Abstract

In the future everyday objects will be smart and Internet will be used to send data all over the world.
To achieve this goal the *Internet of Things* (IoT) has become an emerging research area, gaining a great attention by the scientific community.
Wireless Sensor Networks (WSNs) are a well-known research area related to the IoT. They consist of spatially distributed sensor nodes, which are used to study physical or environmental conditions and to send their data everywhere through the routing protocol IPv6. This standard, applied to WSNs, realizes the *IPv6 Low Power Personal Area Network* (6LoWPAN), which connects the wireless domain to Internet through the IP Stack.
In order to check the data communication exchanged between the nodes, I will develop a metrics collection tool for the COOJA simulator; in other words, this metrics collection tool will be used to analyse the physical, data-link, network and transport layers with the aim to detect, for example, the possible losses of packets or the congestion of a node.

My research will be organized in the following five chapters: chapter 1 introduces the 6LoWPAN networks and explains the objective work; chapter 2 contains an explanation on basic concepts of my dissertation, illustrating the related works of the scientific community about COOJA and the missing points to optimize the analysis of 6LoWPAN networks; chapter 3 represents the development of the plugin used to collect metrics; chapter 4 studies some particular COOJA simulations in order to show the results of my tool; chapter 5, finally, reports the conclusions and the future possible developments.

Sommario

Nel futuro qualsiasi oggetto sarà intelligente e utilizzerà Internet per inviare informazioni in ogni parte del mondo.
Al fine di raggiungere tale obiettivo, l'*Internet of Things* (IoT) è diventata un'area di ricerca emergente, guadagnando una grande attenzione dalla comunità scientifica.
Wireless Sensor Networks (WSNs) sono un'importante componente dell'IoT.
Queste reti sono costituite da nodi sensore spazialmente distribuiti, utilizzati per studiare condizioni fisiche o ambientali e per inviare le informazioni rilevate ovunque attraverso il protocollo di routing IPv6. Tale standard, applicato alle reti WSN, realizza l'*IPv6 Low Power Personal Area Network* (6LoWPAN), la quale connette il dominio wireless a Internet attraverso lo stack IP.
Con lo scopo di controllare la comunicazione dei dati scambiata tra i nodi, sarà sviluppato uno strumento di collezione metriche per il simulatore COOJA. Nello specifico il nuovo plugin sarà utilizzato per analizzare il livello fisico, il livello collegamento, il livello rete e il livello trasporto al fine di rilevare, per esempio, le possibili perdite di pacchetti oppure la congestione di un nodo.

Il lavoro svolto sarà distribuito nei seguenti 5 capitoli: il primo capitolo introduce le reti 6LoWPAN e illustra gli obiettivi della tesi; il secondo capitolo mostra i concetti base di riferimento, evidenziando i lavori svolti dalla comunità scientifica relativi al simulatore COOJA e i punti mancanti per ottimizzare l'analisi di reti 6LoWPAN; il terzo capitolo illustra la progettazione del plugin; il quarto capitolo esamina alcune particolari simulazioni COOJA al fine di mostrare i risultati ottenuti dallo strumento di collezione metriche; il quinto capitolo riporta le conclusioni e i possibili lavori futuri.

Contents

1. Introduction 5
1.1 IP Networking For Smart Objects 5
1.2 Objective Work 6
2. Related works 7
2.1 Basic Notations 7
2.1.1 IEEE 802.15.4 Protocol 7
2.1.2 IPV6 in Low-Power Wireless Personal Area Network (6LoWPAN) 8
2.1.3 RPL: IPV6 Routing Protocol for Low-Power and Lossy Networks (LLN). 9
2.1.4 TINYOS, short description and implementation of 6LoWPAN networks. 11
2.1.5 COOJA 13
2.2 Scientific Community Works 16
2.3 Original Contribution 18
3. Plugin Development 19
3.1 Application Field 19
3.2 Cooja Architecture 22
3.3 Plugin General Functioning 23
3.4 Plugin Basic Architecture 26
3.5 L1 Components 30
3.5.1 Generation Radio Connections 32
3.5.2 Capture Phase - Node Metrics 33
3.5.3 Capture Phase - Couple Metrics 35
3.6 Implementation Data-Link Layer and Networking Layer in COOJA 37
3.7 L2 Components 41
3.7.1 Generation Frames 43
3.7.2 Capture Phase - Node Metrics 47
3.7.3 Capture Phase - Couple Metrics 49
3.8 L3 Components 49
3.8.1 Generation Datagrams 52
3.8.2 Capture Metric - Delay End-To-End 58
4. Plugin Testing 59
4.1 Realization COOJA Simulation 59
4.2 Analysis Metrics – Physical Layer 62

4.3 Analysis Metrics – Data-Link Layer 65
4.4 Analysis Metrics – Networking Layer 73
4.5 Statistics 75
4.6 Testing Multiple Simulations 77
5. Conclusion 82
Appendix 84
Section A – Components of TestRPLC 84
Section B – Details Generation XML 85
Section C – Details Radio Power Level in UDGM 85
Section D - Installation Metrics Collection Tool In Cooja 86
Section E – Statistics Values 89
References 91

1. Introduction

1.1 IP Networking For Smart Objects

Internet of Things is the new Internet frontier. It refers to the interconnection of IP smart objects, such as sensors and actuators. Such devices are used in various fields of industry, i.e. Smart Grid, Smart Cities, building and industrial automation to detect physical parameters like power quality, temperature, level of pollution, pressure and sound.

Wireless Sensor Networks have been recognized as a very important block of IoT; they consist of a large number of nodes deployed in the environment, which are sensed and controlled through wireless communications.

The main features of a WSN can be defined as follows: power consumption constraints for each node depending on the use of batteries or of energy harvesting; ability to cope with node failures; mobility of nodes; dynamic network topology; communication failures; heterogeneity of nodes; scalability of large scale of deployment.

The network traffic in the WSNs is divided in two main types namely defined as application and routing traffic. The application traffic consists of data captured in the environment of interest (for example home automation, pollution monitoring, temperature monitoring etc.) by sensors and then sent to the *sink* node that gathers the detected information with the aim to forward it all over the world via Internet. The routing traffic allows to route the application traffic through a hierarchy graph divided into levels, using routing protocols suitable for low-power and lossy networks (LLN [1], typical WSN), like RPL (Routing protocol for LLNs).

Considerable work has been done from IETF that first introduced the IPV6 protocol to address the data detected by the sensors on IEEE 802.15.4 links (standard protocol for the physical and data-link layer in WSNs) and to perform necessary configuration functions to form and maintain a routing graph. The arrival of IPV6 protocol routing in LLN links leads to the configuration of 6LoWPAN (acronym of IPV6 over Low power Wireless Personal Area Network) networks.

In order to check the data traffic exchanged in 6LoWPAN networks, various telecommunication research centres use specific simulators, for example COOJA, Worldsens, MiXiM, Castalia and NS-2.

These tools allow the user to build network topologies of sensor nodes, spatially distributed in a limited environment, with the aim to respect the binding characteristics of WSNs. Once the network is carried out, the software simulates the network applications (developed using specific libraries of the embedded operating systems, running on devices, for example tinyOS and Contiki) in order to analyse the communications of packets through the main layers of the ISO-OSI Stack, which are the physical, the data-link, the networking and the transport layers. Considering exclusively the 6LoWPAN networks, the components of the IP-stack are implemented by the 802.15.4, IPV6 with RPL and UDP protocols.

1.2 Objective Work

The objective of my dissertation consists in presenting a new plugin for the simulator COOJA.

The metrics collection tool has the function to collect detailed metrics about the packets sent on the radio medium for each layer of the IP Stack.

The study of the messages exchanged between nodes changes according to the layers. On the physical channel it is very interesting to analyse each radio connection established independently from the type of the sent packet. The aim is to detect the number of radio packets correctly sent and the number of losses due to collisions or random errors on the channel, giving considerable attention to the evolution of the radio power for each instant.

At data-link layer, the plugin COOJA controls the frames building in order to check possible losses caused by transmission or reception errors on DATA or ACK messages.

Finally, for the higher levels, the metrics collection tool focuses on the observation of the trend of the datagrams queue and the delay end-to-end, which is necessary to send an IPv6 packet from a source node (environment sensor) to the root, also called sink node.

Once metrics have been obtained, the plugin will process important statistics, such as packet loss rate, average delay end-to-end and throughput, useful to evaluate reliability, speed and capacity of the links.

2. Related works

This chapter shows the basic notations behind the development of my metrics collection tool, i.e. IEEE 802.15.4 protocol, 6LoWPAN mechanism, RPL, TinyOS and COOJA simulator. It also illustrates the works done by the scientific community about COOJA and the missing points to optimize the analysis of 6LoWPAN networks.

2.1 Basic Notations

2.1.1 IEEE 802.15.4 Protocol

IEEE 802.15.4 [2] [3] is a protocol, which establishes the physical layer and the MAC layer (media access control) for low-rate wireless personal area networks that focus on low-cost and low-speed communication.

The basic framework defines a communication range of 10 metres, a data rate of 256 Kbits and a milliwatt of transmission power.

The main feature of 802.15.4 WPANs is the importance of offering extremely low manufacturing and operation costs, technological simplicity, without sacrificing the flexibility and the multiplicity of use.

Another important feature is the reservation of guaranteed time slots, the collision avoidance through CSMA/CA and the integrated support for secure communication.

The architecture of the protocol is based on the ISO OSI model; although the IEEE 802.15.4 Protocol contains only two layers, it can interact with higher layers.

The physical layer provides the data transmission service, managing the transceivers, performing channel selection and energy and signal management functions; on the contrary the MAC layer manages the transmission of frames on physical channel, controls frame validation, guarantees time slots and handles the linking of nodes.

Moreover, the second layer provides a small packet whose size is of 127 bytes, which requires an adaption layer with fragmentation and reassembly to allow the transmission of packets bigger than 127 bytes.
Additionally, it does not provide a full-broadcast domain where all nodes are able of receiving messages from all the other nodes using a single physical transmission; on the contrary, 802.15.4 links are composed of overlapping broadcast domains, where a set of neighbour radios is defined through those sensors reachable with a single transmission.

2.1.2 IPV6 in Low-Power Wireless Personal Area Network (6LoWPAN)

The IETF chartered in 2005 the 6LoWPAN working group (WG) [2] so to standardize the use of IPV6 over IEEE 802.15.4 radios [1]. These radios in fact have highly different characteristics and problems (listed in the previous paragraph) unlike older link technologies, such as Ethernet or WiFi. To optimize the costume of network protocol the WG has focused on two important items:

- How to carry IPV6 datagrams in 802.15.4 frames
- How to perform IPV6 neighbour discover functions

To obtain the first goal, the Working Group has introduced a fragmentation and reassembly link layer mechanism, since IPV6 specifies that the link must support a MTU (Maximum transmission unit) of no less than 1280 bytes.
The fragmentation mechanism does not include an end-to-end recovery of loss datagrams. Indeed it uses the link layer acknowledgment service to provide a sufficient success rate.
The fragmentation header (4-5 bytes) contains three fields: datagram size (bulk of the datagram being fragmented), datagram tag (index associated to each fragment of a datagram) and datagram offset (fragment position within the datagram).
At the reception side the destination node allocates a reassembly buffer that recomposes the packet through the “datagram size” included in all fragments.
Another important mechanism is the Header Compression; traditional IP Networks use flow-based compression techniques, observing which portions of the header change more rarely across packets within a flow and eliding those portions when possible. For the Wireless Sensors Networks the WG has developed a new

compression format that does not require per-flow state, because the path of a flow may frequently change due to time-varying dynamics of low-power wireless communications.
This new format defines the header in two ways: firstly, by removing redundant information across ISO-OSI layers; secondly, by assuming common values for header fields and defining compact forms for those values.
The header compression also allows stateful compression for IPV6 address prefixes. Each node keeps a context table, where each entry contains an IPV6 prefix.

To reach the second target, the 6LoWPAN Working Group has optimized the existent IPV6 Neighbour Discovery procedure (ND), defined in [4].
The ND protocol uses multicast communications and assumes that the link provides a single domain broadcast.
Since the MAC layer of 6LoWPAN networks does not support multicast, broadcast communications are used to deliver multicast packets. This solution can cause some problems due to the lack of the acknowledgment service for the broadcast messages and moreover this type of transmission is usually more expensive.
The challenges, applied by WG, remove the critical points and minimize the ND's reliance on multicast, obtaining as result the 6LoWPAN ND.
This protocol uses Router Advertisement (RA) and Router Solicitation (RS) messages to give the nodes the possibility to find neighbouring routers. The RS notifications are unique messages, sent on multicast mode. Two are the main reasons: first, link-local IPV6 addresses are obtained from MAC addresses, so nodes do not have to perform address resolution; second, IPV6 addresses, different from link layer addresses, are assumed not to be on-link and therefore communications with these nodes proceed by routers.

