SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks

Laura Galluccio*, Sebastiano Milardo†, Giacomo Morabito*, Sergio Palazzo*
*University of Catania, Catania, Italy
†CNIT Research Unit Catania, Catania, Italy

***Abstract*—In this paper SDN-WISE, a Software Defined Networking (SDN) solution for WIreless SEnsor networks, is introduced. Differently from the existing SDN solutions for wireless sensor networks, SDN-WISE is stateful and pursues two objectives: (i) to reduce the amount of information exchanged between sensor nodes and the SDN network controller, and (ii) to make sensor nodes programmable as finite state machines so enabling them to run operations that cannot be supported by stateless solutions. A detailed description of SDN-WISE is provided in this paper. SDN-WISE offers APIs that allow software developers to implement the SDN Controller using the programming language they prefer. This represents a major advantage of SDN-WISE as compared to existing solutions because it increases flexibility and simplicity in network programming. A prototype of SDN-WISE has been implemented and is described in this paper. Such implementation contains the modules that allow a real SDN Controller to manage an OMNeT++ simulated network. Finally, the paper illustrates the results obtained through an experimental testbed which has been developed to evaluate the performance of SDN-WISE in several operating conditions.**

I. Introduction

In the early 2000s micro-electro-mechanical systems (MEMS), wireless communications and digital electronics have reached the maturity level needed to develop tiny, low-cost, low-power wireless sensor nodes able to wirelessly communicate with each others without a pre-deployed infrastructure, i.e. to form what are commonly referred to as *wireless sensor networks* (WSN)s [8]. Driven by the promise that WSNs would have produced a radical impact in several application scenarios, in the last decade the networking research community has devoted an immense effort to the study of WSNs and the definition of appropriate solutions for them. While such effort has resulted in a deep understanding of the WSN related matter, the expected large scale deployment of WSNs has not fully happened till today.

The reasons of the slow commercial take off of WSNs are multifold. Nevertheless, at the very basis there is a technical reason: WSNs are characterized by profoundly different requirements depending on the specific application and deployment scenario. Accordingly, as widely recognized [8], there is not something like a *one-fits-all* solution for WSNs. Instead, there is a plethora of vertical application-specific solutions that have resulted in extremely fragmented context and market.

The above problem can be overcome by making WSNs *programmable* and thus, there has been significant research effort devoted to design programmable WSNs [9]–[11]. However, in most current real-world WSN deployments, programming is typically very tightly related to the operating system, requiring the application developers to focus on intensive low-level details rather than on the application logic.

The *Software Defined Networking* (SDN) paradigm and OpenFlow [3], which currently is the most popular instance of SDN, have been recently proposed to solve analogous issues in the wired domain [1]. In OpenFlow the network nodes handle incoming packets as specified in the so-called *Flow Table*. Each entry of the Flow Table is related to a *flow* and is composed by three sections: (i) a *matching-rule* which specifies the values of the header field that must be found in the packets belonging to the flow; (ii) the *action* that must be executed on the packets of the flow (e.g., drop, forward to, etc.); and (iii) some statistical information about the flow. If the Flow Table does not contain any entry specifying how to deal a certain packet, the node sends a request to a software entity called *Controller* that has a high level abstraction of the network elements. The Controller can run on a remote server in a (logically) centralized manner. The Controller replies with information required to fill a new Flow Table entry for handling the packet.

In this way, OpenFlow clearly separates (even *physically*) the *data plane* from the *control plane* and delivers a network

- which is easy to configure and manage,
- which can *evolve* because, in principle, new services and management policies can be introduced in the network as simply as it is to install a new software on a PC [1], [2],
- in which a given network node can be replaced with another produced by any vendor, so freeing the operator from the vendor lock-in and allowing to use commodity hardware.

As a result, rarely the interest in a new networking paradigm has increased at such a pace as it is happening for SDN. Most network operators are running pilot experimentations of OpenFlow networks, manufacturers are producing OpenFlow compliant network equipment, and the research community (both academic and industrial) is involved in a vast amount of SDN-related R&D activities. A quick look at the list of members of the *Open Networking Foundation*, an organization promoting the development of SDN-related standards, suffices to understand that this hype has spread to the wireless domain, as well.

A few works have recently appeared that are aimed at

extending the SDN concepts to wireless sensor networks (WSNs) and other wireless personal area networks [4], [5]. The above papers represent important contributions to the SDN literature as they provide convincing motivations for the extension of the SDN paradigm to the WSN and W-PAN domains and offer a new and interesting perspective of the SDN paradigm but have a few shortcomings:

- protocol details, which are fundamental for the correct operations of the network, are not provided;
- no performance evaluations of the proposed solutions have been carried out.

In this paper we overcome the above shortcomings and go beyond the state of the art literature by defining a *stateful* SDN solution for wireless sensor networks[1] in line with a recent proposal by Bianchi et al. [6]. We call such solution *Software Defined Networking for WIreless SEnsor networks* (SDN-WISE).

The rest of this paper is organized as follows. In Section II we provide a survey of the related work. In Section III an overview of SDN-WISE is given. The details of the major features of the proposed solution are explained in Section IV, whereas, in Section V we describe the SDN-WISE prototype we have developed. Performance of SDN-WISE are evaluated experimentally in Section VI. Finally, conclusions are drawn in Section VII.

II. Related work

Recently, solutions have been proposed that extend the OpenFlow approach to the wireless sensor networks domain.

In [4] the technical challenges that must be faced to extend the OpenFlow approach to WSNs are identified and then the *Sensor OpenFlow* solution is proposed. Differently from traditional OpenFlow, Sensor OpenFlow supports in-network packet processing and various types of addressing defined for WSNs. In [12] the Sensor OpenFlow approach is integrated with other WSN programming techniques. In [5], *Software Defined Wireless Networking* (SDWN) is introduced. When compared to Sensor OpenFlow, SDWN offers a more flexible specification of the rules to classify packets, i.e., flow matching can consider any part of the packet, and supports the use of duty cycle to achieve energy efficiency in WSNs.

