UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

MSc degree in Computer Engineering

Implementation and performance evaluation of efficient algorithms for slicing in real SDN-based 6LoWPAN networks

Candidate

Tavoletta Simone

Supervisors

Prof. Mingozzi Enzo

Ing. Vallati Carlo

Graduation session of 28/09/2018

Thesis archive in electronic version: etd-09052018-134140

Academic Year 2017/2018

Table of contents

Introduction

During the last years, the implementation of computational and communication capabilities in all kind of objects has given traction to the diffusion of the Internet of Things (**IoT**) paradigm. Being sensor nodes the main building block of an IoT infrastructure, Wireless Sensor Networks (**WSN**) are expected to play an important role in allowing the interconnection between smart devices. Hence, to overcome the rapidly increasing[1] needs of this kind of networks, multiple solutions have been proposed in order to allow efficient management and scaling.

In the field of traditional wired networks, Software Defined Networking (**SDN**) represents an innovative technology for effective management and configuration, since it decouples the control plane from the data plane, thus migrating the control logic from nodes to a central controller, which can be easily managed in order to virtually modify the underlaying physical infrastructure. This approach could be extended to WSNs, so to alleviate all the scale-dependent network management challenges brought by the diffusion of the IoT archetype. The merging of these two concepts gives rise to a new paradigm, designated as Software Defined Wireless Sensor Networks (**SDWSN**).

In order to efficiently implement SDN over WSN, multiple issues must be considered, with particular attention to the fact that SDN is specifically designed for wired networks, hence it does not take into account the problematics that will arise in case of wireless communication, being the wireless medium by definition asynchronous and not perfectly reliable. Henceforth, multiple studies have been persevered in order to propose an efficient solution to the different challenges raised by the implementation of SDN over WSN, being one of them a thesis work achieved by a former student of the University of Pisa, Giulio Micheloni. As the majority of proposed solutions, the one presented by Micheloni is state of the art and does not provide an actual implementation of the SDWSN over a real working network. For this reason, the present work of thesis aims to bring those theoretical statements into practice and prove the truthfulness of results that were achieved through simulation.

After reaching this milestone, another objective of this work will be to show and prove the advantages of utilizing SDN over WSN with regards to a normal network architecture, in particular for what concerns the concepts of multi tenancy and network slicing. To perform this task, some algorithms will be constructed and utilized inside the controller in order to build and manage a virtual backbone for covering multicast groups inside the WSN.

Chapter 1

Prerequisites

Before going into details, this chapter aims to explain some core knowledge needed in order to better understand some aspects that will be analyzed later in this document.

1.1 Internet of Things

The evolution of electronical devices during the last years has given new opportunities to the implementation of smart systems for covering all possible aspects of the everyday life, spanning from smart fridges that allow the modification of different parameters, to smart grid systems for deciding where to enable or disable power provisioning inside a building, all through a simple user interface. This paradigm takes the name of **Internet of Things**[2] (IoT), describing the fact that all these objects are able to connect to the internet in order to exchange data with the external world.

1.2 Wireless Sensor Networks

Typically, in a local environment, all the smart objects composing an IoT infrastructure will be able to interconnect wirelessly and communicate between each other without the need of any internet access. As an example, one may think about a smart home application, in which the security system is directly communicating with a smoke detector inside the kitchen, which in turn is exchanging data with the oven. If the temperature inside the oven reaches a certain threshold, the smoke detector could trigger an alarm inside the security system even without sensing anything, but allowing the immediate detaching of the oven from its power source, thus possibly preventing a fire outbreak. Also, if needed, the security system could access the internet and send a message to a specific mobile phone.

From the above example, it can be easily noticed that both the smoke detector and the oven must have at least one sensor for collecting data from the external environment, a radio in order to communicate, some limited computational capabilities and a power source. More in detail, a device corresponding to this description is called **sensor node** and represents the core component of a **Wireless Sensor Network**[3] (WSN). In these infrastructures, multiple sensor nodes are *wirelessly* interconnected between each other in order to form a complex network, on top of which there is typically a special device, called the **sink node** (in our example, the security system's control unit), which is unconstrained and collects all the data received from the other nodes.

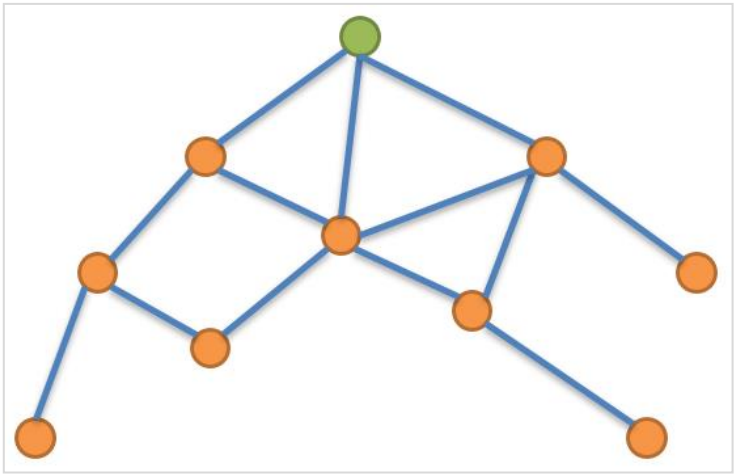


Fig. 1.1 – An example WSN infrastructure. The green node is the sink, while blue links are just representative of possible wireless reachability between nodes.

1.2.1 IEEE 802.15.4

To explicitly underline the fact that sensor nodes are constrained devices, this type of networks are also called **low-rate** Wireless Personal Area Networks (LR-WPAN), thus implying the need of a dedicated protocol to allow transmission of packets between multiple nodes without consuming too much resources. The IEEE 802.15.4 standard was hence developed specifically for managing the MAC and physical layers of networks with low power consumption and short-range communication. Among other things, this protocol defines multiple aspects regarding congestion avoidance and control of the wireless communication medium, by taking into account the constrained nature of sensor

nodes, hence providing efficient management of energy resources. Furthermore, it also considers the *low bandwidth* of wireless links, thus providing a buffering queue for packets to be stored and successively forwarded as soon as the communication medium is free and available (one at a time).

1.2.2 6LoWPAN

Another issue raised by the constrained nature of sensor nodes composing a WSN is represented by the fact that the standard TCP/IP network stack cannot be utilized as it is, this because it is considered to be too heavy for them to handle. This lack of an addressing standard brought over the years to multiple proposals, until the introduction of the IPv6 addressing protocol for Low-Rate Wireless Personal Area Networks (6LoWPAN).

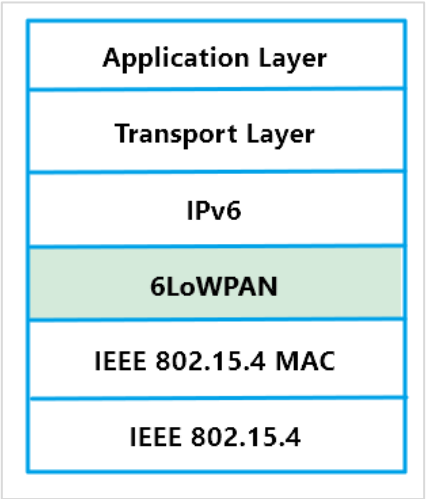


Fig. 1.2 – 6LoWPAN stack example

6LoWPAN (RFC 4944) represents an adaptation layer implemented in-between the Network and MAC layers, which are respectively IPv6 and IEEE 802.15.4. As explained, because of the multiple constraints that must be considered when dealing with sensor nodes, this new layer is needed in order to allow the transport of IPv6 packets over the low-rate IEEE 802.15.4 protocol. In particular, multiple issues need to be considered when implementing IPv6 on WSN:

- The Maximum Transmission Unit (MTU) supported by IEEE 802.15.4 is of 127 Bytes, while the *minimum* MTU needed at link-layer for IPv6 to work is of 1280 Bytes, hence **fragmentation** of packets will be needed in order to send data correctly;
- due to the impossibility of applying fragmentation directly in the network layer (because IPv6 doesn't natively support it for performance reasons), a mechanism of **packet adaptation** must be introduced;
- this procedure should not only allow the subdivision of packets into fragments, but more importantly the **compression of headers**, in order to save space that will be instead utilized for the real payolad to transport.

For these reasons, 6LoWPAN is needed as an intermediate layer to allow fragmentation and reassembly of packets (also by implementing header compression) in a completely transparent manner with regards to both IPv6 and IEEE 802.15.4. Moreover, a peculiar aspect of IPv6 is represented by the capability of devices utilizing this protocol to automatically assign to themselves a network address when joining the system, without the need of any external configuration (like the DHCP server in IPv4). This procedure takes the name of **stateless address auto configuration** and is also implemented in 6LoWPAN, allowing nodes joining the network to automatically assign themselves an IPv6 address (with global unicast scope) by appending the Interface ID (IID) taken from the node's MAC address to the network's prefix (received by a Router Advertisement message, see next chapters for a better understanding of these concepts). This means that the physical address of a node will be easily extracted from the IP one, also allowing a better header compression.

1.2.3 RPL

The last basic concept that must be grasped for what regards Wireless Sensor Networks is represented by how exactly nodes inside the system are able to organize themselves in a topology that is used for forwarding data to the sink node. In fact, to avoid too much power consumption, sensor nodes mount low power radios, meaning that their communication range will not allow to always directly reach the sink node, wherever it

is placed. For this reason, a **multi-hop** technology is implemented, which allows the transmission of messages from one node to another by passing through middle ones, called in this case *relay nodes*.

In order to organize a 6LoWPAN network in such a way to allow the reachability of the sink node from *any* other node inside the system (and vice versa), a solution must be implemented. This collides with another hidden issue brought by IEEE 802.15.4, which is the impossibility of performing multi-hop transmissions. For this reason, two possible solutions have been proposed in the 6LoWPAN architecture[4]:

- **Mesh-under:** this fix involves the implementation at the layer 2 of four fields inside the packet's header in order to identify the original transmitter and the final destination, together with the next hop that must be traversed in order to reach it. Unfortunately, this method presents multiple issues related to the assumptions that are made by the IPv6 model for what regards the association between subnets and links, together with poor interoperability between layer 2 and 3;
- **Route-over:** this solution is instead implemented at the layer 3, where it is directly possible to manage a routing protocol that applies different assumptions with regards to the ones presented by IPv6 and that now can work with *multi-link* subnets without causing any conflict.

If using a route-over configuration, 6LoWPAN can implement a routing protocol for low power and lossy networks, known as **RPL**. This methodology is based on a distance vector algorithm that looks for the best paths between nodes by taking multiple constraints into account. Based on the (not completely true) idea that the majority of packets will traverse the network in order to go inside the sink and eventually exit the system, RPL organizes the nodes in such a way to form a tree topology where the root is indeed represented by the sink and each node selects a *preferred* parent to which it will forward packets. It is important to underline that this topology is only logical and does not take into account the real physical distribution of nodes, hence considering only the optimization of a pre-defined *Objective Function* (OF) with several constraints. For this reason, **multiple instances** of RPL can be built on top of the same network, thus allowing the implementation of different OFs (that in turn depend on the type of application to be served).

The topology in which nodes are organized to forward messages is here represented by a Destination Oriented (because it wants *only one* sink / root node) Directed Acyclic Graph (**DODAG**). To build such graph and successively perform maintenance, the concept of **node rank** is introduced and utilized: each node has an integer value that determines its rank in the topology, with the root having rank *zero*. Consequently, a node that is children of another one in the DODAG will need to have a rank that is bigger than its parent, thus allowing loop avoidance and detection. Additionally, because of the logical nature of a DODAG, it will be possible to build several of them in the *same* RPL instance, one for each sink. A node belonging to one DODAG in a RPL instance *cannot* belong to another one in the same instance.

A DODAG is initialized by means of a message broadcasted from its root, called DODAG Information Object (**DIO**), in which are stored multiple configuration parameters that will be utilized to build the complete topology, together with the rank, in this case equal to zero. After a node receives a DIO message, it updates the rank information (if it joins the DODAG) and forwards the packet to other nodes inside the network, also selecting (based on the OF) as parent one among all the nodes from which it received a DIO (of course, that node must have a smaller rank value in order to be chosen). When the DODAG is formed, nodes belonging to it will be able to reach the sink by just forwarding the packet to their parents.

Being RPL specifically designed for multipoint-to-point communication (MP2P), if the sink (or any other) node wants to communicate with one or multiple nodes inside the network, the support for point-to-point (P2P) and point-to-multipoint (P2MP) communication will be necessary. Accordingly, to allow downward routes, the RPL protocol specifies also a Destination Advertisement Object (**DAO**) message, which carries information about the node(s) to be reached. Two possible routing modes are supported:

- **Storing mode:** assuming nodes have enough memory to store a routing table, the DAO message will be forwarded from the sending node to its parent and so on, until it reaches one device that knows the destination by looking inside its routing table. In the worst case, it will be needed to first reach the sink node;

- **Non-storing mode:** no routing table is stored by any node, except for the sink. If a sensor node wants to send a message to another device inside the network, it will need to first send it to the root and then *source routing* (complete route stored inside the DAO) will be utilized in order to reach the correct destination.

1.3 Web of Things

It is easily understandable that, on top of the basic communication layer provided by WSN for IoT architectures, a number of applications with different APIs will be installed so to allow nodes to connect to the internet and provide or utilize services. This application layer is here represented by the **web of things**, describing the fact that it allows real-world objects to join the World Wide Web, by taking into account their constrained nature.

Instead of inventing new concepts to allow communication of distributed applications inside a sensor network, the concept of *resource*, introduced by the Representation State Transfer (**REST**) software architecture for the standard internet, is considered. In particular, this paradigm identifies a set of principles that should be utilized when developing a distributed application, which will not provide services, but indeed resources, that will be accessed by means of HTTP (HyperText Transfer Protocol) standard operations (PUT, GET, POST and DELETE). The problem with a RESTful API is that an application providing it will by definition utilize HTTP for communication, and this is not in line with a constrained sensor node. More in detail, since the HTTP protocol was conceived for much powerful links (and devices), it runs over TCP (Transmission Control Protocol), a transport layer protocol that provides reliable communication by establishing a connection between two ends. This means that a certain number of messages between nodes will be needed even before transmitting the data of interest, hence causing high overhead inside the WSN and thus being inappropriate for such systems.

1.3.1 CoAP

For this reason, a new web protocol able to run on top of constrained devices without deteriorating their capacities has been defined, named **Constrained Application**

Protocol (CoAP). CoAP represents a generic web protocol based on HTTP, but utilizing the UDP transport protocol instead of TCP, this because typically reliable communication is not a requirement for most of the applications running in the Web of Things. By not utilizing the transport layer for reliability, this new protocol reduces significantly the overhead of messages previously caused by HTTP. Furthermore, it is also able to provide *low header overhead* and *parsing complexity* thanks to the use of binary encoding on resource headers instead of using ASCII strings (like it was done in HTTP, for readability and debugging).

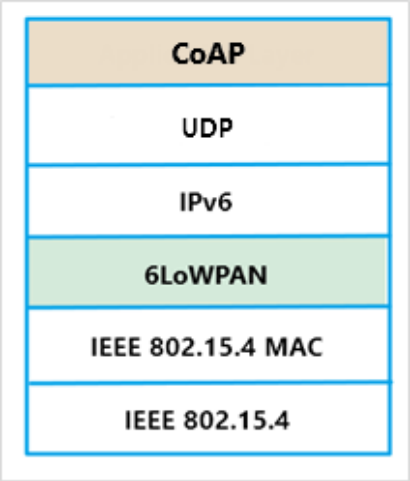


Fig. 1.3 – 6LoWPAN stack updated with CoAP

Without describing the full standard, which can be found in the RFC 7252 specification, some important features must be outlined to support the basic knowledge needed for understanding next chapters:

- **Client-Server architecture:** as in HTTP, CoAP allows for a client-server communication, in which the client is the one asking for services and the server the one providing them. Typically, a sensor node will take the role of both client and server.

- **Stateless HTTP mapping:** this protocol allows for intercommunication between application supporting HTTP and the ones utilizing CoAP, thus allowing easy translation of messages coming from outside the sensor network (and vice versa).

- **Support for multicast communication:** this standard allows the exchange of messages from one client to multiple servers (called *multicast group*, defined in the Internet Protocol).

- **Support for discovery and observing of resources:** each server will provide a number of resources to other clients, thus it will be possible to discover them by utilizing a well-known relative URI ("/.well-known/core") in which there will be a list of all available ones. A client can then *observe* one or multiple resources of a server, meaning it will receive an update message periodically, when the value changes or when it goes above a threshold.

- **Support for reliable transmission:** as explained, CoAP utilizes UDP instead of TCP, hence renouncing to transport-layer reliability. In order to allow some sort of reliable transfer for critical applications, a mechanism is implemented inside the protocol itself. This means that, when communicating, two nodes will need to specify inside the CoAP message if an acknowledgement (ACK) should be sent for confirming the correct reception of packets. Two possibilities are implemented:
    - *Confirmable* messages (**CON**), which specify that the node receiving such packet **must** answer with an ACK;
    - *Unconfirmable* messages (**NON**), which specify that the node receiving such packet **must not** answer with an ACK, even if it could.

  Those packets will be matched by the same *Message ID*, which allows the detection of duplicates, and by a *token* to match responses to requests (because there is no connection, since CoAP is using UDP).

1.4 Networking challenges

As already explained, sensor nodes represent the main building block of an IoT infrastructure, meaning that the WSN technology must indeed be utilized in order to manage the interconnection between smart objects. During the last years, the use of IoT

architectures has become more and more popular in a very large amount of applications, thus allowing a fast growth in scale of sensor networks, where the "things" are now also producing traffic other than just acting as consumers. Because of this significant number of interconnected devices, the administration of IoT networks is experiencing an increasing amount of problems that go from difficult control of devices to overloaded systems[5].

On the other hand, traditional WSN technologies are not prepared for such dense networks, thus risking that this flood caused by the IoT will bring to a worldwide paralysis. Moreover, wireless sensor networks are typically built for application-specific purposes and are hence not able to support the diversity of appliances that want to communicate with a single IoT infrastructure. For this reason, multiple solutions are nowadays being proposed in order to efficiently maintain the interconnection between a large number of nodes belonging to the same system and allow a more flexible control over those devices.

Chapter 2

Software Defined Networking

With the rapid improvement of information and communication technology, an imponent amount of traffic is nowadays being generated in all sectors, thus demanding ubiquitous accessibility, high bandwidth, and dynamic management of network infrastructures. Being traditional approaches (based on manual configuration of devices) cumbersome and very easily subject to errors, new systems are being proposed in order to better utilize the capabilities of physical network infrastructures. Among all proposed solutions, **Software Defined Networking**[6] (SDN) stands out as one of the most promising approaches for solving this problem.

2.1 Architecture

The most important idea behind SDN is to separate the control layer from the data plane, thus allowing to easily modify the network infrastructure by just managing a central device, called **SDN Controller**, which will in turn change the behavior of network switches, in this case called **SDN Switches** to emphasize their lack of control logic. By utilizing this approach, service providers are thus able to virtually adjust the physical network infrastructure in order to provide better performance to each customer, enabling the possibility of implementing traffic engineering (TE) and quality of service (QoS) techniques in a more immediate and cost-effective way.

In order to perform those operations, an Application Programming Interface (API) must be provided between the SDN Controller and Switches for communicating without the need of complete code exposure. This is called **Southbound Interface** (SI) and allows the SDN Switches to receive instructions on how to forward the traffic that is flowing inside the network. A second interface, called **Northbound Interface** (NI), is instead provided by the controller in order to allow management of the network from the application layer. Furthermore, the SDN Controller will typically be implemented in a

distributed fashion, allowing better scalability of the entire system. In this case, two more interfaces will be needed for communication between different controllers, namely the **Eastbound Interface** (EI) and the **Westbound Interface** (WI).

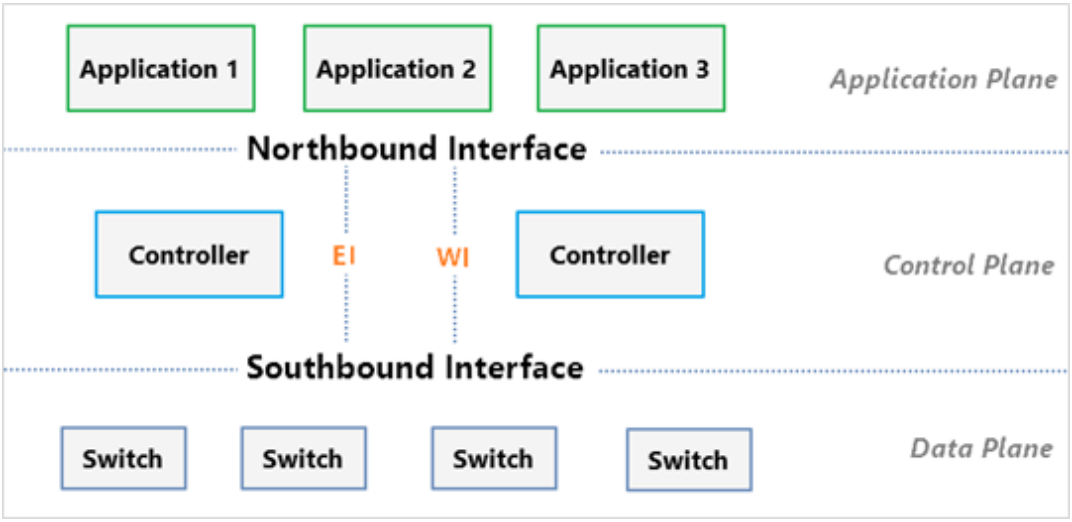


Fig. 2.1 – SDN architecture and interfaces

2.2 Communication protocol

As outlined, decoupling the control logic from the data plane represents the main concept of SDN and, for this reason, it is evident that an infrastructure implementing such paradigm will be mainly characterized by how switches are able to exchange information regarding the network with the controller, thus implying the need of a standardized protocol for defining the southbound API. Although none of the proposed protocols has been selected as an official standard, **OpenFlow**, supported by the Open Networking Foundation[7], represents the first and probably the most known southbound interface utilized in SDN architectures and for this reason it can be seen as a de-facto standard for software defined networking. Without going into advanced technical details, an overview of the OpenFlow protocol is hereby presented in order to build a basic knowledge that will be necessary to better understand some design choices presented in the next chapters.

2.2.1 OpenFlow: an overview

The OpenFlow[8] communication protocol takes its name from the fact that all traffic *flowing* inside the network is indeed abstracted as *flows*, and the switches implement one or more **flow tables** in which there will be multiple entries installed by the controller. When a packet belonging to a specific flow enters an OpenFlow Switch, the device will check if there is at least one **entry** inside the tables that matches the current data and, if there isn't any, it will ask to the SDN Controller for instructions on how to treat the packet. If instead an entry with **rules** matching the flow is found, the switch will apply to the packet a set of **actions** defined inside that entry.

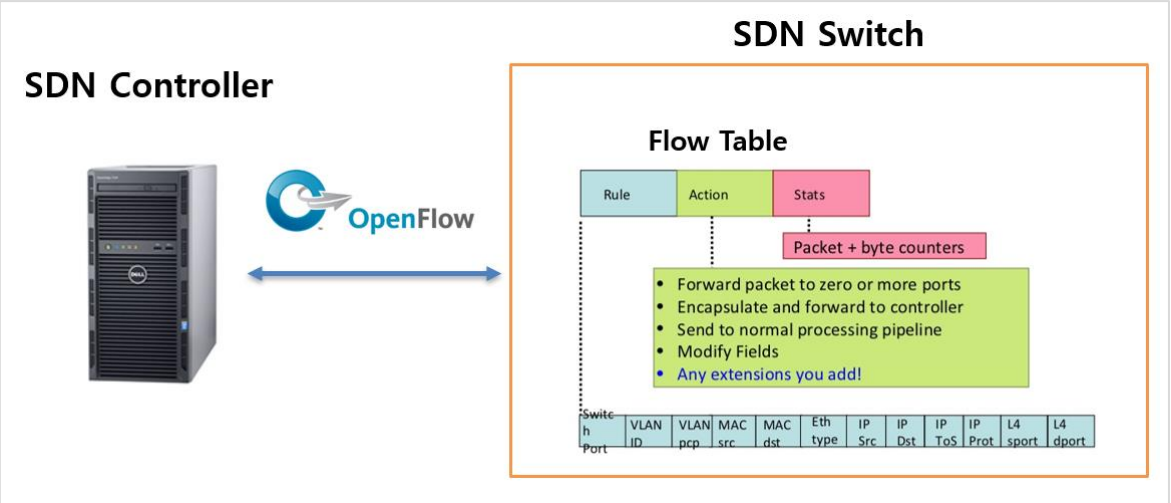


Fig. 2.2 – OpenFlow interface

Summing up, a switch supporting the OpenFlow API will contain one or more *flow tables*, in which there will be multiple *entries* installed by the SDN controller through specific messages defined by the protocol. An entry will be composed of three fields:

- a field containing one or more *rules* in order to match packets belonging to certain flows. These rules look at various information taken mostly from the packet's header and must be satisfied for the entry to match the packet;
- a set of *actions* to apply, selected from:
    - *Forward*: route the packet through one or more ports.

  - *Drop*: delete the packet and do not take any more actions.
  - *Enqueue*: put the packet into a queue, for performing QoS.
  - *Modify*: change the value of different header fields.
  - Other custom actions can be defined.
- a field for collecting some useful *statistics.*

If no entry is found, the switch will need to ask for instructions to the controller. This involves the necessity of sending a copy of the entire packet to the SDN Controller, which, after processing it, can act in three different ways:

- Answer to the switch with a *temporary set of instructions* on how to treat the packet, only for this time;
- *directly apply modifications* to the packet and send it back to the switch;
- answer with a *set of instructions* that must be installed and kept inside the flow table.

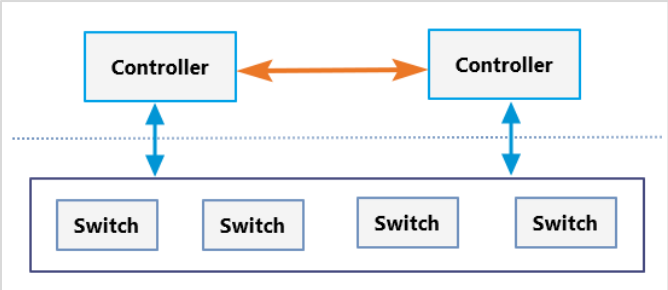
Another aspect that must be considered regarding this protocol is how a flow table is maintained inside the switch. In particular, it is easy to understand that multiple entries matching the same packet should typically be executed in a specific order, so to avoid unexpected issues. For this reason, a **priority** value is assigned to each of the entries maintained inside the table, thus ensuring a correct sequential execution of each instruction. Furthermore, it could happen that a packet matching a specific entry travels the network just one time, thus leaving the flow table with an occupied space that should be removed. In order to avoid this to happen, and to prevent the flow table from filling up completely (which could result in packet drops because of no matching entry and no space to install one), two **timeouts** (*hard* and *idle*, the former for evicting an entry after the set amount of time, the latter for removing the entry if no packets are matched in the given amount of time) are assigned to each entry and a cleanup procedure is performed autonomously by the switch.

2.3 Controller positioning

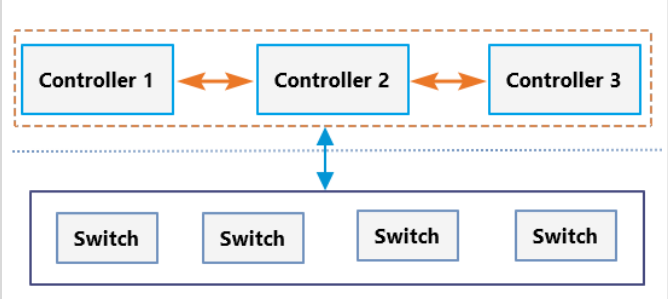
A crucial aspect that should also be considered while building an SDN architecture is the controller placement, this because of issues related to security and performance reasons.

