PyFUNS: A Python Framework for Ubiquitous Networked Sensors

Stefano Bocchino[1], Szymon Fedor[2,⋆], and Matteo Petracca[3]

[1] Scuola Superiore Sant'Anna, Pisa, Italy
s.bocchino@sssup.it
[2] United Technologies Research Centre Ireland, Ltd. Cork, Republic Of Ireland
[3] National Inter-University Consortium for Telecommunications, Pisa, Italy

Abstract. In recent years Wireless Sensor Networks (WSNs) have been deployed in wide range of applications from the health and environment monitoring to building and industrial control. However, the pace of prevalence of WSN is slower than anticipated by the research community due to several reasons including required embedded systems expertise for developing and deploying WSNs; use of proprietary protocols; and limits in scalability and reliability. In this paper we propose PyFUNS (Python-based Framework for Ubiquitous Networked Sensors) to address these challenges. PyFUNS handles low level and networking functionalities, using the services provided by Contiki, and leaves to the user only the task of application development in the form of Python scripts. This approach reduces required expertise in embedded systems to develop WSN based applications. PyFUNS also uses 6LoWPAN and CoAP standard protocols to enable interoperability and ease of integration with other systems, pursuing the Internet of Things vision. Through a real implementation of PyFUNS in two constrained platforms we proved its feasibility in mote devices, as well as its performance in terms of control delay, energy consumption and network traffic in several network topologies. As it is possible with PyFUNS to easily compare performance of different deployments of distributed application, PyFUNS can be used to identify optimal design of distributed application.

1 Introduction

Research in Wireless Sensor Networks (WSNs) has started over a decade ago with great enthusiasm and community expectations to revolutionize our daily life. In those years WSNs have been described as "distributed systems of numerous smart sensors and actuators connecting computational capabilities to the physical world which have the potential to revolutionize a wide array of application areas by providing an unprecedented density and fidelity of instrumentation". Since the first testbeds, numerous deployments of WSNs have been described for a wide range of applications (e.g., climatic monitoring, structural monitoring

⋆ Szymon Fedor is currently affiliated with MIT Media Lab.

T. Abdelzaher et al. (Eds.): EWSN 2015, LNCS 8965, pp. 1–18, 2015.
© Springer International Publishing Switzerland 2015

of building), with the aim of introducing enhancements, and underlining open issues in the WSNs research field.

After numerous deployments in research projects, WSNs are nowadays reaching the industrial and consumer markets for large scale deployments. As matter of example it is possible to cite the GINSENG and SmartSantander projects where the potential of WSNs have been proved through real large scale deployments. Distributed smart sensors able to interact with the physical world exchanging data through wireless communications are nowadays considered the key components in the envisioned Smart City scenario.

However, to reach a wide adoption of the WSNs in several domains still several limitations persist. In this respect some of the main issues are: interoperability, ease of reprogramming and reliability. New generation of standards for WSN enables interoperability with Internet world (using IP and HTTP-type of protocols) and they need to be adopted in future smart sensors in order to reduce required effort for integration of WSN with other systems. The ease of reprogramming is a main requirement to be taken into account in large scale systems where the application logic must be changed remotely and without physical access to nodes. Network reliability is another key point to consider, in fact, this issue affects the real capability of the WSN to sense and interact with the physical world. Single point of failure must be avoided in order to prevent the possibility of losing data from several devices deployed in the field.

In respect of the above mentioned issues some progress has been made in WSN interoperability. In particular, it has been improved by adopting low level standard protocols (e.g., IEEE802.15.4), and by adapting IPv6 to the WSN scenario, thus really enabling the so called Internet of Things (IoT) vision. The IPv6 for WSN (i.e., 6LoWPAN) is only the first step towards a global interoperability, further improvements have been reached by enabling HTTP-based transactions in WSNs. CoAP is nowadays a standard protocol solution to enable the RESTful architecture in IoT-based WSNs. Progress has also been made in facilitating nodes reprogramming and programming although the proposed approaches are either not so easy, limited to a specific scope, and not really suitable for constrained devices such as those used in WSNs. In this direction a very promising and challenging approach is that following a virtual machine based design where Python scripts can be installed through RESTful transactions.

To address all the above mentioned issues we propose PyFUNS, a Python framework for ubiquitous sensor networks. By leveraging on IoT-based protocols (i.e., 6LoWPAN and CoAP) PyFUNS guarantees a higher interoperability and reliability with respect to old-style WSNs. Moreover, PyFUNS enables ease of reprogramming by introducing a virtual machine design based on Pymite, a reduced Python virtual machine for embedded systems.

The rest of the paper is structured as follows. Related works are described in Section 2, followed by the design of PyFUNS framework in Section 3. In Section 4 PyFUNS performance is presented in various network topologies and distributed application configurations. Section 5 concludes the paper.

2 Related Work

PyFUNS provides a number of features and several relevant solutions which have been described in WSN literature. We have divided them into (i) techniques for remote reprogramming, and (ii) frameworks enabling easier programming.

2.1 Techniques for Remote Reprogramming of WSNs

System Reprogramming. Such a method consists of replacing the node full firmware. It is very inefficient because even a minor application change requires reloading node binary image. Therefore they require more power and time to reprogram a node than other approaches in which only a reduced set of modules or functions is modified. Moreover, during the updating process, the new firmware must be stored in an external flash memory before being copied into the internal flash memory when the system restarts. Therefore, the nodes must have available external flash to store full software image. System level reprogramming technique are used in some existing WSN monolithic operating systems (e.g., TinyOS [HC1]) in which the whole application consists of a single image file.

