T-Res: enabling reconfigurable in-network processing in IoT-based WSNs

Daniele Alessandrelli*†, Matteo Petracca†*, and Paolo Pagano†*
*National Laboratory of Photonic Networks, Consorzio Nazionale Interuniversitario per le Telecomunicazioni, Pisa, Italy
*Real-Time Systems Laboratory, Scuola Superiore Sant'Anna, Pisa, Italy
d.alessandrelli@sssup.it, matteo.petracca@cnit.it, paolo.pagano@cnit.it

***Abstract*—Internet of Things (IoT) standards are a key technology for enabling multi-purpose Wireless Sensor Networks (WSNs). Indeed, by providing a common interface for interacting with sensor nodes, they help decouple the sensing infrastructure from the applications running on top of it. To this end, much work has been done to define the semantics for controlling sensors and actuators. On the contrary, a solution for manipulating the in-network processing performed by an IoT-based WSN is still undefined. As a result, IoT applications usually run "out-of-the-network", e.g., in high-end application servers or in the Cloud. This paper addresses such a limitation by proposing T-Res, a new programming abstraction for IoT-based WSNs. The proposed solution allows using IoT protocols for changing the data-processing performed by sensor nodes and the interaction among them, thus enabling "in-network" applications. To prove the feasibility of T-Res, a real implementation based on Contiki OS is presented and evaluated. Evaluation results show that the use of T-Res, compared with the traditional "out-of-network" approach, lowers the risk of network congestion, reduces the power consumption, and can improve application response time.**

I. Introduction

Wireless Sensors Networks (WSNs) allow gaining new knowledge about the physical world in which they are deployed in order to increase the human control over it. A WSN is a distributed system composed of tiny autonomous devices, usually identified in literature as motes. A mote is a battery powered embedded device equipped with a microcrontroller, a little amount of memory, a radio interface and one or more sensors. Moreover, latest motes can also be attached to actuators for remote control applications. WSN solutions have been developed during the past years with the aim of accomplishing a specific task, thus developing specific software solutions and mote devices. Although specific and tailored WSN solutions can reach remarkable performance levels, multi-purpose WSNs open new opportunities for efficient resource utilization and adaptation to changing system requirements [1].

The emerging Internet of Things (IoT) standards are the enabling technology for the effective creation of multi-purpose WSNs. By abandoning proprietary protocols in favor of interoperable Internet standards, IoT-based WSNs can be easily integrated with other systems, and the sensing infrastructure can be decoupled from the running applications. Hui and Culler [2] have been the first to propose a complete IPv6-based network architecture for WSNs and to validate it with a production-quality implementation. Their work proved that IPv6-based WSNs can compete in terms of latency, data-reception rate, and network life, with traditional WSNs employing proprietary protocols. However, the use of standard network protocols has been just the first step toward a multi-purpose IoT-based WSN implementation. Indeed, to support a wide variety of concurrent applications (not necessarily known at network design time) a common application layer was still missing. Shelby [3] addressed this issue by proposing to extend the successful web architecture, based on the REST paradigm (see Section II), to the sensor network domain. Specifically, he suggested to use an HTTP-like protocol, called CoAP (Constrained Application Protocol) [4], for building embedded web services running on sensor nodes. Such web services provide applications with a common interface for manipulating sensor-node resources (i.e., sensors and actuators).

However, to achieve real multipurpose IoT-based WSNs, one further step is necessary: defining a common interface for reconfiguring the interactions among IoT-nodes and the application logic running on them. The current dominant approach to this issue is that described by Kovatsch et al. [5], who suggest to move the application logic from the firmware to the Cloud. In the proposed architecture, the application logic runs only on back-end servers, whereas IoT nodes simply provide access to their sensors and actuators through a CoAP server. While such an approach allows for easy application development, it sacrifices direct communication among IoT nodes and makes applications rely on third entities (e.g., the back-end servers, the border router, etc.), thus reducing the robustness of applications and increasing the risk of network congestion. The last point is especially noteworthy, indeed, as discussed in the paper presenting Actinium [6], an implementation of the previously described architecture, the problem of network congestion already shows up in small WSNs where all nodes are just one-hop away from the border router.

In this paper, we addresses the limitations of the current IoT approach by proposing a novel programming abstraction called T-Res, the Task Resource Abstraction. IoT-based WSNs equipped with T-Res allow for IoT applications that are decoupled from the network infrastructure, but still capable

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 317671 (ICSI – Intelligent Cooperative Sensing for Improved traffic efficiency).

DOI: 10.1109/DCOSS.2013.75

of in-network processing. We stress that T-RES does not oppose the current IoT approach, but aims at complementing it. Indeed, complex IoT applications may benefit from having part of their logic running "out-of-the-network" (e.g., complex computations performed in the Cloud) and some other part running "in-the-network" (e.g., simple event-detection functions).

The remainder of the paper is organized as follows. Section II describes the REST paradigm, on which IoT-based WSNs are designed. Section III presents the "Task Resource Abstraction" and discusses its main benefits. Section IV presents an evaluation of the proposed solution through a real implementation for WiSMote devices [7]. Section V discusses related work. Finally, Section VI provides concluding remarks.

II. REST PARADIGM IN IOT-BASED WSNS

REpresentational State Transfer (REST) [8] is an architectural style for distributed systems introduced and defined in 2000 by Fielding in his doctoral dissertationn [8]. REST-style architectures consist of clients and servers. Clients send requests to servers, which reply with appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of the resource. The most relevant example of a system conforming to the REST architectural style is the World Wide Web, in which resources are manipulated using the HTTP protocol.