More in detail, in the traditional SDN architecture there will be a central device, with global knowledge of the network, that acts as a controller and therefore represents a single point of failure, meaning that a distributed denial of service attack (DDoS) to this machine would be able to compromise the entire system. Moreover, as the network grows, a centralized controller could lead to slower response times and congestions, caused by a higher computational load due to multiple requests coming from the switches, thus not allowing the infrastructure to easily scale with the underlying physical architecture. For allowing a better subdivision of the network and to increase the overall performances, other possible approaches have been proposed[9], namely:

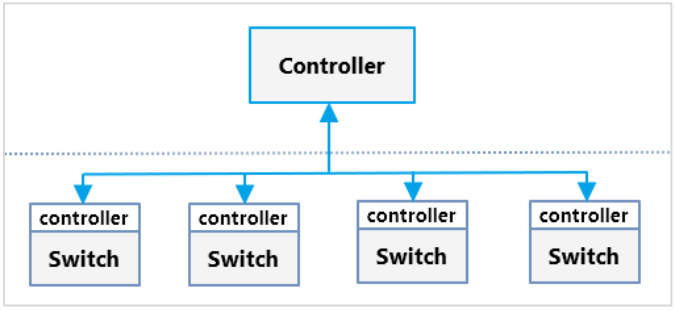
- **Distributed controller:** multiple machines are deployed in order to perform the role of controllers over different subsets of the network, thus allowing faster processing and a better scalability of the infrastructure. The peculiar downside of this method is that it implies the need of synchronization messages between controllers and an escalation of maintenance costs due to increased system complexity;



- **Physically distributed, logically centralized controller:** in this case, the controller is virtually deployed on multiple devices, so that the software runs in a faster way and has quicker response times, with all of them forming together just one global controller instead of multiple independent ones. Synchronization messages will again be needed, but the overall costs of maintaining the network will be lower than the physically distributed case;



- **Data plane extension:** the SDN Switches are modified so to implement a limited local control logic, thus allowing better performances, reduced overhead of messages and improving the overall system's security. This approach still depends on a central controller with global knowledge, which will represent again a single point of failure.



As it can be noticed, because of the balanced number of advantages and disadvantages regarding the proposals for solving the problem of controller placement, there is basically no optimal solution, but one must be chosen based on the specific network's needs.

2.4 Network virtualization in SDN

The concept of network virtualization represents the ability of building multiple virtually isolated subsets of the same network, called **slices**, which are able to seamlessly work on top of a shared physical architecture. Typically, this concept is utilized in cloud computing in order to provide multiple clients, called **tenants**, with different services that indeed use the same underlying physical infrastructure, but in an isolated manner. In order to be able to virtualize the network resources and share them among multiple tenants (defining the concept of **multi-tenancy**), a virtualization-ready approach must be utilized for managing the overall architecture, meaning that every resource inside each network device will need to be properly abstracted.

Unsurprisingly, Software Defined Networking represents a natural platform for implementing network virtualization[10], this because of the implicit capacity (provided by the separation of the control plane from the data plane) to easily adjust the network's configuration and resources by just modifying the forwarding rules installed inside the SDN Switches. In this way, it will be possible to give each tenant the illusion of having its own dedicated topology, controller and address space.

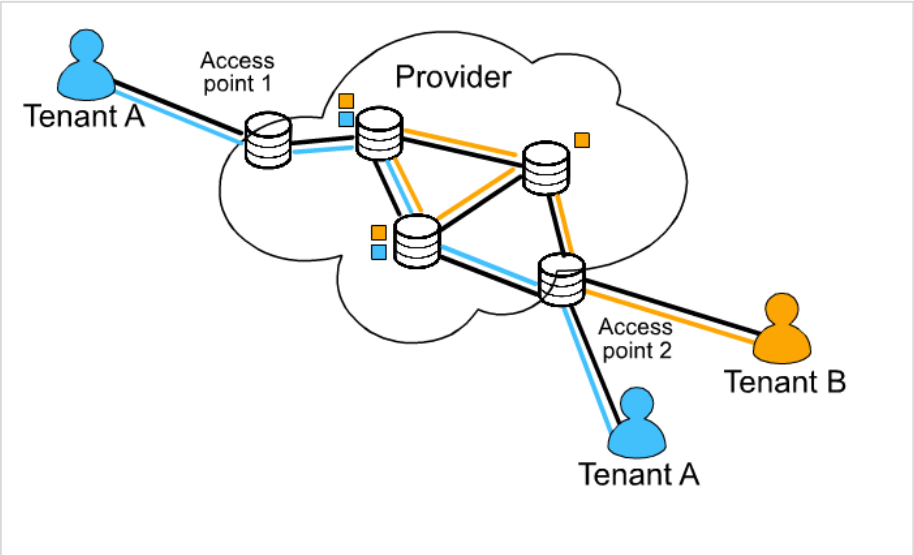


Fig.2.3 – Example of resource sharing in a virtualized network, black links represent physical ones, while colored links represent the ones abstracted for each tenant.

To better understand how this works, a possible example is represented by the *FlowN* solution for cloud computing proposed in [10], where each tenant is able to run its own controller application on top of a virtualization layer (typically called **hypervisor**), that in turn acts on the real SDN Controller in order to apply client-specific rules inside the assigned topology, thus implying that multiple clients will be able to manage their own virtual slice of the network by means of a custom controller defined by themselves. It is needless to say that the principal role of the hypervisor will be to check for permissions of each tenant before applying the requested updates to the shared SDN architecture.

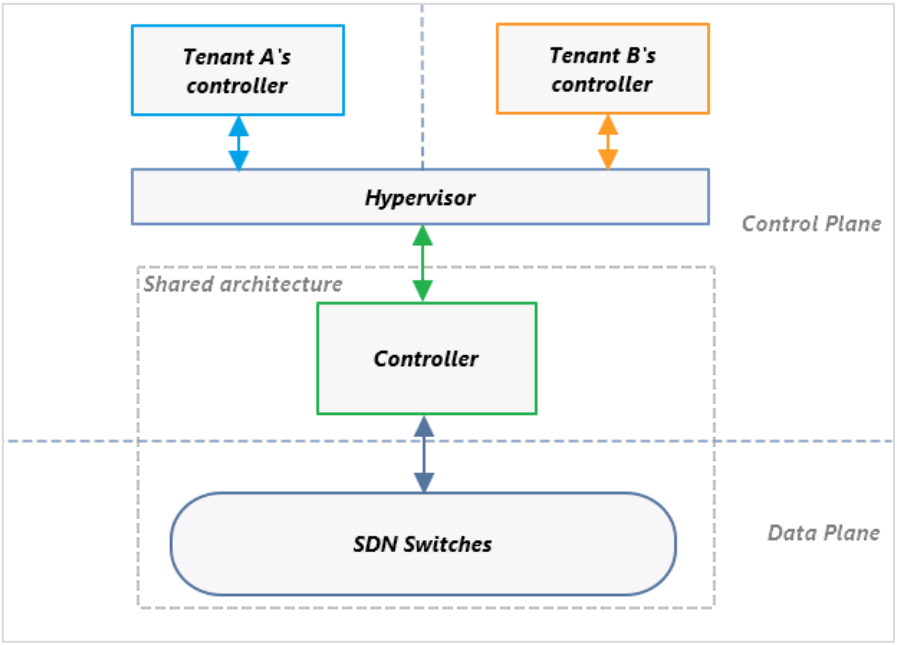


Fig. 2.4 – Typical multi-tenant SDN architecture

In conclusion, the described built-in capacity of SDN to perform network virtualization gives another reason for utilizing this paradigm to upgrade traditional infrastructures.

Chapter 3

Software Defined Wireless Sensor Networks

As explained in section 1.3, the principal challenge represented by the rapid diffusion of IoT networks consists in being able to efficiently manage a *large number* of interconnected nodes and be able to support the needs presented by *multiple clients*. Thanks to the previously provided knowledge, it is now possible to understand that these issues could be very likely solved by the implementation of Software Defined Networking over Wireless Sensor Networks, giving birth to a new paradigm that takes the name of Software Defined Wireless Sensor Networks (SDWSN).

3.1 The benefits of SDN over WSN

Other than providing a natural approach to **multi-tenancy** and **flexible network management**, SDN represents a very promising paradigm to utilize in WSNs because of the **similarity in architecture** between the two type of networks. It is indeed true that, while in a software defined network there will typically be a central controller with global knowledge to which switches connect in order to receive instructions on how to manage the traffic flowing inside the system, in a wireless sensor network, nodes will normally forward the data to a central sink, that is unconstrained and communicates with external applications.

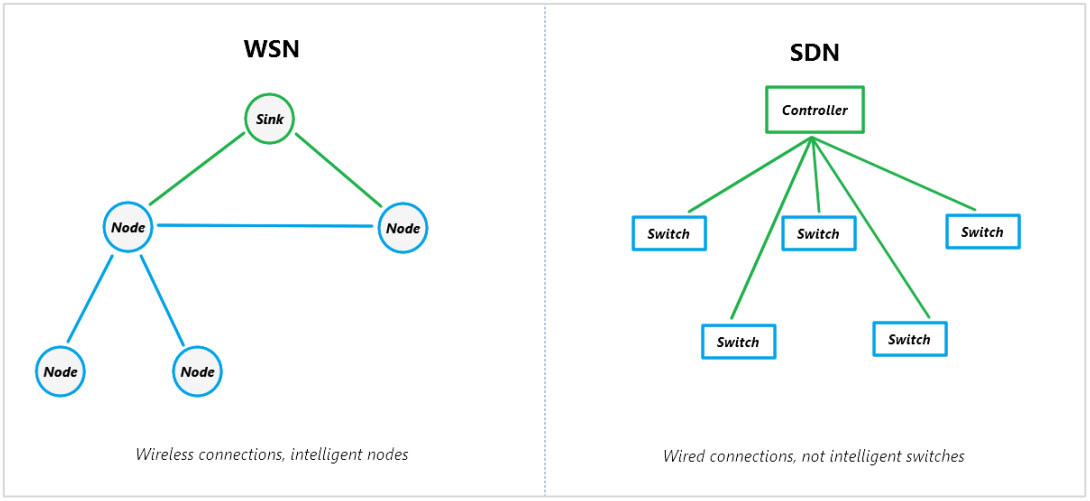


Fig. 3.1 – Graphical comparison between WSN and SDN architectures

Another favorable aspect of utilizing SDN over WSN is also represented by the fact that sensor nodes, as already explained in 1.2, have limited computational capabilities and energy storage, thus moving the data processing and decision-making to a central controller will inherently **lower the energy consumption** at the node level, increasing the overall network lifetime. This gives also the opportunity to facilitate the introduction of **new capabilities** inside the system, such as the ability of performing *Quality of Service*, *Traffic Engineering*, *network slicing* and *load balancing*. Additionally, the use of a standardize network infrastructure will provide **better interoperability** between multiple systems composing an IoT architecture.

3.2 Challenges of SDWSN

Since SDN represents, as described, a very promising approach to the resolution of almost all challenges brought by the diffusion of the Internet of Things, one may ask to himself why this is not already implemented and fully operational inside all kind of wireless networks. The main reason behind this lack of implementation is represented by the very concept of **wireless** sensor networks, which implies a communication channel that is not as reliable as a wired one and presents multiple ambiguities[9][11].

3.2.1 Unreliable links

More in detail, a wireless communication medium is by definition **asynchronous** and for that reason it does not ensure the mutual connection between two nodes inside the network. This means that a node A that thinks to be able to reach a node B, will not always successfully deliver a message to it. For this reason, a re-transmission procedure is implemented inside WSNs and should be consequently supported by SDN, which is instead based on the assumption that messages between controller and switches are always successfully delivered. Furthermore, an SDN architecture will need constant updates from the nodes in order to build a global knowledge of the network topology and subsequently take decisions, thus meaning that **unreliable** links will cause the system to not work properly due to the loss of some update packets. In addition to that, a wireless medium is also always exposed to multiple **interferences** that will not allow the correct delivery of messages, hence causing again problems in communication between controller and nodes.

In order to avoid battery consumption and to perform efficient power management, sensor nodes composing a WSN will also typically keep the radio in standby for long periods of time, activating it only for transmission or synchronization purposes. This concept takes the name of **duty cycling** and represents another challenge to the implementation of SDN over WSN, principally because it will compromise the already unreliable connectivity between nodes. Furthermore, because of the **multi-hop** technology described in 1.2.3, when sending a message a large number of hops to traverse will result in an higher unreliability of the connection, thus increasing one more time the chance of control packets being dropped along the way.

3.2.2 Network restrictions

The communication protocol utilized in wireless sensor networks is represented by the IEE 802.15.4 standard and, as described in 1.2.2, allows a Maximum Transmission Unit of 127 Bytes, meaning that packets flowing inside the system must be fragmented if above this size. In normal SDN implementations, control messages exchanged between

controller and switches will for sure be above that limit, thus implying that there will be the need of transmitting multiple packets for a single request from a node to the controller. Considering the limited storage capabilities of sensor nodes, it is easily predictable that there will often be no information inside the node on how to treat a packet, hence leading to a possible **network congestion** due to **fragmentation** and consequent multiplication of control messages flowing inside the system, also causing the battery to deplete faster.

3.2.3 The need of a new Southbound API

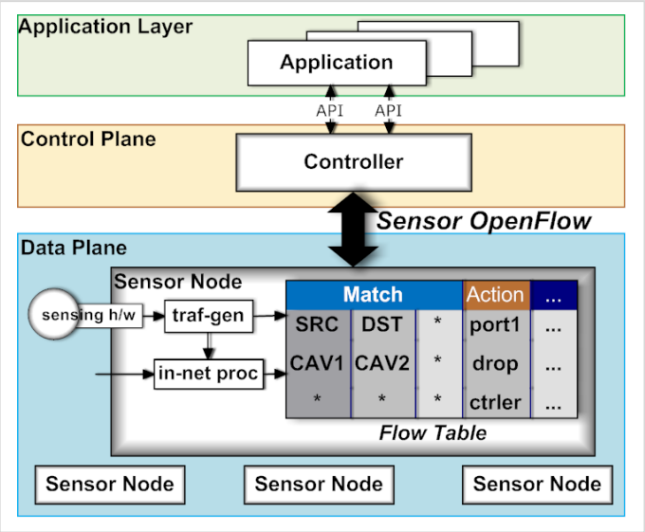
Since the de-facto standard utilized by SDN is represented by OpenFlow, it could be reasonable to think that building a SDWSN will be possible thanks to the implementation of the OpenFlow API on sensor nodes, thus creating a plug-and-play type of system. Unfortunately, being WSNs specifically designed with data collection and analysis in mind, without emphasis on *who* sent them (hence, with a **data-centric** approach instead of the address-centric one typical in Ethernet networks), there won't be any concept of **flow** definable.

Moreover, the OpenFlow standard supposes the presence of a direct and **dedicated** communication link between the controller and each node in the network, thus avoiding the need of taking into account limits in bandwidth when exchanging control messages. This concept is completely obliterated in WSN architectures, since here the same wireless link will serve both for *data* and *control* messages exchange.

It is hence possible to say that all the issues listed here and in above sections represent an enormous challenge to the implementation of SDN over WSN, mainly due to the implicit need of defining a completely new interface for communication between the SDN Controller and nodes. This could of course be done by utilizing some of the main concepts behind OpenFlow, but innovative solutions for what regards *unreliable links*, *fragmentation* and a proper definition of *flow* will be necessary.

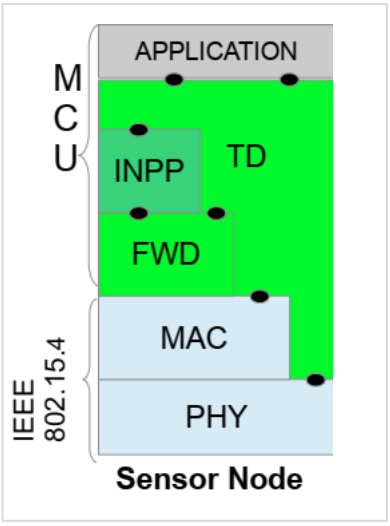
Among all proposed solutions, two relevant ones are briefly described:

- **Sensor OpenFlow**[12]**:** communication protocol based on the exploitation of the



OpenFlow Extensible Matches (OXM) field in order to provide two new matching classes for building a new concept of flow. The *Compact Network-Unique Addresses* class aims to match network addresses that are typical in WSNs, while the *Concatenated Attribute-Value pairs* class allows conditions regarding the packet's payload (such as for example a selection of a range of values for an attribute). Moreover, to avoid network congestion caused by message overhead, a sensor node will be expected to ask for instructions whenever there is a table miss and *not send* any other request for the same destination address until a response from the controller is received.

- **SDN-WISE**[13]**:** acronym of *SDN* solution for *WIreless* and *SEnsor* networks,



SDN-WISE represents the most complete implementation of SDN over WSN proposed as of today. By entirely redefining the traditional network stack, this technology aims to reduce the amount of overhead caused by control messages and make sensor nodes programmable (stateful), thus allowing them to take local decisions. Focusing on sensor nodes, the proposed stack involves a forwarding layer (**FWD**) that handles the addressing of packets based on the rules contained in the *WISE Flow Table* and an In-Network Packet Processing layer (**INPP**) that is responsible for performing data aggregation and other possible in-network processing of packets. In addition, a Topology Discovery (**TD**) layer has

the ability of cross-talking between INPP, FWD and MAC, thus allowing to perform local control functions inside the sensor node (such as selecting the next hop for reaching the sink) and maintaining the *WISE Neighbor List*. Another important aspect that should be considered is the proposed Topology Discovery Protocol, defined in order to make nodes able to reach the controller and to know the status of their neighbors (e.g. battery life, RSSI and so on) by periodically exchanging TD packets and successively updating the neighbor list (so to take better local decisions). Unfortunately, the biggest downside of SDN-WISE is represented by the use of a proprietary Southbound Interface, hence not compliant with IoT standards.

As for all mentioned concepts, those ideas will be kept in mind when presenting the proposed solution for SDN over WSN in this work of thesis.

3.2.4 Controller positioning

As illustrated in 2.3, controller placement is of crucial importance when trying to emarginate security issues and to increase the network scalability. Consequently, it should be expected that this kind of issue must be examined also in the case of SDWSN, being this indeed a proposed solution for large scale networks caused by the diffusion of the IoT paradigm.

Below are listed again the three possible positionings of the controller, but this time the two concepts of *power management* and *network congestion* are taken into account:

- **Physically distributed controller:** considering how a traditional WSN is structured, multiple sinks could be deployed inside the network, thus allowing for a completely distributed approach thanks to multiple controllers. As a good aspect, this will imply faster response times and easier reachability of the controller for each subset of the network, but synchronization messages will cause a large amount of *control* packets to flow inside the system, thus reducing the available space of bandwidth that can be utilized by the ones carrying important data;

- **Logically centralized, physically distributed controller:** another idea could be to place the controller over multiple nodes, not compulsory selecting the sink ones, but this will again cause possible congestions and increased energy consumption;

- **Data plane extension:** a very smart idea could be to, instead of placing the controller only on a sink node (which is contacted by the nodes also for sending data to it, through the same link), place a small amount of control capacity also inside each one of the nodes composing the network. Being now able to take simple decisions immediately, nodes will hence need to send less messages, reducing the probability of congestion and the total radio activity.

Chapter 4

The proposed solution

During the last years, there has been a plethora of studies involving the proposal of new solutions for the implementation of SDN over WSN, with most of them being supported by only theoretical studies and no implementation on a real network. This is also the case of the solution proposed by a former student of the University of Pisa, Giulio Micheloni[14], in his thesis work, where the fundamental bases of the system utilized in the hereby document were defined. Before examining what novelty has been brought by the hereby thesis, it is therefore necessary to give a brief explanation of the state-of-the-art system that stands at its basis.

4.1 System Architecture

By utilizing all the presented standards implemented in WSN, namely 6LoWPAN, RPL and CoAP, the proposed system successfully integrates the SDN approach to the IoT paradigm, defining three main types of devices:

- **SDN Sensor Node:** this is a basic sensor node in which an additional layer is implemented in order to provide the network programmability typical of SDN. The RPL protocol is utilized only to allow each node to reach the central SDN Controller, which is placed outside of the network and directly communicating with the sink. For allowing a better management of network resources, the *data plane extension* solution is considered, thus providing the SDN Sensor Node with the ability to take local decisions and implicitly reducing control message overhead.
- **SDN Sink Node:** basically the network's sink node, which allows communication between the local system and the external world, enhanced with the same SDN

layer implemented in normal nodes, thus allowing complete programmability of the entire system.

- **SDN Controller:** remote application that is completely external to the WSN and able to directly communicate with the sink and other nodes through CoAP messages.

4.1.1 A new network stack

Focusing on sensor nodes and taking into account all issues described in previous chapters, the traditional 6LoWPAN stack is improved by inserting a new layer (SDN) between the *adaptation* and *MAC* ones, which takes care of packet forwarding based on instructions provided by the controller and installed inside a *Flow Table*.

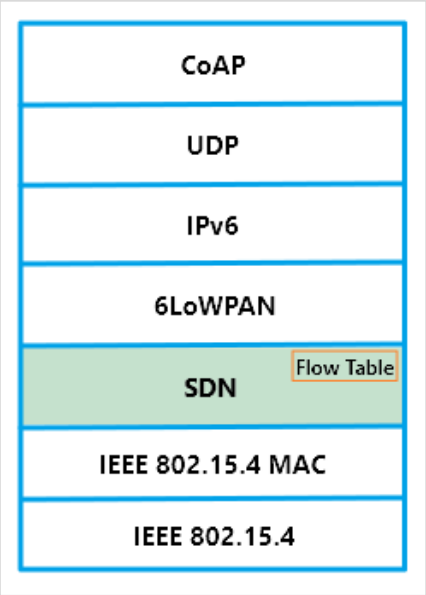


Fig. 4.1 – The enhanced network stack

This new layer will intercept packets traversing it and perform a lookup inside the flow table, searching for available entries that match the current data and utilizing the *mesh-under* approach to allow multi-hop communication (hence performing layer 2 forwarding). Since the RPL standard is instead based on a *route-over* approach, to avoid collisions (and possible loops) this SDN layer will bypass all packets addressed to the node itself or belonging to the routing protocol. Another useful aspect of this architecture

is represented by a software module inserted inside each sensor node, called *Control Agent*, which is able to install inside the flow table an upstream rule obtained by RPL and periodically gather information about neighbors (like it was done in SDN-WISE) so to take local decisions and directly add new entries to the table, without asking anything to the controller. Moreover, it will periodically report the collected information to the SDN Controller by means of a so-called *Topology Update* message, hence allowing the controller to maintain an updated global knowledge of the entire network.

4.1.2 The Flow Table

As in every SDN implementation, some sort of instruction table is needed inside switches (here, nodes) in order to allow dynamic network configuration through the separation between control and data plane. Since the presented system is (as every other solution) influenced by the OpenFlow protocol, a definition of *flow* is inherently needed and some rules must be defined in order to correctly identify packets. Taking into account memory constraints, the flow table is here defined by a data structure in which the maximum number of rows will be pre-defined and fixed beforehand, instead of being dynamically allocated.

Each row represents a **flow entry** and is composed of the same three fields already described in 2.2.1, namely *rules*, *actions* and *statistics*. Moreover, a TTL (Time To Live) value will be stored inside the statistics field to enable hard timeouts and an additional *priority* field is added so to manage the correct ordering of rules inside the table (the lower this number, the higher the priority, with minimum value being *zero*). For what regards *flow* definition, it is simply achieved by checking for multiple fields inside the packet. To perform this task, each **rule** inside an entry will be structured as follows:

- **Field:** the field of the packet to take as operand for comparison;
- **Offset:** index of the first bit from which the comparison will start;
- **Size:** number of bits that need to be compared;
- **Operator:** a mathematical operator used for comparison (as for example > and <);

- **Value:** the value to which this operand must be compared (maximum size of 128 bits).

This allows to check for virtually any portion of the packet, enabling for a very flexible definition of flow. On the other hand, the **action** field will be populated by one or more operations to perform if a packet is matched by all the rules. An action will be composed again by the *field*, *offset*, *size* and *value* attributes, with the addition of a *type* field, which specifies the actual operation, selected from:

- **Forward:** send the packet to a specific *MAC address* (one-hop);
- **Broadcast:** send this packet to all neighbors (this is a MAC broadcast, since IPv6 does not support this procedure);
- **Modify:** change the value of a specific field in the packet;
- **Drop:** discard the packet;
- **Decrement:** decrease a specific value in a field;
- **Increment:** increase a specific value in a field;
- **To Upper Layer:** deliver this packet to the layer above SDN in the stack;
- **Continue:** analyze the flow table in search for other matches.

Moreover, thanks to the implementation of this table, utilization of DAO messages for allowing P2P and P2MP communication won't be needed, thus completely disabling this RPL feature and creating a DAO-free environment.

4.1.3 Communication protocol

Since this system aims at being fully standardized and not utilizing any proprietary protocol, CoAP is utilized in order to exchange messages between the SDN Controller and the SDN Sensor Nodes. To achieve this, some RESTful resources are defined inside both the nodes and controller(s), which can be accessed by all the standard HTTP methods like GET or POST. Each resource will represent one component of the SDN architecture, namely:

- **Main controller resources**
  - *6lo*: resource queried by the SDN nodes in order to send topology updates and keep the controller informed about the current network status.
  - *fe*: resource queried by the SDN nodes in order to ask the controller for instructions when no match is found inside the flow table.
- **Main node resources**
  - *ft:* used by the SDN controller to remotely manage data inside the node's flow table;
  - *neighbors table:* queried by the controller for retrieving info about the node's neighbors;

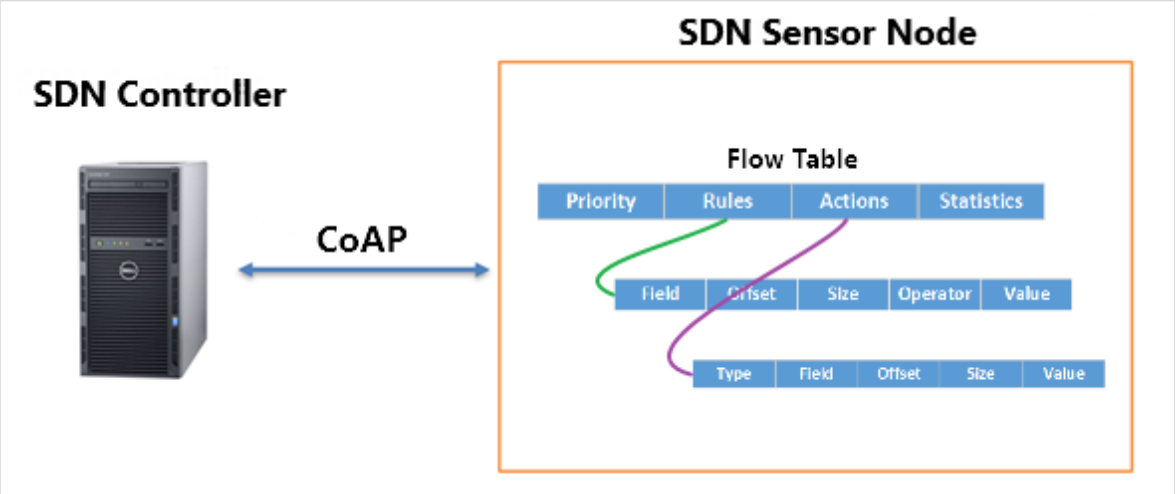


Fig. 4.2 – Southbound interface and flow table design

4.2 System Implementation

Although not all of the initially listed functionalities were implemented, the core features of the described architecture are as of today correctly in place and allow for a completely working simulated system. The selected operating system on which all of this takes life is represented by Contiki[15], which is open source and based on the *C* language. Moreover, in order to avoid overhead of messages caused by an high volume of fragments derived from the transmission of topology update messages (which could be large in size,

proportionally to the number of neighbors), the CBOR[16] data format is utilized, so to build smaller packets.

Of course, being a good part of the architecture represented by standardized protocols, only the SDN software was needed to be coded from scratch inside the Contiki OS.

4.2.1 The SDN layer

From the design specifications, an SDN layer must be inserted in-between the 6LoWPAN adaptation and the IEEE 802.15.4 MAC ones, by also providing a Flow Table data structure and a Control Agent definition for implementing a data plane extension mechanism. This was done by modifying the Contiki's network stack so to intercept packets traversing between 6LoWPAN and the lower layers, successively analyzing them with multiple algorithms in order to match them to the flow table. Since a mesh-under technology is needed in order for the system to work, it was also successfully introduced inside the Contiki OS, which didn't provide it by default.

