Python In Real Time Application For Mobile Robot

RABBAH Mahmoud Almostafa
RITM Laboratory, ESTC, Hassan II University, BP. 8012
Casablanca, Morocco
mrabbah@ieee.org

RABBAH Nabila
Laboratory of Structural Engineering, Intelligent Systems and Electrical Energy, ENSAM, Hassan II University
Casablanca, Morocco

BELHADAOUI Hicham
RITM Laboratory, ESTC, Hassan II University, BP. 8012
Casablanca, Morocco

RIFI Mounir
RITM Laboratory, ESTC, Hassan II University, BP. 8012
Casablanca, Morocco

Abstract— **Commanding some part of mobile robots requires real time capability, especially for industrial need and applications where predicted behavior is paramount. Many programming language can be used to develop robot application, but only few of them can satisfy real time capability. C and C++ are the most used when addressing this problem. Our study will focus on major problem that must be overmastered to be able to develop a real time application using interpreted language. Various strategies for developing on top of a real time operating system (RTOS) are explored. The patterns focus on the use case of Python development on Linux-based RTOS's (such as RT_PREEMPT), but the general concepts are applicable to other platforms.**
This contribution is related to our study for middleware for mobile robots in smart environment [1], and will be a first step to meet the criterion "Real Time Capability" of the middleware and will be an answer in which language we can develop the middleware to met this criterion.

Keywords— Middleware; Mobile robot; embedded system; Python; RTOS

I. INTRODUCTION

In general, an operating system (OS) can guarantee that the tasks it handles for the developer, such as thread scheduling, are deterministic, but the OS may not guarantee that the developer's code will run in real-time. Therefore, it is up to the developer to know what the deterministic guarantees of an existing system are, and what she must do to write hard real-time code on top of the OS.

In the last few years, Python language was used massively for mobile robot and artificial intelligence (AI) thanks to its simplicity, and we can find a lot of framework that were developed base on this language among other: SimpleAI for general AI, Pybrain for machine learning, NLTK for natural language processing, NEUROLAB for neural networks, and for mathematics science and engineering there are plenty of useful libraries like: Numpy and Scypy, and for robotics application we can find among other : TensorFlow, OpenCV…

Python has become quite the trending program language over the last few years, ranking first related to IEEE Spectrum recent study [2]. Named after the famous Monty Python comedy group, the language is object oriented and interpreted (not compiled). This attribute has resulted in Python being adopted on platforms such as Linux and Windows, and on single board computers such as the Raspberry Pi. With such a wide and growing adoption, one might wonder if there is a place for Python in real-time embedded systems.

When people speak of Python they often mean not just the language but also the CPython implementation. Python is actually a specification for a language that can be implemented in many different ways like (Jython, IronPython, PyPy, Cython or MicroPython).

This paper is organized as follows: In section 2 we will explain our motivation and background. Section 3 briefly surveys the related work. Section 4 presents the problem and the aspects that we must be wary of them when developing Real time mobile robot application. Section 4 introduces our solution and the recommended practices. Section 5 concludes the paper and outlines some possible future work.

II. MOTIVATION AND BACKGROUND

Before developing any real robot application, most developers choose a middleware that will provide the application the necessary architecture for communication, managing data in distributed context, and allowing independence from the heterogeneity of hardware and OS. There are some famous middleware that target robotics or connected objects application like ROS, ACOSO, JCAF, AURA, UBIWARE and Voyager. In our last work [1], we present the challenges facing these middleware; one of them was the real time capability. And based on this study we launched the development of the Collaborative Open Platform for Distributed Artificial Intelligence (COPDAI).

While starting developing the COPDAI middleware, we were faced to a challenge of choosing suitable programming language, one of the criteria that must be satisfied is the real time capability. So, the normal way is to go for a language like

C or C++, but as we want our middleware to be independent from OS and the hardware, the other way was to choose a height programming language like JAVA, Python or NodeJS. If we choose the first option we must develop as many version as existing OS and hardware, and it will take a lot of time. If we choose the second option, our middleware will be OS independent but we must guarantee the real time capability of the chosen language. We decided to study the second option in this contribution. We started by eliminating JAVA and NodeJS because there isn't enough supported driver on embedded cards that we use in our robots projects, like (Raspberry PI3, BeagleBone…), and also, the biggest communities that develop application on these embedded card use either C or Python. Therefore, we decided to deeply studying Python programming language, and see what are the advantages and disadvantages of this language, and how they can be surpassed.

Noted that although if we choose a specific programming language in which we will develop our middleware, we will allow the end developer, how will use COPDAI to choose any programming language, throw exposing a Remote API.