Modular Reprogramming. According to this approach the node application is composed of independent, re-loadable modules. Contiki [DG1] is an example of a modular system which consists of two main components: system core and loaded program. The Contiki Core, with the boot loader exception, is a non-reprogrammable component. Therefore, any change in the code of the kernel, program loader, symbol table and communication interfaces is not supported. However, enhanced functionalities (e.g., file system support, shell support, power management) are loaded modules and are reprogrammable. The modular reprogramming is suitable for over-the-air reprogramming. Unlike the monolithic method, any system change is local, only the updated modules need to be transmitted. However, a large-memory footprint and slow system execution are disadvantages of any modular system. There are also other solutions implementing modular reprogramming (e.g., Dynamic TinyOS [MA1], LiteOS [CA1], RETOS [HS1]), similarly to Contiki their use requires embedded system experts.

Virtual Machine. In Virtual Machine (VM) based WSN, every node runs an instance of the virtual machine. The VM is used for the execution of both on-network applications and byte code instructions. In the literature there are several VM based approaches proposed for WSN [LC1][SC1]. Mate [LC1] is a VM built on TinyOS which uses the concept of capsules - a small set of high level primitives of up to 23 bytes. Mate-based applications are composed of several capsules which can propagate throughout the network to deliver an objective. Another VM for WSN is Squawk [SC1], a scale-down version of Java VM that runs without an OS on memory constrained devices. Squawk allows deployment

and execution of multiple, isolated applications on a node. The use of a VM-based approach requires sensor nodes with improved resources with respect to well-known target platforms. This is because the virtual machine could be demanding in terms of CPU and memory. Considering the general trend in providing sensor nodes with higher performance at lower costs, the VM approach can be nowadays considered an effective and powerful solution in WSNs.

Differential. The use of a differential reprogramming is mainly based on the use of code patches: a patch is generated using the difference between the old and the updated program. Rsync [TM1] is a differential update scheme, and its functionalities has been demonstrated in WSNs [JC1]. As working principle, Rsync divides the program into different blocks and calculates their hash values. The evaluated hash values are then matched to determine the block insertion, deletion, or modification. There are many other examples of differential reprogramming systems [KP1][RL1], and in general it has been shown that the size of the deltas produced by the differential-based approaches is very small compared to the full binary image. However, most of them poorly perform when there is a change of both program and variable layout. This is because such update requires full flash memory writing, and large amount of additional external flash memory. Differential solutions can be easily used only by embedded system experts.

2.2 Frameworks Enabling Easier Programming of WSN

Many solutions for enabling an easier WSN programming have been described in the literature [MP1]. They were designed with different objectives, including energy-efficiency, scalability, failure-resilience or collaborative data processing. In this respect it must be underlined that one of PyFUNS main goals is to reduce required expertise in embedded systems for programming WSNs, as this has been previously identified by domain experts [MD1] as one of the major barriers for deploying WSNs. In that study the authors implemented the BASIC programming language for sensor networks and conducted a user study with novice programmers. Half of users with no previous programming experience of any kind were able to program simple network tasks using developed BASIC programs while only 0-17% could do so in TinyScript. Therefore the authors concluded that current WSN languages require knowledge of either very low-level systems development (including the details of sensor hardware and embedded system design), or high-level programming concepts and abstractions that are not obvious to most application domain experts. And because application domain experts have little programming experience, most of which is with simple single-threaded imperative programming models, the authors have ported a small BASIC interpreter to a WSN platform. Authors motivations are coherent with ours although our solution provides more features (e.g., interoperability due to IP and CoAP protocols) and is based on Python interpreter.

Recently several publications [AP1][C1] described solutions to program WSNs in Python language, due to its popularity and ease-of-use. In fact, according to [P1], Python requires no more than half as much time as writing in C, and it appears to be more intuitive with respect to C for new students [F1]. Regarding previously cited Python-based solutions, they must be considered at the early stage of development and incomplete to be used nowadays in real applications, though the most promising in this respect is T-Res. In fact, T-Res enables programming of the node to execute simple data-processing tasks performing the following actions: (i) monitoring one or more resources, (ii) executing some processing on their values, and (iii) sending the resulting output to other resources. The main lack of T-Res is in the possibility of monitor resources only: a method to retrieve the current resource state by using Python scripts is not supported.

3 PyFUNS Design

Having identified the limitations of literature of systems aiming at enabling remote reprogramming and an easier programming in WSNs, we have designed PyFUNS, a framework that can be used in a easy way to reprogram WSNs. Our framework leaves to the user only the application development task in the form of Python scripts, while abstracting low level and networking functionalities.

3.1 Dynamic Services over WSN

Traditional WSNs enable the development and deployment of pervasive networks aiming at providing many simple services, such as the environmental monitoring or the basic actuation control through basic operations. With the introduction of the IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) protocol and Constrained Application Protocol (CoAP), following the IoT vision, WSNs have acquired enough resources to accomplish more complex services, such as the capability of exposing equipped sensors in Internet to perform automatic control operations. The next natural step in the WSNs domain is to build a smart management of dynamic services, thus enabling the possibility of remotely reprogramming the services provided by an IoT-based WSN.

In general terms a service provided by a WSN is a set of operations to be performed to accomplish a specific task. For instance, a service can be the automatic light control in a room and the operations to be performed are: (i) check the light value periodically, (ii) check the presence of people in the room, (iii) switch on the lamp while setting the power according to the desired light value, and (iv) switch off the lamp when people leave the room.