On the other hand, the Flow Table was implemented by means of a fixed data structure composed by multiple memory pools (one for each substructure, namely *entries*, *rules* and *actions*) of predefined size (maximum 40 entries, by default). The Control Agent was then implemented by writing another piece of software and including its initialization from the node's core code. More precisely, this module is able to exchange messages with the controller through the previously defined CoAP resources and by parsing the received responses. When a table miss happens (meaning that no rule was found inside the node's flow table), this Control Agent will send a request to the controller, waiting for a response for a certain period of time and, following the idea of Sensor OpenFlow, temporarily storing the current destination address so to avoid querying the controller repeatedly and cause overhead. Furthermore, the same Control Agent module will allow for a local control logic (through the exploitation of a neighbor table that is already implemented in the Contiki OS) and the periodic sending of topology update packets.

To not neglect any detail, it must also be specified that all the part regarding the CoAP and REST resource management is by default available in Contiki under the name Erbium (Er), which is a REST Engine that provides both server and client implementation.

4.2.2 The SDN Controller

Although the SDN Controller could be implemented by utilizing any possible language that supports CoAP, the chosen programming language is Java, specifically with the help of the Californium framework[17], which allows to deploy a custom CoAP client or server by using a simple API.

The *6lo* resource was implemented by writing a Java class that handles topology updates coming from the WSN and stores collected information into a network graph by utilizing the GraphStream[18] library, also with the help of CBOR in order to correctly decode the received packets.

On the other hand, the *fe* (flow engine) resource was implemented by writing a Java class that handles POST requests and answers with rules by mainly computing the shortest path based on *hop count* (derived from the network graph through the Dijkstra Algorithm).

4.2.3 The SDN Sink Node

Finally, a sink node is of course needed in order to allow SDN Sensor Nodes to reach (and be reached by) the external word. This is simply done by utilizing a *RPL border router* node already defined in Contiki and injecting the Control Agent into it (through the same line of code used for sensor nodes). Both the SDN Sensor Nodes and the SDN Sink Node are therefore currently supporting SDN and at the same time exploiting the Contiki's RPL implementation for organizing into a tree topology.

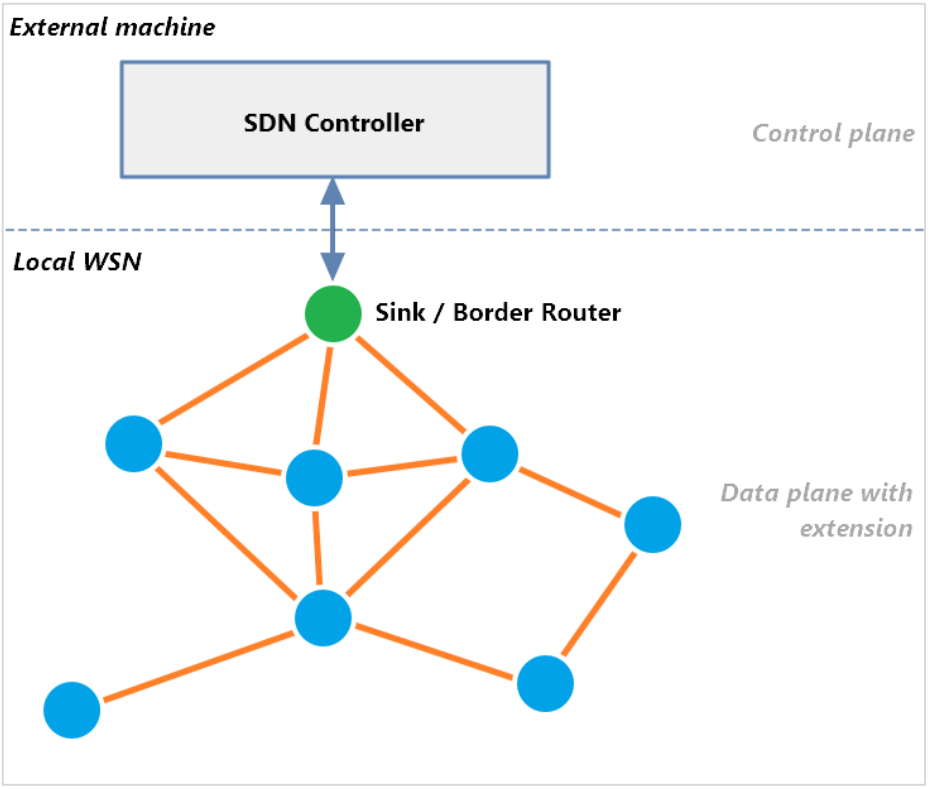


Fig. 4.3 – An overview of the proposed system's architecture. As it can be noticed, the controller resides on a machine which is completely external from the WSN and it is able to communicate with nodes by first traversing the Border Router. Orange links are representative for wireless connections.

Chapter 5

From simulation to testbed

As anticipated, even though the proposed system is completely functional on the standard Contiki simulator (i.e. Cooja, all the results are available in the original document), it presented multiple issues when dealing with real sensor nodes and for this reason it was not successfully implemented on a concrete network. Starting from this situation, the first objective of this thesis work was therefore to extend this system from the ideal world typical of simulations to a real environment, where all issues described in the previous chapters represent a true menace for its correct functioning.

5.1 The university's testbed

The real environment utilized during this thesis activity is represented by an IoT testbed deployed inside the Information Engineering Department of the University of Pisa, which is composed of 21 RE-Mote sensor nodes produced by Zolertia[19]. This hardware platform features a Zoul[20] as its core module, which, among other things, implements an ARM Cortex-M3 CPU with 512 KB of flash space, 32KB of RAM and 32Mhz of processing power. Those nodes are placed in multiple rooms of the department (one for each office, distributed on two floors) and attached to a stable power source, thus avoiding the necessity of power management during the testing phase.

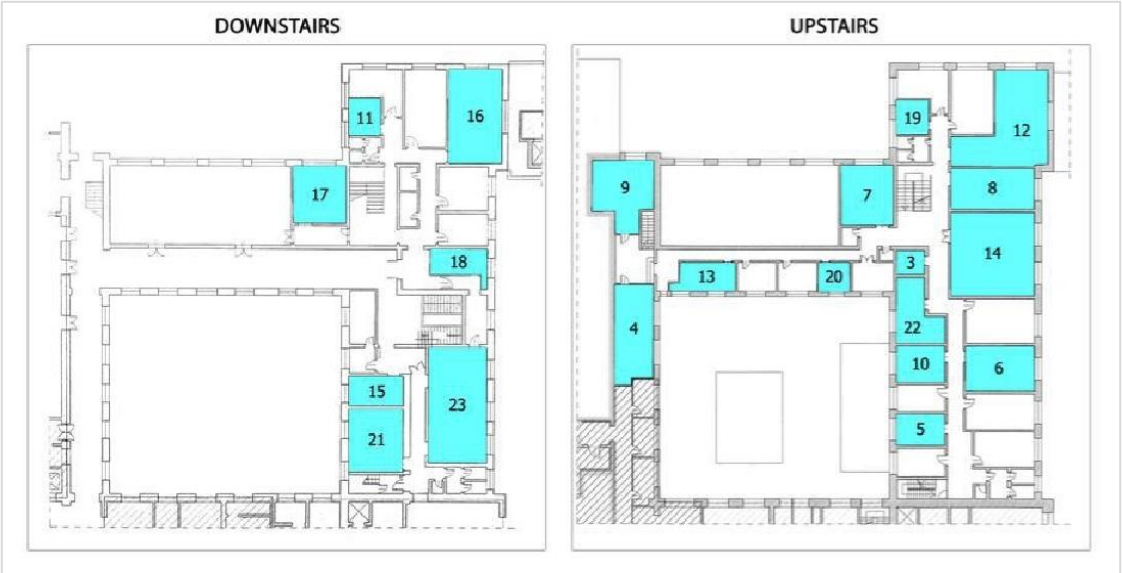


Fig. 5.1 – The actual testbed topology, nodes go from ID 3 to 23, while the sink is hidden (ID 1)

This infrastructure also presents a Linux server working 24/7, which allows to directly flash any firmware inside all the sensor nodes and collect results from triggered experiments. Moreover, an additional RE-Mote node is directly attached to this machine, so to allow the deployment of a sink inside the network (also called border router, since it provides the ability to exchange messages between the local system and the internet). All of this will therefore represent the topology utilized for testing purposes.

5.2 Resolution of problems

Without any surprise, multiple issues were faced while trying to implement this system on top of the described testbed, both because of the constrained nature of sensor nodes and of some incorrect assumptions that were made in its first definition. In this section, the main blocking problems that were faced are analyzed, together with their respective solutions. The enhanced code obtained from this work is available on GitHub[21].

5.2.1 Memory footprint

Before even being able to check for the correct functioning of the proposed system on the testbed, a first problem was represented by its RAM usage. In particular, there was no

possibility to compile the original source code and install it on top of the RE-Mote nodes, due to a memory overflow of precisely 3826 Bytes. Therefore, through a detailed analysis of the source code regarding SDN Sensor Nodes, the reduction of some parameters was performed without altering the original system's design. Starting from the *flow table* data structure, its dimension was drastically reduced with regards to the default 40 maximum *entries* per node, also reducing the size of the pools in which *actions* and *rules* are stored. Moreover, multiple parameters of Contiki were changed (in the *project-conf* file) in order to properly fit the available memory space, together with the reduction of resources allocated for CBOR packet decoding, since they were overabundant.

5.2.2 Native border router

A careful reader should have noticed that the testbed's sink is represented by a **constrained** device completely equal to a normal sensor node, hence not properly respecting the assumptions made in previous chapters, where it was considered to be unconstrained and able to perform computational-heavy tasks. Even if the original architecture design expected SDN Controllers to reside completely outside of the network and to be directly connected to the sink, a blocking issue was faced when testing this framework for the first time on top of a real infrastructure. More precisely, since the SDN Sensor Nodes must communicate with the controller(s) if a table miss happens (or to send a topology update message), it is easy to foresee that the network's bottleneck will be of course represented by the sink itself, which is the one entitled of allowing this kind of communication. Therefore, a rule for each node in the system will be installed inside the border router's flow table, so to allow the correct forwarding of control messages from SDN Controller to SDN Sensor Nodes. This brings to the necessity of having space inside the flow table for at least as many entries as the number of sensor nodes populating the WSN.

The above condition (assumed to be implicitly true in the original system design) was not satisfied in this testbed with 21 nodes (due to the previously made modifications) and, in any case, was precluding the theoretical correctness of the entire architecture (since WSNs are by definition dynamic and therefore nodes could join or leave the network at any time), implying a high probability of faults if the sink's memory is fulfilled. To solve

this critical issue, a **native border router** solution was implemented, meaning that the SDN software is now running on top of an unconstrained device (i.e. the server computer directly attached to the sink), while the node itself is utilized only as a radio device (called **slip-radio**) for communication purposes. In this way, a flow table that is much bigger than the network's size can be allocated and the probability of faults is drastically reduced.

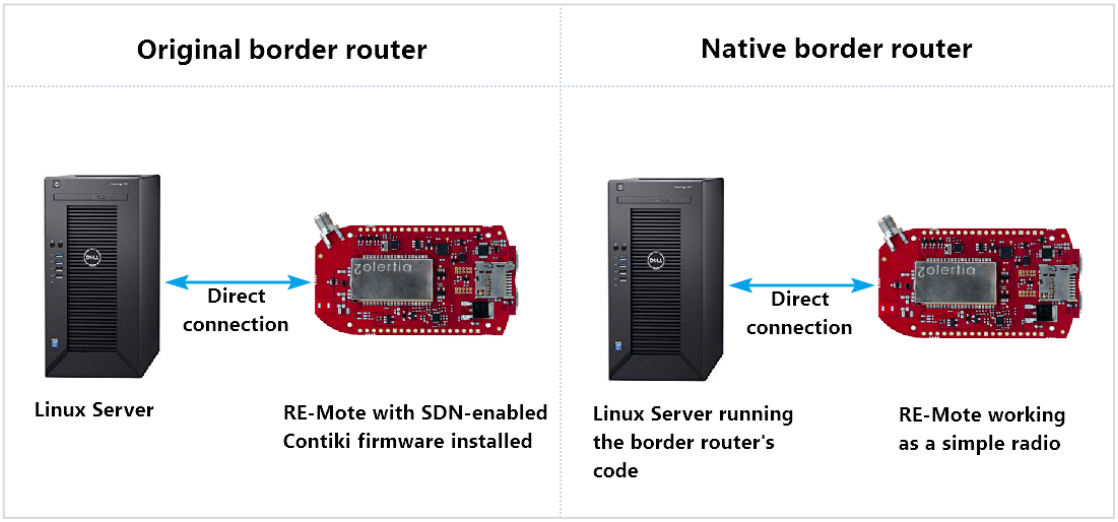


Fig. 5.2 – Graphical comparison between the originally designed border router and the implemented one. As it can be seen, in the first case the server machine was just utilized for initially flashing the firmware on top of the mote, while in the latter it performs an active role inside the overall system architecture.

5.2.3 RPL rule allocation

It is specified by the original architecture design that each SDN Sensor Node must support the RPL protocol and utilize it in order to participate to the formation of a DODAG, instantiated by the SDN Sink Node. Therefore, RPL is specifically utilized for establishing a tree topology and allow each node to correctly reach the border router through a so-called *upward route*, this by installing a forwarding rule inside the flow table. Such entry should match a packet that is directed to the SDN Controller IPv6 address and select as next hop the node's preferred parent (chosen by the RPL procedure), thus allowing for a chain mechanism that will bring to the sink and subsequently to the SDN Controller itself (which resides outside of the network).

For unknown reasons, the source code of the solution proposed by Micheloni implemented such feature by assuming an SDN Controller which was local to the network and consequently addressable by means of a well-known MAC address. Therefore, in all nodes a static entry was hardcoded inside the flow table for allowing the detection of this MAC address and subsequent forwarding to the preferred parent, violating the system's theoretical correctness. To solve this issue, a new piece of code has been implemented inside the 6LoWPAN layer, so to correctly detect when a packet is destined *outside* of the network (by looking at the destination's IPv6 prefix) and consequently apply the previously defined static forwarding rule. This implies that there is no more the need of knowing what exactly is the controller MAC address, since as long as it stays outside of the WSN, it will be reached by forwarding the packet to the SDN Sink Node, which will take care of its correct delivery.

5.2.4 Neighbor information management

Thanks to the *topology update* messages that periodically flow inside the network towards the SDN Controller, it will be able to build a global knowledge regarding the entire system's architecture and take optimal decisions on this basis. Unfortunately, a very important problem was detected during the testing of the original solution, namely the fact that the *neighbor table* resource was not properly refreshed and therefore it was not reporting an updated status of the network. This was evident due to the crucial step of bringing a theoretical solution from a simulative workspace to a real environment, in which links do not remain the same forever and the connectivity is dynamic because of the already explained issues related to the wireless medium. For this reason, the **6LoWPAN neighbor discovery** procedure, already implemented in Contiki, was activated (to perform periodic pooling of neighbors in order to get their updated status) and some new code was added inside the SDN and 6LoWPAN layers to allow a proper management of the neighbor table.

Moreover, since the Control Agent module present in all SDN Sensor Nodes is able to autonomously (*data plane extension*) install rules inside the flow table whenever a new neighbor is detected (i.e. a message is directly received from it), this could lead to wrong

assumptions when trying to reach another device inside the network (even though those rules have a certain lifetime, after which they are deleted, if not refreshed). As for an example, if node A knows node B, which in turn is neighbor of C, it could be possible that A selects B as next-hop for reaching C, but at the same time it could happen that B is unfortunately **no more** able to reach C, thus implying packet loss and multiple retransmissions (since B always finds a match inside its flow table, because of the previously installed entry). Therefore, by exploiting this newly introduced management mechanism of the neighbor table, a solution to this additional issue was implemented by adding a function inside the Control Agent that is called whenever a node is removed from the neighbor table, consequently *removing all rules* that imply a direct forwarding to that device. In this way, node A will forward the packet to B, which now hopefully won't have any rule regarding C and will consequently ask for instructions to the controller (which in turn has now a *truly updated* vision of the entire network).

5.2.5 6lo resource adjustments

As described in 4.1.3, the *6lo* CoAP resource is provided by the SDN Controller in order to grant the correct reception and decoding of topology update packets sent periodically by SDN Sensor Nodes. This should theoretically allow the controller to build a graph of the network and keep it up to date. However, in the original implementation of the Java code for the SDN Controller, the only aspects taken into account were the *addition* of new edges (links) in the graph and the update of information regarding all sensor nodes, not considering the possible *deletion* of nodes and links. Therefore, as a consequence of 5.2.4, a new function was implemented also inside the SDN Controller in order to allow the removal of edges that are discovered to be no more existent thanks to the topology update messages, together with nodes that may leave the network. In this way, a true global knowledge will be stored by the controller and utilized to correctly manage the entire network.

5.3 Testing

After resolving the memory footprint issue presented in the previous paragraph, the SDN-enabled Contiki operating system was successfully compiled and implemented on top of all nodes of the testbed. This represented the achievement of the first milestone proposed in this work of thesis, but didn't immediately satisfy all theoretical results obtained in the original proposal. It is indeed evident that, even though the system was now completely functional and working on a real-life architecture, some tests were needed in order to check for possible hidden issues and to validate the performance results obtained during simulation. This analysis led to the discovery of all other problems analyzed in the previous section, that were resolved (as described) step by step.

5.3.1 Principal features

It has already been explained that normal WSN architectures typically utilize the RPL protocol for all type of communication (MP2P, P2P and P2MP), while the proposed system exploits it only for determining upward paths (MP2P) in a distributed manner. Indeed, in this SDN-enabled architecture, the controller's global knowledge substitutes RPL in order to trace downward (P2MP) and direct (P2P) routes between nodes, thus allowing for a more flexible network management and introducing new possibilities for performing QoS, TE or even implement virtualization of the underlying infrastructure. Therefore, all testing activities were carried out by considering the comparison between a standard WSN and the SDWSN obtained during previous chapters, so to check if all these advantages brought by SDN are really worth the cost of implementing it over WSN. In those experiments, three main communication scenarios were defined, namely:

- **UDP Up:** each node in the topology sends periodically a not confirmable CoAP message to an application outside of the network. This implies the need of only upward routes, that are in both cases defined by RPL. For this reason, this scenario was tested just once in order to check that all was properly working and then excluded from the successive experiments;

- **UDP Up & Down:** in this case, each node periodically sends a confirmable CoAP message to an application placed outside of the network. Because of its nature, the CON message will need an ACK from the receiving machine to the sending node, thus implying the need of a downward route (defined by RPL in one case, and by the controller in the other case);
- **UDP Down & Up:** in this last communication scenario, the external application periodically polls individually all nodes inside the network through CON CoAP messages. As opposed to the Up & Down scenario, in this case a downward route will be needed for each message to correctly reach the respective node, while an upward path will be necessary for the correct delivery of the acknowledgement packets.

Moreover, some peculiar measures were selected in order to properly examine the results of each experiment, namely:

- **Bytes transmitted**: this measure represents the number of Bytes transmitted by each node inside the network, thus allowing to discover what is the *total overhead* caused by the implementation of SDN over a WSN architecture, together with the information regarding what nodes represent a bottleneck for the proper network communication;
- **Packet loss:** regarding the Up & Down and Down & Up cases, this measure represents the number of messages that were lost during the communication between application and node, thus indicating the reliability of paths selected by RPL versus the ones of the SDN controller. Additionally, the packet loss is influenced by the available buffer space for packets inside each node (since if the buffering queue is full, they will be discarded), and for this reason it will also be able to summarize what is the impact on reliability given by the overhead introduced by the SDN system (hence, the probability of network congestion);
- **Round Trip Time (RTT):** this is the time elapsed between the sending of a message and the subsequent reception of the relative acknowledgement. It indicates the overall performances of a WSN implementing SDN versus a standard one, with emphasis on how well a SDWSN can treat applications that need fast delivery of data;

- **Control messages overhead:** in order to better quantify the cost payed when implementing SDN over WSN, the number of bytes that flows inside the network purely for control purposes (both for RPL control messages like DIOs and table miss or topology update messages in SDN) is also considered.

As anticipated, by utilizing the presented communication scenarios and relative measures, multiple experiments were defined during the testing phase of the proposed architecture. In this regard, it is imperative to underline that, since those tests involved a real infrastructure with real nodes, sometimes not all of them were available for the whole duration of an experiment due to technical reasons, even if of course not in a significative number (typically only one or maximum two nodes). Furthermore, each experiment scenario was run in the same conditions (as much as reality permitted) for a total of five times, thus implying that the *mean values* for all proposed measures were computed. A *confidence level* of 95% was also defined so to compute the relative intervals that indicate with how much certainty the obtained results are correct. More precisely, if an experiment of this nature runs 100 times, a confidence level of 95% suggests that 95 out of 100 times the resulting values will be inside the computed interval (hence, with a 5% error margin).

After the definition of all these parameters, another aspect was considered before starting with the actual experiments, that is the amount of time needed by the SDN system to install fundamental routing rules inside the network, defined as the **warmup time**. To perform this measurement, all forwarding rules installed by the controller were changed to be of infinite duration (TTL equal to zero) and therefore never deleted from the SDN Sensor Node's flow table, thus implying that the time elapsed from the start of the test until the last installation of a flow table rule was exactly the warmup period. This value, on an average computed from 10 repetitions of a 1-hour experiment, was of 15 minutes.

Parameter	Value
Network topology	University of Pisa's IoT Testbed
Testing scenarios	UDP Up & Down, UDP Down & UP
Measures	Mean Bytes transmitted Mean Packet Loss Mean Round Trip Time Mean Control messages overhead
Confidence Level	95%
Warmup period	15 minutes
Experiment repetitions	5 runs for each scenario (total of 10)
Comparison method	UDP Up & Down on WSN versus SDWSN UDP Down & UP on WSN versus SDWSN
Total experiment runs for one complete test	20 runs (10 for the SDWSN and 10 for the normal WSN)
Experiment duration	1 hour (15 of warmup plus 45 of steady state)
Total duration of one complete test	20 hours (10 for Up & Down and 10 for Down & Up)

Table 5.1 – Overview of the main experimentation parameters

5.3.2 Testing timeline

Although not all results will be deeply analyzed in this document, it is important to understand that multiple tests were performed and some adjustments to the entire SDN-

enabled system were derived from their results. Therefore, a brief summary of what was done is hereby presented so to describe the flow of events.

Experiment zero – Warmup period definition: the first real test performed on top of the testbed topology was aimed to determine the warmup time of the SDWSN architecture, as previously described. Of course, this allowed to focus on the analysis of the system's *steady state*, by no more considering this first period of time in all the successive experimentations.

First experiment – Native border router implementation: after the warmup period was defined, a first experiment was performed in order to check whether the results obtained from simulation were already verified. Unfortunately, even though the current system was correctly working (hence already representing an achievement), this did not have good performances due to the multiple obstacles brought by a real infrastructure with interferences and technical issues. Moreover, something wrong was happening inside the sink node, which was fulfilling its flow table way too fast. After this first test, the UDP Up case was no more considered (because of the already explained reasons) and some adjustments were made both to the parameters regarding SDN Sensor Nodes capabilities and to the SDN Sink Node (as discussed in 5.2.2).

Second experiment – SDN layer enhancements: after those modifications, another complete test was run to check if the system was now working better, indeed reaching mediocre performances in the UDP Up & Down case, but remaining in almost the same position for what concerned the Down & Up scenario. This was indicating that something wrong was happening when defining downward routes (since RPL was correctly delivering the messages, while in most of the cases nodes didn't receive anything) and not because of network parameters. Therefore, the focus was moved on the enhancement of the SDN layer, as explained in 5.2.3, 5.2.4 and 5.2.5.

Third experiment – From Hop Count to ETX: this time the results were satisfying, but something was not allowing the correct comparison between the Mean RTT offered by RPL versus the one provided by the use of downward paths defined by the SDN Controller. Since both systems should have considered the shortest paths weighted with the traversed number of hops (hop count), a deep analysis of the RPL routing code implemented in Contiki was performed, discovering that it was instead utilizing the ETX

(expected number of retransmissions) of links between nodes as a cost measure. This was true even if its "zero" objective function (which should theoretically consider normal hop count[22]) was specifically selected. For this reason, a new update was made to the controller's Java code in order to utilize the ETX information provided by topology update messages as the new cost measure for computing the shortest path between nodes and consequently install rules inside their flow tables.

Fourth experiment – Dynamic topology updates: even if the overall system performances were now satisfying and comparable with the standard WSN case, the number of bytes transmitted inside the network was still indicating that the SDN system introduced too much overhead of messages. This new experiment was then made by completely removing the topology update after the warmup period (hence, having a steady state without any of those messages), resulting in a substantial reduction of overhead. Therefore, a dynamic topology update system was proposed instead of sending those messages on a periodic basis, meaning that such packets should be sent only when there is an important change in the network's topology (as for example a node losing multiple neighbors). This system was not implemented during this thesis work, but is strongly suggested since it has been empirically proven to allow better performances.

During these tests, the controller's global knowledge was also exploited in order to display the network topology as a graph in real time. An example picture obtained from one run of an experiment is reported below.

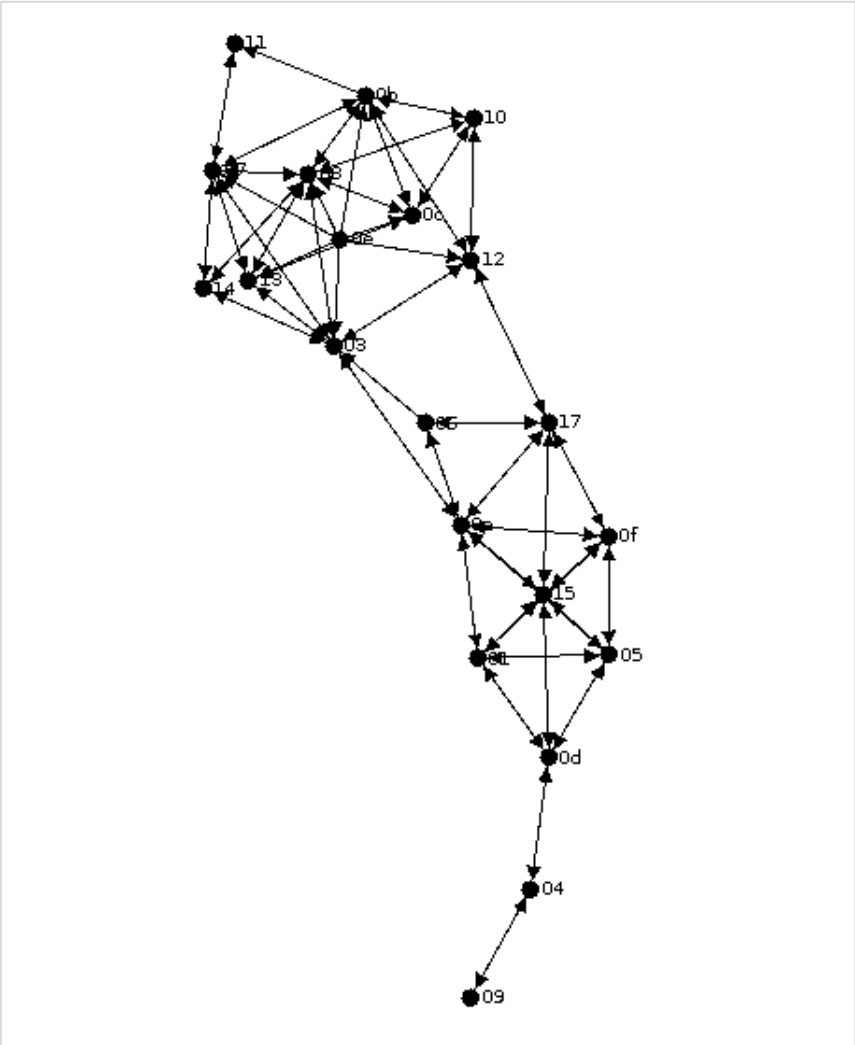


Fig. 5.3 – Testbed topology seen by the controller's point of view

As it can be noticed, the topology built by means of update messages highlights some nodes (like 0d, which corresponds to node 13 when converting from hexadecimal to decimal notation) as chokepoints and hence bottlenecks for the communication between the two subsystems that they are connecting. To better emphasize the validity and importance of this graph from a theoretical point of view, following is reported a picture comparing the architectural testbed topology versus the one obtained through the SDN-enabled system (node 14 and 22 were unfortunately disabled because of technical reasons).

