TinySDN: Enabling Multiple Controllers for Software-Defined Wireless Sensor Networks

Bruno Trevizan de Oliveira
Escola Politécnica
Universidade de São Paulo
São Paulo – SP
Email: btrevizan@larc.usp.br

Cíntia Borges Margi
Escola Politécnica
Universidade de São Paulo
São Paulo – SP
Email: cintia@usp.br

Lucas Batista Gabriel
Escola Politécnica
Universidade de São Paulo
São Paulo – SP
Email: lucasbg@usp.br

***Abstract*—Software-Defined Networking (SDN) has been envisioned as a way to reduce the complexity of network configuration and management, enabling innovation in production networks. While SDN started focusing on wired networks, there were proposals specifically for Wireless Sensor Networks (WSN), but all of them required a single controller to be coupled to the sink. This paper presents TinySDN, a TinyOS-based SDN framework that enables multiple controllers within the WSN. It comprises two main components: the *SDN-enabled sensor node*, which has an SDN switch and an SDN end device, and the *SDN controller node*, where the control plane is programmed. TinySDN was designed and implemented to be hardware independent. Experiments were conducted on COOJA simulator, and results concerning delay and memory footprint are presented.**

I. Introduction

Software-defined networking (SDN) is an emerging paradigm that uses a logically centralized software to control the behavior of a network. It has been envisioned as a way to reduce the complexity of network configuration and management, enabling research and innovation in production networks. The first initiatives to implement SDN have targeted wired networks, but this paradigm raised the interest of the wireless networking community and enterprises working in the field of wireless and mobile communications have joined SDN-related initiatives [1]. In fact, there are proposals to use SDN in the context of wireless networks, such as SDN in substitution networks [2] and in capacity sharing [3].

Considering Wireless Sensor Networks (WSN), an interesting feature that could be achieved through SDN enabled devices is node and resource management. For instance, when the controller determines a route to be used, it could consider the energy available in a given node (or set of nodes) to determine which route will provide the best network lifetime. Also, the controller could determine when a given sensor node fails, since it does not receive any messages from them. Furthermore, WSN nodes are usually considered disposable and cheap devices, which could be deployed for a specific task. But consider *iCities* or smart cities, where sensor nodes should collect, process and transmit different types of data for different applications [4]. If these sensor nodes and other devices collecting data could be managed by the SDN paradigm, one could achieve a much better usage of the underlying infrastructure through dynamic node retasking and routing.

Software-defined wireless sensor networks (SDWSN) approaches include Flow-Sensor [5], Sensor OpenFlow [6] and SDWN [1]. However, these works do not address common WSN characteristics, such as possible disruption and delay in communication, low energy supply and reduced data frame length. Furthermore, typical devices in wireless multi-hop ad hoc networks and WSN have only one radio that either transmit or receive in one given frequency at a given time, and thus in-band control is required.

We present TinySDN, a TinyOS-based SDN framework that enables multiple controllers within the WSN. The main challenges addressed are: (i) in-band control, opposed to SDN implementation in wired networks that can leverage from out-band control; (ii) higher communication latency; (iii) smaller link layer frames; and (iv) limited energy supply.

The SDN paradigm considers the separation of data and control planes. But, despite recent efforts [7], typical WSN devices have only one radio that either transmit or receive in one given frequency at a given time. Thus in wireless networks, data and control planes must share the same communication link and available bandwidth. This in-band control limits the amount of data that can be forwarded through a given link, and potentially increases the delay. Therefore, our work is focused on reducing the control traffic on the WSN. Moreover, because of the in-band control it is important to differentiate control flow and data flow.

Another issue that must be considered is the limited bandwidth provided by IEEE 802.15.4 standard. If one considers the controller on a wired network, the communication latency is increased by the average 250 Kbps and the multiple hops to be traversed until reaching the sink and then the controller. When placing the controller on the sink, as done in [5], [6] and [1], latency decreases compared to the previous scenario, but still yields high communication latency. In order to decrease even more the overall latency we explore locality, as suggested by Schmid and Suomela [8], by using multiple controllers on the WSN and thus placing one of them closer to end nodes.