By introducing SDN-WISE we go beyond the above works in the following way. We define a complete architecture which allows software developers to implement their Controllers using any programming language of their choice. Also, SDN-WISE introduces a software layer which allows several virtual networks to run on the same physical wireless sensor or W-PAN network, similarly to what FlowVisor does in OpenFlow networks. Furthermore, SDN-WISE provides tools for running a real Controller in an OMNeT++ simulated network, analogously to Mininet [7].

Furthermore, as proposed in [6] for the wired domain, SDN-WISE defines simple mechanisms for the definition and handling of the Flow Table that make SDN-WISE stateful as compared to traditional OpenFlow which is stateless. In this way WSN nodes can be programmed as finite state machines which can be helpful to reduce the signaling between nodes and Controller and allow to implement policies that cannot be supported in a stateless manner.

Finally, we have implemented a complete prototype of SDN-WISE which we have extensively tested in our laboratories and made the source code publicly available at http://www.diit.unict.it/users/gmorabi/sdn-wise/.

To the best of our knowledge, this is the first real implementation of an OpenFlow-like solution for WSNs.

[1]SDN-WISE applies to wireless sensor and actor networks as well; however for the sake of readability in the rest of the paper we refer to WSNs only

III. SDN-WISE overview

In this section we provide an overview of the SDN-WISE solution. More specifically, we will first briefly give the requirements to be satisfied in the SDN-WISE design; then we will provide an overview of the SDN-WISE technical approach.

A. Requirements

Requirements for extending the SDN paradigm to WSNs have been already analyzed in [4] and [5]. Such requirements are the obvious consequence of the features of WSNs which are significantly different from those of wired networks. In fact, WSNs are characterized by low capabilities in terms of memory, processing, and energy availability. Furthermore, WSNs applications are typically non demanding in terms of datarate. Therefore, SDN-WISE must be efficient in the use of sensor resources, even if such efficiency will result in lower datarate.

In order to be energy efficient, SDN-WISE supports *duty cycle* [13], that is the possibility to periodically turn off the radio interface of a sensor node, and *data aggregation* [14]. These features were neglected in OpenFlow wired scenarios.

Furthermore, the interactions between sensor nodes and Controllers must be reduced as much as possible to achieve system efficiency. In this context, some level of programmable control logic in the sensor nodes may enable them to take decisions without interacting with the Controller when local information only is needed. This however, requires the introduction of a *state* whereas the standard OpenFlow instance of SDN is stateless [6].

Furthermore, since WSNs are intrinsically data-centric, several solutions have been proposed that make network protocols aware of the packet content [15]. Accordingly, SDN-WISE nodes can handle packets based on the content stored in their header and payload. Also, in OpenFlow packets are classified based on the equality between a certain field in the packet header and a given string of bytes; differently from that, in SDN-WISE such classification can be done based on other and more complex relational operators, e.g., *higher than*, *lower than*, *different from*, etc. Finally, the data-centric nature of WSNs involves another significant difference between the expected behavior of SDN-WISE and OpenFlow. In fact, in OpenFlow network resources are divided by the FlowVisor in *slices*, each assigned to a Controller, and a packet can belong to one slice only. In WSNs, instead, the same piece of data can be of interest to several applications using different Controllers. Therefore, in SDN-WISE a packet is not necessarily tied with

one Controller, i.e., different Controllers can specify different rules for the same packet.

B. SDN-WISE approach

The behavior of SDN-WISE *Sensor Nodes* is completely encoded in three data structures, namely: the *WISE States Array*, the *Accepted IDs Array*, and the *WISE Flow Table*. Like in most SDN approaches, such structures are filled with the information coming from the Controllers, running in appropriate servers. In this way the Controllers define the networking policies which will be implemented by the Sensor Nodes.

At any time SDN-WISE nodes are characterized by one current state for each active Controller. A state is a string of s_{State} bits. The WISE States Array is the data structure containing the values of the current states.

Given the broadcast nature of the wireless medium, sensor nodes will also receive packets which are not meant for them (not even for forwarding). The Accepted IDs Array allows each sensor node to select only the packets which it must further process. In fact, the header of the packets contains a field in which an Accepted ID is specified.

A node, upon receiving a packet, controls whether the ID contained in such field is listed in its Accepted IDs Array. If this is the case, the node will further process the packet; otherwise it will drop it.

In the case the packet must be processed, the sensor node will browse the entries of its WISE Flow Table. Each entry of the WISE Flow Table contains a *Matching Rules* section which specifies the conditions under which the entry applies. In SDN-WISE Matching Rules may consider any portion of the current packet as well as any bit of the current state. If the Matching Rules are satisfied, then the sensor node will perform an action specified in the remaining section of the WISE Flow Table entry. Note that such action may refer to how to handle the packet as well as how to modify the current state.

If no entry is listed in the WISE Flow Table whose Matching Rules apply to the current packet/state, then a request is sent to the Controllers.

In order to contact the Controllers, a node needs to have a WISE Flow Table entry indicating its best next hop towards one of the sinks. This entry is different from the others because it is not set by a Controller but is discovered by each node in a distributed way.

