UNIVERSITÀ DEGLI STUDI DI PALERMO

Information and Communication Technologies
Dipartimento Energia, Ingegneria dell'Informazione e Modelli Matematici
Ingegneria delle Telecomunicazioni

Progetto: TECLA – "Ricercatori ed Esperti Di Alta Tecnologia e Innovazione Tecnologica Applicata al Settore dei Beni Culturali" a valere sull'Avviso n. 713/Ric. del 29 ottobre 2010
TITOLO III
"CREAZIONE DI NUOVI DISTRETTI E/O NUOVE AGGREGAZIONI PUBBLICO – PRIVATE"

Codice identificativo progetto: PON03PE_00214_1/F7

UNIONE EUROPEA
Fondo Europeo di Sviluppo Regionale

PON Ricerca e Competitività 2007-2013 Piano di Azione Coesione

Ministero dell'Istruzione, dell'Università e della Ricerca

Ministero dello Sviluppo Economico

Governo Italiano - Presidenza del Consiglio dei Ministri
Ministro per la Coesione Territoriale

A Software Defined approach to the Internet of Things: From Wireless Sensor Networks to Network Operating Systems

IL DOTTORE
SEBASTIANO MILARDO

IL COORDINATORE
CHIAR.MA PROF.SSA ILENIA TINNIRELLO

IL TUTOR
CHIAR.MO PROF. SERGIO PALAZZO

CICLO XXX
ANNO CONSEGUIMENTO TITOLO 2018

to my family and Silvia

Sommario

Di recente il mondo delle reti di telecomunicazioni è stato caratterizzato dall'emergere del paradigma di Networking Software Defined (SDN). Grazie a questo approccio è possibile fornire un'interfaccia standard per lo sviluppo di software in grado di gestire i dispositivi di networking e i flussi di traffico che li attraversano, disaccoppiando il management della rete dal forwarding dei dati. Questa soluzione, di cui il protocollo di comunicazione OpenFlow è tra i maggiori esponenti, ha riscosso un notevole successo nelle reti cablate permettendo di superare il *vendor lock-in* e permettendo la gestione di dispositivi eterogenei tramite un unico punto di accesso logicamente centralizzato.

Un ulteriore ambito che potrebbe trarre notevole beneficio da un simile approccio è quello dell'Internet of Things (IoT), cioè l'insieme di quei dispositivi embedded di uso quotidiano interconnessi tramite Internet, di cui le reti wireless di sensori rappresentano un importante caso d'uso. Anche in questo ambiente, l'eterogeneità dei dispositivi e la necessità di soluzioni *cucite* attorno alla singola applicazione ha creato una moltitudine di protocolli e soluzioni che difficilmente riescono a cooperare, creando così una moltitudine di *Intra*net of Things invece di un unica *Inter*net of Things. Di recente, sono state presentate alcune soluzioni per implementare un approccio Software Defined anche nelle reti di sensori wireless.

Il lavoro presentato in questa dissertazione si colloca in questo quadro presentando un framework completo per la gestione di reti di sensori tramite un'interfaccia OpenFlow-like chiamata SDN-WISE e la sua integrazione all'interno del sistema operativo per networking ONOS.

In questa dissertazione sono presentati i dettagli architetturali e le conclusioni maturate durante la progettazione della soluzione in questione.

Abstract

Recently, telecommunications have been characterized by the emerging of the Software Defined Networking (SDN) paradigm. This approach provides standard interfaces for the development of network management software. Software that can control networking devices and the flows of traffic passing through them, decoupling the network management from the data forwarding. This solution, of which the OpenFlow communication protocol is one of the major players, has been successfully applied in wired networks allowing to overcome the *vendor lock-in* and allowing the management of heterogeneous devices through a single logically centralized point of control.

Another area that could greatly benefit from such an approach is the Internet of Things (IoT), that is, an interconnection via the Internet of devices embedded in everyday objects, of which wireless sensor networks represent an important use case. Even in this environment, the heterogeneity of the devices and the need for solutions *tailored* around each single application has created lots of protocols and solutions that can hardly cooperate, thus creating multiple *Intra*nets of Things instead of the envisioned unique *Inter*net of Things. Recently, some solutions have been presented to implement a Software Defined approach in wireless sensor networks.

The work presented in this dissertation belongs to this context and it presents a complete framework for the management of sensor networks through an OpenFlow-like interface called SDN-WISE and its integration within the ONOS network operating system.

This dissertation presents the architectural details and the conclusions reached during the design of the proposed solution.

Contents

List of Figures

Chapter 1

Introduction

The *Software Defined Networking* paradigm is changing the way in which networks are conceived with disruptive implications on network design, deployment, operation, and maintenance. In fact SDN has been envisioned as a way to reduce complexity and increase flexibility of network configuration and management [1]. In SDN networks management operations (Control Plane) are logically centralized and physically decoupled from forwarding operations (Data Plane), so allowing to easily update the behavior of the network. In the last few years, the industrial and academic communities have devoted relevant efforts to SDN development, and nowadays well established SDN solutions are available for both wired and wireless infrastructured network domains.

In the meantime, market forecasts released by different institutions anticipate the deployment of larger and larger numbers of *Internet of Things* (IoT) devices in the near future. For example, the most recent study conducted by IDC for Intel and the United Nations forecasts that there will be around 200 billion IoT devices by 2020[1]. As a result of these high expectations, industry is investing huge resources in the development and deployment of IoT technologies and platforms. The academic research community is also following this wave and today IoT is one

[1]`http://intel.ly/1i8O2ec`

of the hottest research topic in computer science, whereas it is also spreading to other domains such as transportation, production and manufacturing, as well as environmental and social sciences.

Software-Defined Networking (SDN) and Internet of Things (IoT) are today two hot topics deemed in the short term to be fused with each other. Indeed the flexibility and the ability of SDN to balance traffic on different paths in such a way to implement an optimized network resources' usage makes it a valuable ally in the implementation of the IoT vision. In fact, one of the most critical issues for the IoT realization is the huge data management and processing and the consequent need for optimized network routing and balancing.

Wireless sensor networks (WSN) are a fundamental ingredient of the IoT ecosystem. These networks are composed of sensor nodes that cooperatively monitor the physical environment [2] and track events [3] generating incessant streams of data that the IoT can use to improve our lives and our businesses in many ways. Unlike traditional networks, WSN nodes are resource constrained in terms of energy, communication range, bandwidth, processing power, and memory [4]. Additionally, a WSN has to deal with a great variety of applications that require different network structures [5, 6], and may exploit mobile nodes [7, 8].

Sensors, in particular, offer unprecedented access to *granular data that can be transformed into powerful knowledge*[2]. Integrated analytics platforms will be used to overcome the data burst and avoid that sensor data will just add information overload and noise escalation.

In order to implement this unified IoT scenario, integration between different SDN platforms and implementations is needed. Therefore, this thesis focuses

[2]`http://bit.ly/2mLtG7t`

on SDN-like platforms for wireless sensor networks (SDWSN) by defining and implementing a Data Plane and a Control Plane for such devices.

Having a Control Plane for Software Defined IoT devices is extremely important as the existing controllers are typically bundled with some sample applications and the support for other devices is tightly coupled with the particular switch behavior.

To this purpose, one of the main focus of this thesis is to introduce an extended controller based on the Open Network Operating System (ONOS), as the starting point for the implementation of the IoT vision. By using ONOS to control heterogenous networks it is possible for any wireless sensor device to communicate and interact in a standard and transparent way with external nodes.

Leveraging this approach it is possible to unlock new possibilities for the management of Software Defined IoT devices, overcoming the constraints in terms of computational power and energy consumption and making life easier for network administrators.

1.1 Structure of this Dissertation

This dissertation is organized in 8 chapters (including this introduction) as follows:

Chapter 2 provides basic definitions and concepts related to SDN and SDWSN. It also presents the State of the Art for SDWSN and in particular it focuses on SDWN, one of the first solutions for SDWSN.

Chapter 3 shows a comparison between different communication protocols for IoT devices. ZigBee, 6LoWPAN, and SDWN are compared to reveal the difference between distributed solutions and a centralized one.

Chapter 4 presents SDN-WISE, a stateful software defined solution for wireless sensor networks based on SDWN which is the starting point for the following chapters.

Chapter 5 analyzes how SDN-WISE can be used to support QoS policies in WSN. In particular in the context of heterogeneous sensors for environmental monitoring.

In Chapters 6 and 7 the need for a Network Operating System for Software Defined IoT devices is underlined and an extension of ONOS is presented.

Then, in Chapter 8 a routing application based of Geographic forwarding and in Chapter 9, a novel declarative approach to NOS applications for IoT based on Long Short-Term Memory Recurrent Artificial Neural Networks is given.

Finally, Chapter 10 summarizes the conclusions and proposes further works related to the presented subject.

1.2 Acknowledgements

Part of the results described in this dissertation comes from the research funded by Progetto: TECLA Ricercatori ed Esperti Di Alta Tecnologia e Innovazione Tecnologica Applicata al Settore dei Beni Culturali (PON03PE_00214_1/F7). The code produced while working on this project is freely available at `https://github.com/sdnwiselab`.

Chapter 2

Technical Background

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 (generally referred to as *Motes*) 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 [9].

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 [9], 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 [10, 11, 12]. 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.

2.1 Software Defined Networking

The *Software Defined Networking* (SDN) paradigm and OpenFlow, which currently is the most popular instance of SDN, have been recently proposed to solve analogous issues in the wired domain [1]. Thanks to standardized interfaces, which can work on networks made of heterogeneous switches, 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 *Control Plane* 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 [13],
- 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, 14],
- 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.

The above are crucial advantages for network operators which, thus, are investing large efforts in the SDN domain in terms of new equipment acquisition and/or knowhow development.

As a result, rarely the interest in a new networking paradigm has increased at such a pace as it is happening for SDN. In fact even if OpenFlow was initially meant for universities and campus networks the huge benefits introduced in terms of simplicity and evolvability made it suitable for other contexts and it is now possible to find even carrier-grade networking devices that are OpenFlow compliant [15]. Therefore, 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. In fact, the applications of SDN are now everywhere. SDN has been used in campus networks [1], wide area networks [16], carrier networks [17] but also wireless and mobile devices [18] 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.

2.2 Software Defined Wireless Sensor Networks

Despite the vast adoption of OpenFlow, and SDN in general, in the wired domain, there has been no such widely accepted solution in wireless networks and, especially, WSNs[1]

Sensor OpenFlow [20] was the first attempt to implement an SDN protocol for WSNs. It follows the OpenFlow architecture, by considering that the nodes should maintain a flow table with entries of specific, predefined format. Sensor OpenFlow supports in-network processing mainly to enable data aggregation, as commonly done in WNS for energy preservation. Note that Sensor OpenFlow cannot support the wide range of protocols, either standard or proprietary that have been proposed in the context of WSNs. Furthermore, in [21] the Sensor OpenFlow approach is integrated with other WSN programming techniques.

Authors in [22] also presented the idea of exploiting the OpenFlow technology to address the reliability in WSNs. The authors claim that OpenFlow-based sensors are more reliable than typical sensors, and simulation results show that the proposed approach achieves better performance for large networks.

The use of OpenFlow in a wireless mesh network allows a rapid change of forwarding and routing algorithms [23]. A survey on challenges and opportunities in using wireless SDNs is presented in [24]. This chapter claims that the SDN technology will have to face problems regarding slicing, isolation, status reporting,

[1]Actually, even before the break-in of SDN in the WSN arena, there have been studies about WSNs in which the behavior of nodes was dependent on *rules* that can be changed over time – see [19], for example.

and handoffs, whereas it will improve connectivity, QoS, planning, security, and localization. Ref. [25] proposes an SDN system, where experimentations show that the proposed solution reduces the energy consumption and provides a higher level of flexibility in network management.

Another SDN-like solution is TinySDN [26] which focuses on the support of SDN operations across different platforms which is achieved by building on TinyOS. TinySDN enables interoperability of SDN-enabled nodes with several controllers, and has been implemented and tested with the Cooja simulator.

However these approaches, as well as the ones presented in [27] and [28], are directly derived by OpenFlow and thus, require all nodes to be instructed by the Controller to process the packets they are called to handle. Therefore, they involve large amount of signaling exchange between nodes and Controller. Also, since each flow traversing a node must have an appropriate entry in a given data structure (denoted as *flow table* in OpenFlow) in order to specify how to distinguish packets belonging to the flow and how to treat them, it is likely that nodes will have to maintain large flow tables.

2.2.1 SDWN

The first implementation of SDWN was developed in October 2012 [29]. The main idea behind the protocol is to adapt a centralized approach, such as the one proposed in SDN networks, to a wireless environment, thus giving the opportunity to support the flexible definition of rules and topology changes.

The SDWN protocol stack is shown in Fig. 2.1: physical (PHY) and MAC layers are those of the 802.15.4 [30], whereas upper layers are inspired by the SDN paradigm. A typical SDWN network is composed of a controller device, a sink node, and several other nodes. The controller gathers the information from nodes,

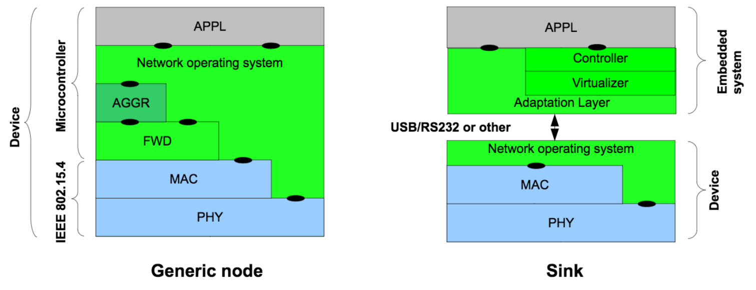


Figure 2.1: SDWN protocol stack.

maintains a representation of the network, and establishes routing paths for each data flow. The sink is the only node that is directly connected to the controller, and it acts as a gateway for nodes. Usually the sink coincides with the network coordinator and its protocol stack is equivalent to that of a generic node. The stack of a generic node is divided into three parts: 1) the forwarding layer (FWD); 2) the aggregation layer (AGGR); and 3) the network operating system (NOS). The MAC layer provides incoming packets to the FWD layer that identifies the type of the packet. Six different types of packets are defined as follows:

1. data: generated (delivered) by (to) the application layer;
2. beacon: periodically sent in broadcast by all nodes in the network;
3. report: containing the list of neighbors of a node;
4. rule request: generated when it receives a packet for handling which it has no information (i.e., the path);
5. rule response: generated by the controller as a reply to the rule request;
6. open path: used to setup a single rule across different nodes.

When a nonbeacon packet is received by the FWD layer, it is sent to the NOS that searches for the corresponding rule in an appropriate data structure called flow table. The flow table stores all the rules coming from the controller. For each rule, there are three types of action that could be executed: forward to a node, modify the packet, or drop it. If a packet does not match any of the rules in the table, a rule request is sent to the controller.

The path between the sink and the node for sending/receiving rule request/rule response packets must be chosen effectively, considering both reliability of the path and its length. Each node constantly stores its distance (in number of hops) from the sink, and the received signal strength indicator (RSSI) that is the level of power it receives from the next hop toward the sink. During the network initialization, each node is in a quiescent state waiting for messages. When the sink turns ON, it sends a beacon, containing the number of hops from the sink (zero in this case). When a node A receives the beacon, it performs the following four operations.

1. Add the source of the beacon and the RSSI received in the list of nodes (neighbors table) that are one hop distant from A.

2. Analyze the distance contained in the beacon and the RSSI of the received message, then compare these values to the corresponding stored values: if the number of hops is lower and the RSSI is higher, the source of the beacon is elected as the best next hop toward the sink, and the values stored in A are updated.

3. The beacon timer is activated and node A will periodically send its own beacon in broadcast.

4. The report message timer is activated: the neighbors table of A is sent periodically to the sink node using the best next hop toward the sink. After each transmission, the list of neighbors is deleted to have an updated view

of the network. The report period must be greater than the period used to broadcast beacon messages (beacon period).

The information included in the report messages are used by the controller to create a map of the network. Based on this representation, the controller is able to respond to rule requests and to decide the routing paths for data packets, while rule request will keep following the previously discovered path. The actual implementation of the controller uses Dijkstra's routing algorithm to solve rule requests. The weight of the edges in the topology representation is a function of the received RSSI.

A possible change in the network is notified to the controller using report messages. As specified above, the controller obtains periodically all the lists of neighbors, according to the report period that is bounded by the beacon period. By decreasing the latter period, a faster responsiveness to environmental changes could be obtained to the detriment of having larger overhead. In the actual implementation of SDWN, the controller sends a rule response only after receiving a rule request from a node and rules contained in the nodes expire after a configurable period of time. Therefore, at the end of this period, the controller receives a new rule requests for the unmanageable packets.

As previously mentioned, more than one action can be executed for an incoming packet, thus achieving the multicast communication. By performing multiple actions, the controller is able to clone an incoming message into multiple outgoing messages. Unfortunately, a drawback of this approach is that the multicast is locally executed by transmitting a series of unicast messages. In other terms, the broadcast nature of the wireless communication is not exploited.

This work has been used as a base to develop an enhanced version of SDWN called SDN-WISE that is described in detail in Chapter 4.

Chapter 3

Comparing IoT protocols

The internet of Things (IoT) is an emergent paradigm evolving around the concept of things (objects, cars, etc.), equipped with radio devices and uniquely addressable. The notion of IoT has been recognized by industrial leaders and media as the next wave of innovation, pervading into our daily life [31, 32]. Sensors are increasingly becoming more pervasive and attempt to fulfill end users needs, thus providing the ease of usability in our everyday activities.

The common standards for IoT applications are ZigBee [33] and IPv6 over low-power wireless personal area networks (6LoWPAN) [34]. Both these standards are implemented on top of the IEEE 802.15.4 standard [30]; however, ZigBee uses 802.15.4 medium access control (MAC) addresses, whereas 6LoWPAN uses IPv6 addresses.

Recently, a third approach based on the software defined network (SDN) paradigm has been proposed [29]. It is called Software Defined Wireless Networking (SDWN) and uses a centralized routing protocol, as already stated in Chapter 2. The coordinator/gateway gathers information on the status of the network of things, and brings this knowledge to a controller that can decide on the exploitation of resources within the wireless network. The controller has a centralized vision of the network of things and can even control things that lie behind several coordinators/gateways. This approach brings the potential advantage of optimal resource exploitation, provided that the overhead is controlled and the environment does not change too frequently.

The aim of this chapter is to fairly compare ZigBee to 6LowPAN and both to SDWN. The three solutions presented above are compared by experiments performed on the European Laboratory of Wireless Communications for the Future Internet (EuWIn) platform.

In particular, the flexible topology testbed (Flextop) facility of EuWIn has been used. Located inside the University of Bologna, Flextop consists of 53 nodes, equipped with IEEE 802.15.4 interfaces. Flextop provides a controllable and a priori known environment for experimentation, thus enabling the fair comparison among different protocols, even though tests are performed at different time instances. This chapter presents results of an extensive measurement campaign evaluating different performance metrics, such as packet loss rate, round-trip-time and overhead generated in the network, considering different network topologies and sizes, payload sizes, and environmental conditions, from static to dynamic.

Results demonstrate that SDWN achieves the best performance in terms of all considered metrics in static and quasi-static scenarios. However, a severe performance degradation has been observed when the changes in network topology are frequent and significant.

The rest of this chapter is organized as follows: In Section 3.1 the related work are reported, Sections 3.2 and 3.3 report the details on ZigBee and 6LoWPAN protocol stacks. Section 3.4 introduces the platform used for the experiments and Section 3.5 describes the experimental setup and parameter settings. Finally, results are shown in Section 3.6 and conclusion is drawn in Section 3.7.

3.1 Related Work

Many research papers deal with the implementation of ZigBee networks. For example, [35, 36, 37] refer to the implementation of a ZigBee network for smart home applications. Ref. [24] measures the impact of Wi-Fi interference over ZigBee networks. An experimental analysis of star and tree ZigBee networks based on commercially available hardware and software is provided in [38], to determine the limitations of technology. Finally, Ref. [39] provides a comparison between ZigBee Pro and ZigBee IP, in terms of latency, where a network is composed of five nodes.

Referring to 6LoWPAN, Ref. [40] presents an implementation over Texas Instruments (TI) MSP430 devices. A star topology with an edge router and three nodes was deployed, and IP addressability features were tested. In [41], a novel architecture for supporting applications in the field of intelligent transportation systems is presented. The implementation and evaluation of different neighbor management policies applied to routing protocol for low-power and lossy networks (RPL) are given in [42]; experiments were conducted on the TU-Berlin TWIST testbed with 100 TelosB motes spread over a three-floor office building. Several papers are also comparing ZigBee and 6LoWPAN: Ref. [43] provides a qualitative comparison, without addressing any quantitative evaluation of protocols' performance. In [44], the authors present a comparative performance assessment of ZigBee and 6LoWPAN protocols for industrial applications. The testbed is composed of four TelosB nodes deployed in a linear topology.

Unfortunately, there are no works in the literature dealing with the comparison of the SDWSN approach and the distributed approach represented by ZigBee and

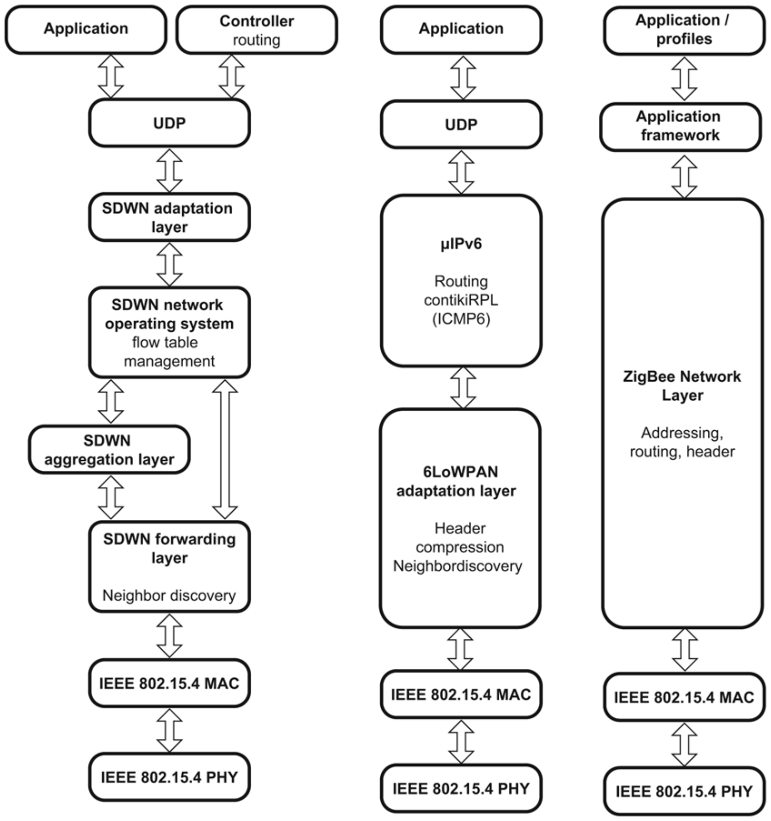


Figure 3.1: Protocol architectures. Left: SDWN. Center: 6LoWPAN. Right: ZigBee.

6LoWPAN.

3.2 ZigBee

In this work, we consider the ZigBee-Pro 2007 release specified in [33], whose protocol stack is shown in Fig. 3.1. The home automation profile is considered, and many-to-one (MTO) routing, described below, is implemented.

MTO routing allows to establish a tree topology, rooted at the coordinator. In order to form and maintain the tree, the coordinator periodically sends an MTO route request (MTO-RR) packet in broadcast. Each node, receiving an MTO-RR before retransmitting it, reads the accumulated path cost (i.e., the sum of the costs of the links of the reverse path toward the coordinator) included in the packet and selects the next hop toward the coordinator. In particular, if a node receives several MTO-RRs from different nodes, it elects as a next hop the node characterized by the minimum total path cost to the coordinator. At the end of this MTO-RR transmission, all nodes in the network are aware of the next hop to be used to transmit their data to the coordinator, that is their parents in the tree. However, if the coordinator wants to know the path to reach a specific node in the network (or a set of nodes by multicasting), MTO routing should be combined to source routing (SR). After the MTORR transmission, once a node has a data packet to be sent to the coordinator, it first sends a route record (RREC) packet through the selected path. Each node in the path receiving the RREC packet, adds in the relay list field its own address and forwards the new RREC packet toward the coordinator. The coordinator analyzes the RREC packet and stores that information in the source route table. Each time, the coordinator has to send a packet to a node, it reads the relay list from this table and sends the packet through the selected path.

In order to let nodes compute the link costs to be used in the MTO routing for the selection of the path, each node in the network periodically sends link status packets in broadcast at one hop. Each node receiving the link status packet computes the link cost, being a function of the link quality indicator of the received packet.

Even though MTO-RRs are periodically sent by the coordinator and are not generated on-demand (which would make the protocol proactive), ZigBee saves the reactive feature through the use of ad hoc on-demand distance vector (AODV) protocol [45], when needed. In particular, in case of link failure, AODV is used for discovering a new path toward the destination. According to AODV, a node searching for a destination node sends a route request packet (RREQ) in broadcast, which is retransmitted by all receiving nodes until it reaches the destination. During the process of rebroadcasting the RREQ, intermediate nodes record in their route discovery tables the address of the RREQ sender, and the corresponding total cost of the reverse path to the source. The comparison among paths' costs of packets related to the same RREQ allows choosing the best path. Once the destination receives the RREQ, it responds by sending a route reply (RREP) in unicast back to the source along the reverse path.

In the case of multicast transmission, a path between the coordinator and the multicast group should be established. In our experiments, we use AODV to establish the route between the coordinator and the multicast group; in this case, the RREQ packet, sent in broadcast, includes the address of the multicast group to be discovered. Nodes in the network that are linked to the target multicast group send an RREP back to the coordinator through the selected path. The latter path is used for the transmission of query packets. In the uplink direction, that is from the queried nodes to the coordinator, nodes use the same protocol as for the unicast transmissions, therefore MTO.

3.3 6LoWPAN

The IETF 6LoWPAN working group published first document in August 2007 [34]. Among the several available 6LoWPAN solutions, the IPv6 stack, implemented in Contiki, has been used and ported on the Flextop platform.

3.3.1 Protocol Stack

The 6LoWPAN protocol stack is shown in Fig. 3.1. The lowest layers are based on IEEE 802.15.4 PHY and MAC layers. Due to the fact that the direct integration between IPv6 and IEEE 802.15.4 lower network layers is not possible, the IETF 6LoWPAN working group has specified an adaptation layer and header compression scheme for transmission of IPv6 packets over IEEE 802.15.4 radio links. The purpose of adaptation layer is to provide a fragmentation and reassembly mechanism that allows IPv6 packets (maximum transmission unit for IPv6 is 1280 bytes) to be transmitted in IEEE 802.15.4 frames, which have a maximum size of 127 bytes of the MPDU (MAC protocol data unit). At network layer, the IPv6 routing protocol for low-power and lossy networks (RPL) is used (see below for details). At the transport layer, user datagram protocol (UDP), providing best-effort quality of service, is applied. Finally, at the application layer, constrained application protocol (CoAP) is present.

3.3.2 Routing Protocol

According to RPL, a destination-oriented directed acyclic graph (DODAG), where each node may have more than one parent toward the root, is built [46]. One of the parents is called preferred parent, and it is used for routing toward the root. In our case, the coordinator acts as the root. The topology is set-up based on a rank metric, which encodes the distance of each node with respect to its reference root, as specified by the objective function. In particular, we use the hop count metric as objective function; therefore, the rank of a given node represents the number of hops separating the node from the coordinator. Paths in the DODAG are selected to minimize the rank. RPL nodes exchange signaling information in order to setup and maintain the DODAG. The construction of DODAG is initiated by the root that sends DODAG information object (DIO) messages to its neighbors to announce a minimum rank value. Upon receiving a DIO message, an RPL node will:

1. update the list of its neighbors;
2. compute its own rank value;
3. select its preferred parent used as next hop to reach the root as the strongest one (i.e., the one from which, it received the largest power); and
4. start transmitting DIO messages, containing its respective rank in the DODAG (a distance to the DODAG root according to the hop-count).

RPL nodes may also send DODAG information solicitation (DIS) messages when joining the network to probe their neighbors and solicit DIO messages. Finally, destination advertisement object (DAO) messages are used to propagate the destination information upward along the DODAG. DAO messages are sent in unicast by the RPL node to the selected parent to advertise its address. When a node receives a DAO, it updates its routing table and then this information is used by the DODAG root to construct downward paths. In our implementation, each router in the path records the route identifier and the corresponding next hop toward the destination. RPL uses an adaptive timer mechanism, called the trickle timer, to control the sending rate of DIO messages. The trickle algorithm implements a check model to verify if RPL nodes have out-of-date routing information. The frequency of the DIO messages depends on the stationarity of the network, and the frequency is increased when the inconsistency is detected. Once the network becomes stable, the trickle algorithm exponentially reduces the rate at which DIO messages are emitted. RPL supports both unicast and multicast traffics. In the case of multicast, we used the stateless multicast RPL forwarding (SMRF) protocol. According to the latter, nodes join a multicast group by advertising its address in their outgoing DAO messages, which only travel upward in the DODAG. Upon reception of message from one of its children, a router makes an entry in its forwarding table for this multicast address. This entry indicates that a node in the DODAG is a member of the group. This router will then advertise this address in its own DAOs, and relay multicast datagrams destined to this address.

3.4 EuWin Experimental Platform

The facility used in this paper, called Flextop [24], is an experimental platform composed of 53 EMB-Z2530PA sensor nodes based on the TI CC2530 single chip 8051 8-bit controller. The TI CC2530 is IEEE 802.15.4 compliant; therefore, our three solutions SDWN, ZigBee, and 6LoWPAN are implemented on top of the

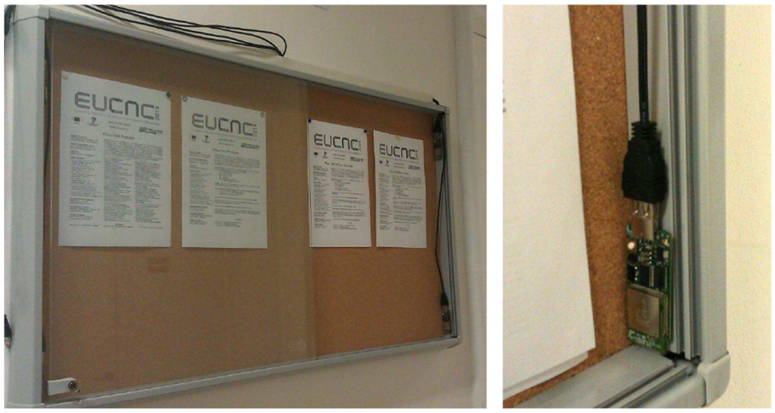


Figure 3.2: Flextop deployment at the University of Bologna.

IEEE 802.15.4 PHY and MAC layers. We consider the 2.4 GHz ISM band PHY, characterized by a bit rate of 250 kbit/s and by a minimum shift keying modulation on top of which direct sequence spread spectrum is applied. In the case of MAC layer, the nonbeacon-enabled mode is used, employing a carrier sense multiple access with collision avoidance protocol (CSMA/CA). We refer to [30] for details about the protocol, and to Section VII-B for the PHY and MAC layers parameter settings in the software.

Nodes are located into boxes on the walls of a corridor at the University of Bologna. Thirteen boxes are deployed in the corridor, and four nodes per box are deployed at fixed positions (see Fig. 3.2). Node 53, at the end of the corridor, acts as the coordinator of the network in all the cases.

The main strength of Flextop is that the experimental environment is stable for the total duration of the experiment, thus making the results replicable, based on:

1. nodes are at fixed and known positions;

2. channel gains between each pair of nodes are measured at the beginning of each test; and

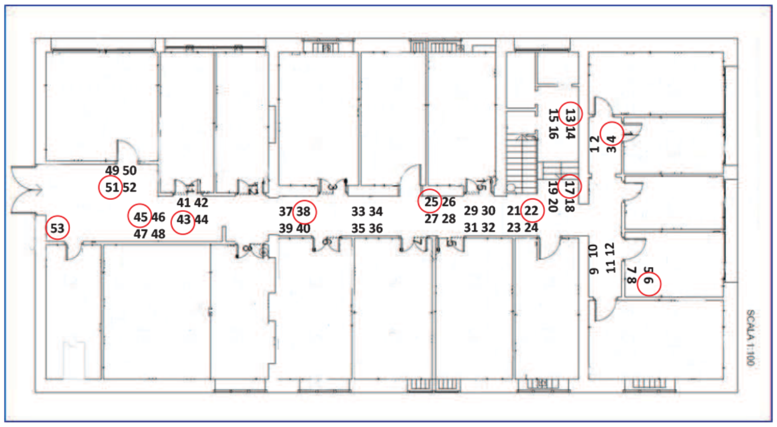


Figure 3.3: Flextop nodes map.