Furthermore, the SDN control information that needs to be exchanged should fit in one link layer frame, and the IEEE 802.15.4 standard determines that the frame is at most 127 bytes. Lastly, the increase in control data transmission will lead to an increase in the energy consumption, which is a concern given the limited energy supply available in WSN. We address these issues by avoiding the use of end-to-end delivery mechanisms and dealing with possible packet loss.

The main contributions in this work include the design and implementation of TinySDN, a hardware independent TinyOS-based SDN framework that enables multiple controllers within the WSN. It comprises two main components: the *SDN-enabled sensor node*, which has an SDN switch and an SDN end device, and the *SDN controller node*, where the control plane is programmed. Experiments were conducted on COOJA simulator, and results concerning delay and memory footprint show TinySDN feasibility.

This paper is organized as follows. The main approaches for SDWSN are discussed in Section II. Then TinySDN architecture is presented in Section III, while implementation and experiments are described in Section IV. Final considerations and future work are presented in Section V.

II. Related Work

Flow-Sensor [5] was the first initiative to apply the SDN concept to WSN. This architecture aims to provide sensor nodes with OpenFlow [9] basic features. The authors claim that flow-sensor nodes are much more reliable in comparison with typical sensor nodes since data packets, control packets and the sensor nodes themselves can be easily monitored, regulated and routed whenever required. This enables management schemes from a cost, energy-effectiveness and overall network performance perspective. Moreover, the paper presents simulation results that show that applying the SDN concept is suitable for WSN, since a flow-sensor based network performed similarly or better than a typical sensor network, in terms of the total number of packets generated for different scenarios (varying topologies, density and transmission power).

The Sensor OpenFlow proposal [6] has two components: an architecture featuring a clear separation between data plane and control plane, and a core component that standardizes the communication protocol between the two planes. Besides providing the basic OpenFlow functionalities, Sensor OpenFlow enables dynamic sensor tasking through the control plane. The authors address resource underutilization and counter-productivity by promoting sharing of hardware resources between different applications and reuse of implemented functionalities accelerating prototyping, respectively.

SDWN [1] is a more complete proposal that includes all the features present in Sensor OpenFlow and provides other important features for WSN, such as in-network data aggregation, duty cycles configuration, flexibility to define rules and actions to enable cross-layer optimizations. SDWN architecture defines two types of devices: *generic nodes*, in which the flow tables are instantiated, as well as applications that manage sensing tasks; and a *sink node*, composed of two different modules, one being a device to handle the communication with *generic nodes* and another being a Linux-based embedded system that combines the tasks of controller and network virtualizer.

While there are interesting proposals to apply the SDN paradigm to WSN, none of them enables multiple controllers deployment. Furthermore, most of them do not discuss or take into account differentiation between control flow and data flow. Finally, these proposals are not available to devices running TinyOS [10], the most popular operating system for networked applications in wireless embedded systems compatible with widely used WSN platforms.

TinyOS is an open source event-driven operating system designed for low-power and very resource-constrained wireless devices, such as sensor networks and ubiquitous computing platforms. It provides useful software abstractions of the underlying device hardware, and includes a component-based architecture written in the nesC programming language [11].

III. TinySDN Design

TinySDN is an architecture (along with its system implementation) to enable multiple controllers for software-defined wireless sensor networks and the SDN paradigm in TinyOS compatible platforms. It transforms the wireless sensor node into a component comprised by an SDN switch and an SDN end device, which we call *SDN-enabled sensor node*. Then the control plane is programmed through an SDN controller hosted in network components that we call *SDN controller nodes*.

Figure 1 depicts the two types of specified nodes in TinySDN architecture: *SDN-enabled sensor node* and *SDN controller node*. Each *SDN-enabled sensor node*, where the data plane is executed, connects through multi-hop wireless communication to an *SDN controller node*, where the control plane logic is executed, allowing the interaction between the two planes.

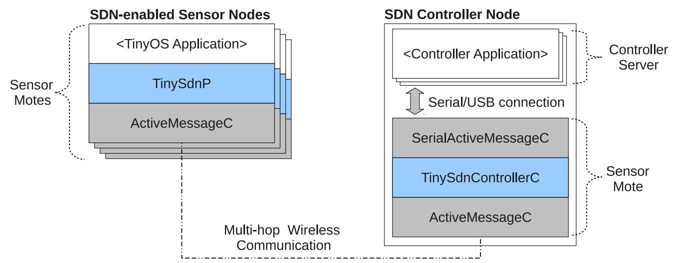

Fig. 1. Architecture Components

Regarding the end-to-end reliability, we consider that the hop-to-hop delivery mechanism of the IEEE 802.15.4 standard [12] is enough and packet loss may occur. End-to-end security services could be achieved using WSN-ETESec [13].