To this purpose an appropriate protocol is run by the Topology Discovery (TD) layer as it will be described layer, which is based on the exchange and processing of appropriate packets called TD packets. Such packets contain information about the battery level and the distance from the (nearest) sink in terms of number of hops. Every time a node receives one of such packets it compares the current best next hop with the information just acquired, then it chooses the best next hop giving priority to the number of hops, then the RSSI value received with the message and finally the residual battery level. This information is also used to populate a *WISE Neighbors list*. This list contains the addresses of the neighboring nodes, their RSSIs and their battery levels. This table is sent periodically to the Topology Management (TM) layer, as detailed in the following, in order to build a graph representation of the network. After that, the table is totally cleared and rebuilt with incoming TD packets in order to always have an updated view of the local topology.

One of the Controllers acts as a proxy between the physical network and the other Controllers. This is called *WISE-Visor* and is the analogous of the FlowVisor in traditional OpenFlow networks.

Controllers specify the network management policies which must be implemented by the WSN and can be application dependent. Accordingly, the Controllers can interact with the application.

Note that sensor nodes have limited capabilities in terms of memory, therefore, selection of the size of the different data structures is very important[2]. The optimal choice of such size depends on several deployment specific features set by the WISE-Visor during the initialization phase.

C. SDN-WISE protocol architecture

In SDN-WISE networks *Sensor Nodes* and one (or several) *Sink*(s) can be distinguished. Sinks are the gateways between the Sensor Nodes running the Data plane and the elements implementing the Control plane. The protocol stack of the Data plane, mostly run by Sensor Nodes, is shown in the left side of Figure 1. The protocol stack of the Sink and the other elements implementing the Control Plane are described in the right side of Figure 1. Sensor Nodes include an IEEE 802.15.4 transceiver and a micro-control unit (MCU). Above the IEEE 802.15.4 protocol stack, the **Forwarding** layer runs in the MCU which handles the arriving packets as specified in a *WISE flow table*[3]. This table is continuously updated by the Forwarding layer according to the configuration commands sent by the Controllers.

The **In-Network Packet Processing** (INPP) layer runs on top of the Forwarding layer. This is responsible for operations like data aggregation or other in-network processing. In current SDN-WISE implementation the INPP layer concatenates small packets that must be sent along similar paths. This would reduce the network overhead. Furthermore, we are developing solutions that enable the INPP to perform network coding which is very efficient in several WSN scenarios [16], [17].

The **Topology Discovery** (TD) layer, instead, can access all layers of the protocol stack by means of appropriate APIs. Thus, it can gather local information from the nodes and control their behavior at all layers, according to the indications provided by the Controllers. The TD layer provides an API to the application layer as well, which extends the IEEE 802 APIs. This guarantees legacy with existing applications.

In the Control plane, the network management logics are dictated by one or several Controller(s), one of which is the **WISE-Visor**. The WISE-Visor includes a **Topology Management** (TM) layer which abstracts the network resources so that different logical networks, with different management

[2]In particular note that the size of the WISE State Array gives an upperbound on the number of active Controllers that can be supported by the network.

[3]We derive our terminology from OpenFlow [3].

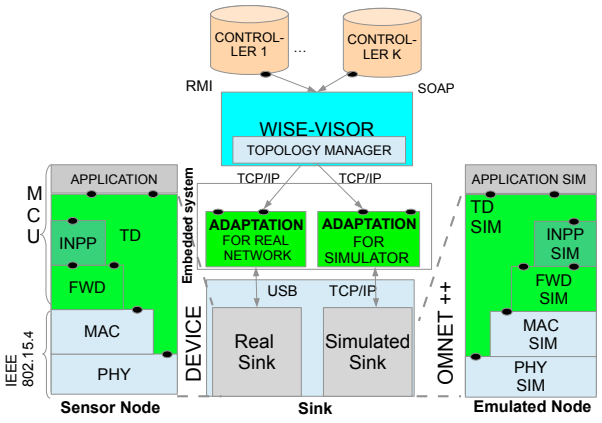


Fig. 1: SDN-WISE protocol stack.

policies set by different Controllers, can run over the same set of physical devices. The TM layer has access to APIs offered by all the protocol layers. Such APIs enable to control the behavior of all protocol layers and therefore to implement cross-layer operations. The use of the TM layer is driven by two requirements: i) collecting local information from the nodes and sending them to the Controller(s) in the form of a graph of the network (reporting information related to topology, residual energy level, SNR on the links, etc.) ii) controlling all protocol stack layers as specified by the Controller(s). To this purpose, between the sink device (characterized by the same protocol stack of Sensor Nodes) and the WISE-Visor there is the **Adaptation layer** which is responsible for formatting the messages received from the Sink in such a way that they can be handled by the WISE-Visor and *viceversa*.

The Controllers may run either in the same node hosting the TM layer or in remote servers. As a consequence, the interactions between the Controllers and the TM layer can occur in several ways, as shown in the central part of Figure 1. In fact, in the case the Controllers run in the same node hosting the TM layer, interactions will occur through the Java methods offered by the TM layer. Alternatively, interactions can occur through the Java *remote method invocation* (RMI) or the *Simple Object Access Protocol* (SOAP). In this way, programmers can implement Controllers either in Java or in some Web programming languages.

Finally, note that the SDN-WISE protocol stack also includes a specific Adaptation layer which can interact with a simulated sink (not a real sink). In this way the Control plane can set the networking policies of a simulated network. In other words SDN-WISE offers a tool which is very similar to Mininet [7].

IV. SDN-WISE PROTOCOL DETAILS

In this section we describe in detail the major features of the SDN-WISE protocols. More specifically, in Section IV-A we will explain the Topology Discovery protocol. Then, in Section IV-B we will describe in detail how sensor nodes behave when they receive a new packet.

A. Topology Discovery

In Section III-C we have explained that the Topology Manager module in the WISE-Visor builds a consistent view of the current network status. To this purpose it requires to collect local topology information generated by sensor nodes. The Topology Discovery protocol run by all sensor nodes, is responsible for generating such information and delivering it to the WISE-Visor.