The REST paradigm is also used in IoT-based WSNs, where resources usually represent sensors, actuators or some kind of information. However, in IoT-based WSNs, the CoAP protocol is used instead of HTTP. CoAP is similar to HTTP, but it was especially designed for constrained devices. CoAP allows sensor nodes to run embedded web services through which their resources can be manipulated. Specifically, CoAP provides four methods for manipulating resources: PUT, which requests that the resource identified by the URI specified in the request be updated or created with the transmitted representation; POST, which requests that the representation transmitted in the request be processed; GET, which retrieves a representation of the resource identified by the URI specified in the request; and DELETE, which requests the deletion of the resource identified by the URI specified in the request. Both PUT and POST can be used to create or change a resource[1]. CoAP also provides a resource observation mechanism [9] which allows a node to receive notifications about changes in resources it has previously subscribed to.

Fig. 1 depicts a simple example of an IoT-based WSN installed in an office. There are two nodes: one is equipped with a temperature sensor (on the right), while the other one is connected to a heating device (on the left). For each node, the IPv6 address and the available resources are shown. The current temperature value can be retrieved by issuing a GET request on the URI `coap://[aaaa::1]/sens/temp`. Similarly the heating device can be switched on (off) by sending a PUT

[1]The difference between PUT and POST is that PUT is idempotent, whereas POST is not. However, for the sake of simplicity, in this paper they are used interchangeably.

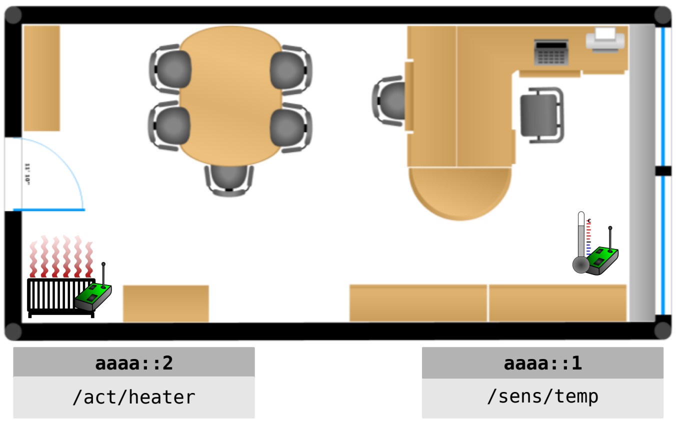


Fig. 1. Example of an IoT-based WSN with two nodes, one equipped with a temperature sensor (right), the other one attached to a heating device (left). For each node, the IPv6 address and the available REST resources are shown.

request for the URI `coap://[aaaa::2]/act/heater`, containing the "on" ("off") control message in its payload.

The use of standard protocols (i.e., IPv6 and CoAP) allows the nodes to be used for many different applications. However, such protocols alone do not provide a way for changing the devices application logic once nodes are installed and configured for a specific task. Consider, for instance, the case in Fig. 1: if the two nodes are installed and configured at different times just for sensing and actuating, thus without a joint logic in which they communicate to each other, an automatic heating control application cannot be built up in an easy way. In a traditional WSN scenario the new application logic would require a total or partial firmware update for at least one node, while in state-of-the-art IoT-based WSNs an external entity (e.g., another node, the Cloud, etc.) would be necessary for subscribing to the temperature resource, running the control logic, and sending the proper PUT request to the heater resource. The solution presented in the following section solves this issue by proposing the "Task Resource Abstraction", a mechanism capable of manipulating the local processing of an IoT node and its interactions with other nodes without transferring firmware updates.

III. THE TASK RESOURCE ABSTRACTION

The goal of T-RES is to enable easy tasking of IoT nodes. We limit the scope of T-RES to simple data-processing tasks performing the following actions: (i) monitoring one or more resources, (ii) executing some data-processing on their values, and (iii) sending the resulting output to other resources. The main idea beyond T-RES is to represent data-processing tasks as CoAP resources. For instance, a node computing the average temperature in a room can expose the `/tasks/roomavgtemp` "task resource". A task resource, like any other classic resource, can be manipulated using CoAP methods: it can be created, deleted, modified, or even retrieved (duplicated). The proposed approach is simple in its vision, yet powerful. Both push and pull models are, indeed, supported: at any moment a third entity (e.g., the user) can assign (push) a new task to a sensor node using a PUT/POST request, or the sensor node can autonomously retrieve (pull) a task from another node or a repository by performing a GET

TABLE I. T-RES APIS

API	Description
`getInput()`	Get the current input
`getInputTag()`	Get the tag of the current input
`setOutput()`	Set the output
`getState()`	Get the last saved state
`saveState()`	Save the state

request. Moreover, complex scenarios in which sensor nodes automatically find and get the tasks they need can be realized.

A. T-Res RESTful interface

A data-processing task is completely defined by its input sources, the data-processing it performs, and the destination of its output. T-RES keeps separated those three elements by defining a sub-resource for each of them. The resulting structure of a task resource is shown in Fig. 2. Both the input-source resource (`/tasks/[task_name]/is`) and the output-destination resource (`/tasks/[task_name]/od`) contain URIs, i.e., references to other resources. T-RES monitors input resources for new values by using the CoAP observe mechanism. The processing-function resource (`/tasks/[task_name]/pf`) contains the code performing the desired data-processing. Such code is simply a function that is called every time a new input value is available and that may produce an output value which is sent to the output resource (more details are provided in the next section). As shown in Fig. 2, there is one additional resource, the last-output resource (`/tasks/[task_name]/lo`), which contains the last output produced by the task. Its purpose is to allow the concatenation of tasks. Indeed, the last-output resource of a task can be set as an input resource of another task.