Next, we describe the two specified types of nodes, the specification of flows and actions, as well the strategies adopted for the *SDN-enabled sensor nodes* in order to find *SDN controller nodes* and to establish communication with them, and how the network topology information is collected.

A. SDN-enabled Sensor Node

This is the component that runs on sensor nodes. As discussed in [6], end devices are considered peripheral to SDN and hence out of the scope of OpenFlow [9], the main SDN project nowadays. On the other hand, sensor nodes behave like end devices by generating data packets to transmit data collected by sensing, in addition to merely forwarding data as SDN switches do. Thus, *SDN-enabled sensor node* is a type of node that plays both roles: SDN switch and SDN end device.

Each *SDN-enabled sensor node* must find an *SDN controller node* to join and then receive flow specifications and perform flow request when necessary. As shown in Figure 1 (left), *SDN-enabled sensor node* is split into three parts: *TinyOS Application*, *TinySdnP* and *ActiveMessageC*.

The *TinyOS Application* portion is the equivalent to the end device; it generates data packets and then places them on the network using the programming interface provided by the TinySDN component. It is written by the network designer (or programmer/user), according to the WSN application.

The *TinySdnP* is the main component of TinySDN, which is responsible for checking if a received packet matches a flow in the `flow table` and then performs the related action, or otherwise sends a flow setup request (named `packet-in`) to an *SDN controller node*. Consequently, it is responsible for performing `flow table` update when receiving a flow setup response (named `packet-out`).

The *ActiveMessageC* is a TinyOS component that manages and provides programming interfaces to interact with the radio module of the sensor node. It is used by the TinySDN to perform all tasks related to wireless communication, such as data/control packets forwarding and topology information collection.

B. SDN Controller Node

SDN Controller Node is the node (or nodes in case of multiple controllers) that performs SDN controller tasks, i.e., applies definitions of applications, creating and managing network flows. As can be observed in Figure 1 (right), it is composed of two different modules: sensor mote module and controller server module.

Sensor mote module: It runs on a sensor mote and is responsible for communicating with *SDN-enabled sensor node* using *ActiveMessageC*. It uses *SerialActiveMessageC* (the serial communication component of TinyOS) to forward received messages to the *controller server module* and receive messages to be sent to the network from the *controller server module*. The *TinySdnControllerC* portion adapts messages and manages this communication.

Controller server module: It contains the control plane logic, i.e., hosts controller applications and manages network flows and topology information.

C. Specification of Flows and Actions

The *SDN-enabled sensor nodes* are responsible for classifying packets in different flows and performing the corresponding actions. Two actions are specified in TinySDN, "forward" and "drop", similar to OpenFlow [9]. The "forward" action consists in performing packet forwarding to a specified next hop. The "drop" action indicates that a packet should be dropped.

There are two types of flows: control flows that are used for control traffic between *SDN-enabled sensor nodes* and *SDN controller nodes*, and data flow that are used for applications data traffic. Flows are specified as entries in the flow tables, where each entry is composed of four fields: specific identification field; *Action* field, which specifies the action to be performed; *Value*, field that contains a specific value related to action (e.g., in case of "forward" action it specifies the next hop); and *Count*, field incremented when each packet is matched to the flow entry, providing statistics.

In `data flow table` the identification field is *Flow ID*, which is used to identify the flows and for data packet matching. Table I contains an example of the `data flow table` and some entries.

TABLE I. DATA FLOW TABLE EXAMPLE.

Flow ID	*Action*	*Value*	*Count*
1	Drop	N/A	100
2	Forward	5	10
101	Forward	10	27

In the case of `control flow table`, the identification field is *Destination Node ID*, which is used to identify destination nodes of control flows and for control packet forwarding. Table II contains an example of the `control flow table`.

TABLE II. CONTROL FLOW TABLE EXAMPLE.