As previously stated, PyFUNS enables the management of dynamic services in WSNs. In the rest of paper we follow the aforementioned definition of service (i.e., a set of operations) calling the operations to be performed applications.

3.2 Application Components

An application deployed on a sensor node has several components:

- *Name:* string of characters that uniquely identifies the application;
- *Period:* it is related to the periodicity of the application execution. Values bigger than zero mean periodicity, equal to zero is for one time executions, while less than zero mean application blocked waiting for an answer. Application flow changes based on the Period value;
- *Timer:* used for periodic applications, it fires when executing the application;
- *State:* indicates the current state of the application in its internal Finite State Machine (FSM);
- *Script:* it contains the Python byte code performing the specific task which the application has to provide;
- *Variables:* list of variables required to store data to be exchanged among different scripts of the same application or among different applications;
- *Requests:* list of active requests. A request is used to retrieve the current representation of a resource through network messages. Each request is associated to both a callback function, called when a reply is received, and a variable, which is used to store the received data.

To the end of building an abstract framework that allows to implement applications able to perform data communication through the network (e.g., request/reply paradigm), we decomposed the application in three sub-scripts: *PreScript* (optional), *MainScript* (mandatory) and *PostScript* (optional). *PreScript* allows to send data request messages to a specific node in the network, and the answer will be processed in the *MainScript*. Moreover, it allows to set up the application environment (e.g., to create the variables required), and to retrieve the resource representation. *PostScript* is executed when the application has been stopped, and is mainly used to clean the application environment (e.g., to delete active requests). *PreScript* runs once at the application start, whereas *PostScript* runs once at the application stop. *MainScript* is the only mandatory byte code to be installed on the nodes, and represents the application core. It can be run once or several times according to the Period value. The *MainScript* execution can be triggered by a periodic event, the expiration of a timer, or by a sporadic event, the reception of a message. Fig. 1.a illustrates the script flow for an application using all the three described scripts.

3.3 Application Life-Cycle

The FSM model has been used to implement the application life-cycle, that can be dynamically installed, started, stopped, updated and uninstalled. To enable the aforementioned operations, five different states have been defined: (i) NEW, all the memory required to store the application structure has been allocated successfully; (ii) INSTALLED, scripts have been installed on the node; (iii) RESOLVED, application is ready to execute; (iv) RUNNING, application is active

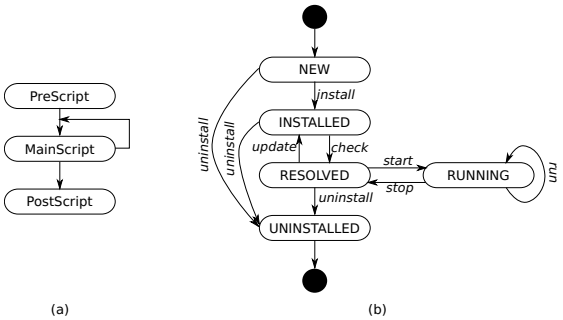

Fig. 1. (a) Scripts flow chart. (b) Application finite state machine.

and performs its operations; and (v) UNINSTALLED, the application structure has been deleted and the memory has been released. Fig. 1.b depicts the application life-cycle and the possible transitions among states.

The application life starts in the NEW state, in which the necessary memory is allocated to store the components described in Section 3.2. All the components are set to a default value, except for the name which is filled when the application is created. In the NEW state it is possible to install *PreScript*, *MainScript* and *PostScript* on the node. As previously stated *MainScript* is mandatory for each application and installing it implies a change of state to INSTALLED. In the NEW state it has been enabled the possibility to uninstall the application through a defined uninstall event. In the INSTALLED state all necessary components for the application are set, even though they are still waiting for a control check aiming at verifying the compatibility among scripts (e.g., check scripts version). The check is triggered by a defined check event, and in case all the tests are passed, the state changes to RESOLVED. Also in the INSTALLED state it is possible to trigger an uninstall event to delete the application. Once the application reaches the RESOLVED state it has been successfully checked and it is ready to be executed. Three different events can be triggered from this state: (i) start, to run the application, *PreScript* is executed in case it is present, otherwise *MainScript* is interpreted, as result the state moves to RUNNING; (ii) update, to perform any changes concerning the scripts (e.g., install, update or delete scripts on the node), in this case the state moves to INSTALLED and the check compatibility on the new installed scripts must be redone; and (iii) uninstall, to remove the whole application and release the memory used by the application, next triggered state is UNINSTALLED. In RUNNING state the application can be executed one or many times according to the Period, and can be stopped through a dedicated stop event. *PostScript*, if present, is executed during the transition from RUNNING to RESOLVED. Last state is UNINSTALLED where the application is deleted from the node. Table 1 summarizes the state transitions of the above described FSM.

Table 1. Application state transition

CurrentState	*Input*	*fNextState*	*Output*
NEW	install	INSTALLED	At least *MainScript* has been installed
	uninstall	UNINSTALLED	Application deleted
INSTALLED	check	RESOLVED	Application ready to execute
	uninstall	UNINSTALLED	Application deleted
RESOLVED	start	RUNNING	*PreScript* executed, if installed
	update	INSTALLED	Changes in installed scripts
	uninstall	UNINSTALLED	Application deleted
RUNNING	run	RUNNING	None
	stop	RESOLVED	*PostScript* executed, if installed

3.4 Application Flow

As mentioned in Section 3.2, the application flow, in particular when *MainScript* is executed, depends on the value of the application period. Two different period categories have been defined: period equal to zero when *MainScript* runs one time, and period not equal to zero when *MainScript* can run zero, one or many times. Fig. 2 shows different flow chart depending on the value of period, Fig. 2.a is for the first category, while Fig. 2.b and Fig. 2.c for the second.

