WARM: WSN Application development and Resource Management

Henrique C. Silva, André Hahn Pereira, Yuka K. Solano, Bruno T. de Oliveira and Cíntia B. Margi

Abstract— **Wireless Sensor Networks (WSN) play an important part in enabling the Internet of Things, but WSN application development and resource management are still a costly endeavour. This cost is related to the diversity of resource constrained platforms and network protocols, which demand specialized knowledge and practice. To solve this, we propose WARM: a framework that employs Web Service and Software Defined Networking paradigms to enable the parametrized scheduling of tasks. WARM allowed us to configure typical applications running on a simulated WSN using a web browser. Furthermore, we consider that WARM enables the concept of *Sensing as a Service*.**

Keywords— **Wireless Sensor Networks, Software Defined Networking, Application development, Resource management**

I. Introduction

The Internet of Things (IoT) is a new term that has drawn a lot of attention from the computing community [2]. It is composed of devices capable of sensing/actuation, communication and processing [21]. There are different types of IoT applications and approaches to implement them, being the two main approaches centralized services and peer-to-peer [21].

IoT centralized services benefit from IETF standards, such as 6lowpan [19] and CoAP (Constrained Application Protocol) [4], and from cloud infrastructure such as the European initiative FIWARE [10]. On the other hand, on the peer-to-peer approach, IoT envisions a network of physical objects enabled with the capability of collaborating with each other in order to make smart environments and objects. Wireless Sensor Networks (WSN) are part of this vision and become a key enabling technology. It is worth to notice that hybrid solutions are likely to be used.

While WSN are key enabling technology for IoT, a larger usage requires the easing of the development of applications, of the infrastructure management, and of the access to the configuration and the data produced by such networks. Furthermore sharing the WSN infrastructure with different applications is key to cost efficiency.

As an answer to this need, we propose a Software Defined Networking (SDN) based framework, WARM, aimed for the development of WSN applications. With WARM, *Sensing as a Service* could become a reality.

WSN are composed of spatially distributed autonomous sensors characterized by resource constraints including limited processing speed, storage capacity and communication bandwidth [7]. In face of such constraints, the paradigm of SDN can be an advantage on the management of WSN nodes and resources, improving reliability and performance of WSN. Such improvements can be achieved mainly because of the flexibility given by the SDN paradigm, which allows the network administrator to easily change routes within the network to fit new policies. These policies could address WSN's main problems: limited node energy, link quality changes, node failures and limited node processing power [20], [8], [13].

A framework such as WARM could greatly benefit from the SDN paradigm, not only due to the network monitoring and routing flexibility it allows, but also due to the main concept behind: the separation between control and data planes, with particular data flows for each of them. This separation provides an abstraction that allows the Framework Controller to be built dealing with the data plane, leaving the network control plane to the SDN controller. This way, WARM's Framework Controller can use status information gathered by the SDN controller in order to balance the load of multiple network applications throughout the more appropriate nodes, saving energy and ensuring resource use optimization.

Thus the main contribution of this paper is to present WARM, an SDN-based framework for developing and managing WSN applications. Its specification and implementation are described, as well as the experiments executed showing how WARM enables the concept of *Sensing as a Service*.

This paper is organized as follows. Section II describes context and related work. Section III describes WARM specification, while Section IV discusses its implementation. Results are presented and discussed in Section V. Section VI concludes this paper presenting final remarks and future work.

II. Context and related work

In this section we present our review on the challenges concerning the development and management of WSN applications, as well as previous solutions created as an attempt to reduce these difficulties. We first analyze several frameworks which can be divided in the categories of query processors and macroprogramming frameworks, discussing the advantages and disadvantages of each one. We then review the contributions that the Software Defined Networking paradigm (SDN) can provide in order to make WSN more flexible and manageable. Finally, we present the only work to our knowledge that proposed a WSN application framework based on the SDN paradigm.