Destination Node ID	*Action*	*Value*	*Count*
0	Forward	4	5
1	Forward	4	2
7	Forward	6	2

D. Finding a Controller and Communicating with It

The first task of an *SDN-enabled Sensor Node* at network startup is to find an *SDN Controller Node* and to assign to it. For this discovery process, TinySDN runs the collection tree protocol (CTP) [14] in the background. CTP is a widely used protocol in multi-hop TinyOS-based applications. We selected this protocol considering two goals:

- **Hardware independence –** CTP uses `ETX` value as metric. This value is derived from the TinyOS Four-Bit Link Estimator Component [15], a component that estimates the link quality between single-hop neighbors from the amount of delivered or lost messages, instead of using Received Signal Strength Indication (RSSI), a hardware-dependent and specific feature.
- **Multiple SDN controllers –** CTP uses a tree structure to deliver data sensed over the network to a data sink, named root. When multiple roots are announced each node joins the nearest root, without being aware that there is more than one of them, building a collection tree for each root. The *SDN controller nodes* of TinySDN are set as CTP roots, which enables the *SDN-enabled sensor node* to find them.

In CTP, at network startup, the roots take `ETX`=0 and other nodes take `ETX`=∞. Each node discovers its neighborhood by broadcasting `beacon packets` and receiving answers, and it considers the lowest ETX neighbor its `parent`. Then, each node updates its `ETX` value following equation below, where *le(node)* is the function that provides link quality estimation with a given neighbor node.

$$\texttt{ETX} = [\texttt{parent ETX}] + le(\texttt{parentID})$$

Once the CTP is stabilized, the *SDN-enabled sensor node* can send an assignment request to an *SDN controller node* (configured as CTP-root) through CTP path, forwarding the request to its `parent`. Upon receiving the request, the *SDN controller node* sends a specification of flow control leading to it to the requesting *SDN-enabled sensor node*.

E. Network Topology Information Collection

TinySDN Network Topology Information Collection comprises two steps: (i) each *SDN-enabled sensor node* recognizes its neighbors and measures the link quality between them, and (ii) it sends this information to the *SDN controller node* through a CTP path or control flows, if already specified. These steps are executed periodically, according to the user/network programmer configuration.

In order to recognize the neighborhood, *SDN-enabled sensor nodes* broadcast `beacon packets` and wait for responses; upon receiving a response, each response sender is added to the `neighbor table` along with its link quality.

Table III contains an example of `neighbor table`. The Neighbor ID column contains neighbors *node id* and the Link Quality column contains neighborhood measured links quality. The lower Link Quality value represents the better quality. We decided to adopt this representation to make the link representation easier in weight graph data structure.

TABLE III. NEIGHBOR TABLE EXAMPLE.

Neighbor ID	*Link Quality*
5	10
8	50
9	12

TinySDN implements two methods for link quality measurement: Link Estimator [15] (using TinyOS component) and specific platform RSSI. Specific platform RSSI provides a much more accurate link quality measurement; on the other hand Link Estimator is hardware-independent.

It is worth noting that RSSI is a measurement of the power measured in the received radio signal, i.e., a link quality measurement from the receiver's point of view, whereas Link Estimator provides a link quality estimation from the sender's point of view. It must be considered by the controller for the network topology information collection and its representation in data structure, e.g., directed graph.

IV. IMPLEMENTATION AND EXPERIMENTS

We developed a full implementation of TinySDN in nesC [11], the language adopted in the TinyOS programming framework. Validation tests succeeded when conducted using the TelosB mote [16] as platform. The SDN controller node was implemented as standalone software (also in nesC language) specifically to provide a proof of concept and experimentation.

The experiments were executed as simulations on COOJA [17], a cross-level sensor network simulator that adopts a hybrid approach being able to simulate the network level (through the environment implemented in Java) and the operating system level (based on native sensor node program, integrating it with the network simulation environment using Java Native Interface). It allows the use of the platform compiled code (i.e., the exactly same binary code to be uploaded on physical device) by emulating actual TelosB [16] sensor running on the TI MSP430 emulator.

Since COOJA runs the binaries (i.e., processor instructions) produced by a cross-compiler, it can emulate both Contiki OS [18] and TinyOS applications programs. Thus, we decided to adopt this environment because of the simplicity of topologies deployment and because of the use of emulation, allowing testing of system features in a resource similar to a real sensor node.