2.1.3 RPL: IPV6 Routing Protocol for Low-Power and Lossy Networks (LLN)

In these years many routing protocols have been created for IP networks (for example RIP, OSPF, BGP), but routing in networks made of IP smart objects has the following characteristics: devices constrained in terms of resources, low-speed lossy links where the packets drop ratio may be quite high and the presence of hundreds of thousands of nodes in the networks. Therefore IETF has designed a new Routing Protocol, called

RPL (Routing Protocol for LLNs) [1] [5].
RPL is based on the following concepts: “routing state propagation”, “spatial diversity” and “expressive link and node metrics”.
Routing state propagation is implemented in RPL by the Trickle algorithm, which quickly reacts to changes in routing state and tapers off as the rate of state changes decreases.
Spatial diversity is a technique to achieve reliability in environments where nodes are likely to fail due to environmental causes and interference can quickly change good links into useless links. In practice, the routers can maintain multiple potential parents towards a destination instead of a single one.
The third concept refers to the link costs; unlike the existing routing protocols the LLN links present time-varying dynamics with no static link costs.
In order to fulfil this feature, RPL contains a flexible framework for incorporating dynamic link metrics such as ETX [6] (Estimated number of Transmissions).

The routing protocol supports three traffic patterns: multipoint-to-point traffic, point-to-multipoint traffic, and point-to-point traffic.
For the LLN applications the dominant traffic is the multipoint-to-point type.
Usually there are some routers with the aim to connect the 6LoWPAN network to the Internet. Thus RPL works out a destination oriented directed acyclic graph (DODAG) and uses this chart to route data traffic. As mentioned above, the protocol makes use of the Trickle Algorithm to exchange the routing information on the DODAG. These data are called Objective functions (OFs) and are used to compute the rank value for each node that is a scalar representation of the device location within the DODAG. Nodes use these data to determine the set of parent nodes closer to the graph root.
OFs are included in particular packets, named DIO (DODAG information objects).
The emission of DIO packets is regulated through the DIS messages (DODAG information solicitations). Their function is the same of the Router Solicitations, used in IPv6.
There is another type of RPL packets, the DAO messages (Destination Advertisement Object) with the aim to support routing to various destinations within the graph. We can consider two kinds of DAO messages: not-storing mode and storing mode.
In the not-storing mode, nodes have not got enough memory to store routes to all

possible destinations. In this case the DAO messages, which contain information about the parent set of a destination node, are propagated up the DODAG until they reach the root. The root receives the RPL messages, stores the routes and redirects the traffic to its destinations using the source routing.
In the storing mode, nodes maintain a routing table. In this way the traffic between two nodes travels along the DODAG in an upward direction to a common parent; at this point datagrams are redirected to their destination.

Security is an important concept for LLNs since they cover an important part in the critical infrastructure. An often used protection mechanism is the acknowledgements service. Unfortunately it cannot guarantee the integrity of routing datagrams. For this reason RPL introduces the advanced encryption standard (AES) and RSA signatures for checking the integrity and authentication of routing messages.

2.1.4 TINYOS, short description and implementation of 6LoWPAN networks

TinyOS [7] is an open source operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters.
Tinyos supports many hardware nodes, including Texas Instruments Chipcon CC2420, CC1100, CC2500 and the Atmel RF212 and RF230 [8].
Particularly interesting are the CC2420 radios, since this technology is leader among radio chips for IEEE 802.15.4 standard.
The software radio stack, that drives the CC2420 chip, consists of many layers that sit between the application and the hardware. The highest levels of the radio stack modify data and headers of each packet, while the lowest levels determine the actual behaviour in sending and receiving radio packets.
Moreover, it includes a *digital direct sequence spread spectrum baseband modem* providing a spreading gain of 9 dB and an effective data rate of 250 Kbps.
The operating system can be installed on different microcontrollers, i.e. Texas Instruments MSP430 family, Atmel At-mega128, Atmega128L, Atmega1281, and the Intel px27ax processor [8].
Given the low power consumption, the “Texas Instruments MSP430 family” is the perfect microcontroller for TinyOS. Other features of MSP430 are the following:

analogue-to-digital converter, real time clock, timers, asynchronous/synchronous serial interface and a DMA controller.

The MSP430 microcontroller and the CC2420 radio together create the mote called *TELOSB*. The TELOSB mote operates at 4.15 MHz, it has 10 Kbytes of RAM and 48 Kbytes of ROM. Moreover it is one of the first platforms to provide an open implementation of the 6LoWPAN and RPL protocol.

To reach this goal TinyOS provides a particular library called *blip* [1], which gives the possibility to form multi-hop IP networks consisting of different motes communicating over shared protocols. Motes are divided in two categories, hosts and routers: a host does not forward packets and does not participate in routing protocols, while a router does it.

The smallest unit of network management in *blip* is the subnet; it consists of a number of devices (router nodes) and one or more particular motes (edge router nodes), which perform many routing functions for the network and for other networks.

Since 6LoWPAN protocol uses a fragmentation mechanism, *blip* implements this fragmentation mechanism through the *layer 2.5 fragmentation tool*. *Layer 2.5* provides larger payloads to upper-layer protocols and manages the addresses assignment. The addresses assignment may take place statically during the instant in which the program compile is performed, or dynamically using the DHCPV6 process [9]; both the static and the dynamic ways configure three addresses: two link-local addresses and one global address.

The first link-local address is configured from the Extended Unique Identifier 64 (EUI-64) [10]. The second link-local address is chosen according to the *panid* (identifier of personal area networks) and *nodeid* (identifier of motes) as specified in [11]; the default *panid* is 0x22, so the corresponding address for the mote with *nodeid* 1 is fe80::22:ff:fe00:1.

Finally, the global address is obtained using a particular prefix defined at the moment in which the program compile is carried out; the default value is fec0::/6,4 so if the mote ID is 3, the IPV6 address is fec0::3.

If the prefix is not set at the compile time, BLIP starts the DHCP procedure.

In this case we can see a self-configuration of nodes only for the first link-local

[1] TinyOS library - http://docs.tinyos.net/tinywiki/index.php/BLIP_2.0

address. On the contrary, for the global address, nodes send DHCPv6 solicitations to find neighbouring server. Each router also runs a DHCPv6 relay agent, which forwards address solicitations along the routing tree up to the edge where the DHCPv6 server is presumed to be located.
Once the DHCPV6 server has assigned the global address to the mote, it starts to run the RPL routing protocol.

2.1.5 COOJA

Cooja is a java-based simulator developed for simulations of sensor nodes running the operating system Contiki.

Contiki [12] is another open source operating system for embedded systems and wireless sensor networks written in C language. It provides both an IP layer and a low-power radio communication mechanism.
Contiki can be run on several different hardware platforms. As for TinyOS, the *telosb mote* (microcontroller MSP430 with radio chip CC2420) is the most used device.
With regard to the used protocols, it employs a 6LoWPAN mechanism to compress and exchange frame packets over 802.15.4 radio links.

The conceivers of COOJA claim that it can work on different levels – therefore COOJA is also called “cross level simulator” [13]. Other simulators, i.e. ns2 [14], TOSSIM [15] and Avrora [16], can use only some layers: the first implements the “networking” level without touching the hardware properties; the second develops the “operating system” level, simulating exclusively the behaviour of TinyOS; the third implements the “machine code instruction set” level, supporting only AVR microcontrollers.
A short description of levels is reported below.
The “networking” level is mainly used by the developers of routing protocols, where the hardware statements are left out. The level mainly manages the radio propagation and the specific sensor nodes in order to replace them with abstract Java implementations so that there is no connection with the operating system.
The “operating system” level has the task to simulate the executing native operating system code. So the aim is to test and evaluate the changes introduced to the Contiki

libraries or to those of other OSs.
Finally, the "machine code instruction set" level allows nodes having different underlying structures to simulate them using java-based microcontrollers, instead of a compiled Contiki system.

Cooja supports all the described levels and for this reason it can simulate the operating system TinyOS, required to implement 6LoWPAN and RPL, and exchange radio packets between emulated nodes and java-based nodes.
The messages can be sent on different radio mediums; actually the simulator proposes four wireless channels that are *No Radio Traffic, Unit Disk Graph Medium (UDGM) – Constant Loss*, *Unit Disk Graph Medium (UDGM) – Distance Loss* and *Directed Graph Radio Medium (DGRM)*.

No Radio Traffic does not permit the radio communication on the channel and therefore cannot be employed to simulate WSNs.
UDGM – Constant Loss is a wireless channel model where the transmission range is modelled as an ideal disk and all nodes behind it do not receive packets, while the sensors within the transmission distance receive all the messages.
The predefined maximal transmission range is multiplied by the ratio of the current output power to the maximal output power of the simulated device and the resulting transmission power is compared to the distance in the simulation. For example, if the transmission range of the mote is 200 m and the current output power is half of the maximum, the disk has a radius of 100 m.
UDGM – Distance Loss is a radio medium similar to the previous one but that extends it in two ways. First, the interferences are considered and, in case of interfered packets, they are lost due to the interference distance that is higher than the transmission distance. So, all communications running at the same time are deleted leading to an unreal behaviour. Second, the success ratio of the transmission and reception can be set: a packet is transmitted or received on the basis of two probabilities, SUCCESS_RATIO_TX (if unsuccessful, no device receives the packet) and SUCCESS_RATIO_RX (if unsuccessful, only the destination of the packet does not receive it).
DGRM is a model that creates the topology of nodes through edges. It is mainly used

to set the transmission success ratio and the propagation delay on the asymmetric links.

The configuration of COOJA is flexible so that many parts of the simulator can be easily replaced or extended with additional functionality. Example parts are the radio mediums just described, interfaces and plugins.
The interfaces represent some properties of the node, for example the position, the radio, the buttery or the serial port.
Instead the plugins are used to interact with a simulation. They often provide the user with a graphical interface to observe something interesting in the simulation, but the tools may be used without GUI, running COOJA by the terminal. The principal plugins are the following: *Simulation Visualizer*, *Timeline*, *Log Listener* and *Contiki Text Editor*.
Simulation Visualizer simply permits to configure the topology of the network. The user can *drop & drag* the nodes, change the settings of the radio medium (transmission and interference range for example) and show some useful information of the nodes i.e. ID, IP Address, Log output, Radio Traffic, Mote type, Mote attributes, Radio environment and LED states.
Timeline displays, instant for instant, the state of the radio for each simulated node through different colours: on (grey), off (no colour), packet transmission (blue), packet reception (green) and interfered (red). To obtain more details just right click on timeline of interest.
Log Listener analyses every radio connection established between nodes of the simulation, reporting on the GUI the time, the source node, the destination nodes and the payload of the packet sent during the connection.
Lastly, *Contiki Text Editor* controls and automates the execution of the simulations by scripts, written in Java Script. They permit to start and stop the simulation, set the simulation timeout, add and remove nodes dynamically, capture the output of the sensor nodes and interact with COOJA plugins.

Once simulation stops, the user can save it (including used plugins) in files with format *csc*. The configuration file presents a XML structure, making easy the change of some parts of the simulation.

2.2 Scientific Community Works

The SICS, Swedish Institute of Computer Science, has developed several projects concerning the use of COOJA simulator in order to analyse the data communication between nodes in a Wireless Sensor Network.

Considerable work has been done in [13], where the authors of the publication present the structure of COOJA based on cross level platform. This architecture permits a novel type of wireless sensor network simulation that enables simultaneous simulation at different levels: the network level, the operating system level and the machine code instruction set level. The feasibility of the cross-level simulation is demonstrated by the obtained advantages in terms of effectiveness and memory usage and by the possibility for the user to combine simulated nodes from different abstraction levels.

This result is obtained with the study on simulators that operate for a single layer i.e. NS-2 as *network* simulator, TOSSIM as *operating system* simulator and MSPSim [17] as *machine code instruction set* simulator.

The latter is an extensible simulator for MSP430 microcontrollers-equipped sensors in order to simulate the chip instructions, trying to reduce development and debugging time. Therefore, MSPSim enables testing without access to the target hardware. The simulator has been designed to be incorporated in COOJA.

Various researches have been carried out by SICS together with other research centres about COOJA-MSPSIM.

An interesting work has been elaborated with the University of Bonn and Frankfurt for testing the interoperability of the cross level simulator [18].

In practice, COOJA-MSPSIM has been modified to permit WSN simulations with sensor nodes that emulate different embedded operating systems. Their experiment runs Contiki and TinyOS nodes together in a single simulation and demonstrates the complete interaction between them to exchange networking packets.