Since the inception of WSN, frameworks have played a role in easing the work of dealing with specific tasks and patterns

H. C. Silva, A. H. Pereira, Y. K. Solano, B. T. de Oliveira and C. B. Margi¸ Laboratório de Arquitetura e Redes de Computadores, Universidade de São Paulo, São Paulo-SP, Brazil, E-mails: henrique.c.s@usp.br, andre.hahn@usb.br, yuka.solano@usp.br, brunotrevizan@usp.br, cintia@usp.br.

common to sensor network applications. The first approach for WSN frameworks was to make these networks as close as possible to a real-time updated database. One such framework is TinyDB [14], a distributed query processor system. SWISS QM [17] is also designed as a query processor, but is built on top of a virtual machine (VM) in order to be independent of the query language. GSN [1] is a WSN middleware also designed to represent WSN tasks as SQL queries, but based on the concept of swappable XML defined *virtual sensors* which abstract sensor node hardware.

Despite their flexibility for data sensing applications, the query based data modelling provided by TinyDB, SWISS QM and GSN is not focused on controlling roles. Such roles, however, are an important part of the IoT infrastructure [21].

Another category of WSN frameworks makes use of the macroprogramming paradigm as a means to provide easy yet powerful ways to develop WSN applications [3], [5]. Their goal is not to prune development by focusing applications on specific scenarios such as sensing data.

PyoT [3] aims to be a complete solution for the Internet of Things utilizing the Python script language. PyoT abstracts low-level hardware details through a message-oriented middleware, which handles the tasks distributed by a network control center. Terra [5], on the other hand, implements a VM on top of each sensor node so it can execute the bytecode generated by compiling statically checked script applications. Its focus is to decrease application binary size in order to reduce the overhead of transmission during remote application updates.

Although powerful, the macroprogramming approach also has its limitations. One of them is denying the possibility of giving each sensor node an individual role, since the application development process consists of specifying a general behavior for the whole network. It also encumbers the developer with the need to write and debug embedded code, unlike query-based solutions. Finally, it restrains network behavior and policy configuration to be maintained through software which must be remotely updated in case of change. This last aspect is one of the reasons we look into SDN-based solutions as a possible approach to build upon.

Initially conceived for high speed and low latency wired networks, the SDN paradigm is based on the separation of a network's control and data planes. This allows to configure network policies through software, as if the whole network was one single virtual entity [11]. In practice, this means packets inside an SDN are labeled as belonging to a flow and are routed according to rules specific for this flow. These rules are defined by one SDN controller, and then configured on the switches' flow tables.

Through the usage of SDN, previous work addresses and answers some of WSN well-known problems: complexity of configuration and management, low flexibility and network resource underutilization. They propose architectures where the SDN paradigm is adapted to WSN, making SDN switches of sensor nodes and addressing the challenges of applying it to low rate and high latency wireless networks. Based on the OpenFlow [16] protocol concepts, Sensor OpenFlow [13] is one of such solutions, which achieves in-network processing through rules applied to packets pertaining to individual flows.

Another SDN solution for WSN is SDWN [6] that provides flow rules capable of performing actions regarding a packet after being applied to any particular set of bytes of its payload. As with SDWN, TinySDN [20] also relies on an underlying OS. Unlike Sensor OpenFlow and SDWN it fully separates control and data flows and, additionally, divides the network among end devices and controller nodes, allowing multiple controller nodes which not necessarily need to be sink nodes. Furthermore, TinySDN also allows the definition of rules composed by actions and values to be applied to specific flows.

All of the above-mentioned works successfully present the advantages of SDN to solve WSN problems they describe, however they do not address the difficulty of deploying a WSN application. What makes such a task a very demanding one is the necessity of developing customized application code for different platforms of sensor nodes without compromising application code that is already running on top of a WSN.

De Gante et al. [8] were the first, to our knowledge, to propose an SDN-based framework to ease the development, deployment and management of WSN applications. However, this architecture does not concern the data plane in a WSN. The developer is left with an automatically configured WSN infrastructure, but still needs to tackle embedded application development and node resource management after deployment.

In order to provide such capabilities, we believe it is necessary to develop a data plane protocol to handle application level activities in a similar way that SDN communication protocols do for the SDN control plane. This would exempt an application developer to be concerned with embedded systems programming and let development focus on high-level application logic, much like the solutions like TinyDB and Terra allow, but also with the advantages provided by SDN.

