_meas_stop(); // To stop the measurements
```

In addition a metadata file describes the way the frame will look like as shown below:

```
/* CTF 1.8 */
typealias integer { size = 8; align = 8; signed = true; } := int8_t;
typealias integer { size = 8; align = 8; signed = false; } := uint8_t;
typealias integer { size = 16; align = 8; signed = false; } := uint16_t;
typealias integer { size = 32; align = 8; signed = false; } := uint32_t;
typealias integer { size = 64; align = 8; signed = false; } := uint64_t;
typealias integer { size = 8; align = 8; signed = false; encoding = ASCII; }
    := ctf_bounded_string_t;
typealias enum : uint32_t {
    MUTEX_INIT = 33,
    MUTEX_UNLOCK = 34,
    MUTEX_LOCK = 35,
    SEMA_INIT = 36,
    SEMA_GIVE = 37,
    SEMA_TAKE = 38
} := call_id;

typealias enum : uint8_t {
    FUNCTION_ENTER = 0,
    FUNCTION_EXIT = 1,
} := function_exec;

typealias enum : uint8_t {
    COM_TX = 0,
    COM_RX = 1,
} := com_start_end_src_t;

clock {
    name = cycle_counter_sync;
    uuid = "62189bee-96dc-11e0-91a8-cfa3d89f3923";
    description = "Cycle counter synchronized across CPUs";
    freq = 1000000000;          /* frequency, in Hz */
    /* precision in seconds is: 1000 * (1/freq) */
    precision = 100;
    /*
     * clock value offset from Epoch is:
     * offset_s + (offset * (1/freq))
     */
    absolute = TRUE;
};

struct event_header {
    uint64_t timestamp;
    uint8_t id;
};

trace {
```

```
    major = 1;
    minor = 8;
    byte_order = le;
};

stream {
    event.header := struct event_header;
};

event {
    name = thread_switched_out;
    id = 0x10;
    fields := struct {
        uint32_t thread_id;
    };
};

event {
    name = thread_switched_in;
    id = 0x11;
    fields := struct {
        uint32_t thread_id;
    };
};
....
```

Measuring the output can be achieved by using an UART device. iit is possible to run the following command, on a PC, to retrieve the data on from a UART device connected to the NuttX's device:

```
cat /dev/ttyUSBXXX > output # or /dev/ttyACMX
```

The device will start to post a flag 0xdeadbeaf signaling that the software is running and that the benchmarks starts.

Once the application is finished, an analysis is performed using the BabelTrace API as explained in the following section.

5.2 BabelTrace

The Babeltrace is a tool and API that help a user to interpret the output from a trace following the common trace interface.

More about the API and the software interpreter https://babeltrace.org [11]

Example using python script is available in the following link https://babeltrace.org/docs/v1.5/python/babeltrace/examples [12]

It's up to the final user to write a Python script to retrieve and interpret the data as needed.

5.3 Other benchmarking tools

This was covered in the deliverable 5.5 - Section 4 https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf[9]

5.4 Serial Wire Debug

This was covered in deliverable 5.5 - Section 4.1 https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf[9]

5.4.1 Performance execution analysis

This was covered in the deliverable 5.5 - Section 5.1 https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf[9]

5.4.2 Memory analysis

This was covered in the deliverable 5.5 - Section 5.2 https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf[9]

5.5 Network emulation tool

In order to benchmark an application under various network conditions, *Netem* https://man7.org/linux/man-pages/man8/tc-netem.8.html [13] was used.

This was covered in the deliverable 5.5 - Section 5.2 https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf[9]

5.6 Executor Benchmarking Suite

This was covered in deliverable 5.5 - Section 2.6 https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf[9]

6 Conclusions

Tools described here were developed to improve quality micro-ROS framework and to make development and benchmarking more convenient. Owing to the very generic approach taken for these tools, any other C/C++ projects could use them as needed. The potential and simplicity of theses tools use will allow their extensive adoption in the future.

References

[1] A. M. OFERA, 'Benchmarking concepts.' [Online]. Available: https://micro-ros.github.io/docs/concepts/benchmarking/

[2] A. M. OFERA, 'Benchmarking with the Shadow Builder.' [Online]. Available: https://micro-ros.github.io/docs/tutorials/advanced/benchmarking/

[3] Łukasiewcz-PIAP, 'Shadow Builder Tutorial.' [Online]. Available: https://github.com/micro-ROS/benchmarking_shadow-builder/tree/master/tutorial/01_create_plugin_time

[4] Łukasiewcz-PIAP, 'Shadow Builder.' [Online]. Available:%20https://github.com/micro-ROS/benchmarking_shadow-builder

[5] Łukasiewcz-PIAP, 'NuttX Instrumented.' [Online]. Available: https://github.com/micro-ROS/NuttX

[6] Łukasiewcz-PIAP, 'NuutX application Instrumented.' [Online]. Available: https://github.com/micro-ROS/nuttx_apps

[7] Łukasiewcz-PIAP, 'MCB.' [Online]. Available: https://github.com/micro-ROS/mcb

[8] Łukasiewcz-PIAP, 'Micro-ROS benchmarking tools.' [Online]. Available: https://github.com/micro-ROS/benchmarking

[9] Łukasiewcz-PIAP, 'Micro-ROS benchmarking and validation tool release.' [Online]. Available: https://github.com/micro-ROS/benchmarking-results/blob/master/pdfs/OFERA_56_D5.5_Micro-ROS_benchmarking_and_validation_tools_Release_-_Beta.pdf

[10] efficios, 'Common Trace Format.' [Online]. Available: https://diamon.org/ctf/

[11] efficios, 'BabelTrace.' [Online]. Available: https://babeltrace.org

[12] efficios, 'Babletrace API examples.' [Online]. Available: https://babeltrace.org/docs/v1.5/python/babeltrace/examples

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