III. Related work

In this paper [3] authors present a middleware implemented based on Real-Time Specification for Java, named RT-MED, which corresponds to a software layer to be placed above the OS, it allows also managing reconfigurations that are unpredictable and depend on the reaction between user, system and environment. The proposed solutions in this middleware affect both the hardware part by acting in the processor speed during execution and the software part by changing the basic parameters of OS tasks such as deadlines and periods.

In this work [4], authors evaluate performance of ROS2, which is the upgrade of ROS1 by utilizing the Data Distribution Service. The main goal of ROS2 is to provide the real time capability, for now ROS2 is under heavy development, it support the real time communication capability but doesn't support until now neither the real time abstraction over RTOS nor ARM board, knowing that most mobile robots use embedded cards based on ARM architecture like (Jetson TX2, Raspberry Pi, BeagleBone, Orange Pi…).

MicroPython is a Python interpreter (with partial native code compilation feature). It provides a subset of Python 3.5 features, implemented for embedded processors and constrained systems. MicroPython is not alone though. Companies such as Synapse and OpenMV are using either Micro Python or their own Python port within embedded systems. In this work [5] MicroPython was ported to LEON Platforms, and in this paper [6] authors investigate the potential use of MicroPython in CubeSats special project, by analyzing the language and tools in practical examples from the MOVE-II.

The authors in this contribution [7] tried to use Real-time Java for industrial robot control; Real time threads were used for all the time critical tasks, including 1 kHz position control loop.

IV. The Problems

When using a programing language for real time application, we can face many problems, some exist regardless the programming language chosen, and others are specific. In this section, we will present these problems:

- Latency: We must guarantee that our process doesn't be pre-empted while dealing with a critical event, and we must also guarantee the latency of a process. The variation in latency is usually called "jitter", 1ms maximum jitter means that an interrupt arriving repeatedly will have a response latency that varies by at most 1ms, there isn't any direct way proposed by Python to set process priority or scheduler.

- Garbage collector (GC): Each object in python can have lots of name "Label". We have two types of objects: simple and containers; a reference is a name or a container object pointing at another object, and the reference count is the number of reference object have. The "del" operator doesn't delete the object its decrease the reference number, another ways to decrease reference number are affecting another value to a label or name, and going out of scope (leave a function). The garbage collector (GC) is responsible for releasing a memory when an object that taken this memory is no longer in use. There are two types of GC: reference counting and tracing, the GC reference counting delete object from memory when the reference equal zero, the tracing strategy uses "Mark and Sweep" algorithm, also Python use the generation strategy to lake to cyclic reference problem. Python's stop-the-world garbage collector makes latency non deterministic. When python decides it needs to run the garbage collector, the program gets stopped until it finishes.

- Global Interpreter Lock (GIL): Another notorious problem (at least for the standard CPython implementation of Python) is the GIL. The GIL prevent multiple python thread execute same code at the same time, so there are one GIL for each interpreter, allowing reference count being changed concurrently, the down side is in Python program, no matter how many threads exist, only one thread will be executed at a time, which means any attempt at using multithreading in order to gain concurrency benefits will be futile.

- Device I/O: It is essential that real-time processes have all their memory kept in physical RAM and not paged out to swap. There is no good way of controlling this in Python, especially running on Windows. Interacting with physical devices (disk I/O, printing to the screen, etc.) may introduce unacceptable latency in the real-time code path, since the process is often forced to wait on slow physical phenomena. Additionally, many I/O calls such as open result in page faults.

- Memory management: Proper memory management is critical for real-time performance. In general, the programmer should avoid page faults in the real-time code path. During a page fault, the CPU pauses all computation and loads the missing page from disk into

RAM (or cache, or registers). Loading data from disk is a slow and unpredictable operation. However, page faults are necessary or else the computer will run out of memory.

- Copy-on-Write (CoW): Linux kernel has a mechanism called CoW that serves as an optimization for forked processes. A child process starts by sharing every memory page with its parent. A page copied to the child's memory space only when the page is written to, because of reference counting, every time we read a Python object, the interpreter will increase its refcount, which is essentially a write to its underlying data structure. This causes CoW and leads to page faults.
- Exception: How can we catch all exceptions and how long it will take to find a matching catch? The throw is typically banned in hard real-time applications.

V. Solution

In this section, various strategies for developing on top of a real-time OS are explored. Each actuator or sensor will be presented in COPDAI as Node. So, we will have just one process per task, and no need to use multithreading, against we will use multiprocessing, and so avoid the GIL downside, collaboration between these processes will not be discussed in this contribution. We design a base class named "BaseNode" to encapsulate all the patterns that we will implement for our Node.

It is a common pattern to section real-time code into three parts: a non real-time safe section at the beginning of a process that pre-allocates memory on the heap, a real-time safe section (often implemented as a loop), and a non-real-time safe "teardown" section that deallocates memory as necessary. The "real-time code path" refers to the middle section of the execution.