The TD protocol maintains information about the next hop towards the Controllers and its current neighbors updated. To this purpose all sinks[4] in the SDN-WISE network periodically and (almost) simultaneously transmit a Topology Discovery packet (TD packet) over the broadcast wireless channel. Such packet contains the identity of the sink that has generated it, a battery level, and the current distance from the sink which is initially set to 0.

A sensor node A receiving a TD packet from sensor node B (note that B can be a sink) performs the following operations[5]:

1) inserts B in the list of its current neighbors along with the current RSSI and the battery level.
   Obviously, if B is already present in the list of current neighbors, then only the RSSI and battery level values are updated;
2) controls whether it has recently received a TD packet with a lower value of the current distance from the sink. If this is not the case, then node A updates the value reported in the TD packet to the current value plus one and sets its next hop towards the Controllers equal to B;
3) sets its battery level in the corresponding field of the TD packet;
4) transmits the updated TD packet over the broadcast wireless channel.

Periodically, each sensor nodes generates a packet containing its current list of neighbors and sends it to the WISE-Visor. Note that the list of neighbors is periodically cleared. Nodes receiving packets directed towards the WISE-Visor or the Controllers relay them to the node set as their next hop towards the Controllers.

The rate of TD packets generation as well as of the packets containing local topology information impacts the performance of SDN-WISE. In fact, the higher such frequencies is, the higher is the overhead generated by the protocol. However, such frequencies cannot be too low in dynamic scenarios (with rapid topology changes); accordingly, their setting is application specific and can be controlled by the WISE-Visor.

B. Packet handling

In this section we describe how the Forwarding protocol described in Section III-C operates upon receiving a packet. To this purpose we first provide a description of the WISE packet format; then, a description of the structure of the WISE flow table and, finally, we will explain how the WISE flow table is utilized upon reception of a packet.

[4]We already said that there might be several sinks in the same SDN-WISE network.

[5]TD packets received with RSSI lower than a given threshold will be neglected. In our current implementation such threshold is set to -60 dBm.

Matching Rule					Matching Rule					Matching Rule					Action					Statistics	
Op.	Size	S	Addr.	Value	Op.	Size	S	Addr.	Value	Op.	Size	S	Addr.	Value	Type	M	S	Addr.	Value	TTL	Counter
=	2	0	2	B	>	2	0	10	x_{Thr}	=	1	1	0	0	Modify	1	1	0	1	122	23
=	2	0	2	B	≤	2	0	10	x_{Thr}	=	1	1	0	1	Modify	1	1	0	0	122	120
=	2	0	2	B	-	0	-	-	-	-	0	-	-	-	Forward	0	0	0	D	122	143
=	2	0	2	A	=	1	1	0	0	-	0	-	-	-	Drop	0	0	-	-	100	42
=	2	0	2	A	=	1	1	0	1	-	0	-	-	-	Forward	0	0	0	D	100	32

Fig. 3: WISE flow table.

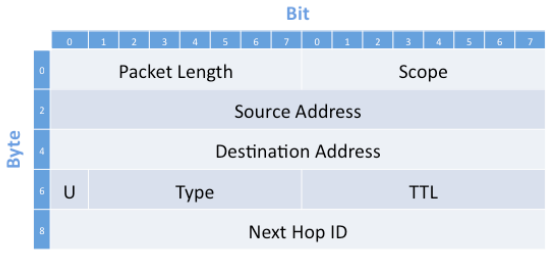


Fig. 2: WISE packet header

As shown in Figure 2, SDN-WISE packets have a fixed header consisting of 10 bytes divided in the following fields:

- The *Packet length* field provides the length of the packet, included the payload (if any), in bytes.
- The *Scope* identifies a group of Controllers that have expressed interest in the content of the packet. The Scope value is initially set to 0 (as default) but can be modified through appropriate entries in the WISE flow table of the sensor node generating the packet. In our current implementation Scope values have global validity as the WISE-Visor guarantees network-wide consistency.
- The *Source* and *Destination Addresses* obviously specify the addresses (we use two bytes addresses in our implementation) of the node which has generated the packet and the intended destination.
- The flag *U* is used to mark packets that must be delivered to the closest sink.
- The *Type of packet* field is used to distinguish between different types of messages in fact besides data packets, TD packets and packets containing local topology information, which we have already discussed, SDN-WISE uses other types of packets for the request of a new entry to the Controllers, for the introduction of a new entry in the WISE flow table of a given sensor node, for opening a path in a sequence of sensor nodes, and for turning the wireless interface of a sensor node off for a certain time interval. The type of packet will determine the interpretation of the packet payload.
- The *TTL* is the time to live and is reduced by one at each hop.
- Finally, the *Next Hop ID* is the field which must be present in the Accepted IDs Array for the packet to be further processed by the sensor node (as explained in Section III-B).

The structure of the WISE flow table is shown in Figure 3 and extends the one proposed in [5].

Like in the OpenFlow case we can distinguish three sections: Matching Rules, Actions, and Statistics. The Matching Rules specify up to three conditions. If such conditions are satisfied then the corresponding Action is executed and the information reported in the Statistics section is updated. Each Matching Rule consists of a field (S) which specifies whether the condition regards the current packet ($S = 0$) or the state ($S = 1$); the fields *Offset* and *Size* specify the first byte and the size, respectively, of the string of bytes in the packet or the state which should be considered, the *Operator* field gives the relational operator to be checked against the *Value* given in the rule. For example, the second Matching Rule of the first entry in the WISE flow table given in Figure 3 is satisfied if the first 2 bytes (Size = 2) after byte 10 (Offset = 10) of the current packet (S=0) assume a value which is higher (Op = "$>$") than x_{Thr} (Value = x_{Thr}).