The main goal is to measure and compare the time sender nodes hosting TinySDN take to established data flows and to send a message to the sink node, both in a software-defined wireless sensor network with a single SDN controller and multiple controllers. Since we tested small topologies, which does not require high processing power or high memory capacity, in our test scenarios the controller nodes were also hosted on TelosB mote.

Given the fact that nodes must boot when the simulations start and it takes 4.166 seconds for the first node to boot, we subtracted this value from all the time measurements (with the exception of latency).

A. Test Scenarios Description

In our experiments, we considered three different scenarios with seven sensor nodes hosting TinySDN (i. e., they are *SDN-enabled sensor nodes*), where two of them are sender nodes (i.e., they generate the data packets to be sent to the sink node), four just perform forwarding tasks and one node acts as a sink. The number of *SDN controller nodes* is variable according to the scenario proposed. Scenarios `a` and `b` depict a single controller software-defined wireless sensor network, and scenario `c` illustrates the multiple controller case (two SDN controllers, in case). These test scenarios are depicted in Figure 2, in which nodes `1` and `7` are the senders, and `node 4` is the sink.

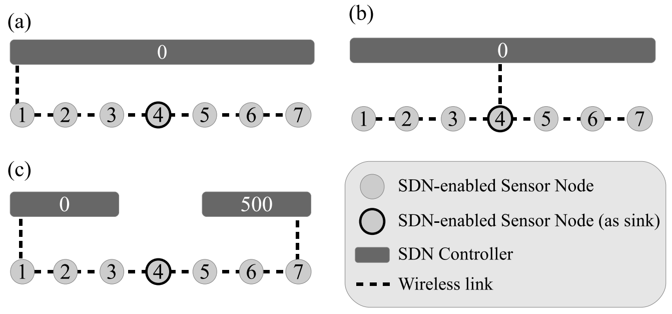

Fig. 2. Test scenarios.

In scenario `a` there is a single SDN controller, directly connected to `node 1` (i.e., a single hop to reach it). Thus, although both sender nodes are equidistant from the sink node, it is expected that `node 1` delivers its first message before `node 7`, since it is closer to the controller and therefore tends to receive flow specifications earlier.

Scenario `b` is very similar to scenario `a`, the only difference being that the sender nodes are equidistant from the single SDN controller node, which is connect to `node 4` (the sink node). It is expected that this change balances the time of flows establishment for the sender nodes and consequently the time for the first message delivery.

In scenario `c` there are two SDN controllers (`node 0` and `node 500`) that together cover all the seven SDN-enabled sensor nodes, dividing the network into two subdomains. During the finding controller process, the nodes `1`, `2` and `3` are assigned to the SDN controller `node 0`, while the nodes `4`, `5`, `6` and `7` are assigned to SDN controller `node 500`. It is worth noting that the `node 4` could join either SDN controller `node 0` or `node 500`, but CTP decided that it was "closer" to `node 500`. This scenario represents one possible multiple controllers scenario.

Concerning the memory footprint, we used the *MSP430-size* and *MSP430-ram-usage* tools [19] for measuring, respectively, programmable flash (ROM) and RAM usage. Thus it was possible to compare the memory usage of our scenario test application with a conventional protocol (in case, the CTP) with this same application over TinySDN.

B. Results

Figure 3 contains a chart that shows differences in time to controller assignment for sender nodes (considering the control flow specification to the controller in the SDN-enabled sensor nodes).

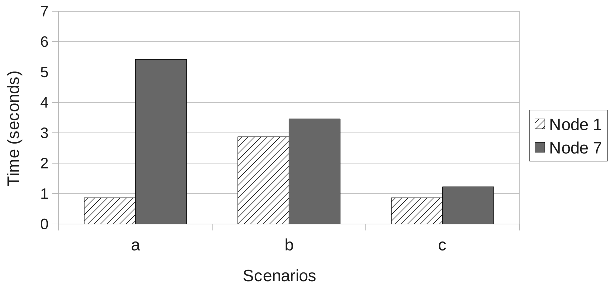

Fig. 3. Time to controller assignment.