B. T-Res processing functions

As previously mentioned, the processing function of a task resource contains the code to be executed when a new input is received. To fully decouple applications from the sensing infrastructure, processing functions must be platform independent. Therefore, they cannot be written in languages that must be compiled into native code, like C, C++, etc.; instead, languages that can be compiled into bytecode or directly executed by an interpreter must be used. After considering the advantages and disadvantages of the most common interpreted languages (namely, Python, Java, and Javascript) we decided to use Python for defining processing functions. Python was preferred over Java because we believe that processing functions are better implemented as scripts than objects. Javascript was discarded as well because, unlike Python, it cannot be compiled into bytecode and, therefore, Javascript scripts are too big to be installed in constrained devices.

T-RES also provides a set of APIs that can be used in Python scripts to define processing functions. Such APIs, summarized in Table I, are used to: get the value and the tag of the last input received, set the output value, and save and retrieve the task state. A tag is a string associated with an input source and allows the script to identify the source that generated the current input. The state of a task can be represented by an object having only data fields, one for each state variable the developer wishes to store among executions.

C. Example of usage

To better describe T-Res functionality, we extend the example presented in Section II with an application that tries to keep the temperature between 19°C and 21°C. To build such an application, a new T-Res resource can be created in any node of the network. We decide to create it in the heating device. To do that we issue a PUT request for the URI `coap://[aaaa::2]/tasks/heatcontrol`, where `heatcontrol` is the name of the new task to be created. Since we want `heatcontrol` to monitor (using the CoAP observe mechanism) the temperature sensor, we set its `/is` sub-resource as follows:

```
coap://[aaaa::1]/sens/temp
```

Similarly, since we want the task to control the heating device, we set its `/od` sub-resource as follows:

```
coap://[aaaa::2]/act/heater
```

Finally, its `/pf` sub-resource must contain the bytecode for performing the desired action when a sensor notification is received. Such bytecode can be obtained by compiling the following Python script.

```
from tres import *
t = getInput()
if t < 19:
  setOutput("on")
if t > 21:
  setOutput("off")
```

The script simply switches on the heating device when the temperature is below 19°C and turns it off when the temperature is above 21°C.[2] As shown by the code, T-Res supports also situations in which no output must be produced. In our example application that happens when the temperature is in the target range. Indeed, in that case, we do not need to change the state of the heating device.

If we suppose that an additional temperature node is deployed, we can change the application to make the heating control depend on the average temperature, and not on just one of the two measurements. This improvement can be done by modifying the existing T-Res resource or by adding a new one which computes the average. The second approach is preferable, because it allows the added task resource to be used also in other applications. Moreover the first option is not always possible, because the Python script to modify may not be available (e.g., because created by another developer).

Therefore, we decide to create a new task resource on the first temperature sensor, while leaving unchanged the `heatcontrol` task on the heating device. To do that we issue a PUT request for the URI `coap://[aaaa::1]/tasks/avgtemp` where `avgtemp` is the name of the new task to be created. The content of the its `/is` sub-resource is set as:

```
coap://[aaaa::1]/sens/temp "Temp1"
coap://[aaaa::3]/sens/temp "Temp2"
```

[2]This script could actually be improved. Indeed, it keeps sending messages for turning on (off) the heating device as long as the temperature is below 19°C (above 21°C). To avoid sending such unnecessary messages, a state variable for storing the last output can be used. However, for the sake of brevity, such an improvement is not shown.

```
/tasks                      # list currently installed tasks [GET]
      /[task_name]          # retrieve/create/delete a specific task [GET|PUT|DELETE]
                   /is      # retrieve/update the input sources [GET|PUT|POST]
                   /pf      # retrieve/update the data-processing function [GET|PUT]
                   /od      # retrieve/update the output destinations [GET|PUT|POST]
                   /lo      # retrieve/observe the last output produced [GET]
```

Fig. 2. The structure of a task resource. Each task resource is a child of `/tasks` and contains four sub-resources: the "input sources" resource (`/is`), the "processing function" resource (`/pf`), the "output destinations" resource (`/od`), and the "last output" resource (`/lo`). For each resource, the CoAP methods for manipulating it are shown.

Where `aaaa::3` is the address of the new temperature node and "Temp1" and "Temp2" are the tags of the first and the second temperature node, respectively.

The content of its `/od` sub-resource is left blank (because we do not want the `avgtemp` task to send its output to any actuator) and its `/pf` sub-resource is set to contain the bytecode generated compiling the following Python script.

```
1  from tres import *
2
3  class state:
4    def __init__(self):
5      self.t1 = -273.15
6      self.t2 = -273.15
7
8  s = getState(state)
9  tag = getInputTag()
10 if tag == "Temp1":
11   s.t1 = getInput()
12 else:
13   s.t2 = getInput()
14 saveState(s)
15 if (s.t1 > -273.15) and (s.t2 > -273.15):
16   setOutput((s.t1 + s.t2) / 2)
```

The script shows the use of both state and tags. State is used to save the last measurement of each temperature node. That is necessary because new values are processed one at a time. Tags are used to identify the source of the input currently being processed. Lines 3–6 of the script define the class representing the state and its default values. Line 8 retrieves the old state using the function `getState()`. The class state is passed as an argument because it is used by the function to create a new state instance during the first execution of the script (i.e., when no old state is available). Line 9 retrieves the tag of the current input using the `getInputTag()` function and stores it to a temporary variable. Lines 10–13 use the tag to check whether the input is from the first or the second temperature node, and store it to the proper state variable. Line 14 saves the state. Finally, lines 15–16 set the output to the average of the two temperatures, but only if they are both different from the default value.