If all the conditions specified in the Matching Rules section are satisfied (if Size = 0 then the Matching Rule is not considered), then the corresponding Action is executed. An Action is specified by five fields. The *Type* specifies the type of action. Possible values of the Type field can be "Forward to", "Drop", "Modify", "Send to INPP", "Turn off radio". The flag *M* specifies whether the entry is exclusive ($M = 0$) or not ($M = 1$). In the first case, if the conditions are satisfied, the sensor node executes the action and then stops browsing the WISE flow table. In the second case, instead, after executing the action, the sensor node continues to browse the WISE flow table and executes other actions if the corresponding conditions specified in the Matching Rules section are satisfied.

The meaning of the other two fields (i.e., *Offset* and *Value*) depend on the type of action. For example, if the action is "Forward to" they must specify which is the Next Hop ID (which will be written in the packet), if it is "Drop" they give the drop probability as well as the next hop ID in case the packet is not dropped, if it is "Modify" they specify the Offset and the new Value to be written, if it is "Send to INPP" they specify they type of processing that must be executed, if it is "Turn off radio" they specify after how much time the radio must be turned on again.

In case the action is "Modify", the flag *S* specifies whether

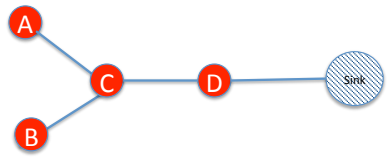


Fig. 4: Exemplary topology.

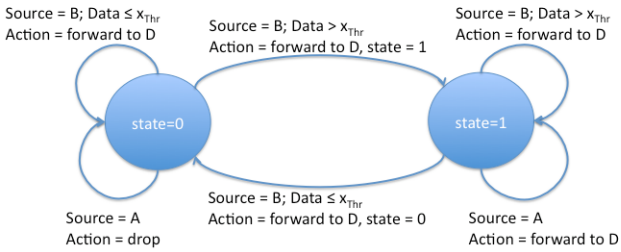


Fig. 5: Finite state machine implementing a policy such that packets generated by A are dropped if the last data measured by B is lower than (or equal to) x_{Thr}.

the action must be executed on the packet or the state.

Statistics are used like in standard OpenFlow and thus, we do not discuss them further in this paper.

In the following we will show how sensor nodes use their data structures in an exemplary scenario highlighting the specific features of SDN-WISE. Consider the network topology shown in Figure 4 and suppose that data measured by sensor A is significant only if the data measured by sensor B is higher than a given threshold x_{Thr}. Therefore, if we pursue energy efficiency a network policy should be implemented that enforces node C to drop packets if the packet received by B contains a measured data lower than x_{Thr}. Using traditional OpenFlow-like solutions it is impossible to enforce the above behavior for the following reasons:

- matching is executed only verifying the equivalence between a field in the packet header and a specific value, i.e., it is not possible to look at the payload and "higher than"-type relationships are not supported;
- in stateless solutions it is impossible to make the handling of the packet dependent on the content of another packet.

Instead, in SDN-WISE the above policy can be easily realized through the finite state machine represented in Figure 5 which can be implemented through the five WISE flow table entries shown in Figure 3. In fact, the first two lines specify the transitions between states 0 and 1 and *viceversa*, depending on the value contained in the 10th byte of the packets generated by node B. More specifically, note that in the first entry the first Matching Rule selects packets coming from node B, the second Matching Rule selects those that have in the tenth and eleventh bytes a value higher than x_{Thr}, finally the third Matching Rule selects the cases in which the current state of the node is 0. If all the above rules are satisfied then the state is set to 1 as shown in the Action section. Analogously, the second entry selects the cases in which the incoming packet has been generated by B contains a measured data lower than or equal to x_{Thr}, and the current state is one; and in such cases sets the state to 0. The third entry in the table is executed any time a packet generated by B is received and specifies that the packet must be forwarded to D in any case. Finally, the fourth and fifth entry specify that packets coming from A must be dropped if the current state is 0 (see the fourth entry) or forwarded to D if the current state is 1 (see the fifth entry).

V. Prototype and Testbed

In our testbed we used EMB-Z2530PA based sensor nodes. EMB-Z2530PA is a wireless module developed by Embit for LR-WPAN applications. The module provides IEEE 802.15.4 wireless connectivity in the 2.4 GHz ISM band. It is based on a Texas Instruments CC2530 single chip device which is an 8051 8-bit controller. Each node is equipped with 8kB of RAM and 256 kB of Flash memory 40 kB of which are used for MAC layer (TIMAC for CC2530 v1.4.0) and 10 kB are used for the SDN-WISE protocol.

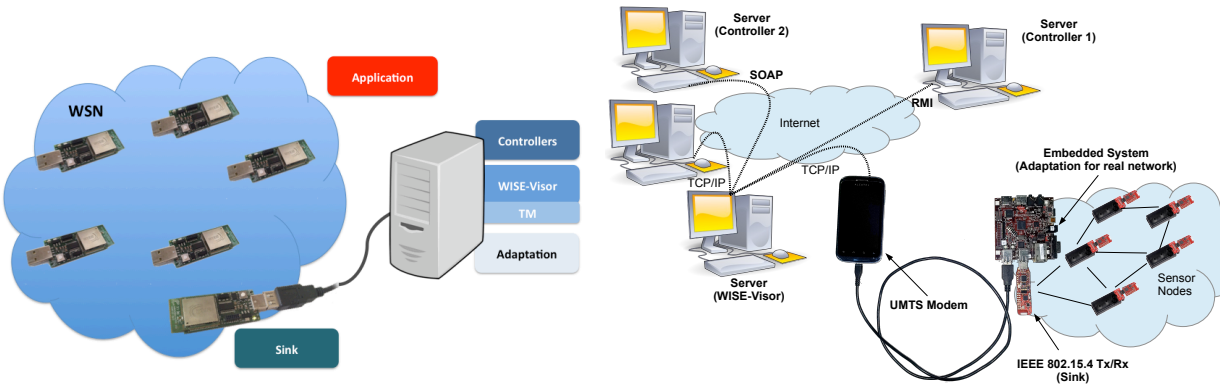
For what concerns the Control plane, our prototype supports different deployment options. The simplest is the one depicted in Figure 6(a), in which the node hosting the sink is attached to the desktop computer using USB 2.0. In our testbed the WISE-Visor as well as the Controllers are hosted in this desktop computer which is equipped with Intel(R) Core(TM) 2 CPU 2.40 GHz and 4GB of RAM running Windows 7, 32 bit. The Controllers have been implemented using Java 7. Topology information is stored in a JGraphT's Graph object.