III. Framework specification

We have seen how the SDN paradigm can improve the tasks of WSN configuration and management, being an interesting solution to most of the problems encountered in the WSN application frameworks we studied. Therefore we present WARM, an SDN-based framework for developing and managing WSN applications. WARM makes use of the SDN paradigm not only to provide a flexible WSN infrastructure, easily configurable and manageable, but also to provide a flexible application layer for sensor nodes.

To achieve this, WARM is based on the concept of application tasks, which are hardware dependent routines whose instances can be scheduled in sensor nodes. These routines should implement lightweight processing algorithms, environmental sensing or actuator actions, each one representing a different capability of a sensor node. Furthermore, they can also make use of the provided SDN-based IEEE 802.15.4 stack in order to receive input data from other tasks or to send their output data as input for other tasks through the sensor network.

Tasks are designed to be platform dependent and to be implemented by hardware vendors with the goal of using a device's resources in order to perform common WSN activities. Examples of tasks would include sensing ambient temperature, light and humidity, drive a motor, activate a relay, compute an average, store data to an SD card or output it through a serial connection – important for enabling sink nodes to output gathered data to a network gateway.

Tasks are to be loaded to sensor nodes according to an application's foreseeable needs. They sit on top of the Framework Middleware, one of WARM's two architectural counterpoints. The Middleware is a piece of embedded software to be configured and loaded once to each of the nodes in a WSN. As depicted on Figure 1, WARM's architecture also includes a centralized controller, which comprises an SDN controller, an application controller we call the Framework Controller, and a web service API based on the Representational State Transfer (REST) software architecture [9].

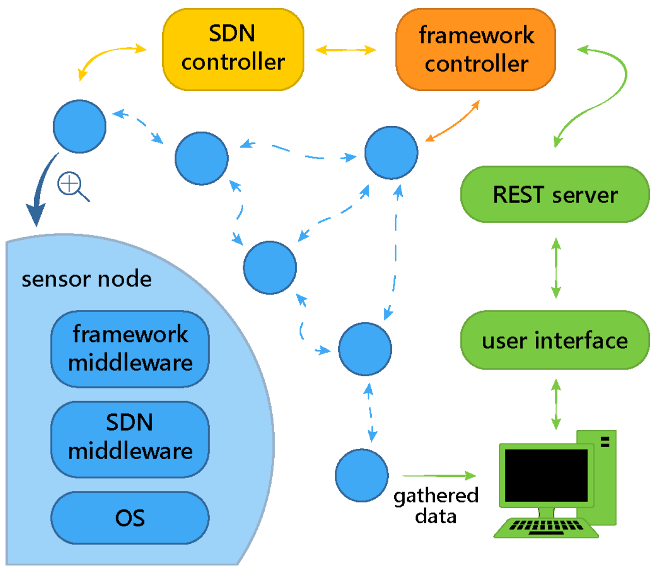

Fig. 1. Overall view of the system.

To configure and manage a WSN application, the Framework Controller must communicate with each node's Middleware. The Controller communicates with the purpose of scheduling task instances, querying nodes or tasks descriptions and retrieving status information. For this purpose, WARM was designed with an application data plane protocol. This protocol has a set of request and reply messages to be exchanged between Middleware and Controller, as well as a set of messages for data exchange among running task instances.

Concerning the user interface, WARM's functionalities include: to schedule new task instances for the sensor network to perform, and also to retrieve nodes and tasks descriptions and status. For instance, in order to schedule a new sensing task the user only needs to set an address to where the gathered data will be sent and a period, if it is a periodic task. Node descriptions made available by WARM include location, memory, operating system and existing tasks, while status include battery level and the number of executions of specific task instances.

The architecture proposed in Figure 1 assumes an SDN layer such as designed in TinySDN [20]. The layer provided by TinySDN focuses on the differentiation of data and control flow, and includes SDN-enabled sensor nodes and SDN controller nodes. This design allows WARM's Framework Controller to be built on top of SDN controller nodes and its Framework Middleware to be on top of every SDN-enabled sensor node. Besides, it addresses the overhead imposed to a WSN due to the presence of control flows, regarding to the increase of network delay and energy consumption. The role of the SDN controller is mainly to create and manage flows according to the SDN rules it will apply, which are provided by the application (in our case, the Framework Controller). The SDN-enabled nodes can thus behave both as end devices and as SDN switches. This behavior means the forwarding of data in and out of application layer (in this case, the Framwork Middleware) or to other nodes, as defined by the applied SDN rules regarding the flow to which data belongs.