However, the output value is not sent to any actuator, because none is configured in the `/od` sub-resource. The output is simply stored in the `/lo` ("last output") sub-resource, which can be observed by other nodes. Therefore, to complete the improvement of our heating control application, we just reconfigure the `heatcontrol` task to use the output of the new `avgtemp` task as its input. We do so by setting the content of the `/is` sub-resource of `heatcontrol` as:

```
coap://[aaaa::1]/tasks/avgtemp/lo
```

Fig. 3 summarizes the resulting task configuration and the relationship among resources.

D. *T-Res benefits*

As mentioned before, the main goal of T-RES is to decouple applications from the sensing infrastructure without sacrificing in-network processing. However, T-RES provides also some other benefits. First, as shown by the previous examples, T-RES greatly simplifies application development. Indeed, it completely hides the communication details, thus allowing developers to focus on the application logic. Moreover, T-RES does not require developers to be experienced in embedded programming: a little knowledge of Python is enough, at least for simple applications.

Another relevant characteristic of T-RES is that relationships among resources are explicit and separated from the processing logic. As a consequence, input and output resources can be changed without modifying the Python script: just a PUT request for the `/is` or `/od` resource is required. Similarly, if a task resource is already deployed, its input and output resources can be discovered by issuing GET requests. That allows a third entity, e.g., an Application Manager, to build a graph of relationships among resources and tasks, with the aim of using it for identifying possible conflicts among tasks (e.g., tasks that control the same actuator). The graph can also be compared with a network topology graph for optimizing the deployment of task resources. Indeed, as discussed in Section IV, the decision about where (i.e., in which node) to create a task resource has an important impact on performance (e.g., network traffic, application delay, etc.).

IV. EVALUATION RESULTS

In this section we prove the feasibility of T-RES and we evaluate its performance with a real implementation. First, we check that both the code size (flash memory occupancy) and the memory requirements (RAM occupancy) are within the limits posed by the latest generation of WSN motes. Then, we evaluate the impact of different task allocation choices on T-RES performance. Finally, we compare the performance of T-RES with that of an external logic approach, in which the application logic runs "out-of-the-network". Results show that T-RES reduces network traffic and node power consumption, whereas it may lead to higher execution times due to Python overhead. Nevertheless, the whole application response time remains comparable to that of the external logic approach and, in some cases, it is even reduced.

A. *T-Res implementation*

We implemented a prototype of T-RES on top of the Contiki Operating System [10]. Contiki provides native support for IPv6 and CoAP, but it lacks a Python interpreter. To overcome this limitation, we ported PyMite [11], a reduced Python

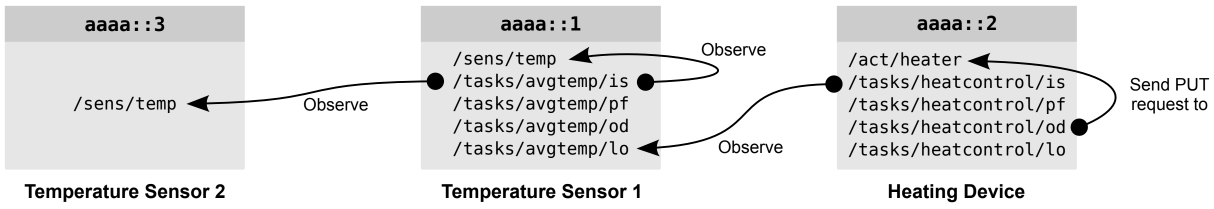


Fig. 3. An application for controlling the heating according to the average room temperature. The application is composed of two tasks: `avgtemp` and `heatcontrol`. The first monitors the two temperature sensors (using CoAP observe mechanism), computes their average value, and stores it in its `/lo` ("last output") resource. This resource is monitored by the second task, which uses its content (i.e., the computed average value) for controlling the heating device.

TABLE II. CODE SIZE AND RAM REQUIREMENTS FOR A WISMOTE DEVICE WITH AND WITHOUT T-RES.

Node type	RAM [bytes]	Flash [bytes]
Regular CoAP server	7 470 (46%)	45 623 (18%)
T-RES node	15 078 (93%)	95 455 (37%)

virtual machine for embedded systems, to Contiki. We also extended Contiki with two new features: (1) we added client-side support for CoAP Observe, which is needed by T-RES to "observe" the input resources; and (2) we implemented IPv6 loopback communication (i.e., the possibility for a process to communicate with another process on the same node using sockets), needed by T-RES tasks to interact with resources belonging to the node on which they are deployed.

We tested the resulting implementation on the WiSMote platform [7], which is equipped with an MSP430F5 micro-controller having 16 kB of RAM and 256 kB of flash memory. As shown in Table II, a sensor node running T-RES requires more than twice the RAM and the flash memory needed by a traditional IoT sensor node. Such a notable increase in memory requirements is mainly caused by PyMite, which alone occupies 45 kB of flash memory and requires almost 8 kB of RAM, mainly used as a heap for Python scripts. As a result, 93% of the RAM available on a WiSMote is occupied and, since the current prototype stores the uploaded Python bytecode in RAM, a WiSMote can provide just one task resource. We plan to overcome this limitation in future implementations by storing the uploaded Python scripts in the flash memory, which is mostly free. Indeed, processing functions can share the same PyMite heap, because they are executed one at a time and the heap is freed after each execution. Therefore, if Python scripts are stored in the flash memory, adding additional task resources leads to just a small increment in RAM requirements.