The above deployment option requires the presence of a node (the PC) with significant computational resources in the area where the sensor nodes are deployed.

In several scenarios, however, it is not possible to deploy such powerful nodes in the network area. In these cases, the sink is usually attached to an embedded system that access the Internet through some communication interface. For example, in the experimental testbed represented in Figure 6(b), the sink is a TI CC2500 device attached via USB to a Beagleboard running a Linux operating system (Ububtu 12.04). The Adaptation layer is implemented in the Beagleboard which sends control packets to the WISE-Visor on a remote server. In our testbed the Beagleboard is equipped with an UMTS interface (the smartphone in Figure 6(b)) and communication between the Adaptation and WISE-Visor occurs through TCP/IP connections.

The Controllers may be hosted by other PCs (or virtual machines) and interact with the WISE-Visor layer in several ways. In our testbed we support both SOAP and RMI interaction models.

Finally, simulations modeling the behavior of the sensor nodes and the sinks can be executed on another PC. In Figure 7 we show a screenshot from an OMNeT++ simulation showing the topology of the simulated sensor network. Node 0 is the sink and interacts through the Adaptation module with a real instance of the WISE-Visor. Accordingly, Controllers 1 and 2 can be real controllers determining the policies which are applied by the simulated sensor nodes. In addition, the (emulated) Sink can be used to create a virtual network extension so that simulated and real nodes are fully integrated and can interact with each others. This can be useful for testing a real network scenario in which there are not enough real devices. In this case only one Controller is used for both nodes



(a) Simplest deployment option.

(b) Distributed deployment option.

Fig. 6: SDN-WISE deployment options.

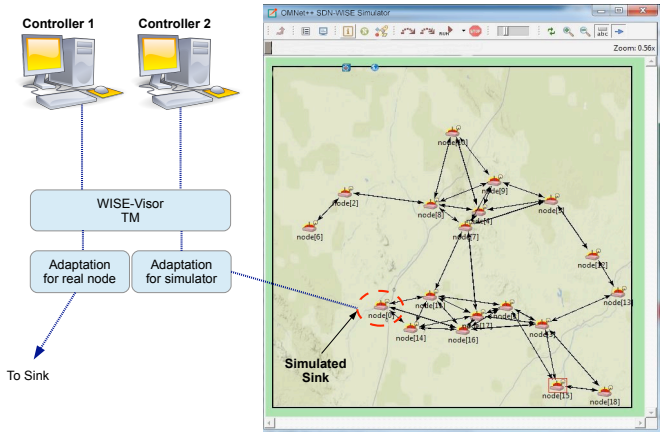


Fig. 7: Integration with the OMNeT++ simulator.

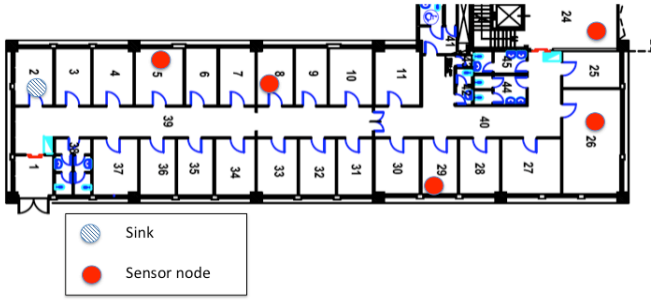


Fig. 8: Nodes deployment.

(real and simulated) and it treats all of them without making any distinction.

VI. Performance Evaluation

Due to space constraints we will omit the results obtained by using OMNeT++ simulations. Instead, in this section we will illustrate the results obtained by the SDN-WISE platform in a physical testbed. More specifically, 6 nodes (5 sensor nodes and a sink) have been deployed as shown in Figure 8. In our experiments the sink was connected via USB to a PC which was running the Adaptation layer and the entire Control plane functionality, like shown in Figure 6(a). Finally, the Controller has been implemented in Java and simply executes the Dijkstra algorithm.

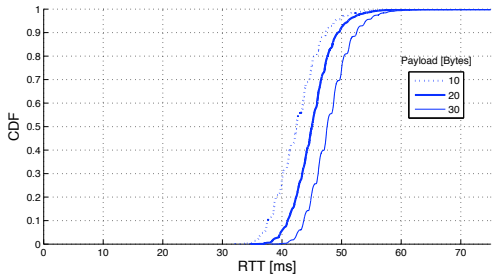
In each measurement campaign 5000 data packets have been sent, each every 15 seconds. Different payload sizes have been considered for such packets (10, 20 and 30 bytes). Also, we have changed the time interval, T, between two consecutive generations of TD packets. In each campaign we have set the time interval between the transmissions of local topology information to twice the value of T.