Also the University of Lübeck (Germany) collaborated with the Swedish Institute to improve COOJA simulator [19]. In this case the research centres have incorporated in COOJA a system to monitor, capture and record the radio interference at runtime since it affects the reliability and robustness of wireless communications. The

captured data are used to simulate and study the impact of realistic radio interface on sensornet communications and routing trees.

COOJA has been subjected to other important extensions to permit it to simulate each aspect of a WSN; for example, most simulators have not provided support for visualizing the power consumption of a sensor network and have not allowed a complete integration between the development environment for embedded operating systems (i.e. CONTIKI and TinyOS) and the simulator. To reach the first goal, [20] and [21] present a plugin developed from SICS, *TimeLine* that makes easy to visualize power consumption and also inspect network traffic and synchronization between sensors. These results have been obtained through the demo illustrated in the paper, where the plugin has been used to test mechanisms and protocols used by Contiki nodes. Instead, to solve the second inefficiency, [22] illustrates a solution to bridge the gap between development and simulation. In other words, the paper demonstrates the advantages of a link between YETI, a development environment for TinyOS, and COOJA. The necessary steps are mainly four: *simulation setup* that creates the scenario in COOJA, configuring network topology and installing the operating system in the nodes; *debug session* that has the function to launch a new session in YETI, starting a TCP connection for each connected node in the simulation; *breakpoints* that allows to create a link between the breakpoints in the YETI source code and COOJA through *debug session*; *Read/modify node state* that registers and modifies the node state in YETI debugging point every time a breakpoint is reached by a node.

Cooja is not the unique simulator for WSNs; actually, a lot of tools are used to analyze the exchange of packets on wireless channels. Therefore, [8] proposes an interesting research to analyze channel models, energy consumption models, physical, MAC and network layer protocols supported by the simulators. To do this, some applications for sensor nodes have been developed to compare the WSN tools to the reality. After analyzing the results obtained from experiments, the work has arrived at the conclusion that the simulators are not reliable enough. Thus the thesis demonstrates how the performance is influenced by a correct configuration of the radio medium.

2.3 Original Contribution

The works, illustrated in the previous section, show the state-of-the-art concerning the analysis of low-power sensor networks through COOJA. The simulator contains interesting plugins that allow the user to know important characteristics about a radio connection, i.e. radio status and type of the packet for example, but it does not dispose of tools that show metrics about each link, such as the number of frames losses in reception by the root, the evolution of the queue for a *relay* node or the number of frames correctly transmitted from a sensor node to the parent.

Without this information, it is difficult to check reliability of the wireless links and therefore, in case of faults on the network, there are not the needed instruments to act.

In the next chapter the development of a COOJA plugin is presented. This plugin allows you to solve the actual inefficiencies of the simulator with the aim to measure application traffic when IP is in use.

3. Plugin Development

The chapter presents the needed steps to develop the COOJA Metrics Collection Tool.

Firstly, the application field of the project work is discussed, analyzing the network application that the metrics collection tool will test.

Secondly, the COOJA architecture is shown, illustrating the java classes that will be used to create the plugin.

Thirdly, the general functioning of the plugin will be analyzed. In details, this section will describe the following features: the realization of the metrics, the way whereby the plugin captures the metrics data and the design of the component *Frame-IP* to capture the metrics of the second and the third IP Stack layer.

Fourthly, the plugin basic architecture is illustrated through the principal java classes, which capture and elaborate the metrics.

Once the overview is finished, the chapter analyses the processes that allow the metrics collection for each layer of the IP Stack. For the physical layer the dissertation will present the way to generate the radio packets (paragraph **3.5**) and the methods that permit the plugin to capture the node and the couple metrics. For the data-link and the networking layer, an extension to the COOJA simulator will be presented to make possible the metrics collection in that case (paragraph **3.6**). After, for the data-link layer, the chapter shows the mode to generate frame packets and the methods that consent the plugin to capture the node and the couple metrics (paragraph **3.7**). Instead, for the networking layer, the chapter presents the way to generate datagram packets and the policy to capture the metrics (paragraph **3.8**).

3.1 Application Field

RPL is the routing protocol for 6LoWPAN networks with the aim to route the application traffic toward the sink node. The project work has the goal to analyse applications that use this routing protocol in order to detect general metrics for the

Wireless Sensor Networks; in details, the metrics collection plugin will test a particular TinyOS application, TestRPL, which can be downloadable from TinyOS SVN repository [2].

A TinyOS application, like all the TinyOS code, is written with the programming language *nesC* [7], which is C with some additional language features for components and concurrency.

The components are assembled, or wired, to form an application executable; they are divided in two types i.e. *configurations* and *modules*.

Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others, while *modules* have the goal to implement one or more interfaces.

The provided interfaces represent the functionality that the component supplied to its user in its specification; instead the used interfaces represent the operations that the component needs to perform its job.

In our case *TestRPLAppC* represent the *configuration* component, while *TestRPLC* represent the *module* component.

The first file contains the links between the used interfaces by the module and the components that realize them; instead, the second document shows the implementation of the interfaces in order to obtain a network application that uses RPL to route the application traffic.

Below the essential structure of the module is reported. More information is presented in the section **A** of the appendix.

- Definition of the routing tree root
- Definition UDP port for the data transport
- Start of the routing protocol RPL and of the application traffic
- Realization procedure that manages the packets reception
- Definition of the length of the experiment

[2] http://code.google.com/p/tinyos-main/source/browse/ - svn%2Ftrunk%2Fapps%2Ftests%2FTestRPL%2Fudp

- Realization procedure that handles the sending of application packets to the root

- Creation procedure that stops the simulations and prints the results

TestRPL, like every TinyOS program, requires a *makefile* to connect the application to the build system. This file includes essential directives for the *nesC* compiler in order to compile the program without errors.
In the specific instance, the TestRPL Makefile contains the name of the top-level component of the application (COMPONENT=TestRPLAppC) and other important options to manage the application and routing traffic:

- *CFLAGS+=-DCC2420_DEF_CHANNEL=X* defines the channel used by CC2420 radios;

- *CFLAGS+=-DRPL_ROUTING-DRPL_STORING_MODE-I$(LOWPAN_ROOT)/tos/lib/net/rpl* enables the RPL protocol in storing mode;

- *CFLAGS+=-DBLIP_L2_RETRIES=X -DBLIP_L2_DELAY=T* sets blip parameters in order to reduce media overload;

- *CFLAGS+=-DPACKET_INTERVAL=XUL* sets the packet generation interval where 1UL stands for 1/1024 second. Therefore the rate of transmission of the motes is equal to 1024/PACKET_INTERVAL (if PACKET_INTERVAL is set to 1024UL, then the rate transmission is 1 packet/s);

- *CFLAGS+=-DPACKET_NUMBER*=X establishes the number of packets that the clients send to the root.

Once MakeFile and TinyOS application is defined, the next step is to compile the program using the terminal command *make telosb blip*.
Telosb and blip are the targets of the command *make* and they specify to *NesC compiler* that TestRPL will be installed on telosb motes and the library *blip* will be required to realize a 6LoWPAN network (refer to section **2.1.4**).
If the process concludes without errors, *NesC compiler* generates the program binary image (with extension .exe), which will be used to simulate wireless sensor networks through COOJA simulator.

3.2 Cooja Architecture

In this section the structure of the COOJA simulator is analysed.

The software is written in JAVA language and it is included in the Contiki project; [3] to access to COOJA source code you have to follow this path: *contiki/tools/cooja/java/se/sics/cooja*.

Once *cooja* directory is reached, you can change any part of the simulator; below it is reported below a short description of the JAVA classes that will interact with the plugin.

- *interfaces/Radio*: *Radio* allows to simulate the behaviour of a transceiver, defining methods that manage the status, the position and the power of the radio and the sending and receiving of the messages.
- *Mote*: The file abstracts the features of a mote. Its methods return the ID of the node, the mote type i.e. *telosb*, *telosa* and *micaz* [7] and the mote memory.
- *Positioner*: The class is used to determine the initial positions of motes.
- *RadioConnection*: It represents a radio connection between a single source radio and any destination and interfered radios. This file is used by the radio medium classes to create the application traffic.
- *ConvertedRadioPacket*: It is the most used radio packet in COOJA. In order to support cross-level communication, all the radios must support the transmitting and receiving of this packet.
- *Simulation*: The file manages the execution of each simulation. It is an observable class and therefore if the simulation undergoes updates (for example the insertion, the deletion of motes and the change of simulation state), every COOJA plugin can observe them.
- *RadioMedium*: This class is abstract and so every COOJA radio medium has to implement it. The registered radios in this medium can both send and receive radio packets.
- *radiomediums/AbstractRadioMedium*: It provides basic functionality for implementing radio mediums, it handles the radio registrations, radio loggers, active connection between nodes and it observes all registered radio interfaces.

[3] git command to download Contiki - git://contiki.git.sourceforge.net/gitroot/contiki/contiki

The registered radios signal power is updated whenever the radio medium changes; there are three fixed levels, i.e. no traffic, noise and data heard.

- *radiomediums/UDGM*: Unit Disk Graph Radio Medium abstracts radio transmission range as circles. It uses two different range parameters, one for the transmissions and one for the interfering with other radios and transmissions; both radio ranges grow with the radio output power indicator. Moreover, the class uses two different success ratios to simulate transmission and reception random errors on the channel.
- *radiomedium/DirectedGraphMedium*: The radio medium is implemented through a edges structure, where each link supports propagation delays and transmission success ratios.
- *plugins/analyzers/PacketAnalyzer*: The file analyses each packet sent via radio medium, getting information about payload, size and much more. All the analysers will extend this class.
- *plugins/analyzers/IEEE802154Analyzer*: The analyser studies the packets through the 802.15.4 layer, distinguishing between the data message and the ACK message, which form together a "frame": for each frame the class identifies the source address and the destination address.
- *plugins/analyzers/IPHCPacketAnalyzer*: The class scans each datagram (IPV6 packet), sent from the clients to the root node. In this case the analysis is performed using the 6LoWPAN protocol, determining the following data for each message: IPv6 source address, IPv6 destination address, UDP source port and UDP destination port.
- *plugins/analyzers/ICMPv6Analyzer*: The file has the function to analyse the routing traffic. Since the application traffic is routed by RPL protocol, the class divides the messages in the following categories: DIS packet, DIO packet, DAO packet and DAO ACK packet.

The metrics collection plugin will be included in the directory *plugins*.

3.3 Plugin General Functioning

This section presents the basic features of the metrics collection tool. After, in next paragraphs, the implementation of the java components will be shown.

The plugin has the aim to capture the metrics data of the 6LoWPAN networks.
The metrics are divided in two main typologies: node metric and couple metric.
Node metric is applied to a mote of the simulation, while couple metric studies the behaviour of a motes couple during the experiment.
Then, each single metric has its own scope, that is the IP stack layer; the metrics collection tool analyses the physical, the data-link and the networking layer.
All the metrics are reported below.

Node Metric

Physical Layer	*Number of connections successfully sent*
Physical Layer	*Number of connections not successfully sent*
Physical Layer	*Number of connections successfully received*
Physical Layer	*Number of connections not successfully received (collision)*
Physical Layer	*Number of connections not successfully received (error on the channel)*
Physical Layer	*Power level*
Data-Link Layer	*Number of frames successfully sent*
Data-Link Layer	*Number of frames not successfully sent*
Data-Link Layer	*Number of frames successfully received*
Data-Link Layer	*Number of frames not successfully received*
Networking Layer	*Datagrams queue*

Couple Metric

Physical Layer	*Number of connections successfully received*
Physical Layer	*Number of connections not successfully received (collision)*
Physical Layer	*Number of connections not successfully received (error on the channel)*
Physical Layer	*Transmission Window*
Data-Link Layer	*Number of frames successfully received*
Data-Link Layer	*Number of frames not successfully received*
Networking Layer	*Delay end-to-end*

The metrics of the first layer analyse the number of radio packets sent on the radio medium. Each radio packet is also called radio connection. These metrics have the function to detect radio collisions and eventual random errors on the radio channel.

The metrics of the second layer determine the number of frame packets exchanged correctly between the nodes, highlighting eventual losses.
Finally, the metrics of the third layer study the trend of the queues for the *relay* nodes, which act as routers to forward the application traffic. Moreover, these metrics analyse the delay end-to-end between a client and the application server.
More information, about the metrics, is illustrated in the next paragraphs that analyse the implementation of the plugin components.
All the metrics must be captured by the plugin. To reach this goal, the plugin includes a dynamic capture component that collects the data required by the metrics.
For the first IP Stack layer, this component listens each new radio packet, generated by the radio medium (UDGM or DGRM). For each connection the capture component checks if the packet includes information required by the metrics. If the control is positive, the metric data is stored in a container that maintains it, so that the plugin can create XML files and graphics, representing the metrics, when the experiment ends.
For the second and the third IP Stack layer, the capture phase contains a new module. This component has the function to generate frames and datagrams from the radio packets exchanged on the radio medium, since COOJA does not consider this type of messages.
Generally, each frame consists of two radio packets: the DATA message containing the application data and the ACK message to verify if the DATA message is correctly received by the destination.
Instead, each datagram is made up of just described frames.
Once a frame or a datagram is generated, the dynamic capture module matches it with the metrics selected by the users. If the message contains the required information, the metric data is stored in a container for the same reasons seen for the physical layer.
The general functioning is represented with the following chart, realized with ARGOUML tool [4].