B. Evaluation setup

We evaluate our T-RES implementation using Cooja, the Contiki network simulator [12]. Cooja integrates MSPsim [13], a tool that can emulate motes based on the MSP430 microcontroller, including WiSMotes. Cooja/MSPsim provides cycle-accurate simulation of the individual devices, as well as bit-level accurate simulation of their radio transceivers. As a consequence, Cooja/MSPsim allows running the exact same binaries in the simulator as on actual hardware.

For our evaluation, we consider two test applications, which, for the sake of simplicity, have just one input and one output. Specifically, one is a simple event detector which monitors a sensor and sends an alarm to an actuator every time the sensor data exceeds a certain threshold. The other one is a more complex PID controller which must hold a state and perform some floating-point operations. The PID controller monitors a sensor that generates data periodically, and, for every received input, sends a control message to an actuator. Both applications are implemented as a single T-RES task.

We evaluate such applications on a simulated multi-hop IoT-based WSN, composed of 5 WiSMotes: one sensor producing data periodically (every second), one actuator (i.e., a node with a resource that can receive PUT/POST requests), one border router connecting the WSN to the Internet, and two other nodes that just forward packets. Every node is also a T-RES node, i.e., it can host a task resource. Nodes are configured to use static routing and no radio duty cycling mechanism.

C. Task allocation

We first evaluate the impact of the deployment configuration on T-RES performance. Indeed, the decision of where to install the task resource affects the resulting network traffic, application delay, power consumption, etc. If the goal is that of minimizing network traffic, we can use the following heuristics for deploying single-input/single-output tasks (like those we are considering): if the average data generated and received by the sensor (i.e., the notification and the corresponding ack) is less than the average data sent to and received from the actuator (i.e., the PUT request and the corresponding ack), place the task on the actuator; otherwise, place it on the sensor.

To verify the correctness of such a rule, we simulate the topologies in Fig. 4, in which node 3 is the network border router, node 1 is the sensing node, and node 5 is the actuator node. When the sensor resource is observed, the sensing node periodically sends a CoAP notification packet of 63 bytes (including the full UDP, IPv6 and IEEE 802.15.4 overhead) and waits for a 54-byte-long acknowledgment. The actuator can be controlled with a 69-byte-long PUT request and generates a 56-byte-long acknowledgment/response message. Therefore, we expect the actuator node to be the best choice for hosting the PID controller, since $63 + 54 < 69 + 56$. For the event detector, instead, we must also consider the event probability. Indeed, if the sensor data does not exceed the threshold (i.e., the event does not occur), the task does not send any output to the actuator. Therefore, if we impose an event probability of 0.5, the event detector should be installed on the sensor, since $63 + 54 > .5 \times (69 + 56)$.

Fig. 5 shows simulations results for topology #0 (results for other topologies are similar and not reported for the sake

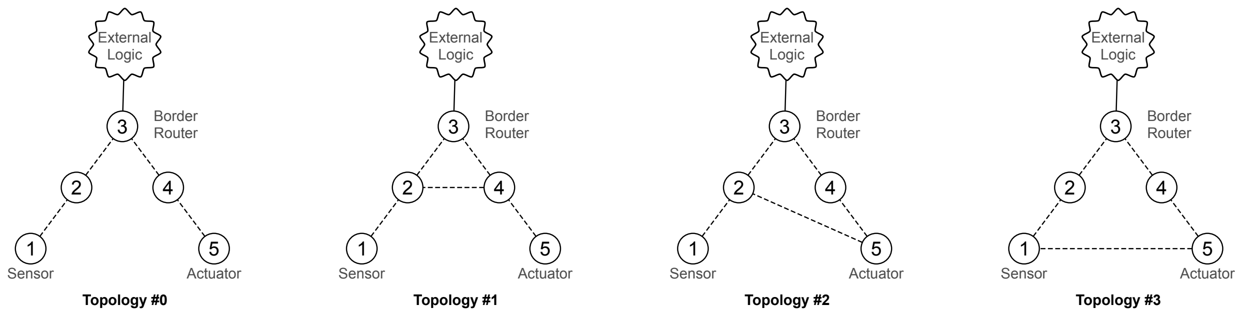


Fig. 4. The four different topologies used in our simulations. The identifier number of each topology is also the number of hops saved when using T-Res instead of the external logic approach.

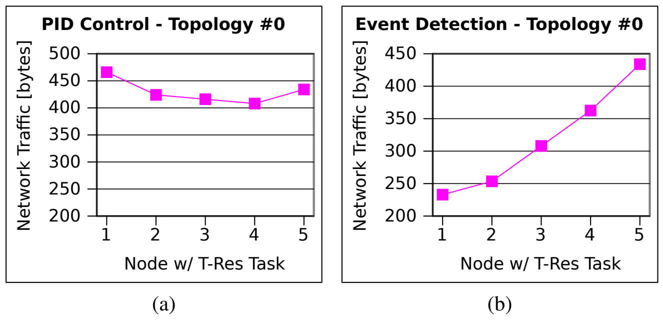


Fig. 5. Network traffic as a function of the task position. For the PID scenario (a), the graph shows the exact network traffic, which is the same for every task execution. For the event detection scenario (b), instead, the network traffic depends on the actual occurrence of the event. Therefore, the graph shows the average network traffic computed on 5000 executions (99% confidence level and relative error less than 1%). The event probability was 0.5.

of brevity). Specifically, the graphs show how the network traffic changes while moving the task along the path from the sensor (i.e., node 1) to the actuator (i.e., node 5). Surprisingly enough, the results do not match our expectation for the PID controller application (Fig. 5a). Indeed, in this case, the minimum network traffic is achieved when the task is placed in node 3, i.e., one hop before the actuator node. This result is due to the 6LoWPAN IPv6 header compression mechanism [14], which makes IPv6 overhead not constant among hops. Specifically, for every packet, the IPv6 source address and the Hop Limit field are completely elided in the first hop, thus saving 9 bytes, and the IPv6 destination address is likewise completely elided in the last hop, saving additional 8 bytes. Therefore, to further exploit IPv6 header compression, it is better to place the PID controller not in the node identified by our heuristics, but in its neighbor (along the path from the sensor to the actuator). Fig. 6 further explains this result.