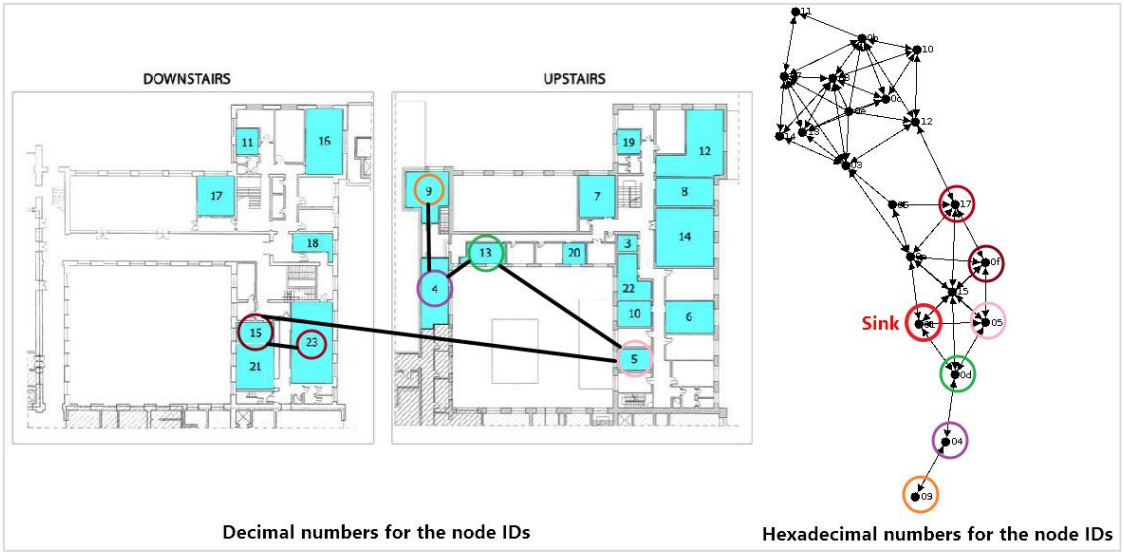


Fig. 5.4 – Comparison between the real topology and the one obtained at the SDN Controller

Although not displaying all connections for simplicity, it can be easily seen that indeed the highlighted nodes are neighbors in the real network topology, with for example node 5 being exactly on top (upstairs) of node 15 and 21, thus connecting the system in a tri-dimensional way (node 15 in the graph, which is not emphasized, represents node 21 when converting in decimal format).

5.3.3 Final Results

After defining and implementing all adjustments described previous chapters, a final test was performed in order to extrapolate what are the actual capabilities of the proposed SDWSN architecture. Before going into details, a table summarizing the core system parameters is hereby illustrated (these must be considered together with table 5.1).

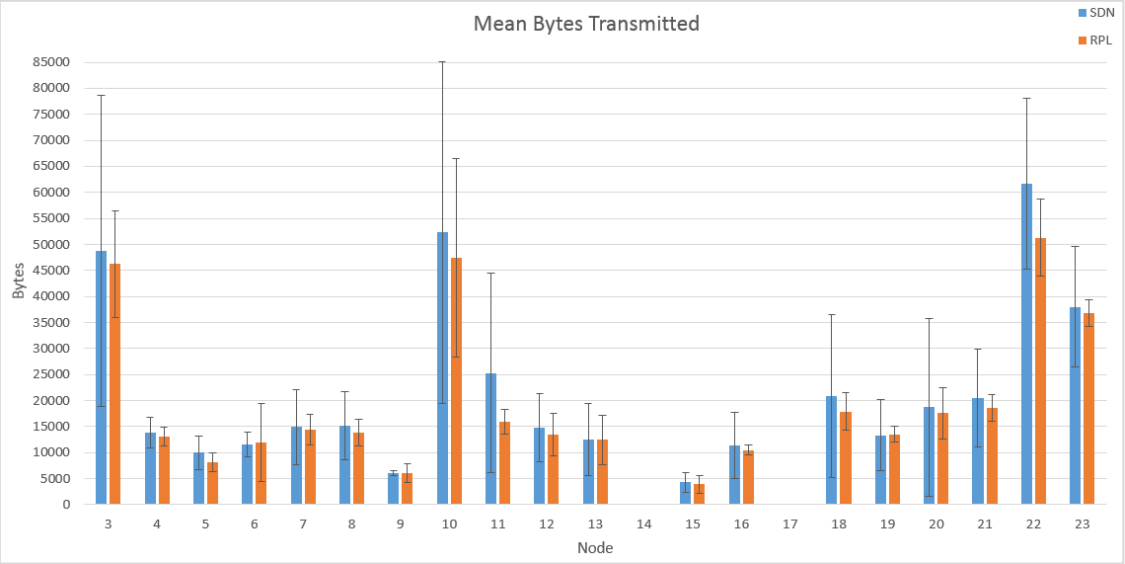
Parameter	Value
Shared parameters	
Packet buffer size	8
UDP messages period	2 minutes
Total number of messages	30 in both scenarios (7 during warmup period)
RPL Objective Function	OF0 (zero)
Border router type	Native border router
Max number of neighbors	10
Normal WSN experiment parameters (only RPL)	
Routing table size	20 entries
DAO messages	Enabled
Mode	Storing mode
SDWSN experiment parameters (SDN over WSN)	
Controller routing algorithm	All Pair Shortest Path with ETX cost measure
DAO messages	Disabled
Topology update period	1 minute, then after 5 minutes and then every 10 minutes
Border router	Native
Flow table size	Maximum 19 entries (80 for the border router)
Entry lifetime	20 minutes (after which it is deleted from the flow table)

Table 5.2 – System parameters for the final experiment

Since the system's behavior during the transient period is in general very similar to what was already analyzed in previous simulation experiments and due to its irrelevancy for what concerns the real performance interests, all results that will hereby be presented are specific for what happens during the steady state (hence, during a normal operational workflow). Moreover, because of the already explained issue represented by the fact that the system is now implemented on top of a real network infrastructure, two nodes were missing when this test was performed, namely 14 and 17.

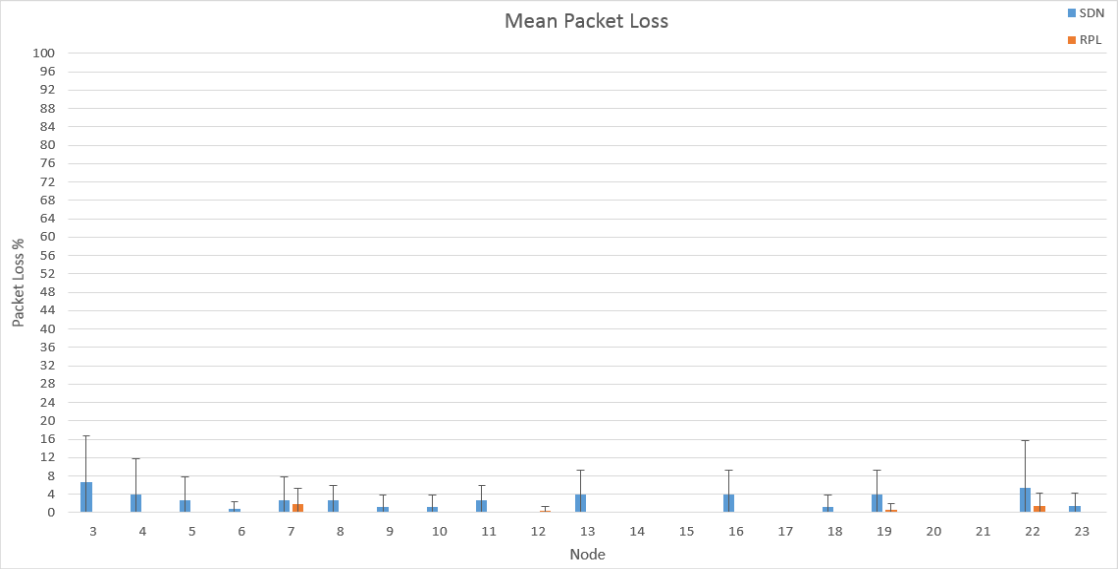
UDP Down & Up

Following are presented the resulting measures obtained by comparing the RPL performances versus the SDN-enables system, each of which is summarized through a bar plot (with relative confidence intervals) where orange bars represent the normal WSN architecture running RPL, while blue bars indicate the SDWSN implementation.

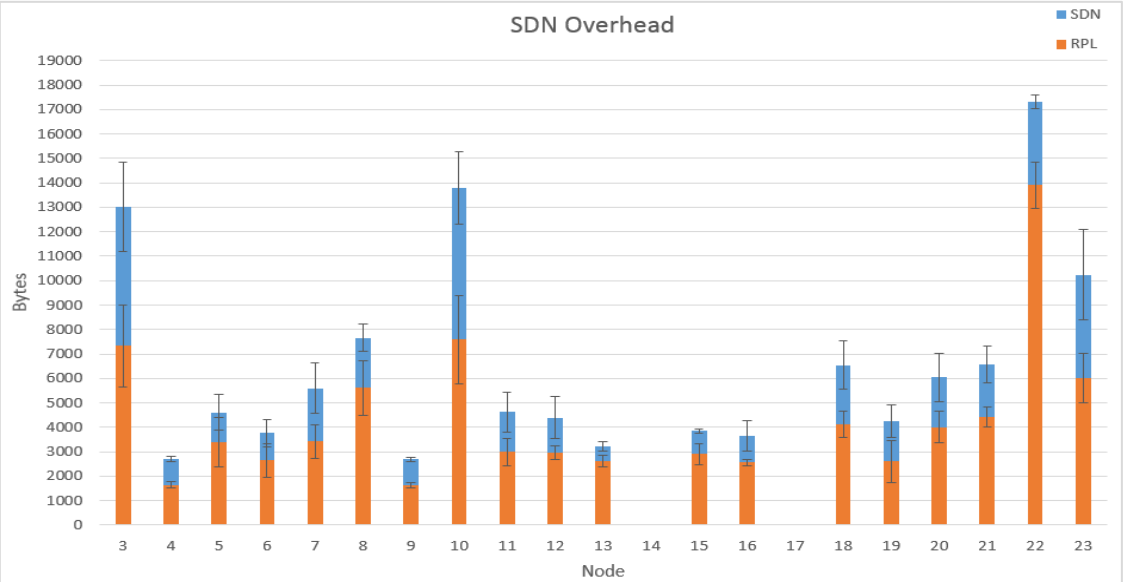


For what regards the bytes transmitted in mean by each node during the run of a Down & Up scenario, it is evident that the overhead introduced by SDN is not very high with regards to the normal RPL performances, thus implying the need to pay just a small price in terms of bandwidth utilization in order to enable all powerful features provided by a software defined network. This is mainly caused by topology update messages and the exchange of data between nodes and controller due to table misses, together with the additional 17Bytes of packet size introduced by the utilization of a mesh header (for the

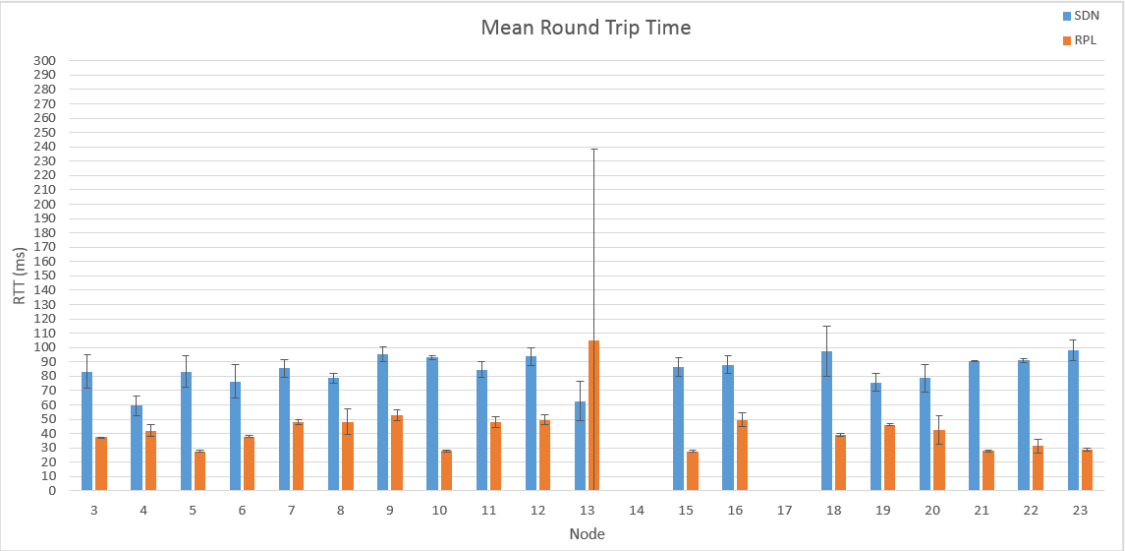
described mesh-under approach). Another peculiar aspect that can be emphasized by looking at the above plot is the fact that some nodes (namely 3, 10, 22 and 23) are needed to transmit slightly more data than other ones. This can be easily explained by looking at the network's topology (pictures 5.3 and 5.4), in which those nodes represent indeed virtual chokepoints and must therefore perform relay actions more often.



Concerning the overall communication reliability, it is possible to see that unfortunately the SDN-enabled system has some packet loss (maximum 16% if considering the confidence intervals), but nothing too compromising for a non-critical application. Since RPL performs way better, this indicates that the problem mainly resides in the lost of packets because of table misses, together with the downward routes selected by the controller, which are sometimes not optimal because of a topology that is not completely up to date. Furthermore, another key factor is represented by the limited buffer capacity of sensor nodes (8 maximum packets in queue), which is even more reduced due to the overhead introduced by SDN control messages and topology updates, thus causing packet loss and multiple retransmissions.



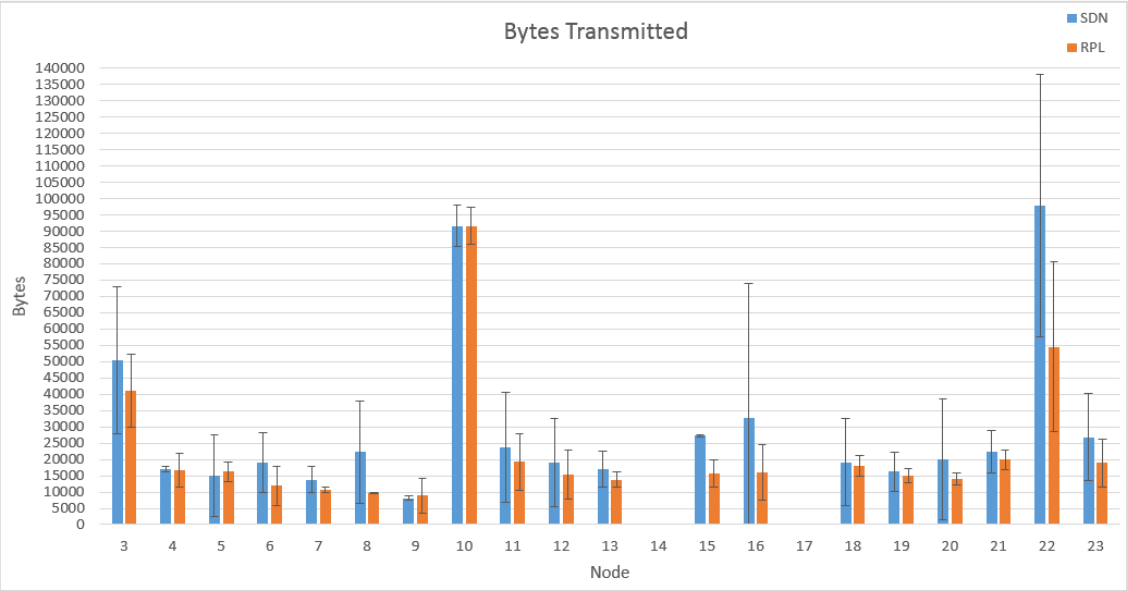
The previous conclusions are indeed supported by the above plot, which represents the mean overhead introduced by SDN in terms of *control messages* being transmitted throughout the network. As it can be seen, because of topology update messages and, more importantly, exchanges triggered by table misses, a substantial overhead is introduced in addition to the Bytes needed for normal RPL management, hence resulting in smaller space for the actual relevant data traveling inside the network. This cost was of course expected and can be accepted if SDN truly allows for an overall better architecture. Moreover, if a comparison between this and previous charts is made, it can be noticed that it proves once more time the above-the-mean overhead introduced in some nodes in order to perform the relay role (in fact, they will need to know how to forward those additional packets, hence causing a domino effect).



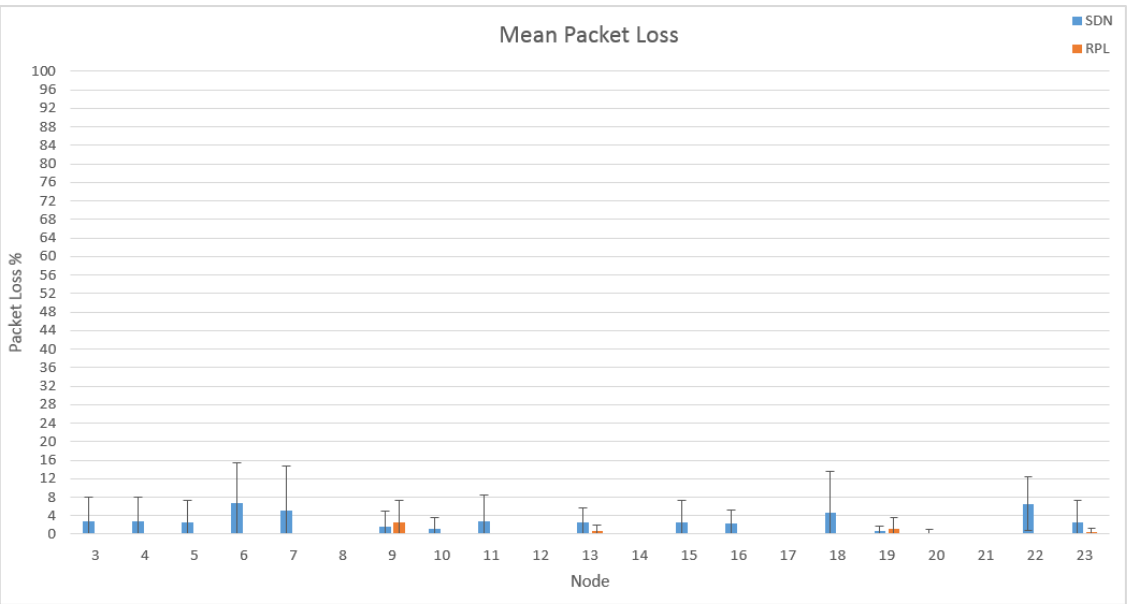
The last measure shows what is the mean time elapsed from the departure of a message to the reception of an acknowledgement coming from its destination. Since the SDN system introduces some packet loss, it is reasonable to say that this should cause multiple retransmissions and hence require more time for a message to reach its destination correctly, also causing a bigger number of Bytes being forwarded throughout the network (as seen likewise in previous plots). This is again a cost that could be considered to be not so disruptive if the overall architecture is providing more functionalities than a standard WSN.

UDP Up & Down

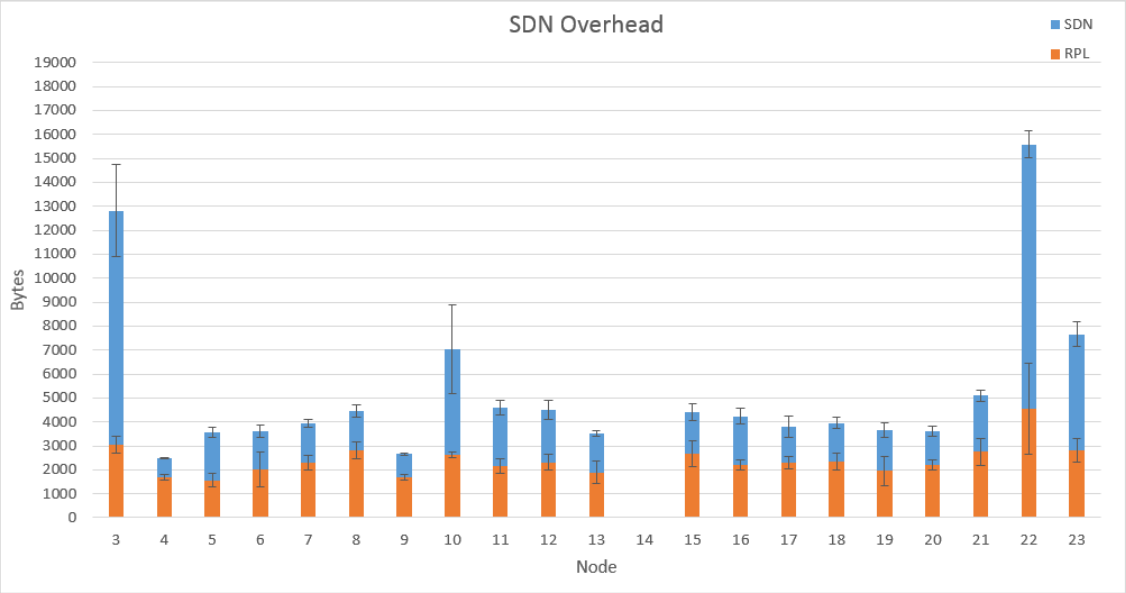
This second scenario involves the sending of messages from an external application to nodes inside the network. For this reason, it should imply more Bytes to be forwarded because of the table misses caused by the downward routes needed for correctly polling nodes. This can be explained by the fact that it indeed represents a different problem with regards to the Down & Up scenario, since in that case nodes were querying an external application and only ACK messages were traveling in a downward path, thus requiring less space to be packed up inside a table miss request and subsequently causing a smaller number of Bytes to travel the network for SDN-specific reasons.



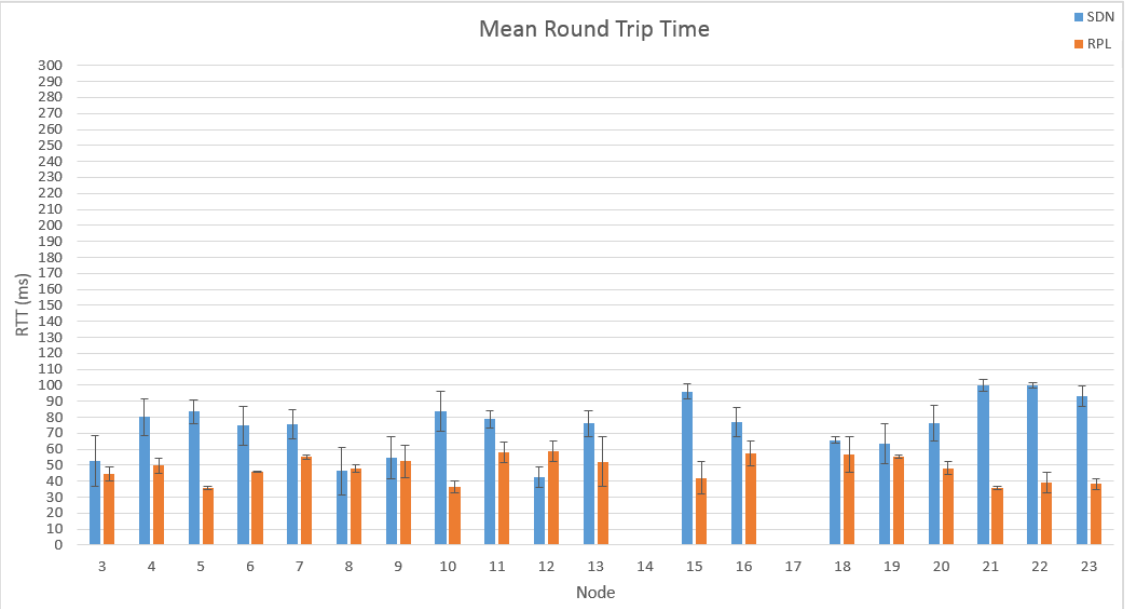
The above plot proves what was in fact expected regarding the mean bandwidth utilization in terms of Bytes transmitted. Indeed, through a graph comparison, it can be noticed that the minimum value increased from the almost 5KB of the Down & Up case to about 10 KB, while the maximum value is now close to 100 KB, as opposed to the 60KB reached in the Down & Up case. Since this is true for both the SDWSN and the standard WSN case, an approximated conclusion could be represented by the fact that a communication scenario implying the polling of nodes from an external application requires (almost) double the bandwidth with respect to the reverse procedure.



The communication reliability in terms of packet loss is almost identical to the previous scenario, hence all explanations already provided are valid also in this case.



The mean overhead introduced by SDN control messages is also very similar to the previous case, in fact the only difference between this and the former scenario is represented, as explained, by the size of packets traveling upward and downward in the network. Indeed, it is possible to notice that here the use of RPL does not provide so much overhead for control messages, while SDN adds a larger number of Bytes with regards to the former case, even if the total sum is almost the same. This can be explained thanks to the fact that, since most of the traffic is coming from outside the network (Up & Down), table miss requests (considered as SDN overhead) will be of bigger size (encapsulating a whole message instead of just acknowledgements), while RPL control messages will be needed for just determining the upward routes.



The mean round trip time chart displays again very similar values to what was obtained in the Down & Up scenario. This is mainly caused by the mean packet loss experienced in both cases, which is almost identical, that induces multiple retransmissions and subsequent delays in the delivery of a message.

5.3.4 Conclusions

Although the testbed's topology was small (21 nodes), dense and almost fixed (with changes only depending on interference and wireless asymmetricity), the results obtained in these experiments suggest that it is indeed possible to implement an SDN architecture over a real WSN, without paying unaffordable costs. Furthermore, by performing multiple tests, during this thesis work some enhancements were implemented in the original system, allowing for an almost optimal solution. As a conclusion for the achieved results, it can be said that, even though these reasonings are specific to university's testbed conditions, they represent an important contribution to the work that has been made during the last years regarding the SDN over WSN argument. However, what was done up to this point aimed for much more than simply utilizing this SDN-enabled system for performing routing of packets inside the network, also because the RPL solution is already in place and (as seen) working better than the proposed one.

Chapter 6

Exploiting SDN

The reader should at this point be aware of the real reason that brought to the selection of the SDN paradigm as a solution for IoT environments. More in detail, as already described in previous chapters, a network in which it is possible to separate the control plane from the data layer will be able to provide much more flexibility and scalability than normal ones, provided that a proper controller positioning is carefully selected. Therefore, the second objective of this thesis work was to utilize the SDN Controller placed on top of the testbed's WSN for performing more than just simple routing activities.

6.1 Network Slicing

As described in 2.4, one of the new possibilities introduced by the SDN paradigm is to easily support network virtualization and multi-tenancy, hence creating general-purpose environments that can be virtually shared between multiple clients while utilizing the same physical infrastructure. Although in standard sensor networks utilizing distributed routing protocols (like RPL) this is practically unfeasible, thanks to the correct implementation of SDN over WSN, this concept represents now a concrete feature that may be provided by IoT architectures.

6.1.1 Slicing engine

In order to properly implement a multi-tenant SDWSN, an initial definition of how to subdivide it into multiple virtual slices must be provided. In particular, given that a virtualized network architecture should be able to supply different functionalities to numerous clients without exposing the real physical infrastructure to any of them, a first mechanism must be introduced inside the control layer in order to manage requests from

each tenant's application. While considering this necessity, it should be reminded that the proposed architecture is in an experimental state and for this reason there is no need to implement a full-featured system, but it is instead essential to highlight the basic components that are needed in order for it to correctly work. Therefore, even though multiple high-level options can be considered when defining a multi-tenant system, such as the introduction of an hypervisor and the assumption that each client will have its own control logic to manage the assigned piece of network, in this case the selected approach was to utilize the centralized SDN Controller as an entry point for all tenants. As a consequence, the controller was programmed in order to expose a simple northbound API, reachable from all clients, through which they are able to request a specific set of nodes (a *slice*) inside the network. This interface is represented by a new CoAP resource, called *se* (slicing engine), that supports GET messages for retrieving the list of available nodes and PUT messages for requesting a new slice (by providing the previously obtained IPv6 addresses of preferred nodes).