WARM's Framework Middleware is responsible for managing application data. It can receive data from the network through the SDN layer, forward this data to schedule tasks or to be processed by tasks. It is also responsible for sending any output data to the SDN layer, so that it can be forwarded to the network. In order to manage this data, the Middleware depends heavily on the concept of SDN flows, basing the decision of how to process each received packet according to the flow it belongs to. The Middleware's architecture is centered on the concept of tasks. These are specific routines executed by sensor nodes, implemented through the Middleware's Task API. This model allows the implementation of tasks to acquire data through sensors, control actuators, perform data aggregation and many kinds of routines using other resources a sensor node may have.

IV. Framework Implementation

WARM's implementation can be divided on two main software bundles. The first bundle is the Framework Controller and REST application server, which are implemented with version 2.7.11 of the Python language, meant to run on top of a GNU Linux platform. The second is the Framework Middleware, an embedded application for sensor nodes running version 2.1.2 of TinyOS [12], implemented with the nesC language. Connecting both is the SDN network layer provided by TinySDN [20].

The REST application server is designed to run alongside the Framework Controller on a GNU/Linux base station connected to the WSN. It uses the Python API provided by the Framework Controller to provide a RESTful interface for this API. Its implementation is based on the Flask microframework, which allows the server to map each of the Controller's API functionalities to an URL path and its return values.

The Framework Middleware is implemented as a series of interconnected modules following the programming paradigm adopted by the nesC language. NesC restrains TinyOS application development to the implementation of modules and their respective configurations. The Middleware is composed of five modules: Scheduler, Receiver, Emitter, Protocol Processor and Task API. The last one is responsible for abstracting the functionalities of all the other Middleware components, decoupling them from task application logic in order to ease the task development process for hardware maintainers.

The Framework Controller is implemented using the SQLAlchemy Object Relational Mapper over an SQLite DBMS. The implemented databases are responsible for storing network topology and task schedules. By analysing this information, the Controller is able to assign SDN flows for scheduled tasks, requesting them to the TinySDN controller.

V. Experimental results and discussion

To evaluate WARM, we developed a simple web application to provide a graphical interface for the framework's RESTful API. This web application allows a user to query information about existing nodes, tasks and task parameters as well as schedule and cancel task instances in network nodes across a test WSN. It also displays a map showing the network sensor nodes and the way they are wirelessly connected to each other.

For our experiments, we setup a WSN of TelosB[1] motes in the simple topology depicted on Figure 2, simulated using the COOJA tool [18], a simulator distributed with version 3.0 of the Contiki operating system. The setup for this simulated network consisted of seven nodes, including: an SDN controller node, a Framework Controller node and five generic nodes. The Framework Middleware was loaded to each one of the generic sensor nodes present in this network. Along with it, a set of tasks implemented in order to exemplify common WSN application activities was also uploaded to the emulated motes. This set included tasks for sensing temperature and humidity, to turn a LED on and OFF, to calculate the average of input data received from other sensor nodes and to output network received data through a serial connection (useful for the sink node to output gathered data).

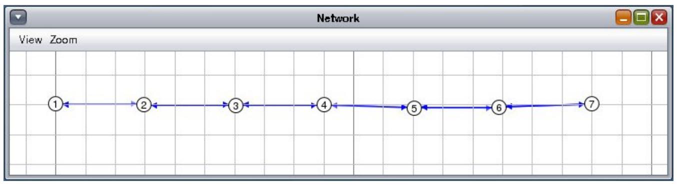

Fig. 2. Simulated WSN topology screenshot from the COOJA simulator tool

The first experiment consisted in measuring the amount of ROM and RAM occupied by the Framework Middleware in the TelosB mote we used for our tests. In order to measure the overhead represented by our set of loaded tasks, we first measured the amount of memory occupied by a Middelware configuration with no tasks loaded, compiled along with TinyOS and TinySDN's middleware. This setting resulted in a total of 4,954 bytes of occupied RAM and 27,258 bytes of ROM, representing respectively 49.54 % and 56.79 % of RAM and ROM in the device. The result with a configuration that included the additional set of tasks for our tests environment was of 6,256 bytes of RAM and 38,106 bytes of ROM, representing respectively 62.56 % and 79.39 % of RAM and ROM in the device. These results mean an overhead of 1,302 bytes of RAM and 10,578 bytes of ROM for our designated set of tasks. Table I compares the memory results obtained for WARM with the results presented Terra, TinyDB and Swiss QM, discussed on section II. Notice that WARM requires the least amount of ROM to execute, while it uses the largest amount of RAM when compared to the others.