3. experiments are performed during the night, when nobody is present, avoiding uncontrollable channel fluctuations.

With reference to point 2), and in order to properly describe the environment, before the start of experiments, we measured the average received power matrix P. The generic element of P, denoted as $P_{i,j}$, represents the average power received by node i, when node j is transmitting. The matrix is obtained as follows: each node, including the coordinator, sends a burst of 10 000 short packets to let other nodes compute the average power received. We consider two nodes as connected if the percentage of packets received over the link is larger than 90%. Therefore, if more than 90% of packets are received over the link, we compute the average received power that is included in the matrix; if a lower number of packets is received, the two nodes are considered as not connected. The links are rather stable, therefore the 90% threshold is actually not relevant, as links either exist or do not.

Table 3.1: Average Received Powers (dBm) Matrix for -5dBm.

IDs	4	6	13	17	22	25	38	43	45	51	53
4	-	-74	-54	-63	-62	-66	-	-	-	-	-
6	-73	-	-80	-64	-75	-83	-	-86	50	50	-88
13	-53	-81	-	-55	-60	-69	-83	-	-85	-	-
17	-64	-65	-56	-	-63	-58	-86	-82	-85	-89	-83
22	-64	-79	-62	-65	-	-55	-80	-86	-84	-87	-85
25	-66	-84	-69	-57	-53	-	-63	-67	-69	-79	-74
38	-	-	-82	-85	-78	-61	-	-58	-80	-84	-72
43	-87	-88	-	-82	-83	-68	-60	-	-48	-71	-66
45	-	-	-85	-85	-82	-69	-77	-47	-	-54	-59
51	-	-	-	-	-87	-79	-	-80	-56	-	-65
53	-	-	-	-83	-85	-75	-75	-65	-60	-65	-

3.5 Experimental Setup

We consider two network setups: 1) a network consisting of 10 nodes (nodes 4, 6, 13, 17, 22, 25, 38, 43, 45, and 51, underlined with red circles in Fig. 3) and 2) a network of 20 nodes, where we add the following nodes: 1, 8, 10, 11, 15, 20, 23, 30, 31, and 33. In all cases, the node 53 at the end of the corridor, is used as the network coordinator. Nodes were selected according to their level of connectivity, measured by the matrix P described above, to have nodes that could reach the coordinator through different number of hops. The matrix P, characterizing the level of connectivity among the selected nodes in the case of 10 nodes and the coordinator, is reported in Table 3.1, where values are expressed in decibel-milliwatt and where "-" indicates absence of connectivity. The level of transmit power, set to obtain the matrix and used during experiments, was -5 dBm. In the case of 20 nodes, the matrix is not included for the sake of conciseness.

3.5.1 Data Generated and Environmental Conditions

We consider a query-based application, where the coordinator periodically sends a query packet to one or several target nodes and waits for the reply from it/them. Both queries and replies are data packets with a given payload that is the same in both cases, and we consider different payload sizes. Two different communication configurations are evaluated: 1) unicast, where the coordinator sends the query to one specific node that could be one, two or three hops far from the coordinator and 2) multicast, where the coordinator queries contemporaneously a subset of nodes, and waits for replies from all of them. As for the environmental conditions, all experiments were performed during the night, when no people were moving around, to avoid uncontrollable environmental changes and to ensure a fair comparison. However, in order to measure the level of reactivity of protocols to possible changes such as in real environments, we investigated results in quasi-static and dynamic conditions. In particular, experiments were still performed during the night, but we introduced the ?disturbs? specified below. In the case of quasi-static environment, we emulated a day-like situation, where people move around, by letting two people walk along the corridor at a constant speed, following a predefined path. The comparison among protocols is still fair, since we reproduced exactly the same situation (same people, path, and speed) during all experiments. This case is denoted as quasi-static, since only two people were moving without creating huge obstacles and fast fading. In the case of dynamic environment, we emulated the movement of nodes leaving the network and possibly coming back, by switching OFF and on nodes at random instants. In particular, we implemented the following procedure: 1) once a node switches ON, it remains in this state for at least 5 s; 2) after which, it generates a random and uniformly distributed delay between 0 and 10 s; and 3) at the end of which, the node switches OFF for 1 s, and then it switches ON again [back to step 1)]. The comparison among protocols is still fair, since the above described duty cycling is implemented in the tests identically. Moreover, the channel conditions could be considered as extremely dynamic, since nodes switch OFF frequently and at random time instants.

3.5.2 Parameters Settings

All the parameter settings related to PHY and MAC are the same for the three protocols, and they are provided in Table 3.4. It also includes the network layer parameters, different for the three protocols stacks, but set to the same values, when possible. In particular, for the fair comparison, we set the SDWN beacon packets period equal to the ZigBee link status period, as well as the SDWN flow tables refreshing time equal to the ZigBee MTO-RR period. Therefore, when the environment is static, routing tables are refreshed and new paths are discovered with the same frequency (i.e., every 150 s). Broadcast packets used to compute link costs/RSSI values are sent with the same frequency (i.e., every 10 s). Obviously, in the presence of changes in the environment, the two protocols behave differently. In case of 6LoWPAN, as stated above, the frequency of generation of DIO packets is managed by the trickle algorithm: the RPL router will schedule the emission of a DIO at some time in the future, depending on the events in the network and real-time environment condition. In our case, we selected the default period between two consecutive DIO messages equal to 12 s. We refer to the different standards for the setting of the remaining parameters, since, in all cases, they were set to the default values.

In relation to the packet sizes, all protocols use an MAC acknowledgement of 11 bytes and a PHY header of 6 bytes. The MAC header is 18 bytes in the

Table 3.2: Parameter Settings: PHY

PHY layer	
Bit rate	250 kbit/s
Frequency band	channel 11, at 2.405 GHz
Transmit power	-5 dBm
Receiver sensitivity	-92 dBm
PHY layer header	6bytes

Table 3.3: Parameter Settings: MAC

MAC layer	
BE_{min}	3
BE_{max}	5
NB_{max}	5
Max number of retransmission at MAC level	3
MAC header for ZigBee and SDWN	18 bytes
MAC header for 6LoWPAN	14-22 bytes

Table 3.4: Parameter Settings: NET

NET layer	
SDWN	
Beacon packet period	10 s
Report packet period	20 s
Flow tables refreshing time	150
Maximum number of children per parent	6
ZigBee	
Link status period	10 s
MTO-RR period	150 s
MTO-RR number of retransmissions	3
Maximum number of children per parent	6
Random jitter for broadcast packet	(0,127) ms
6LoWPAN	
Minimum DIO period	12 s
DIO period doublings	8 s
Maximum number of children per parent	6
Random jitter for DAO packets forwarding	(0,4) s
Random jitter for DIS packets generation	(30-60) s

Table 3.5: MAC service data unit lengths

SDWN packet type	MAC service data unit length (bytes)
Data	10 + Payload
Beacon	10 + 2
Report	10 + 3 + (3 * no. of neighbors)
Rule request	10 + Payload
Rule response	10 + (16 * no. of rules sent)
Open path	10 + (2 * no. of nodes in the path)
ZigBee packet type	MAC service data unit length (bytes)
Data	15 + Payload
MTO-RR	15
RREC	13 + (2 * no. of nodes in the path)
Link status	13 + (2 * no. of neighbors)
6LoWPAN packet type	MAC service data unit length (bytes)
Data	15 + Payload
DIO	85
DAO	48
DIS	6

case of ZigBee and SDWN, since short addresses are used, while it is 22 bytes for 6LoWPAN in the case of unicast packets (data packets and DAO), and 14 bytes in the case of broadcast packets (DIO and DIS), due to the use of long addresses. The MAC service data unit lengths for the different packets and the different protocols are presented in Table 3.5.

3.5.3 Performance Metrics

We consider the following performance metrics: 1) packet loss rate (PLR); 2) round-trip-time (RTT); 3) overhead; and 4) throughput. In all experiments, the coordinator is sending one query every 300 ms toward the target node(s), and a total number of 5000 queries are generated at the application layer. To compute the PLR, in each experiment, we count the number of replies received at the coordinator nRX from each target node. Therefore, we have a loss if we lose the query or the reply independently from the link in which the packet is lost. In the case of unicast transmission, $PLR(\%) = (5000 - n_{RX}) * 100/5000$, while in the case of multicast, we compute an average PLR, averaged among the target nodes. The resolution of the PLR is approximately 0.5%, since 5000 packets were transmitted. The RTT is defined as the interval of time between the transmission of the query at the application layer of the coordinator, and the instant in which the reply is received at the application layer of the coordinator as well. In order to compute the RTT of each packet, we use a software-defined timer implemented at the application layer of the coordinator, having a resolution of 1 ms. Results are then averaged over all packets received in each experiment, and among the target nodes for the multicast case. Two definitions are used for the overhead. 1) The ratio between the total number of packets transmitted in the network (being data packets transmitted for the first time or retransmitted, acknowledgement, or control packets), and the number of queries generated at the application layer of the coordinator. 2) The ratio between the total number of bytes transmitted in the network, and the number of bytes of information included in the generated replies. We computed the latter by processing the data gathered by two sniffers located at fixed positions at the end (near the coordinator) and in the middle of the corridor. We measured the network throughput by counting the average number of payload bits of the replies per second, correctly received by the coordinator. Finally, note that results related to energy consumption are not provided in this paper. However, being this metric strictly related to both, delays and reliability, the best solution in terms of RTT and PLR is expected to be the best also from the consumption viewpoint.

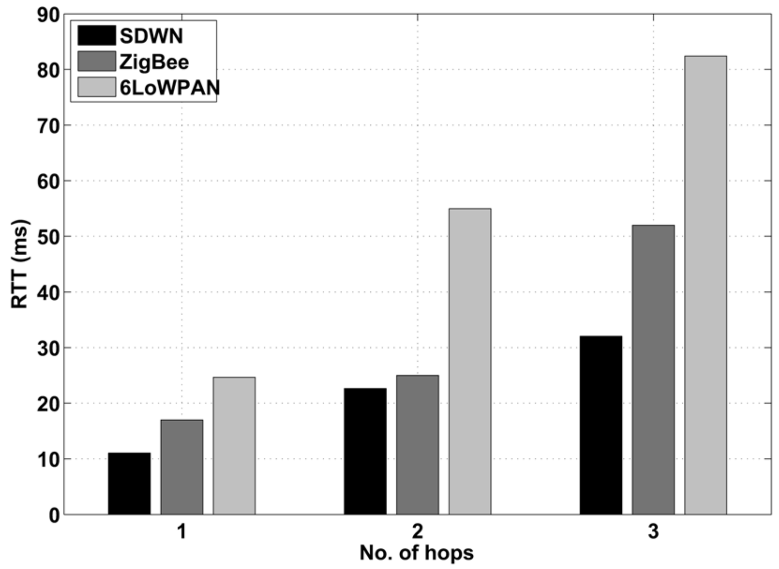


Figure 3.4: Unicast traffic: RTT as a function of the number of hops when transmitting 20 bytes of payload in static conditions.

3.6 Numerical results

In this section, we report the numerical results obtained in the experimental campaign. More specifically, we first provide the results for the static and quasi-static cases, then the dynamic case is addressed.

3.6.1 Static and Quasi-Static Environments

We first compare results among all considered protocols, for the case of static and quasi-static environments. In Fig. 4, we show the RTT as a function of the number of hops for the case of 20 bytes of payload, unicast transmission and static environment. The set of target node(s) is different for the different protocols, since different topologies are generated. In particular, the set of target nodes is reported in Table 3.6, with the

corresponding number of hops and path connecting the node to the coordinator. It can be observed that the node 51 is always directly connected to the coordinator.

Table 3.6: Target node(s) with the number of hops and paths.

Protocol	1 hop target node	2 hops target node	2 hops path	3 hops target node	3 hops path
SDWN	51	22	22-45-53	4	4-22-45-53
ZigBee	51	4	4-22-53	6	6-22-46-53
6LoWPAN	51	13	13-43-53	4	4-25-38-53

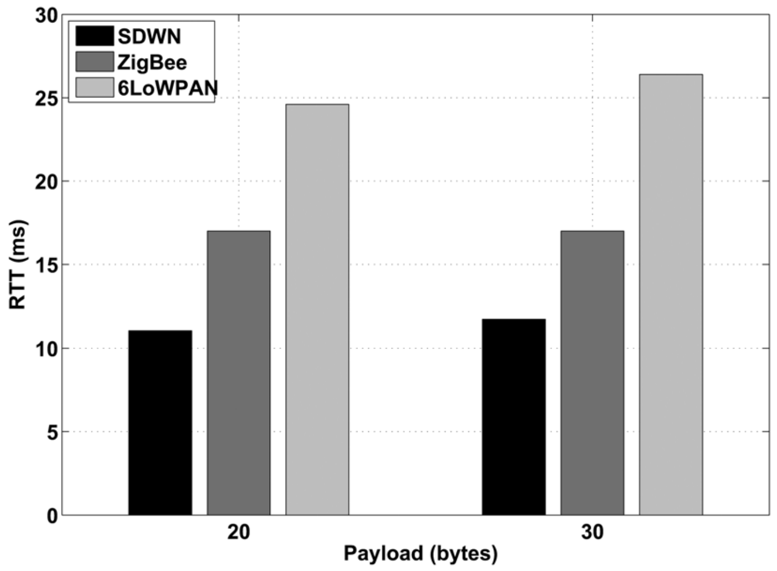


Figure 3.5: Unicast traffic: RTT as a function of the payload size in the case of one hop and static conditions.

Table 3.7: Overhead: comparison among protocols.

Protocol	Packets: 1 hop	Packets: 2 hop	Bytes: 1 hop	Bytes: 2 hop
SDWN	2.6	5.6	2.5	5.6
ZigBee	4.7	8.7	6.5	11.4
6LoWPAN	6.2	9.5	10.9	16.8

For example, the node 4 is connected by three hops in the case of SDWN and 6LoWPAN, while for ZigBee only two hops are needed. As expected, the RTT increases with the number of hops, since the packet has to pass through more routers. In Fig. 3.5, we show the RTT as a function of the payload size in the case of one hop, considering unicast and static environment. We observed that the RTT slightly increases with increasing the payload size. In both figures, we can notice that SDWN achieves better performance than other solutions, resulting in the lowest RTT in all cases. This is due to the fact that in SDWN, once the path between source and destination is established, forwarding at intermediate routers is very quick, since intermediate nodes just have to check the action corresponding to the received packet. In ZigBee and 6LoWPAN, instead, routing must be performed at each intermediate node, resulting in increased delay. Moreover, we can observe that ZigBee notably outperforms

6LoWPAN. The reason is that the protocol stack implemented by 6LoWPAN is more complex (see Fig. 1), resulting in longer processing time, especially at the adaptation layer (implementing addressing and fragmentation). Finally, the packet size in the case of 6LoWPAN is larger due to the use of IP addresses. In Table 3.7, we compare the overhead generated by the different protocols by considering a payload of 20 bytes, static environment, unicast traffic, and different number of hops. As expected, the overhead is almost doubled by passing from 1 to 2 hops.

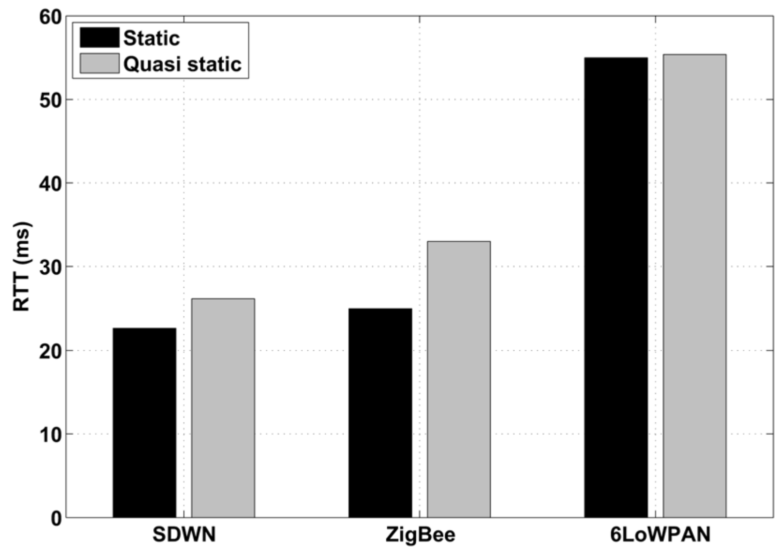


Figure 3.6: Unicast traffic: RTT for the different protocols in the case of static and quasi-static conditions, setting 20 bytes of payload and 2 hops.

Moreover, it is increasing by passing from SDWN to ZigBee and to 6LoWPAN solution. This is due to the fact that, in static conditions, SDWN keeps under control the number of packets transmitted during the path formation phase, while optimizing paths reduces the number of data retransmissions. Referring to the overhead in number of bytes, the difference is also more notable, since headers in SDWN are shorter than in ZigBee and 6LowPAN (see Tables 3.4 and 3.5).

We also want to emphasize that, for all protocols and in all cases, the PLR was below 0.5%.

In Fig. 3.6, we compare the RTT achieved in case of static and quasi-static environments, particularly considering the case of unicast traffic, 20 bytes of payload, and 2 hops. As can be seen, in all cases, the RTT increases when passing from static to quasi-static conditions, due to: 1) the need for searching for new paths when links become unreliable and/or 2) links being unreliable inducing more retransmissions, thus increasing the latency. However, in the considered environment, SDWN still remains the best solution, since the channel fading is still

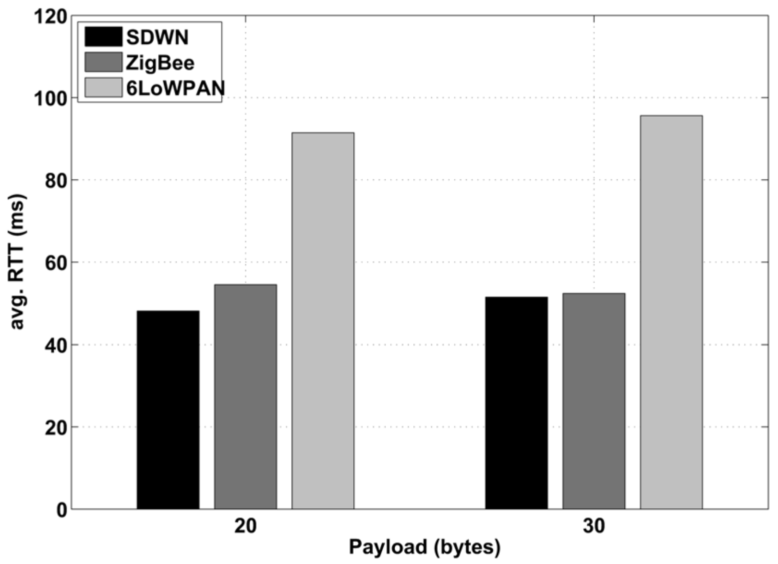


Figure 3.7: Multicast traffic: average RTT as a function of the payload size.

Table 3.8: Throughput (kb/s) comparison: unicast and multicast

Protocol	Uni. 20 bytes	Uni. 30 bytes	Multi. 20 bytes	Multi. 30 bytes
SDWN	0.53	0.8	1.06	1.59
ZigBee	0.53	0.8	1.05	1.57
6LoWPAN	0.53	0.8	0.97	1.43

quite low and changes in the environment are slow, such that SDWN could properly react and work. Finally, note that 6LoWPAN shows the lowest performance degradation when passing from static to quasi-static, since the implemented trickle algorithm allows for better adaptation of routing to environmental changes. Moreover, in the case of quasi-static environment, the PLR remains below 0.5% for all the cases, demonstrating the good reactivity of protocols when the environment changes slowly.

We also report results related to the multicast traffic, when triggering a multicast group that consists of nodes 4 and 6. Fig. 3.7 shows the average RTT, averaged between the two trigged nodes, whereas Fig. 3.8 compares the average

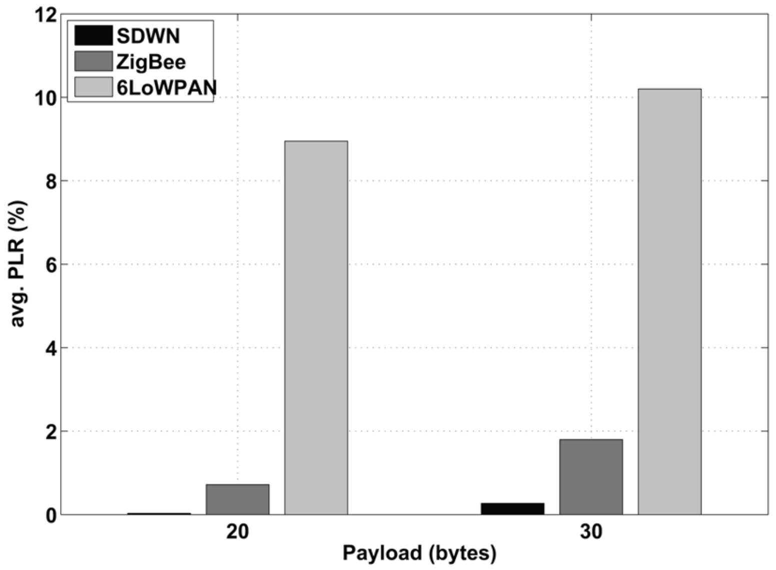


Figure 3.8: Multicast traffic: average PLR as a function of the payload size.

PLR. As can be seen, RTT is much higher than in the unicast case, especially for 6LoWPAN. The latter is due to an increase of the PLR that was below 0.5% in the case of unicast; losses due to collisions between data packets originating from the nodes 4 and 6 that cause retransmissions, and consequently, the increase of delays. However, the multicast traffic increases the network throughput, as shown in Table 3.8. The throughput was computed by considering an offered traffic of one query for every 300 ms. Results demonstrate the improvement of the throughput when passing from unicast to multicast, since more than one node is queried at the same time. Note that, in the case of unicast, the throughput is the same for all the three protocols, since, in all cases, the PLR is lower than 0.5%.

We conclude this section by considering a network composed of 20 nodes (selected nodes are reported at the beginning of Section 3.5), implementing the unicast application with 20 bytes of payload. The coordinator queries node 4, and static and quasi-static environments were considered. Results are reported in Table 3.9, where only the cases of SDWN and ZigBee are considered, having

Table 3.9: Twenty nodes network: comparing RTT and PLR

Protocol	RTT (ms): static	RTT (ms): quasi-static
SDWN	44	49
ZigBee	51	76
Protocol	PLR (%): static	PLR (%): quasi-static
SDWN	1.5	2
ZigBee	13	21.5

Table 3.10: Dynamic conditions: comparing SDWN and ZigBee

Protocol	RTT (ms)	PLR (%)
SDWN	40	96
ZigBee	61	33.5

already demonstrated that 6LoWPAN has the worst performance in all cases. As can be seen, SDWN is again performing better than ZigBee, since environmental conditions are still almost static; therefore, for larger networks, SDWN is also performing well. Obviously, for both protocols, RTT and PLR are larger with respect to the case of 10 nodes network, since more nodes are transmitting packets during the path discovery phase, resulting in more collisions and possibly longer and suboptimal paths.

3.6.2 Dynamic Environment

We conclude this chapter by considering the case of dynamic environment, whose performance in terms of RTT and PLR are reported in Table 3.10. Results have been achieved by considering the 10 nodes network, unicast application, and 20 bytes of payload, where the coordinator queries node 4. In this case, a highly dynamic environment is emulated by making routers switched ON and OFF at random instances of time. This requires nodes to refresh routes very quickly, because a router in a path already established could switch OFF and the source should search for a new relay for reaching the destination. All performance metrics have worsened both for ZigBee and SDWN. However, SDWN reaches a very large PLR, since most of the packets cannot find a proper route to reach the coordinator. The average RTT of SDWN still remains lower than in case of ZigBee, since when a packet manages to find a proper route with all routers switched ON, forwarding is still very quick. This demonstrates that SDWN presents some issues in the case of highly dynamic environments, as expected.

3.7 Conclusion

This work has presented a comparison among different solutions for the IoT paradigm: ZigBee, 6LoWPAN, and a software defined-based solution, SDWN, implementing a centralized routing. Results of an extensive measurements campaign performed over the EuWIn laboratory are reported. Results show that in static and quasi-static conditions SDWN outperforms the other solutions, independently on the network size, payload size, traffic generated, and performance metric considered. The reason for this is the fact that SDWN allows to optimize paths selection and minimize forwarding time at routers. However, SDWN presents some limitations when high dynamic environments are considered, because of the time needed to refresh paths. As a conclusion, we can state that SDWN is more suitable for applications where nodes are in fixed positions and under low mobility scenario, as for the case of smart home and buildings applications. However, when the situation is dynamic and there is a node mobility, a distributed solutions like ZigBee and 6LoWPAN could work better. As an example, the case of smart city applications, where nodes could be mounted over lamp posts in streets where object (e.g., cars and people) are moving around, or where nodes could be directly carried by moving objects, requires solutions characterized by high reactivity rather than lower delays.

Chapter 4

SDN-WISE

As already mentioned, different works have recently appeared aiming at extending the SDN concepts to wireless sensor networks (WSNs) and other wireless personal area networks [20, 29].

By introducing a new solution called Software Defined Networking for Wireless Sensor Networks (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, as proposed in [47] 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.

The rest of this chapter is organized as follows. In Section 4.1 an overview of

SDN-WISE is given. The details of the major features of the proposed solution are explained in Section 4.2, whereas, in Section 4.3 we describe the SDN-WISE prototype we have developed. Performance of SDN-WISE are evaluated experimentally in Section 4.4. Finally, conclusions are drawn in Section 4.5.

4.1 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.

4.1.1 Requirements

Requirements for extending the SDN paradigm to WSNs have been already analyzed in [20] and [29]. 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* [48], that is the possibility to periodically turn off the radio interface of a sensor node, and *data aggregation* [49]. 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 [47].

Since WSNs are intrinsically data-centric, several solutions have been proposed that make network protocols aware of the packet content [50]. 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.

4.1.2 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 later, 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[1]. The optimal choice of such size depends on several deployment specific features set by the WISE-Visor during the initialization phase.

[1]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.

4.1.3 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 4.1.

The protocol stack of the Sink and the other elements implementing the Control Plane are described in the right side of Figure 4.1. 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* [2]. 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 [51, 52].

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

[2]We derive our terminology from OpenFlow [1].

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 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 4.1. 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 Adap-

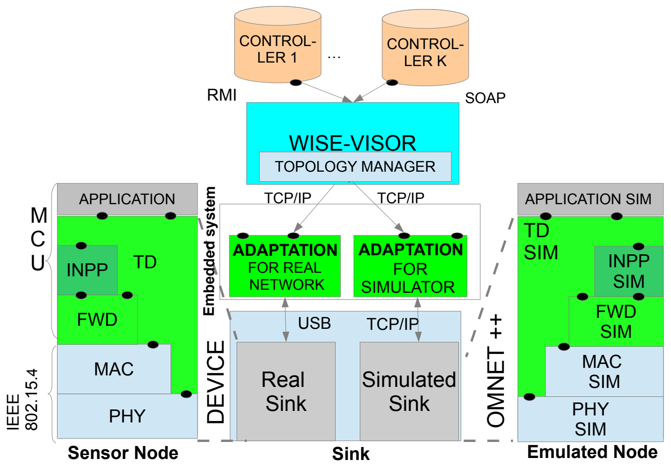


Figure 4.1: SDN-WISE protocol stack.

tation 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 [53].

4.2 SDN-WISE protocol details

In this section we describe in detail the major features of the SDN-WISE protocols. More specifically, in Section 4.2.1 we will explain the Topology Discovery protocol. Then, in Section 4.2.2 we will describe in detail how sensor nodes behave when they receive a new packet.

4.2.1 Topology Discovery

In Section 4.1.3 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 [3] 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 [4]:

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.

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

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

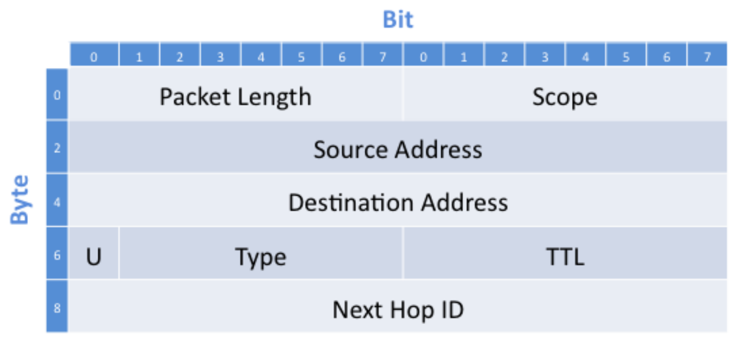
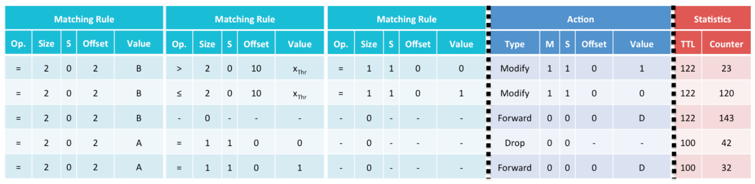


Figure 4.2: WISE packet header.



Matching Rule					Matching Rule					Matching Rule					Action					Statistics	
Op.	Size	S	Offset	Value	Op.	Size	S	Offset	Value	Op.	Size	S	Offset	Value	Type	M	S	Offset	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

Figure 4.3: WISE flow table.

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.

4.2.2 Packet handling

In this section we describe how the Forwarding protocol described in Section 4.1.3 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.

As shown in Figure 4.2, 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 4.1.2).

The structure of the WISE flow table is shown in Figure 4.3 and extends the one proposed in [29].

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 4.3 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

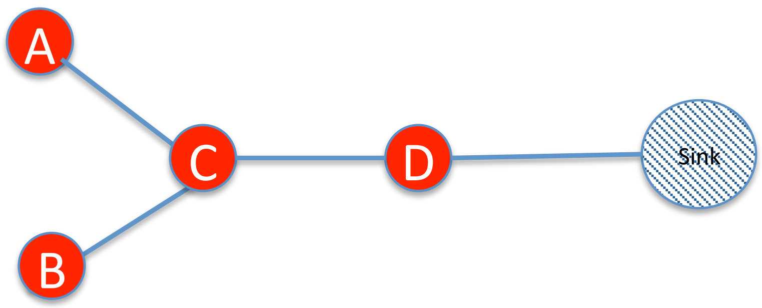


Figure 4.4: Exemplary topology.

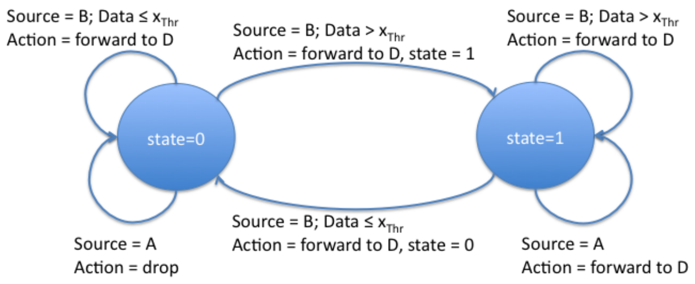


Figure 4.5: Finite state machine implementing a policy such that packets generated by A are dropped if the last data measured by B is lower than (or equal to) x_{Thr}.

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.

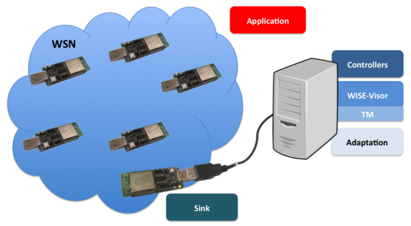
In case the action is “Modify”, the flag S specifies whether the action must be executed on the packet or the state.

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

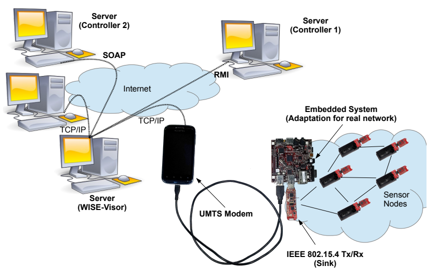
In the following we will show how sensor nodes use their data structures in an exemplary scenario highlighting the specific features of SDN-WISE. Consider the network topology shown in Figure 4.4 and suppose that data measured by sensor A is significant only if the data measured by sensor B is higher 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 4.5 which can be implemented through the five WISE flow table entries shown in Figure 4.3. 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



(a) Simplest deployment option.