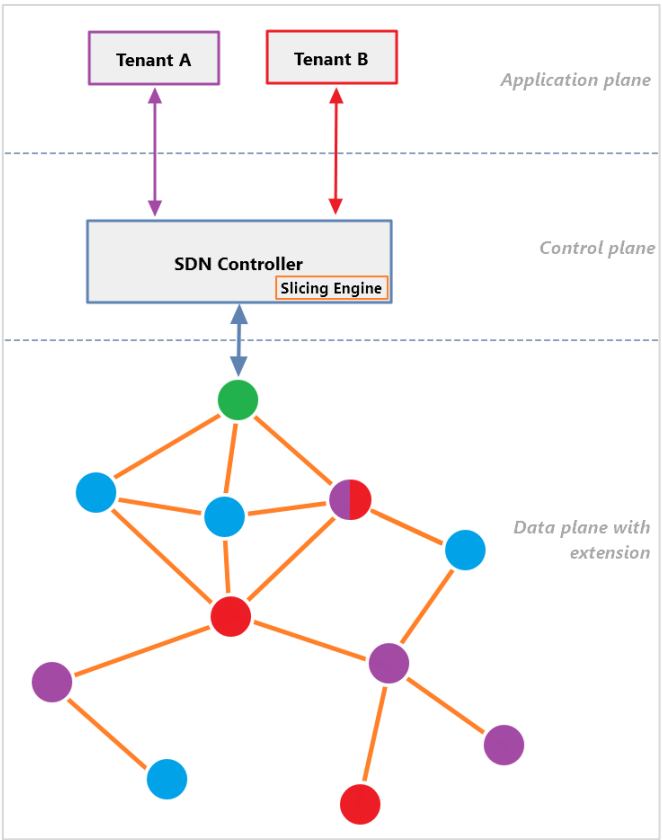


Fig. 6.1 – Example of the proposed architecture. The tenant A and B's applications will ask to the SDN Controller for two different slices through the CoAP "se" resource. As it can be seen, one of the nodes will also be virtually shared among A and B.

Once a slice is registered, the corresponding tenant will typically ask for services to all nodes belonging to that set. Since the links that must be utilized are shared between all clients, and considering their limited bandwidth (due to the already explained reasons), some expedient should be utilized in order to avoid network congestion and overall reduction of the quality of the service provided to all clients. From the point of view of IP (both v4 and v6), this type of communication can be translated into the notion of *multicast.* Indeed, a multicast address represents an identifier for a group of interfaces inside the network, which are in turn belonging to devices intended to provide a specific service. When a client sends a message to a multicast address, all the devices that are part of that group will receive it and consequently be able to answer. For this reason, the slice engine provided by the SDN Controller will also assign to each slice a specific multicast address and communicate it to the interested tenant.

6.1.2 The need of a virtual backbone

After the implementation of a northbound API, the second step is to manage the flow tables in order to correctly deliver messages to all nodes appertaining to each slice. Since the wireless mechanism is by definition a *broadcast* medium, in order to achieve the best performances, such type of communication should be preferred with regards to a *unicast* solution when forwarding packets from a tenant to its slice. Therefore, to achieve an efficient routing, some nodes will need to broadcast (at layer 2) the message, while others will simply need to drop it, based on a specific reasoning. Indeed, if all sensors belonging to the network participate every time to the transmission of multicast messages, this will imply a waste of computational capabilities and energy resources, and therefore it should be avoided. This brings to the necessity of defining a so-called *virtual backbone*, which represents a set of nodes that are able to cover an entire network in the best way possible.

Being the network topology seen (on the SDN Controller's side) as a graph, it is possible to identify a virtual backbone for the entire system by constructing a **Connected Dominating Set** (CDS)[23]. From the graph theory, a set of nodes $C \subseteq V$, where V represents the set of vertices of a graph G, is a dominating set of G if every node in V, except the ones of C, has at least one neighbor in C. This can be also a *connected* dominating set if (and only if) the subgraph of G induced by C is itself connected.

6.2 Defining a minimum CDS

In general, one may want to find a minimum Connected Dominating Set, so to include as less nodes as possible and therefore avoid the unnecessary waste of resources. Unfortunately, this problem is known to be hard to solve in a computational manner (NP-hard) and for this reason only approximated algorithms exist[24] in literature. This implied the need of devising a possible approach to the definition of a minimum CDS and consequent optimization of the slicing procedure.

6.2.1 Minimum Steiner Tree

Taking a step backward and starting from the initial concept of covering the entire slice with the smallest number of nodes, an analogy could be made by considering a tourist that wants to visit all points of interest inside a city, in the least time possible. This is known as the "Traveling tourist problem"[24] and takes a more formal shape in the definition of a minimum Steiner Tree. The latter is represented by simply a binary tree, call it T, that covers all required vertices inside a graph G, while having the minimum cost if summing up the weights of all edges connecting those nodes. This can be easily seen as a sub-class of CDS, in which the set is constrained to be a binary tree (each node has only one parent and a maximum of two children). Considering the Steiner Tree problem, a first algorithm was therefore defined in order to provide an approximated solution to this issue.

Algorithm A – Slice covering set

Before introducing the algorithm that actually computes a Steiner Tree for the given set of preferred nodes, an auxiliary procedure is defined in order to correctly identify the connections between nodes in the slice. More in detail, the presented algorithm builds a connected set between the nodes belonging *only* to the specific network slice. Considering each node u in the preferred set P (the vertices that must be visited), this procedure computes the *weighted shortest path* between u and each other preferred vertex in P. The result is a set D of paths from each node of the slice to each other node of the same slice.

The set *P* will always contain the Sink node, which for simplicity is assumed to be known. This is based on the idea[25] of building a Minimum Spanning Tree, which is basically a set of edges that connects all nodes in *P* without cycles and with the minimum cost of the total edge weight, in order to be successively capable of constructing an optimal connected dominating set from it. The parameters that will be utilized are hereby described:

- **G(V, E, w):** network graph (V is the set of vertices, E the set of edges, w the weight);
- **P:** set of preferred vertices (the ones belonging to the slice);
- **path(v, u):** path from v to u (all vertices to traverse and relative link weights);
- **D:** Set of paths.

Algorithm A

Input: G(V, E, w), P

Output: D

```
1.        D ← {}
2.        While (P is not empty) do
2.1.        Take v ∈ P
2.2.        P ← P \ v
2.3.        For each u ∈ P do
2.3.1.        path(v, u) = weighted shortest path from v to u
2.3.2.        If path(v, u) exists then
2.3.2.1.            D ← D ∪ path(v,u)
2.3.3.        End if
2.4.        End for
3.        End while
```

To better understand what this algorithm does, a graphical example is reported at the end of this paragraph. In these pictures, the blue nodes represent the vertices belonging to the graph (hence, to *V*), while the green nodes depict the ones belonging to the slice (i.e. to the preferred set *P*). Each wireless link between two nodes has a weight associated, that in the considered SDWSN case is represented by the ETX value computed locally by every device. It is indeed important to highlight the fact that in this and successive

procedures the network graph is considered (for simplicity) to be *undirected*, therefore not properly depicting the reality, in which wireless links are asymmetric and can have different costs if traversed in one or the opposite way. To avoid this issue, a simple solution can be applied to this first algorithm so to compute both paths for each couple of nodes and store them in *D*. This will imply a practical correctness also for all subsequent algorithms and represents what was done in the real implementation.

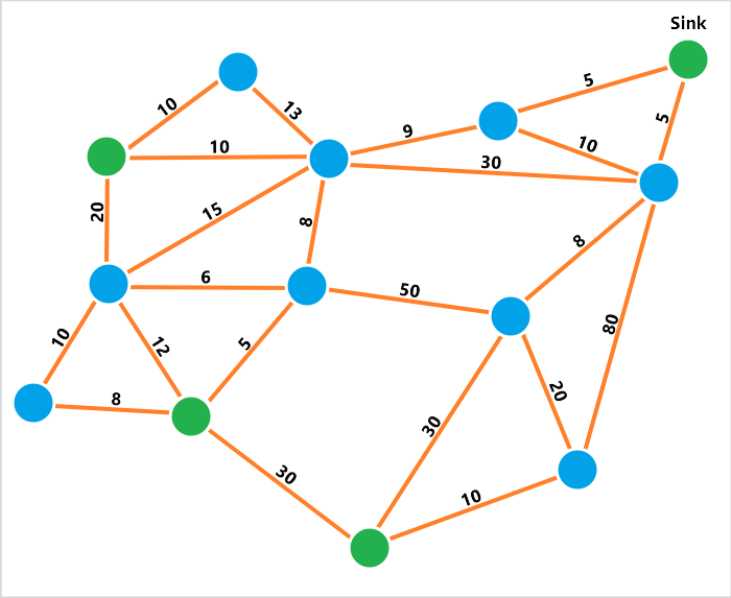


Fig. 6.2 – Example input for the algorithm A

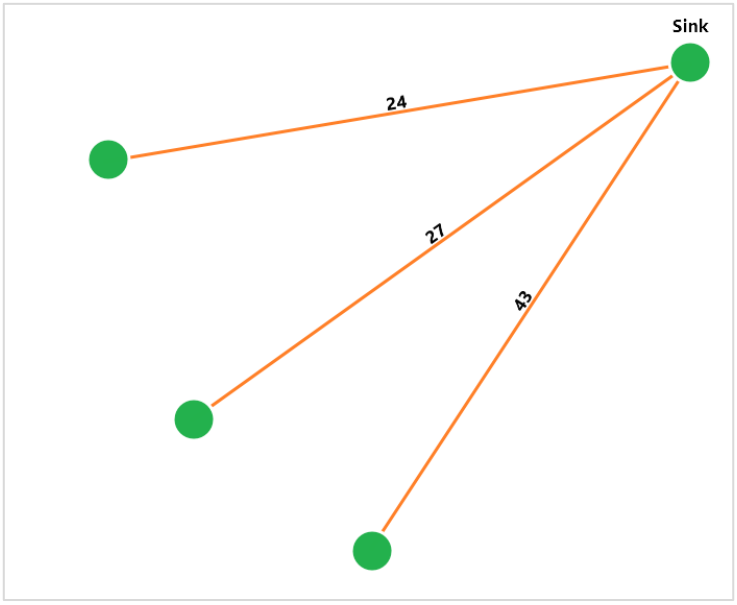


Fig. 6.3 – First run of the loop. The displayed connections are not showing all nodes that are present in each path(u, v), but only the two extremes, for convenience.

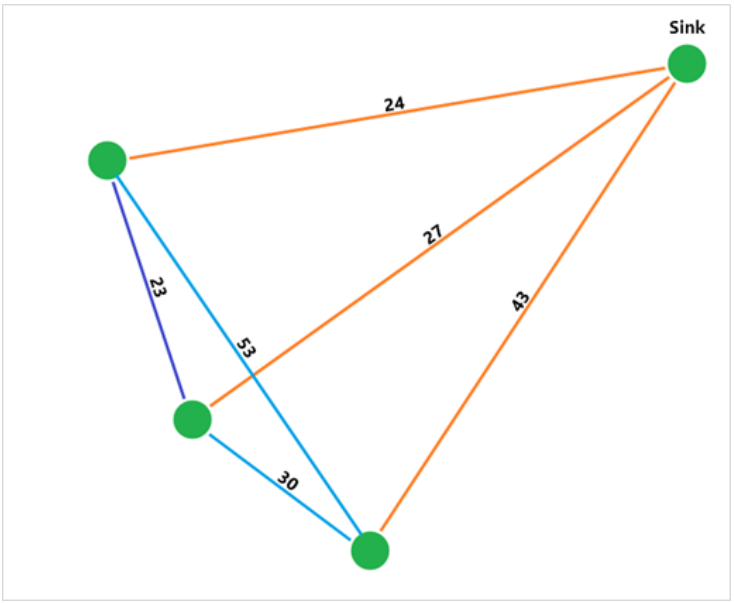


Fig. 6.4 – Final result

Algorithm B – Approximated Minimum Steiner Tree

This second algorithm generates a Steiner connected dominating set by building a tree starting from the Sink node and covering all nodes in the slice. In particular, first of all

the *Sink* is set as root of the tree *T*, removing it from the preferred set *P*. An auxiliary set *N* is also built in order to keep track of nodes in the network that are not in the slice (not belonging to *P*). While *P* is not empty, the procedure takes a node *v* belonging to the slice and which is present in *T* that provides the minimum (total) cost to reach another node *u* not yet present in *T*, but belonging to *P*. At this point, for each node *n* in the path between *v* and *u*, it is added to *T* (if not already present), selecting as its parent the previous node in the path (and so on, also removing it from *P* if it is a preferred node). This algorithm assumes the network described by *G(V, E, w)* to be completely connected. Summarizing, the main parameters are:

- **G(V, E, w):** network graph (V is the set of vertices, E the set of edges, w the weight);
- **P:** set of preferred vertices (the ones belonging to the slice);
- **path(u, v):** ordered list of nodes defining the path from u to v, with link weights;
- **D:** set of paths path(u,v);
- **T:** Steiner tree covering all the nodes in the slice;
- **Sink:** the sink node, known;
- **N:** set of nodes in the graph that are not the preferred ones (not belonging to the slice).

Algorithm B

Input: G(V, E, w), P

Output: T

```
1.        D ← Algorithm_A (G(V, E, w), P)
2.        Set Sink as the root of T
3.        N ← V \ P
4.        P ← P \ Sink
5.        While (P is not empty) do
5.1.          Select v ∈ T \ N and u ∈ P \ T such that the cost of path(v, u) is minimum
5.2.          For each n belonging to path(v, u) do
5.2.1.            If n != v  and n ∉ T then
5.2.1.1.              Set n as children of v in T
5.2.1.2.              If n ∈ P then
5.2.1.2.1.                P ← P \ n
5.2.1.3.              End if
5.2.2.            End if
5.2.3.            v ← n
5.3.          End for
6.        End while
```

A graphical example is reported also in this case, for a better understanding of what happens in each point of the algorithm. The considered topology and slice are exactly the same presented in the previous illustration, only addition being that here red nodes indicate the ones forming the Steiner tree *T* and the preferred nodes belonging to it are also marked with a special name (namely, Sink, P1, P2, and P3).

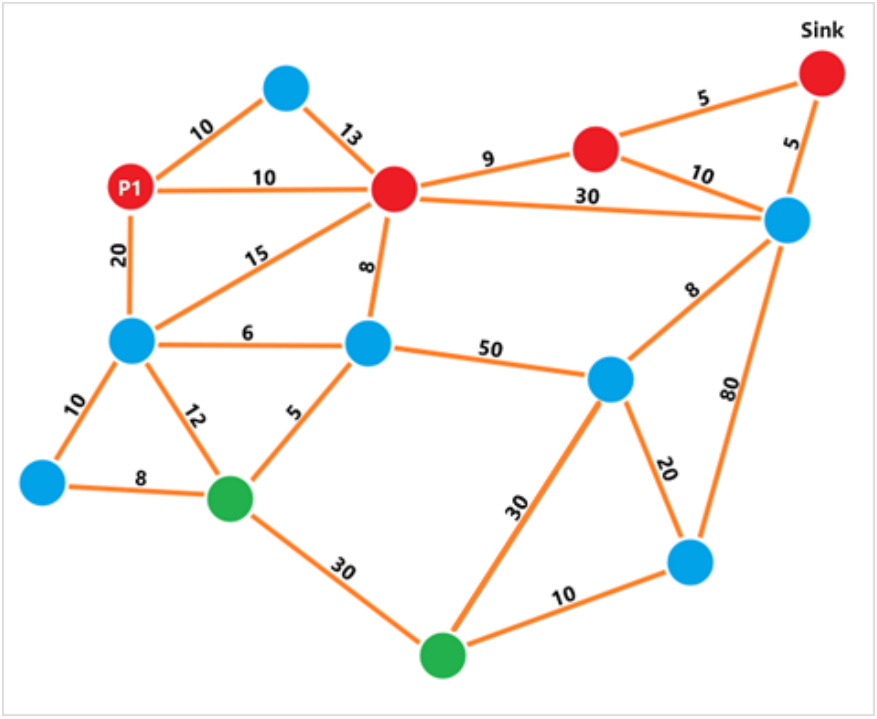


Fig. 6.5 – First run of the while loop in algorithm B. This considers the set D obtained in picture 6.4. Now P1 and Sink are both in T, so the minimum cost path to be selected at next step will be P1 – P2, hence adding two children to the parent of P1. After P2 is also in T, another path that will be selected is P1 – P2, which has indeed the minimum cost in order to correctly reach also P3.

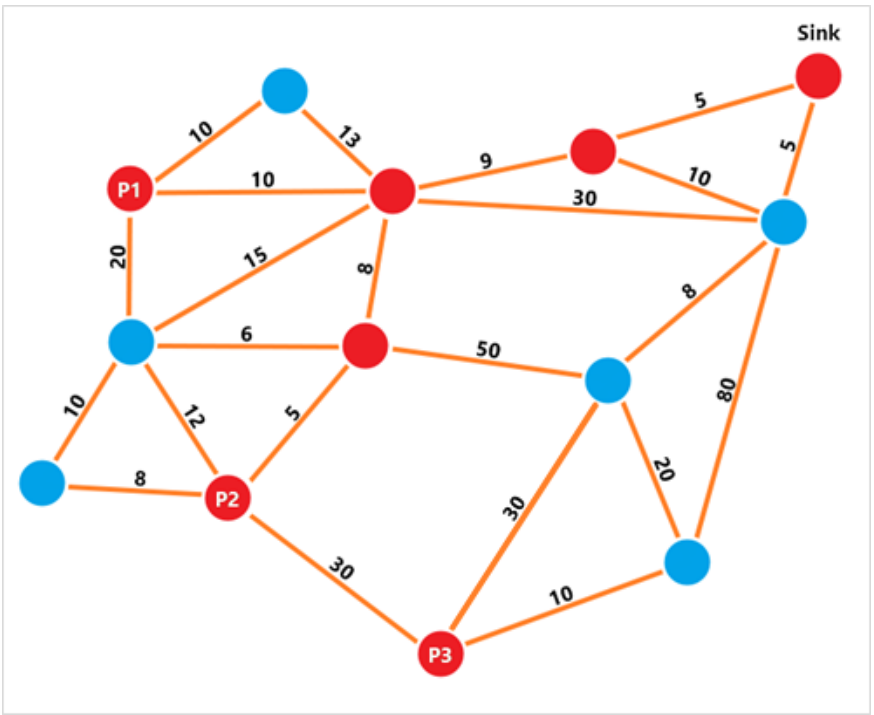


Fig. 6.6 – Steiner tree obtained at the end of the procedure

6.2.2 A fault-tolerant CDS

Once again, it must be kept in mind that the concept of Connected Dominating Set is utilized in this case to build a virtual backbone for covering *only* the nodes belonging to a specific slice, instead of the whole network. This was previously identified by the Traveling Tourist Problem and two algorithms were proposed for building an approximated Minimum Steiner Tree in order to satisfy this purpose. In particular, it can be noticed that the obtained set is able to connect all nodes *inside the slice* by building a binary tree through the selection of a small number of other ones, that simply act as relays. Since each preferred node is covered by at least one neighbor belonging to the tree, this type of CDS can be classified as *1-connected* and *1-fold*. More in general, a CDS C is said to be k-connected, m-fold (or (k, m)-CDS), with $m \geq k$, if every node in the network that does not belong to C has at least m neighbors in the CDS and the subgraph generated by C is k-connected. Applied to what concerns the coverage of only nodes belonging to a specific slice, this concept can be translated into the definition of a CDS C such that each node in P is connected to at least m neighbors belonging to C.

Being wireless sensor networks subject to interferences and other communication issues related to the constrained nature of sensor nodes, it is easy to understand that the proposed algorithms are able to build a CDS which is totally not fault tolerant. That is, if a link between two nodes in the set is broken, its whole coverage could be compromised. In order to avoid such problem, a new algorithm must be proposed so to build a (*1,2*)-CDS and ensure at least tolerance from one fault per node.

Algorithm C

The proposed algorithm aims at generating a *1*-connected, *2*-fold CDS by following the following steps:

1. Generate a *1*-connected, *1*-fold CDS by utilizing Algorithm B;

2. starting from the root of the obtained Steiner Tree (i.e. the Sink), try to remove the connections between parent and children in the CDS. If this removal compromises the reachability of nodes in P, roll back to a previous state;
3. run again the Algorithm B on the new obtained subgraph and perform union with the previous CDS.

More in detail, an auxiliary set *C* is built for marking all the explored nodes that are belonging to the original CDS, while the cycle will run until all connections have been analyzed. In order to check whether or not the deletion of a particular parent-children relationship affects the reachability of (at least one) nodes in *P*, the cardinality of the set *D* returned from algorithm A applied on *G(V, E', w)* is confronted with the expected one, which is exactly the combination of |P| nodes over 2 spots. This ensures the possibility of finding a suboptimal solution (i.e. provide fault tolerance for the highest number of slice nodes possible) in any case, even when there are some links that cannot be removed due to a consequent lack of connectivity in the whole graph. Below are listed the main parameters:

- **G(V, E, w):** network graph (V is the set of vertices, E the set of edges, w the weight);
- **P:** set of preferred vertices (the ones belonging to the slice);
- **2CDS:** *1*-connected, *2*-fold CDS covering all the nodes in the slice.

Algorithm C

Input: G(V, E, w), P

Output: 2CDS

1. 2CDS ← Algorithm_B (G(V, E, w), P)
2. C ←{Sink}
3. E' ← E
4. **While** |C| < |2CDS| **do**
4.1. Take the next $v \in C$
4.2. **For each** children u of v in 2CDS **do**
4.2.1. C ← C ∪ u
4.2.2. E' ← E' \ {(v, u)}
4.2.3. **If** |Algorithm_A(G(V, E', w), P)| < $\binom{|P|}{2}$ **then**
4.2.3.1. E' ← E' ∪ {(v, u)}
4.2.4. **End if**
4.3. **End for**
5. **End while**
6. 2CDS ← 2CDS ∪ Algorithm_B(G(V, E', w), P)

Continuing the previous example, a detailed graphic view of what happens in this algorithm is hereby presented. In particular:

- Nodes in 2CDS belonging to the slice are named Sink, P1, P2 and P3;
- **Red nodes** are the ones belonging to 2CDS at step 1 of the algorithm;
- **Violet nodes** are the ones added to 2CDS when performing the union at step 3;
- **Blue nodes** are the ones excluded;
- In the final result, connections between nodes are highlighted to show the respective sub-tree.

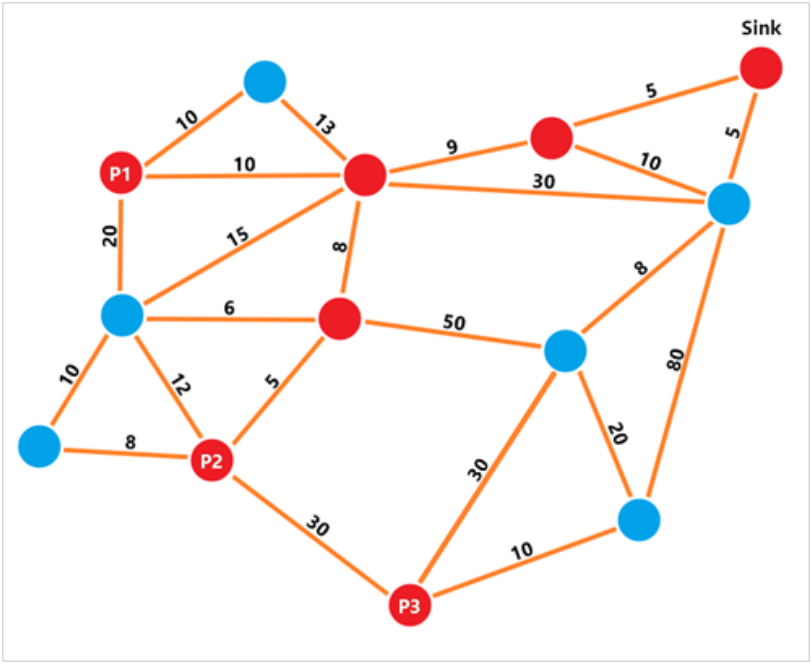


Fig. 6.7 – Initial CDS obtained from algorithm B

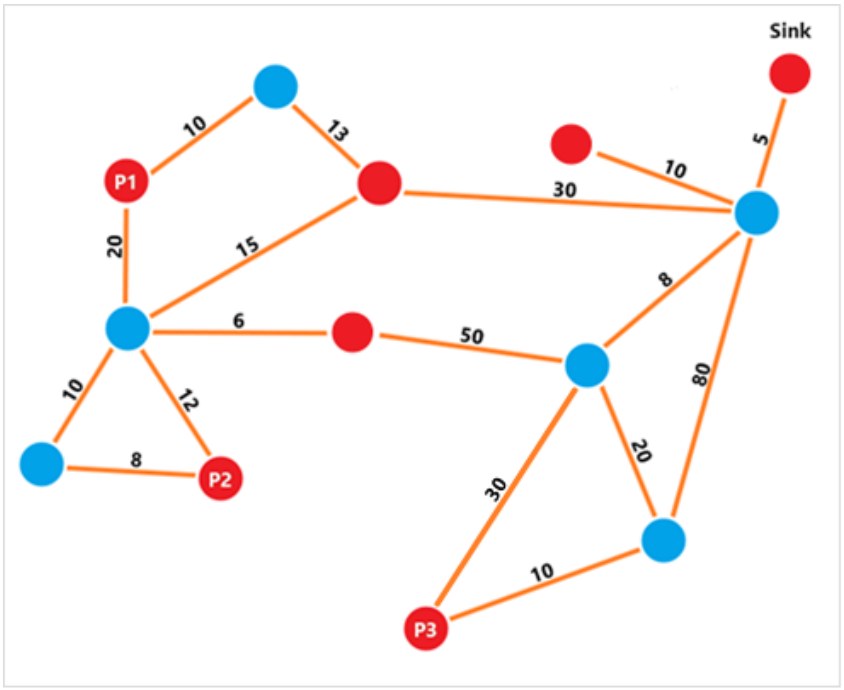


Fig. 6.8 – New G(V, E', w) obtained by removing links as explained. It can be noticed that, if for example there was no other link connecting the Sink to the network, the algorithm would have rolled back when trying to delete the topmost connection, since otherwise the graph wouldn't be connected anymore, thus ensuring 2-fold connectivity to all other nodes in the slice, except for the Sink.