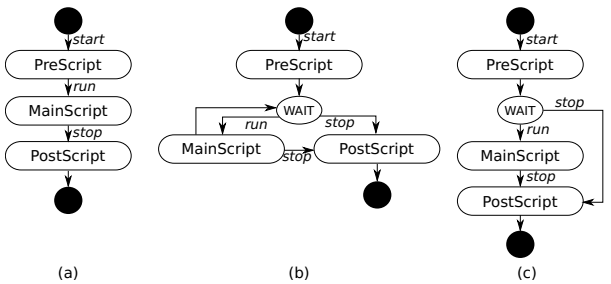

Fig. 2. Script flow chart for period equal to zero (a), period not equal to zero (b) and period less than zero, particular implementation (c)

In case of period equal to zero (Fig. 2.a), the application goes from *PreScript* to *PostScript* directly, running *MainScript* one time. It is not possible to stop the application once it is started. This setting of period is useful for applications changing the resource representation only one time.

With period not equal to zero (Fig. 2.b), after *PreScript*, the application waits for an event to continue its execution. We have defined two types of events that trigger *MainScript*: periodic and sporadic. Periodic applications have a period greater than zero and they wait the timer expiration before to interpret *MainScript*. This setting is useful to implement applications changing the resource representation periodically. For sporadic applications the period is less than zero and *MainScript* is called when a sporadic event happens (e.g., message received).

This type of setting is useful to implement applications that perform activities when observed resources change. A particular application flow based on a period less than zero has been implemented, Fig. 2.c, to provide applications able to run *MainScript* once after resources representation are retrieved.

3.5 Application RESTful Interface

The goal of PyFUNS is to enable easy management (in terms of parameter reconfiguration and code deployment) of dynamic application installed in ubiquitous WSNs. To reach a seamless integration of the framework in motes it is necessary to abstract the application and its attributes. This can be done by using the REST paradigm in the context of IoT-based WSNs, or in other words by using the CoAP protocol, thus allowing sensor nodes to abstract resources and run embedded web services. Abstracting application and its attributes as CoAP resources enables the use of well known HTTP methods, GET, PUT, POST and DELETE, to administer code installed in a WSN. Moreover management of the application, (e.g., start or stop) can be performed by a user through a web site, or by another application through simple CoAP messages.

As described in Section 3.2, an application is defined by its components which are managed in PyFUNS as sub-resources of `/apps`. The resulting application structure is shown in Table 2. Resource `/apps` is created statically during the start up phase. This resource is the container of all applications installed on the node and it can be managed through CoAP methods to list currently installed applications and check their validity. The methods of `/[app_name]` provide the services to create/delete a specific application, retrieve the current state of the application, and start/stop its execution. The `/[app_name]` resource and its sub-resources are created by allocating the required memory only once, when the application is installed. The use of CoAP methods to manage the execution of a specific application (start/stop) enables the possibility to install on a node several applications related to each other in order to implement complex services.

Resource `/period` represents the current application period value, and must be set following the rules described in Section 3.4. A set of methods are provided

Table 2. The structure of an application resource

/apps			# list currently installed apps [GET]
			# check a specific app [POST]
/[app_name]			# retrieve the application state [GET]
			# create/delete a specific app [PUT|DELETE]
			# start/stop a specific app [POST]
	/period		# retrieve/update the period [GET|PUT]
	/preScript		# retrieve/update/delete the *PreScript* [GET|PUT|DELETE]
		/version	# retrieve/update the *PreScript* version [GET|PUT]
	/mainScript		# retrieve/update/delete the *MainScript* [GET|PUT|DELETE]
		/version	# retrieve/update the *MainScript* version [GET|PUT]
	/postScript		# retrieve/update/delete the *PostScript* [GET|PUT|DELETE]
		/version	# retrieve/update the *PostScript* version [GET|PUT]
	/variables		# list currently variables [GET]
		/[var_name]	# retrieve/observe/update the value [GET|PUT]

to manage the scripts: for each script, *PreScript*, *MainScript* and *PostScript*, it is possible to retrieve/update/delete the byte code and retrieve/update the version of them. `/variables` resource is the container of the variables used by the application to accomplish its functionalities. By interacting with it, the list of current variables can be retrieved. For each variable a new resource is created and it is possible to retrieve/update the value. The purpose of this resource is to exchange data among different scripts of the same application, or among different applications. Each `/[var_name]` resource can be observed, even by other applications, enabling a smart functionality to be used in complex systems.

3.6 PyFUNS Implementation

Native code replacement and loadable modules on the one hand enable services updates, on the other hand imply a higher cost since downloaded modules are more coarse-grained compared to a virtual machine application. Moreover, these methods require to maintain information about the software version in each node, and the implementation is hardware dependent. To fully decouple applications from the sensing infrastructure we use a virtual machine to run the applications.