In scenario `a`, `node 1` took less than 1 second while the `node 7` took more than 5 seconds. As expected, scenario `b` presents a more balanced result, `node 1` took just under 3 seconds and `node 4` approximately 3.5 seconds. However, as can be observed in Table IV, these both single SDN controller scenarios present approximately the same average to controller assignment, just under 3.3 seconds.

In scenario `b`, theoretically nodes 1 and 7 should complete controller assignment almost at the same time. However, in the COOJA simulation it was noted that `node 1` booted before `node 7`. Thus, `node 1` started sending messages before. This also occurred in the results for the scenario `c`.

TABLE IV. AVERAGE TIMES TO CONTROLLER ASSIGNMENT AND FIRST DELIVERED PACKET (IN SECONDS).

	Scenario		
Event	A	B	C
Controller assignment	3.26	3.25	1.07
First delivered packet	4.17	3.5	2.36

According to the results presented in Figure 3, the time to controller assignment to multiple controllers scenario took about 1 second to occur, i.e., less than half time if compared to single SDN controller scenarios, which is due to controllers distribution.

Figure 4 depicts timings for the first packet delivered from both sender nodes for all scenarios. The results are similar to the time to controller assignment. For scenario `a`, `node 1` is almost three times faster than `node 7`, as expected, since it performed the controller assignment before and it is closer to the single SDN controller. This trend also occurs in `b` and `c`.

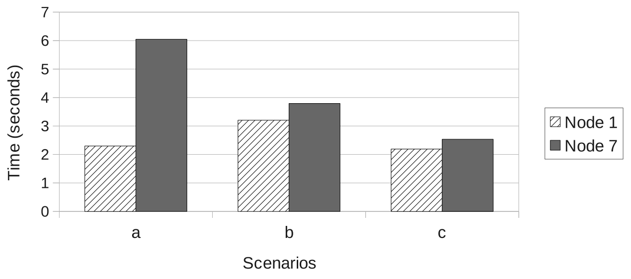

Fig. 4. Time to first packet delivered.

In order to compare the proposed scenarios of software-defined wireless sensor networks with traditional WSN, we measured the times to first delivered assignment request packets to the SDN controller nodes (i.e., the first delivered packet via CTP path), which is presented in Table V.

TABLE V. TIME TO RECEIVE THE FIRST CTP PACKETS BY EACH OF SDN CONTROLLER NODE (IN SECONDS).

	Scenario		
Controller node ID	A	B	C
0	0.84	0.89	0.84
500	-	-	1.21

Considering the average of these values (0.97 second), the scenario that presented the best performance with a single SDN controller (scenario `b`), takes on average about 3.5 times longer than the CTP to deliver the first packet. Scenario `c` takes about 2.4 times longer. This disadvantage is expected since TinySDN uses CTP to find the controllers.

We also measured latency for a given node to send a packet to the sink both with CTP and TinySDN after the establishment of data flows. CTP and scenarios `a`, `b` and `c` presented average latency of approximately 33 milliseconds, which indicates that TinySDN packet matching process does not introduce considerable delay when compared to a traditional protocol, since the longest delay is concentrated at the network startup.

Regarding the memory footprint of TinySDN, Table VI contains the measured results. It was observed that the test application over TinySDN requires 52.89% more RAM and 10.53% more ROM compared to the test application over CTP only. The test application over TinySDN uses (in total, including TinyOS components) 31.68% of RAM and 51.96% of ROM available on the used device [1], which leaves considerable memory to allocate other WSN applications running on the sensor nodes. Therefore, the memory overhead introduced by TinySDN does not hinder other features related to the WSN application, while enabling the achievement of flexibility in

[1]TelosB mote platform has 48 KBytes of programmable flash (or ROM) and 10 KBytes of RAM.

communication provided by the SDN paradigm. Still, it is possible to reduce CTP memory footprint instantiated in TinySDN by improving the integration between their implementations.

TABLE VI. MEMORY USAGE OF A CONVENTIONAL WSN PROTOCOL AND TINYSDN (IN BYTES).

Protocol	**RAM**	**ROM**
CTP	2072	22562
TinySDN	3168	24940

V. FINAL CONSIDERATIONS AND FUTURE WORK

We presented TinySDN, a multiple controller TinyOS-based architecture and hardware independent system for software-defined wireless sensor networking nodes. TinySDN enables the achievement of flexibility in communication provided by SDN paradigm, while its introduced memory overhead does not hinder other features related to the WSN application.