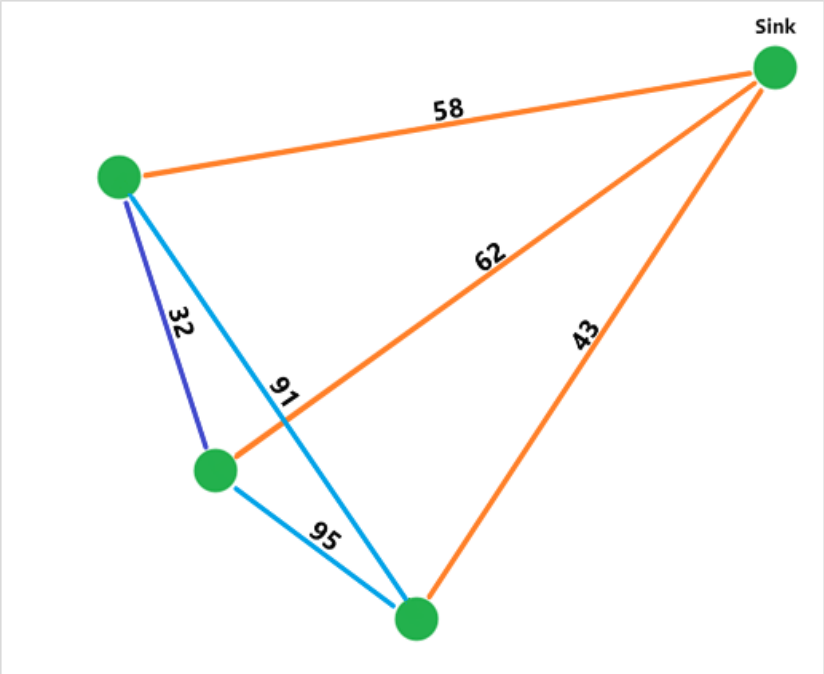


Fig. 6.9 – The new (simplified) D obtained from algorithm A

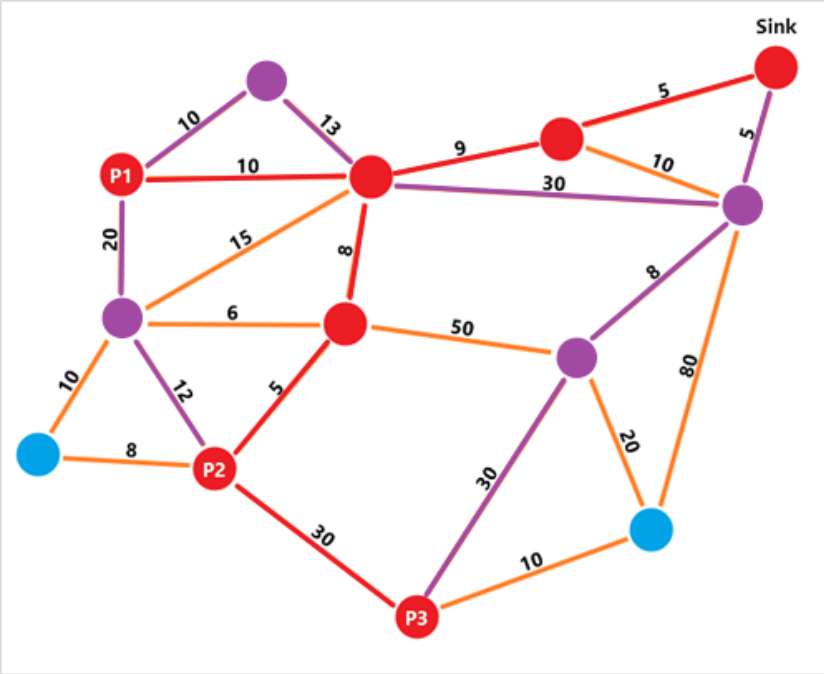


*Fig. 6.10 – Final result. As it can be noticed, all preferred nodes are now covered by at least **two** neighbors, meaning that the CDS is redundant and provides a basic fault tolerance.*

6.2.3 Implementation

After building the CDS by using Algorithm B or C, the SDN Controller will be able to install forwarding rules inside each node belonging to the set. In this way, the SDWSN will also be able to provide a multicast mechanism from the tenant to all nodes belonging to his slice. Since layer 2 *broadcast* is utilized for transmission, also all excluded nodes that are neighbor with those belonging to the CDS will need to have a rule installed for dropping the packet once received, so to avoid unnecessary flooding. More in detail, two communication cases are identified:

- **Application → Multicast group**
  One client application (implemented as explained in 6.1.1) will have its own IPv6 address and will ask the controller to create a slice by selecting some preferred nodes by utilizing their IPv6 addresses. After this step, the SDN controller will assign to the built CDS (which is different for each slice) a multicast IPv6 address and all messages sent from the application to this address will be forwarded in the network using the previously cited rules, therefore reaching all the nodes in the slice in a fault-tolerant way.
- **Application → Single node**
  If the tenant wishes to talk to only one of the nodes belonging to a slice, forwarding in the network will be performed normally (without utilizing the CDS) and the response to the application will for sure travel through the Sink because of its IPv6 address, which will be recognized as off-link and hence forwarded using the RPL upward route (this solves also the **Single node → Application** case).

It is hereby important to highlight an issue that arises when dealing with flooding techniques, which is the possibility of loops when broadcasting packets between multiple nodes, that could lead to the so-called *broadcast storms*. In order to avoid this problem, which would cause possible network congestions and performance losses (together with major energy consumption), a solution must be implemented. In traditional SDN systems, this issue is simply resolved by forwarding the packet on all ports of the SDN Switch except the one from which it was received, but this does obviously not represent an option

when utilizing a wireless medium. For this reason, three possible approaches were elaborated for the SDWSN case and are hereby proposed.

In order to avoid loops when forwarding multicast messages, a first implementation could consider the installation of rules in each node belonging to the CDS, so that the packet is accepted and broadcasted if and only if it is coming from one of the node's parents in the constructed virtual backbone (one rule for each parent, retransmitting only once). This solution is very straightforward and doesn't introduce particular complexity, but it would imply an ordering when distributing the multicast message from the application to the selected slice, hence causing additional delays and reducing the overall communication reliability.

A more precise solution should instead consider the possibility of receiving a multicast message from *any* node belonging to the CDS (hence, multiple senders), without the implicit definition of particular hierarchies. In this case, in order to avoid broadcasting the same packet multiple times, its *sequence number* must be utilized. Therefore, a data structure must be maintained in each SDN Node belonging to the virtual backbone, so to store the information about every packet's *sequence number* and utilize it to properly manage the forwarding of such messages. More precisely, just one rule will be installed in the node's flow table, that will access this data structure and check whether or not it is needed to forward the received packet (i.e. if sequence number is less then or equal to the current one, drop the packet). This implies of course the introduction of additional complexity to the local control agent of each node, but represents a more reliable solution (with the introduction of some redundancy) and allows for reception of multicast messages in a faster way.

Since both the first and second proposed approaches have critical downsides, a trade-off between complexity and reliability can be found by avoiding the utilization of sequence numbers and, at the same time, installing multiple rules in the node's flow table to match the possibility of receiving the same packet from different nodes, even if they are not selected parents in the CDS. This can be done by exploiting the controller's global knowledge and therefore superimposing a Directed Acyclic Graph (DAG) to the constructed CDS, so to accept the situation in which *some* of the node's neighbors (belonging to the backbone) can send the same message and hence it will be forwarded

multiple times. This obviously introduces redundancy in packet transmission, with some overhead of messages, but allows for faster delivery and higher reliability with regards to the first proposed solution, also allowing for smaller complexity with respect to the second one.

Between the three presented approaches, the first one was selected when implementing the actual code inside the SDN Controller. This means that all obtained results that will be shown in next chapters represent the worst-case scenario and can be therefore improved by utilizing a different solution.

More in detail, all the cited rules will be installed exactly after the slice is accepted by the Controller and a multicast address is generated and communicated to the tenant. Furthermore, any other node that does not belong to the slice will of course ask for instructions to the SDN Controller as soon as a packet belonging to that set is received. To deal with this case, it is obvious that the Controller will need to store all created slices and hence be able to examine the node and successively install a *drop* rule inside its flow table. Additionally, it is possible to notice that the native border router solution introduced in 5.2.2 is here very useful due to the high number of rules that will be needed inside the SDN Sink Node for maintaining the whole network connectivity, together with slice information (since all traffic from and for the slice passes through this device).

6.3 Slice maintenance and repair

Being the wireless medium not stable and sometimes asymmetric, possible changes in the network topology must be taken into account when maintaining an optimal CDS. In order to keep the previously built set correctly working, a maintenance procedure must be implemented. In this phase, fixing of links can be achieved with two possible approaches:

- **Local repair**, in which a single node executes some local algorithms in order to resolve a fault locally;
- **Global repair**, where a complete knowledge of the entire network topology is needed to perform maintenance.

Obviously, since the proposed architecture implements a central SDN Controller, the selected approach is represented by a global repair. Indeed, being the SDN architecture centralized, a repair algorithm can be run at the controller side whenever the considered CDS is no more optimal with regards to the updated topology graph. This could happen, for example, if a node crashes and is unavailable for a certain period of time, if a link between two neighbors is completely removed, or also when the cost of traversing one or more paths changes. For that reason, a maintenance algorithm must be proposed.

Local Repair

Even though a local repair in its true meaning is not considered, a first possibility could be to manage a broken or updated link by simply recomputing the shortest path between the two nodes at its extremities. This could not be optimal if looking at the entire topology, but will for sure avoid some communication overhead due to less control messages traveling the network in the case in which such procedure is applied with respect to a global repair. Therefore, the proposed algorithm performs a local maintenance procedure on the provided CDS by taking into account the set *P* of paths to repair. For each route to restore, the procedure first removes all the comprehended nodes from the original CDS and then computes a new shortest path between the two end nodes (by considering the updated topology), inserting the interested nodes inside the repaired CDS. If at least one of the paths cannot be restored, this algorithm must stop and exit by returning an empty set, thus signaling the impossibility of a local maintenance. Otherwise, the repaired CDS will be returned by first merging it to the original one (from which some useless nodes were striped out during the previous steps). The utilized parameters are:

- **G(V, E, w):** the updated network graph;
- **CDS:** the connected dominating set to repair;
- **path(v, u):** path from v to u (all vertices to traverse and relative link weights);
- **P:** set of paths to repair;
- **rCDS:** repaired CDS.

Local Repair

Input: G(V, E, w), CDS, P

Output: rCDS

```
1.      rCDS ← {}
2.      While (P is not empty) do
3.          Take path(v, u) ∈ P
3.1.        P ← P \ path(v, u)
3.2.        CDS ← CDS \ {n | n ∈ path(v, u)}
3.3.        path(v, u) ← shortest path from v to u on the updated graph G(V, E, w)
3.4.        If path(v, u) does not exist then
3.4.1.          Exit the algorithm returning an empty rCDS (cannot repair locally)
3.5.        Else
3.5.1.          rCDS ← rCDS ∪ {n | n ∈ path(v, u)}
3.6.        End if
4.      End while
5.      rCDS ← rCDS ∪ CDS
```

Also in this case, a graphical example is reported in order to better understand what this algorithm does. In particular:

- **Green nodes** are the ones belonging to the slice (and to the CDS);
- **Red nodes** are the ones belonging to the CDS, for connecting green ones;
- **Blue nodes** are excluded from the CDS;
- Weights are associated to each link (ETX);
- For simplicity, this example considers a *1*-fold CDS.

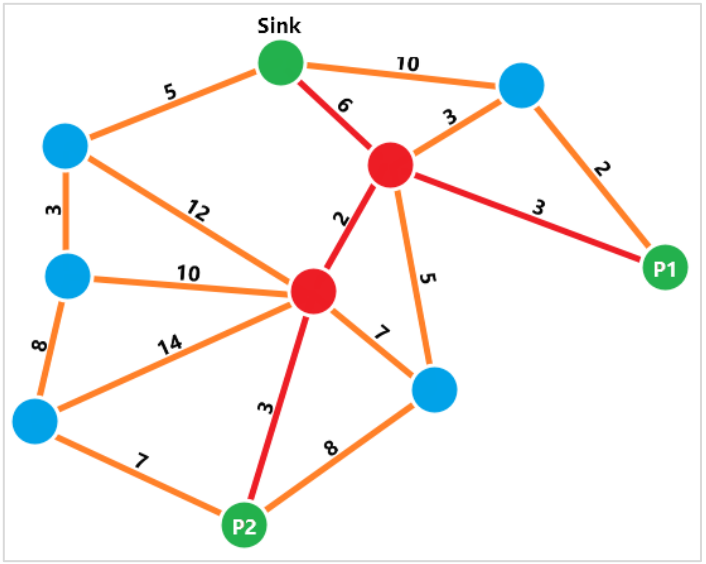


Fig. 6.11 – Starting topology

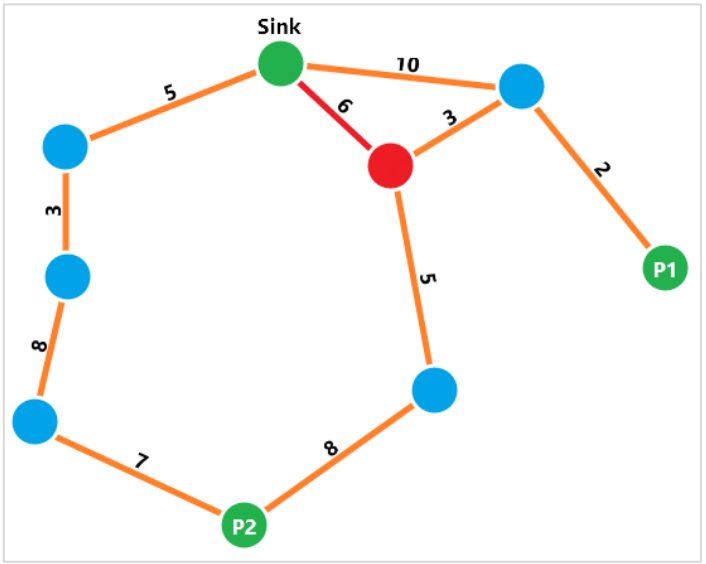


Fig. 6.12 – The new topology retrieved after some time

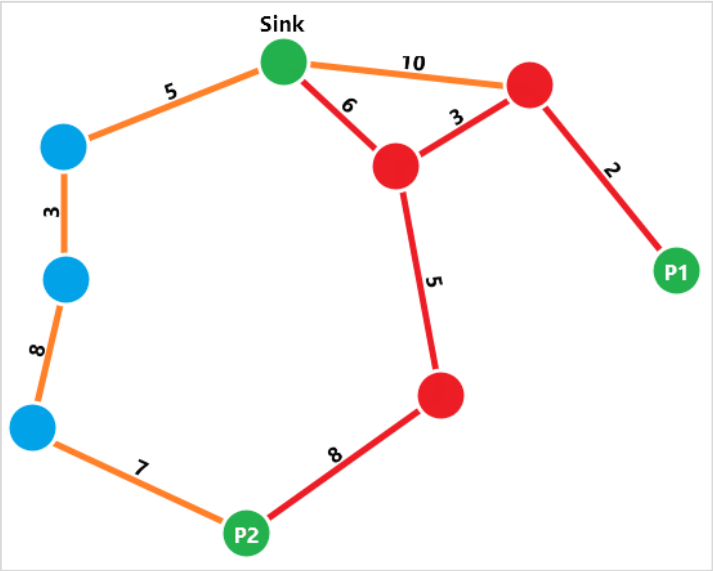


Fig. 6.13 – The repaired CDS obtained from a local repair

Maintenance algorithm

The actual maintenance algorithm considers the modified network topology and performs an update of the CDS only if it is needed. Specifically, the new set that would be obtained by applying a *global* repair procedure (thus, recomputing it completely from scratch) is saved and compared to the dominating set that would be instead obtained by utilizing the *local* repair algorithm previously described. To achieve this, first of all a subgraph containing all differences between the new and old network topology is computed, then checking if the CDS is affected or not by these changes. If there are at least one or more nodes (or links) of the CDS affected by the update, the set of modified paths is computed and utilized for the local repair procedure. At this point, if no local procedure was possible (i.e. the returned set is empty), the algorithm stops by providing as result the CDS obtained by global repair. Otherwise, a comparison is made between the two obtained sets in terms of number of nodes in which is needed to install or remove flow table rules. In this comparison, a threshold value is applied in order to *give priority* to the global repair procedure, that because it will provide most of the times a better performance with regards to the local one. As an example, if the threshold value is equal to 2, it means that the algorithm will tolerate an overhead of maximum two nodes from the side of a global

repair, thus selecting this option if for instance the number of nodes to be updated is 5 versus 4 of the local repair procedure (5 – 2 = 3 < 4). The considered parameters are:

- **G(V, E, w):** old network graph;
- **G(V', E', w'):** updated network graph;
- **CDS:** the considered CDS;
- **rCDS:** repaired or updated CDS (if needed);
- **t:** threshold value for the maximum overhead in terms of #nodes to update;
- **P:** set of preferred nodes covered by the CDS;
- **path(v, u):** path from v to u, with associated link costs.

Maintenance Algorithm

Input: G(V, E, w), G(V', E', w'), CDS, P

Output: rCDS

1. G(V'', E'', w'') ← Subgraph made of differences between G(V, E, W) and G(V', E', w')
2. Check if any node or link in CDS is affected by G(V'', E'', w'')
3. **If** no node or link is affected **then**
3.1. Terminate the algorithm as there is no need to perform any maintenance
4. **End if**
5. Global_rCDS ←Algorithm_C(G(V', E', w'), P)
6. D ← Set of paths(v, u) (with v and u in CDS) that are affected by G(V'', E'', w'')
7. Local_rCDS ← Local_Repair(G(V', E', w'), D)
8. **If** Local_rCDS is empty **then**
8.1. rCDS ← Global_rCDS
9. **Else**
9.1. n_global ← number of nodes affected by the global repair procedure
9.2. n_local ← number of nodes affected by the local repair procedure
9.3. **If** (n_global – t) > n_local **then**
9.3.1. rCDS ← Local_rCDS
9.4. **Else**
9.4.1. rCDS ← Global_rCDS
9.5. **End if**
10. **End if**

As it can be noticed, this algorithm takes into account not only all the broken paths, but also modifications in the costs of routes between two nodes in the CDS, thus performing a proper maintenance: update and (if needed) repair.

6.4 Testing

In order to properly test the correct functioning of all described algorithms in the proposed SDWSN scenario, and therefore to prove that the utilization of the SDN paradigm over standard WSNs is worth the additional costs examined in 5.3, multiple experiments were proposed. Before executing those tests, the SDN Controller was enhanced with the additional *se* CoAP resource (also providing the implementation of all mentioned algorithms) and a Client application was also programmed in order to access those functionalities (it simply asks for a set of nodes and then periodically sends multicast messages to it). The complete Java code of both the SDN Controller and the Client are available for free and provided as open source on GitHub[26].

6.4.1 Test plan

As explained, the fundamental reason that brought to the research of a correct implementation of SDN over WSN is essentially represented by the need of supporting a very large number of devices and additionally provide new functionalities that will be for sure useful in IoT scenarios. Unfortunately, the considered testbed is composed of only 21 nodes and therefore it does not represent a fitting scenario for what regards network virtualization and slicing. Since the proposed SDWSN system was successfully tested and working on top of this real-life physical infrastructure, what can be claimed by theoretical assumption is that any new mechanism implemented inside the SDN Controller should not affect its proper functioning. Therefore, the proposed tests were performed by utilizing the Cooja simulator instead of the University's testbed. Below are reported the definitions of each experiment that was performed during this thesis work.

Experiment 1

This test aims to emphasize the differences between the *1*-fold and *2*-fold connected dominating sets, checking whether it is better to lose fault tolerance in order to gain a smaller overhead introduced by the slicing system, depending on the number of nodes inside the network. Two core scenarios are proposed, one that utilizes the basic *1*-fold

technology and another with the *2*-fold one. The network topology is built in Cooja by following a specific criterion, in order to allow replicable experiments (or at least the convergence to the obtained results for a big number of repetitions):

1. Place the SDN Sink Node in a square area, with random position;
2. place the remaining number of SDN Nodes in the same area, with random positions;
3. if the network is not fully connected, discard the topology and repeat from step 1.

The number of SDN Nodes in each topology will be of 50, 100, 150 and 200, hence the square area must be scaled with the square root criterion in order to maintain constant the node density (otherwise, with a linear increment it would be inversely proportional to the number of nodes). Of course, the topology must be the same for both scenarios, together with the selected slice, which will be of size 10, 15, 25 and 35, constructed by selecting respectively IDs from 2 to slice size minus one (node 1 is the Sink). Below are reported the tables defining the core parameters considered in this experiment.

Scenario 1

Parameter	Value(s)
Number of SDN Nodes composing the network	50 – 100 – 150 – 200
Square length	200 – 282 – 346 – 400
Topology update period	10min
Covering technique	1-connected, 1-fold CDS
Dissemination method	Selective broadcast (see 6.2.3)
Number of allowed slices	1
Number of nodes in the slice	10 – 15 – 25 – 35
Number of multicast messages from app to slice	12 (1 every 5 min)
Simulation time	60min

Scenario 2

Parameter	Value(s)
Number of SDN Nodes composing the network	50 – 100 – 150 – 200
Square length	200 – 282 – 346 – 400
Topology update period	10min
Covering technique	1-connected, 2-fold CDS
Dissemination method	Selective broadcast
Number of allowed slices	1
Number of nodes in the slice	10 – 15 – 25 – 35
Number of multicast messages from app to slice	12 (1 every 5 min)
Simulation time	60min

The measures that must be taken into account when performing those experiments are hereby defined:

- **Mean number of Bytes** flowing inside the network *for installing slicing rules*;
- For one multicast from application to slice, the mean number of *bytes being forwarded* inside the network;
- **Mean time** for reaching all nodes in the slice;
- **Mean number of nodes** inside the slice that correctly receive a multicast message.

Experiment 2

This second experiment extends the previous one by comparing the performances of the proposed CDS algorithms with regards to a naïve flooding technique. This considers the same topologies defined in previous test, that must be equal for all scenarios, in which the number of nodes is increased gradually.

Scenario 1 & 2

Exactly the same as previous experiment.

Scenario 3

Parameter	Value(s)
Number of SDN Nodes composing the network	50 – 100 – 150 – 200
	200 – 282 – 346 – 400
Topology update period	10min
Covering technique	/
Dissemination method	Flooding
Number of allowed slices	1
Number of nodes in the slice	10 – 15 – 25 – 35
Number of multicast messages from app to slice	12 (1 every 5 min)
Simulation time	60min

The considered measures are the same of experiment 1, except for the mean number of bytes needed for installing slice rules, which is in this case excluded due to the flooding technique needing only simple *broadcast* rules. It is also needless to say that each one of these measures will have an associated confidence interval, computed from a selected 95% confidence level.

Experiment 3

This last experiment aims to show the working principle of the proposed maintenance algorithm, with particular attention to when the local repair procedure is preferred with regards to the global one. The topology is this time predefined, with a fixed number of nodes and a slice established in advance. After the first run of the CDS algorithm, some of the connecting nodes will be moved away in order to break particular links.

Parameter	Value(s)
Number of SDN Nodes composing the network	20
Topology update period	10min
Covering technique	1-connected, 1-fold CDS
Dissemination method	Selective Broadcast
Overhead threshold value t	2

6.4.2 Experimental results

In this chapter all results obtained by the above described tests are analyzed in detail. Since in both experiment 1 and 2 the same measures are considered in the same scenarios (1 and 2), only the plots regarding the latter test will be displayed for a more compact and clean representation. Moreover, hereby are displayed the reference topologies that were obtained by following the previously proposed criterion.

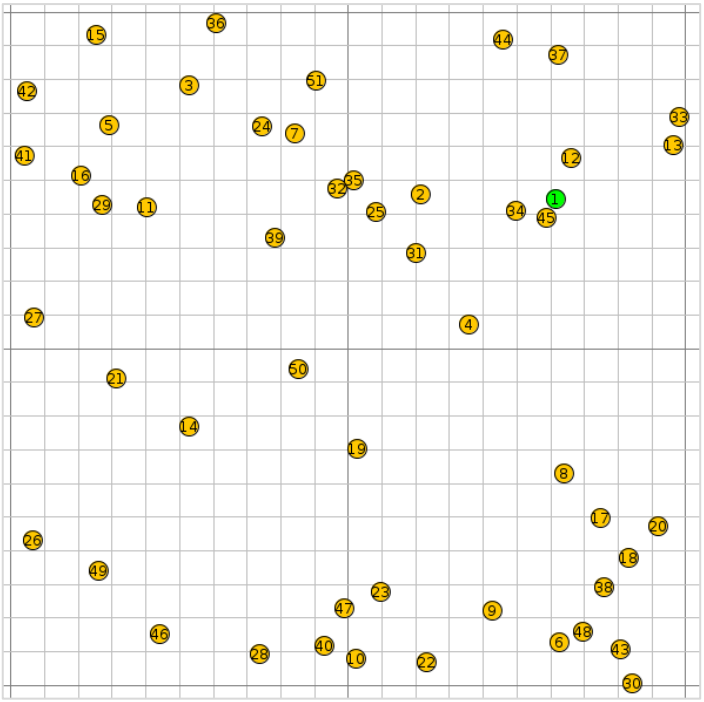


Fig. 6.14 – Topology composed of 50 SDN Nodes in Cooja. The green one is the Sink.

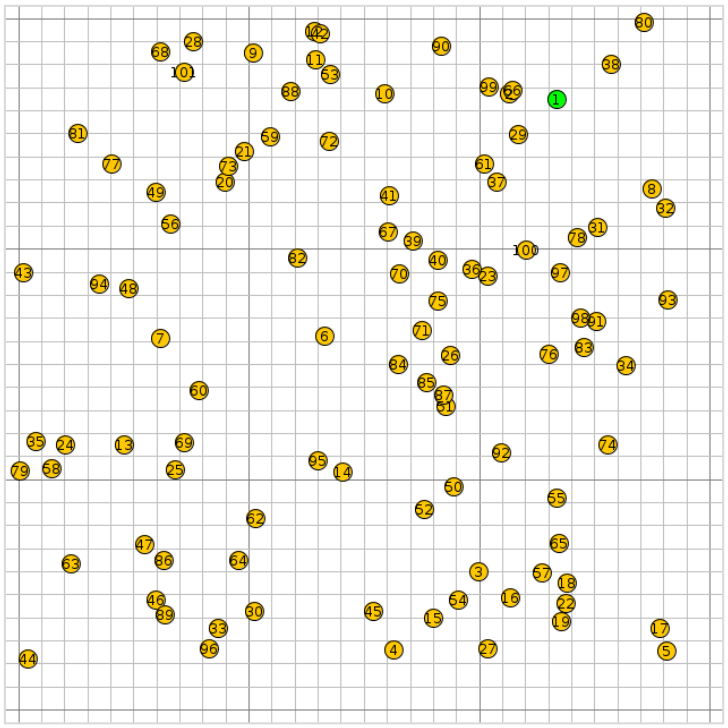


Fig 6.15 – Topology of 100 SDN Nodes

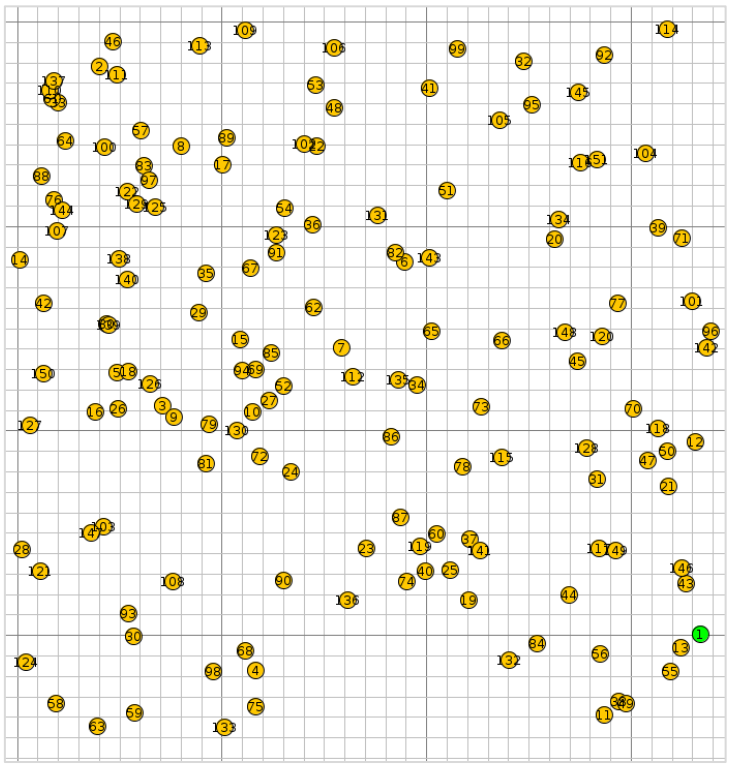


Fig. 6.16 – Topology of 150 SDN Nodes

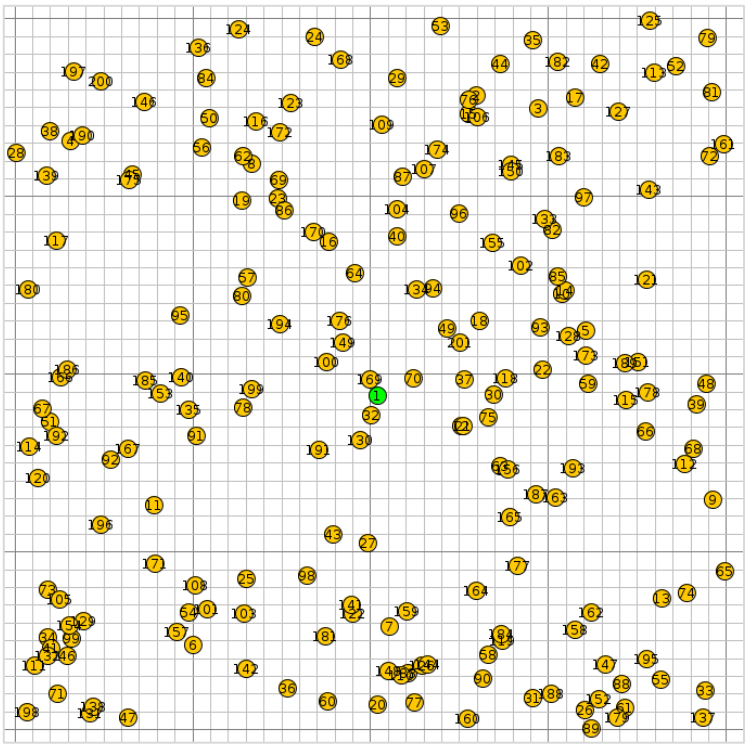
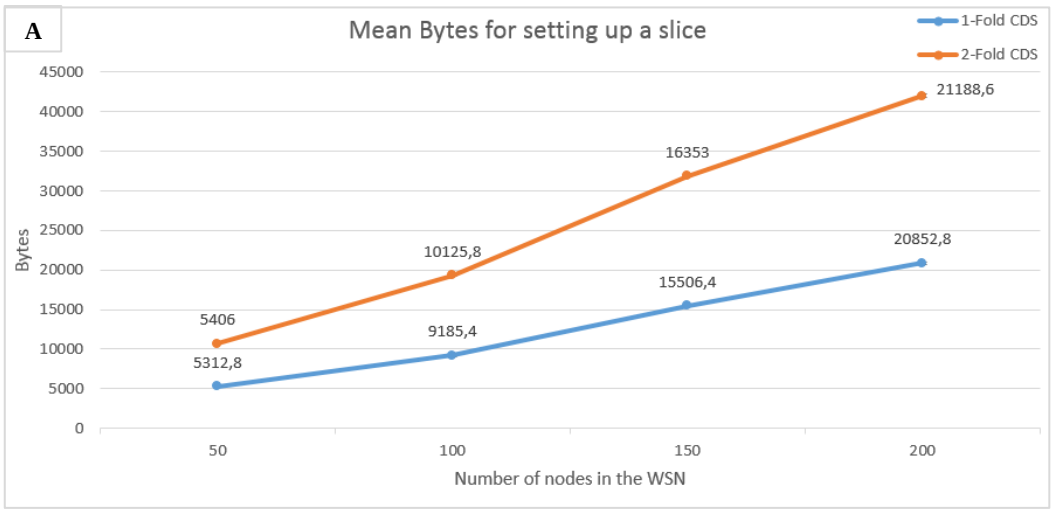