(b) Distributed deployment option.

Figure 4.6: SDN-WISE deployment options.

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).

4.3 Prototype and testbed

In our testbed we used the same EMB-Z2530PA described in Chapter 3 Sect. 3.4 in a different testbed inside the University of Catania. Considering the SDN-WISE deployment, it is important to notice that 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 4.6a, 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 4.6b, the sink is a TI CC2500 device attached via USB to a Beagleboard running a Linux operating system (Ubuntu 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 4.6b) 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 using SOAP, REST, RMI or UDP.

Finally, simulations modeling the behavior of the sensor nodes and the sinks can be executed on another PC.

In Figure 4.7 we show a screenshot 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

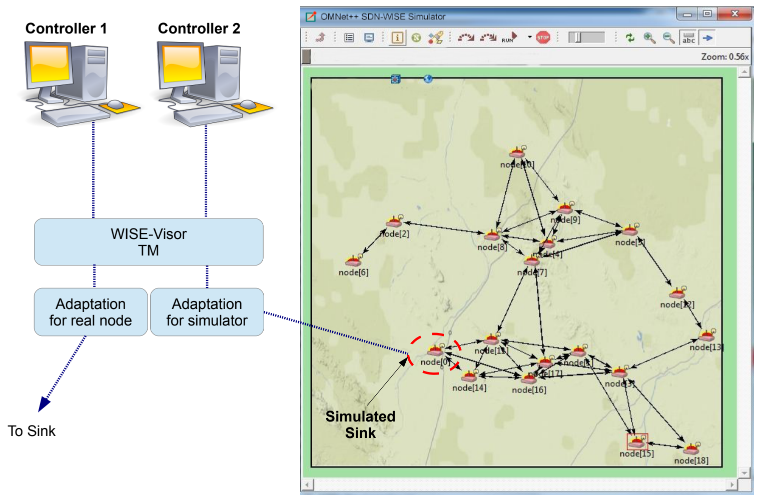


Figure 4.7: Integration with the OMNeT++ simulator.

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

4.4 Performance evaluation

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 4.8. 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 4.6a. 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

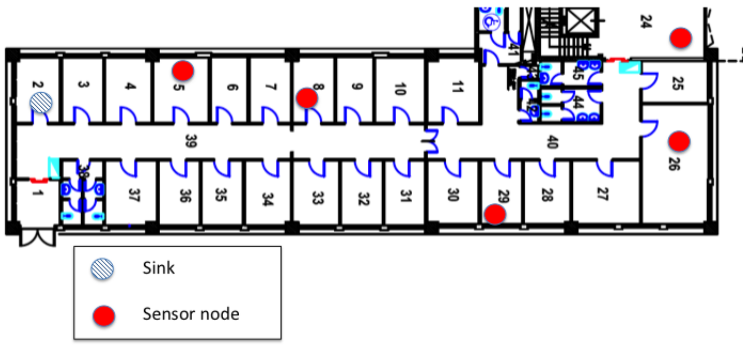
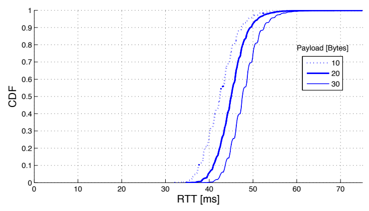
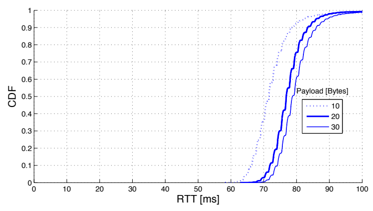


Figure 4.8: Nodes deployment.



(a) Number of hops = 3.



(b) Number of hops = 5

Figure 4.9: CDFs of the RTT for different payload sizes and different distances between the source and destination node.

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.

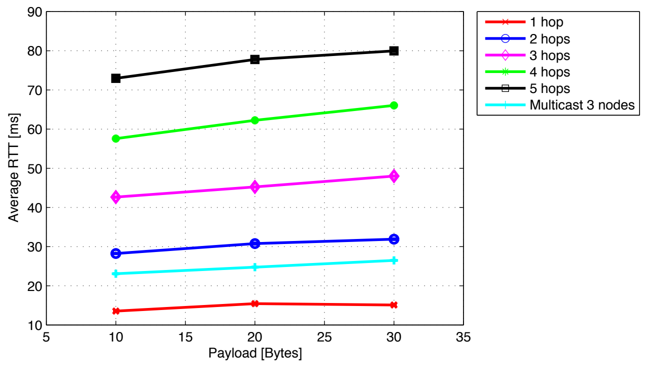


Figure 4.10: Average RTT vs. the payload size, for different values of the number of hops.

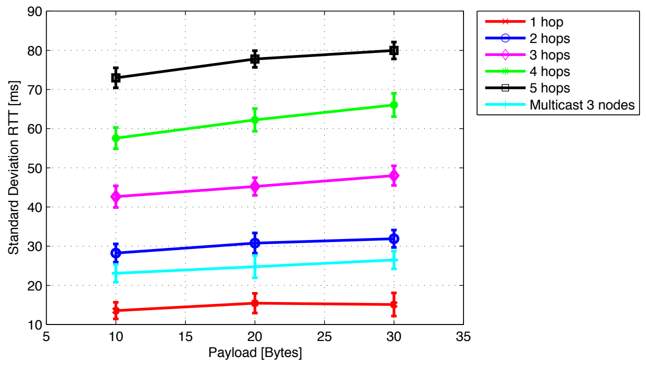


Figure 4.11: Standard deviation of the RTT values vs. the payload size, for different values of the number of hops.

In Figures 4.9a and 4.9b 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 4.10 and 4.11 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 4.10 and 4.11 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 4.12.

The performance in terms of efficiency are shown in Figures 4.13 and 4.14. More specifically, in Figure 4.13 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 4.14 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 4.15 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.

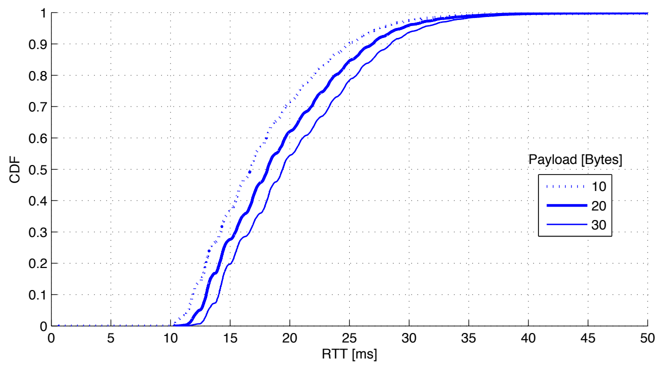


Figure 4.12: CDF of the RTT in the multicast case for different payload sizes.

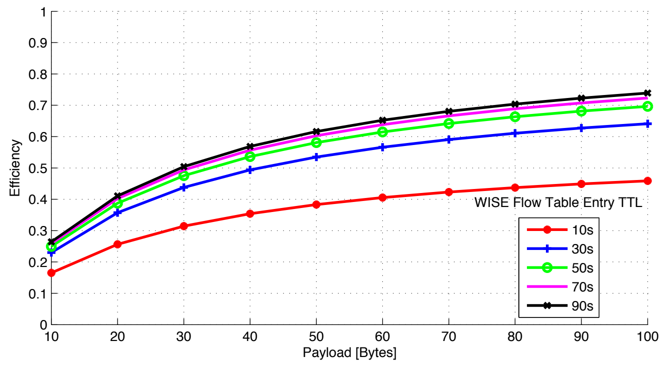


Figure 4.13: Efficiency for different values of maximum WISE Flow Table entry TTL.

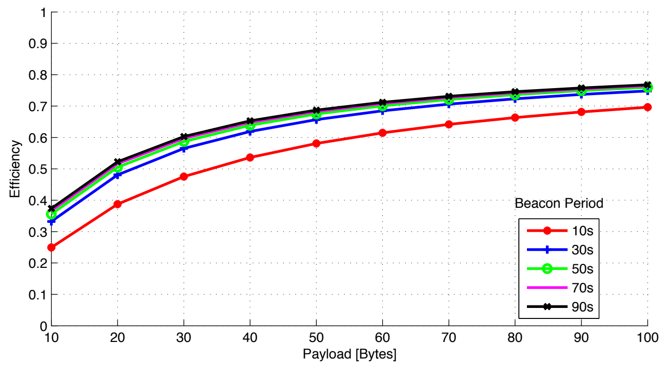
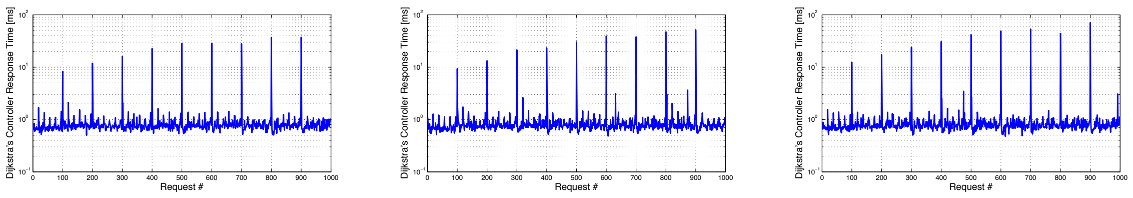


Figure 4.14: Efficiency for different values of beacon sending period.



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

Figure 4.15: Controller response times for different topologies.

4.5 Conclusions

In this chapter 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.

Chapter 5

A Software Defined QoS for IoT devices

The development of new technologies related to the Internet of Things (IoT) resulted in a remarkable increase in the number of Wireless Sensor Networks (WSNs). According to [54], the devices involved are creating an enormous amount of heterogenous data whose sources are extremely heterogeneous and geographically distributed.

Nowadays it is possibile to find WSNs in many places where wired networks are impossible to deploy. For example, WSNs are best suited in museums inside historical buildings where, on the one hand, the architectural constraints limit the deployment of wires, on the other hand, it is necessary to periodically collect data coming from many different types of sensors (thermometers, hygrometers, lux meters, but also accelerometers and smoke detectors) and all of them have different constraints in terms of response time and relevance. Even more different are the requirements in Wireless Multimedia Sensor Networks (WMSNs) where, by including videos and images, strict real-time requirements are encountered [55]. The above examples highlight the relevance of treating different kinds of data differently.

Unfortunately managing these diversities is complex as the networking protocols for WSNs are usually hard to configure. This condition is caused by the peculiar features of the WSNs scenarios where the limited availability of resources, in terms of processing, storage and energy, and the need to support resource demanding applications, requires the use of efficient and effective networking solutions. Thus the main focus is on achieving the best possible performance.

To solve similar issues in the wired domain, the Software-Defined Networking (SDN) approach has been introduced in traditional networking scenarios. As underlined in Chapter 2, in the wireless domain this paradigm has been proposed only recently but thanks to its high flexibility it is a good candidate to deal with the provisioning of QoS policies.

In particular, thanks to the computing architecture of sensor nodes, it is possible to achieve an even higher grade of reconfigurability compared to wired solutions by leveraging a *stateful* approach to control the behavior of nodes upon processing packets, thus supporting differentiated levels of QoS in WSNs. QoS support in wireless networks through the use of SDN has been addressed only lately [56]. In fact, some works addressing the scenario of a software defined wireless network appeared [57, 58], but all of them focus on infrastructured networks, i.e., cellular networks or WiFi. In the past a large number of solutions have investigated the topic of QoS support in WSNs [59] but none of them investigates a software-defined approach.

Therefore, this chapter proposes a novel approach to support QoS in software defined WSNs. To this purpose, we exploit the *state* information envisioned by SDN-WISE. In fact, state can represent the level of congestion of the node and this can be used in a twofold manner:

1. assign different packet drop probabilities to different traffic flows depending on the current level of congestion of a node;

2. inform the Controller about the current level of congestion of a node so that it can calculate alternative rules for traffic flows in order to mitigate congestion.

Performance results obtained by using the OPNET simulation tool assess the effectiveness of the proposed approach. Such results have been found by considering a feasible deployment inside the "Regional Art Gallery of Palazzo Bellomo" in Siracusa, Italy.

The rest of this chapter is organized as follows. In Section 5.1 the literature on the topic is discussed. In Section 5.2 the basic idea of the proposed scheme is presented whereas details of the operations are described in Section 5.3. In Section 5.4 numerical results are presented and, finally, in Section 5.5 conclusions are drawn.

5.1 Related Work

Supporting QoS in a WSN is a very important task to be accomplished, given the ever increasing number of QoS demanding applications in WSNs. The QoS support mechanisms developed for wired networks and traditional wireless networks cannot be applied in WSNs because they are too complex. A complete survey on the QoS provisioning issues in WSN is [60] where Cross-Layer approaches are considered as well. Many of the works described in [60] focus on the integration between the Application and the Network layer, while others like [61] focus on the MAC layer, only. In this chapter we will exploit state information to support QoS in SDN-WISE.

Note that a few papers targeting QoS support in SDN scenarios, recently appeared [56]. As an example, in [62] a new OpenFlow Controller called OpenQoS is proposed. This allows to support dynamic routing where routes are dynamically optimized to ensure QoS for multimedia traffic. In [63] a new QoS framework, called QoSFlow, which enables QoS management in OpenFlow environments is presented. QoSFlow introduces QoS functions to manage QoS resources, e.g., bandwidth, queue size, etc. without changing the SDN architecture. In [64] PolicyCop, a QoS policy management framework based on SDN, is presented, which also provides an interface for specifying the QoS-based service level agreement (SLA). In [65], authors illustrate an Interactive Parallel Grouping Algorithm (IPGA) to ensure QoS which can be implemented through a CUDA system within a SDN Controller.

Concerning the support of QoS in Software Defined Wireless Networks, in [57] *Ethanol*, an example of Software Defined Networking for 802.11 Wireless Networks, was proposed. In Ethanol support of QoS is achieved in a simple way by adding a functionality to the WiFi access points used in the wireless network. This functionality is accomplished by using an additional API for controlling wireless links and for defining the QoS of wired flows. The Ethanol Controller controls this operation. In [58] a set of programming abstractions modeling three fundamental aspects of a WiFi network in a 5G perspective are presented. The focus is in particular on state management of wireless clients, resource provisioning, and network state collection for support of QoS intended as requirements in terms of SINR and bandwidth.

5.2 Basic Idea

The wireless sensor network envisioned in our work consists of sensor nodes that implement SDN-WISE and one (or several) sink(s) that is (are) connected with a Controller. Sensor nodes can generate different traffic flows which can be categorized as belonging to k different priority classes, where C_1 is the highest priority class and C_k is the lowest priority class.

Each node has a transmission buffer that stores data packets before they can be forwarded. The occupancy level of such buffer increases when the traffic that the node must handle increases. Accordingly, the level of buffer occupancy gives information about the level of congestion of the node. Indeed each node is characterized by a congestion state which represents how close a node is to incur in a congestion condition. To this purpose, thresholds of buffer occupancy are defined, each associated to a different congestion state. When the number of packets in the buffer becomes larger (or lower) than each of these thresholds, a change in the state of the node is invoked. The idea is to diversify the packet handling procedure depending on both the current congestion state of the node (i.e. the buffer occupancy) and the priority level of the packets arriving at the node. For each state and for each traffic priority level, the Controller will provide the sensor nodes with the corresponding set of routing/forwarding rules. The proposed approach is in fact aimed at supporting QoS differentiation through a drop probability which changes based on the congestion level at the sensor nodes along the path going from the source sensor node to the sink. For this purpose the proposed mechanism exploits the *stateful* approach available in SDN-WISE.

The proposed mechanism also includes procedures aimed at mitigating the congestion in the network through load balancing. To this purpose the proposed solution leverages the SDN-WISE report message sent by the node to the Controller by adding a field for state information. Upon receiving such messages, the Controller searches for new paths where it is possible to divert the traffic flow and sends the new corresponding rules to the nodes of the network in such a way that alternative paths are used. Nevertheless, the proposed procedure requires only a minor addition to the already existing report message proposed in SDN-WISE and therefore the payload of the report message consists of the following fields:

- *No HOP* (1 byte), which denotes the distance in hops from the sink,
- *Battery Level* (1 byte), which is used to specify the residual battery level of the node,
- *Congestion Level* (1 byte), which is a field added to give information about the congestion state of the node,
- N (1 byte), which is the number of neighbors of the node,
- $Address_n$ (2 bytes), which is the address of the n-th neighbor,
- $RSSI_n$ (1 byte), which gives information about the RSSI towards the n-th neighbor.

The complete report message is shown in Figure 5.1.

5.3 Detailed Operations

An incoming packet at a sensor node, labeled as belonging to a specific priority class C_i, can cause a state transition and, therefore, a change in the behavior of the node. This occurrence depends on the set of rules stored in the WISE Flow Table. A simplified example of this table is reported in Table 5.1. Each entry of

	Bit 0-7	**Bit 0-7**
Byte 0-9	SDN-WISE Header	
10	No. Hop	Battery Level
12	Congestion Level	N
14	$Address_1$	
16	$RSSI_1$	...
18	...	
...	$Address_n$	
...	$RSSI_n$	

Figure 5.1: New SDN-WISE report message.

the table can be divided in two parts: a set of *matching rules*, and an *action*. The matching rules provide the conditions that must be satisfied in order to match the entry. Each *matching rule* consists of four fields:

- *Location*, which is a field that specifies if the operation matching is referred to the incoming packet or to the state array,
- *Offset*, which gives information about the index of the byte in the packet or the state that has to be considered,
- *Operator*, which gives information about the relational operator used to check the rule.
- *Value*, which indicates the value to which it has to compare to check the rule validity.

If the entry is matched, the corresponding *action* will be executed. The latter is specified by two fields: the *type of action* (for example forward, modify, drop) and the *Value* field which gives information depending on the type of action. For example if the action is forward, it has to specify the next hop address and the drop rate.

In this chapter, we propose to use the *state*-already available in SDN-WISE to give information about the congestion state of the node. More specifically, we assume that the state at a node can assume three different values related to as many as $L = 3$ different thresholds of buffer occupancy:

- *Green* (G) when the traffic load is low,
- *Yellow* (Y) when the traffic load is approaching the maximum level that the node can tolerate but did not yet reach it,
- *Red* (R) when the node is fully congested.

As a consequence of the identification of these states, some thresholds are available. In particular, we introduce the T_{GY} threshold to characterize the state transition between Green and Yellow states, and the T_{YR} threshold to identify the state transition between Yellow and Red states.

If the node state is Yellow or Red, in order to support the agreed QoS and mitigate the congestion, more resources are allocated to traffic flows with higher priority level. This can be obtained by using a *dropping policy* which depends on both the current state of the node and the priority level of the traffic flow C_i to which the current packet is related. More specifically, a *drop probability* is used and this is differentiated based on the node state:

- if the node state is Green, no drop is executed, independently of the priority level of the traffic flow C_i of the packet,

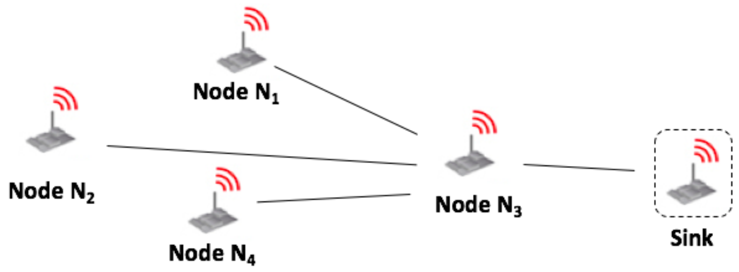


Figure 5.2: Example of network topology.

- if the node state is Yellow or Red, drop is executed according to the priority level of the traffic flow; in particular, the drop probability is inversely proportional to the priority level of the traffic flow.

In the following, we illustrate an example of a sensor network that implements the above described discipline. Consider a network consisting of five nodes (one sink and four sensors) as shown in Figure 5.2.

Let us suppose that node N_1 issues a traffic flow at a rate of 30 kb/s with a high priority level C_1 and node N_2 issues a traffic flow of 30 kb/s with a low priority level C_3. For both of them, the next hop is node N_3. If the relay node N_3 can accomodate at most 50 kb/s, its buffer will fill up and data packets will be dropped. If no QoS policy is implemented, node N_3 will drop packets of nodes N_1 and N_2, regardless of their priority level. On the other hand, by using the stateful approach to ensure QoS, node N_3 will drop packets of nodes N_1 and N_2 with different drop probabilities, depending on the priority levels of the related traffic sources and the current node congestion state. In this way, more resources will be allocated to traffic with higher priority level and QoS can be supported. To illustrate how the node N_3 works, let us consider the WISE Flow Table at node N_3. Table 5.1 shows a representation of a possible WISE Flow Table of node N_3.

For each entry of the WISE Flow Table at node N_3, the three matching rules

Table 5.1: WISE Flow Table for the node N_3 in Figure 5.2.

Matching Rules	Action
PACKET [SRC_ADDRESS] $==N_1$ and STATE_ARRAY [0] $==$ *RED* and PACKET [PRIORITY_LEVEL] $==C_1$	DROP (10%, *Sink*)
PACKET [SRC_ADDRESS] $==N_2$ and STATE_ARRAY [0] $==$ *RED* and PACKET [PRIORITY_LEVEL] $==C_3$	DROP (80%, *Sink*)

check respectively the source of the incoming packet, the current state and the priority level. For example the first row (i.e. entry) is matched if the node N_3 is in a Red state and, accordingly, drops with probability 10% a packet originated by node N_1 with priority C_1. Alternatively, if the second row is matched, node N_3, in Red state, will drop with probability 80% a packet originated by node N_2 with priority C_3. In both cases if the packet is not dropped it is forwarded to node N_4.

5.3.1 Estimation issues

The approach of identifying the congestion state at a node by measuring the number of packets currently enqueued in the transmission buffer, could imply fluctuations in the observation. In fact, the use of instantaneous values of the level of buffer occupancy could create oscillations between the node's states due to possible sudden traffic bursts in the network.

To avoid such oscillations, a filtered estimation of the buffer occupancy is used to control the transitions among node's congestion states. More specifically, upon receiving a packet, a node updates the buffer occupancy, x_i, as follows:

$$x_i = \alpha x_{i-1} + (1-\alpha)b_i \tag{5.1}$$

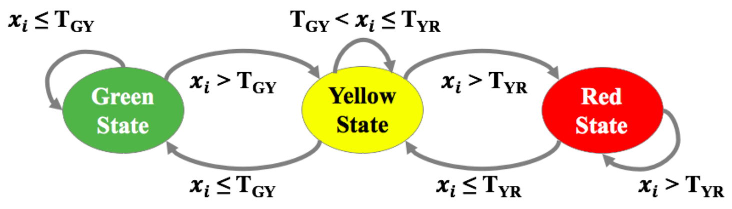


Figure 5.3: Finite state machine implementing a policy such that a node changes state depending on the buffer occupancy x_i.

where:

- b_i, denotes the instantaneous value of the buffer occupancy,
- α, is a coefficient, in the range between 0 and 1, that characterizes the degree of filtering fluctuation. More specifically, if α is low, fluctuations are not filtered; viceversa if α is close to 1, fluctuations are smoothed.

The above filter is the well known Holt exponential smoother [66] which is a very simple and widely used recursive filter characterized by a tunable smoothing parameter, i.e. α. This is indeed a low-pass filter which weights both the recent values, i.e. x_{i-1}, and the instantaneous values, i.e. b_i, so providing a suitable forecasting procedure to be applied to irregular components.

In Figure 5.3 we show the finite state machine implementing this policy: observe that, according to the outcome of the filtering procedure in eq.(5.1), the node can change state based on the result of the comparison with the assigned thresholds.

5.4 Simulation Results

In order to assess the network performance achieved by using the proposed approach for the QoS support, we implemented a real Controller that manages a wireless sensor network modeled using the OPNET Modeler 14.5 [67]. The behavior of the Controller is emulated through an application written in C++

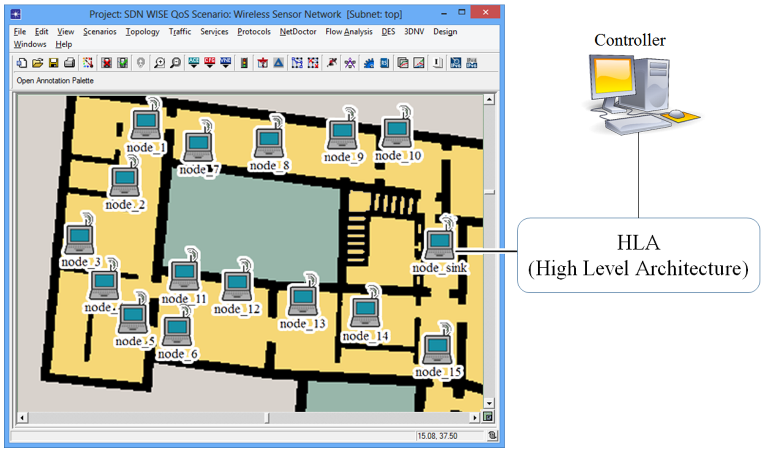

Project: SDN WISE QoS Scenario: Wireless Sensor Network [Subnet: top]
File Edit View Scenarios Topology Traffic Services Protocols NetDoctor Flow Analysis DES 3DNV Design Windows Help
Open Annotation Palette
node_1
node_2
node_3
node_4
node_5
node_6
node_7
node_8
node_9
node_10
node_11
node_12
node_13
node_14
node_15
node_sink
Controller
HLA
(High Level Architecture)

Figure 5.4: Simulation scenario.

and it is hosted on a notebook computer, equipped with an Intel(R) Core (TM) i5-2410 M CPU 2.30 GHz, and 8 GB of RAM running Windows 7, 64 bit. The simulations itself has been executed on another notebook computer, equipped with an Intel(R) Core (TM) i5-3230 M CPU 2.60 GHz, and 8GB of RAM running Windows 8, 64 bit. The Controller interacts with the sink node modeled in OPNET through the High Level Architecture (HLA) technology [68]. This provides all the specifications in order to ensure that two or more simulations can inter-operate with each other to exchange data or information within a simulators' federation. In the OPNET Modeler, we modeled a sensor network consisting of sensor nodes and a sink implementing SDN-WISE with the QoS management feature.

Figure 5.4 shows a possible deployment inside the "Regional Art Gallery of Palazzo Bellomo" in Siracusa, Italy. The network consists of 16 nodes (15 sensor nodes located next to the artworks and 1 sink).

In our experiments, the sink node implements the HLA interface to exchange messages with the Controller(s) and the sensor nodes implement the SDN-WISE paradigm. Each sensor node has a transmission buffer able to store a maximum of 120 data packets. Initially we choose to use two different thresholds for the buffer size: 75 packets to identify a transition between Green state and Yellow state (T_{GY}) and 95 packets to identify a transition between Yellow and Red states (T_{YR}). The simulation lasts 30 minutes and there are six sources which generate traffic as follows:

- $node_1$, $node_2$, $node_3$ generate a traffic of 10 kb/s with priority level C_1, C_2, C_3, respectively, from the beginning to the end of the simulation time,
- $node_4$ generates a traffic of 10 kb/s with priority level C_1 from time 300 s to time 1500 s,
- $node_5$ generates a traffic of 10 kb/s with priority level C_2 from time 500 s to time 1300 s,
- $node_6$ generates a traffic of 10 kb/s with priority level C_3 from time 800 s to time 1000 s.

All traffic flows must be forwarded towards the sink that is the *destination* node for these traffic flows, and the best path to reach it is through $node_7$, $node_8$, $node_9$ and $node_{10}$. Each sensor node, when does not know how to handle a packet, sends a request rule to the external Controller. This will forward a rule response containing rules for each state in which it may be. Table 5.2 reports the drop probabilities used depending on the node congestion state and the traffic priority level. The figure shows also two different options for the congestion level 3: one with low drop probability (*Option 1*) and the other one with high drop probability (*Option 2*).

The simulation campaign is organized in three parts: at first, we simulate a scenario in which the network does not support any kind of QoS, i.e. all traffic flows are treated in the same way. Then we consider the possibility to support

Table 5.2: Drop probabilities in simulations.

			Option 1	Option 2
	Green State	**Yellow State**	**Red State**	**Red State**
C_1	NO DROP	1%	5%	10%
C_2	NO DROP	3%	20%	45%
C_3	NO DROP	5%	40%	80%

QoS but without dynamic routing update, i.e. no congestion notification at the Controller is considered. This approach is denoted as *WCD*. In this case the default path to reach the sink is never changed and thresholds set at each node are not varied. Finally, we consider a scenario in which the network supports the QoS by implementing an SDN-WISE report messages exchange with the Controller to guarantee dynamic routing update in case of congestion at network nodes. In this case the default routing path can be integrated by using an alternative path through $node_{11}$, $node_{12}$, $node_{13}$, $node_{14}$ and $node_{15}$. We denote this approach as *CD*. In the three cases we show the performance achieved in terms of: i) *Dropped data packets* measured as the number of data packets dropped due to congestion; ii) *Drop rate* measured as the percentage of data traffic dropped due to congestion; iii) *Impact of the buffer occupancy thresholds* that is, the drop rate measured with different values of the buffer occupancy thresholds, which cause the change in node's state.

Figure 5.5 shows the dropped data packets obtained by simulating the scenario without QoS support. In this case, as expected, the three curves of the dropped data packets for the flows generated by $node_1$, $node_2$ and $node_3$ have almost the same trend because there is no difference in the treatment of nodes' traffic flows.

In Figure 5.6 we report the dropped data packets obtained by simulating the

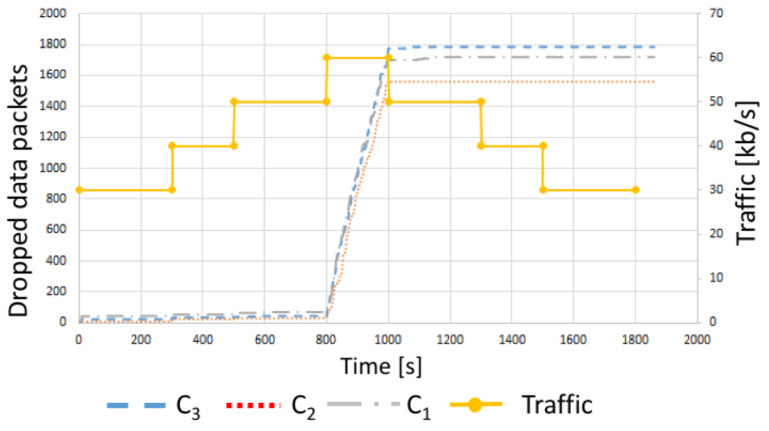


Figure 5.5: Dropped data packets without QoS support.

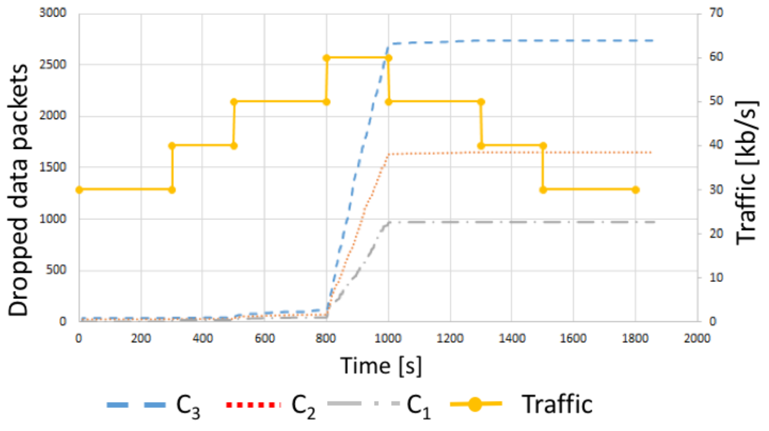


Figure 5.6: WCD: Dropped data packet $T_{GY} = 75$, $T_{YR} = 95$ (option 1).

scenario WCD with QoS support and using low drop probability (Option 1). We observe an improvement for the traffic flow with priority level C_1 as compared to the previous case. In order to obtain a higher improvement for the traffic flow with priority level C_1, the use of higher drop probabilities has been also investigated.