In the following we show the performance achieved by SDN-WISE in terms of

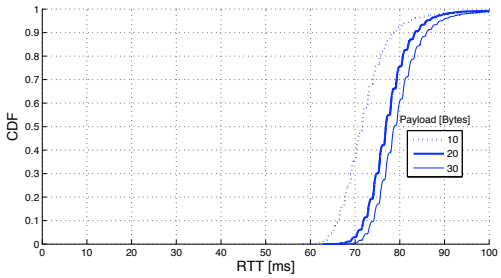
- Round Trip Time (RTT), that is, the time interval between the generation of a data packet and the reception of the corresponding acknowledgment;
- Efficiency, measured as the ratio between the number of payload bytes received by the intended destinations and the overall number of bytes circulating in the network;
- Controller response time, measured as the duration of the time interval when the Controller receives a request for a new entry and the time instant when the Controller sends the corresponding entry.

In Figures 9(a) and 9(b) we represent the *Cumulative Distribution Functions* (CDF) of the RTT when the distance between the packet source and the packet destination is equal to 3 and 5, respectively. In each figure we represent three curves obtained for different values of the payload size (10, 20, and 30 bytes). As expected, RTT increases as the distance and the payload increase. Furthermore, we expect a similar behavior from the standard deviation. Indeed, this is reflected in Figures 10 and 11 where we show the average and the standard deviation of the RTT vs. the payload size for different values of the distance between source and destination.

In Figures 10 and 11 we plot a curve for the multicast case, as well. This has been obtained by measuring the time instant between the transmission of a packet and the reception of the acknowledgement from the last destination. In this case, only three destinations were considered and were deployed within the radio range of the source. Obviously, the average and the standard deviations of the RTT is slightly higher than in the analogous (one hop) unicast case. The corresponding CDFs are represented in Figure 12.



(a) Number of hops = 3.



(b) Number of hops = 5

Fig. 9: CDFs of the RTT for different payload sizes and different distances between the source and destination node.

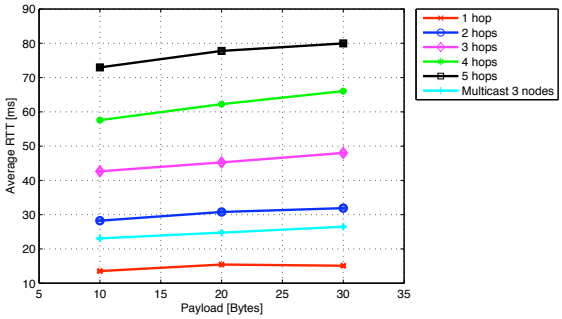


Fig. 10: Average RTT vs. the payload size, for different values of the number of hops.

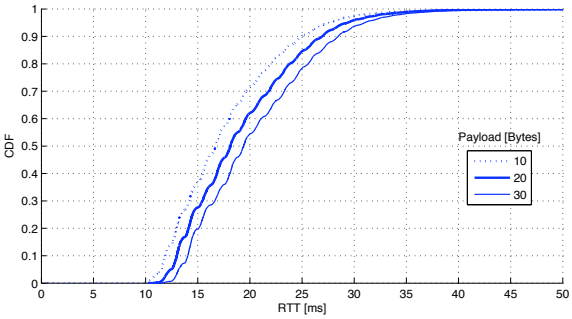


Fig. 12: CDF of the RTT in the multicast case for different payload sizes.

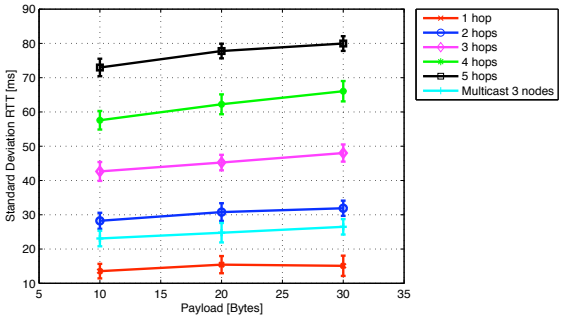


Fig. 11: Standard deviation of the RTT values vs. the payload size, for different values of the number of hops.

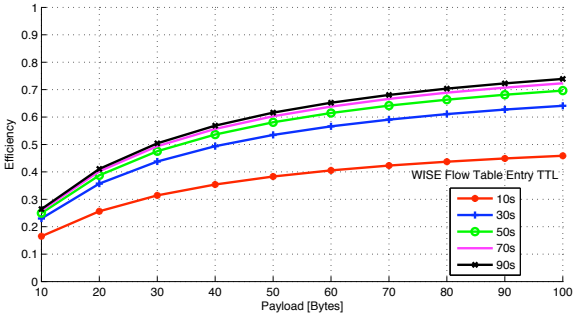


Fig. 13: Efficiency for different values of maximum WISE Flow Table entry TTL.

The performance in terms of efficiency are shown in Figures 13 and 14. More specifically, in Figure 13 we represent the efficiency vs. the payload size for different values of the lifetime of an entry in the WISE flow table, which we denote here as TTL, instead in Figure 14 we show the same curves obtained for different values of the interval between consecutive transmissions of the TD packets, T.

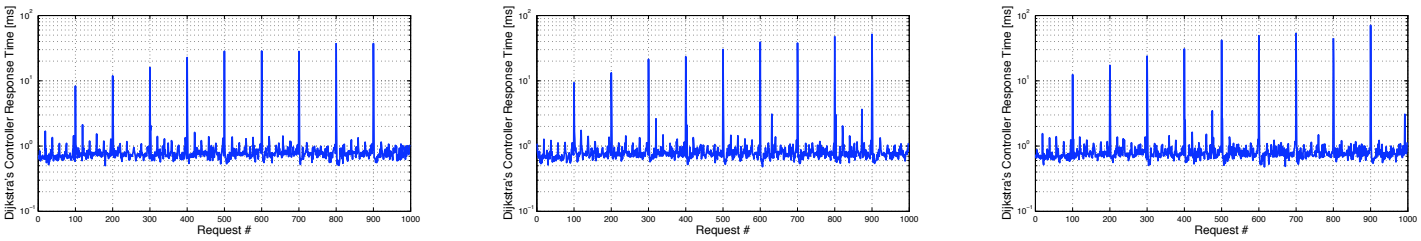
Note that most of the inefficiency is due to the high ratio between the header size and the payload size.