Fig. 6.17 – Topology of 200 SDN Nodes

Experiments 1 & 2

Mean Bytes for slice setup



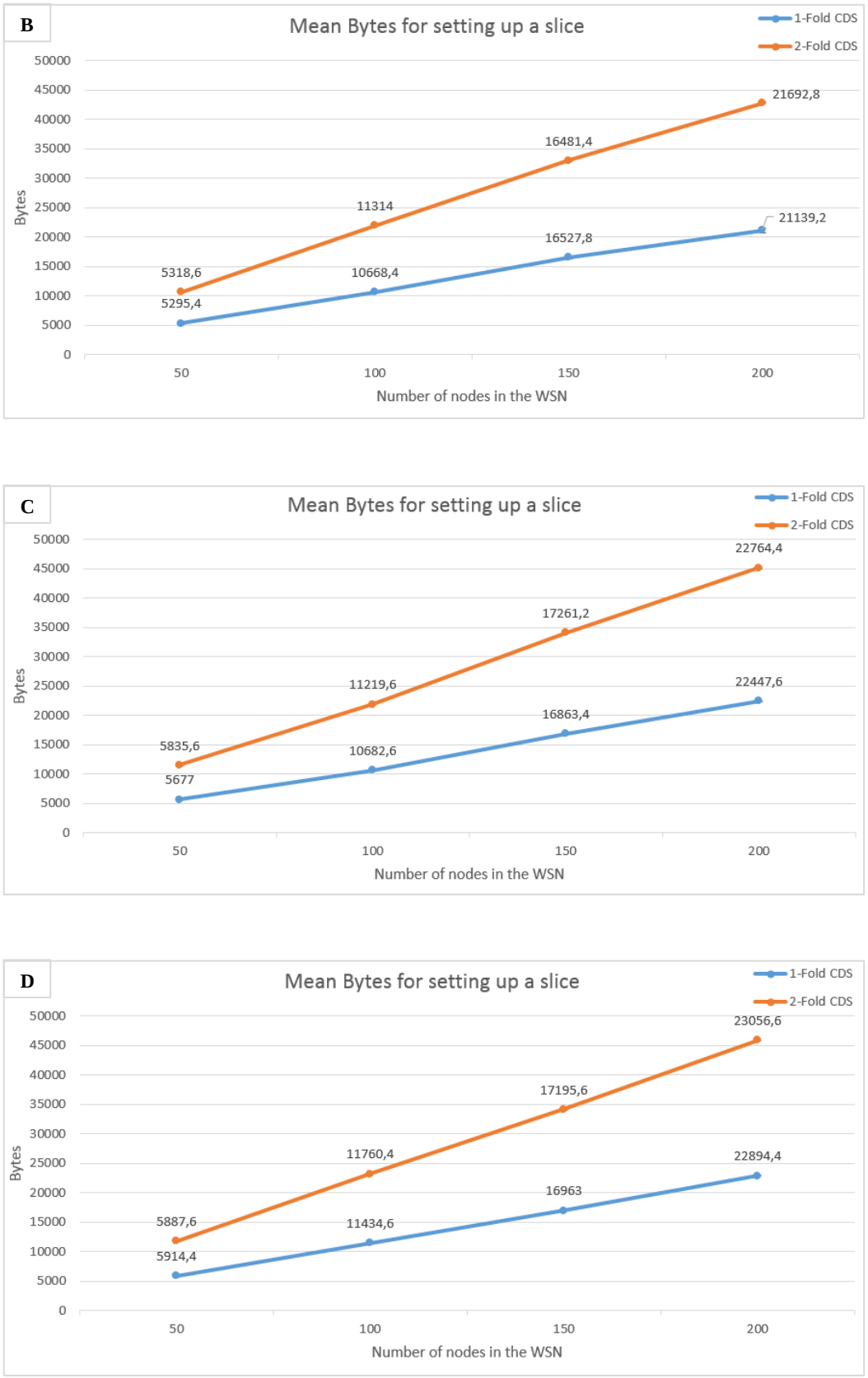
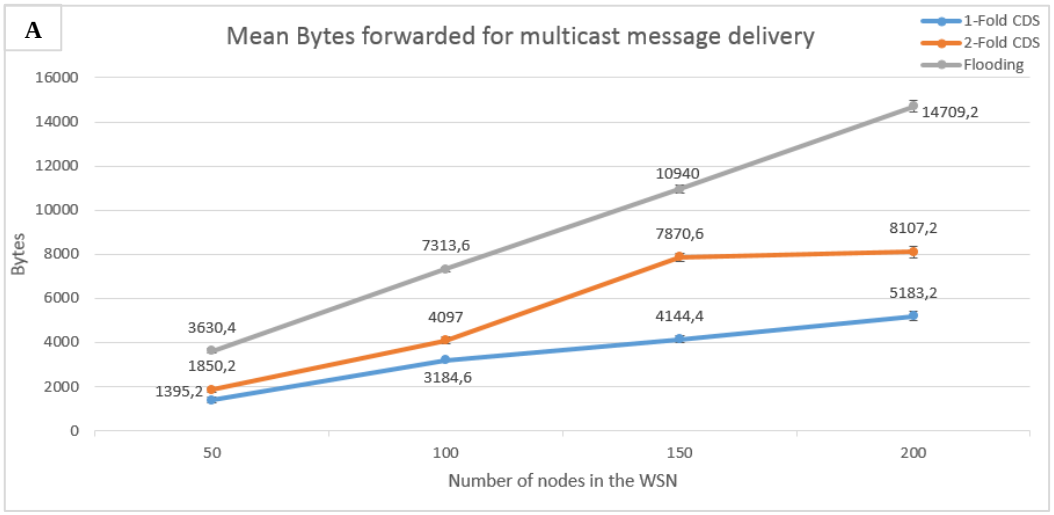


Fig. 6.21 - Mean bytes for setup in the case of slice composed by respectively 10, 15, 25 and 35 nodes (stacked lines).

Concerning the mean bytes forwarded inside the network in order to correctly set up all slicing rules (including *drop* ones eventually needed during the simulation), it can be noticed from the above plots that, as expected, the standard CDS implementation provides overall less overhead with regards to the fault tolerant one in the different cases (even though they are very close). This can be explained by the fact that typically a *2*-fold CDS will select more nodes than the ones chosen by the standard procedure, thus implying the need to install *broadcast* and *drop* rules on more devices inside the network. Indeed, since more nodes will belong to the slice, also more neighbors not belonging to it will inherently receive such broadcast messages, thus requiring the installation of additional *drop* rules with respect to the *1*-fold scenario. Moreover, it can be noticed that both solutions present an almost linear increase of Bytes forwarded for any size of the slice, meaning that they will for sure diverge with very large-scale networks.

Mean Bytes forwarded for multicast message delivery



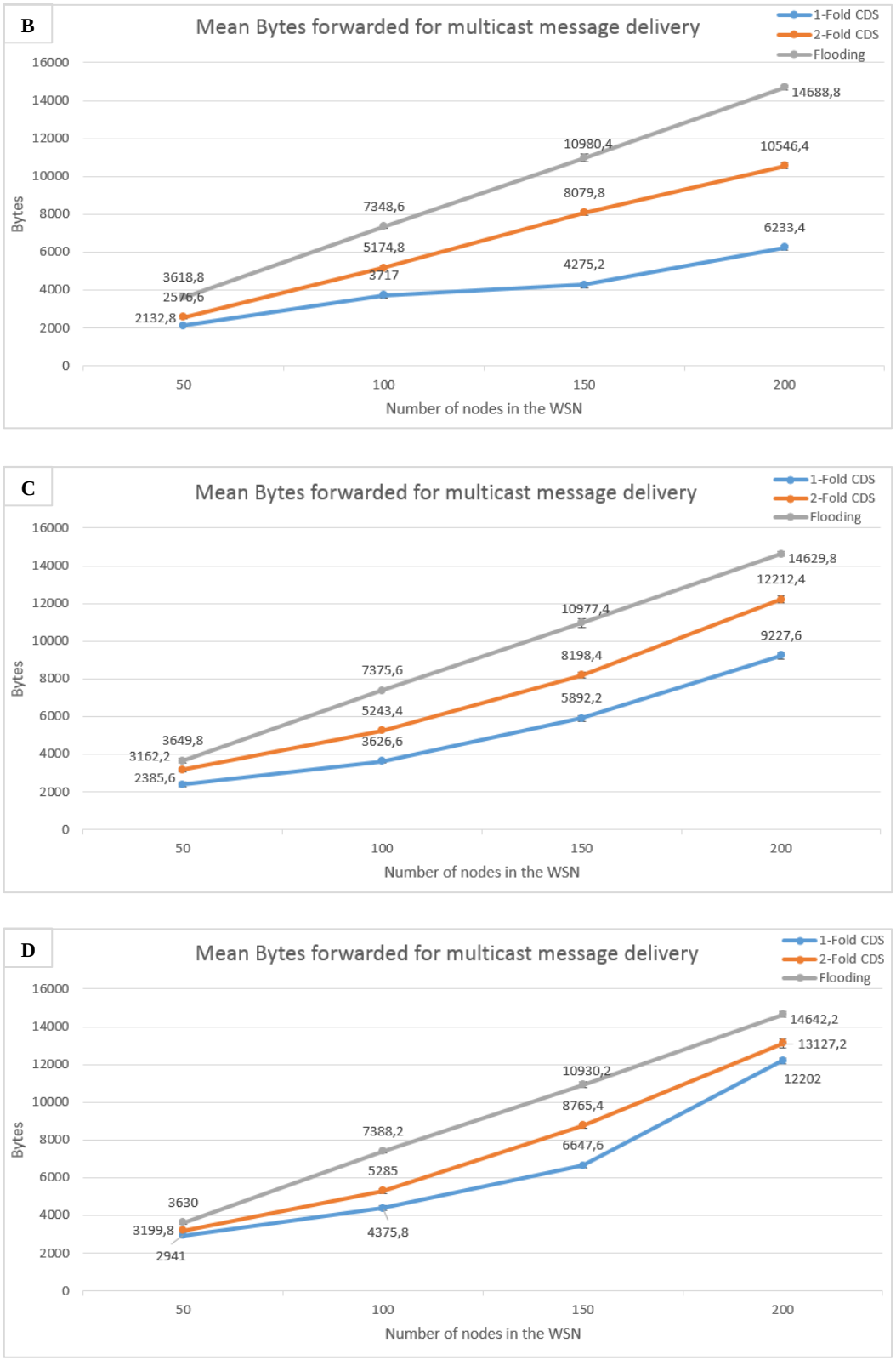
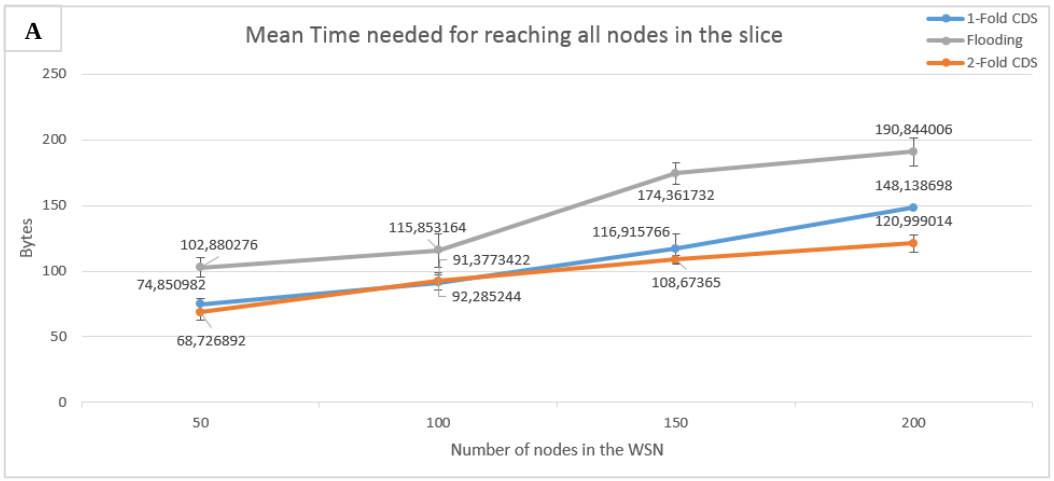


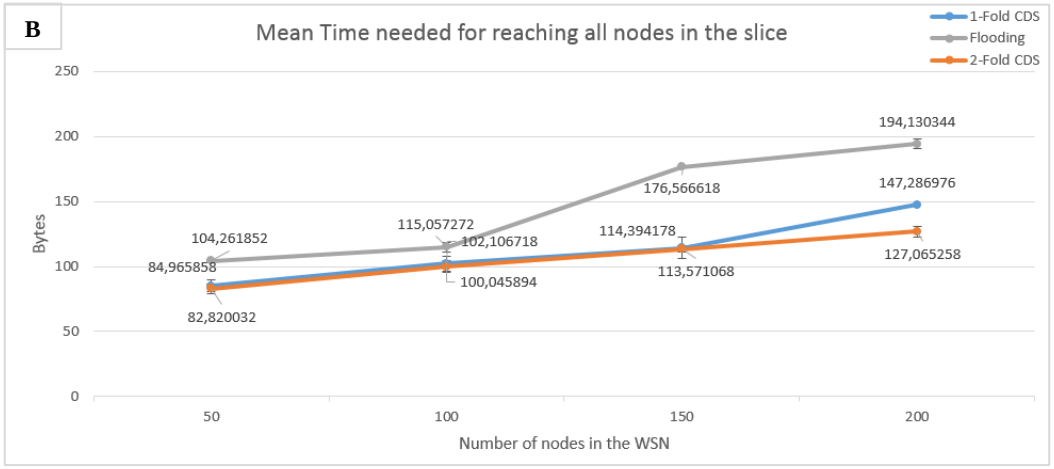
Fig. 6.22 - Mean bytes forwarded in the case of slice composed by respectively 10, 15, 25 and 35 nodes.

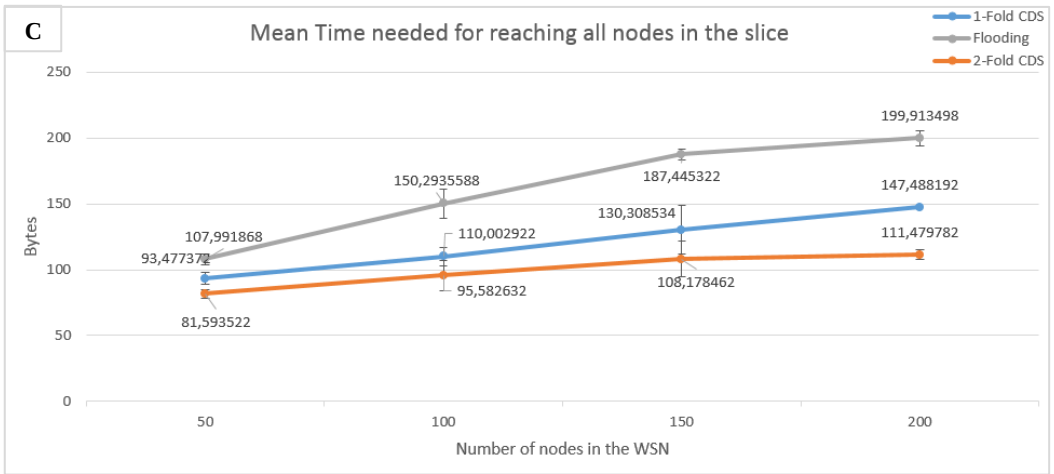
This second set of plots confirms what previously said. It was in fact expected that the *2*-fold scenario would imply more overhead than the non-fault-tolerant one. Moreover, it can be also noticed that the naïve flooding technique is outperformed by both proposed algorithms, since indeed they contemplate some reasoning and do not include the whole network in a multicast communication like it happens in a standard flooding scenario. A peculiar aspect that can be easily extrapolated from above graphs is that, as the size of the slice increases (plots A, B, C, D), the *1*-fold and *2*-fold CDS curves assume a trend that is more and more comparable to an exponential function, hence suggesting a non-linear growth for a fixed slice and an increasing network size. Focusing on the case of 10 preferred nodes, it can be also seen that the two curves seem to converge when increasing the size of the topology, hence indicating that in general the trend will most likely be represented by an S-shaped function, upper bounded by the flooding scenario.

Indeed, the flooding algorithm presents always a linear shape, thus indicating that such simple technique is not scalable at all and leads to loss of performance and higher power consumption, hence implying a smaller network lifetime. On the other hand, the two proposed algorithms are generating a virtual backbone that will of course see an increase of the communication overhead as the network grows, but (as explained) they will be always upper bounded by (and therefore more scalable than) the flooding technique, which represents the worst-case scenario. Another aspect that must be kept into account is the higher cost payed when utilizing a fault-tolerant backbone instead of the standard one. From the plots, this can be indeed considered as a not too expensive price to pay if fault tolerance is needed inside the network, hence representing a good compromise between reliability and communication overhead.

Mean time for message delivery







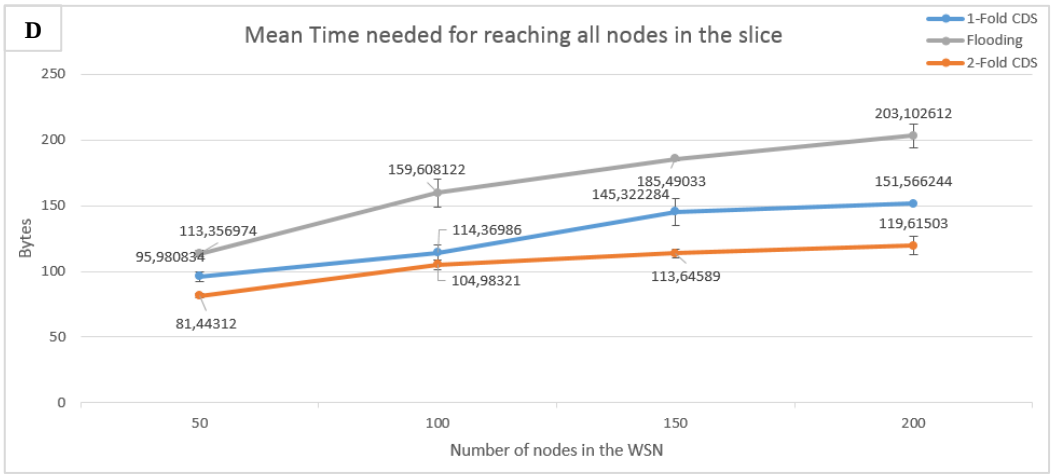
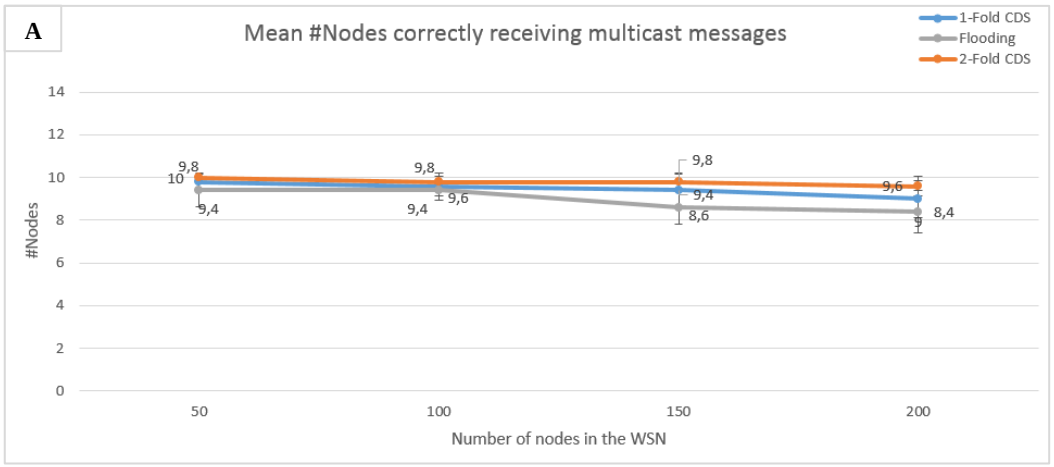
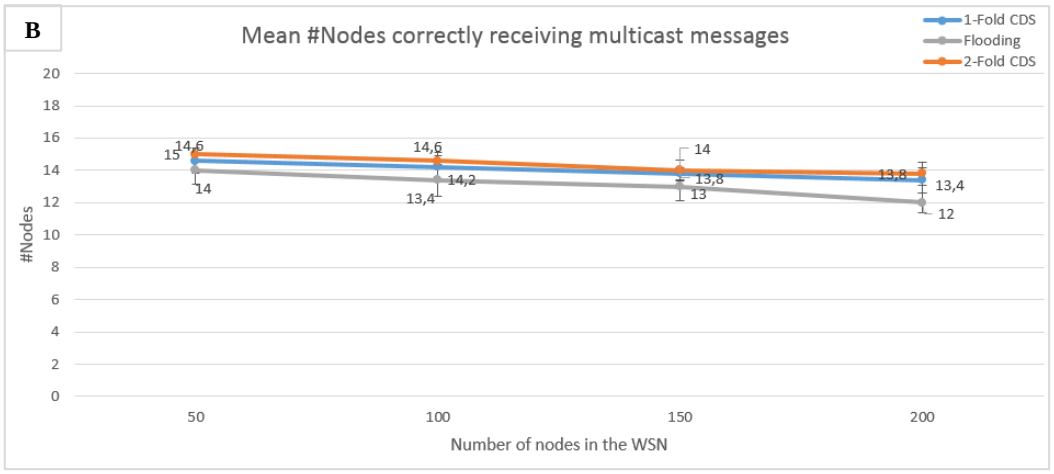


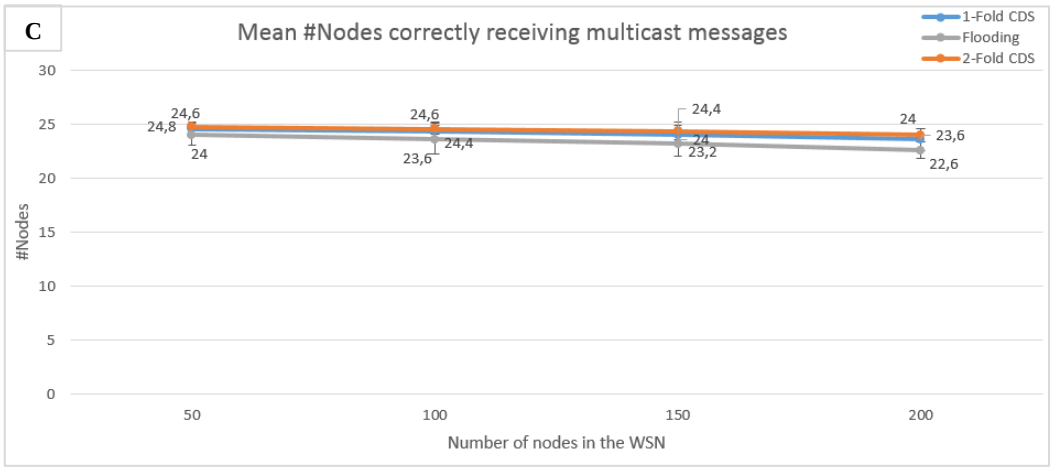
Fig. 6.23 - Mean time for message delivery in the case of slice composed by respectively 10, 15, 25 and 35 nodes.

As it could be expected, in terms of time needed for reaching all nodes inside the slice whenever a multicast communication happens, the flooding algorithm performs worse than the others due to the propagation of the packet by utilizing all the nodes in the network. Indeed, since sensor nodes have limited storage capabilities and there is the necessity of forwarding a big number of messages when such communication happens, this leads to network congestion and consequent loss of packets (which consequently increase the time needed to reach all preferred nodes). For what regards the two CDS scenarios, it is instead possible to notice that the proposed *2*-fold algorithm performs better than the *1*-fold case, thus suggesting a better performance for what regards the delay introduced in message delivery. The *1*-fold scenario provides in fact only a single route from the Sink to each node in the slice and therefore implies higher communication times due to its unicast-like nature, determined by the utilized system explained in 6.2.3.

Message delivery ratio







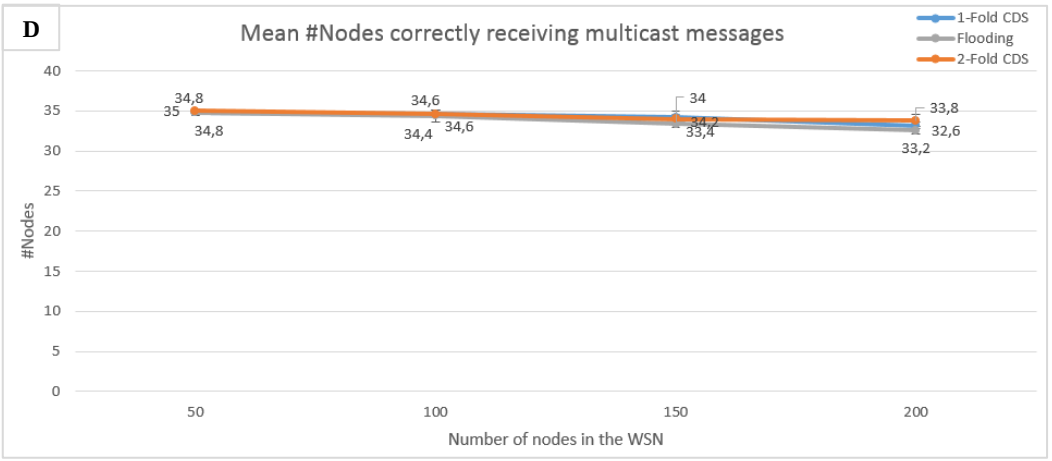


Fig. 6.24 - Mean number of nodes receiving multicast messages in the case of slice composed by respectively 10, 15, 25 and 35 nodes.

Finally, as explained in the previous paragraph, it can be noticed from the above plots that the loss of multicast messages is indeed higher in the flooding scenario. Although all three algorithms perform relatively well in an ideal simulative environment with no interferences and no risk of asymmetricity, this measure helps in understanding the fact that a less flooded network will, in general, provide a better delivery ratio (hence, a smaller packet loss) than one in which too many packets are being broadcasted throughout the system.

In conclusion, from all presented results, it can be understood that the *2*-fold technique performs generally better than the other ones, therefore indicating that a fault tolerant CDS can be implemented without any excessive drop in performances, but rather with very good reliability and delivery times, at the cost of some overhearing.

Experiment 3

In this last test, a specific topology of 21 nodes (Sink included) was designed in order to prove the correct functioning of the maintenance algorithm and to show what happens when a local repair procedure is preferred to a global one. The selection of a smaller network with regards to previous experiments is needed in order to better understand what happens in each section of the algorithm and why some choices are taken instead of others (indeed, a bigger number of nodes would have implied a less human-friendly architecture). Moreover, all tests took place in a simulative environment and the pictures

that will be shown in this document were obtained by the GraphStream "*display()*" function (where also some styling rules were automatically applied, for a more understandable result). For this reason, the figures won't depict all nodes always in the same position, but it must be kept in mind that the network topology is indeed the same for all of them.