Most of the virtual machine based approaches enable highly efficient updates: low cost for transmitting new code and abstraction from the platform. The software updates sent from front-end-device to different nodes (based on different platform) are always the same. However, VMs introduce overhead in term of memory and computational overhead, which is overcome by more powerful devices present on the market. Python, Java and JavaScript are the most common interpreted languages used for virtual machine approaches with substantial libraries of pre-written code. The last two are object-oriented languages; whereas Python supports multiple programming paradigms, including object-oriented, imperative and functional programming styles. JavaScript script is too big to be installed in a WSN node and it cannot be compiled into byte code. Using byte code for reprogramming leads to an extremely powerful system in which microcontrollers can be programmed interactively without the typical compile/link/flash/run cycle. Both Python and Java allow for platform-independent processing functions that can be freely exchanged among nodes, but we preferred the former approach because, as discussed in Section 2.1, programming in Python is really simple and supports multiple programming styles.

We implemented PyFUNS on top of Contiki OS [DG1] that provides native support for 6LoWPAN and CoAP. A Python interpreter has been ported to the target operating system to enable script interpretation on constrained devices. We ported PyMite [PM1], a reduced Python interpreter that runs a significant subset of the Python language on microcontrollers with very few resources.

PyFUNS provides a set of APIs, summarized in Tab. 3, that can be used in Python scripts to implement applications. Such APIs allow: (i) to manage variables (create/delete/get/set); (ii) to send a generic CoAP message specifying the method (GET, POST, PUT, DELETE), the node address, the URI of target resource, the eventually payload and the eventually variable where store the result of the operation; (iii) to set/unset observation to a specific resource

defined by its IPv6 address and URI; and finally (iv) to stop the execution of the application. The IPv6 address parameters are expressed without the prefix (e.g., [0,0,0,2]), as we have provided the messages exchanged among different applications that can be performed only inside the same network. Notice that sendMsg and obs functions have a parameter var to be associated with the request. In case of var is not present, it is automatically created inside the functions.

Table 3. PyFUNS APIs

Function	*Description*
`newVar(name, value)`	Create new variable
`delVar(name)`	Delete variable
`getVar(name)`	Get variable value
`setVar(name)`	Set variable value
`sendMsg(met, addr, uri, payload, var)`	Send CoAP message
`obs(addr, uri, var)`	Send CoAP observe
`delObs(addr, uri)`	Delete CoAP observe
`exit()`	Stop the application

A prerequisite of PyFUNS is that each node runs a web service to expose its resources, since the framework uses CoAP methods to interact with them. Instead, PyFUNS framework can be installed only on a subset of nodes.

3.7 Example of Usage

To evaluate PyFUNS performance, we implemented a Security service application which has the purpose to detect any motion in a room and trigger an alarm. In such example the network is composed of three PIR sensors, on nodes 2, 3 and 4 with the URI `coap://[aaaa::2]/sen/pir`, `coap://[aaaa::3] /sen/pir` and `coap://[aaaa::4]/sen/pir` respectively, and one alarm, on node 5 with URI `coap://[aaaa::5]/act/alarm`. The application implementing the service can be installed in any node inside the network using the RESTful interface defined in Section 3.5. The intent of Security service is to observe the PIR sensors, and trigger the alarm whenever a notification of motion detection is received. To implement such envisioned application we need to write and install the *PreScript*, *MainScript* and *PostScript*. *PreScript*, Listing 1.1, issues OBSERVE messages to all three PIR sensors and associates the requests to variables, p1, p2 and p3, used to maintain the representation of the sensors. Since the *MainScript* runs whenever a notification is received, the period of the application is set with a number less than zero: execute *MainScript* after a sporadic event happens (Fig. 2.c).

Listing 1.1. The *PreScript* of Security application

```
from pyfuns import *
obs([0,0,0,2], "sen/pir", "p1")
obs([0,0,0,3], "sen/pir", "p2")
obs([0,0,0,4], "sen/pir", "p3")
```

MainScript(Listing 1.2) is called whenever a notification from observed sensors is received. The operations carried out are very simple: retrieve the representation of the variable associated to each PIR sensors and issue a POST request to `coap://[aaaa::5]/act/alarm` to trigger the alarm, if one of the variables is equal to one, or to stop the alarm otherwise. Listing 1.3 shows the Python script related to *PostScript*. It sends messages to the PIR resources in order to delete the subscription when the application has stopped. The scrips byte code to be installed on nodes can be obtained by compiling the presented Python scripts.

Listing 1.2. The *MainScript* of Security application

```
from pyfuns import *
if getVar("p1") or getVar("p2") or getVar("p3"):
  sendMsg(2, [0,0,0,5], "act/alarm", "1")
else:
  sendMsg(2, [0,0,0,5], "act/alarm", "0")
```

Listing 1.3. The *PostScript* of Security application

```
from pyfuns import *
delObs([0,0,0,2], "sen/pir")
delObs([0,0,0,3], "sen/pir")
delObs([0,0,0,4], "sen/pir")
```

4 Performance Evaluation

To evaluate PyFUNS performance we implemented it on top of Contiki OS by integrating/porting PyMite on two constrained platforms: (i) WiSMote, equipped with a MSP430F5 microcontroller having 16 kB of RAM and 256 kB of flash, and (ii) CC2538dk, equipped with an ARM Cortex^{TM} M3 microcontroller having 32 kB of RAM and 512 kB of flash. In the rest of the section we first prove the feasibility of PyFUNS by checking that in both selected target platforms the performed implementation requires flash memory and RAM which are within the physical limits. Then we evaluate PyFUNS overhead in terms of run time and energy consumption. Finally we present an extensive evaluation of PyFUNS framework by implementing one real service: Security. To deploy the system bases on real platform, and test it in a real life scenario, we integrated: (i) sensors, such as PIRs, and (ii) actuators, such as alarms, on target platforms.