In Figure 5.7 we report the dropped data packets obtained by simulating the scenario with high drop probabilities (Option 2). As expected, the number of dropped packets increases proportionately to the traffic load; in particular it increases exponentially when $node_6$ starts to send data packets, i.e. at time 800 s; this is because when $node_6$ starts to transmit, the full congestion state is incurred

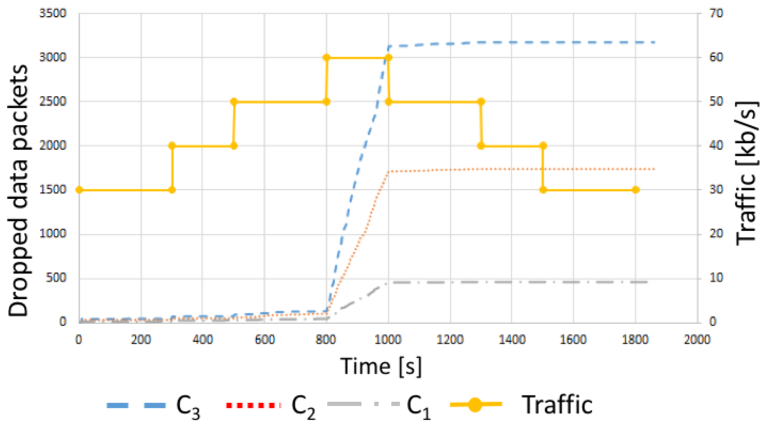


Figure 5.7: WCD: Dropped data packet $T_{GY} = 75$, $T_{YR} = 95$ (Option 2).

at $node_7$. Concerning the QoS, it can be observed that the data packets dropped for the traffic with priority level C_1 is significantly lower than the data packets dropped when the priority level is C_3.

We also performed other simulations by considering two different thresholds for the change in nodes' congestion state. Figures 5.8 and 5.9 report these results. In each figure, we represent the dropped data packets and the traffic obtained for different pairs of values of the thresholds, i.e. $T_{GY} = 65$, $T_{YR} = 85$ and $T_{GY} = 85$, $T_{YR} = 105$. If we choose large thresholds, it is possible to observe that the curves of the three traffic flows tend to get closer with an increase of dropped data packets for the flow with the highest priority. If instead we choose small thresholds, more data packets, in particular for the flow with the lowest priority, are dropped also in a condition in which the network could be able to manage them.

In Table 5.3 we show the dropped data rate obtained for the different thresholds and drop probabilities used. If we choose larger thresholds the drop probability increases for the traffic flow with priority C_1 and decreases for traffic flow with priority C_3.

Another set of results is shown in Figures 5.10 and 5.11 where we report

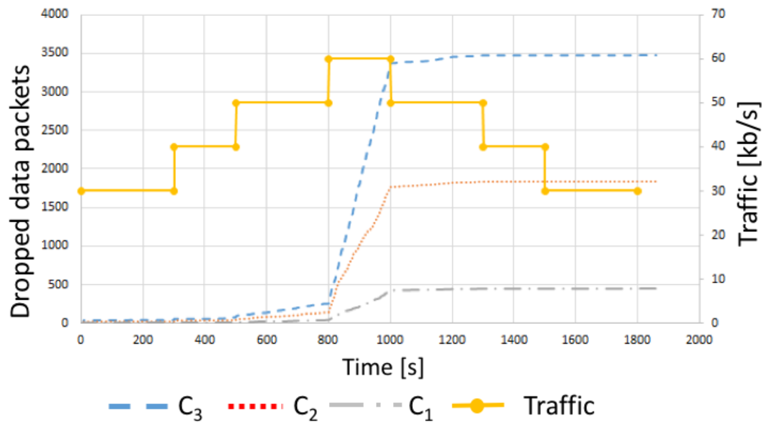


Figure 5.8: WCD: Dropped data packet $T_{GY} = 65$, $T_{YR} = 85$.

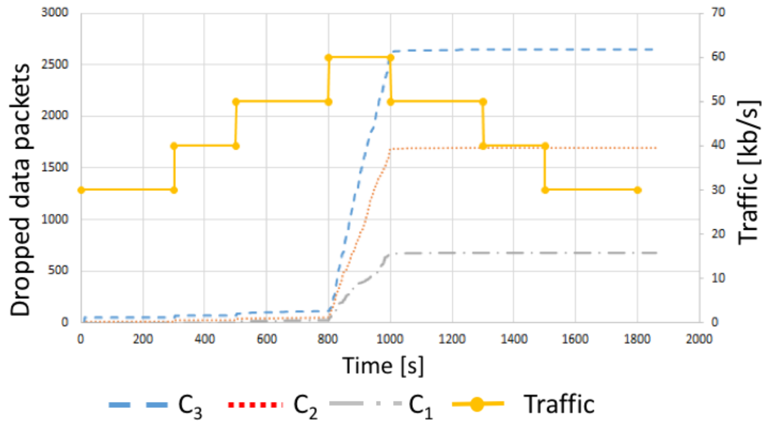


Figure 5.9: WCD: Dropped data packet $T_{GY} = 85$, $T_{YR} = 105$.

Table 5.3: Drop rate results.

	No QoS support	WCD low drop probability	WCD high drop probability		
Thres.	-	75-95	65-85	75-95	85-105
C_1	16.23 %	9.39 %	3.81%	4.17%	6.46%
C_2	16.96 %	15.65 %	16.16%	16.01%	16.44%
C_3	15.97 %	25.33%	30.93%	29.77%	26.26%

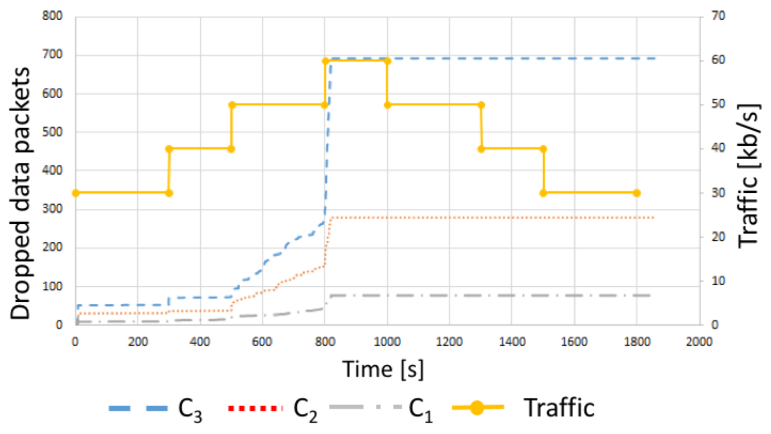


Figure 5.10: CD: Dropped data packet $T_{GY} = 65$, $T_{YR} = 85$.

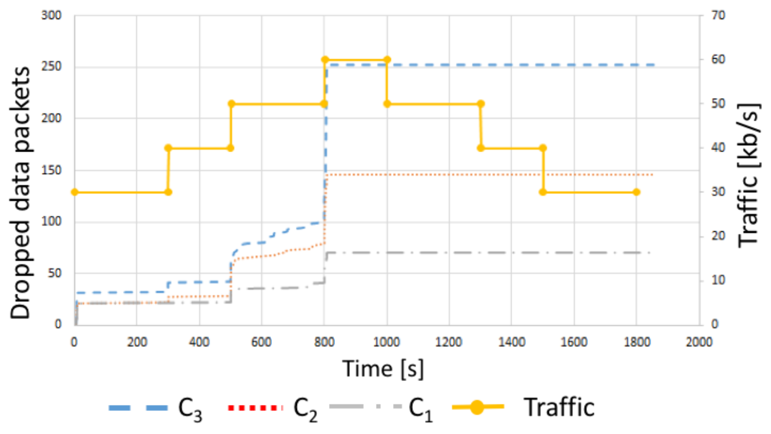


Figure 5.11: CD: Dropped data packet $T_{GY} = 85$, $T_{YR} = 105$.

the results obtained when sensor nodes implement the congestion notification mechanism to the Controller for the different set of thresholds considered. When the congestion notification is received by the Controller, it forwards new rules to drive the traffic flows with the lowest priority level towards another network path. As expected the dropped data packets and the drop rate remarkably decrease.

Unlike the WCD case, the congestion notification mechanism implemented at the Controller allows to solve the congestion in a short time and, as expected, to decrease the dropped data packets and the drop rate.

The solution implemented has been also tested in a scenario with a variable and

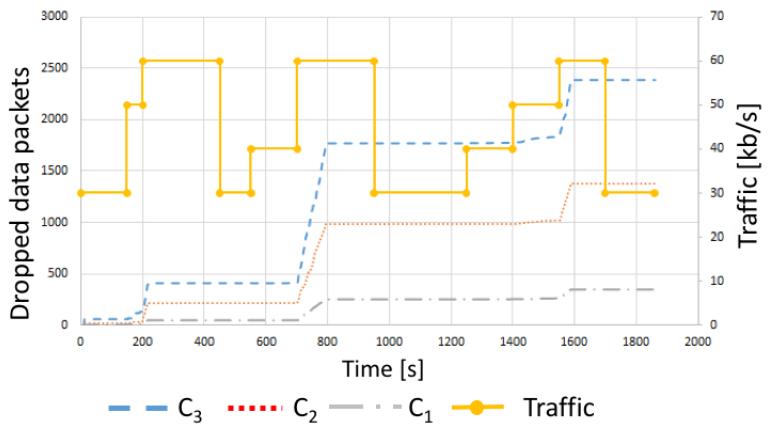


Figure 5.12: CD: Dropped data packet $T_{GY} = 65$, $T_{YR} = 85$, Variable Traffic.

dynamic traffic flow where $node_4$, $node_5$ and $node_6$ transmit in bursts at different time instants. Figure 5.12 shows the results obtained in this case. Observe that also in time varying conditions the mechanism allows to preserve QoS requirements.

5.5 Conclusions

In this chapter we have introduced a mechanism that exploits the *stateful* nature of SDN-WISE to support differentiated levels of QoS in WSNs. The mechanism is based on the usage of state to give information about the congestion condition at the nodes. Each node, as shown by simulations, is able to handle traffic flows with different levels of QoS in different ways. Simulation results assess the effectiveness of the proposed solution to handle QoS.

Chapter 6

Extending ONOS to support IoT devices

As the name itself suggests, heterogeneity is one of the major characteristics of the Internet of Things (IoT): several platforms are available which employ very different technological solutions from each others. Software developers willing to use resources belonging to different platforms have hard time in coping with such heterogeneity. A similar issue in the computer science domain has been addressed leveraging the use of operating systems. Thus, in recent years operating systems (OS)s have been proposed to mask the heterogeneity of IoT devices. Relevant examples of such OSs in the open source arena are Contiki [69] and RIOT OS [70]. By exploiting the above operating system software developers can implement applications which can be used on large number of hardware platforms.

However, such solutions implement specific protocols at the networking layers, usually based on 6LowPAN [34] and IPv6 [71]. Instead, a large body of literature exists demonstrating that there is no one-fits-all networking solution in resource constrained domains like those included in the IoT. Therefore, solutions are needed which allow to program the networking layers in a platform independent way.

In the wired domain several control planes have been proposed along the years to solve similar issues.

The first software solutions for control plane management focused on providing just an easy-to-use interface to network control messages. Into this category we can include control plane softwares like POX and NOX [72, 73]. Subsequent attempts include SANE [74], ETHANE [75], 4D [76] and RCP [77], NOX [], Floodlight [78], and Beacon [79]. Among them TUNOS [80] was proposed to provide open device management, cognitive network status, global network view, virtual forwarding space, applications context management, and general network control APIs designed for user-friendly network programming.

Even though these controllers were successfully applied in the first days of SDN, they were centralized and therefore, have been later replaced by distributed solutions, called Network Operating Systems (NOSs). In addition to traditional control plane solutions a NOS provides new services and capabilities. A NOS is independent from the specific protocol used on the Data plane, by providing a specific Southbound interface to the network devices thus allowing the interaction with different SDN technologies. A NOS also provides a global overview of the network and the ability to replicate and distribute this information for better scaling and fault tolerance. Finally, all these services are provided to network application through a Northbound interface (usually Web GUIs, REST APIs, or a Java libraries).

Here we provide a brief description of the most relevant NOSs:

ONIX [81] is a distributed SDN controller, which identifies the scalability constraints of the aforementioned centralized solutions and builds an architecture, which distributes the network management functionality to several instances. Control planes written within ONIX operate on a global view of the network, and use basic state distribution primitives provided by the platform. However, its development has been discontinued, whereas it has been developed mainly for data centers and it is closed-source, as also described in [15].

OpenDaylight [82] is a distributed network operating system, which has been developed as a collaborative project among several universities and vendors. Using these technologies it is possible to create different southbound plugins for different protocols like OpenFlow, BGP-LS/PCEP, and NETCONF. The northbound of OpenDaylight supports different types of applications/service functions such as Base Network Functions and Service Functions. The first one is related to topology and device management whereas the second one implements a Forwarding Rules Manager for OpenFlow and a PCEP Service function provider for RSVP-TE tunnels. OpenDaylight provides a project, IoTDM, which focuses on the IoT, but it follows the data-level integration approach, rather than a network-oriented solution, which would actually complement existing solutions for networks of switches[1]. More specifically, IoTDM does not enable the direct communication among devices, whereas it relies on application-level protocols, such as CoAP[2], in order to integrate heterogeneous devices. Apart from being a hybrid, rather than a pure network-level approach, IoTDM considers only one of the several existing standards and as it has been documented [83, 84], much effort is still required in order to settle to a particular standard for the IoT.

Finally, the Open Network Operating System (ONOS) has been proposed in [15]. ONOS is based on Floodlight, but it is distributed and provides an extensible, layered architecture, in order to integrate other devices and protocols, besides OpenFlow, which is inherently supported. In fact, ONOS has been considered in

[1] `https://wiki.opendaylight.org/view/IoTDM_Overview`
[2] https://tools.ietf.org/html/rfc7252

the context of this work, due to its scalability properties and its highly modular architecture, which allows the seamless integration of new protocols.

Anyway, regardless the IoTDM solution, all the above network operating systems cannot cope with the specific characteristics of fundamental IoT components such as wireless sensor and actor networks.

For example, they do not consider the limitations in terms of processing and energy resources of such IoT devices. Therefore, they do not handle duty cycles, data aggregation, value based routing, etc. Instead, sensors and actuators are fundamental ingredients of the IoT ecosystem because they generate incessant streams of data that the IoT can use to improve our lives and our businesses in many ways. Sensors, in particular, offer unprecedented access to granular data that can be transformed into powerful knowledge. Integrated analytics platforms will be used to overcome the data burst and avoid that sensor data will just add information overload and noise escalation.

In this chapter we will extend ONOS in order to consider such features.

The rest of this chapter is organized as follows. In Section 6.1, background information on the ONOS architecture is provided. The proposed architecture is discussed in Section 6.2, whereas a software prototype is presented in Section 6.3. Finally, in Section 6.4, the conclusions are drawn.

6.1 ONOS Architecture

Open Network Operating System (ONOS) is an open source, distributed network operating system, which has been implemented for managing network operations following an SDN approach. The managed network elements and operations are abstracted and decoupled from the underlying network architecture,

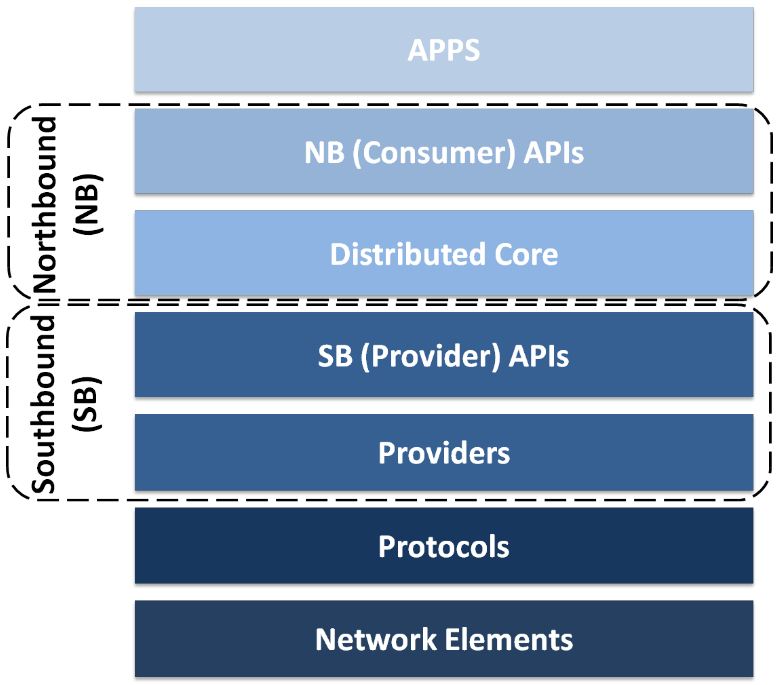


Figure 6.1: ONOS Layered Architecture

in order to enable interoperability across heterogeneous networks.

Even though its design is strongly influenced by OpenFlow, ONOS follows a layered architecture, depicted in Figure 6.1, which allows integration of several network management protocols, under the same abstractions. Components belonging in multiple layers are vertically integrated in the context of the so-called *subsystems*, which are services implementing key system functions. Such functions include the management of devices, links, flow rules, topology and more. Typically, communication from high layer components to lower layer ones is performed using the corresponding APIs (NB/SB), whereas communication from low layer components to the higher layer ones is performed by using events.

The components belonging to the *Protocols* layer are handling communication with the *Network Elements*, which are the devices connected with ONOS (e.g. OpenFlow switches). More specifically, the *Protocols* layer includes the drivers implementing each device communication protocol as well as the association between the devices registered in the system with their driver.

In the southbound (SB) part of ONOS, the *Providers* layer includes the components responsible for translating the abstractions used in higher layers to network management protocol-specific operations and vice versa. In particular, a provider service receives from its south part low-level events and encapsulates them in the proper data format to deliver them to higher layers, where the corresponding information will be further processed or stored. Higher layer components, are using the *SB (Provider) API* to invoke actions necessary to translate the high level abstractions to protocol-specific formats before sending them to the network elements.

In its northbound (NB) part, ONOS includes the components managing the abstractions of the network elements and the available operations. In particular, abstractions of flow rules, incoming and outgoing packets, network devices, hosts and more are represented by dedicated APIs in the *NB (Consumer) API* layer. Then, these abstractions are used in the context of the *Core* layer, which provides access to all information maintained by the system, including the devices, the topology and the links, whereas it offers additional functionality based on this information, such as the calculation of forwarding paths between devices deployed in a certain topology.

Finally, the *Applications* layer components leverage information regarding the network status, provided through the NB API, and performs complex actions requiring several subsystems, such as packet forwarding. Typical ONOS applications are triggered upon specific events coming from the network elements, then, based on certain criteria, they are generating the appropriate flow rules that they send

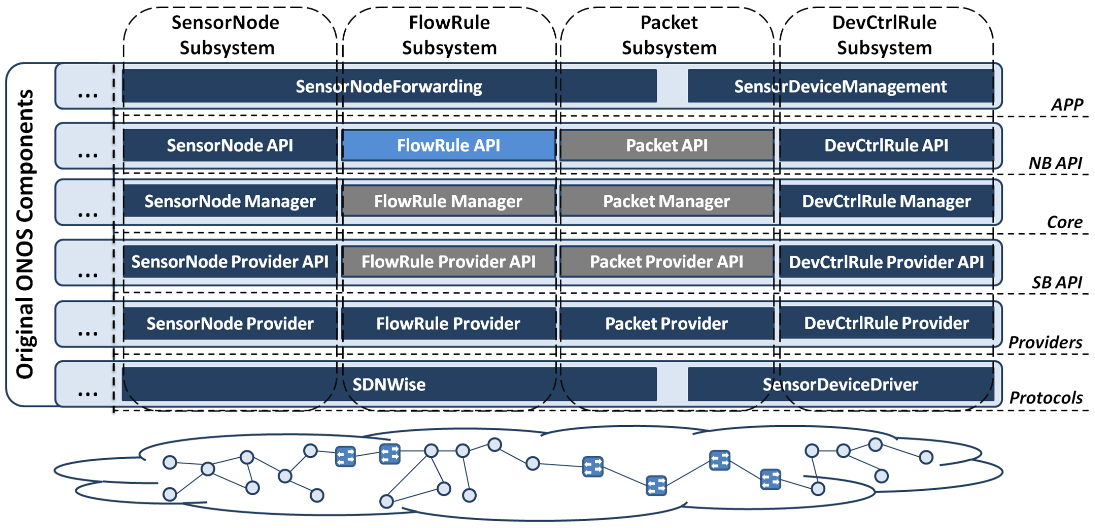


Figure 6.2: ONOS Extended Architecture for IoT

back to the network elements through the corresponding subsystems.

6.2 Proposed Architecture

The proposed architecture is integrated into ONOS, enabling the seamless integration between SDN-WISE wireless sensor networks and OpenFlow networks. Figure 6.2 depicts the components required in each one of the ONOS layers in order to achieve both network flow and device control in WSN. Components marked with dark blue have been developed from scratch; the FlowRule API, marked with light blue, has been extended in order to support SDN-WISE features; the rest of the components marked with gray color have been kept in their original ONOS version, even though interacting with different implementations. In the following sections, the proposed architecture will be explained, by following a top-down approach. The components of the northbound and southbound parts will be explained in the context of the following subsystems that they form: *SensorNode Subsystem*, *FlowRule Subsystem*, *Packet Subsystem* and *DeviceControlRule Subsystem*. Applications and protocols will be separately explained, as they participate in several subsystems.

6.2.1 Applications

ONOS *Applications* layer includes two new components, which perform packet forwarding in SDN-WISE WSNs and remote sensor device management for the devices that have the corresponding driver integrated with the system. More specifically, the *SensorNodeForwarding* application processes incoming packets from both SDN-WISE nodes and OpenFlow switches and installs the appropriate forwarding rules to the corresponding devices. When a message arrives at the application, the latter will first identify the source and the destination network devices and then, send a request to the built-in *PathService* to calculate the best path, by considering the global topology, consisting of both OpenFlow and SDN-WISE nodes. Then, depending on the device type (SDN-WISE or OpenFlow), the corresponding FlowRules API will be used in order to create and install the forwarding rules to the devices. At this point it should be noted that it is not necessary to use the extended FlowRules API for forwarding a packet to the next hop even in a WSN; however, as SDN-WISE has been designed specifically for WSNs, it provides functionality, like setting the whole path from source to destination in one message, that cannot be directly supported with the current ONOS features.

The *SensorNodeDeviceManagement* application goes beyond the standard ONOS functionality, which is focusing on the network flows and enables management operations of the networking devices themselves. We believe that this functionality is vital for a network operating system, since the network is not only about packet transport, but also about the devices performing it. This becomes even more evident in a WSN, where the networking devices have limited resources and their proper management may increase their effectiveness as well as their lifetime. Currently, the *SensorNodeDeviceManagement* application provides some simple primitives to external services in order to perform basic operations on the devices, such as turn them on and off. However, we envision the use of this component in more complex scenarios, where we will optimize resource usage by fine tuning each individual networking device from ONOS, which has a global view of the network requirements in every timeslot.

6.2.2 SensorNode Subsystem

The *SensorNode Subsystem* introduces new components in both the northbound and the southbound part of ONOS and it is responsible for handling information regarding the sensor nodes connected with the system. More specifically, when a message is received by a sensor node in the *Protocols* layer by *SDNWise* protocol, the latter will first extract the message source (note that ONOS will receive the message from the sink, however the source can be any node) and then, raises an event to notify the *SensorNodeProvider* that a device has possibly arrived in the system. Then, the *SensorNodeManager* is notified with all information accompanying the sensor node in order to update the description it currently holds. Such description includes only technical information of the node, such as serial number and operating system version. The *SensorNodeManager* will have to update the *SensorNodeStore*, which is used as an assisting service for persisting data, with the new information. This operation, even though seeming redundant, is very important for managing a WSN, where the network status can be unstable, since it is necessary to keep the network operating system up-to-date on the nodes' status, so that it can make the right decisions, e.g. for packet forwarding. For this reason, the *SensorNode Provider* also offers functionality to actively check the status of a sensor node, instead of waiting for its beacons. This operation is provided through the *SensorNodeProvider API* and it is useful in case higher layer protocols require accurate information on a node status (e.g. battery level or connectivity), before taking a decision and installing a rule.

All data handled in the context of the *SensorNode Subsystem* are accessed through the *SensorNode API*, which introduces in ONOS new data structures designed for the representation of sensor nodes, as the current developments are only considering switches as networking devices. The sensor-specific APIs provide access to information such as the battery level of a node, its neighborhood and the signal strength for each neighbor, its coordinates in 3-D space as well as specific data structures for sensor node address representation.

6.2.3 FlowRule Subsystem

The *FlowRule Subsystem* combines existing and new components. More specifically, the new components have been introduced to address requirements that are specific for WSNs and SDN-WISE in particular. This way, the *FlowRuleProvider API* has been implemented considering SDN-WISE flow tables and messages, since the existing ONOS implementation is provided specifically for OpenFlow, which has different structures. However, the use of the same API allows the transparent use of both the implementations, by leveraging an ONOS Providers layer feature that binds a service with a specific naming scheme throughout the whole platform. Therefore, flow rules meant to be sent to sensor nodes will be always sent through the new provider via the existing API. This is the functionality that enables the *SensorNodeForwarding* application to create and install the rules transparently, as described earlier in Section 6.2.1.

In the *NB API* layer, the *FlowRule API* has been extended to support rules containing conditions and actions that are specific for sensor networks, whereas it has been mostly inspired by SDN-WISE. Most importantly, it adopts the SDN-WISE flexibility to create traffic matching criteria on variable areas in a network packet, instead of using fixed addresses, like the existing API. This is needed, since the IoT is characterized by the dispersiveness of architectures and therefore, it is both unrealistic and inefficient to include every possible option in the API. Rather than that, we provide an extensible component, which can easily be adapted to the specific requirements of each underlying platform. The existing *FlowRuleManager* can handle the extended actions, so nothing has been added in this context.

6.2.4 Packet Subsystem

The *Packet Subsystem* is extended only in the *Providers* layer, since the data structures already included in the other layers were generic enough to address the requirements of an IoT ecosystem. The *PacketProvider* implemented for SDN-WISE WSNs is responsible for handling all possible message types arriving at ONOS. Even though, there are several message types handled in this context, in this section we are going to present the two most common ones: the *Report* and the *Data* message. *Report* messages carry information regarding the status of a sensor node, such as its battery level, its neighbors and the received signal strength indication (RSSI) with each one of them. This information is very critical for WSNs network operations, such as routing and therefore, ONOS should maintain a consistent view. Therefore, the SDN-WISE-based *PacketProvider* updates the appropriate services upon the reception of such messages. *Data* messages are in fact the typical forwarding requests. The *Packet Provider* wraps them in the format provided by the *Packet API* and delivers them to the packet processing applications, an instance of which is *SensorNodeForwarding*.

6.2.5 DeviceControlRule Subsystem

The *DeviceControlRule Subsystem* introduces a new modus operandi in ONOS, by extending its scope to incorporate control not only of the network flows, but also of the network devices. A major design difference, compared to the standard ONOS operation, is that actions in this context may be triggered *proactively*, without any prior request from a network device. This can happen because the operating system may decide that certain device operations may be unnecessary or even impeding other devices operations. This behavior is mostly expected in wireless networks, due to shared medium and power limitations and therefore, the design of this subsystem has been largely based on requirements drawn from this area.

The *DeviceControlRule API* in the northbound provides the device treatment instructions, with respect to the existing APIs for traffic treatment. Currently, such actions include turning the device on and off, setting the transmission and reception power, loading an executable function and updating the device about its coordinates. The rules created based on these instructions are passed to the *DeviceControlRuleManager* in the core layer, which is responsible for keeping statistics on the rules and then, forwarding them to the lower layers in order to be sent to the appropriate device.

In the southbound part, the *DeviceControlRuleProvider* receives requests from higher layers, through its API, which is specified in the SB API layer, and creates the corresponding messages to be sent to the devices. Then, it uses the *SensorNodeDriver* to send the messages to the sensor nodes. Note that, currently, the flow of operations in this subsystem is only top-down, as no events are expected to come from the network.

6.2.6 Protocols

SDNWise protocol has a double role: on the one hand it provides access to SDN-WISE-specific data structures, so that ONOS can understand the message format required for communicating with the sensor nodes and, on the other hand, it maintains a mapping between the sensor nodes and their communication driver. Its main responsibility is to listen for incoming sensor connections, pass incoming messages to the higher layers and send messages generated in higher layers, such as flow control rules, to the sensor nodes.

SensorNodeDriver handles the low-level communication with the sensor devices. Its implementation depends on the operating system running on each connected device, such as Contiki or RIOT. In the same respect as *SDNWise*, this protocol provides access to device-specific message formats in order to properly translate the instructions specified in the context of the *DeviceControlRule API.*

6.3 Prototype Implementation

The software prototype provides a proof-of-concept implementation of IoT integration in ONOS. In particular, it enables packet forwarding across OpenFlow and SDN-WISE networks through a single application, which considers a holistic view of the network provided by ONOS in order to decide the routing path.

More specifically, the scenario presented in this section shows how the proposed architecture can support communication between a sensor node and a host, which can in principle represent a service running in a data center, which stores the information generated by the sensor network. As shown in Figure 6.3, suppose that the sensor node belongs to an SDN-WISE network and its address is 01:00:10, whereas the host is connected to an OpenFlow switch and its address is 10.0.0.1.

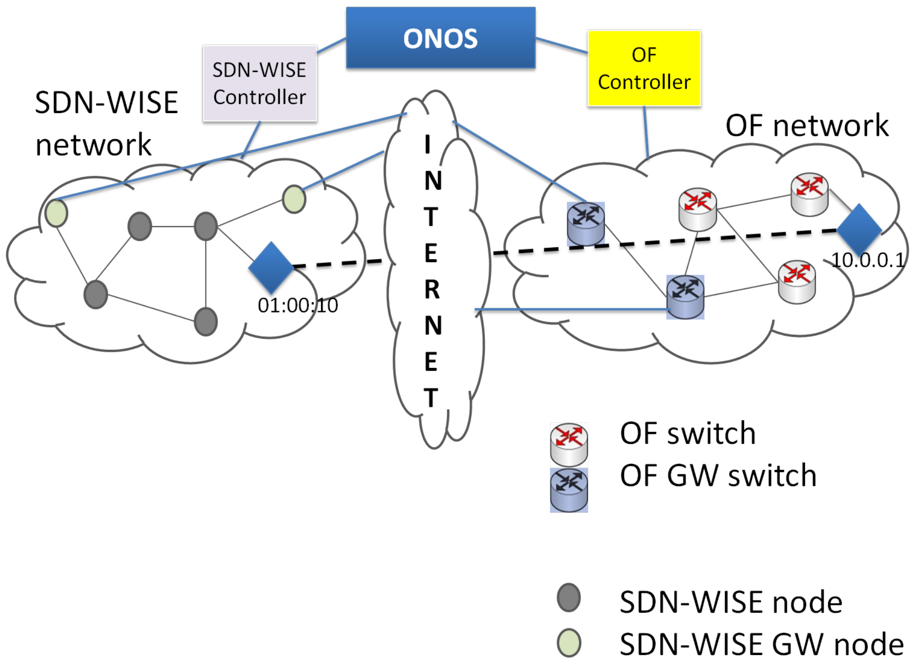


Figure 6.3: Integrated Network Scenario.

Both these networks will be represented inside ONOS using common, generic abstractions for devices and links, which allow for the creation of a single topology view of the whole system.

The result is shown in Figure 6.4, which shows ONOS representation of the topology through its built-in user interface. The gray nodes represent the SDN-WISE nodes, the blue ones the OpenFlow switches and the black ones, the hosts connected to the switches. In order to make the scenario more interesting, we suppose that there are two gateways between the networks and that the sender sensor node is in the middle of the SDN-WISE network, which means that it can use both gateways at the same cost. In the standard operation, the sender would select one of the gateways to send the packet and then, the gateway would send it to the host, regardless the location of the latter. By using the proposed ONOS extended architecture though, the path decision algorithm will be executed

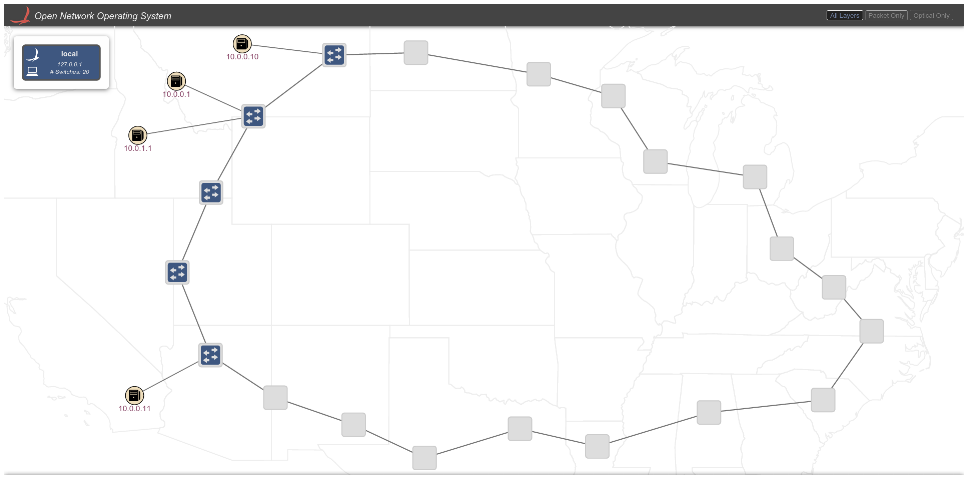


Figure 6.4: ONOS Snapshot of the Prototype.

considering the whole topology and therefore, the selection of the gateway will be made in an optimal way.