The experiment results indicate that, compared to CTP, TinySDN does not introduce considerable delay in the forwarding task due to packet matching after data flows specification. Delay observed in experimental scenarios affects only the first generated packets, that must wait until data flows are specified on sensor nodes.

Given the importance of energy consumption, as future work, we intend to measure and model the energy consumption of TinySDN, as well as to extend the experimental scenarios to include mobility and integration with different types of networks. Furthermore, we plan to add more action types in TinySDN, such as data plane tasks, concerned mainly to in-network processing/data aggregation as proposed in SDWN paper [1].

Related to the implementation, we are currently working on a manager application server that provides APIs to facilitate the development of new applications for TinySDN controller nodes, to replace the standalone controllers used nowadays. Once that is achieved and tested, we intend to make TinySDN publicly available.

ACKNOWLEDGMENT

The authors would like to thank the Laboratory of Computer Networks and Architecture (LARC-USP) for providing the necessary infrastructure to develop this work. This work is partially funded by São Paulo Research Foundation (FAPESP) under grants #2013/15417-4 and #2014/06479-9.

REFERENCES

[1] S. Costanzo, L. Galluccio, G. Morabito, and S. Palazzo, "Software defined wireless networks: Unbridling sdns," in *Proceedings of the 2012 European Workshop on Software Defined Networking*, ser. EWSDN '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/EWSDN.2012.12

[2] D. Venmani, Y. Gourhant, L. Reynaud, P. Chemouil, and D. Zeghlache, "Substitution networks based on software defined networking," in *Ad Hoc Networks*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, J. Zheng, N. Mitton, J. Li, and P. Lorenz, Eds. Springer Berlin Heidelberg, 2013, vol. 111, pp. 242–259. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36958-2_17

[3] M. A. S. Santos, B. T. de Oliveira, C. B. Margi, B. Nunes, T. Turletti, and K. Obraczka, "Software-defined networking based capacity sharing in hybrid networks," in *Proceedings of Capacity Sharing Workshop (CSWS'13), In conjunction with ICNP'13*, May 2013.

[4] M. Naphade, G. Banavar, C. Harrison, J. Paraszczak, and R. Morris, "Smarter cities and their innovation challenges," *Computer*, vol. 44, no. 6, pp. 32–39, June 2011.

[5] A. Mahmud and R. Rahmani, "Exploitation of openflow in wireless sensor networks," in *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, vol. 1, 2011, pp. 594–600.

[6] T. Luo, H.-P. Tan, and T. Q. S. Quek, "Sensor openflow: Enabling software-defined wireless sensor networks." *IEEE Communications Letters*, vol. 16, no. 11, pp. 1896–1899, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/icl/icl16.html#LuoTQ12

[7] A. Sabharwal, P. Schniter, D. Guo, D. W. Bliss, S. Rangarajan, and R. Wichman, "In-band full-duplex wireless: Challenges and opportunities," *IEEE JSAC Special Issue on Full-duplex Wireless Networks, accepted.*, 2014. [Online]. Available: http://arxiv.org/abs/1311.0456

[8] S. Schmid and J. Suomela, "Exploiting locality in distributed sdn control," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 121–126. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491198

[9] 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.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Notices*, vol. 35, no. 11, pp. 93–104, 2000.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/781131.781133

[12] IEEE Standard, "IEEE 802.15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs)," 2006.

[13] B. T. OLIVEIRA and C. B. MARGI, "Wsn-etesec: End-to-end security tool for wireless sensor networks," http://www.larc.usp.br/~cbmargi/wsnetesec, 2012.

[14] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. New York, NY, USA: ACM, 2009, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/1644038.1644040

[15] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis, "Four bit wireless link estimation," in *Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets VI)*, 2007.

[16] MEMSIC, "Telosb datasheet," http://www.memsic.com/userfiles/files/DataSheets/WSN/telosb_datasheet.pdf, 2011.

[17] 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*, Nov 2006, pp. 641–648.

[18] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[19] MSPGCC Development Team, "Gcc toolchain for msp430," http://sourceforge.net/projects/mspgcc/, 2013.