[4] http://argouml.tigris.org/

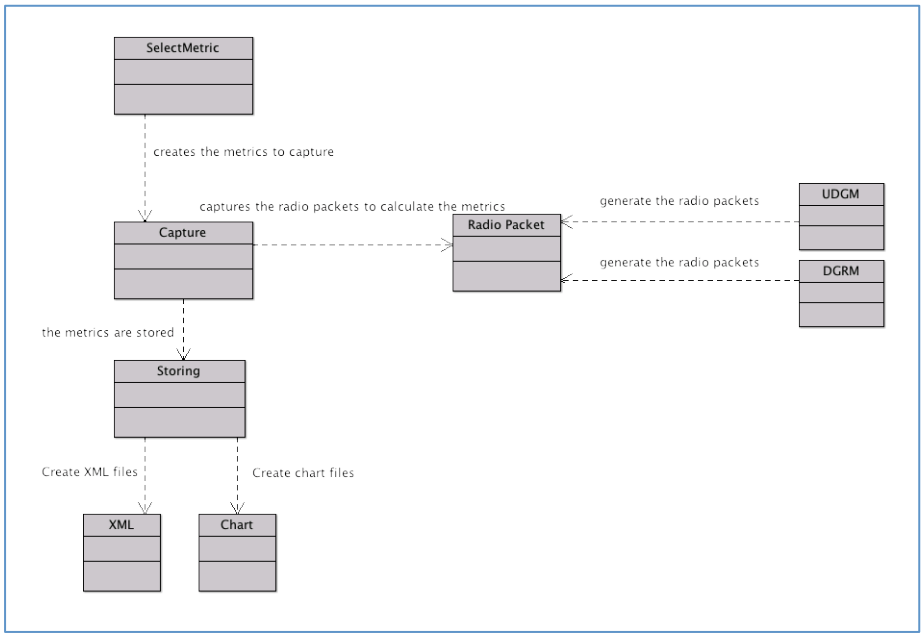

Figure 1: Diagram of classes – Main components

3.4 Plugin Basic Architecture

This section presents the basic structure of the metrics collection tool. In other words this part analyses the development of the principal java classes that support the entire plugin, omitting the specific components for the metrics collection.

The basic architecture is showed below with the relations between the objects through the diagram of classes.

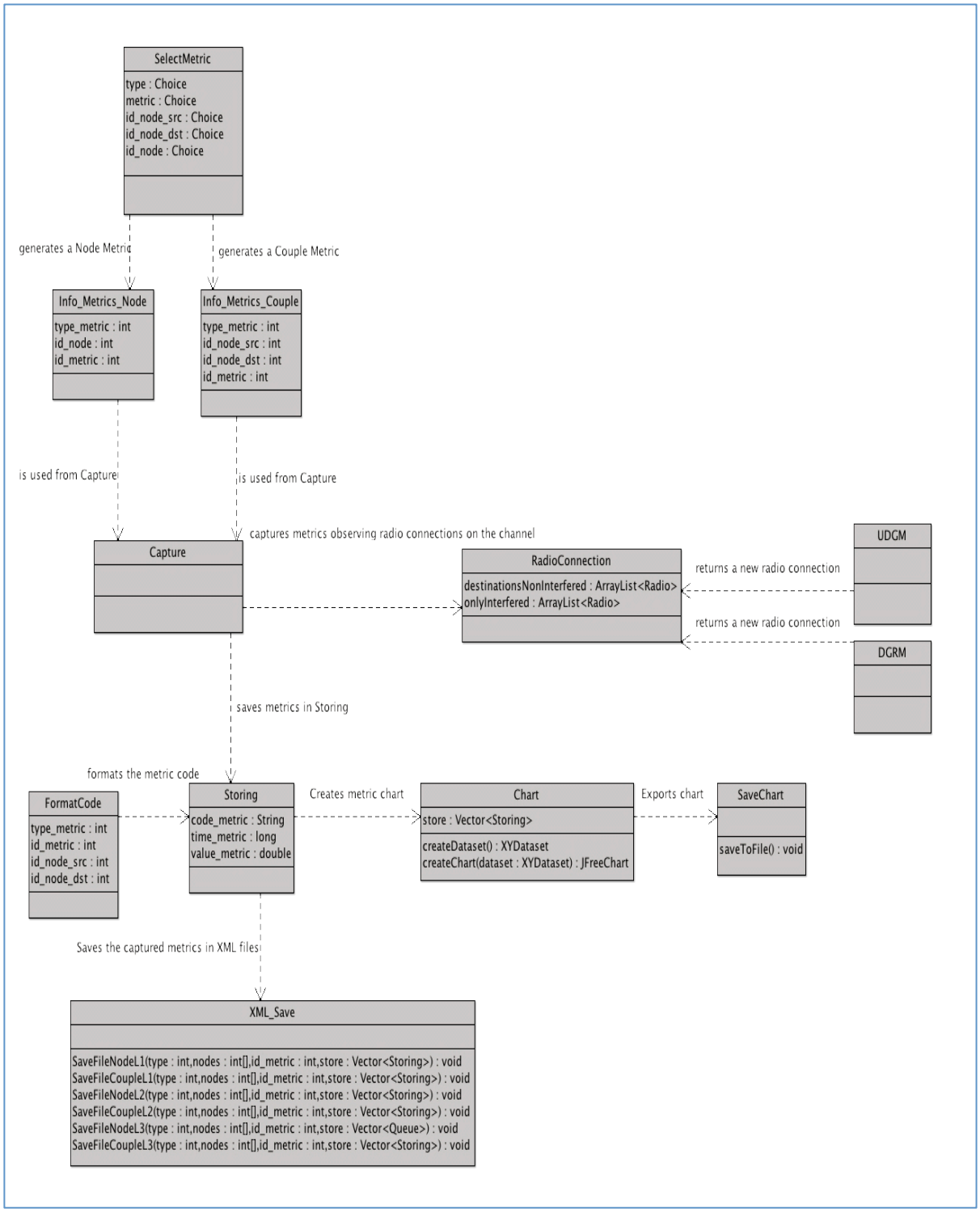

Figure 2: Diagram of classes – Plugin Basic Architecture

The chart does not display the main class, *Plugin_Metrics*; the cause comes from its function to manage all files reported in the diagram. So the next lines will illustrate

the development of the plugin, paying attention to the interaction between *Plugin_Metrics* and the other classes.

Plugin_Metrics is the component representing the graphical interface of my tool. The GUI allows the user to choose the desired metrics, classified according to the IP Stack layers, through three buttons, one for each layer, placed in the lower part of the window (see Figure 3). Clicking on a button a new frame appears, the class *SelectMetric*, allowing you to choose a precise metric by the drop-down menu, implemented by the java class *Choice* [5]. In details, the frame consents you to decide if you want study the metric of a node or the metric of a couple of nodes (*type*): in the first case you select the interest node (*id_node*) and the metric (*metric*); in the second case you select the interest source node (*id_node_src*) and the interest destination node (*id_node_dst*) that form the couple and finally the metric (*metric*).

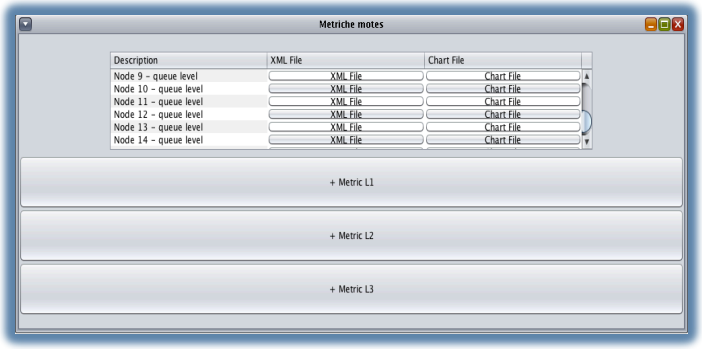

Figure 3: Graphical interface of the Metrics Collection Tool

The *metric* field value changes depending on the selected button.

Once metric has been selected, *SelectMetric* notifies to *Plugin_Metrics* the insertion of a new metric. The notification between different objects is possible through two classes of the packet *java.util* [6] that are *Observable* and *Observer*. The class *Observable* is characterized by a particular method, which contains the following commands: *setChanged* that marks the *Observable* object since it has been changed; *notifyObserver* that notifies its change to all observers of the *Observable* object.

In this case *SelectMetric* is the *Observable* object, while *Plugin_Metrics* is the *Observer* object that receives the metrics data from *SelectMetric*.

---

[5] Java class – Documentation in http://docs.oracle.com/javase/6/docs/api/java/awt/Choice.html

[6] Java packet – Documentation in http://docs.oracle.com/javase/6/docs/api/java/util/package-summary.html

Then the metric data is saved as an *Info_Metrics_Node* object or as an *Info_Metrics_Couple* object, depending on the *type* of metric. Also the selected metrics are reported in a table, above the three buttons.

Concluded the metrics selection phase, the capture mechanism is presented with the function to analyse the radio packets sent on the radio medium in order to find the metrics values during a COOJA simulation.

To detect each message exchanged between the nodes, the JAVA classes *Observable* and *Observer* are used. In this work only two radio mediums are considered, i.e. UDGM and Directed Graph Medium, because the others classes are not fully implemented. Therefore an *Observable* object is installed in UDGM.java and in DirectedGraphMedium.java; it notifies to main class, the *Observer* object, the making of a new connection between a source node and a set of destination motes. In this way *Plugin_Metrics* receives the last radio connection through a *RadioConnection* object and so it can start the capture phase to calculate the metrics by the *Capture* object.

Whenever the value of a metric undergoes an update due to a new connection, this change is registered in the class *Storing*. Each *Storing* instance is identified by a code (*code_metric*), a time instant (*time_metric*) and a value (*value_metric*).

The *code_metric* is realized through a specific class, *FormatCode*, which has the aim to create a unique identifier for each *Storing* record. The code development follows a precise logic: if the modified metric type is Node Metric, the code is obtained through a concatenation of the following attributes: type (*type_metric*), metric (*id_metric*) and node ID (*id_node_src*); otherwise, if the modified metric type is Couple Metric, the code is obtained by the same previous concatenation, adding the node ID for the destination mote (*id_node_dst*).

Instead *time_metric* represents the time instant of metric updating and finally *value_metric* indicates the new value of the metric. The capture phase will be discussed in details in the later sections.

Once COOJA ends its simulation, the plugin offers users the possibility to convert the obtained metrics in XML or images files through buttons, located in the table; in this way the collected metrics are not lost when the user runs a new COOJA simulation. To implement this conversion, three java classes are developed: *XML_Save*, *Chart* and *SaveChart*.

XML_save is formed by six methods, two per each IP-Stack layer.

The methods *SaveFileNodeL1* and *SaveFileCoupleL1* convert respectively the L1 [7] metrics applied to a node and the L1 metrics applied to a couple of motes in XML files.

The same logic is followed by the methods *SaveFileNodeL2*, *SaveFileCoupleL2*, *SaveFileNodeL3* and *SaveFileCoupleL3*.

All the methods accept as input arguments the information that identifies a specific metric and the vector containing all the *Storing* instances. More information is illustrated in the section **B** of the appendix.

Instead, Chart and SaveChart have the function to create first a dataset, given the information saved in the class Storing, and then to create the chart through the dataset using the library JFreeChart [8].

3.5 L1 Components

This section illustrates the development of the java components, which have the function to collect L1 metrics.

The L1 metrics analyse radio packets through the physical medium. In this work an 802.15.4 radio link between low-power devices (motes) is considered.

COOJA, UDGM and DirectedGraphMedium (DGRM) are the java classes that implement the first IP-Stack layer.

UDGM uses ideal circles as radio transmission range; instead DGRM realizes the radio medium through edges, implemented with the class *Edge*. Every *Edge* instance consists of two elements: a source radio defined by the *Radio* object; a destination radio made by the *DestinationRadio* object, which establishes for each edge the delay, the success rate and the *Received Signal Strength Indication* (RSSI).