Another advantage of this approach is that the gateways do not need to be aware of the services that data should be transfered, which can become very complex when several providers require access to different parts of the information generated by the WSN. By using ONOS, each application can set its own forwarding rules and the network will perform forwarding accordingly.

6.4 Conclusion

In this chapter an architecture of a Network Operating System for the Internet of Things has been presented. The proposed architecture starts from ONOS and extends its current scope, which mainly covers OpenFlow networks, to enhance WSNs. As a result, interaction between SDN-WISE and OpenFlow networks becomes seamless, with the NOS deciding the forwarding paths considering the whole topology and providing the appropriate commands for each device type.

Chapter 7

Implementing the IoT Vision using ONOS

The fragmentation of the IoT landscape, due to the deployment of many *Intra*nets of Things, which cannot cooperate effectively and efficiently, instead of a unique *Inter*net of Things [85], sets interoperability constraints despite the integration efforts [83, 86].

Even though the SDN-driven approach is definitely a step towards the vision of an easily (re-)configurable IoT environment, as also outlined in [84], there is still a major issue that must be addressed. More specifically, traditional SDN solutions focus on device-level protocols, thereby providing isolated views of the individual network segments consisting an IoT ecosystem.

In fact, considering for example that the network of switches of Fig. 7.1 supports OpenFlow [1] and that the WSN supports SDN-WISE or Sensor OpenFlow [20], there will be two different SDN controllers managing each network segment. Bearing in mind the large number of network segments involved in a standard IoT ecosystem, it is obvious that, compared to the existing application-level integration solutions, this approach introduces very high complexity and therefore, does not scale.

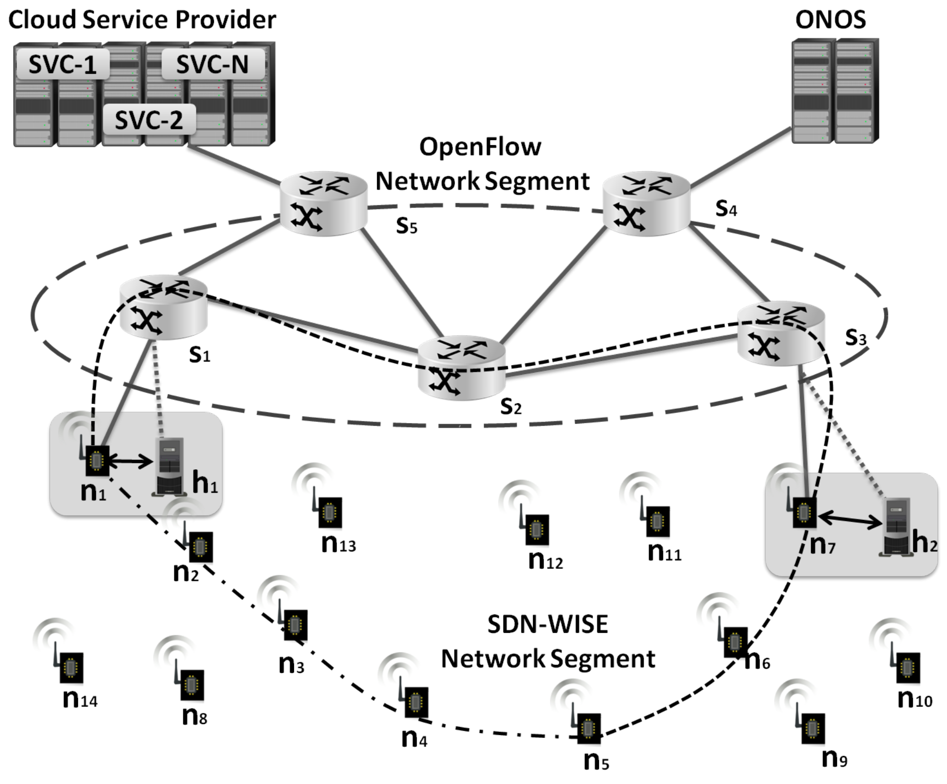


Figure 7.1: Typical IoT Ecosystem.

Actually, a similar problem arose in the first days of SDN even for fixed networks, where several controllers existed, each one optimized for different scenarios. In order to tackle with this issue, the traditional SDN controllers evolved to become *Network Operating Systems* (NOSs), which provide high-level abstractions for several heterogeneous devices and protocols and can be used to manage even large networks. The Open Network Operating System (ONOS), depicted in Fig. 7.1, is an example of such a system. Unfortunately, NOSs have not been sufficiently exploited in the context of WSNs, furthermore, no attention has been paid to the network-level integration of IoT devices.

In this chapter, we leverage the developments in NOSs and we specify a unified system, which controls an IoT ecosystem at the network layer in a holistic way. More specifically, our contributions are as follows:

- We provide generic abstractions for sensor nodes as well as encapsulations of non-IP packets, in order to enable IoT service providers to deploy their network configuration over *any* allocated infrastructure, transparently.
- We propose an architecture enabling the IoT service providers to create networks of sensors and switches, while supporting packet forwarding through different network segments.
- We develop a framework which supports a shift in the IoT communication paradigm from device-to-cloud to device-to-device, allowing the direct communication of sensor nodes over networks of switches. In this way, *clouds of sensors* can be created in an ad-hoc manner, exploiting subsets of the already deployed WSN devices, thereby fostering granularity and reusability.
- We introduce the *Software Sensor Nodes* concept which acts as an integration

element for different IoT protocol stacks so enabling the communication of devices adopting different standards. This is achieved by exploiting both the extensible nature of the NOS, which allows the implementation of any device-level protocol, and the common abstractions for sensors and rules, which are enforced, irrespectively of the underlying protocol.

Note that in this chapter we are not proposing a new protocol (or set of protocols) which should replace existing ones, like ZigBee and 6LoWPAN. Instead, we are proposing a general approach that leverages the existence of SDN solutions in both the infrastructured and infrastructureless segments of the IoT to provide application developers with abstractions of the whole network and the related services that can be easily utilized in efficient and flexible manner.

In this context, we have focused on SDN-WISE for the infrastructureless segment because it is stateful and, as a consequence, Turing complete. Therefore, it can implement any protocol if correctly instructed by the corresponding network application. For example, we have implemented a network application that programs the sensor nodes so that they execute geographical routing [87] and we are implementing another network application that enables nodes to interpret and generate typical 6LoWPAN signaling messages such as Router Solicitation, Router Advertisement, Node Registration and Node Confirmation so that interaction with standard 6LoWPAN nodes is possible.

As a whole the proposed solution has several advantages for all actors involved in the IoT ecosystem. In fact, by exploiting the overall view of the network topology it is possible to establish the optimal communication path according to the application requirements. Considering for example the network of Fig. 7.1, it is typically more efficient for nodes n_2 and n_6 to communicate through the switches than through the WSN, which would be the standard solution. This is valuable for final users (they can obtain higher levels of QoS) and network operators (they can increase resource efficiency). Furthermore, by leveraging the virtualization/abstractions of physical resources it is possible for application developers to focus on the implementation of the application, ignoring the specific low level details of the physical objects and their interconnections. This would involve significant reduction in the development cost and time to market. Finally, by leveraging the possibility to *program* the behavior of nodes through network applications running on top of the NOS it becomes possible to have IoT nodes that, in principle, can execute any existing or novel IoT protocol. It is even possible to have IoT nodes that handle packets belonging to different applications according to different protocols. This very high level of flexibility can be exploited to

- integrate IoT platforms running different IoT solutions, so contributing to solve the IoT fragmentation problem, and making it possible to implement and deploy applications that exploit the services offered by a larger pool of IoT resources.

- improve the Quality of Service offered to applications and increase efficiency. This is possible because it is well known fact that there is no *one-fits-all* solution in the IoT context, and by exploiting the proposed approach any application can rely on the most appropriate set of protocols (which should be implemented as network application on top of the NOS).

The individual components of the proposed architecture have been described in 6. The main goal of this chapter is to propose an architecture which accommodates the above components and explain how they interact in order to achieve the aforementioned contributions. Also, in this chapter the overall system is assessed in an application scenario involving the use of MapReduce to process the data produced by IoT devices.

In order to maximize compatibility with relevant standards, major requirement we considered in our work has been to minimize the number of new abstractions needed in the ONOS framework and avoid any need for modifications of the OpenFlow (and other) standard(s).

The rest of this chapter is organized as follows. Then, Section 7.1 provides a general view of the overall system architecture and summarizes the motivation for the proposed system. Section 7.2 introduces the concept of the Software Sensor Node, which is used to integrate WSNs and networks of switches. The fundamental operations offered by the proposed system, namely sensor node registration and integrated packet forwarding are explained in Sections 7.3 and 7.4, respectively. A MapReduce-based use case for the system is presented in Section 7.5 and it is used to extract performance metrics, which highlight measurable benefits of the proposed approach. Finally, conclusions are drawn in Section 7.6.

7.1 Motivation and Architecture

The key motivation of this work is the realization of a unified system which enables the communication among heterogeneous devices in order to create IoT ecosystems exploiting the infrastructure already in place. Following a pure network-level approach, the proposed system does not depend on any particular IoT architecture. This is achieved by:

1. Employing SDN technologies in both the transport and the sensor networks.

2. Specifying generic, protocol-agnostic abstractions for sensor nodes and net-

work packets.

3. Introducing the concept of the *software sensor node*, an SDN-enabled sensor node, which can be easily integrated with any existing WSN deployment, by implementing the corresponding protocol procedures inside the NOS.

Fig. 6.2 depicts the extended ONOS architecture, where the dark-colored components are required to integrate software defined sensor nodes [88]; whereas the light colored components have been kept back in their original ONOS implementation.

The proposed architecture follows the logical separation of ONOS into subsystems. The SensorNode Subsystem provides the abstractions as well as an up-to-date registry of the sensor nodes connected to the system. More specifically, the *SensorNode API* is a generic, extensible, API, which specifies a generic sensor node, so that it can be efficiently accessed from the other layers of the NOS:

- The *SensorNode API* is used for both sending information, such as forwarding rules, to the sensor nodes as well as receiving information, like the battery level and their neighbors, from them. The API is extensible, so that application-specific sensors can also be easily integrated.

- The *SensorNode Manager* implements the *SensorNode API* functionality. Essentially, it is a registry for all the sensor nodes managed by the NOS by maintaining up-to-date information either for logging reasons (e.g. forwarding rules) or for status maintenance (e.g. battery level).

- The *SensorNode Provider* translates protocol-specific data to data that can be accessed through the *SensorNode API* and vice-versa. More specifically, incoming data are decoded into ONOS-specific APIs, so that they can be effectively managed by the higher layers. Outgoing data are encoded into protocol-specific APIs, by leveraging the *Protocols* layer.

- The *SensorFlowRule Subsystem* is used to create and install flow rules, by leveraging the protocol supported by the target device.
- The *SensorFlowRule API* specifies the high-level semantics for creating rules that are sensor-node specific and can look at any portion of the packets to classify them into flows. This is supported by SDN-WISE and enables a high degree of flexibility which goes well beyond what OpenFlow can obtain.
- The *SensorFlowRule Provider* is used to encode flow rules, by leveraging the corresponding protocol, so that they can be sent to the sensor nodes. Let us note here that the *FlowRule Manager* and the *FlowRuleProvider API* have been kept unchanged with respect to their original ONOS implementation. This has been achieved by exploiting the high-level semantics of ONOS combined with the generic design of the *FlowRule API*.
- The *SensorPacket Subsystem* handles incoming and outgoing packets from/to the sensor nodes, by using a non-IP, generic packet encapsulation API. In particular, the *SensorPacket API* provides access to incoming and outgoing packets from/to sensor nodes. Even though the network-level packet format depends on the corresponding underlying network protocol, the *SensorPacket Provider* encapsulates all this information into ONOS high-level abstractions.

The current API has been designed by considering requirements of generic sensor nodes and provides access to information such as host and destination address, protocol version and other low-level details. However, this API can be extended to support application-specific protocols by building on top of the existing system.

In the Protocols layer, the specific semantics of the underlying network protocol are implemented. More specifically, the proposed architecture includes the server-side SDN-WISE implementation, which is responsible for communicating with the sensor nodes and provides access to low-level protocol semantics. This is achieved through a protocol-specific API, with respect to the already existing one in ONOS for OpenFlow, which is able to encode/decode information coming from the WSN/NOS.

Observe that in the APP layer, ONOS supports the implementation of network applications. In fact, these applications are responsible for creating the rules, which are then installed in the nodes by leveraging the corresponding subsystems. When a sensor node, which is connected to ONOS, receives a packet that it cannot handle, it attaches this packet to a request and sends it to ONOS. ONOS decapsulates the various packet layers as described previously in the context of the subsystems and, in case it cannot directly be handled by them (e.g. it is not a trivial beacon packet), it is delivered to the network application associated with this node. Then, the network application decides how to handle this packet. Note here that a network application essentially implements a network protocol. Therefore, even existing protocols, such as 6LoWPAN or ZigBee can, in principle, be implemented in ONOS. For instance, if an SDN-WISE node receives a 6LoWPAN packet, it will encapsulate it in an SDN-WISE packet by inserting it in the payload. Then, the network application will decode that payload, it will understand that it is a 6LoWPAN packet and will send the corresponding reply packet to the sensor node.

Given the heterogeneity of the supported devices (sensor nodes and switches), ONOS has to associate each device type with a set of subsystems in order to handle them accordingly. This is achieved by adopting system-wide naming schemes in the form of Uniform Resource Identifiers (URI) for every component. For example, an OpenFlow switch is represented as *of:000000001* and an SDN-WISE node as *sdnwise:00000101*. In this way, every device connected to the system as well as every device-specific software module are registered using the same prefix. As a result, ONOS can deliver packets and events coming from the devices to the respective subsystems and viceversa, transparently.

Despite the complexity and the variety of operations provided by the NOS, in this chapter we focus on a fundamental subset, in order to illustrate how the proposed solution achieves the contributions outlined at the beginning of this section are accomplished. More specifically, in the following, we first introduce the concept of the *Software Sensor Node* and then, we explain the procedure implemented for registering and representing a sensor node in the system. Then, we illustrate how different sensor network applications can register their packet types to the system, so that they can efficiently make use of the corresponding flow rules capabilities. Finally, we present the integrated forwarding network application, which essentially bridges the gap between networks of switches and sensors by leveraging both existing and newly introduced NOS components.

7.2 The Software Sensor Node

The fundamental component introduced at the device level for network-layer integration is the *Software Sensor Node (SSN)*. SSNs are essentially SDN-WISE-enabled nodes, which do not necessarily correspond to a physical device, with respect to the concept of OVSwitch for OpenFlow networks. In fact, such nodes

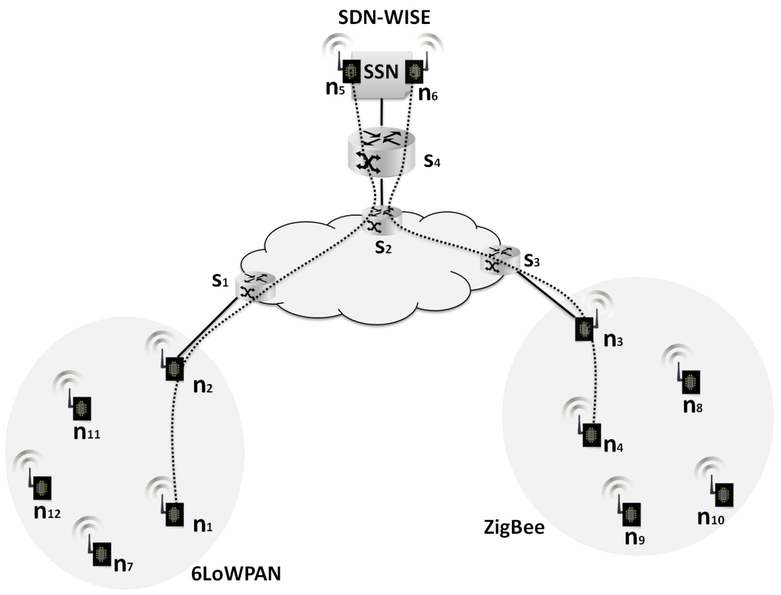


Figure 7.2: Heterogeneous Network Integration with SSN

may as well be deployed as services in the network, e.g. using Network Function Virtualization (NFV), by leveraging the SDN-WISE node implementation, which has already been used for WSNs emulation.

Due to their programmability stemming from their SDN support, SSNs can be easily integrated with any IoT protocol, by just implementing the corresponding APIs as a NOS application. This way, a SSN can be virtually attached to any WSN by just sending the corresponding signaling messages to the nodes it has to connect to. For example, in Fig. 7.2, suppose that nodes n_1 and n_2 belong to a 6LoWPAN network, whereas nodes n_3 and n_4 belong to a ZigBee network. Then, the SSN, can send the appropriate neighborhood discovery packets to n_1 and n_4, through the respective sinks n_2 and n_3, and become neighbor with both them. Furthermore, it can also exchange neighborhood information with both these nodes, thereby creating a symmetric link $n_1 - SSN - n_4$, which in fact hinders the path of the intermediate sinks and switches, giving the impression to the sensor nodes that they have two hops distance.

This way, every SSN actually represents the set of sensor nodes of a particular WSN that are connected to it. Therefore, the overall network topology can be abstracted as network of SDN-enabled switches and sensors. Moreover, *dynamic* ad-hoc networks can be created by connecting individual nodes over an heterogeneous infrastructure, consisting of switches and SDN-WISE sensor nodes, either physical or virtual (software) ones. As a result, transferable network configurations can be enforced *on the fly* to already existing WSNs, with respect to the IoT vision presented in [84], by controlling the whole infrastructure in a holistic way.

The proposed NOS performs holistic management of the networks of switches and sensors by introducing generic abstractions for sensor nodes and packets and by performing integrated packet forwarding transparently across these networks. Note that the implementation of 6LoWPAN and other WSN protocols in the NOS has not been prototyped in the context of this paper, since it mostly involves development work, whereas it does not add any complexity in the overall architecture; eventually, the challenge is for SDN-WISE nodes to communicate over OpenFlow switches.

7.3 Sensor Node Registration and Representation

A key requirement of the proposed system is the unified representation of the whole network, consisting of both sensor nodes and switches. This is achieved by introducing new and extending existing core ONOS abstractions, which provide fine grained access to the network components, namely the devices and the links. More specifically, two access layers are considered for devices and links, the base and the enhanced layer. The base layer is used to create an internal graph representation of the network topology, where every node is represented by a `Device` and every edge by a `Link`. The enhanced layer for the sensor nodes, `SensorNode`, extends the

`Device` and it is used to access implementation-specific information, such as the energy level of a sensor, its geographic coordinates and its neighborhood. Whereas, the enhanced layer for the sensor links, `SensorLink`, extends the `Link` to provide information on the Received Signal Strength Indicator (RSSI) of a wireless link, which is typically used as a metric for the weight of an edge representing a wireless link.

Information provided at the enhanced layer can be further extended through annotations, implemented as key-value pairs. Such annotations are typically useful for passing application-specific parameters to ONOS, which are required to make network-layer decisions. For example, consider a WSN that contains two types of sensor nodes, temperature and pressure and assigns a different aggregator to each type. In order to forward the packets to the corresponding aggregator, ONOS needs to be aware of the sensor type each node carries. This information cannot be contained in any network packet header, though. Therefore, through the annotations, such information can be passed to the system from an external service, rather than the network, so that it can be later used for taking networking decisions, such as packet forwarding.

The registration of SDN-WISE nodes is quite straightforward and is achieved through the SDN-WISE protocol mechanisms by leveraging the `SensorLink` abstraction provided by the proposed system. Links between sensors are periodically reported to ONOS through the *REPORT* packets issued by every sensor and containing the list of its neighbors and the associated Received Signal Strength Indicator (RSSI). Links between sensors (sinks) and switches are reported through the *DP_ CONNECTION* packets, which contain the MAC address of the sink nodes.

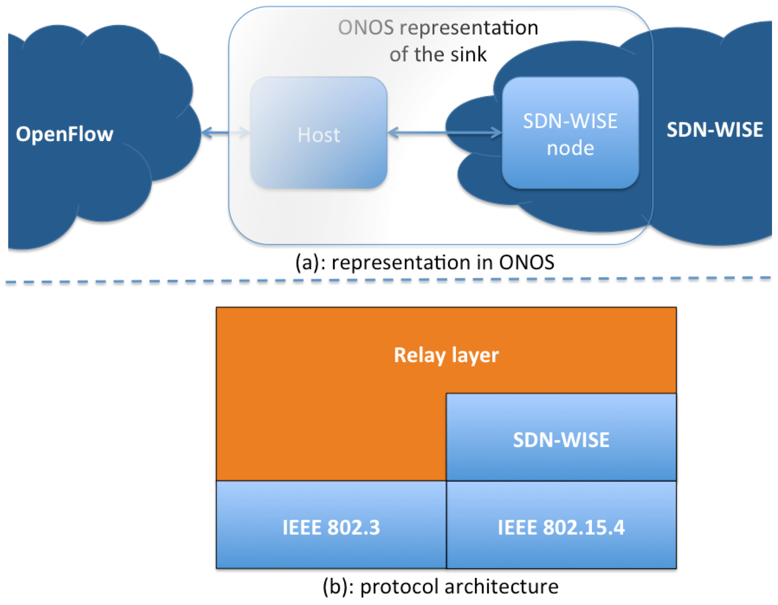

ONOS representation of the sink
OpenFlow
Host
SDN-WISE node
SDN-WISE
(a): representation in ONOS
Relay layer
SDN-WISE
IEEE 802.3
IEEE 802.15.4
(b): protocol architecture

Figure 7.3: Representation of (a): the sink node in ONOS, and (b): its protocol stack.

In fact, as shown in Fig. 7.3(a) our extension of ONOS represents sinks both as sensors (for the wireless link) and as edge hosts (for the wired link), because they typically carry two different types of network interfaces. This design approach is the consequence of the two major requirements (i) to avoid any change in the standard OpenFlow and (ii) minimize the number of novel abstractions to be introduced in ONOS. To this purpose sink nodes will run the protocol stack represented in Fig. 7.3(b).

At the physical and link layers two sets of protocols will be implemented that are specific of the infrastructured and WSN segments, such as IEEE 802.3 and IEEE 802.15.4, respectively. SDN-WISE is executed on top of the IEEE 802.15.4 interface; whereas a Relay Layer is also executed which is responsible of the operations needed to bridge the two different network segments; one of its major roles is packet adaptation, for example.

Finally, links between SDN-WISE nodes and others implementing any other

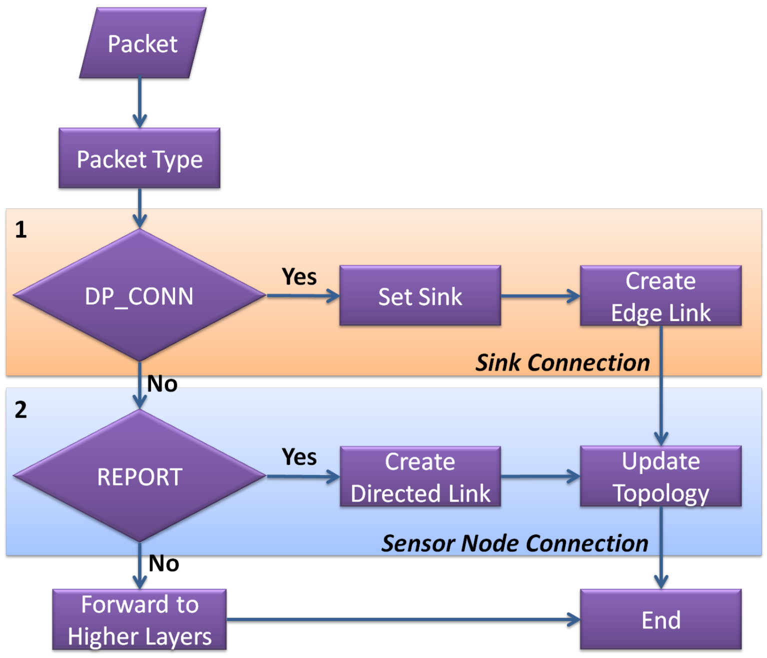


Figure 7.4: Sensor Node Registration Flowchart.

stack (e.g., 6LoWPAN), are established by leveraging the communication protocol of the standard nodes. More specifically, SDN-WISE node can be instructed by the NOS to send the corresponding protocol-specific packets and then, use the corresponding response to identify the link, in the same respect as with links between two SDN-WISE nodes. For example, SDN-WISE nodes can generate and interpret typical 6LoWPAN signaling messages such as Router Solicitation, Router Advertisement, Node Registration and Node Confirmation. In this way it is possible for SDN-WISE nodes to build a network topology overlayed at the 6LoWPAN nodes.

This is a fundamental feature of the proposed solution as there is already a large number of WSN devices already deployed worldwide (e.g., according to a ABI market research 850 million IEEE 802.15.4 devices have been shipped in 2016).

On the ONOS side, there are two subsystems involved in the sensor node registration process: the *SensorPacket Subsystem* and the *SensorNode Subsystem*.

The flowchart given in Fig. 7.4 depicts the role of the *SensorPacket Subsystem*, for example. When a *REPORT* packet arrives, a direct *SensorLink* is created between that node and all its neighbors. In the special case of the *DP_ CONNECTION*, a standard edge *Link* is created, so that ONOS treats the sink as the edge of the network of switches. Links between SDN-WISE and standard sensor nodes are created in the context of the ONOS application implementing the corresponding protocol, based on the *SensorLink* abstraction. Note that these are also edge *Links*, as the standard nodes cannot accept SDN rules and, thus, they are treated as regular hosts. After the links have been established, the network topology is updated accordingly.

ONOS keeps an updated view of the WSN using the *SensorNode Subsystem*. More specifically, the SensorNode Provider checks for sensor nodes connectivity, based on the reports received by them, whereas it creates a new *Device* upon a sensor node arrival to the system. At the high level, the SensorNodeManager introduces functionality to store and access fundamental information on the nodes, such as their address, their battery level, the sink they are connected to or, in case the node is a sink itself, the switch that it is attached to. Note that this information is either directly coming or implicitly derived from the *REPORT* packets. Sensor nodes are abstracted by the *SensorNode* data structure, which extends the *Device*, thus directly enabling its integration with the *Topology*, while providing the abstractions described earlier in this section. All information regarding sensor nodes are currently stored in hash tables. However, in case the networks grow larger, more sophisticated and highly efficient in-memory databases can be leveraged (e.g. Hekaton, VoltDB, etc.), in order to address any scalability issue, without any changes to the existing API.

7.4 Network-Wide Packet Forwarding

In our solution, the packets sent by sensor nodes to ONOS, in order to obtain a corresponding rule, are delivered to the *Integrated Forwarding* application. The latter extracts the source and the destination and calculates the shortest path considering the whole topology. Wireless link cost is calculated based on the link RSSI and is dynamically passed to the path extraction algorithm. Accordingly, in the example of Fig. 7.1, a packet from node n_1 to node n_5 will follow the path including switches, since it contains fewer wireless links, which are most costly. Whereas, a standard controller would suggest the path of sensor nodes only, since it would not have a full view of the topology.

Unfortunately, packets coming from sensor nodes have different format than the standard IP packets, sent by hosts and are currently supported by ONOS. Accordingly, we have introduced a new API, the *SensorPacket API*, on top of the existing *PacketManager*, as shown in Fig. 6.2, in order to enable ONOS to deal with the new packet formats. *SensorPacket API* includes the *PacketType*, which represents sensor packet types similarly to the *type of service* of IP packets. The main novelty of this API is that it allows different network applications to *register* their own packet types, without interfering with each other, thereby fostering reuse of existing packet headers with application decoupling.

Once ONOS can deal with packets generated by sensors, then, in order to achieve integrated forwarding we need to define how ONOS will instruct the network nodes in such a way that the traffic flow can traverse sequences of switches and sensors, transparently. A major problem in this context is the passage between one segment of switches and one segment of sensors (or viceversa), since they use different instruction sets. In our solution, when a packet arrives to the system,

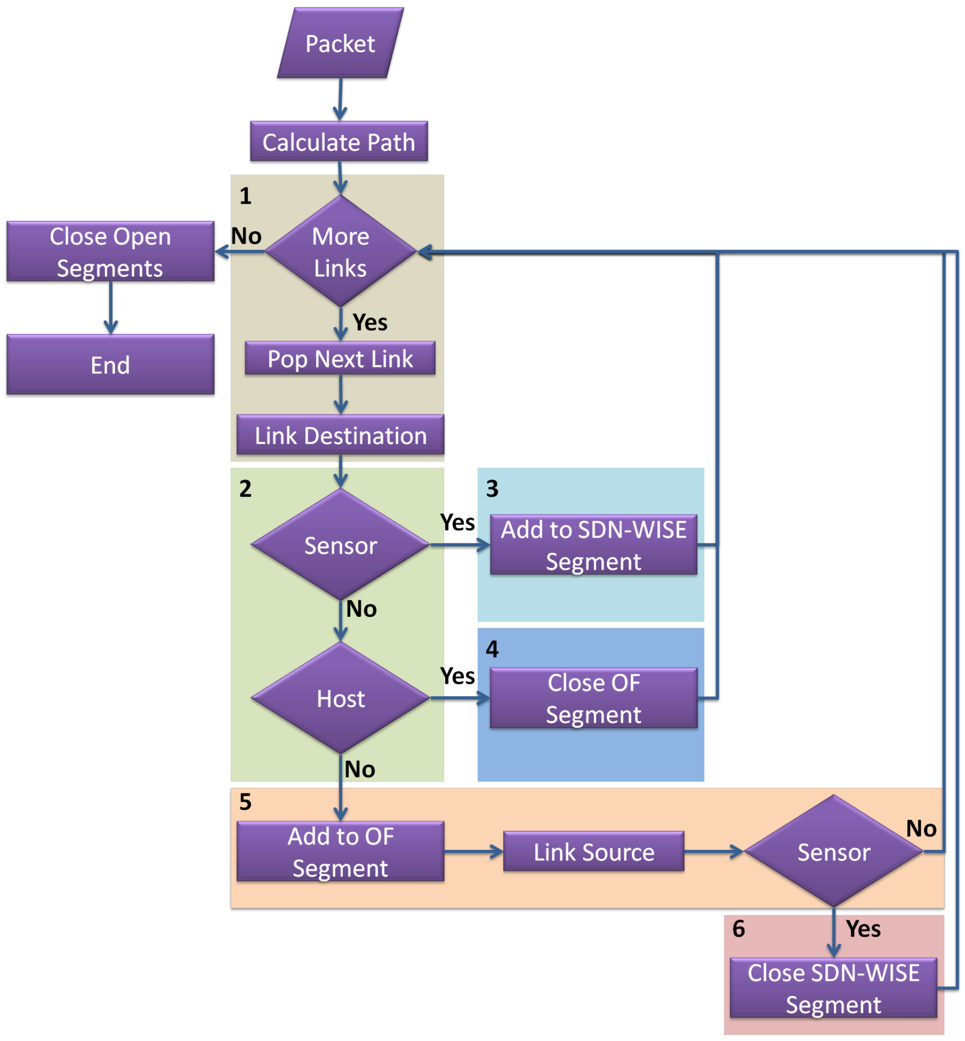


Figure 7.5: Integrated Forwarding flowchart.

the *SensorPacket Provider* checks its type and, if it is not used for signaling, it encapsulates it into an ONOS-specifc packet format that can be accessed by the *SensorPacket API*. Accordingly, incoming packets are transformed to Ethernet frames, using ONOS core APIs. Then, they are delivered to the *PacketProcessor*s, which are typically implemented in the application layer.

Fig. 7.5 describes the steps followed by the *Integrated Forwarding* application when it receives a packet (typically from the *Packet Provider*). After the whole path from source to destination has been calculated, it is broken into individual segments, which separate the sequence of switches and sensor nodes, as described in the flowchart of Fig. 7.5. Each segment is represented as a set of links connecting *only* devices of the same type. Accordingly, every time a link connecting devices of different type is met, the current segment is closed and a new one is opened.