4.1 Flash and RAM Requirements

To assess the possibility of deplying PyFUNS on the selected devices we measured both the flash and RAM occupation. Table 4 shows the memory occupied by the software for both platforms, the WiSMote and the CC2538dk. The software installed on each WSN node includes the Contiki OS, the PyMite interpreter, PyFUNS, plus the possibly required memory to install two PyFUNS applications. In case of WiSMote platform the whole firmware occupies 93% of the available RAM and 38% of the available flash. In case of the CC2538dk platform the firmware requires the 62% of the available RAM and the 19% of

Table 4. Code size and RAM requirements for a WiSMote and CC2538dk devices

Nodetype	*RAM*[*bytes*]	*Flash*[*bytes*]
WiSMote	14 918 (93%)	98 077 (38%)
CC2538dk	19 904 (62%)	96 732 (19%)

the available flash. Such a notable occupation of memory, especially RAM, is mainly due to PyMite, which alone requires 45 kB of flash and 8 kB of RAM. In order to reduce the RAM occupation we are planning to implement a tool to store Python byte codes into the flash. The current version of PyFUNS stores the Python scripts in RAM, which is usually more constrained comparing to the flash memory.

4.2 Native Code versus Python Script

PyFUNS overhead in terms of run time and energy consumption has been evaluated with respect to a native code solution. Both performance figures have been measured by using two different set of benchmarks: (i) five test applications implementing algorithms showing a different complexity level; (ii) three applications implementing CoAP methods. Each benchmark has been executed by considering a C language based native code solution, and its Python version.

The first benchmark set is composed of five algorithms, characterized by different complexity levels, and chosen from *"dada's perl lab"*[1]. More specifically, we selected the following algorithms, listed in function of their complexity (from lower to higher): (i) ACK - Ackermann's Function(3, N) that is a classic recursive function with N=3; (ii) FIB - Fibonacci Numbers(N) that computes the Fibonacci sequence with N=17; (iii) MAT - Matrix Multiplication(N) that performs the multiplication between two matrices with size 5 and N=10; (iv) HEAP - Heapsort(N) that sorts a vector with a size N=100 of integer numbers, and initialized with strictly decreasing value; and (v) MET - Method Calls(N) that implements activation of class methods using object-oriented style. The second benchmark test, instead, includes: (i) an application that issues a POST request to a resource installed in a neighbor node (POST); (ii) an application that issues a POST request to a resource installed in a neighbor node and waits the acknowledgement message from the resource (POST2); and (iii) an application that issues a GET request to one resource installed in a neighbor node, waits the reply, processes it and sends a POST request to another resource installed in a neighbor node (GET). All performance results are reported in Table 5.

All results have been obtained by running each test 1000 times in Cooja, the Contiki network simulator. Cooja allows to run the same binary files to be used on real platforms while enabling a quick testing and debugging of the system. In the simulator all tests have been performed by using only the WiSMote platform

[1] A benchmark comparison of a number of programming languages: `http://dada.perl.it/shootout/craps.html`

Table 5. Performance benchmarks in Cooja

	C		*Python*		*Python/C*	
	Time(ms)	*Energy(μJ)*	*Time(ms)*	*Energy(μJ)*	*Timeratio*	*Energyratio*
ACK	4.08	0.029	645.25	4.765	158.1	164.3
FIB	9.95	0.072	1344.84	9.932	135.2	137.9
MAT	5.06	0.037	687.31	5.076	135.8	137.1
HEAP	1.95	0.014	379.68	2.804	194.7	197.7
MET	1.16	0.009	207.28	1.531	178.8	177.2
POST	1.22	0.009	5.35	0.039	4.4	4.3
POST2	8.61	0.328	12.68	0.357	1.4	1.1
GET	17.26	0.604	26.19	0.671	1.5	1.1

(CC2538dk is not supported at time of writing), moreover to prove the Cooja accuracy we ran also two benchmark tests on a real WiSMote platform. In Table 5 the C and Python columns show the run times and the energy consumption for all benchmark applications, while the last column labeled as Python/C reports the ratio between PyFUNS and native code approaches. For the first benchmark set the time performance penalty of PyMite is between 135 and 195, while showing a performance gap between 137 and 198 in energy consumption. Such a difference between C and Python is mainly caused by the extensive use of the heap memory in PyMite when performing complex operations such as recursive calls. On the contrary, in CoAP methods tests the run time performance penalty is between 1.5 and 4.4 with an energy consumption performance gap between 1.1 and 4.3. This is the overhead introduced by PyMite to perform CoAP methods in WSNs, while enabling a powerful tool providing platform abstraction and reconfigurable in-network processing that can compensate the overhead. To prove the validity of the aforementioned results obtained with Cooja simulator, we also ran the Python version of Ackermann's Function and POST method on a real WiSMote platform. The obtained results are reported in Table 6, and they are very similar to those obtained by using the Cooja simulator.

Table 6. Performance benchmark on WiSMote

	Time(ms)	*Energy(μJ)*
ACK	649.79	4.799
POST	5.52	0.040

4.3 Real Case Evaluation

Performance of a distributed application depends on the network topology and in-network distribution of application components. We evaluated PyFUNS performance in terms of energy consumption, actuation delay and network traffic, to provide real services such as the one presented in Section 3.7. The application components were distributed among the nodes or centrally placed at the border router. For the energy consumption we considered the overall network consumption. The actuation delay represents the elapsed time between the detection of

the event and the associated actuation, while network traffic measures the total amount of bytes exchanged in the network. As we want to evaluate the impact of PyFUNSonly, we take into account only CoAP messages without counting traffic generated by underlying layers (e.g. RPL messages).