As mentioned in the previous section, UDGM and DGRM use the *Observable* object since they have the function to notify the creation of a new packet to *Plugin_Metrics* (*Observer* object). To create a new packet, the classes implement the method *CreateConnections*. Once the radio connection has been realized, the plugin (*Plugin_Metrics*) starts the phase capture in order to collect the L1 metrics, distinguishing the node metrics from the couple metrics.

---

[7] L1 – Physical layer

[8] Java library - http://www.jfree.org/jfreechart/

To obtain more information about implementation layer, you refer to the section **D** of the appendix.

The diagram of classes for the L1 components is reported below.

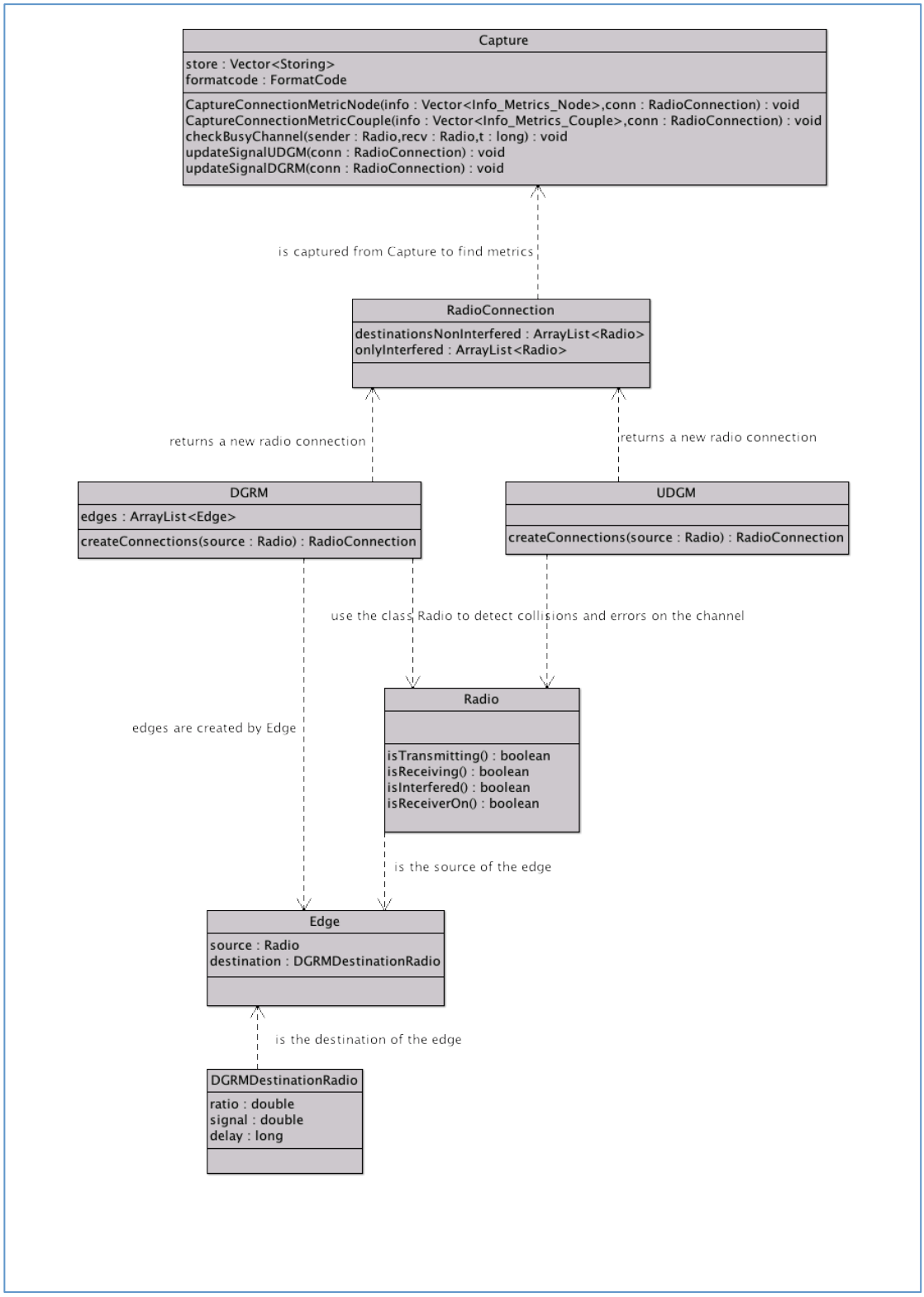

Figure 4: Diagram of classes – L1 components

3.5.1 Generation Radio Connections

CreateConnections generates a *RadioConnection* object, given a source node.
To reach this result, the method attempts to create a connection between the source node and all the reachable destinations through the source radio transmission range. To do this, the java code contains some controls in order to determine possible collisions or random errors on the channel for each possible destination, using the methods of the class *Radio*.

- *Check Channel*: the control fails if the source and destination radio use different channels.
- *Check Radio State*: the control fails if the destination radio is switched-off (***Radio*** **method** -> *isReceiverOn*).
- *Check Random Error*: the control fails if a random error occurs on the channel. In this case the destination node is added to *RadioConnection* object, like interfered node.
- *Check Radio Receiving*: the control fails if the destination radio is actively receiving. Also in this case the destination node becomes an interfered node (***Radio*** **method** -> *isReceiving*).
- *Check Radio Interference*: the control fails if the destination radio is interfered in another connection. The node becomes interfered (***Radio*** **method** -> *isInterfered*).
- *Check Radio Transmitting*: the control fails if the destination radio is actively transmitting. The node becomes interfered (***Radio*** **method** -> *isTransmitting*).

If the controls do not fail, the destination node is included in *RadioConnection* as destination; otherwise as interfered node.

Once controls have been carried out for each possible destination radio, the new packet (*RadioConnection* object) and the detected failures in the control phase are notified to the *Observer* object, *Plugin_Metrics*, that starts the capture phase.
The capture phase is implemented by the class *Capture* that processes the L1 metrics.

3.5.2 Capture Phase - Node Metrics

The Node Metrics for the first layer of IP-stack are obtained through the method *CaptureConnectionMetricNode* that is applied to node selected by the user during the metrics selection phase.

CaptureConnectionMetricNode requires the radio connection, the detected failures, and the metrics selected by users, in *Info_Metric_Node* format as parameters. The method is launched each time the *Observable* object notifies a radio medium change. The general functioning of *CaptureConnectionMetricNode* is based on the comparison between the last radio connection and the metrics, selected by users: if *RadioConnection* object contains the information required by a specific metric, the metric value undergoes an update; otherwise, the metric value does not undergo any change.

Each capture can lead to add a new record for the class *Storing;* this case occurs when the metric node is included in the last radio connection as a source node, as a destination node or an interfered node.

In details, the method manages each single metric through a structure "*if -else*", outlined below.

```
1  for each metric
2          if (id_metric = 1)
3                process   Number of connections successfully sent
4          else if (id_metric =2)
5                 process   Number of connections not successfully sent
6          else if (id_metric = 3)
7                 process   Number of connections successfully received
8          else if (id_metric = 4)
9               process   Number of connections not successfully received (collisions)
10         else if (id_metric =5)
11                process   Number of connections not successfully received (errors on the channel)
12         else if (id_metric = 6)
13                process   Power level
```

The first metric (lines 2-3) represents the number of radio connections without transmission errors. In this case the metric value increases when the following events occur: first, the radio connection has to include at least a destination (therefore the connection reaches the destination); second, the metric node ID must be equal to the source node of the current connection.

The second metric (lines 4-5) defines the number of radio connections not correctly sent due to transmission errors. In order to detect this fail, the method runs the next control: if the metric node ID is equal to the source node of the current connection and if the radio connection includes any destinations (so the packet is lost during the transmission) then the metric increases of 1.

The third metric (lines 6-7) counts the number of radio connections successfully received without reception errors.
If the metric node is a destination sensor for the last radio connection, then the metric increases of 1. Therefore the node correctly receives the radio packet.

The fourth metric (lines 8-9) counts the number of radio connections not successfully received due to collisions. To process this metric, the failures detected during the creation of the radio connection are used. In order to update the metric value, the metric node is required to be interfered due to the failure of at least one of the following checks: *Check Radio State*, *Check Radio Receiving*, *Check Radio Transmitting* or *Check Radio Interference* (see **3.5.1**).

The fifth metric (lines 10-11) calculates the number of radio connections not successfully received due to random errors on the channel. As for the previous metric, the control phase during the creation of a new radio connection is used. If the *Check Random Error* fails (**see 3.5.1**), the metric value increases of 1.

Finally, the sixth metric (lines 12-13) determines the radio power level of the metric node, for each time instant of the simulation. In this case the method *CaptureConnectionMetricNode* calls a specific method to process the desired metric depending on the selected radio medium. With UDGM the class *Capture* runs

UpdateSignalUDGM, while *Capture* executes *UpdateSignalDGRM* when COOJA uses DGRM.

Both methods require the last radio connection as parameter, but the calculation of radio power level is different.

For the method *UpdateSignalUDGM*, the radio signal strength depends on the distance between the source radio and the destination radio and on the transmission maximum distance of the source radio. Detailed information is presented in the section **C** of the appendix.

For the method *updateSignalDGRM*, unlike the previous method, the power level is set to -10 dB for each node of the *RadioConnection* instance.

3.5.3 Capture Phase - Couple Metrics

In order to capture the couple metrics, the method *CaptureConnectionMetricCouple* is developed.

The method requires the same parameters of *CaptureConnectionMetricNode*.

The only difference relates the metrics chosen by the user; in this case the *Info_Metrics_Couple* object is used. The basic functioning does not present significant differences; the changes relate the way to process the metric.

The method is structured through a simple architecture that matches each metric with the last radio connection.

```
1  for each metric
2          if (id_metric = 1)
3                process   Number of connections successfully received
4          else if (id_metric = 2)
5                process   Number of connections not successfully received (collisions)
6          else if (id_metric =3)
7                process   Number of connections not successfully received (errors on the channel)
8          else if (id_metric = 4)
9                process   Transmission window
```

The *Number of connections successfully received* (lines 2-3) determines the amount of radio packets correctly sent from the metric source node and received without errors by the metric destination node. Therefore for each new *RadioConnection* the method checks:

- if the connection source coincides with the metric source node;
- if the metric destination node is included in the radio connection as a destination node (not interfered).

If both controls do not fail, the metric value increases of 1.

The second metric (lines 4-5) counts the number of connections correctly sent from the metric source node and not received from metric destination node due to radio collisions. In this case the needed controls are the following:

- Checks if the connection source coincides with the metric source node;
- Checks if the metric destination node belongs to the interfered nodes set of the last radio connection through *Radio* checks, that are *Check Radio State*, *Check Radio Receiving*, *Check Radio Transmitting* or *Check Radio Interference* (see **3.5.1**);

If at least one of these methods fails, the metric value is updated.

The third metric (lines 6-7) processes the number of radio packets sent with success from the metric source node to the metric destination node. At reception level these connections fail due to random errors on the channel.
To increment the metric value, the necessary condition is to have the metric source node as packet transmitter and to have the metric destination node as interfered node of the radio connection by *Check Random Error* (see **3.5.1**).
If the condition is respected, the metric value increases of 1.

The last metric (lines 8-9), *Transmission Window*, determines for each new *RadioConnection* the communication state for a couple of nodes.
To reach this result two methods are developed: the *checkBusyChannelUDGM* and the *checkBusyChannelDGRM* that require the *Radio* interfaces of two motes as parameters.

The methods process the communication state through the same checks carried out during the creation of the radio connection: if two radios are able to communicate, then *checkBusyChannel* sets the value 1 in *Storing* in the field *value_metric*; otherwise, it sets the value 0.

3.6 Implementation Data-Link Layer and Networking Layer in COOJA

Actually COOJA does not dispose of classes that implement the data-link layer and the networking layer of the IP Stack; in fact the simulator contains only java classes to implement the physical medium, such as UDGM or DGRM.

Since the plugin has the aim to detect metrics for each IP Stack layer, this paragraph presents the development of java classes that allow the implementation of frames and datagrams.

These classes are positioned in the directories cooja/*sixlowpan* and *cooja/plugins/analyzers*. More information is reported in the section **D** of the appendix.

In the first directory the files are divided in two subdirectories: **Datagrams** and **Nodes**.

Datagrams contains the source code that realizes IPv6 packets.

The main class is *Datagram*. A *Datagram* object is uniquely identified through two attributes, *ID_Source* and *ID_Thread*; *ID_Source* establishes the datagram sender, while *ID_Thread* specifies the amount of IPV6 packets, sent by datagram sender.

Furthermore, a *Datagram* object is represented through the packet payload size, the headers, the hops and the frames.