For example, considering that node n_1 of Fig. 7.1 sends a packet with destination node n_5 and that ONOS calculates as shortest path the one passing through the OpenFlow switches, *Integrated Forwarding* will split the path into two different segments. More specifically, when ONOS receives the packet, it triggers the process described in the flowchart of Fig. 7.5. After the path is calculated, the first link is considered and switch s_1 is added to the segment of switches, by entering the flowchart parts 1, 2, 5 and 6. Then, the switches s_2 and s_3 are also added to the same segment by entering parts 1, 2 and 5. When the link $s_1 \rightarrow n_7$ is visited, the segment of switches is closed, by executing the operations shown in parts 1, 2 and 4. Observe that ONOS understands n_7 as host h_2 in this case, since it is connected to switch s_3. Finally, for each one of the two remaining sensor links from n_7 to n_5, parts 1, 2 and 3 are considered. After that step, there are no more links in the path and hence, the active sensor node segment is closed as well and the process ends.

For each generated segment, the appropriate flow rule subsystem is used in order to specify the forwarding rules. More specifically, if the request is made by a sensor, the *SensorFlowRule Subsystem* is used, whereas if it is made by a switch, the *OpenFlowRule API* is leveraged.

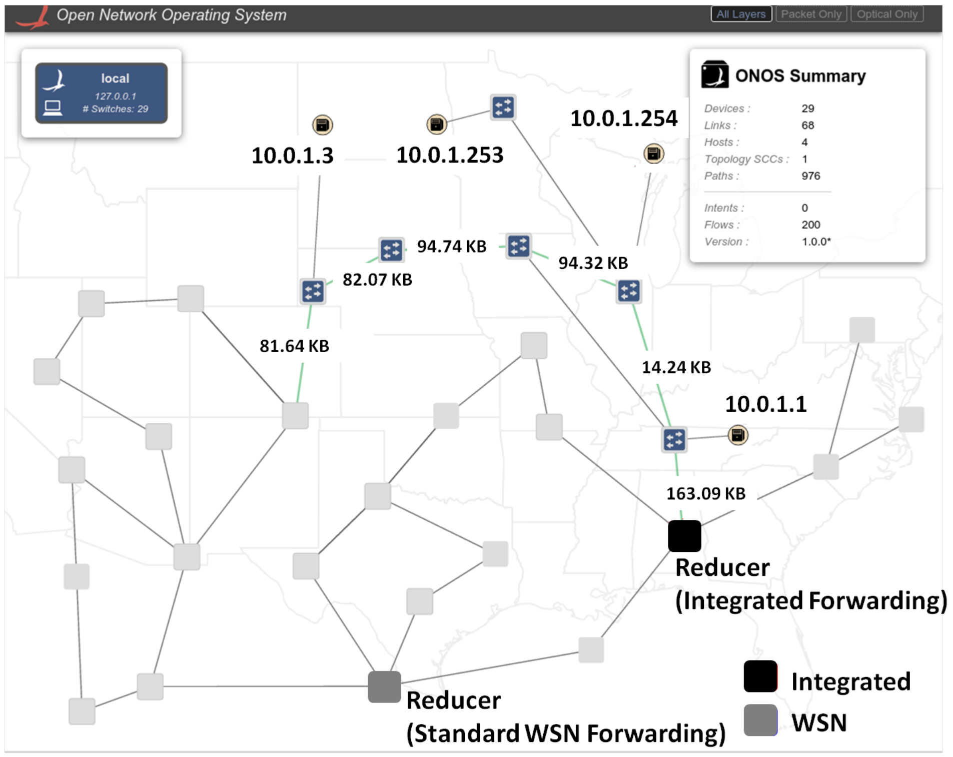
7.5 Case Study: MapReduce In-Network Processing

As we have already discussed, the proposed solution has several advantages ranging from very high flexibility to the possibility for developers to implement IoT services and applications ignoring the specific low level details of the physical objects and their interconnections. In this section, however, we will focus on the performance improvement that can be achieved by exploiting the overall view of the network topology, including both infrastructured and infrastructureless segments.

To this purpose we will compare the performance obtained by the proposed approach to the performance obtained when the infrastructured segments of the network implement a shortest path routing whereas the infrastructureless segments implement RPL[1], which is the most widely deployed routing protocol for 6LoWPAN networks [46].

More specifically, the effectiveness of the proposed system is assessed by considering the case where MapReduce operations are executed inside the WSN network and the results are sent to a service deployed in a host outside the WSN, as suggested in [89]. For example, considering the system of Fig. 7.1, suppose that the sensor nodes are executing MapReduce operations and then, the results are sent to service SVC-1, deployed in the cloud, by sending the data over the OpenFlow network. Let us explicitly observe that in-network processing for big data applications has already been successfully applied in data centers [90, 91]. In the use case addressed here, we shift this paradigm to in-network processing for

[1]Although a large number of RPL improvements have been proposed in the recent past, we have chosen the standard RPL implementation because we are focusing on the advantages obtained by exploiting the complete and integral view of the overall network rather than the possibility to select shorter paths inside the infrastructureless segment.



Devices : 29
Links : 68
Hosts : 4
Topology SCCs : 1
Paths : 976

Intents : 0
Flows : 200
Version : 1.0.0*

Figure 7.6: Network topology with different reducers for each case.

sensor networks.

We consider a WSN of 23 nodes connected with a transport network of 6 switches, as depicted in Fig. 7.6. Each sensor node implements the SDN-WISE protocol, whereas each switch implements OpenFlow. Switches and sensor nodes are deployed in Mininet, which has been extended to support the SDN-WISE software sensor nodes. Sensor nodes are equipped with sensors of four different types: temperature, humidity, pressure and noise. Values generated by each of these types are associated with the corresponding key in the context of the `map` function. According to MapReduce, there has to be exactly one node, namely the `reducer`, which collects all values associated with a single key. The selection of the reducer for each key is made as explained in [89] and is outside the scope of this work.

In order to select the appropriate reducer for each key, it is essential to know the sensor types that every node carries, which is not included in the signaling messages. This is achieved in the application layer by leveraging the extensibility of the *SensorNode API*, based on the inherent ONOS *Annotations API*. This enables the update of the nodes description by inserting the corresponding sensor types as key-value pairs of the form $\langle sensor, sensor\ type \rangle$.

In our experiments, every node carries a random number of sensor types. Furthermore, a *map* function is loaded in each node by ONOS, which periodically generates key-value pairs for each sensor type and pushes them to the network layer. At the first execution, the node does not know how to treat the packet generated by the *map* function and therefore, it makes a request to the NOS. The latter selects the reducer and, through *Integrated Forwarding* application, installs the appropriate rules to every device, as described in the previous section.

We have implemented a prototype of the proposed solution for TI CC2530 devices as well as the software modules that extend Mininet to consider sensor nodes also.

Our prototype has been released in open source and can be downloaded at the following URLs:

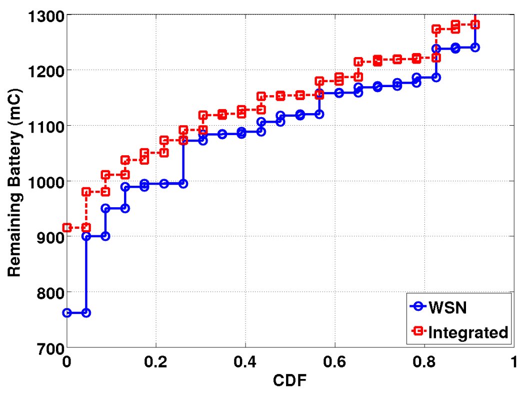
- `https://github.com/sdnwiselab/onos` (source code of ONOS)
- `https://github.com/sdnwiselab/sdn-wise-cc2530` (source code for the TI CC2530 devices)
- `https://github.com/sdnwiselab/onos-sdn-wise-app-samples/tree/master/mapreduce` (source code of the MapReduce application and the Mininet scripts)

Since we have a small number of sensors, the results shown in the chapter have been obtained by using the Mininet simulator. In our experiments energy related metrics have been calculated by elaborating the logs of the simulation considering the information about battery and energy consumption provided in the datasheets of the TI CC2530.

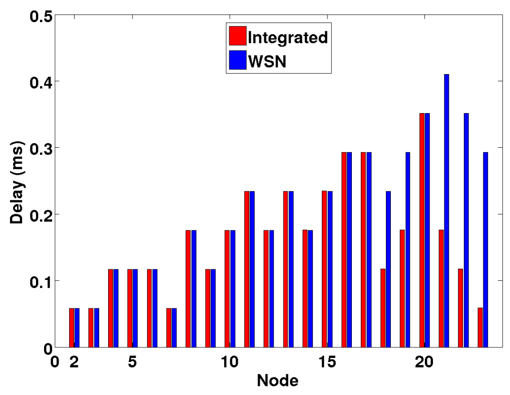
The use of *Integrated Forwarding* firstly affects the selection of the reducers, which is made considering the cost of communication among the nodes, derived from the forwarding path. In particular, as shown in Fig. 7.6, in the standard WSN case, the node in the center of the network is selected to be a reducer. On the other hand, when *Integrated Forwarding* is enabled, the optimal reducer is one of the sinks (node 1 in this case). In order to have a common point of reference, the evaluation will be made considering that the reducer will be node 1 in both the standard WSN and the integrated forwarding approach, since the goal is to study the communication aspects of the proposed architecture and not the MapReduce implementation as a whole.

The use of sensor nodes resources after executing the experiment for 10 minutes is shown in Fig. 7.7. More specifically, Fig. 7.7a shows the CDF of the remaining energy level of the sensor nodes after the experiment is over.

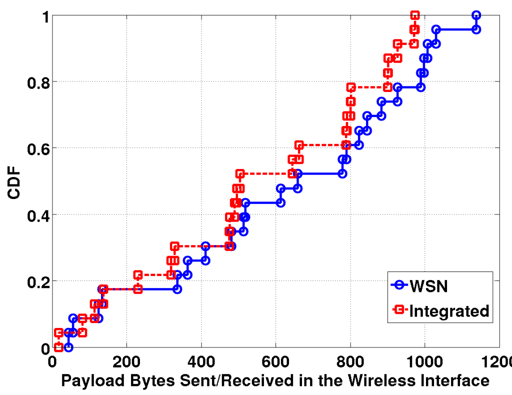
As shown in the figure, in the integrated forwarding case, there is always a larger fraction of nodes with more residual battery than in the typical one. Moreover, since the difference increases as the energy levels become lower, the network lifetime is expected to be significantly higher when using the proposed approach. Furthermore, as shown in Fig. 7.7b, in case the network of switches is used, i.e. when nodes 18, 19, 21, 22 and 23 are sending data to node 1 (reducer), the delivery times are significantly smaller than in the typical WSN-specific approach.



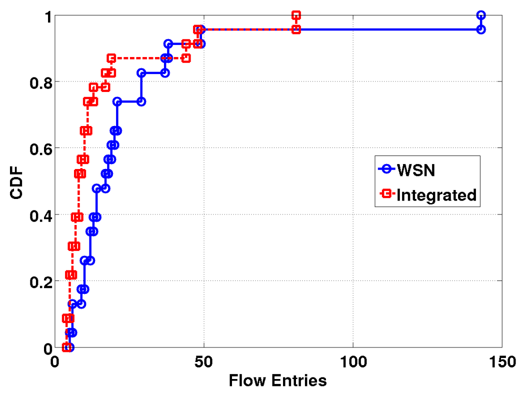
(a) Remaining Battery CDF



(b) Delay



(c) Nodes Traffic CDF



(d) Rules CDF

Figure 7.7: Nodes resource usage.

The reason is that the core network has much larger capacity, and therefore packets are forwarded much faster than in the context of the WSN.

The effectiveness of the use of the core network also becomes clear by considering Fig. 7.7c, which shows the CDF of the overall number of packets sent and received by the sensor nodes in the plain WSN and the integrated forwarding approach. Overall, the integrated forwarding approach results in less communication between the nodes, which is essential for WSNs and is the key factor for the reduction on the energy consumption shown in Fig. 7.7a.

Finally, the proposed integrated forwarding approach results in less flow rules in the nodes, as depicted in Fig. 7.7d. This means less signaling messages and lower memory occupancy for the nodes, which is very important given the strict limitations on both communication and hardware resources. The reduction of flow rules in the integrated forwarding approach is the result of the use of the core network. Indeed, when using the WSN alone to forward packets, each intermediate node requires 1 rule per destination. Therefore, for each path of N hops, which traverses the core network, a reduction of N rules at most (e.g. in sink-to-sink communication) is achieved for the WSN.

7.6 Conclusions

This chapter presents a network operating system which is used to integrate networks of switches with networks of sensors. Starting from ONOS, an existing solution for OpenFlow networks, we have introduced new components, which efficiently abstract both the network elements and their operations. Moreover, we have explained how the proposed extensions can work together with existing modules, eventually enabling ONOS to perform holistic network management and integrated packet forwarding in software defined transport and wireless sensor networks.

The proposed system has been prototyped, whereas its evaluation is performed in the context of a novel in-network packet processing approach, which enables the execution of MapReduce operations in WSNs.

Chapter 8

Reducing energy footprint with GEO Routing

In most cases sensor nodes have low resources in terms of energy, processing capabilities, and memory. Therefore, it is fundamental to reduce as much as possible the signaling exchange and the memory occupancy.

This can be achieved by enabling nodes to execute forwarding, that is, the most common operation performed in multihop communication networks (such as WSNs) autonomously. Indeed, a large number of distributed forwarding/routing protocols for WSNs are available in the literature which however are outperformed even by the earliest SDN solutions proposed for WSNs [92]. As discussed in [47] the use of a stateful solution such as SDN-WISE, reduces the need for message exchange between nodes and Controller when changes in the node behavior are necessary due to variations in network conditions; however, according to [93] in each node there is still the need for one entry in the flow table (and the corresponding signaling message from the Controller) for each flow traversing the node.

In this chapter we aim at defining a solution which allows sensor nodes of a software defined WSN to take forwarding/routing decisions *without explicit control from the Controller*. Thus, the proposed solution does not need signaling exchange or a flow entry for each flow traversing each node. To this purpose we leverage on geographic forwarding which has been largely applied in WSNs for both unicast and multicast communications [94, 95].

A node applying geographic forwarding relays incoming packets to its neighbor which is nearest to the destination. To do so, therefore, it only needs to know the position of the destination, carried in the packet, and the positions of its immediate neighbors. This information can be obtained using a simple protocol that will be presented later on.

In this chapter we show that by exploiting geographic forwarding/routing sensor nodes achieve several advantages. First of all, overhead due to signaling purposes and the number of flow entries needed in the flow tables decrease significantly. In fact, in principle it is sufficient for the Controller to generate and transmit a unique entry for each unicast data flow and at most D flow table entries for each multicast session in intermediate nodes, where D is the number of multicast destinations.

Concerning multicast, the processing load for the evaluation of the optimal multicast tree at the Controller can be reduced dramatically. In fact, it is well known that the problem of finding the optimal multicast tree can be formulated as a *Steiner tree problem* which is known to be NP complete in the number of nodes and destinations. By leveraging geographic forwarding, instead, the multicast tree can be identified by solving the *Euclidean Steiner tree problem* that has a complexity dependent on the number of multicast destinations which is, most of the time, significantly smaller than the number of nodes in the network.

The rest of this chapter is organized as follows. In Section 8.1 we propose a solution to apply geographic forwarding to software defined wireless sensor networks. In Section 8.2 we provide a description of our prototype implementation. In Section 8.3 we assess the advantages of the proposed approach. Finally, in Section 8.4 concluding remarks are drawn.

8.1 Geographic forwarding in SDWSNs

In this section we will describe the operations executed by the Controller and the nodes (in subsections 8.1.2 and 8.1.3, respectively). Such operations require nodes to be aware of their position as well as the positions of their neighbors, whereas the Controller must have information about the positions of all nodes. Accordingly, in Section 8.1.1 we preliminarily describe how in the proposed solution such awareness of the position is achieved.

8.1.1 Localization

In line with most solutions proposed to extend the SDN approach to WSNs we assume that nodes execute a protocol which allows them to collect information about the neighboring nodes and the RSSI values of the corresponding wireless links. Such protocol is based on the periodic generation of *Beacon* messages by the sink(s); these messages are hop by hop relayed by sensor nodes to their neighbors. As a consequence of the reception of these beacons, nodes send the list of neighbors and the RSSI values to the Controller in appropriate *Report* messages. In existing solutions, the Controller uses the information contained in the *Report* messages to build a representation of the network topology which is used for routing purposes. In our approach, the Controller uses such information to estimate the positions of the nodes in the network, as well. In fact, RSSI values can be used to evaluate the distance between the corresponding pairs of nodes and consequently to localize nodes. Several centralized localization techniques exist (see [96, 97] for overviews)

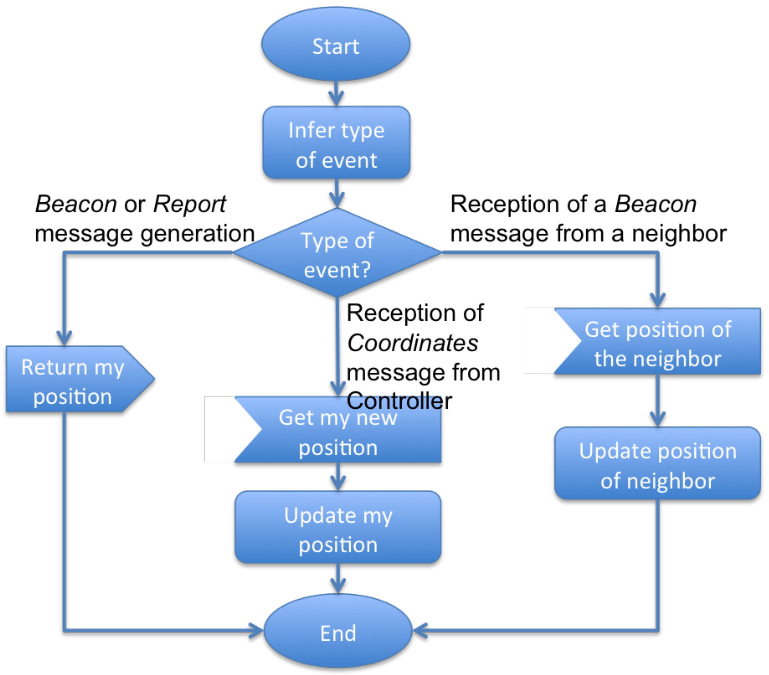

Start
Infer type of event
Type of event?
Beacon or *Report* message generation
Reception of a *Beacon* message from a neighbor
Reception of *Coordinates* message from Controller
Return my position
Get my new position
Update my position
Get position of the neighbor
Update position of neighbor
End

Figure 8.1: Flow diagram of the operations executed at the nodes for localization purposes.

which provide reliable results [98]. Indeed, a large number of studies can be found in the literature that compare different localization techniques for WSNs, e.g., [99] and [100]. The selection of the specific solution utilized to evaluate the position of the nodes is however out of the scope of this work. In fact, we explicitly observe that since the localization algorithm is executed by the Controller, it can be changed very easily. This will be discussed in Section 8.2.1.

Furthermore, note that in the case some sensors are aware of their position[1], they will be considered *anchors* in the localization algorithm performed by the Controller. This might significantly increase the localization accuracy of other nodes [101]. In any case, we observe that the solution we propose for geographic forwarding in software defined WSNs is robust to significant localization errors as we discuss in Section 8.3.1.

[1]The positions of some nodes might be set by the installer at the deployment time or some nodes might be equipped with localization technologies, e.g., GPS.

In Figure 8.1 a high level flow diagram of the procedure performed by sensor nodes for localization purposes is shown. Such procedure is triggered in four different cases:

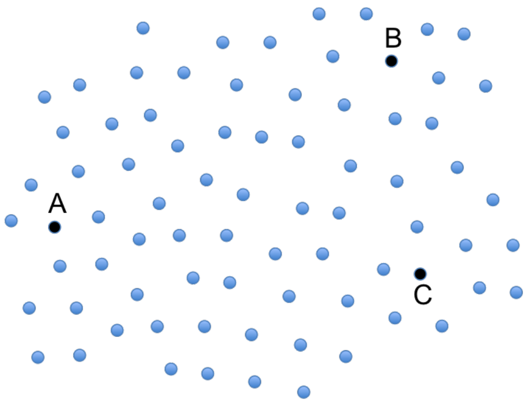
1. when a *Beacon* message, originally generated by the sink, must be relayed by the sensor node to its neighbors. In this case the procedure will just return the current position of the node which will be included in the *Beacon* message so as to allow the neighbor nodes to learn its position.

2. when a *Report* message must be generated. Also in this case the procedure will return the current position of the node which will be included in the *Beacon* message so as to allow the neighbor nodes to learn its position.

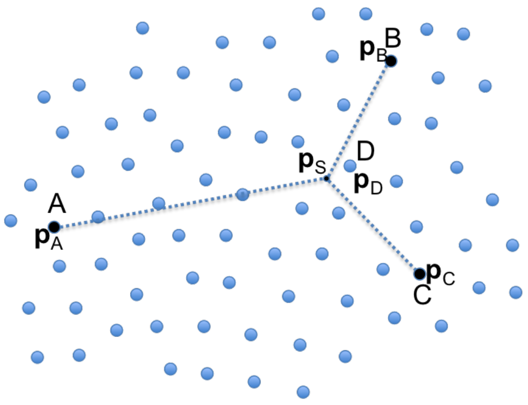
3. when a *Report* message must be generated. Also in this case the procedure will return the current position of the node which will be included in the *Report* message, together with the list of neighbors and the RSSI values. This message will arrive to the Controller which can check whether the node has updated information.

4. when a *Coordinates* message is received from the Controller. The latter is a new[2] message generated by the Controller to give each node information about its position and, optionally, the positions of its neighbors. When a node receives a *Coordinates* message, it updates its position information and, optionally, the positions of its neighbors. Note that several strategies can be devised to identify when the Controller has to generate a new *Coordinates* message for a given node. In our implementation the centralized localization algorithm is performed by the Controller and *Coordinates* messages for a

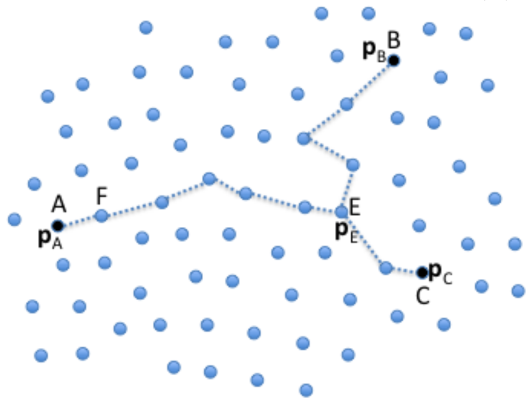
[2]There is no analogous message in other SDN solutions for WSNs.



(a) Remaining Battery CDF



(b) Delay



(c) Nodes Traffic CDF

Figure 8.2: Exemplary case.

set of given nodes are generated when a change occurs in the list of their neighbors.

5. when a *Beacon* message is received from a neighbor node. In this case, the position of the neighbor will be updated.

Operations performed by the Controller are an obvious consequence of what we have described above.

8.1.2 Controller operations

In order to present an overview of the operations executed by the Controller to support geographic forwarding in a software defined wireless sensor network we consider an exemplary scenario. We consider a *dense* network with a large number of nodes as shown in Figure 8.2a and suppose that node A generates a packet which must be sent to nodes B and C. Note that we consider the case of a multicast (rather than unicast) session because it is more general than the unicast case. The latter can be easily derived as a special case of a multicast session with only two members of the group.

When the application in A generates the first packet which must be delivered to B and C, there are no entries in the flow table that can be applied and therefore the packet is sent to the Controller. We assume that the Controller knows that the destinations are B and C[3]. As explained in Section 8.1.1, the Controller knows the position of all nodes including A, B, and C, which we denote as $\mathbf{p}_A$, $\mathbf{p}_B$, and $\mathbf{p}_C$, respectively. Therefore, it can calculate the Euclidean Steiner tree which, in our exemplary case, includes a Steiner (i.e., branching) point in position $\mathbf{p}_S$, as shown in Figure 8.2b.

In the general case, there are no nodes in $\mathbf{p}_S$, therefore, the Controller will select the node which is closest to the Steiner point. In our exemplary case, the Controller selects node D in position $\mathbf{p}_D$. Therefore, node A should send packets towards p_D, whereas D will relay the packets towards $\mathbf{p}_B$ and $\mathbf{p}_C$.

However, the Controller will preliminarily simulate the tree which will be built by nodes executing geographic forwarding. In Figure 8.2c we show the tree which will be built in our exemplary case. This is done for two reasons:

- Check that no deadlock situations will be incurred. In fact, it might happen

[3]How the Controller can learn about the identity of the destinations of the packet is outside the scope of this work. Nevertheless, in our prototype we implemented two mechanisms. In the first, nodes must join the multicast group (similarly to IGMP). In the second, it is the application that informs the Controller that packets with certain characteristics must be delivered to certain destinations.

that, while executing geographic forwarding, the packet arrives to a node which does not have any neighbor closer to the destination than itself. If this is the case, then the packet will be sent using traditional forwarding techniques, i.e., geographic forwarding will not be applied.

- Check whether there are cycles in the forwarding graph and, in case, remove them. To this purpose, note that the cycles will certainly include the branching points (D in our exemplary case) and therefore, detecting them is very simple. For example, in the scenario depicted in Figure 8.2c, the packet will traverse the link between D and E twice. This is useless, therefore, the Controller will consider node E (located in $\mathbf{p}_E$) as branching point instead of D.

Accordingly, the Controller will insert rules in the flow tables of nodes A, E, B, and C. Based on such rules, node A will modify the packets of the multicast flow by inserting a flag informing that the packet must be relayed according to geographic forwarding and that the position of the branching node, i.e. the intermediate destination, is $\mathbf{p}_E$. Node E, instead, will create two copies of each packet of the multicast flow and modify them so that they will be forwarded towards $\mathbf{p}_B$ and $\mathbf{p}_C$, respectively. Finally, nodes B and C will deliver the packet to the application layer.

8.1.3 Nodes operations

Operations performed by nodes to geographically forward the packets are straightforward. In fact, packets that must be relayed according to geographic forwarding contain the position of the intended destination A node, upon receiving one of such packets, calculates the Euclidean distances between the destination and the neighboring nodes, selects the neighbor which is closer to the destination, and forwards the packet to such node. For example, in Figure 8.2c node A sends the packet to node F because it is the neighbor closer to the position $\mathbf{p}_E$.

Operations executed by destinations (or branching points such as E) are the consequence of those executed by the Controller as explained at the end of Section 8.1.2.

8.2 Prototype

This section describes the prototype of the proposed solution we realized using ONOS and SDN-WISE. First, in Section 8.2.1, the necessary ONOS components are described. Then, the operations supporting geographic forwarding in both the unicast and the multicast cases are explained in Section 8.2.2.

8.2.1 ONOS Extended Architecture

The ONOS (Open Network Operating System) [15] is a distributed network operating system, that we used to manage SDN-based networks and WSN networks. Our current implementation is based on SDN-WISE but apart from the original SDN-WISE packet types, we have implemented four new SDN-WISE modules and corresponding messages in order to address the geographic forwarding as well as the multicast case, since we did not want to alter the standard protocol operations. More specifically, the *GroupJoin* and *GroupLeave* message types are used when a node requests to join or leave a particular multicast group. The *MulticastData* message contains the multicast headers, which are going to be presented later in Section 8.2.2, and the payload. The *Coordinates* message type carries the coordinates of a specific node and of its neighbors.

In the *ONOS Application* layer, the *GeographicForwarding* application has been implemented and contains two fundamental modules: the *LocalizationAlgo* and the *MulticastTreeAlgo*. *LocalizationAlgo* is based on the Adaptive n-Triangle method, which has been shown to perform very well in centralized environments [98]. This algorithm is dynamically passed as an argument to each sensor node instance provided by the *SensorNode API* and not stored inside the *SensorNode Manager*, thereby ensuring that different localization techniques can be used in parallel, without interfering with each other. The *MulticastTreeAlgo* implements an application-specific API for creating a multicast tree and retrieving the next multicast hop in the tree, as well as the sequence of intermediate hops between them. This API has been implemented to calculate both the Steiner tree, using Dreyfus-Wagner dynamic programming solution [102], and the Euclidean Steiner Tree, using GeoSteiner [103]. Moreover, it provides a method to check whether there is a cycle in the path to be calculated by the nodes, so that an alternative path can be enforced to the network by the *GeographicForwarding* application.

8.2.2 Geographic Forwarding

The geographic multicast implementation assumes that at the ONOS side destinations for packets belonging to a certain flow are known. To this purpose in the multicast case GROUP_JOIN and GROUP_LEAVE messages, the format of which are shown in Figure 8.3, are used. Those are sent by each node that requires membership to a specific group characterized through a Group ID or wants to leave a group. When ONOS receives this packet, the *Group Manager* module is triggered and registers/deletes the node to the specified group.

At the node side, the geographic forwarding implementation allows each node to find the next intermediate hop towards the destination (either a destination or a Steiner point) in a completely autonomous way.

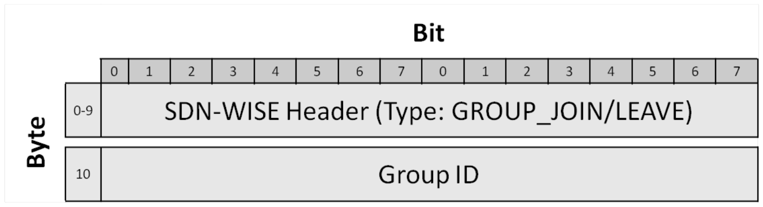


Figure 8.3: Multicast Group Join/Leave message format.

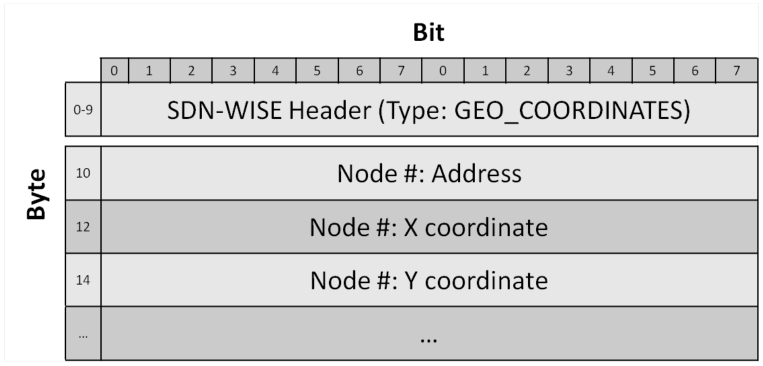


Figure 8.4: Geographic Coordinates message format.

SDN-WISE nodes implementation does not provide geographic forwarding support. In order to overcome this issue, we exploit the SDN-WISE In-Network Packet Processing (INPP) capabilities. More specifically, ONOS *GeographicForwarding* application sends to all nodes a *geographic function* that finds the neighbor which is closest to the position of the intended destination (or the next multicast hop/Steiner point in the multicast case). After this function has been installed into the nodes, it can be called through the flow table, where the action is set to *SEND_ TO_ INPP*. Furthermore, the ONOS Application sends each node its own coordinates, as calculated by the *LocalizationAlgo*, as well as the node's neighbors. These are encoded in the *Coordinates* packet, whose format is shown in Figure 8.4. The *Coordinates* packet is delivered to the geographic function by installing a single rule for geographic-type packets, which will be explained later in this section.

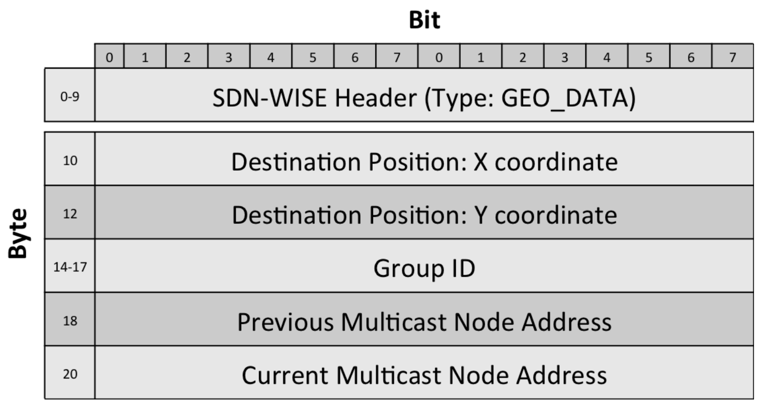

Bit
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
Byte
0-9
SDN-WISE Header (Type: GEO_DATA)
10
Destination Position: X coordinate
12
Destination Position: Y coordinate
14-17
Group ID
18
Previous Multicast Node Address
20
Current Multicast Node Address

Figure 8.5: Geographic Multicast packet format.