[1]`http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf`

TABLE I
OCCUPIED RAM AND ROM BY STUDIED WSN FRAMEWORKS

WSN Framework Solution	RAM Usage	ROM Usage
WARM - Minimal	4,954 Kb	27,258 Kb
WARM	6,256 Kb	38,106 Kb
Terra - Minimal [5]	3,570 Kb	32,162 Kb
Terra [5]	3,580 Kb	48,286 Kb
TinyDB [17]	3,000 Kb	65,000 Kb
Swiss QM [17]	3,000 Kb	33,000 Kb

The second experiment intended to measure the overhead introduced by WARM's communication protocol. To achieve this, we recorded the elapsed time for exchanging protocol messages between the Framework Controller and the Framework Middleware. Table II presents the average time from ten samples obtained using a sensor node positioned in an average distance from the node running the Framework Controller. Notice that the COOJA simulator, which was used for taking these measurements, has a precision of 1 ms, causing standard deviations of 1 ms for most of the measured values. When observing results from Table II, it is worth noticing that sensing humidity takes about 104 ± 14 ms [15].

TABLE II
AVERAGED COMMUNICATION OVERHEAD DUE TO WARM'S MESSAGES.

Exchanged message pair	Average time (ms)
Node association request-reply	8 ± 1
Task description request-reply	5 ± 1
Task scheduling request-reply	4.9 ± 0.3
Task cancellation request-reply	4.8 ± 0.4
Task execution report request-reply	5 ± 1
Node status report request-reply	5 ± 1

This protocol overhead sets the cost for managing a WSN application deployed through WARM. As an example, scheduling a set of tasks implies in exchanging the appropriate protocol messages in order to notify each of the involved nodes. This management cost is not exclusive of WARM, since all of the studied WSN frameworks have their own characteristic cost for propagating messages through the network in order to manage it. TinyDB, for instance, disseminates several messages containing a single query to be executed, with each node broadcasting it to its children in case they were involved in executing the query. This requires the maintenance of network topology information through semantic routing trees [14], not very far from what WARM accomplishes through TinySDN.

SwissQM implements a protocol of its own in order to guarantee a reliable query distribution. Since its queries are earlier translated into small VM bytecode strings, SwissQM is allegedly more efficient than TinyDB [17], taking less messages to transmit one query. Unfortunately, neither TinyDB nor SwissQM provide detailed results of query distribution overhead. Terra [5], on the other hand, report an average of 6.76 seconds for disseminating the 24 messages of an application bytecode through a network comprised of 9 nodes, very close in size to the one present in our simulations.

All of the above mentioned works have similar restrictions regarding flexibility, requiring nodes to be pre-configured with a set of supported functionalities that will allow the deployment of runtime changes. While WARM seems to be better for managing node capabilities in a granular and individual basis, TinyDB and SwissQM appear to be more optimized for managing groups of nodes with similar behaviors. WARM could also achieve such optimizations if TinySDN provides multicast support, which would allow the scheduling of a task at several nodes at once. Terra, as a macroprogramming framework, is the most costly of the compared framework solutions, but also allows greater flexibility in case of applications where all nodes behave in similar ways.

The last experiment targeted user experience. First, when using the developed web application, it was possible to configure a WSN application for our simulated network using a web browser. The test application was configured to periodically sample temperature data from the environment, calculate the average and send it to a sink node that is responsible for outputting it through a serial connection. Second, the same simplicity was observed when modifying this WSN application to turn on one of the mote's LEDs in case the averaged temperature should surpass a value configured on the fly. Changing which nodes had the roles of data sink and of averaging sampled data was also achieved with the same ease, without the need to reprogram any sensor node.

These experiments together show that WARM is feasible, introduces little overhead, and makes programming simpler.