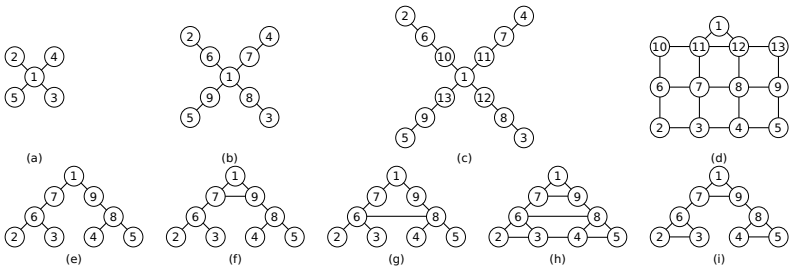

Fig. 3. Network topologies: star (a-b-c), mesh (d) and tree (e-f-g-h-i)

To avoid impact of the changing environment and measurement overhead of real world experiments we installed PyFUNS on Cooja simulator. The Security service was deployed on multi-hop, IoT-based WSN, configured with nine network topologies shown in Fig. 3: three star topologies with 5, 9, and 13 nodes; one mesh topology with 13 nodes; and 5 tree topologies each one of them with 9 nodes and different transmission links. The power transmission of nodes was fixed for all the topologies except for the topology from Fig. 3.a which was evaluated also with a higher transmission power. This was done to compare topology having multi-hop transmissions (Fig. 3.b) with a network having smaller number of nodes but covering similar geographical area.

For the security service scenario nodes 2, 3 and 4 in Fig. 3 were simulated with an attached PIR sensor and the node 5 with an attached buzzer. We tested different placements of security service components as depicted in Table 7.

Table 7. Security Control service deployment configurations

(a)	(b)	$(c)(d)$	$(e)(f)(g)(h)(i)$
BR (1)	BR (1)	BR (1)	BR (1)
PIR2 (2)	PIR2 (2)	PIR2 (2)	PIR2 (2)
Alarm (5)	Alarm (5)	Alarm (5)	PIR4 (4)
	Node 6	Node 6	Alarm (5)
	Node 9	Node 9	Node 6
		Node 10	Node 7
		Node 13	Node 8
			Node 9

Figure 4 shows the energy consumption measurement for all topologies. As we expected the minimum energy consumption for star topology is when PyFUNS application is installed on the Border Router. This is because the amount

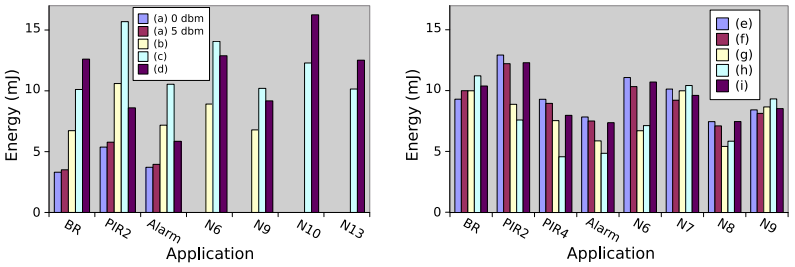

Fig. 4. Energy consumption in star (a-b-c), mesh (d) and in tree (e-f-g-h-i) topologies. The label on the x-axis indicates in which node the Security application is installed.

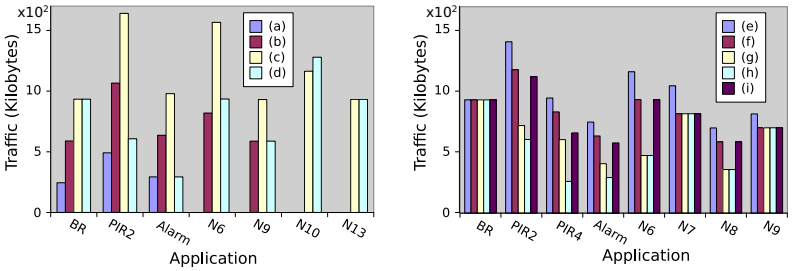

Fig. 5. Network traffic in star (a-b-c), mesh (d) and in tree (e-f-g-h-i) topologies. The label on the x-axis indicates in which node the Security application is installed.

of data exchanged in such configuration is minimum (Fig. 5). In fact, when the transmission is between nodes distant by more than one hop an additional 6LoWPAN header overhead (due to the addressing and hop limit fields) is observed.

However, in case of mesh topology (Fig. 4 right side) the minimum energy consumption of the overall network is observed when the service is distributed among the nodes rather than placed on the Border Router. For all topologies it is the number of transmission hops that plays dominant role in the total amount of network traffic, and consequently in energy consumption. For instance, in topology (e) (purple columns) it is possible to see that the energy consumed when the application is installed in nodes PIR2 and 6 (which are closely located) is bigger than a centralized approach (application on BR). On the basis of energy consumption parameter the best choice for (e) (f) (g) is node 8, with a consumed energy equal to 7.47 mJ, 7.12 mJ and 5.43 mJ respectively, for (h) is node 4 with 5.36 mJ, and for (d) and (i) is node 5 with 5.85 mJ and 7.39 mJ respectively.

We also evaluated delay introduced by the framework in triggering the actuator node when a motion detection event happens. Figure 6 presents the delay for all topologies, it depends on the number of hops between sensor and actuator.