1. The packet payload size is an integer value that represents the packet weight in bytes.

2. The headers contain control information for the correct network functioning.
   A header for each IP Stack layer is developed: 802.15.4, IPv6, CTP, UDP, TCP and COAP. Of these headers, 802.15.4, IPv6 and UDP have been implemented in COOJA.
   FrameHeader (802.15.4 header) identifies the source and the destination of the datagram by the MAC addresses. For simplicity a MAC address is defined

exclusively by the ID of the node, in integer format, without the personal area network ID (0x22).

IPV6Header (IPv6 header) has the function to set important network features to the datagram: IPv6 address for the source and destination node, the time instant in which the packet is sent, the time instant in which the packet is received and the datagram type. The source and destination IPv6 addresses are represented as strings, while the time instants, just mentioned, are *Long* [9] numbers that represent the time in microseconds. Finally IPv6 header permits to define the packet type. Two are the categories, RPL packets or traffic packets: RPL packets are divided into DIO, DAO and DIS; traffic packets use the UDP protocol for the application data transport.

UDPHeader (UDP header) specifies the source UDP port and the destination UDP port, used by the datagram, for the data communication. The class has two attributes, *PortSource* and *PortDestination*, which can be updated with the appropriate methods.

3. A datagram reaches the root of the 6LoWPAN network through some hops.
   The hops coincide with particular nodes called *relay* nodes, which have the function to forward the datagram from a data-link to another; therefore the class *Hop* is designed. A *Hop* object is identified by the *relay* node ID and by the transmission delay, which indicates the time taken by the datagram to cross the *relay* mote. Each *Hop* instance is linked to the datagram by the attributes *ID_Source* and *ID_Thread*, which identify uniquely an IPv6 packet.

4. Each datagram is made of one or plus frames: if an IPv6 packet passes through three data-links, it consists of three frames.
   A *Frame* object is specified by the frame source and by the frame destination with the attributes *ID_Src* and *ID_Destination*. Moreover, each instance defines the time instant in which the frame is sent and the time instant in which the frame is received. Each *Frame* is associated with the parent datagram by the attributes *ID_Source* and *ID_Destination*.

[9] Java class – Documentation in http://docs.oracle.com/javase/6/docs/api/java/lang/Long.html

The class *Datagram* manages the headers, the hops and the frames through the attributes *headers*, *hops* and *frames*:

- *headers* is the *Vector* of *Header* objects. *Datagram* permits to add a new header to the datagram through the method *addHeader*. Moreover, the class consents to return a single header through the following methods: *getFrameHeader*, *getIPV6Header* and *getUDPHeader*;
- *hops* is the *Vector* of *Hop* instances. The main class permits to insert a new hop to the datagram and to get all *Hop* objects, already added. These actions are implemented through the methods *addHop* and *getHop*;
- *frames* is the Vector of Frame objects, which are managed with the methods *addFrame* and *getFrame.*

Nodes contains the source code that implements the networking features of each sensor node i.e. the MAC address, the IPV6 address, the port used for the data transport, the queue etc.
The main class of **Nodes** is *Node*. A *Node* instance represents a mote. *ID* is the fundamental attribute to identify uniquely a node during the simulation. The class has the function to manage the *datagrams queue* and the *endpoints*.

The *queue of datagrams* is implemented through the class *Queue*.
A *Queue* instance represents a datagram that changes the queue value; each object is defined by the following attributes:

- *ID_Source* and *ID_Thread* identify the datagram;
- *state* indicates the causes that lead to update the queue value;
- *value* indicates the current value of the queue;
- *timestamp* represents the time instant in which a datagram changes the queue value.

Each node has *endpoints*. They are the addresses assumed from the node, depending on the IP Stack layer.
Three endpoints for each node of the simulation are developed: *Frame154EndPoints*, *IPV6EndPoints* and *UDPEndPoints*.

- *Frame154EndPoints* performs the MAC address of the node. *Frame154EndPoints* object is characterized by the attribute *mac_address,* which is an integer value, identifying the node in the 802.15.4 links.
- *IPV6EndPoints* establishes the IPV6 address of the node. The class has two attributes, ipv6 and ipv6_string: the first expresses the node address in integer format, converting the last four bytes of the IPV6 address; the second represents the IPv6 address in string format.
- *UDPEndPoints* defines the *socket* (IPv6 address + UDP port) used by the node to transport the data. The class has a unique attribute, *port*, that establishes the UDP port for the data communication. Instead the IPv6 address is not represented by an attribute, since the class *UDPEndPoints* extends *IPV6Endpoints*.

The class *Node* is responsible for the management of the queue and the endpoints, as mentioned above.

In fact, important methods are developed for the class: *inQueue*, *outQueue*, *checkQueue*, *getQueue*, *addEndPoints* and *getEndPoints*.

- *inQueue* creates a new *Queue* object to represent the entry of the datagram into the queue. If the entry of the datagram does not lead the queue value to reach the maximum threshold, the new instance has *state* 0, else it has *state* 4.
- *outQueue* creates a new *Queue* object whose *state* is 2 and whose *value* is decreased of 1; this *Queue* object represents an outgoing datagram from the queue. If this method is invoked, the object reporting the entry of that datagram in queue undergoes an update of the *state*, passing from 0 to 2.
- *checkQueue* has the function to check all the instances of the class *Queue* in order to delete the datagrams that remain in queue for a time upper to a predetermined timeout. If these datagrams have *state* 0, then the *Queue* object, representing the datagram, undergoes an update of the state, passing from 0 to 3. Instead, if these datagrams have state 4, the *state* passes from 4 to 5.
- getQueue returns all the *Queue* objects.
- addEndPoints allows to add a new address to the node for a determinate IP Stack layer.

- getEndPoints returns an *EndPoints* instance, representative of a node for a determinate IP Stack layer.

The second directory, *cooja/plugins/analyzers*, contains two new files: *Analyzer_802154* and *Analyzer_6LoWPAN*.

Analyzer_802154 is the class obtained changing and simplifying the file *IEEE802154Analyzer* (see **3.2**). *Analyzer_802154* has the aim to return important components of a frame: the sequence number, the source MAC address and the destination MAC address.
The sequence number is obtained extracting the sequence number from a particular byte of the radio packet.
The source MAC address and the destination MAC address are calculated by the method *getMAC*, analysing the bytes of the radio packet in order to get the addresses as array of bytes.

Analyzer_6LoWPAN extracts from a radio packet the networking components required for a datagram. Through the method *getAddress*, the class analyses each byte of the packet in order to get the source IPv6 address, the destination IPv6 address, the source UDP port and the destination UDP port.
The directory *analyzers* contains the file *ICMPv6Analyzer* (see **3.2**). A new method, *analyzeRPL*, has been introduced in this class to distinguish the routing traffic. In other words, the method gets the RPL packet type through an integer number.

- if packet type = DIS, *analyzeRPL* returns 0;
- if packet type = DIO, *analyzeRPL* returns 1;
- if packet type = DAO, *analyzeRPL* returns 2;
- if packet type = DAO ACK, *analyzeRPL* returns 3.

3.7 L2 Components

This paragraph illustrates the metrics collection way for the second IP Stack layer.
The second layer is the MAC layer, which has the aim to control the sending of frames on the physical medium.

A frame is formed by a radio connection or by two radio connections.
The first case regards the multicast packets, where the message is sent to all destinations, reachable by the transmission range.
The second case regards the unicast packets, where the frame is composed by a DATA message and by an ACKNOWLEDGEMENT (ACK) message.
DATA message is a radio packet that contains the information captured by the sensors for the root node: the source and the destination have to belong to the same circle or to the same link, depending on the radio medium used (UDGM or DGRM).
When the destination of the frame receives the DATA message, the node replies to the source through an ACK message to notify the correct reception.
If the source does not receive the ACK message, it relays the DATA message after a *timeout* defined in the Makefile of the application. Considering TestRPL, timeout is equal to 103 ms and the number of retries to send a frame is set to 10 (see **3.1**).
Duplicate frames are not accepted to process L2 [10] metrics. Using the *sequence number* of the frame, these messages for the calculation of L2 metrics will be not considered.

In the next sections, the realization of the frame packets and the way to capture L2 metrics will be discussed. During the reading, java classes and java methods will be appointed again. Refer to the section **3.6** and to the section **D** of the appendix.
Furthermore, in order to make easier the relation among classes, the diagram of classes for the L2 components is reported below.

[10] L2 – Data-link layer

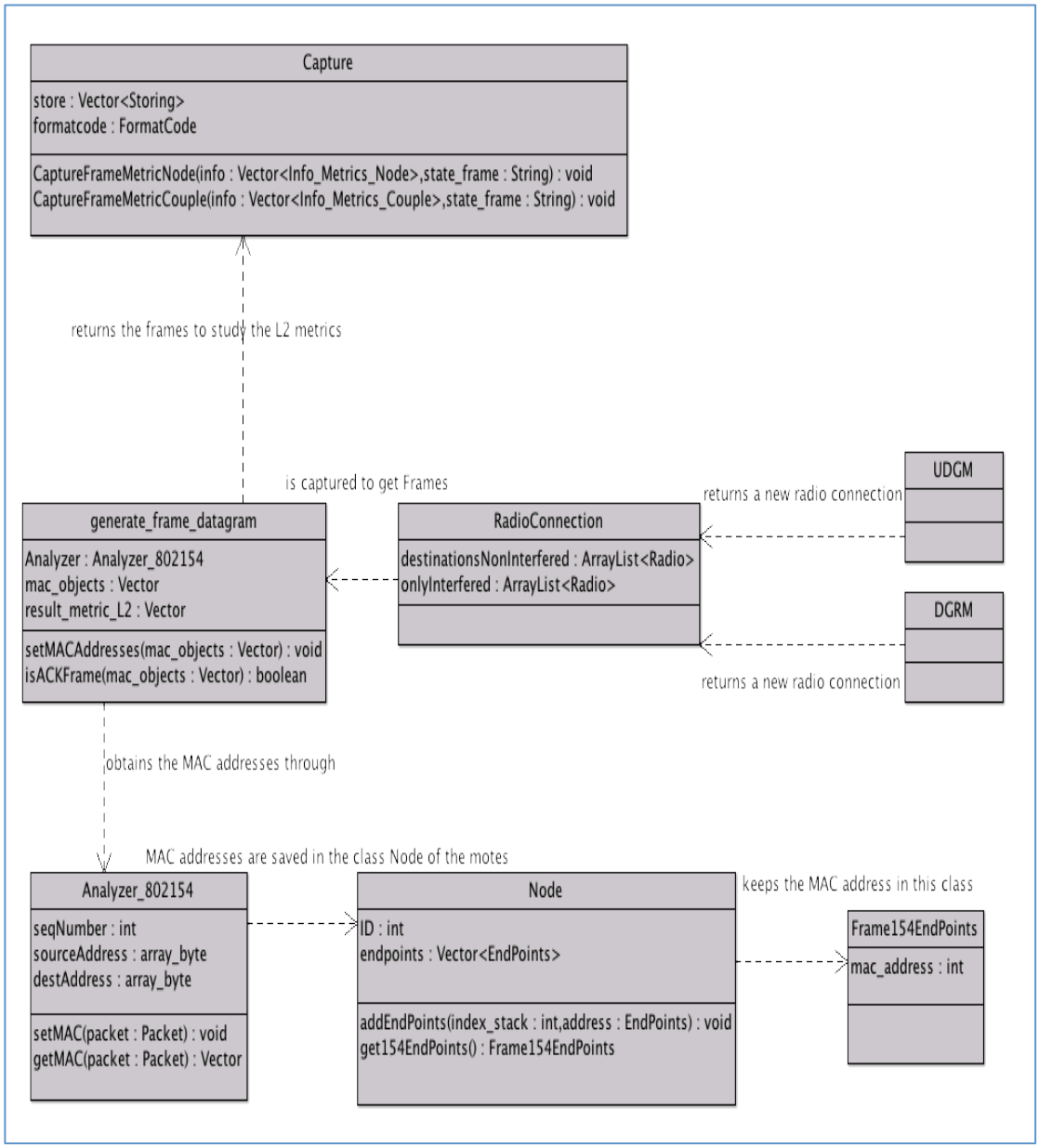

Figure 5: Diagram of classes – L2 components

3.7.1 Generation Frames

As mentioned above, the radio medium generates radio connections during the life of the simulation. In this case the *Observer* object of the radio medium classes is a new class, *generate_frame_datagram*. The file extends the actual COOJA source code, introducing new features in order to generate frames and datagrams within the simulator.

In particular, the class *generate_frame_datagram* collects information to determine L2 metrics through the analysis of each *RadioConnection*, notified from the radio medium.

For each frame this class gathers the frame state, which consists of a transmission without error, a failed transmission, a reception without errors or a failed reception; the frame source node; the frame destination node; the time instant in which the event occurs.