VI. Conclusions

We proposed WARM, an SDN-based framework that abstracts the whole process as the scheduling of tasks done through a RESTful API. This is done without hindering needed flexibility for implementing further customized applications. Such customization can be achieved through the development of tailored sensor node tasks using the API provided by the Middleware presented in our architecture.

Our work also considers the many resource constraints that characterize WSN, which we address through the use of the SDN paradigm. SDN is the foundation which allows WARM's Controller to monitor and balance resource load, as well as handle the typical operational failures a WSN may present.

Additionally, we implemented a full stack for WARM's proposed architecture, using the nesC and Python languages on top of TinyOS, TinySDN and GNU/Linux. This prototype was successfully evaluated for the configuration and management of an application running on a simulated WSN, using only a web application developed as a simple graphical user interface for WARM's RESTful API. Furthermore, memory usage and time overhead confirm WARM's feasibility.

For future work, we plan to experiment WARM with other network topologies and applications. We intend to enable remote addition and removal of tasks from deployed sensor nodes, as well as to port WARM to more platforms. We also consider providing support for the multiple distributed SDN controllers allowed by TinySDN.

VII. Acknowledgments

This work is partially funded by FAPESP research grant #2013/15417-4 and fellowship #2014/06479-9. C. B. Margi is supported by CNPq research fellowship #307304/2015-9.

References

[1] K. Aberer, M. Hauswirth, and A. Salehi. A middleware for fast and flexible sensor network deployment. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1199–1202. VLDB Endowment, 2006.

[2] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, Oct. 2010.

[3] A. Azzara, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano. PyoT, a macroprogramming framework for the Internet of Things. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 96–103. IEEE, 2014.

[4] D. C. Bormann, K. Hartke, and Z. Shelby. The Constrained Application Protocol (CoAP). IETF RFC 7252, Oct. 2015.

[5] A. Branco, F. Sant'anna, R. Ierusalimschy, N. Rodriguez, and S. Rossetto. Terra: Flexibility and safety in wireless sensor networks. *ACM Trans. Sen. Netw.*, 11(4):59:1–59:27, Sept. 2015.

[6] S. Costanzo, L. Galluccio, G. Morabito, and S. Palazzo. Software Defined Wireless Networks: Unbridling SDNs. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 1–6. IEEE, Oct. 2012.

[7] D. Culler, D. Estrin, and M. Srivastava. Overview of sensor networks. *Computer Magazine*, 37(8):41–49, 2004.

[8] A. De Gante, M. Aslan, and A. Matrawy. Smart wireless sensor network management based on software-defined networking. In *Communications (QBSC), 2014 27th Biennial Symposium on*, pages 71–75. IEEE, June 2014.

[9] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[10] FIWARE. FIWARE - open apis for open minds. http://www.fiware.org, 2014.

[11] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel. The power of software-defined networking: Line-rate content-based routing using openflow. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '12, pages 3:1–3:6, New York, NY, USA, 2012. ACM.

[12] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOS: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[13] T. Luo, H.-P. Tan, and T. Q. S. Quek. Sensor OpenFlow: Enabling Software-Defined Wireless Sensor Networks. *IEEE Communications Letters*, 16(11):1896–1899, 2012.

[14] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[15] C. B. Margi, B. T. de Oliveira, G. T. de Sousa, M. A. Simplício, P. S. L. M. Barreto, T. C. M. B. Carvalho, M. Näslund, and R. Gold. Impact of operating systems on wireless sensor networks (security) applications and testbeds. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6, Aug. 2010.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[17] R. Mueller, G. Alonso, and D. Kossmann. Swissqm: Next generation data processing in sensor networks. In *In CIDR'07*, 2007.

[18] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648, Nov 2006.

[19] Z. Shelby, S. Chakrabarti, and E. Nordmark. Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). IETF RFC 6775, Oct. 2015.

[20] B. Trevizan de Oliveira, C. Borges Margi, and L. Batista Gabriel. TinySDN: Enabling multiple controllers for software-defined wireless sensor networks. In *Communications (LATINCOM), 2014 IEEE Latin-America Conference on*, pages 1–6. IEEE, 2014.

[21] R. Want, B. N. Schilit, and S. Jenson. Enabling the internet of things. *Computer*, 48(1):28–35, Jan 2015.