Solving the latency problem: At the constructor level, we will indicate if yes or no this task is real time, also it takes as optional parameters: the scheduling policy and the priority, we can get the Node process name throw __file__ attribute, and after that use the subprocess.check_output to retrieve the process ID, setting the priority and the scheduling policy, the following command (1) set to a NODE named "servo.py" the scheduler FIFO with priority 99:

```
chrt –f –p 99 $(pgrep servo.py)        (1)
```

For the GC problem, we will use the design pattern Decorator to dismiss the GC before running critical section: First we call gc.disable() function, after that we add gc.set_threshold(0), the second instruction will prevent any third-party libraries calling gc.enable() during execution time, and so no libraries can brought the GC back. At The end of Node execution, we want the GC to clean up the memory, so we will register "os._exit" function throw the Decorator like this: atexit.register(os._exit, 0), this will cause Copy-on-Write, including a final GC.

For memory management problem, variable declaration must be done at the initialization of the BaseNode class, and before any critical section, also using slot to declare python object can allow as control memory allocated and prevent adding additional attribute to an object later, slot is a tuple, in python we can use either slot or dictionary to store data in object, dictionary are not limited, but the tuple is allocate limited memory, so we can use __slots__ if we're going to create many instances of class and we know in advance what properties the class should have. Another option is to use global object, as they are allocated at start-up time so that the Node can set aside a fixed amount of memory.

To avoid any device IO problem, retrieving or writing data must be done outside the critical section; also, we must not fork the process to avoid the CoW behavior in Linux OS.

At last for Exception problem: the throw is typically banned in hard real-time applications. Instead, we may rely on return codes to do error handling.

Below "Fig. 1" a class diagram that illustrates the proposed architecture for mobile robot using the Node pattern. The class "BaseNode" will call the init function before calling the run function; it must contain all initialization parameters, the call of the run function will execute the desired task, at the end the finalize function will do all the stuff to write for example data to I/O or other things. The GCDecorator is the garbage collector Decorator; it will inject the strategy explained above to stop the GC in case of the real time task execution. RTNode represent the real time node, it will allow set the policy and the priority, it will execute the run function one time. The RTCyclicNode extend from the RTNode to add a cyclic execution behavior, also it register a listener to external events. The CyclicNode is the same as RTCyclicNode class but without the real time capability.

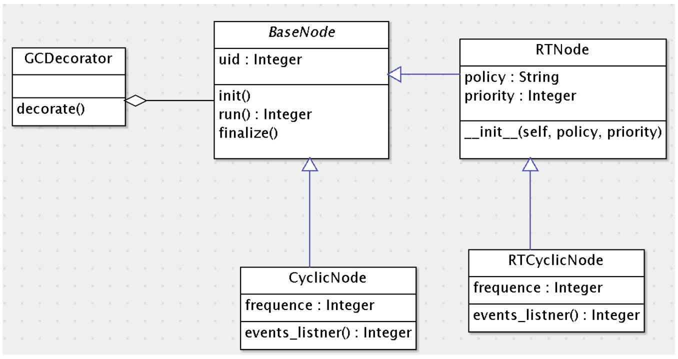
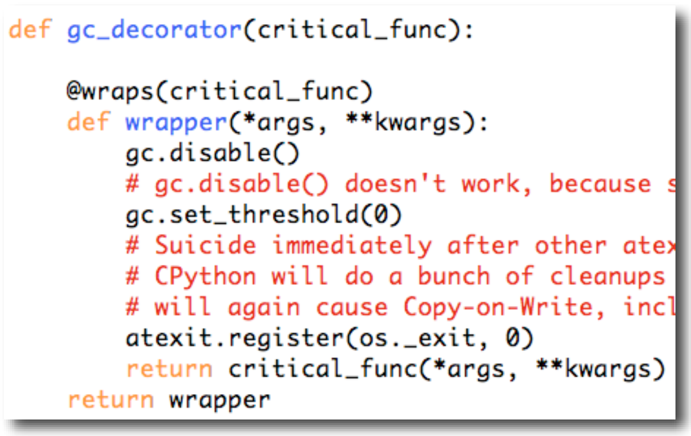