The proposed geographic multicast packet format is depicted in Figure 8.5. Bytes 10-13 of this packet correspond to the coordinates of its destination (Steiner point or a multicast destination). The Group ID is used by the *GeographicForwarding* application in order to retrieve the group members registered to the system under this identifier. The Previous Multicast Node Address is required by the *GeographicForwarding* application on ONOS in order to avoid sending the packet back to the previous node of the multicast tree, as the ONOS controller is stateless and thus, not aware of the path that a packet has followed. Finally, the Current Multicast Node Address is used by multicast nodes in order to know whether they are the recipients of this multicast packet and that they should ask their controller (ONOS) for the next node in the tree.

GeographicForwarding application on ONOS installs two flow rules in each multicast node or Steiner point and one flow rule in all the remaining nodes. The flow rule installed in all nodes instructs them to call the installed INPP geographic function when a Coordinates packet (labeled as GEO_COORDINATES in the type field) or a Multicast packet (labeled as GEO_DATA packet in the type field) is received.

Then, the function will consider the destination coordinates and send the packet to the neighbor which is closest to the destination. In this way nodes can forward the packets until the next multicast node or Steiner point. To this purpose the ONOS application had previously installed another rule in the above nodes, which instructs them to send a request to the Controller upon receiving a GEO_DATA packet with Current Multicast Node Address equal to its own. Observe that this is a high priority rule in the WISE flow table.

When the *GeographicForwarding* application receives a GEO_DATA request, which format is analogous to the one shown in Figure 8.5, it checks the Group ID to extract the nodes of the group and asks the *MulticastTreeAlgo* to get the next multicast node or Steiner point from the multicast tree. After the next node has been retrieved, it simulates the geographic path to see whether there are holes or loops in the path. In case there are, it finds an alternative path and it enforces it with the standard SDN-WISE forwarding rules to the corresponding nodes. Otherwise, it changes the packet by setting the requesting node in the Previous Multicast Node Address and the next one in Current Multicast Node Address field and sends the packet back to the requesting node.

Note that in this way there is no need to send the entire Euclidean Steiner tree calculated by the Controller to sensor nodes and thus, the signaling overhead decreases.

Considering the example of Figure 8.2 where node A is the multicast source and nodes B and C are the multicast destinations, node A will create a GEO_DATA packet, where the Initiator, Previous and Current Multicast Node Address are the same (i.e. node A). When it will attempt to send the packet, the flow rule which instructs the node to ask its Controller will be triggered and the packet will be eventually delivered to the *GeographicForwarding* application in ONOS. The latter will use the GeoSteiner algorithm to find the next node in the tree, node E, which is a Steiner point. Then, it will send the address of node E as the Current Multicast Node Address and send the packet back to A. When A receives it, the geographic forwarding function will be called and it will be forwarded to the next intermediate hop as shown in the figure. Each intermediate hop will run the same procedure until the packet arrives at node E, which will follow the same procedure with A, since the Current Multicast Node Address is its own address. The difference is that, in this case, the *GeographicForwarding* application will reply with two packets, one for each branch towards the multicast destinations B and C.

In case of the geographic unicast forwarding, a simplified version of the above procedures will be executed.

In fact, in the unicast case the operations executed by forwarding nodes located between the source and the destination are analogous to those illustrated so far for nodes located between two multicast nodes (i.e. Steiner points nor sources or multicast destinations).

8.3 Performance evaluation

In this section we will assess the performance of the proposed approach in terms of signaling overhead, number of flow table entries, path length, energy consumption and computation time.

We consider a 80x80 m^2 area with 100 nodes. Positions of nodes were generated randomly according to a uniform distribution. There is one sink (node 0) located at position (79,19) which acts as a gateway between the WSN and the rest of the world (including the ONOS controller). We consider both the case of unicast and multicast forwarding. In particular:

- *Unicast case*: we have a source node i.e., node 1 located in (56,31) which communicates in unicast with any of the other 99 nodes;
- *Multicast case*: we have a source node (i.e., node 1) which sends multicast flows to the multicast group members. In particular the number of multicast group members varies up to 10 members.

All nodes know their own coordinates, as well as the coordinates of their immediate neighbors according to the algorithm discussed in the previous sections.

We emulated the network behavior in Mininet and we extended the SDN-WISE standard data packet by setting the type as `GEO_DATA` when geographic forwarding is applied. The packet header is 10 bytes long and the payload length is 11 bytes.

8.3.1 Unicast case

We preliminarily compare the following approaches:

- **Shortest path** where the Controller estimates the shortest path to reach the destination using Dijkstra algorithm and sends back this information to the source node so that intermediate nodes simply relay the packet according to a pre-computed path;
- **Geographic-CTRL** where the Controller preliminarily executes the GeoSteiner algorithm to identify the routing path so that intermediate nodes simply relay the packet according to a pre-computed path;
- **Geographic-DIST** where the distributed geographic forwarding is implemented as described in the previous sections.

In Figure 8.6 we show the CDF of the overall number of signaling messages sent to install the required flow table entries in the nodes. Observe that by comparing the Geographic-CTRL and DIST it is evident that Geographic-DIST is the one that needs less signaling. The advantage of this distributed approach is also evident by looking at Figure 8.7 where we report the CDF of the number of flow table rules at nodes. Note that Geographic-DIST gives better performance than Geographic-DIST always. Therefore from now on we will not consider the Geographic-CTRL anymore. Furthermore, we will denote the "Geographic-DIST" case as "Geographic".

Also, in Figures 8.7 and 8.6 we observe that the shortest path as compared to the geographic case, requires a higher number of rules and signaling messages.

In Figure 8.8 we report the CDF of the overall energy consumption in mJ. In order to obtain this Figure we have emulated a set of Embit EMB-Z2530PA/IA devices [104] where the consumption is 135 mA in TX mode, 28 mA in RX mode and 6.8 mA in idle mode. Observe that the geographic approach allows to drastically reduce the energy consumption as compared to the shortest path.

Finally we compared the path length obtained in the shortest path and geographic cases. More specifically, in Figures 8.9 and 8.10, we show the PDF of the number of hops and the CDF of the geometric path length needed in the two cases, respectively. Finally we have evaluated the impact of errors in the estimation of the node position on the performance of the proposed scheme. In fact, previous studies have highlighted that geographic routing is vulnerable to even low localization errors: even localization errors in the order of tens percent of the radio coverage can reduce the delivery rate significantly [105]. However, as we have explained in Section 8.1.2, the Controller can predict and thus, avoid protocol

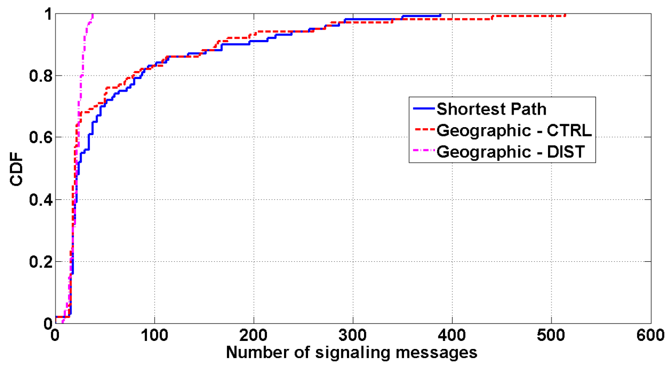


Figure 8.6: CDF of the overall number of signaling messages for different unicast forwarding strategies.

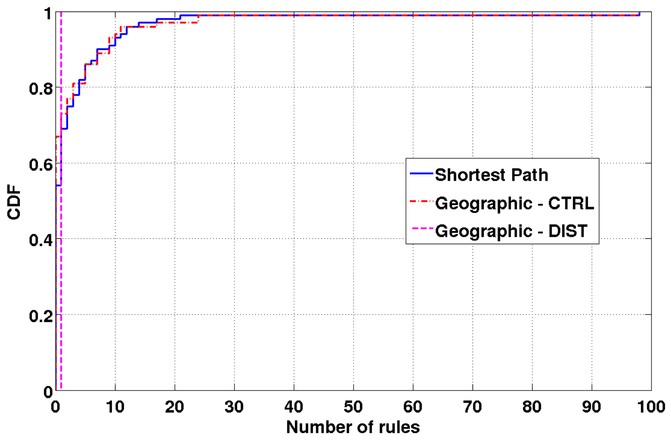


Figure 8.7: CDF of the number of rules for different unicast forwarding strategies.

failure. As a result, the delivery rate is not impacted by localization errors and our experimental results show that when such errors are in the order of the radio coverage, the resulting increase in the path length is below 0.5%.

8.3.2 Multicast case

In Figure 8.11 we show the Steiner tree evaluated by the controller applying the Dreyfus-Wagner algorithm [106] and the tree obtained using our approach. Observe that the two topologies are very close to each other.

In Figures 8.12 and 8.13 we compare the execution time needed to calculate

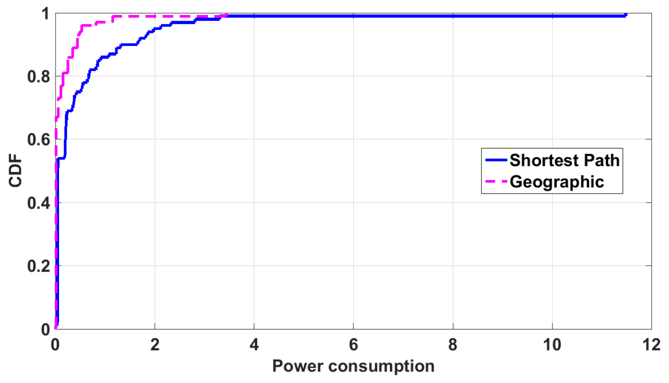


Figure 8.8: CDF of the energy consumption in the unicast case for the considered forwarding strategies.

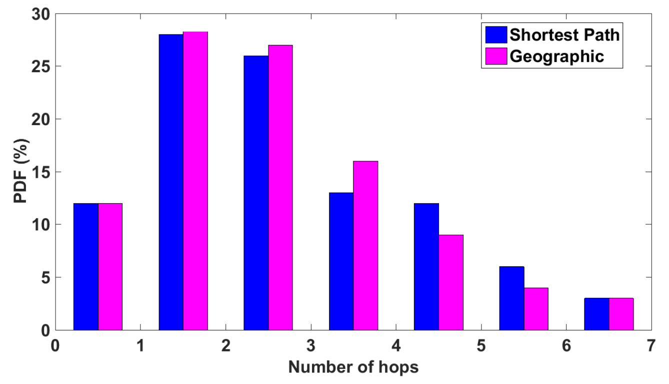


Figure 8.9: PDF of the number of hops needed in the unicast case for the considered forwarding strategies.

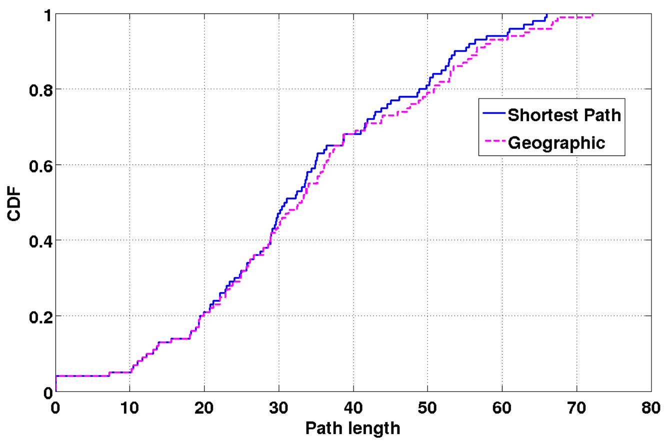


Figure 8.10: CDF of the path length implied in the unicast case for the considered forwarding strategies.

the Steiner tree using the Dreyfus-Wagner algorithm and using our mechanism, respectively. Results have been obtained using a 1.8 GHz Intel Core i5 personal computer equipped with 8GB of RAM running Windows 8. In particular in Figure 8.12 we report the execution time vs. the number of multicast destinations for different number of nodes in the network. Observe that in spite of the very limited number of destinations considered, the time needed is in the order of hundreds or thousands of seconds. Comparing Figure 8.12 and 8.13 it is evident that the execution time obtained by our algorithm is 3-4 orders of magnitude lower than the one obtained applying the Dreyfus-Wagner algorithm. While such performance difference may be impacted by implementation inefficiencies, it is clear that the computation complexity is exponential in Figure 8.12 and linear in Figure 8.13, thus assessing the effectiveness of the geographic approach.

Concerning the path length experimental results show that the performance reduction obtained by the geographic approach compared to what would be obtained using the Steiner tree is negligible. Furthermore, non obvious results are presented in Figure 8.14 where we show the distribution of the path length (in hops) between the source and each of the multicast destinations. In Figure 8.14 the geographic approach gives shorter paths than what would have been obtained using the Steiner tree, which would result in shorter delays.

8.4 Conclusions

In this chapter we have investigated how geographic forwarding can be applied in software defined WSNs and what advantages can be achieved by using it. More specifically we have provided a complete solution that allows to apply geographic forwarding in software defined WSNs which is compliant with major frameworks

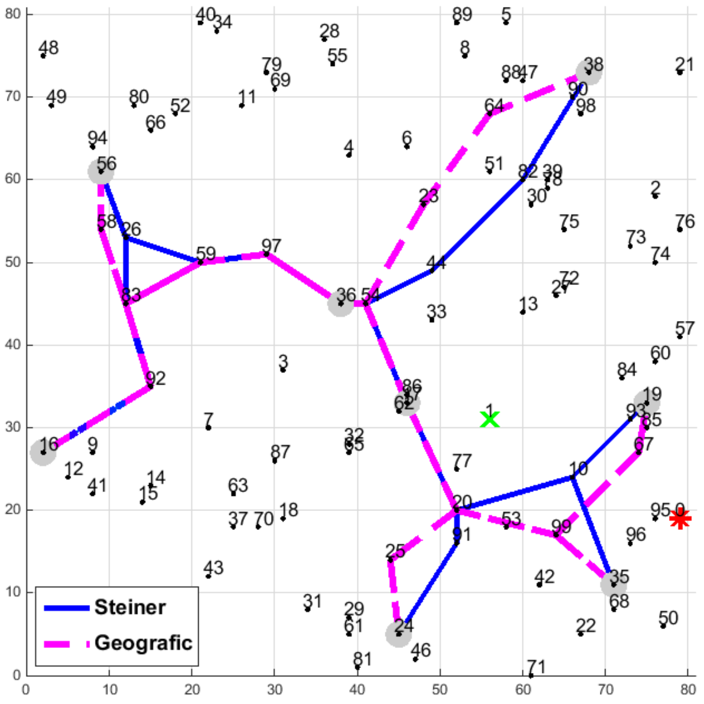


Figure 8.11: Multicast topologies considered in our work.

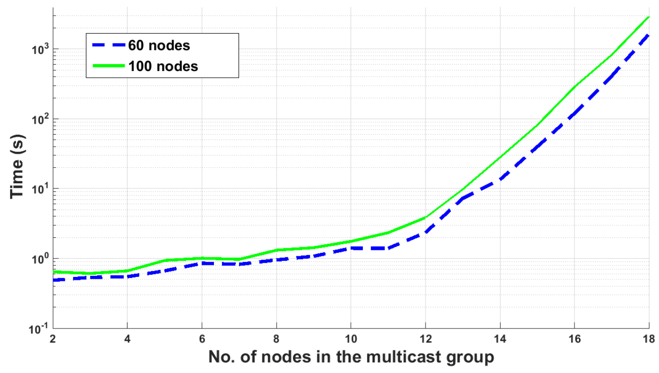


Figure 8.12: Time needed to compute the Steiner tree using Dreyfus-Wagner algorithm.

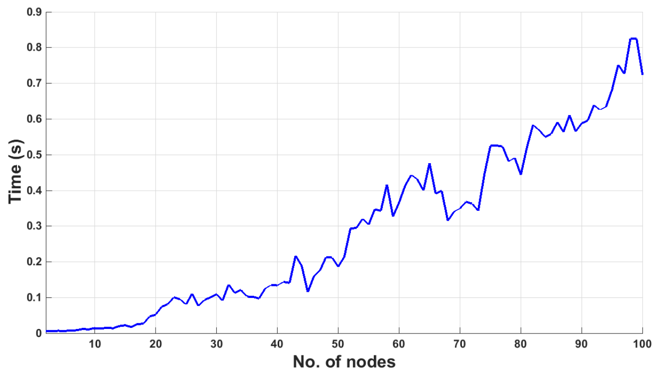


Figure 8.13: Time needed to compute the Steiner tree using GeoSteiner algorithm.

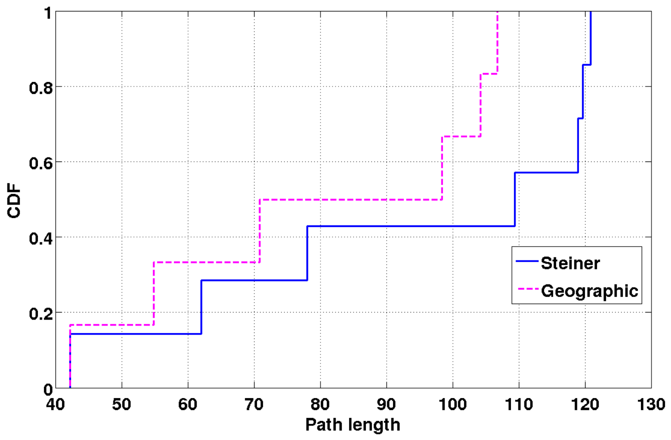


Figure 8.14: CDF of the path length in the multicast case for the considered forwarding strategies.

in the relevant domain.

Also, we have realized a complete prototype of the proposed solution. In our work we have assessed the advantages of applying geographic forwarding in WSNs in terms of reduction of signaling messages and memory occupancy for storing flow table entries. The performance advantages are obtained at the cost of a negligible increase in the path length. Furthermore, in the case of multicast applications the processing required at the Controller to calculate the optimal routes can be dramatically reduced.

Chapter 9

A Declarative Approach to SDWSN

Software Defined Networking (SDN) is a networking paradigm that has garnered a lot of interest among the academic and industrial communities as it promises to dramatically reduce the complexity of network configuration and management.

While there are few protocols on the Data plane side, we have witnessed a flourishing of software solutions on the Control plane even if, in most of the cases, these softwares were just easy-to-use facades to Data plane messages [107]. In fact, their main focus was on easing how to instruct a network to perform the actions needed to achieve a behavior (imperative approach) and not on providing the instruments to define the desired behavior and leave to the Control plane the configuration process (declarative approach).

A recent evolution towards a declarative approach is contaned in the Open Networking Operating System (ONOS) which is the concept of *intent*. An *intent* is an immutable model object that describes an application's request to the ONOS core to alter the network's behavior [15]. For example, given two nodes in a network, say A and B, if there is a connectivity intent between them, the controller will try its best to guarantee a communication channel between this two nodes by installing the required Flow Rules in the network and by autonomously updating them if the state of the network changes.

This chapter proposes a further step in this direction by allowing the administrators to state some general proprieties for a network, while it is the controller that decides how to turn such requests into messages for the Data plane.

In order to achieve such results the Control plane has to learn as much as possible from the network itself and it has to dynamically react to changes without any supervision. To this purpose, we leveraged the SDN paradigm and the programmability of the Control plane.

In fact, as the Control plane became a software module running on commodity hardware, it is easier to use existing libraries and algorithms for machine learning, data analysis, and data forecasting with the result of making the Control plane more *intelligent*.

By *intelligent* we mean that the Control plane can reduce the number of direct human interventions in the configuration process, using unsupervised learning algorithms and *Artificial Neural Networks* (ANN).

Therefore, the main contribution of this chapter is providing a *declarative* architecture for network management and demonstrating how, by leveraging a Long Short-Term Memory Artificial Neural Network (LSTM-ANN), it is possible to implement an intelligent control plane by using a predictive flow instantiation algorithm.

To assess the feasibility of the proposed solution to a relevant use case we used SDN-WISE and ONOS and we have emulated a portion of the SmartSantander testbed using the datasets made available by the FESTIVAL project [108].

Accordingly, the remainder of this chapter is organized as follows: Section 9.1 introduces the related works on the field of machine learning. In Section 9.2 the architecture of the proposed solution is reported while Section 9.3 describes the dataset, the topology, and the testbed used. Section 9.4 presents the predictive flow instantiation algorithm and finally, in Section 9.5, achieved results are shown and then conclusions are stated in Section 9.6.

9.1 Related work

As the nature of this chapter is to bring together different research topics, the content of this section is split into three parts. First we provide a brief description of the current Declarative Control plane solutions in Section 9.1.1, then we describe the existing works on machine learning solutions applied to network management in Section 9.1.2 and finally, since we exploit Artificial Neural Networks, we provide some theoretical background in Section 9.1.3.

9.1.1 Intents

Among the principal NOS, OpenDayLight (ODL) and the Open Networking Operating System (ONOS) are gaining a lot of traction in the research community as already analyzed in Chapter 6. For what concerns the aim of this chapter, we will briefly describe the concept of Intent and the ONOS Intent framework. An Intent, both in ONOS and ODL, is used to explicate a network behavior providing generalized and abstracted policy semantics instead of specific configuration commands. In particular, ONOS uses the intent specifications to translate this requirements into installable intents, which are essentially operations on the network environment. At the lowest levels, Intents may be described in terms of:

- Network Resource : A set of object models, such as links, that tie back to the parts of the network affected by an intent.

- Constraints : Weights applied to a set of network resources
- Criteria : Packet header fields or patterns that describe a slice of traffic.
- Instructions : Actions to apply to a slice of traffic, such as header field modifications, or outputting through specific ports.

Although this can be considered as a first step in the evolution towards a declarative approach, the full potential of a logically centralized and omniscient Control plane can be unlocked only by providing more autonomy to the Control plane itself, therefore a key role is played by Machine learning algorithms.

9.1.2 Machine Learning for Network Management

Machine learning and in particular Artificial Neural Networks (ANN) have been widely used in the past few decades as a tool capable of directing traffic control in wired/wireless networks. The potentiality and theoretical aspects of ANN will be briefly detailed in section 9.1.3, however in this section we provide an overview of the applications of machine learning algorithms in network management. Machine learning can be considered at the base of self organizing network. An extensive literature exists on the topic [109], therefore we consider only those works which exploits these techniques in the SDN context. In [110] the authors provided a machine learning based framework to predict the Quality of Experience in SDN. In [111], the authors compared four well known machine learning algorithms trained on historical network attack data, to predict network attack patterns in SDN networks. In [112] Particle Swarm Optimisation (PSO) and Genetic Algorithms (GA), are employed to find the best set of inputs that give the maximum performance of an SDN. In [113] it is presented a matheuristic for dynamic optical routing implemented as an application into a software-defined mobile carrier network using machine learning to predict tidal traffic variations. Despite the existence of a large corpus of papers on the application of machine learning algorithms to SDN networks, there are no papers about SD-IoT deployments.

9.1.3 Artificial Neural Networks

Artificial Neural Networks have been developed as a way to mimic the behavior of the human brain. An ANN is comprised of multiple interconnected nodes, called neurons, that closely resemble a neural network. Each neuron is connected to other neurons through weighted links and neurons are grouped together into layers. From a functional point of view a neuron contains an activation function which returns a value depending on the values provided by the incoming links multiplied by their respective weights. This value is used as input for other neurons and the process is repeated until the last group (layer) of neurons which returns the output of the ANN. The process that allows to select the weights of the links of the network is called training. For more details on ANN and training algorithms please refer to [114]

Among ANN, it is possible to distinguish two different kinds of networks: feed-forward and recurrent (RNN). In feed-forward networks all the connections between the neurons share the the same direction, from one layer to the next, and there are no connections between neurons at the same layer or connection providing inputs from a neuron to another of previous layers. This restriction is removed in recurrent neural networks. The major effect of such change is that the neural network presents a short term memory, depending on the inputs, as opposed to the long term memory acquired during the training phase. The main drawback of these ANN is the Vanishing Gradient (VG) problem. RNNs learn the weight by measuring how a small change in the weights will affect the network's output. If a change in the input causes a very small change in the output the network is not able to learn effectively [115].

To solve this issue Long Short Term Memory ANN (LSTM-ANN) were introduced in [115]. These RNNs are able to reduce the VG using some special units called gates that can change the weights or truncate the gradient when needed.

The applications of LSTM are multiple: natural language process, handwriting recognition, and, for what concerns the topic of this paper, time series analysis and prediction [116].

9.2 Proposed Solution

In this section we briefly describe the proposed solution. More specifically, we first describe the proposed architecture in Section 9.2.1, then, in Section 9.2.2 we describe how the routing strategy exploits the data produced by neural network.

9.2.1 Architecture

The architecture of the proposed work is mainly built as a software suite on top of the ONOS network operating system. Our software suite can be divided into three modules: the performance specification module, the measurement module, and the prediction module.

The **Performance Specification module (PSM)** is in charge of accepting the requirements from the user and translating such requirements into an objective function which should be maximized. In our current implementation we mainly focused on two performance metrics, i.e., totale energy consumption and fairness. Therefore, the objective function can be easily determined through a real value in the range [0, 1], which identifies the weights of energy consumption and fairness in the objective function.

The **Measurement module (MM)** is based on the ONOS REST APIs which are used to collect the amount of traffic traversing each link of the network. This information is used by the prediction module to train the LSTM-ANN.

The **Prediction module (PM)** includes the LSTM-ANNs that are used to predict the traffic load in a future time slot of traffic. If the ANN predicts a change in the traffic produced by the nodes that justifies a change in the forwarding plane according to the objective function set by the user then new corresponding Flow Rules are deployed in the network.

9.2.2 Routing Strategy

The capability to predict the amount of data generated by each node is used to reshape the routing algorithm of the network. Using the prediction, it is possible to create a weight function in which each vertex has a weight w' depending on the formula:

$$w'(x, y) = a \cdot w(x, y) + (1 - a) \cdot p(y) \tag{9.1}$$

where w is the weight of the edge between nodes x and y[1], $p(y)$ is the amount of data that will be generated by the node y, as predicted by the LSTM-ANN, and a is the tuning parameter imposed by the performance specification module based on the user's preferences. The optimal routing in this context can be implemented using Dijkstra's shortest path algorithm on a new graph G_i obtained from the original graph G in which the weight of each edge is calculated according to the formula in 9.1.

Another important aspect of our analysis consists on deciding how much traffic should be predicted and when to re-run the configuration of the network. In terms

[1]For the sake of simplicity, in our experiments in the following Section 9.5 we consider $w(x, y) = 1$ for all the edges, but in a real environment other values could be included such as the battery level of the node or the link quality

of costs, changing a path in a network means that the controller has to send an OpenPath packet to the nodes belonging to the new path (for details on routing in SDN-WISE please refer to [93]).

The cost of the operation depends on the number of nodes involved in the change and the benefit obtained by changing the routing in the network. Therefore, before selecting a new routing strategy a comparison between the two options is executed and this tradeoff is resolved according to the user preferences.

9.3 Testbed

To test our solution we have implemented a simulated environment based on real data using Cooja, SDN-WISE and ONOS.

9.3.1 Dataset and Topology

The datasets we used has been taken by the FESTIVAL platform exploiting the data produced by the parking spot sensors in the city of Santander (Spain) [117]. The complete SD-IoT system is made of 309 wireless sensor nodes, that collect the data, 37 wireless relay nodes that are used for routing, 3 gateways and a datacenter. The placement of all the devices is provided in Figure 9.1.

The blue dots are the nodes, the red ones the repeaters and the black ones the gateways. Each node is connected to the closest relay node and the relay nodes form a mesh network connected to the gateways. Finally, all the gateways are connected to each other and can communicate with the datacenter.

The topology of the network made of the relay nodes is provided in Figure 9.2. This topology has been simulated using the data provided in [117], therefore there might be some discrepancies between the simulated and the real connectivity matrix. However, considering the purpose of this work, such differences can be

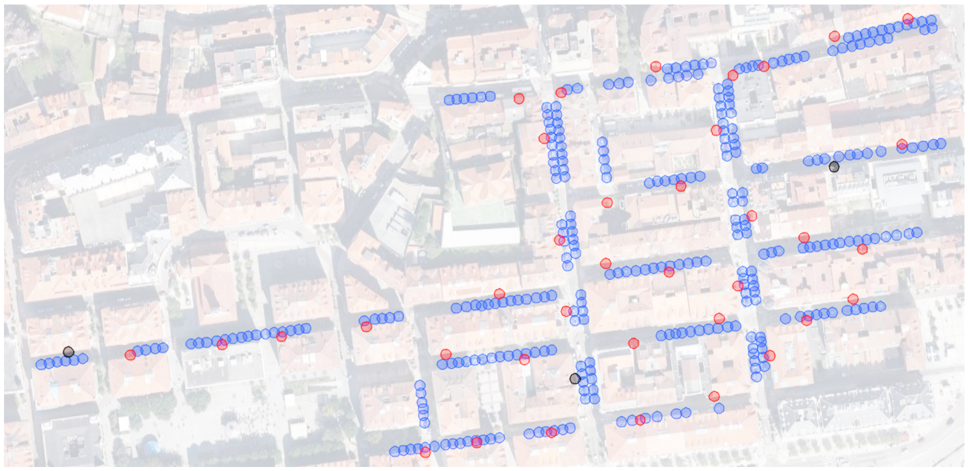
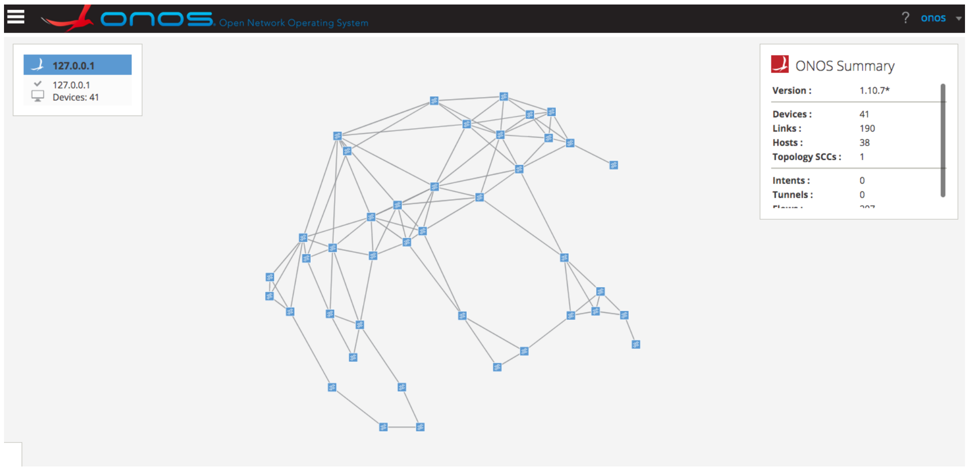


Figure 9.1: Placement of the IoT devices in Santander.



ONOS Summary

Version :	1.10.7*
Devices :	41
Links :	190
Hosts :	38
Topology SCCs :	1
Intents :	0
Tunnels :	0

Figure 9.2: Network topology.

neglected as the focus is on the prediction of patterns in the data not on the exact replication of the testbed.

For what concerns the generation of the data, each node reports every change of state in the parking spot using a message containing: a measurement ID, the ID of the node, a timestamp and the state of the parking spot. Each message is sent to the closest relay node and forwarded to the closest gateway and then collected by the datacenter. However, this work focuses on tuning the network of the relay nodes, thus when in the rest of this chapter we refer to the data produced by a relay node, we actually mean the aggregation of the data produced by the nodes directly connected to a relay node. The dataset considered contains 1,580,807 messages sent from January 1, 2016 to December 31, 2016. The messages have been split into 13542 groups depending on their date and their first relay node. Then, for each of these groups, we counted the number of messages sent for each hour, creating 366 24-dimentional vectors for each relay node. These vectors have been enriched with two additional values indicating the day of the week and if that day was a public holiday or not.

9.3.2 Simulated Testbed

The feasibility of the proposed approach has been proven in a simulated environment made of different software modules, in particular:

- Cooja, Mininet, and SDN-WISE to model the network.
- ONOS to implement the Control Plane of the network.
- MATLAB and Python to train the LSTM-ANNs and predict the network traffic.