Once frame information is obtained, *generate_frame_datagram* acts from *Observable* object notifying it to the *Plugin_Metrics* (*Observer* object). Then the main class can start the capture phase.

The basic structure of *generate_frame_datagram* is summarized below.

```
1 For each radio connection
2 {
3    conn = radiomedium.getLastConnection()
4    packet = analysis_packet(conn)
5    if (packet reaches the destination and it is not ACK)
6    {
7        set MAC_address_source_node(packet)
8        set MAC_address_destination_node(packet)
9        // collect info frame: transmission frame without errors
10       data_frame.add("TX OK");
11       data_frame.add(time);
12       data_frame.add(MAC_address_source_node)
13       notify(data_frame) to Plugin_Metrics
14       if (packet is MULTICAST)
15       {
16            for each destination
17            {
18                 //collect info frame: reception frame without errors
19                 data_frame.add("RX OK")
20                 data_frame.add(time)
21                 data_frame.add(MAC_address_source_node)
22                 data_frame.add(MAC_address_destination_node)
23                 notify(data_frame) to Plugin_Metrics
24            }
25       }
```

```
26      else if (packet is UNICAST)
27      {
28          if (previous packet was UNICAST)
29          {
30              //collect info frame: reception frame fails – Data message without ACK
31              data_frame.add("RX FAIL")
32              data_frame.add(time)
33              data_frame.add(MAC_address_source_node_previous_packet)
34              data_frame.add(MAC_address_destination_node_previous_packet)
35              notify(data_frame) to Plugin_Metrics
36          }
37      }
38  }
39  else if (packet reaches the destination and it is ACK)
40  {
41      if (frame sequence number is new)
42      {
43          //collect info frame: reception frame without errors -> ACK after DATA
44          data_frame.add("RX OK")
45          data_frame.add(time)
46          data_frame.add(MAC_address_source_node)
47          data_frame.add(MAC_address_destination)
48          notify(data_frame) to Plugin_Metrics
49      }
50  }
51  else if (packet does not reach destination)
52  {
53      if (packet is not ACK)
54      {
55          //collect info frame: transmission frame fails -> transmission error for DATA
56          data_frame.add("TX FAILS")
57          data_frame.add(time)
58          data_frame.add(MAC_address_source_node)
59          notify(data_frame) to Plugin_Metrics
60          if (previous packet was UNICAST)
61          {
62              //collect info frame: reception frame fails – Data message without ACK
63              data_frame.add("RX FAIL")
64              data_frame.add(time)
65              data_frame.add(MAC_address_source_node_previous_packet)
66              data_frame.add(MAC_address_destination_node_previous_packet)
67              notify(data_frame) to Plugin_Metrics
68          }
69      }
70  }
71 }
```

The pseudo-code shows how *generate_frame_datagram* determines L2 metrics by *RadioConnection* packets.

All the code is contained in a *for cycle* (line 1) that has the same amount of iterations as the number of radio packets exchanged on the radio medium.

Each packet is analysed through an 802.15.4 analyser, implemented by the class *Analyzer_802154* (line 4).

The class determines the MAC address for the source and the destination of the radio packet by the method *getMAC*.

After, *generate_frame_datagram* checks the packet transmission.

1) If the packet is sent correctly and if it is a DATA message, the class runs the following operations:

- It sets the MAC address for the source of the packet through the method *setMACAddresses* (line 7). This method defines a *Frame154EndPoints* object, containing the MAC address returned by *Analyzer_802154*. After that, the object is saved in the class *Node* for the source mote.
- It sets the MAC address for the destination of the packet through the method *setMACAddresses* (line 8), which has the same functionalities of the previous point.
- It notifies to the *Plugin_Metrics* the transmission of a frame without errors. The notification includes the frame state (TX OK), the current time instant and the source MAC address (lines 9-13).
- It checks the packet type:
    - If packet is MULTICAST, the class notifies to the main class the correct reception of a frame for each packet destination. Every notification contains the frame state (RX OK), the current time instant, the source MAC address and the destination MAC address (lines 14-25).
    - If packet is UNICAST, the frame will have to contain two radio packets, the DATA followed from the ACK. To reach this goal the class checks if the previous packet was a DATA.

If the control is positive, *generate_frame_datagram* sees a DATA followed from another DATA. Therefore, the class notifies the reception error to *Plugin_Metrics*, sending the frame state (RX FAILS), the time instant and the MAC addresses (lines 26-37).

2) Instead, if the sent packet is an ACK message, the class discards the duplicate packets through the sequence number, and it notifies the correct reception of the frame (DATA followed from ACK) with the following arguments: frame state (RX OK), time instant and MAC addresses of the source and the destination (lines 39-50).

3) Finally, if the DATA packet is not sent correctly, the class notifies the failed transmission to the *Plugin_Metrics*. The notification contains the state (TX FAILS), the time instant and the source MAC address (lines 55-59). Moreover, *generate_frame_datagram* carries out the same control done in the lines 26-37, notifying eventual reception errors (lines 60-68).

3.7.2 Capture Phase - Node Metrics

This section illustrates the development of the method *CaptureFrameMetricNode* of the class *Capture*, used to find the metrics applied to individual nodes.

CaptureFrameMetricNode requires the following input arguments: the metrics selected by the user in *Info_Metric_Node* format and the information included in the notifications, sent by the *generate_frame_datagram*.

When the *Plugin_Metrics* receives a notification by the *generate_frame_datagram*, *CaptureFrameMetricNode* checks each selected metric with the second parameter. The aim is to find the data required from the metrics within the information obtained by *generate_frame_datagram*. If this condition is verified, the metric value is updated, creating a new object of the class *Storing*.

The basic structure of the method is reported below.

```
1  for each metric
2          if (id_metric = 1)
3                  process   Number of frames successfully sent
4          else if (id_metric =2)
5                  process   Number of frames not successfully sent
6          else if (id_metric = 3)
7                  process   Number of frames successfully received
8          else if (id_metric = 4)
9                 process   Number of frames not successfully received
```

The first metric (lines 2-3) counts the number of frames sent by the metric node without transmission errors. The metric updating occurs when the frame state reports a transmission without failures (TX OK) and when the metric node is the frame source.

The second metric (lines 4-5) represents the number of frames not sent to destination due to transmission errors. To increment the metric value, the necessary condition is to have the metric node as frame sender and to have the frame state that indicates a failed transmission (TX FAILS).

The third metric (lines 6-7) indicates the number of frames successfully received by the metric node. In this case, the metric value increases of 1 if the metric node is a destination of the frame and if the frame state represents a reception without errors (RX OK).

Finally, the fourth metric (lines 8-9) counts the number of frames not received by the metric node due to reception errors. In this case, the metric value is updated if the metric node receives the frame and if the frame state reports a failed reception (RX FAILS).

3.7.3 Capture Phase - Couple Metrics

This section presents the method *CaptureFrameMetricCouple* of the class *Capture*. The method has the function to collect L2 metrics for couples of nodes.

CaptureFrameMetricCouple has the same parameters of the previous method. The only difference regards the selected metrics, which present the *Info_Metric_Couple* format.

The method basic architecture is the following:

```
1   for each metric
2       if (id_metric = 1)
3           process   Number of frames successfully received
4       else if (id_metric =2)
5           process   Number of frames not successfully received
```

Number of frames successfully received (lines 2-3) counts the amount of frames sent correctly by the source node and received without errors by the destination.

The metric value changes if the source and the destination of the frame, with state "RX OK", coincide with the source and the destination of the metric couple.

The second metric (lines 4-5) shows the number of frames sent correctly by the source node and not received by the destination due to reception errors.

The metric undergoes an update when the notified frame presents as state "RX FAILS" and it has the source and the destination of the couple metric such as the source and the destination of the frame.

3.8 L3 Components

The paragraph shows the metrics collection mode for the networking layer of the IP Stack.

This layer has the function to route the application traffic toward the root through the routing protocol RPL.

As mentioned in the previous paragraphs, COOJA does not support the implementation of the IPv6 packets and therefore the messages have been developed analysing the radio packets exchanged on the radio medium.

To create a datagram, the frame packets generated in the previous section have to be composed. They are made up of the *RadioConnection* objects.
The datagrams are divided in four categories, i.e. DIO, DIS, DAO and UDP.
DIO, DIS and DAO represent the routing traffic. DIO and DIS are multicast packets consisting of a single radio connection; they do not require the ACK packet as control mechanism.
DAO messages are unicast packets formed by two radio connections: a DATA message to send routing information to the parent; an ACK message as control system for the packet transmission. This RPL packet type does not make hops, since the TestRPL application uses the routing protocol in storing mode (see **3.1**).
UDP messages are the packets sent by application clients toward the application server. These datagrams do various hops to reach the destination and therefore they are composed of different frames, one for each data-link.
For this IP Stack layer, only two metrics are captured: datagrams queue and delay end-to-end. The datagrams queue will be obtained during the datagrams generation phase, while the delay end-to-end will be calculated using the same logic of the L1 metrics and L2 metrics.

During the reading of the following paragraph, java classes and java methods will be appointed again. Refer to section **3.6** and to the section **D** of the appendix.
Moreover, given the complexity of the relation between classes, the diagram of classes for the L3 [11] components is shown below.

[11] L3 – Networking Layer

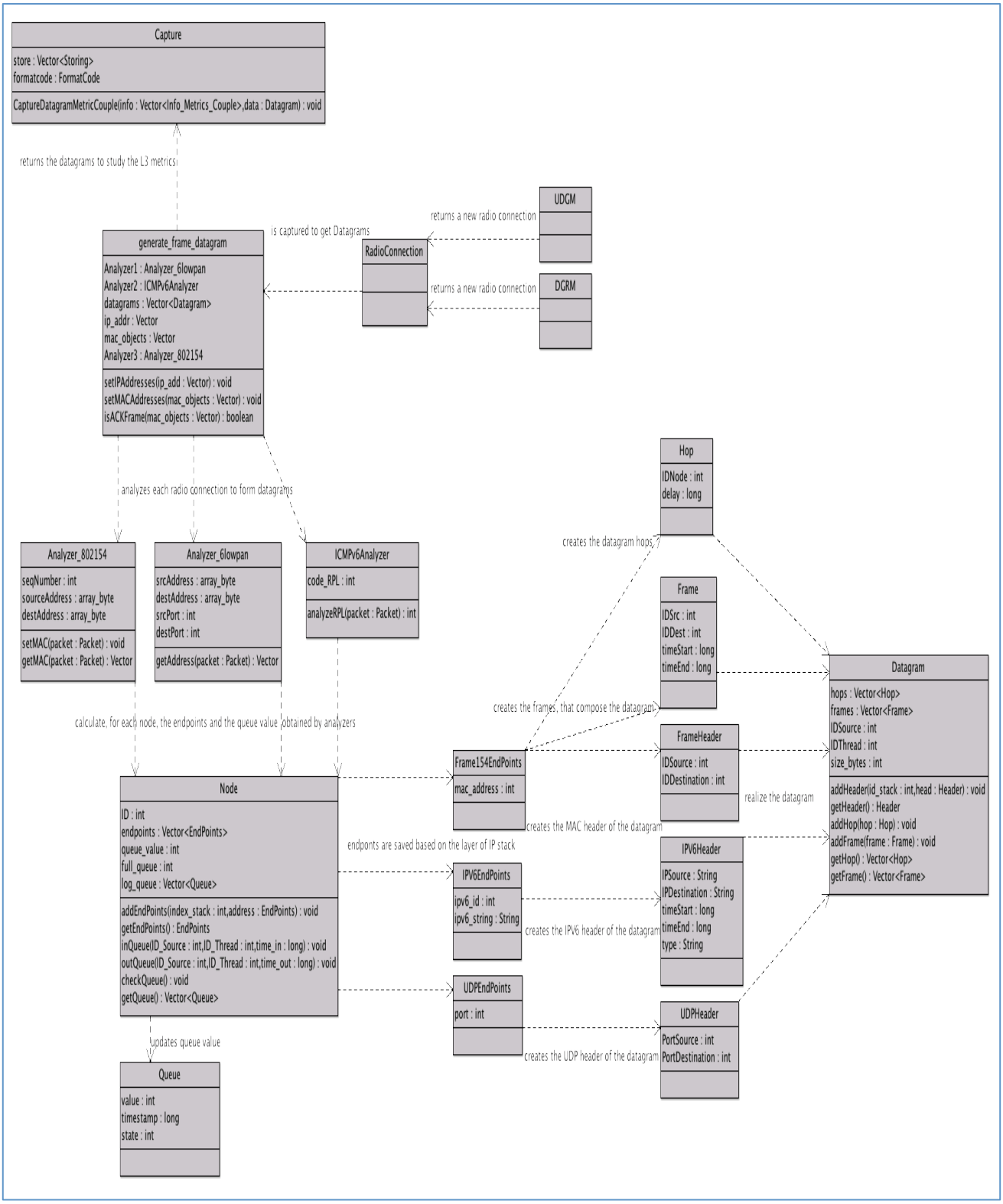

Figure 6: Diagram of classes – L3 components

3.8.1 Generation Datagrams

As for the data-link layer, the class *generate_frame_datagram* will be used to create the IPv6 packets. The development of the datagrams is based on the analysis of each radio packet created by the radio medium, using the analysers *Analyzer_802154*, *Analyzer_6LoWPAN* and *ICMPv6Analyzer*.