Finally, in Figures 15 we show the response times of the Controller to requests from nodes for new entries. We have simulated the process of request generation by the nodes modeling a network consisting of 50, 60, and 70 nodes. Furthermore we have assumed that initially only 10% of possible links are active but we have increased such number by 10% every 100 requests, and at the end we obtain a fully meshed network. What we observe is that there are hypes in the plots which are in correspondence of an increase in the number of links which calls for a new run of the Dijkstra algorithm. In any case the response delay is always below 100 ms. Such value could be further reduced by running the Controller on a more powerful hardware.

VII. Conclusions

In this paper we have introduced SDN-WISE, a Software Defined Networking solution for WIreless SEnsor networks. SDN-WISE is stateful and aimed at reducing the amount of information exchanged between sensors and SDN controllers. Details on the SDN-WISE protocol stack are provided as well as results obtained from extensive measures in a physical testbed. SDN-WISE is a promising approach to the realization of programmable WSNs.

Nevertheless, several issues are still open for further research. The most important is related to security. In fact, solu-



(a) 50 Nodes. (b) 60 Nodes. (c) 70 Nodes.

Fig. 15: Controller response times for different topologies.

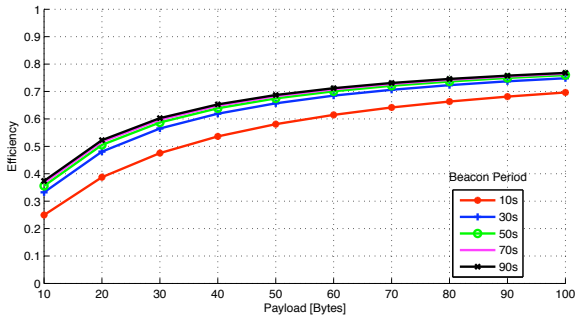


Fig. 14: Efficiency for different values of beacon sending period.

tions are required that make the SDN-WISE network resilient to intentional attacks and bugs in the Controller software. We are addressing such issue by introducing a mechanism that in case of congestion deletes an entry in the WISE flow table when this is used too much (as compared to the others).

Furthermore, we are investigating possible implementations of network coding in SDN-WISE. In this context, interesting insights have been already presented in [18].

Acknowledgments

This work was supported by the 7th Framework Program of the European Commission within the NEWCOM# (Network of Excellence in Wireless Communications) project and by MIUR under the SIGMA contract.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *White paper*. March 2008.

[2] S. Shenker *et al.*. The future of networking and the past of protocols. *OpenNetSummit 2011*. October 2011.

[3] B. Pfaff *et al.*. OpenFlow Switch Specification – Version 1.1.0 Implemented (Wire Protocol 0x02). February 2011.

[4] T. Luo, H.-P. Tan, and T. Q. S. Quek. Sensor OpenFlow: Enabling Software-Defined Wireless Sensor Networks. *IEEE Communications Letter*. Vol. 16, No. 11, pp: 1896–1899. November 2012.

[5] S. Costanzo, L. Galluccio, G. Morabito, and S. Palazzo. Software Defined Wireless Networks: Unbridling SDNs. *In Proc. of EWSDN 2012*. October 2012.

[6] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch. *ACM Computer Communication Review*. Vol. 44, No. 2, pp.: 45-51. April 2014.

[7] B. Lantz, B. Heller, and N. McKewon. A network in a laptop: rapid prototyping for software-defined networks. *In Proc. of ACM Hotnets-IX*.

[8] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*. Vol. 40, No. 8., pp.: 102-114. August 2002.

[9] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava. SensorWare: Programming sensor networks beyond code update and querying. *Pervasive and mobile computing*. Vol. 3, No. 4, pp.: 386412. August 2007.

[10] L. Mottola and G. P. Picco. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Computing Surveys*. Vol. 43, No. 3. April 2011.

[11] J. Qadir, N. Ahmed, and N. Ahad. Building Programmable Wireless Networks: An Architectural Survey. *Available at* http://arxiv.org/pdf/1310.0251.pdf. January 2014.

[12] D. Zeng, T. Miyazaki, S. Guo, T. Tsukahara, J. Kitamichi, and T. Hayashi. Evolution of Software-Defined Sensor Networks. *In Proc. of IEEE MSN 2013*. December 2013.

[13] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. *In Proc. of IEEE Infocom 2002*. Apr. 2002.

[14] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of Network Density on Data Aggregation in Wireless Sensor Networks. *In Proc. of IEEE ICDCS 2002*. July 2002.

[15] A. Manjeshwar and D. P. Agrawal. TEEN: A Routing Protocol for Enhanced Efficiency in Wireless Sensor Networks. *In Proc. of IEEE IPDPS 2001*. April 2001.

[16] C. Fragouli, J.-Y. Le Boudec, and J. Widmer. Network coding: an instant primer. *ACM SIGCOMM Computer Communication Review*. Vol. 36, No. 1, pp.: 63–68. January 2006.

[17] L. Keller, E. Atsan, K. Argyraki, and C. Fragouli. SenseCode: Network coding for reliable sensor networks. *ACM Transactions on Sensor Networks*. Vol. 9, No. 2. March 2013.

[18] J. Wang, J. Ren, K. Lu, J. Wang, S. Liu, and C. Westphal. An optimal Cache management framework for information-centric networks with network coding. *In Proc. of IFIP Networking 2014*. June 2014.