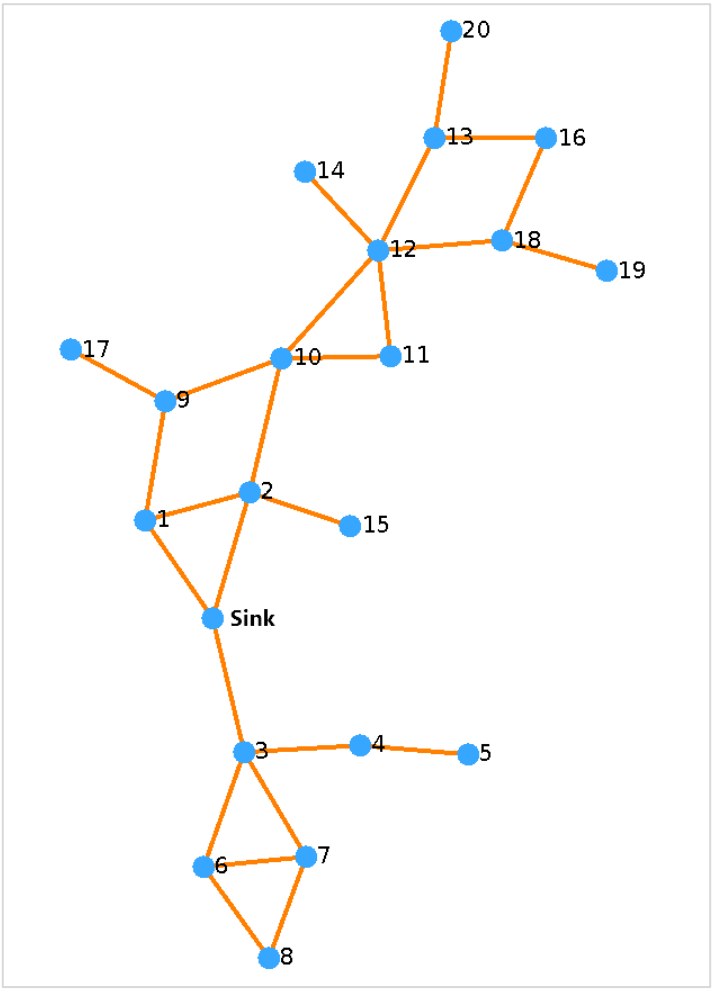


Fig. 6.25 – Initial network topology

This first picture illustrates the starting network topology, with all available wireless connections between nodes (orange lines), that was retrieved by the controller after the first set of topology update messages received. At this point, the client application requested for a slice composed by the nodes 5, 8, 14 and 16, to which the Sink was automatically added by the SDN Controller (it is assumed to be known) and the *1*-fold CDS algorithm was successively executed in order to build a virtual backbone for covering that specific set of preferred nodes.

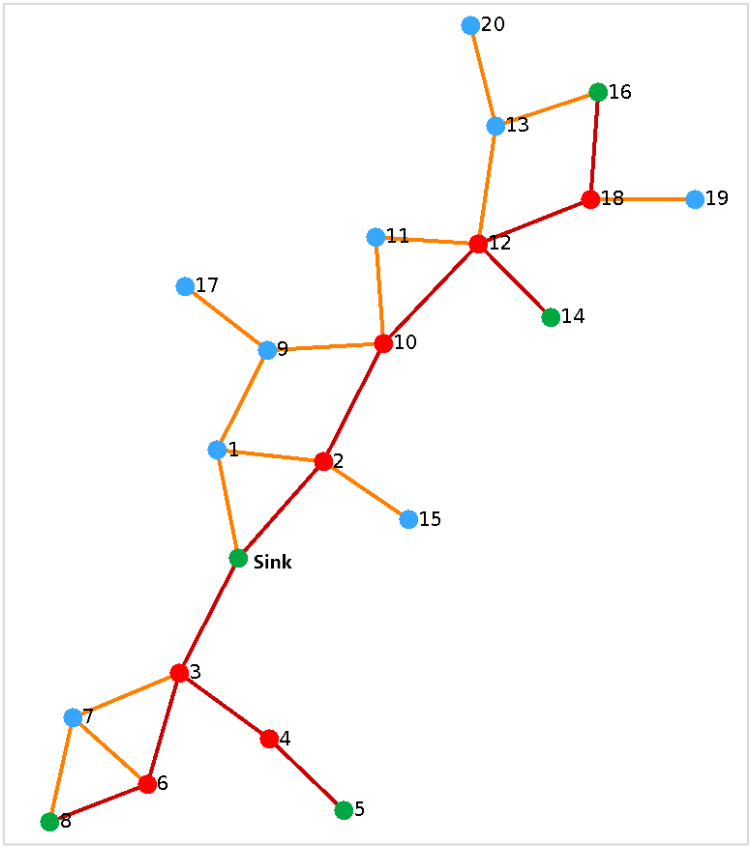


Fig. 6.26 – The obtained 1-connected, 1-fold CDS

As it can be noticed in the picture above, a Steiner Tree was built and all edges connecting nodes belonging to it were automatically colored in red by an additional function that was inserted only for graphical purposes in the algorithm. Furthermore, all nodes belonging to the slice can be recognized thanks to their green color, while all relay nodes are identified by their red tint. The final set is composed by 12 nodes over the 21 residing inside the network.

The third step was to cut two links by moving some nodes away from each other (in terms of communication range) and successively wait for the maintenance procedure to trigger (it starts whenever a new set of topology update messages is received by the SDN Controller). More precisely, the link between Sink and node 2 was cut together with the link between nodes 12 and 18.

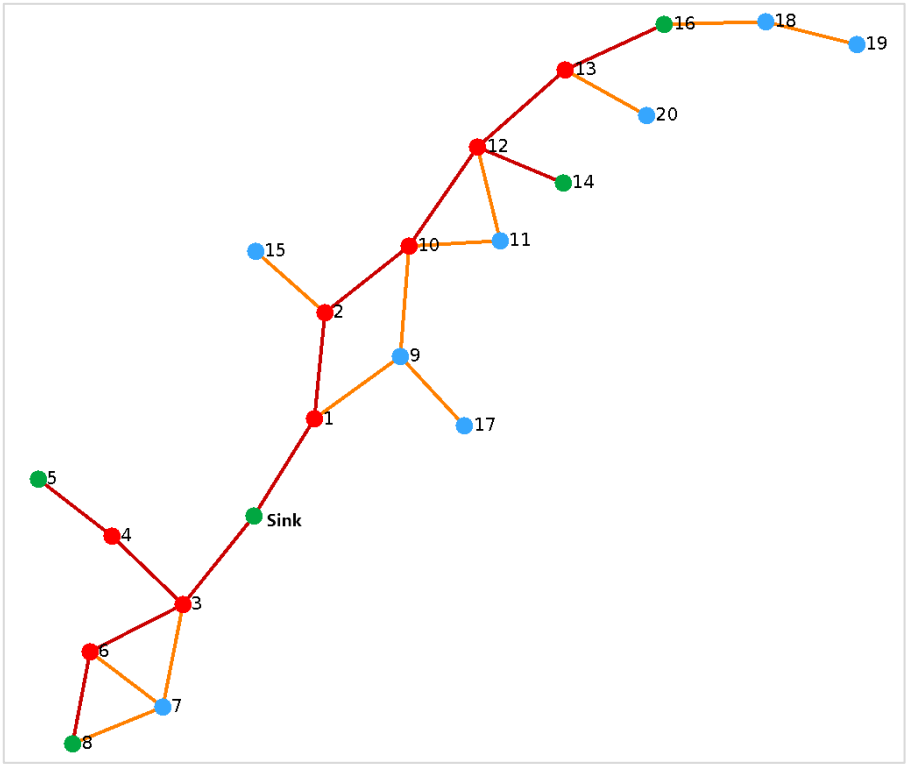


Fig. 6.27 – New CDS obtained from maintenance

Since two modifications to the whole graph were made, the maintenance algorithm, as described in 6.3, started computing both the new CDS obtained by a global repair and the repaired set procured by the local procedure. It can be easily noticed from above pictures that the numbers of nodes which needed an update in their flow table were exactly 5 in the global procedure (namely, node 1 – node 2 – node 13 – node 16 – node 18), countered by the 4 updates needed in case of local repair (node 1 – node 2 – node 13 – node 16). This is better explained below:

1. Node 1 was obviously new to the CDS, so it needed completely updated rules;
2. Node 2 needed to change its parent to be node 1 instead of the Sink;
3. Node 13 was new to the CDS;
4. Node 16 needed to change its parent to be node 13 instead of 18;
5. In case of global procedure, the node 18 was being excluded by the CDS.

Thanks to the threshold value applied in the algorithm (which was equal to 2 by experiment design), the global procedure was selected and the picture above indeed displays the completely new CDS computed and installed inside the network. This is

optimal because only an overhead of one message was payed in order to re-establish connection among the preferred nodes.

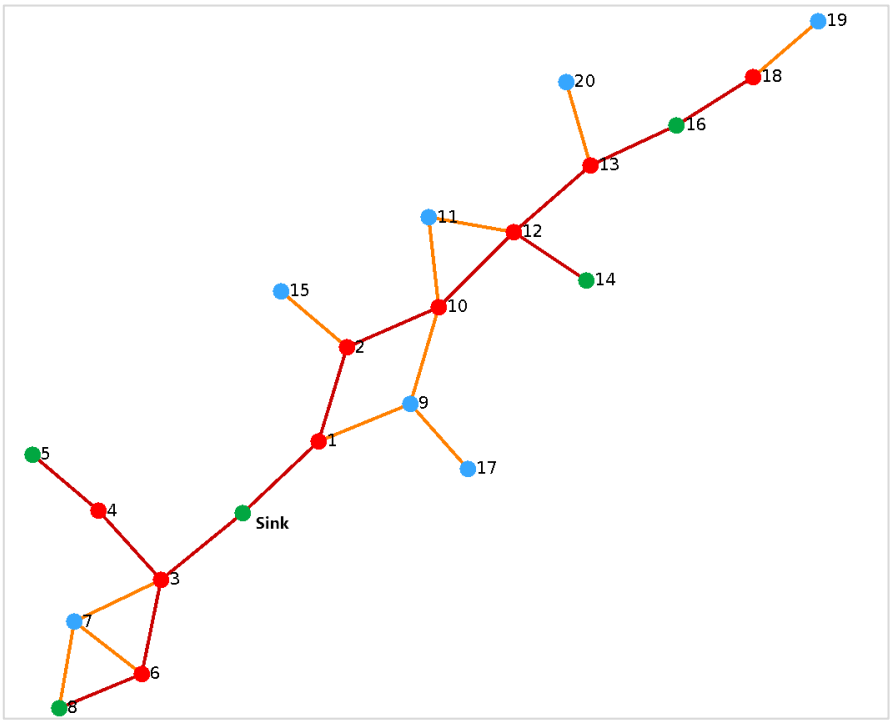


Fig. 6.28 – New CDS obtained from local repair

For completeness, this last picture represents what would have happened if the local repair procedure was selected (hence, with threshold value equal to 0) instead of the global one. As it can be noticed, node 18 would have been still inside the CDS, therefore propagating multicast messages with broadcasts, reaching also node 19 and hence wasting resources in the long term.

This explains why a threshold value was inserted inside the algorithm defined in 6.3 and why also a local procedure is always considered as an option. Indeed, a local repair would in general perform better when restoring the CDS in terms of control messages overhead, but sometimes (like in this case) this could imply a bigger consumption of computational and energy resources during the normal network lifetime. Therefore, a threshold value should always be selected in order to be sure that the global procedure is preferred among the local one and a more optimal solution is generally found. Of course, such value should not be too large, otherwise the local repair algorithm would never be chosen and this could lead to really big overhead of control messages due to the high dynamicity of

WSNs, while a too small value should also not be selected because otherwise the local repair would always win and lead to unnecessary waste of resources in the long-term run. This represents a trade-off between overhead of control messages and (above all) total network lifetime.

6.5 Testbed implementation and testing

After obtaining the presented experimental results from a simulative environment, for completeness, the modified SDN-Enabled Contiki system with slicing was also utilized on top of the real scenario provided by the university's testbed. This implied an additional shaping of the nodes' parameters in order to fit the available RAM, since a bigger flow table was needed in order to properly install slicing rules. Since the testbed is composed by only 21 nodes, two slices of 5 and 10 devices were selected, together with the same parameters and measures considered in the simulative experiments. The obtained results are hence not directly comparable to the previous ones (also because the topology is not the same), but are useful to prove the real and correct functioning of such paradigm on top of a real system. Considering the topology presented in 5.1, the two slices were selected in such a way that, theoretically, not all the nodes in the testbed were needed to construct a CDS. The two sets of preferred nodes are therefore composed by:

- Nodes with ID 4, 6, 13, 15, 21 for the slice of dimension 5;
- Nodes with ID 3, 4, 6, 13, 15, 16, 18, 19, 20, 21 for the slice of dimension 10.

Moreover, due to the non-ideal environment and the optimization of *Algorithm A* (explained in 6.2.1) that takes into account the possibility of asymmetric links, the formation of a proper CDS was sometimes delayed since the controller was not able to find both paths from a node *u* to *v* and vice versa. In that case, the number of multicast transmissions from application to slice were of course less then 12 in one hour. This indicates that, in a real environment, the proposed algorithms will work only after a certain warm-up period, determined by the network configuration.

Experimental results

Since the testbed topology is fixed and there is no variation in the number of nodes composing the network, the obtained results are hereby presented in a different way with regards to the previous ones. More precisely, the displayed plots consider the number of nodes selected to form a slice as a variable, instead of the network's scale. A confidence level of 95% is computed also in this case.

Mean Bytes for slice setup

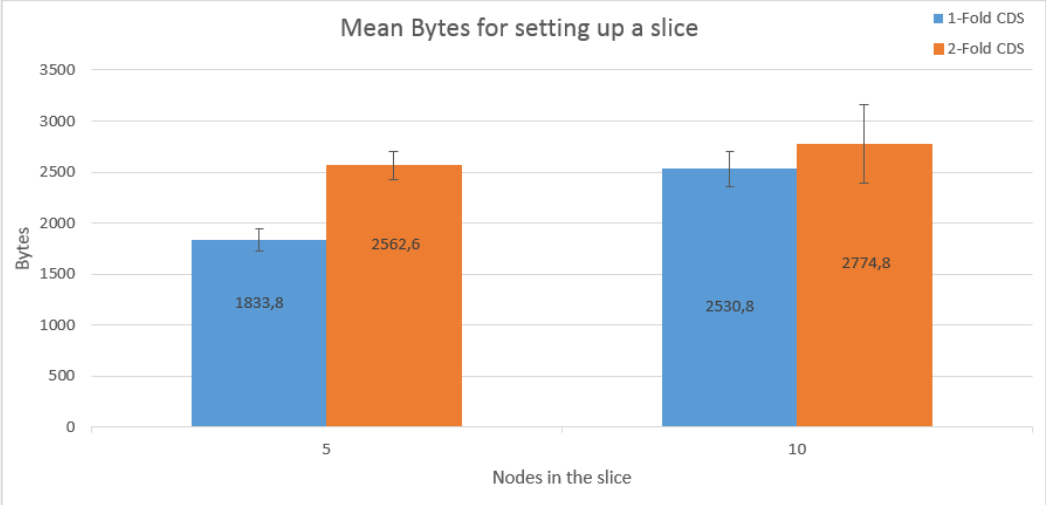


Fig. 6.29 – Mean Bytes needed for slice setup

For what regards the mean bytes transmitted for setting up a slice, it can be easily seen that the *2*-fold algorithm presents a bigger overhead as anticipated by the simulative experiments. This is indeed more visible in the case of slice composed by 5 nodes, since fault tolerance implies the selection of almost all nodes in the testbed (due to its small size). On the other hand, this difference is less visible when considering a slice composed by 10 nodes, which requires in fact at least half of the network to be selected by both algorithms, plus some of the remaining devices for connection purposes.

Mean Bytes forwarded for multicast message delivery

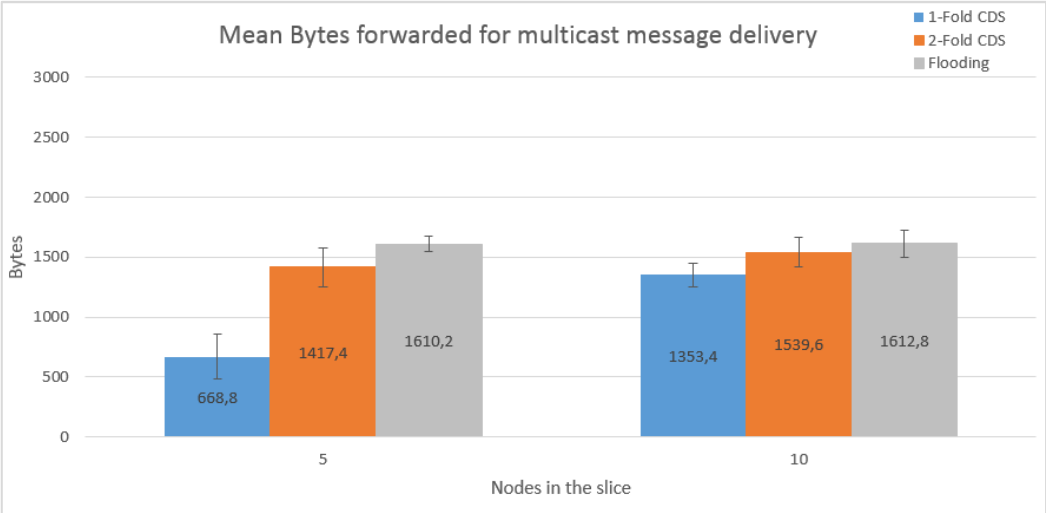


Fig. 6.30 – Mean Bytes forwarded for message delivery

Considering the Bytes forwarded when delivering a multicast message, it is again visible how both proposed algorithms outperform the naïve flooding technique, which also in this case performs in the same way for both sizes of the slice and represents an upper bound to the other scenarios. Once more, when the number of preferred nodes is relatively small with regards to the network size, the *1*-fold CDS presents much less overhead, while, when increasing it to 10, there is no substantial difference with the *2*-fold case.

Mean time for message delivery

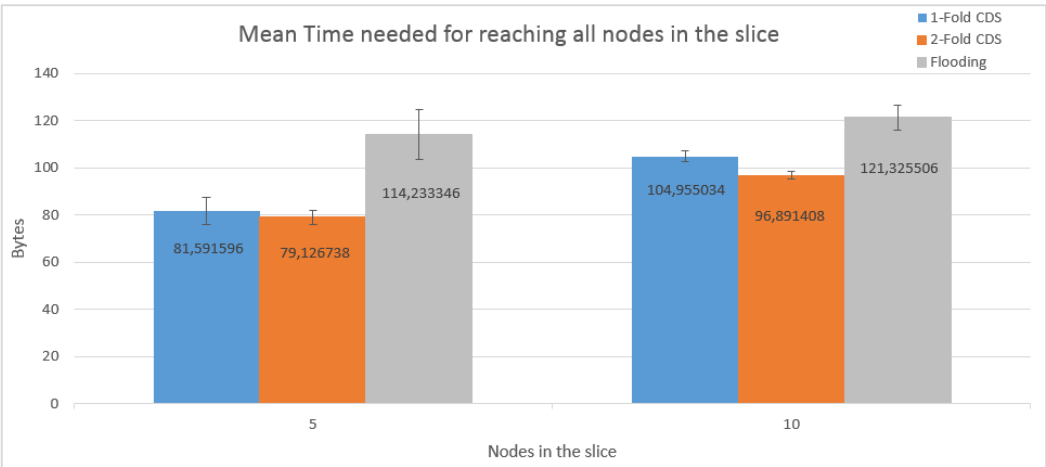


Fig. 6.31 – Mean time elapsed for reaching all nodes in the slice

This third plot shows also here how the *2*-fold technique is able to introduce a smaller delay with respect to other scenarios, thus confirming the analysis already made for the simulative results.

Message delivery ratio

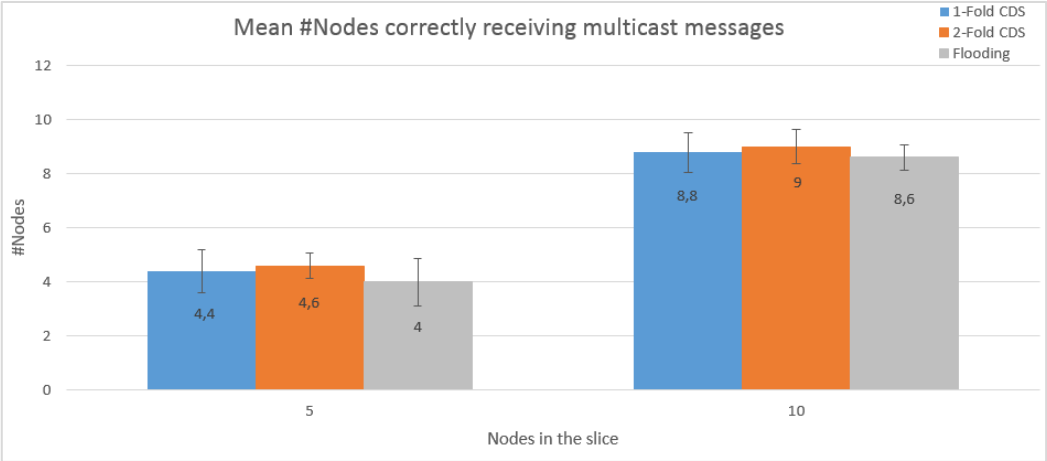


Fig. 6.32 – Mean number of nodes receiving the message

Finally, in terms of reliability, it can be noticed how also in this real environment the *2*-fold CDS is able to provide better performances with regards to the other techniques. This can be explained by following the exact same reasoning proposed in 6.4.2.

From above results, it is therefore possible to confirm that, generally, a *2*-fold approach should be preferred when building a CDS for slicing purposes, since it will provide a very good trade-off between reliability, delay and overhead of messages inside the network.

Conclusions

In this thesis work, two main milestones were reached in order to provide a substantial contribute to the implementation of the SDN paradigm over standard WSN architectures. Starting from an already designed theoretical system, a SDWSN framework was implemented on top of a real physical environment and tested in order to be sure that the costs of utilizing SDN over WSN are not too expensive with regards to what is offered by the exercise of such paradigm for resolving the multiple scalability and flexibility issues arising with the diffusion of IoT infrastructures. From the results of such experiments, it was indeed noticed that a general environment in which sensor nodes are interconnected wirelessly and communicating with a standard distributed routing protocol, such as RPL, could support the separation of the control logic from the data plane and therefore enable an easier and more flexible management of the whole system, by paying a relatively small cost in terms of communication overhead and reliability. This could lead to the implementation of the SDN paradigm over sensor networks utilized, fore example, in the so-called Industry 4.0, where very large-scale IoT infrastructures are in place in order to manage the proper functioning of each department and machinery. A centralized controller that manages the network so to have better scalability, interoperability, mobility and security could in fact lead to more reliable and controllable infrastructures, together with an overall longer battery lifetime for every sensor node that is installed in not easily reachable positions (hence, requiring less maintenance costs).

The introduction of more flexibility for what regards the supervision of wireless sensor networks is also supported by the implicit ability of SDN to implement network virtualization. Considering again the Industry 4.0 example, it could indeed happen that for instance different administrators would need to monitor the status of specific machines inside a building by excluding some others and issuing requests to all of them simultaneously. Furthermore, thinking about a completely different environment, such as a smart city having multiple sensors placed inside street lamps which are wirelessly interconnected and providing temperature or quality of air measures, a central controller would be able to assign and share particular slices between multiple weather authorities

depending on how much they pay and what part of the city is in their interest for providing forecasts (a local authority could need only the measurements from a small set of nodes, while a national one could need values for the most important places inside the city, and so on). Therefore, the second milestone reached during this thesis work was indeed the proposal and subsequent practical implementation of multiple algorithms for constructing virtual backbones in order to reach all nodes belonging to a specific slice, also with the possibility of fault tolerance and an optimized maintenance procedure. Since most of the proposals regarding the application of SDN over WSN presented during the last years did concentrate only on the routing based on a central controller, this newly suggested virtualization system represents an important achievement that shows how it is in fact possible to provide an overall better service and introduce completely new features if such paradigm is applied in an IoT environment.

Summarizing, it has been explained why there is the need of applying a new archetype to the standard WSN architectures which are supporting IoT infrastructures and it has been demonstrated that it is indeed possible to apply SDN over WSN in a real environment, with a price to pay that is compensated by all additional features that a SDWSN provides with respect to a standard WSN. In conclusion, the hope is that what was achieved during this thesis work can successfully encourage and give credit to the further development of such paradigm, which is nowadays still in its state of the art.

References

[1] *"Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are",* Cisco Syst., San Jose, CA, USA, 2016.

[2] Luigi Atzori et al. *"The Internet of Things: A survey"*.

[3] I.F. Akyildiz et al. *"Wireless sensor networks: a survey"* Computer Networks, Volume 38, Issue 4, Pages 393-422 (2002).

[4] Chowdhury, A K M Rezaul Haque et al. *"Route-over vs mesh-under routing in 6LoWPAN."* IWCMC (2009).

[5] P. M. Julia and A. F. Skarmeta. *"Extending the Internet of Things to IPv6 With Software Defined Networking"*, White Paper, 2014.

[6] Wenfeng Xia et al. *"A Survey on Software-Defined Networking"* IEEE Communication Surveys & Tutorials, Vol. 17, No. 1, First Quarter 2015.

[7] A. Lara, A. Kolasani and B. Ramamurthy, *"Network Innovation using OpenFlow: A Survey,"* in IEEE Communications Surveys & Tutorials, Vol. 16, No. 1, Pp. 493 – 512, First Quarter 2014.

[8] OpenFlow Switch Specifcation
https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf

[9] Hlabishi I. Kobo et al. *"A Survey on Software-Defined Wireless Sensor Networks: Challenges and Design Requirements"* IEEE Access, Vol. 5, Pp. 1872 – 1899, February 2017.

[10] D. Drutskoy, E. Keller and J. Rexford, *"Scalable Network Virtualization in Software-Defined Networks"* in IEEE Internet Computing, Vol. 17, No. 2, Pp. 20-27, March-April 2013.

[11] Michael Baddeley et al. *"Evolving SDN for Low-Power IoT Networks",* IEEE International Conference on Network Softwarization, June 2018.

[12] T. Luo, H.-P. Tan, and T. Q. S. Quek, *"Sensor OpenFlow: Enabling software-defined wireless sensor networks"* Commun. Lett., Vol. 16, No. 11, Pp. 1896–1899, November 2012.

[13] 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 Proc. IEEE Conf. Comput. Commun. (INFOCOM), Apr./May 2015, pp. 513–521.

[14] GitHub repository with the initial system implementation
https://github.com/BigMikes/contiki

[15] Contiki: The Open Source OS for the Internet of Things
http://www.contiki-os.org/

[16] RFC 7049 Concise Binary Object Representation
http://cbor.io/

[17] Californium repository on GitHub
https://github.com/eclipse/californium

[18] GraphStream documentation
http://graphstream-project.org/doc/

[19] Zolertia RE Mote platform documentation for Contiki
https://github.com/contiki-ng/contiki-ng/wiki/Zolertia-RE-Mote-platform-(revision-A)

[20] Zolertia Zoul core module documentation for Contiki
https://github.com/contiki-ng/contiki-ng/wiki/Platform-zoul

[21] GitHub repository for the SDN-enabled Contiki OS
https://github.com/skixmix/contiki

[22] RFC 6552 – Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL)

[23] Zhang et al. *"Computing Minimum k-Connected m-Fold Dominating Set in General Graphs"* Informs Journal on Computing. 30. 217-224 (2018)

[24] Guha, Sudipto & Khuller, Samir *"Approximation algorithms for connected dominating sets"* Algorithmica. 20. 374-387 (1998)

[25] Wu, Y.F., Widmayer, P. & Wong, C.K. *"A faster approximation algorithm for the Steiner problem in graphs"* Acta Informatica (1986)

[26] GitHub repository for the SDN Controller and Client Java codes
https://github.com/skixmix/californium-sdn