Cooja is a network simulator developed for Contiki [69]. It allows to create networks of emulated and/or simulated wireless sensors. In our case we used Cooja to build a network of 37 wireless sensor nodes and 3 gateways replicating the relay nodes in Santander. Each nodes uses SDN-WISE at the network layer, and it is controlled by the ONOS control plane. Mininet [53] allows to create virtual networks of OpenFlow devices and virtual hosts. In out testbed we leveraged Mininet to model the network of the gateways and the datacenter. MATLAB and Python have been used to train 37 different LSTM-ANNs, one for each relay node. Each LSTM-ANN is made of 4 neurons in the input layer, one for each of the variables considered (i.e. day of the week, hour of the day, holiday, generated packets), 50 neurons in the first hidden layer, and one neuron in the output layer to predict the number of packets generated by the relay node. Each model is fitted for 50 training epochs with a batch size of 72. This number represents the number of input elements after which the internal state of the LSTM-ANN is reset and in our case it consists of 3 days of data. Each neural network has been trained with the data from January to October, and tested on November and December. A week of prediction for a relay node is shown in Figure 9.3.

9.4 Predictive Flow Instantiation

As described in Section 9.2.2 the routing in the network exploits a prediction of the amount of traffic which will be produced by the sensor nodes. Therefore, a prediction algorithm is run periodically, (e.g. in our testbed we used a 1 hour time slot). More specifically, at the end of each slot the Algorithm in 1 is executed:

If there is no data on the traffic, networking policies are implemented using Dijkstra's shortest path algorithm and the value of a in eq. (9.1) is set equal to 1.

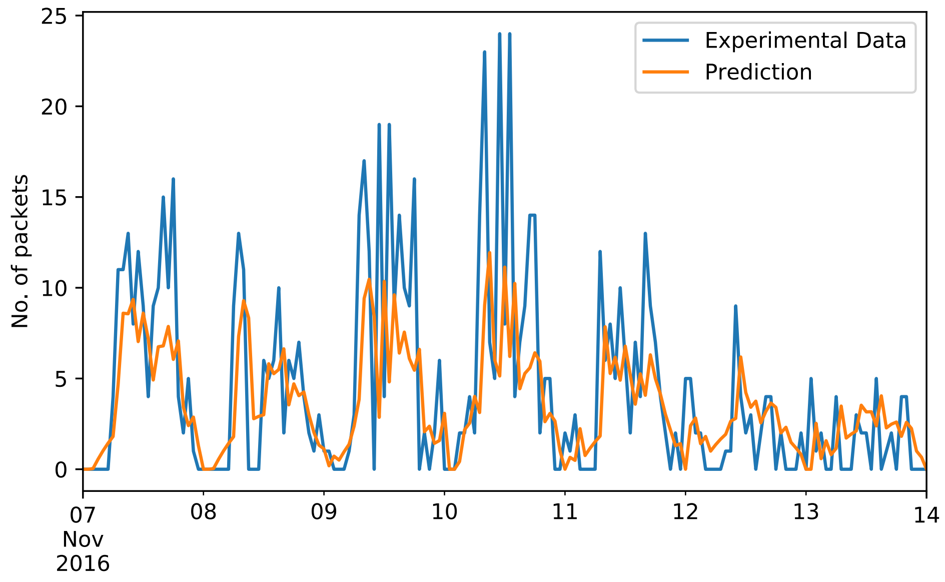


Figure 9.3: A comparison between the experimental data and the predicted data for a relay node.

Algorithm 1 Prediction Algorithm

```
traffic = []
topo = getNetworkTopology()
weightedTopo = setWeights(a=1, b=0, topo)
pThreshold = getThreshold()
while (1) do
    currentTraffic = getTrafficData()
    traffic.append(currentTraffic)
    prediction = predictTraffic(traffic)
    pCurrent = getPacketsToBeSent(weightedTopo)
    newWeightedTopo = setWeights(a, b, topo, traffic)
    pPredicted = getPacketsToBeSent(newWeightedTopo)
    pRules = getUpdateCost(weightedTopo)
    if (pCurrent >pPredicted + pThreshold + pRules) then
        paths = Dijkstra(newWeightedTopology)
        updateFlowRules(paths)
        weightedTopo = newWeightedTopo
    end if
    waitForNextSlot()
end while
```

Otherwise, the prediction algorithm uses the data from t_0 to t to predict the traffic generated by each relay node at $t+1$ (predictTraffic(traffic)). Then it computes:

- $p_{current}$: the number of data packets predicted to be sent in the network in $t+1$, using the actual routing policies.
- $p_{predicted}$: the number of data packets predicted to be sent in the network in $t+1$, using the weight function in eq. (9.1).

If $p_{current}$ is higher than $p_{predicted}$ plus the number of packets needed to reprogram the devices and plus a user defined threshold (which gives some hysteresis) then the Flow Rules implementing the new routing policies are deployed in the network and the weights are updated.

The main outcome of this approach is that by using a predictive flow instantiation, there are situations in which longer paths are used to balance energy consumption among nodes of the network, (Fair Configuration (FC)).

9.5 Results

In order to prove the flexibility of the proposed solution, we have conducted three different measurement campaigns, tuning the network with different values of a. The Dijkstra Configuration ($a = 1$) and Fair Configuration ($a = 0$), as defined in section 9.4, and the Oracle configuration (OC), in which there is no prediction and the data at $t+1$ is returned from the dataset.

Figure 9.4 shows the average number of packets sent in the network from November to December by each of the nodes in the three different configurations. These measurements have been normalized using the DC as reference. It is possible to notice that the DC produces the highest number of packets, ending with a

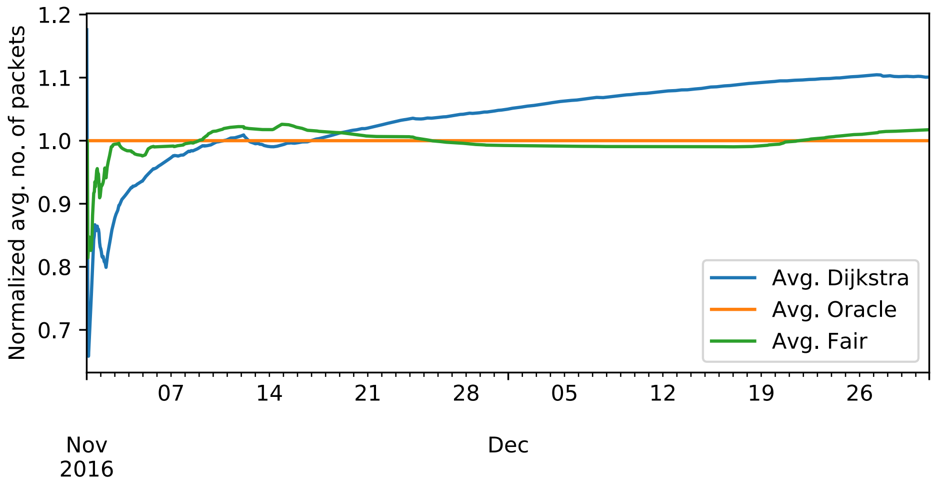


Figure 9.4: Overall number of packets transmitted since November 1st, vs time, normalized by overall number of packets transmitted in the "Oracle" case.

+10% when compared to the OC. On the other hand, to measure the fairness of the proposed solution the standard deviation of the number of packets for each node is reported in Figure 9.5. The OC and the FC achieve the lowest standard deviations. In both figures it is also possible to notice that the FC does not differ too much from the OC. As a result, by tuning the values of a it is possibile to select the preferred tradeoff between the values reported for the FC and for the DC.

9.6 Conclusions

In this chapter we have presented a general architecture for an SD-IoT management system based on a LSTM-ANN. We tested our approach on a real dataset inside a simulated environment and we have shown that it is possible to leverage the predictability of the human behavior to tune the performances of the system.

The proposed solution aims at providing the starting point for a wider declarative, SDN-based, predictive flow rule instantiation system in which the network management depends on a set of desired parameters, not limited to just the

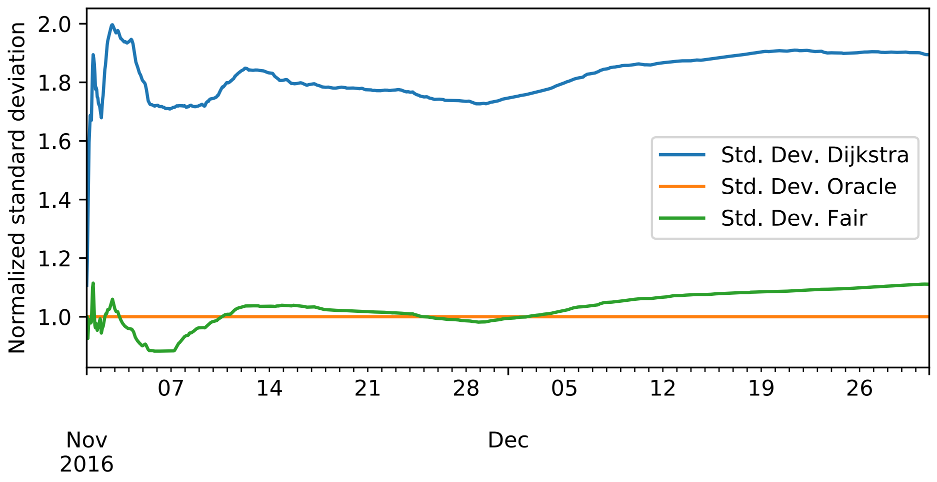


Figure 9.5: Overall standard deviation for the number of packets transmitted since November 1st, vs time, normalized by the Standard deviation for the number of packets transmitted in the "Oracle" case.

relationship between energy efficiency and fairness, and it is the Control plane that autonomously decides the policies to be implemented in order to make such policies effective.

Chapter 10

Conclusions and Future Work

This dissertation presented a novel approach based on Software Defined Networking to control IoT devices. This solution has been extensively tested in different scenarios showing that a SDWSN has many advantages compared to existing solutions, in particular experimental results show that in static and quasi-static conditions a Software Defined approach outperforms standard solutions, independently on the network size, payload size, traffic generated, and performance metric considered. The reason for this is the fact that such approach allows to optimize paths selection and minimize forwarding time at routers. Furthermore:

- SDN-WISE allows the *softwarization* of network management for WSNs. Programming the network from the outside is easier because the interfaces used are platform independent and can be upgraded or modified without changing the firmware running on the wireless sensor nodes.

- SDN-WISE allows to easily implement QoS policies thanks to its statefulness and its programmability. In fact a node can manage different classes of traffic depending on its congestion condition.

- The proposed solution can be used to include WSNs into Network Operating Systems unlocking the interaction between wired and wireless sensor networks.

In particular ONOS which mainly covers OpenFlow networks, can now instruct WSNs. As a result, interaction between SDN-WISE and OpenFlow networks becomes seamless, with the NOS deciding the forwarding paths considering the whole topology and providing the appropriate commands for each device type.

- Such interaction can be used to optimize routing and energy consumption leveraging advanced techniques like Geographical routing or Artificial Neural Networks.

Given these conclusions, further work could be focused on further integrating SDN-WISE within existing IoT solution, both for what concerns IoT Operating Systems, like RIoT and Contiki, and protocols, like 6LoWPAN and ZigBee.

Another step could be done in the direction of extending the software defined approach towards other layers of the communication stack. For example allowing the controller to decide the physical channel used by each of the wireless nodes in a network.

In addition, it is of primarily importance studying and implementing security related aspects in order to guarantee a secure and authenticated communication channel among the nodes and between the nodes and the controller.

Bibliography

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow," *ACM SIGCOMM Computer Communication Review*, vol. 38, p. 69, mar 2008.

[2] M. Abo-Zahhad, O. Amin, M. Farrag, and A. Ali, "A Survey on Protocols, Platforms and Simulation Tools for Wireless Sensor Networks," *International Journal of Energy, Information and Communications*, vol. 5, pp. 17–34, dec 2014.

[3] H. Malazi, K. Zamanifar, and S. Dulman, "FED: Fuzzy Event Detection model for Wireless Sensor Networks," *International Journal of Wireless & Mobile Networks*, vol. 3, pp. 29–45, dec 2011.

[4] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, pp. 2292–2330, aug 2008.

[5] H. T. Malazi, K. Zamanifar, A. Pruteanu, and S. Dulman, "Gossip-based density estimation in dynamic heterogeneous wireless sensor networks," *International Journal of Autonomous and Adaptive Communications Systems*, vol. 7, no. 1/2, p. 151, 2014.

[6] H. T. Malazi, K. Zamanifar, A. Khalili, and S. Dulman, “DEC: Diversity-based energy aware clustering for heterogeneous sensor networks,” *Ad-Hoc and Sensor Wireless Networks*, vol. 17, no. 1-2, pp. 53–72, 2013.

[7] P. M. Pawar, R. H. Nielsen, N. R. Prasad, and R. Prasad, “Mobility Impact on Cluster Based MAC Layer Protocols in Wireless Sensor Networks,” *Wireless Personal Communications*, vol. 74, pp. 1213–1229, feb 2014.

[8] H. K. D. Sarma, A. Kar, and R. Mall, “A Hierarchical and Role Based Secure Routing Protocol for Mobile Wireless Sensor Networks,” *Wireless Personal Communications*, vol. 90, pp. 1067–1103, oct 2016.

[9] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: a survey,” *Computer Networks*, vol. 38, pp. 393–422, mar 2002.

[10] 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, pp. 386–412, aug 2007.

[11] L. Mottola and G. P. Picco, “Programming wireless sensor networks,” *ACM Computing Surveys*, vol. 43, pp. 1–51, apr 2011.

[12] J. Qadir, N. Ahmed, and N. Ahad, “Building programmable wireless networks: an architectural survey,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2014, p. 172, dec 2014.

[13] J. Wickboldt, W. De Jesus, P. Isolani, C. Both, J. Rochol, and L. Granville, “Software-defined networking: management requirements and challenges,” *IEEE Communications Magazine*, vol. 53, pp. 278–285, jan 2015.

[14] S. S. et al., “The future of networking and the past of protocols,” 10 2011.

[15] P. Berde, W. Snow, G. Parulkar, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, and P. Radoslavov, "ONOS," in *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, (New York, New York, USA), pp. 1–6, ACM Press, 2014.

[16] P. Xiao, W. Qu, H. Qi, Z. Li, and Y. Xu, "The SDN controller placement problem for WAN," in *2014 IEEE/CIC International Conference on Communications in China (ICCC)*, pp. 220–224, IEEE, oct 2014.

[17] L. Velasco, A. Asensio, J. Berral, A. Castro, and V. López, "Towards a carrier SDN: an example for elastic inter-datacenter connectivity," *Optics Express*, vol. 22, p. 55, jan 2014.

[18] N. A. Jagadeesan and B. Krishnamachari, "Software-Defined Networking Paradigms in Wireless Networks: A Survey," *ACM Computing Surveys*, vol. 47, pp. 1–11, nov 2014.

[19] F. Dressler, I. Dietrich, R. German, and B. Krüger, "A rule-based system for programming self-organized sensor and actor networks," *Computer Networks*, vol. 53, pp. 1737–1750, jul 2009.

[20] T. Luo, H.-P. Tan, and T. Q. S. Quek, "Sensor OpenFlow: Enabling Software-Defined Wireless Sensor Networks," *IEEE Communications Letters*, vol. 16, pp. 1896–1899, nov 2012.

[21] D. Zeng, T. Miyazaki, S. Guo, T. Tsukahara, J. Kitamichi, and T. Hayashi, "Evolution of Software-Defined Sensor Networks," in *2013 IEEE 9th Inter-*

national Conference on Mobile Ad-hoc and Sensor Networks, pp. 410–413, IEEE, dec 2013.

[22] A. Mahmud, R. Rahmani, and T. Kanter, "Deployment of Flow-Sensors in Internet of Things' Virtualization via OpenFlow," in *Proceedings - 2012 3rd FTRA International Conference on Mobile, Ubiquitous, and Intelligent Computing, MUSIC 2012*, pp. 195–200, 2012.

[23] P. Dely, A. Kassler, and N. Bayer, "OpenFlow for Wireless Mesh Networks," in *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–6, IEEE, jul 2011.

[24] M. Abrignani, C. Buratti, D. Dardari, N. El Rachkidy, A. Guitton, F. Martelli, A. Stajkic, and R. Verdone, "The EuWIn testbed for 802.15.4/zigbee networks: From the simulation to the real world," *Proceedings of the International Symposium on Wireless Communication Systems*, vol. 9, pp. 365–369, 2013.

[25] T. Miyazaki, S. Yamaguchi, K. Kobayashi, J. Kitamichi, Song Guo, T. Tsukahara, and T. Hayashi, "A software defined wireless sensor network," in *2014 International Conference on Computing, Networking and Communications (ICNC)*, pp. 847–852, IEEE, feb 2014.

[26] B. Trevizan de Oliveira, L. Batista Gabriel, and C. Borges Margi, "TinySDN: Enabling Multiple Controllers for Software-Defined Wireless Sensor Networks," *IEEE Latin America Transactions*, vol. 13, pp. 3690–3696, nov 2015.

[27] R. Riggio, K. M. Gomez, T. Rasheed, J. Schulz-Zander, S. Kuklinski, and M. K. Marina, "Programming Software-Defined wireless networks," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, pp. 118–126, IEEE, nov 2014.

[28] C. Bernardos, A. de la Oliva, P. Serrano, A. Banchs, L. Contreras, H. Jin, and J. Zuniga, "An architecture for software defined wireless networking," *IEEE Wireless Communications*, vol. 21, pp. 52–61, jun 2014.

[29] S. Costanzo, L. Galluccio, G. Morabito, and S. Palazzo, "Software Defined Wireless Networks: Unbridling SDNs," in *2012 European Workshop on Software Defined Networking*, pp. 1–6, IEEE, oct 2012.

[30] "IEEE 802.15.4," `http://www.ieee802.org/15/pub/TG4.html`.

[31] J. A. Stankovic, "Research Directions for the Internet of Things," *IEEE Internet of Things Journal*, vol. 1, pp. 3–9, feb 2014.

[32] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the Internet of Things: A Standardization Perspective," *IEEE Internet of Things Journal*, vol. 1, pp. 265–275, jun 2014.

[33] "ZigBee," `http://www.zigbee.org/`.

[34] N. Kushalnagar, G. Montenegro, and C. Schumacher, "Ipv6 over low-power wireless personal area networks (6lowpans): Overview, assumptions, problem statement, and goals," 01 2007.

[35] Lili Liang, Lianfen Huang, Xueyuan Jiang, and Yan Yao, "Design and implementation of wireless Smart-home sensor network based on ZigBee

protocol," in *2008 International Conference on Communications, Circuits and Systems*, pp. 434–438, IEEE, may 2008.

[36] S. D. T. Kelly, N. K. Suryadevara, and S. C. Mukhopadhyay, "Towards the Implementation of IoT for Environmental Condition Monitoring in Homes," *IEEE Sensors Journal*, vol. 13, pp. 3846–3853, oct 2013.

[37] C. Gezer and C. Buratti, "A ZigBee Smart Energy Implementation for Energy Efficient Buildings," in *2011 IEEE 73rd Vehicular Technology Conference (VTC Spring)*, pp. 1–5, IEEE, may 2011.

[38] E. D. Pinedo-Frausto and J. A. Garcia-Macias, "An experimental analysis of Zigbee networks," in *2008 33rd IEEE Conference on Local Computer Networks (LCN)*, pp. 723–729, IEEE, oct 2008.

[39] M. Franceschinis, C. Pastrone, M. A. Spirito, and C. Borean, "On the performance of ZigBee Pro and ZigBee IP in IEEE 802.15.4 networks," in *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 83–88, IEEE, oct 2013.

[40] B. Pediredla, K. I. Wang, Z. Salcic, and A. Ivoghlian, "A 6LoWPAN implementation for memory constrained and power efficient wireless sensor nodes," in *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pp. 4432–4437, IEEE, nov 2013.

[41] G. Pellerano, M. Falcitelli, M. Petracca, and P. Pagano, "6LoWPAN conform ITS-Station for non safety-critical services and applications," in *2013 13th International Conference on ITS Telecommunications (ITST)*, pp. 426–432, IEEE, nov 2013.

[42] S. Dawans, S. Duquennoy, and O. Bonaventure, "On link estimation in dense RPL deployments," in *37th Annual IEEE Conference on Local Computer Networks – Workshops*, pp. 952–955, IEEE, oct 2012.

[43] M. Kovatsch, M. Weiss, and D. Guinard, "Embedding internet technology for home automation," in *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, pp. 1–8, IEEE, sep 2010.

[44] E. Toscano and L. Lo Bello, "Comparative assessments of IEEE 802.15.4/ZigBee and 6LoWPAN for low-power industrial WSNs in realistic scenarios," in *2012 9th IEEE International Workshop on Factory Communication Systems*, pp. 115–124, IEEE, may 2012.

[45] C. Perkins and E. Royer, "Ad-hoc on-demand distance vector routing," in *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*, pp. 90–100, IEEE, 1999.

[46] T. Winter, P. Thubert, A. Brandt, T. H. Clausen, J. W. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, and J. Vasseur, "Rpl: Ipv6 routing protocol for low power and lossy networks," *Work In Progress), http://tools. ietf. org/html/draft-ietf-roll-rpl-19*, no. July, pp. 1–164, 2011.

[47] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 44–51, apr 2014.

[48] Wei Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," in *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 1567–1576, IEEE, 2002.

[49] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann, "Impact of network density on data aggregation in wireless sensor networks," in *Proceedings 22nd International Conference on Distributed Computing Systems*, pp. 457–458, IEEE Comput. Soc, 2002.

[50] A. Manjeshwar and D. Agrawal, "TEEN: a routing protocol for enhanced efficiency in wireless sensor networks," in *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, no. C, pp. 2009–2015, IEEE Comput. Soc, 2001.

[51] C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network coding," *ACM SIGCOMM Computer Communication Review*, vol. 36, p. 63, jan 2006.

[52] L. Keller, E. Atsan, K. Argyraki, and C. Fragouli, "SenseCode," *ACM Transactions on Sensor Networks*, vol. 9, pp. 1–20, mar 2013.

[53] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*, (New York, New York, USA), pp. 1–6, ACM Press, 2010.

[54] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, pp. 2787–2805, oct 2010.

[55] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury, "A survey on wireless multimedia sensor networks," *Computer Networks*, vol. 51, pp. 921–960, mar 2007.

[56] F. Hu, Q. Hao, and K. Bao, "A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014.

[57] H. Moura, G. V. C. Bessa, M. A. M. Vieira, and D. F. Macedo, "Ethanol: Software defined networking for 802.11 Wireless Networks," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 388–396, IEEE, may 2015.

[58] L. Zhong, K. Nakauchi, and Y. Shoji, "Performance analysis of application-based QoS control in software-defined wireless networks," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 464–469, IEEE, aug 2014.

[59] X. Song, C. Wang, and J. Pei, "2ASenNet: A multiple QoS metrics hierarchical routing protocol based on swarm intelligence optimization for WSN," in *2012 IEEE International Conference on Information Science and Technology*, pp. 531–534, IEEE, mar 2012.

[60] I. Al-Anbagi, M. Erol-Kantarci, and H. T. Mouftah, "A Survey on Cross-Layer Quality-of-Service Approaches in WSNs for Delay and Reliability-Aware Applications," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 525–552, 2016.

[61] M. Sujeethnanda, N. Padmalaya, and G. Ramamurthy, "A Novel Approach to an Energy Aware Routing Protocol for Mobile WSN: QoS Provision," in *2012 International Conference on Advances in Computing and Communications*, pp. 38–41, IEEE, aug 2012.

[62] H. Egilmez and S. Dane, "OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks," *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 1–8, 2012.

[63] A. Ishimori, F. Farias, I. Furtado, E. Cerqueira, and A. Abelém, "Automatic QoS Management on OpenFlow Software-Defined Networks," *7th API Think-Tank - Software Defined Networking*, vol. 1, no. i, pp. 2–3, 2012.

[64] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "PolicyCop: An Autonomic QoS Policy Enforcement Framework for Software Defined Networks," in *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, pp. 1–7, IEEE, nov 2013.

[65] B.-Y. Ke, P.-L. Tien, and Y.-L. Hsiao, "Parallel prioritized flow scheduling for software defined data center network," in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, pp. 217–218, IEEE, jul 2013.

[66] R. Brown, *Exponential Smoothing for Predicting Demand.* Little, 1956.

[67] "OPNET Modeler," `http://www.riverbed.com`.

[68] Simulation Interoperability Standards Committee (SISC), "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules," *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pp. 1–38, 2010.

[69] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, IEEE (Comput. Soc.), 2004.

[70] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *2013 IEEE Conference on*

Computer Communications Workshops (INFOCOM WKSHPS), pp. 79–80, IEEE, apr 2013.

[71] A. J. Jara, S. Varakliotis, A. F. Skarmeta, and P. Kirstein, "Extending the Internet of things to the future Internet through IPv6 support," *Mobile Information Systems*, vol. 10, no. 1, pp. 3–17, 2014.

[72] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX," *ACM SIGCOMM Computer Communication Review*, vol. 38, p. 105, jul 2008.

[73] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 87–98, apr 2014.

[74] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: a protection architecture for enterprise networks," *15th USENIX Security Symposium*, pp. 137–151, 2006.

[75] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '07*, (New York, New York, USA), p. 1, ACM Press, 2007.

[76] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, p. 41, oct 2005.

[77] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," *2nd*

conference on Symposium on Networked Systems Design & Implementation, pp. 15–28, 2005.

[78] Project Floodlight, "Floodlight," 2017.

[79] D. Erickson, "The beacon openflow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*, (New York, New York, USA), p. 13, ACM Press, 2013.

[80] T. Feng, J. Bi, and H. Hu, "TUNOS: A novel SDN-oriented networking operating system," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pp. 1–2, IEEE, oct 2012.

[81] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, Others, and S. Shenker, "Onix: A Distributed Control Platform for Large-Scale Production Networks," *9th USENIX Conference on Operating Systems Design and Implementation*, pp. 1–6, 2010.

[82] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller architecture," in *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pp. 1–6, IEEE, jun 2014.

[83] A. Al-Fuqaha, A. Khreishah, M. Guizani, A. Rayes, and M. Mohammadi, "Toward better horizontal integration among IoT services," *IEEE Communications Magazine*, vol. 53, pp. 72–79, sep 2015.

[84] V. Cerf and M. Senges, "Taking the Internet to the Next Physical Level," *Computer*, vol. 49, pp. 80–86, feb 2016.

[85] M. Zorzi, A. Gluhak, S. Lange, and A. Bassi, “From today’s INTRAnet of things to a future INTERnet of things: A wireless- and mobility-related view,” *IEEE Wireless Communications*, vol. 17, no. 6, pp. 44–51, 2010.

[86] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, “Middleware for Internet of Things: A Survey,” *IEEE Internet of Things Journal*, vol. 3, pp. 70–95, feb 2016.

[87] S. Milardo, A. C. Anadiotis, L. Galluccio, G. Morabito, and S. Palazzo, “Offloading software defined WSNs through distributed geographic forwarding.”

[88] A.-C. G. Anadiotis, L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, “Towards a software-defined Network Operating System for the IoT,” in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 579–584, IEEE, dec 2015.

[89] A.-C. G. Anadiotis, G. Morabito, and S. Palazzo, “An SDN-Assisted Framework for Optimal Deployment of MapReduce Functions in WSNs,” *IEEE Transactions on Mobile Computing*, vol. 15, pp. 2165–2178, sep 2016.

[90] L. Mai, L. Rupprecht, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, “Supporting application-specific in-network processing in data centres,” in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM - SIGCOMM ’13*, (New York, New York, USA), p. 519, ACM Press, 2013.

[91] P. Costa, A. Donnelly, A. Rowstron, and G. O. Shea, “Camdoop : Exploiting In-network Aggregation for Big Data Applications,” *NSDI’12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 1–14, 2012.

[92] C. Buratti, A. Stajkic, G. Gardasevic, S. Milardo, M. D. Abrignani, S. Mijovic, G. Morabito, and R. Verdone, "Testing Protocols for the Internet of Things on the EuWIn Platform," *IEEE Internet of Things Journal*, vol. 3, pp. 124–133, feb 2016.

[93] L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, vol. 26, pp. 513–521, IEEE, apr 2015.

[94] M. Zorzi and R. Rao, "Geographic random forwarding (geraf) for ad hoc and sensor networks: energy and latency performance," *IEEE Transactions on Mobile Computing*, vol. 2, pp. 349–365, oct 2003.

[95] K. Chen and K. Nahrstedt, "Effective location-guided tree construction algorithms for small group multicast in MANET," in *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 1180–1189, IEEE, 2002.

[96] J. Bachrach and C. Taylor, "Localization in Sensor Networks," in *Handbook of Sensor Networks*, pp. 277–310, Hoboken, NJ, USA: John Wiley & Sons, Inc., sep 2005.

[97] G. Mao, B. Fidan, and B. D. Anderson, "Wireless sensor network localization techniques," *Computer Networks*, vol. 51, pp. 2529–2553, jul 2007.

[98] V. Daiya, J. Ebenezer, S. A. V. S. Murty, and B. Raj, "Experimental analysis of RSSI for distance and position estimation," in *2011 International*

Conference on Recent Trends in Information Technology (ICRTIT), pp. 1093–1098, IEEE, jun 2011.

[99] E. Elnahrawy, Xiaoyan Li, and R. Martin, "The limits of localization using signal strength: a comparative study," in *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, vol. 00, pp. 406–414, IEEE, 2004.

[100] G. Zanca, F. Zorzi, A. Zanella, and M. Zorzi, "Experimental comparison of RSSI-based localization algorithms for indoor wireless sensor networks," in *Proceedings of the workshop on Real-world wireless sensor networks - REALWSN '08*, (New York, New York, USA), p. 1, ACM Press, 2008.

[101] P. Biswas and Y. Ye, "Semidefinite programming for ad hoc wireless sensor network localization," in *Proceedings of the third international symposium on Information processing in sensor networks - IPSN'04*, (New York, New York, USA), p. 46, ACM Press, 2004.

[102] S. E. Dreyfus and R. A. Wagner, "The steiner problem in graphs," *Networks*, vol. 1, no. 3, pp. 195–207, 1971.

[103] D. M. Warme, "Spanning trees in hypergraphs with applications to steiner trees," 1998. AAI9840474.

[104] "Embit Data Sheet," `https://tinyurl.com/EmbitDatasheet`.

[105] R. Shah, A. Wolisz, and J. Rabaey, "On the performance of geographical routing in the presence of localization errors," in *IEEE International Conference on Communications, 2005. ICC 2005. 2005*, vol. 5, pp. 2979–2985, IEEE, 2005.

[106] "Dreyfus-Wagner Algorithm," `https://tinyurl.com/Dreyfus-Wagner`.

[107] D. Kreutz, F. M. V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, pp. 14–76, jan 2015.

[108] T. Akiyama, S. Murata, K. Tsuchiya, T. Yokoyama, M. Maggio, G. Ciulla, J. R. Santana, M. Zhao, J. B. D. Nascimento, and L. Gürgen, "FESTIVAL: Design and Implementation of Federated Interoperable Smart ICT Services Development and Testing Platform," *Journal of Information Processing*, vol. 25, pp. 278–287, 2017.

[109] P. V. Klaine, M. A. Imran, O. Onireti, and R. D. Souza, "A Survey of Machine Learning Techniques Applied to Self-Organizing Cellular Networks," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2392–2431, 2017.

[110] T. Abar, A. Ben Letaifa, and S. El Asmi, "Machine learning based QoE prediction in SDN networks," in *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 1395–1400, IEEE, jun 2017.

[111] S. Nanda, F. Zafari, C. DeCusatis, E. Wedaa, and B. Yang, "Predicting network attack patterns in SDN using machine learning approach," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 167–172, IEEE, nov 2016.

[112] A. Sabeeh, Y. Al-Dunainawi, M. F. Abbod, and H. S. Al-Raweshidy, "A hybrid intelligent approach for optimising software-defined networks perfor-

mance," in *2016 6th International Conference on Information Communication and Management (ICICM)*, pp. 47–51, IEEE, oct 2016.

[113] R. Alvizu, S. Troia, G. Maier, and A. Pattavina, "Matheuristic With Machine-Learning-Based Prediction for Software-Defined Mobile Metro-Core Networks," *Journal of Optical Communications and Networking*, vol. 9, p. D19, sep 2017.

[114] F. Dario Baptista, S. Rodrigues, and F. Morgado-Dias, "Performance comparison of ANN training algorithms for classification," in *2013 IEEE 8th International Symposium on Intelligent Signal Processing*, pp. 115–120, IEEE, sep 2013.

[115] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, pp. 2222–2232, oct 2017.

[116] "LSTM Applications," `http://people.idsia.ch/~juergen/rnn.html`.

[117] "FEderated interoperable SmarT ICT services deVelopment And testing pLatform," `http://www.festival-project.eu`.