Simulation results also show the importance of carefully choosing where to deploy T-Res tasks. Indeed, a bad choice can hugely increase the resulting network traffic: for example, in the event detection scenario, when the task is deployed on the actuator (i.e., node 5), rather than on the sensor (i.e., node 1), the network traffic nearly doubles (see Fig. 5b). Indeed, if the task is placed on the sensor, no packets are exchanged when the event does not occur; whereas, if the task is placed on the actuator, no communication saving is possible.

D. Comparing T-Res with the external logic approach

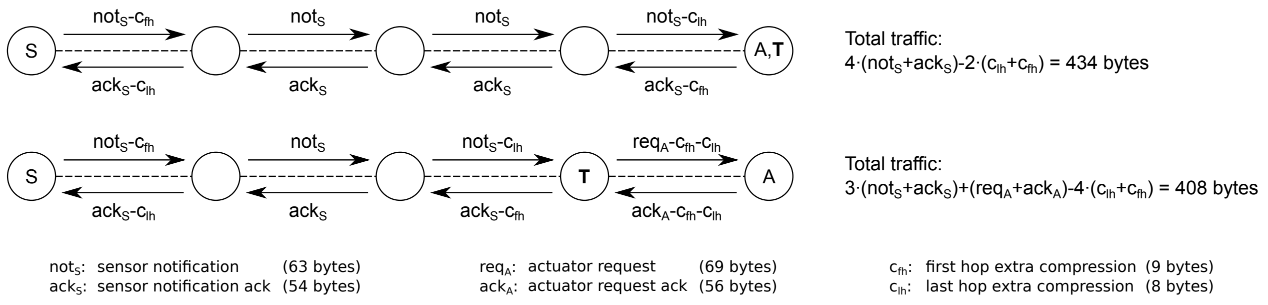
In this section, the performance of our two applications implemented as T-Res tasks is compared with the performance of the same two applications implemented as native code running on the border router. The latter is a convenient approximation of the external logic approach, in which applications run on the Cloud and access the network through the border router.

We consider four performance metrics: network traffic, application delay, average energy consumption and maximum energy consumption. The *network traffic* is defined as the total amount of data exchanged in the network (including forwarding) as a consequence of a single task execution. The *application delay* is defined as the time interval between the availability of a new sensor reading and the delivery of a new control signal to the actuator. The *average energy consumption* is defined as the energy consumed on average by a node during each execution period, i.e., the time interval between one sensor reading and the other (1-second long in our simulations). The *maximum energy consumption* is, instead, the energy consumed by the most power-hungry node. This last metric is especially important because it may affect the network lifetime. Indeed, if a WSN is not redundant enough, it may become nonfunctional as soon as the first node dies. For both the energy metrics, the power consumption of the border router is not taken into account, since border routers are usually mains-powered.

The comparison about network traffic and power consumption is not affected at all by our approximation of the external logic approach, which, on the contrary, has some impact on the application delay. Indeed, the application logic takes longer to execute on a WiSMote than on a high-end device. However, for our test applications, such overhead is less than 500 µs and, therefore, it is offset by the lack of communication delay between the border router and the external server.

Leveraging the results discussed in the previous section, we always install the task resource in the node that minimizes the network traffic. Under such conditions, we expect T-Res to always perform better than the external logic approach in terms of network traffic. On the contrary, T-Res could cause a higher application delay because of the Python execution overhead. Such overhead will also increase the CPU usage and, consequently, the energy consumption of the node running the application logic. However, that increase should be more than offset by the energy saved thanks to the lower network traffic.

For each application scenario, we perform 4 simulations:



not_S-c_{fh}
not_S
not_S
not_S-c_{lh}
S
A,T
ack_S-c_{lh}
ack_S
ack_S
ack_S-c_{fh}
Total traffic:
4·(not_S+ack_S)-2·(c_{lh}+c_{fh}) = 434 bytes
not_S-c_{fh}
not_S
not_S-c_{lh}
req_A-c_{fh}-c_{lh}
S
T
A
ack_S-c_{lh}
ack_S
ack_S-c_{fh}
ack_A-c_{fh}-c_{lh}
Total traffic:
3·(not_S+ack_S)+(req_A+ack_A)-4·(c_{lh}+c_{fh}) = 408 bytes
not_S: sensor notification (63 bytes)
ack_S: sensor notification ack (54 bytes)
req_A: actuator request (69 bytes)
ack_A: actuator request ack (56 bytes)
c_{fh}: first hop extra compression (9 bytes)
c_{lh}: last hop extra compression (8 bytes)

Fig. 6. Resulting network traffic for the PID control scenario with topology #0 when the control logic is deployed on the actuator and on its neighbor. Since IPv6 header compression is higher in the first and last hop and, in this case, the difference between $req_A + ack_A$ and $not_S + ack_S$ is small enough, i.e., less than $2 \cdot (c_{fh} + c_{lh})$, moving the control logic one hop before the actuator reduces the network traffic.

one for each topology depicted in Fig. 4. The main difference between them is the length of the shortest path from the sensor (S) to the actuator (A), while the path from S to A through the border router (B) does not change. Therefore, we expect T-RES to improve its performance when the path from S to A is shorter, whereas the performance of the external logic should be constant for each topology.