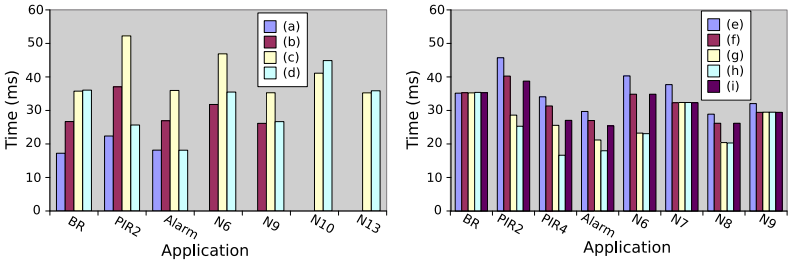

Fig. 6. Delay time (ms) in star (a-b-c), mesh (d) and in tree (e-f-g-h-i) topologies. The label on the x-axis indicates in which node the Security application is installed.

5 Conclusions

As WSNs moved from the academic world to the industrial scenario new challenges have been raised up to reach a wide adoption of the WSNs in several domains. Some of the main issues are: interoperability, ease of reprogramming and reliability. To address such issues we propose PyFUNS, a Python framework for ubiquitous sensor networks. By leveraging on IoT-based protocols (i.e., 6LoWPAN and CoAP) PyFUNS guarantees a higher interoperability and reliability with respect to old-style WSNs. Moreover, thanks to its adopted virtual machine design based on Pymite, a reduced Python interpreter, PyFUNS enables ease of reprogramming in WSNs. In a real scenario PyFUNS can be used as complementary tool of a framework able to allow users to easily write Python-based IoT applications (e.g., through a graphical interface) to be remotely installed on WSN nodes hiding the whole installation process. This feature can be provided by PyoT, a system for macro-programming and managing IoT-based WSNs.

In the paper we first presented PyFUNS by detailing its design and implementation choices by carefully explaining its usage in building simple and complex services. Then we evaluated PyFUNS performance considering the WiSMote and CC2538dk platforms with the aim of proving its feasibility in real constrained devices, and its overhead in terms of run time and energy consumption with respect to native code solutions. Finally PyFUNS performance in star, mesh and tree network topologies were evaluated for a Security service by considering both centralized and distributed application logic solutions. Presented results, aside of proving PyFUNS feasibility and performance, highlight further possible optimization to be investigated: RAM memory requirement reduction, scripts execution time and energy consumption, communication failures handling. While RAM memory occupancy can be merely solved by saving Python scripts in flash and leaving the RAM for regular applications, other optimizations require a deeper analysis, and they will be addressed in future investigations.

References

[AP1] Alessandrelli, D., Petracca, M., Pagano, P.: T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs. In: IEEE International Conference on Distributed Computing in Sensor Systems, pp. 337–344 (2013)

[C1] Carboni, D., Crs Parco Tecnologico Pula: PySense: Python Decorators for Wireless Sensor Macroprogramming. In: ICSOFT, pp.165–169 (2010)

[CA1] Cao, Q., Abdelzaher, T., Stankovic, J., He, T.: The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In: International Conference on Information Processing in Sensor Networks, pp. 233–244 (2008)

[DG1] Dunkels, A., Gronvall, B., Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: IEEE International Conference on Local Computer Networks, pp. 148–157 (2004)

[F1] Fangohr, H.: A comparison of C, Matlab and Python as teaching languages in engineering. In: International Conference in Computational Science, pp. 1210–1217 (2004)

[HC1] Hui, J.W., Culler, D.: The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In: International Conference on Embedded Networked Sensor Systems, pp. 81–84 (2004)

[HS1] Hojung, C., Sukwon, C., Inuk, J., Hyoseung, K., Hyojeong, S., Jaehyun, Y., Chanmin, Y.: RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks. In: International Symposium on Information Processing in Sensor Networks, pp. 148–157 (2007)

[JC1] Jaein, J., Culler, D.: Incremental network programming for wireless sensors. In: IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, pp. 25–33 (2004)

[KP1] Koshy, J., Pandey, R.: Remote incremental linking for energy-efficient reprogramming of sensor networks. In: European Workshop on Wireless Sensor Networks, pp. 354–365 (2005)

[LC1] Levis, P., Culler, D.: Mate: a tiny virtual machine for sensor networks. In: International Conference on Architectural Support for Programming Languages and Operating System, pp. 85–95 (2002)

[MA1] Munawar, W., Alizai, M.H., Landsiedel, O., Wehrle, K.: Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In: IEEE International Conference on Communications (ICC), pp. 1–6 (2010)

[MD1] Miller, J.S., Dinda, P.A., Dick, R.P.: Evaluating a BASIC Approach to Sensor Network Node Programming. In: ACM Conference on Embedded Networked Sensor System, pp. 155–168 (2009)

[MP1] Mottola, L., Picco, G.P.: Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. ACM Comput. Surv. 43, 19:1–19:51 (2011)

[P1] Prechelt, L.: An empirical comparison of seven programming languages. Computer 33, 23–29 (2000)

[PM1] PyMite (2013), `http://code.google.com/p/python-on-a-chip/`

[RL1] Reijers, N., Langendoen, K.: Efficient Code Distribution in Wireless Sensor Networks. In: ACM International Conference on Wireless Sensor Networks and Applications, pp. 60–67 (2003)

[SC1] Simon, D., Cifuentes, C., Cleal, D., Daniels, J., White, D.: Java on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In: International Conference on Virtual Execution Environments, pp. 78–88 (2006)

[TM1] Tridgell, A., Mackeras, P.: The rsync algorithm (1998), `http://rsync.samba.org/tech_report155-190`