Fig. 1. Proposed mobile robot architecture base on Node pattern

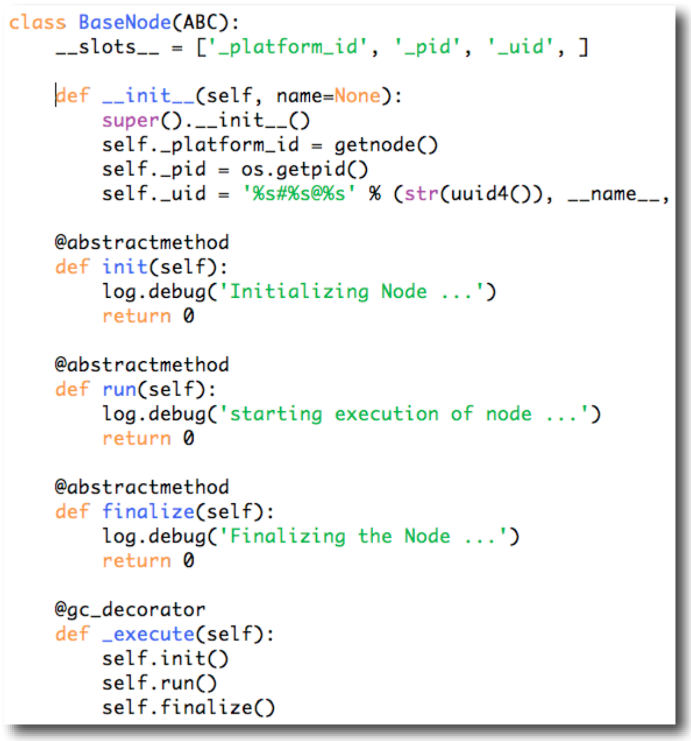
In "Fig. 2" we show a simple implementation of the GC decorator, also in "Fig. 3" we present a draft implementation of the BaseNode following the preview recommendations, the complete code can be found here [8], and also a user guide is presented here [9].



```python
def gc_decorator(critical_func):

    @wraps(critical_func)
    def wrapper(*args, **kwargs):
        gc.disable()
        # gc.disable() doesn't work, because s
        gc.set_threshold(0)
        # Suicide immediately after other atex
        # CPython will do a bunch of cleanups
        # will again cause Copy-on-Write, incl
        atexit.register(os._exit, 0)
        return critical_func(*args, **kwargs)
    return wrapper
```

Fig. 2. Garbage Collector Decorator for real time Python application



```python
class BaseNode(ABC):
    __slots__ = ['_platform_id', '_pid', '_uid', ]

    def __init__(self, name=None):
        super().__init__()
        self._platform_id = getnode()
        self._pid = os.getpid()
        self._uid = '%s#%s@%s' % (str(uuid4()), __name__,

    @abstractmethod
    def init(self):
        log.debug('Initializing Node ...')
        return 0

    @abstractmethod
    def run(self):
        log.debug('starting execution of node ...')
        return 0

    @abstractmethod
    def finalize(self):
        log.debug('Finalizing the Node ...')
        return 0

    @gc_decorator
    def _execute(self):
        self.init()
        self.run()
        self.finalize()
```

Fig. 3. Draft implementation of the BaseNode class

VI. Conclusion

A real-time computer system needs both an operating system that operates in real-time and user code that delivers deterministic execution. Both deterministic user code on a non-real-time operating system, or nondeterministic code on a real-time operating system will result in real-time performance. In this contribution we presented major problem that face developers to write a real time application, and also we proposed some best practices and design pattern to overcome this difficulty, some of them are general to any application, other are specific to Python programming language. A suitable architecture was presented to develop a mobile robot parts by using the Node pattern. This work will be reference for all developers that will contribute to COPDAI middleware development process. After the development of the core of our middleware, we plan to perform tests on mobile robots, and we hope to build a large community around this project.

References

[1] M. A. Rabbah, N. Rabbah, H. Belhadaoui, M. Rifi: “Challenges facing middlleware for mobile robots in smart environment”. International Journal of Scientific & Engineering Research, Vol 7 Issue 11 November 2016.

[2] https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages 22 October 2017.

[3] F. Jarray, H. Chniter and M. Khalgui: “New adaptive middleware for real-time embedded operating systems”. Computer and Information Science (ICIS), 2015 IEEE/ACIS 14th International Conference on. IEEE, 2015.

[4] Y. Maruyama, S. Kato and T. Azumi. “Exploring the performance of ROS2”. Proceedings of the 13th International Conference on Embedded Software. ACM, 2016.

[5] George, Damien, D. Sanchez, and T. Jorge. "Porting of MicroPython to LEON Platforms." In DASIA 2016, vol. 736. 2016.

[6] Plamauer, Sebastian, and M. Langer. "Evaluation of MicroPython as Application Layer Programming Language on CubeSats." In ARCS 2017; 30th International Conference on Architecture of Computing Systems; Proceedings of, pp. 1-9. VDE, 2017.

[7] Robertz, S. Gestegård, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov. "Using real-time Java for industrial robot control." In Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, pp. 104-110. ACM, 2007.

[8] https://github.com/mrabbah/sadasc18/blob/master/nodes.py 31 December 2017.

[9] https://github.com/mrabbah/sadasc18 31 December 2017.