The obtained results, shown in Fig. 7, confirm our expectations. Indeed, both the network traffic and the application delay decrease linearly with the number of transmission hops saved by T-RES. This allows saving up to 85% of the traffic generated by the external logic approach. Fig. 7 also shows that Python overhead does not undermine the responsiveness of our applications. Indeed, just 1 or 2 saved transmission hops are enough to make the application response time shorter than that of the external logic case, thus proving T-RES beneficial in large multi-hop IoT-based WSNs.

Results about power consumption are satisfactory as well. The use of T-RES always leads to lower energy consumption with the only exception of the PID control application with topology #0, in which case the consumption of the two approaches is almost the same. This result can be easily explained noting that, in such a case, T-RES achieves just a small saving in network traffic. Results are, instead, especially good for the event detection case, in which, for every topology, the maximum energy consumption (i.e., the energy consumption of the most power-hungry node) of T-RES is lower than the average energy consumption of the external logic approach. The same happens for the PID control case, but only with topology #2 and #3, i.e., when T-RES can save 2 and 3 transmission hops respectively.

V. Related Work

The use of virtual machines in WSNs is not a novelty. For example, in 2002, Levis and Culler [15] proposed Maté, a tiny virtual machine that can be tailored to specific domain and supports code mobility. More recently general-purpose virtual machines have been developed as well. Most of them, such as TakaTuka [16] and Darjeeling [17], are Java virtual machines that allow using Java for programming sensor nodes, including their peripherals (e.g., their radio, sensors, actuators, etc.). In this regard, T-RES novelty is to limit the scope of the (Python) virtual machine to the execution of simple data-processing functions, while T-RES takes care of monitoring their input sources and sending their outputs to the configured destinations. That limits the size and memory requirements of both the application bytecode and the Python virtual machine.

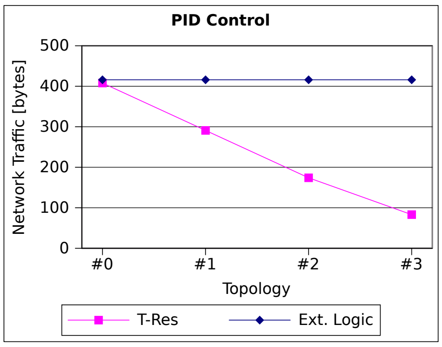
The already mentioned Actinium [6] is a RESTful run-time container allowing dynamic installation, update, and removal of *apps*. An Actinium *app* is somewhat similar to a T-RES task: both of them have a RESTful interface and are implemented as scripts. However T-RES tasks are expected to run directly on motes, whereas Actinium *apps* can only run on resource-rich *app servers*. The main advantage of T-RES over Actinium is the increased degree of in-network processing, while the main advantage of Actinium over T-RES is the support for complex applications (e.g., applications with high computational requirements). However, thanks to their RESTful interface, T-RES and Actinium can be used together, thus allowing developers to exploit the benefits of both.

Duquennoy et al. [18] have been among the first to discuss the use of CoAP for node reprogramming. However, their work addresses the problem of sensor network deployment and, therefore, they mainly focus on proving the feasibility of transferring code-updates using CoAP/UDP. T-RES uses CoAP as well (although in T-RES case, bytecode is transferred instead of native code), however, it also defines a RESTful interface for reconfiguring the interaction among nodes and the data-processing they perform.

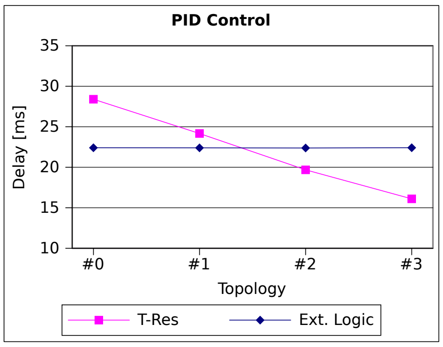
VI. Conclusions

This paper presented T-RES, a programming abstraction for achieving reconfigurable in-network processing in IoT-based WSNs. To the best of authors' knowledge, T-RES is the first solution that allows decoupling IoT applications from the sensing infrastructure without sacrificing in-network processing. Indeed, state-of-the-art solutions achieve such decoupling by running the entire application logic in external servers or in the Cloud. We stress, however, that T-RES does not oppose such an approach, but just aims at complementing it.

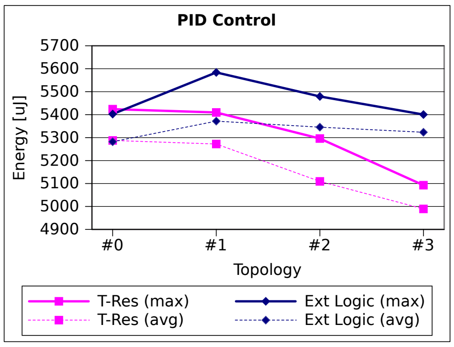
T-RES also simplifies the development of in-network IoT applications. Indeed it completely hides communication details and does not require developers to be experienced with embedded programming. Moreover T-RES RESTful interface allows for a hypothetical application manager that automatically finds relationships among T-RES tasks and node resources, and uses them for detecting conflicts or optimizing task allocation. Indeed, since tasks are represented as CoAP resources, they can be easily created, deleted, modified, or retrieved.



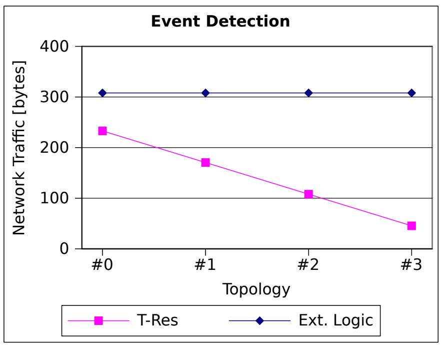
(a)



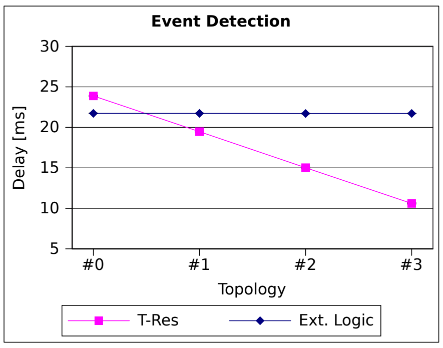
(b)



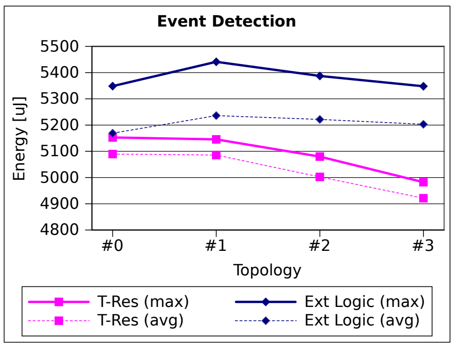
(c)



(d)



(e)



(f)

Fig. 7. Comparison of T-RES with the external logic approach in terms of network traffic, application delay and energy consumption, as a function of the topology used, for the two different scenarios. Each data point is obtained averaging the data from many different executions in order to achieve a relative error lower than 1% (0.1% for the energy consumption measurements) with a 99% confidence level.

The paper proved the feasibility of T-RES with a real implementation capable of running on constrained devices. The implementation was also used to compare the performance of T-RES with that of the external logic approach. Results showed the benefits of T-RES in terms of network traffic, power consumption, and application delay reduction. Specifically, for our evaluation scenarios, T-RES always reduced the network traffic saving up to 85% of the traffic generated by the external logic approach. T-RES benefits in delay reduction were, instead, noticeable only when the tested network topology allowed T-RES to save communication hops over the external logic case. Our evaluation also proved that T-RES, despite Python's overhead, does not increase energy consumption. On the contrary, it saves energy due to the reduced network traffic.

As future work, we plan to use T-RES within the ICSI project, [3] the goal of which is to define a new architecture for enabling cooperative sensing in intelligent transportation systems and to develop a reference end-to-end implementation of it. Specifically, T-RES will be an essential component of the event-based middleware that is being developed as part of the project. The ICSI project will also provide the opportunity of testing T-RES performance on two real-world deployments.

REFERENCES

[1] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann, "Towards multi-purpose wireless sensor networks," in *Proc. Systems Communications*, 2005, pp. 336–341.

[2] J. W. Hui and D. E. Culler, "IP is dead, long live IP for wireless sensor networks," in *Proc. 6th ACM Conf. on Embedded network sensor systems (SenSys'08)*, 2008.

[3] Z. Shelby, "Embedded web services," *Wireless Communications, IEEE*, vol. 17, no. 6, pp. 52 –57, 2010.

[4] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained Application Protocol (CoAP)," IETF Internet-Draft, Dec. 2012. [Online]. Available: http://tools.ietf.org/id/draft-ietf-core-coap-13.txt

[5] M. Kovatsch, S. Mayer, and B. Ostermaier, "Moving application logic from the firmware to the Cloud: towards the thin server architecture for the internet of things," in *Proc. 6th Int. Conf. on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS'12)*, Palermo, Italy, Jul. 2012.

[6] M. Kovatsch, M. Lanter, and S. Duquennoy, "Actinium: A restful runtime container for scriptable internet of things applications," in *Proc. 3rd Int. Conf. on the Internet of Things (IoT'12)*, Wuxi, China, Oct. 2012.

[7] "WiSMote," http://wismote.org, [Apr. 15, 2013].

[8] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000.

[9] K. Hartke, "Observing Resources in CoAP," IETF Internet-Draft, Oct. 2012. [Online]. Available: http://tools.ietf.org/id/draft-ietf-core-observe-07.txt

[10] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annual IEEE Int. Conf. on Local Computer Networks (LCN'04)*, 2004.

[11] "PyMite," http://code.google.com/p/python-on-a-chip/, [Apr. 15, 2013].

[12] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with COOJA," in *Proc. 31st IEEE Conf. on Local Computer Networks (LCN'06)*, Nov. 2006.

[13] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt, "MSPsim – an extensible simulator for MSP430-equipped sensor boards," in *Proc.*

[3]http://www.ict-icsi.eu

European Conf. on Wireless Sensor Networks (EWSN'07), Poster/Demo session, 2007.

[14] N. Hui and P. Thubert, "Compression format for IPv6 datagrams over IEEE 802.15.4-based networks," RFC 6282, IETF internet standard, Sep. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6282.txt

[15] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *Proc. 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X'02)*, 2002.

[16] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. A. Uzmi, "Optimized java binary and virtual machine for tiny motes," in *Proc. 6th IEEE Int. Conf. on Distributed Computing in Sensor Systems (DCoSS'10)*, Santa Barbara, CA, Jun. 2010.

[17] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich VM for the resource poor," in *Proc. 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys'09)*, Berkeley, CA, Nov. 2009.

[18] S. Duquennoy, N. Wirström, N. Tsiftes, and A. Dunkels, "Leveraging IP for Sensor Network Deployment," in *Proc. Int. Workshop on Extending the Internet to Low power and Lossy Networks (IP+SN'11)*, Chicago, IL, USA, Apr. 2011.