During the building phase, moreover *generate_frame_datagram* manages the evolution of the queue for each new analysed radio packet.

Each time a datagram is realized, the class *generate_frame_datagram* acts as *Observable* object to notify it to *Plugin_Metrics* (*Observer* object).

The basic structure of the *Observable* object is shown below.

```
1  For each radio connection
2  {
3     conn = radiomedium.getLastConnection()
4     frame = analyser_802154(conn)
5     if (frame is transmitted correctly and it is not ACK)
6     {
7        set MAC_address_source_node(data_frame)
8        set MAC_address_destination_node(data_frame)
9        data_datagram = Analyzer_6LoWPAN(conn)
10       set IP_address_source_node(data_datagram)
11       set IP_address_destination_node(data_datagram)
12       set UDP_port_source_node(data_datagram)
13       set UDP_port_destination_node(data_datagram)
14        if (ICMPv6Analyzer.analyzeRPL(conn)==0 or ICMPv6Analyzer.analyzeRPL(conn)==1)
//RPL DIS or RPL DIO
15        {
16             for each destination
17            {
18                //create FrameHeader
19                FrameHeader.setSource(MAC_address_source_node)
20                FrameHeader.setDestination(MAC_address_destination_node)
21                // create IPv6Header
22                IPv6Header.setIPSource(IP_address_source_node)
23                IPv6Header.setIPDestination(null)
24                IPv6Header.setTimeStart()
25                IPv6Header.setTimeEnd()
26                IPv6Header.setType("DIS")
27                //create Datagram
28                datagram.add(FrameHeader)
29                datagram.add(IPv6Header)
30                notify(datagram)
31             }
32          }
33          else if(ICMPv6Analyzer.analyzeRPL(conn)==2)
34          {
35               //DAO datagram -> attends the ACK
36           }
37           else
38           {
39               //UDP datagram -> attends the ACK
40            }
41    }
42    else if (packet is transmitted correctly and it is ACK)
43    {
44          if (ICMPv6Analyzer.analyzeRPL(conn)==3 and previous packet was a DAO) // DAO ACK
45          {
46                //create FrameHeader
47                FrameHeader.setSource(MAC_address_source_node)
48                FrameHeader.setDestination(MAC_address_destination_node)
```

```
49                  // create IPv6Header
50                  IPv6Header.setIPSource(IP_address_source_node)
51                  IPv6Header.setIPDestination(IP_address_destination_node)
52                  IPv6Header.setTimeStart()
53                  IPv6Header.setTimeEnd()
54                  IPv6Header.setType("DAO")
55                  //create Datagram
56                  datagram.add(FrameHeader)
57                  datagram.add(IPv6Header)
58                  notify(datagram)
59          }
60          else if (ACK is of type UDP and previous packet was UDP datagram)
61          {
62                   if (data_frame sequence number is new)
63                   {
64                        // create hop
64                        hop.setRelayNode(MAC_address_destination_node)
65                        hop.setDelay();
66                        UDP_Datagram.add(hop);
67                        // create frame
68                        frame.setSource(MAC_address_source_node)
69                        frame.setDestination(MAC_address_destination_node)
70                        frame.setTimeStart()
71                        frame.setTimeEnd()
72                        UDP_Datagram.add(frame)
73                        update_queue
74                        check_queue
75                        if (UDP_Datagram has reached the root)
76                        {
77                              // create FrameHeader
78                              FrameHeader.setSource(MAC_address_source_node)
79                              FrameHeader.setDestination(MAC_address_destination_node)
80                              // create IPv6Header
81                              IPv6Header.setIPSource(IP_address_source_node)
82                              IPv6Header.setIPDestination(IP_address_destination_node)
83                              IPv6Header.setTimeStart()
84                              IPv6Header.setTimeEnd()
85                              IPv6Header.setType("UDP")
86                              //create UDPHeader
87                              UDPHeader.setPortSource(Port_source)
88                              UDPHeader.setPortDestination(Port_destination)
89                              //create Datagram
90                              UDP_Datagram.add(FrameHeader)
91                              UDP_Datagram.add(IPv6Header)
92                              UDP_Datagram.add(UDPHeader)
93                              notify(datagram)
94                        }
95                   }
96          }
97     }
98  }
```

The pseudo-code shows how *generate_frame_datagram* collects the information by *RadioConnection* packets to determine L3 metrics.

All the code is contained in a *for* cycle (line 1) that has an amount of iterations equal to the number of radio packets exchanged on radio medium.

Each packet is analysed through an 802.15.4 analyser, implemented by the class *Analyzer_802154* (line 4).

The class gets the MAC address for the source and the destination of the radio packet in a *Vector* instance through the method *getMAC*.

After that, *generate_frame_datagram* checks the packet transmission.

1) If the datagram is sent correctly and if it is a DATA message the class runs the following operations:

- It sets the MAC address for the source and the destination of the packet through the method *setMACAddresses* (lines 7-8). In other words, the method instantiates two *Frame154EndPoints* objects, one for the source and one for the destination through the MAC address. Then the objects are saved in the class *Node*.
- It runs the class *Analyzer_6LoWPAN* that analyses the radio packets through the method *getAddress*, which returns the IP address and the UDP port for the source and the destination of the packet in a *Vector* object (line 9).
- It sets the IP addresses and UDP ports for the source and the destination of the packet through the method *setIPAddresses*. The method builds two *IPV6EndPoints* objects and two *UDPEndPoints* objects. The first *IPV6EndPoints* object associates the IP address to the source node, while the second *IPV6EndPoints* object associates the IP address to the destination node. Instead, the *UDPEndPoints* objects are set through the UDP ports.

  Once the endpoints are instantiated, they are saved in the class *Node* (lines 10 - 13).
- It checks the packet type:
  - If packet is a DIO datagram or a DIS datagram, a *Datagram* object for each destination will be built.

The datagram is composed of two headers: an 802.15.4 header and an IPv6 header.

The 802.15.4 header is implemented by the class *FrameHeader*, which defines within the datagram the MAC address for the source and the destination of the RPL packet through the data contained in the *Frame154EndPoints* objects.

The IPv6 header is implemented by the class *IPV6Header*, which establishes within the datagram the IP addresses for the source and the destination of the RPL message through the information included in the *IPV6EndPoints* objects.

Once the datagram is built, it is notified to the *Observer* (lines 14 -32).

- If packet is a DAO datagram or a UDP datagram, *generate_frame_datagram* does not carry out any operation; in fact it waits for the ACK message to confirm the correct datagram transmission (lines 33-41).

2) Instead, if the sent packet is an ACK message, the class checks the type of the message:

- If the packet is an ACK message and the previous packet was a DAO message, a *Datagram* object is generated. It includes the *FrameHeader* and the *IPV6Header* as seen previously for the datagrams DIO and DIS (lines 44-59).
- If the packet is an ACK message and the previous packet was a UDP datagram, *generate_frame_datagrams* uses the sequence number to discard the duplicate packets. If the packet is not deleted, two objects, *Hop* and *Frame*, will be created.

  Hop represents the *relay* node that forwards the UDP datagram toward a new data-link. The *relay* node is identified by the destination node MAC address of the UDP datagram and this value is saved in *Hop*. Also *Hop* considers the transmission delay, necessary to the datagram in order to cross the *relay* node. The transmission delay is added in *Hop*.

Frame represents the UDP datagram in a precise 802.15.4 link. If a UDP datagram crosses *N* hops to reach the destination, then the datagram is divided in *N+1* frames.
In this case *generate_frame_datagrams* saves in *Frame* the source node MAC address of the previous packet; the destination node MAC address of the previous packet; the time instant in which the previous packet is sent; the time instant in which the previous packet is received.
Once *Hop* and *Frame* are instantiated, they are inserted in the *Datagram* object through the methods *addHop* and *addFrame* (lines 60-72).

At this point, the class updates the queue value for the *relay* nodes (line 73). If the previous packet has come in the queue of a *relay* node, the *relay* node, with the class *Node*, will start the method *inQueue* that increments the queue value. *inQueue* needs the following input arguments: the incoming UDP datagram in the queue and the related time instant. On the contrary, if the previous packet goes out from the queue of a *relay* node, the *relay* node, with the class *Node*, will set up the method *outQueue* that decrements the queue value. *outQueue* requires the outgoing UDP datagram from the queue and the related time instant.

If the previous packet has the destination node MAC address equal to 1, then the UDP datagram reaches the final destination. In this case the *Datagram* object with tree headers will be generated: *FrameHeader*, *IPV6Header* and *UDPHeader*.
FrameHeader and *IPV6Header* are set with the same way seen for the DIO datagrams. Instead, *UDPHeader* defines the UDP ports used by the source and by the destination for the data transport.
When the headers are complete, they are added to the *Datagram* instance through the method *addHeader*.
Finally, the datagram is notified to *Plugin_Metrics* (lines 75-99).

Each iteration of the *for cycle* runs a control to the queue for the *relay* nodes through the method *checkQueue* of the class *Node* (line 74).
The method has the function to remove the datagrams located in the queue from a time amount upper a predefined timeout.

3.8.2 Capture Metric - Delay End-To-End

The class *generate_frame_datagram* notifies every datagram to the *Plugin_Metrics*.
When the main class receives a *Datagram* object, it starts the capture phase to obtain the metric delay end-to-end.
The delay end-to-end is the time taken for a datagram to be transmitted across the network from the source to the destination.
The metric is calculated through the method *CaptureDatagramMetricCouple* of the class *Capture*. The method needs the following parameters: the couple metric selected by the user and the *Datagram* object.
If the Datagram instance has like source and destination the source and the destination chosen by the user, the method sets up the metric computation, which changes according to the datagram typology.
If the datagram is a RPL packet, the delay end-to-end is obtained calculating the difference between the time instant in which the datagram is received and the time instant in which the datagram is sent. The time instants have been returned through the class *IPV6Header*.
On the contrary, if the datagram is a UDP packet, the metric is calculated as sum between the transmission delay and the queuing delay.
The transmission delay is obtained summing the delays of each hop, done by the datagram (Each hop delay is get by the class *Hop*).
The queuing delay is achieved calculating the duration of time in which the datagram remains in queue.
Once the metric value is calculated, it is saved in *Storing*.

4. Plugin Testing

This chapter presents the testing phase of the metrics collection tool, developed for COOJA.

The testing phase has the aim to verify the correct functioning of the plugin through the analysis of the metrics charts.

Practically, the chapter shows how to realize a simulation for COOJA, creating the network of nodes to study with the plugin; it analyses the metrics for the physical, data-link and networking layers through the graphics obtained by the metrics collection tool; it illustrates the way whereby the metrics collection tool calculates the average delay end-to-end, the packet loss rate and the throughput, demonstrating, through their charts, the utility of the plugin.

4.1 Realization COOJA Simulation

In order to test a network application, the user has to start COOJA by the terminal with the ant [12] command *ant run_bigmem* within the directory *cooja* [13].

When the boot phase finishes, the user can create the sensor network, selecting the desired radio medium. This project work uses DGRM as radio medium, since DGRM permits to realize different network topologies with many links between the motes.

Before the network is created, the user has to select the binary image of the application in order to install it in the motes. In the simulations carried out for this project, the nodes run Test_RPL as program (see **3.1**).

In order to adapt the application to the COOJA simulation, the Test_RPL *Makefile* is set, modifying the following directives:

- *CFLAGS+=-DCC2420_DEF_CHANNEL* = 25 -> the CC2420 radios use the radio channel 25;
- *CFLAGS +=-DBLIP_L2_RETRIES* = 10 *-DBLIP_L2_DELAY* = 103 -> a client carries out 10 retries to send a single packet. Between a message and the other, the client waits 103 ms;

[12] Java library - http://ant.apache.org/

[13] The *cooja* directory is localized in the path *contiki/tools/cooja*

- *CFLAGS+=-DPACKET_INTERVAL* = 128UL -> the packets transmission rate is set to 8 packets/s;
- *CFLAGS+=-DPACKET_NUMBER* =100 -> each client generates 100 packets, addressed to the root.

Once the TinyOS is installed in the sensors, the user deploys them along the physical channel through the plugin *Simulation Visualizer* (see **2.1.5**) obtaining the following chart:

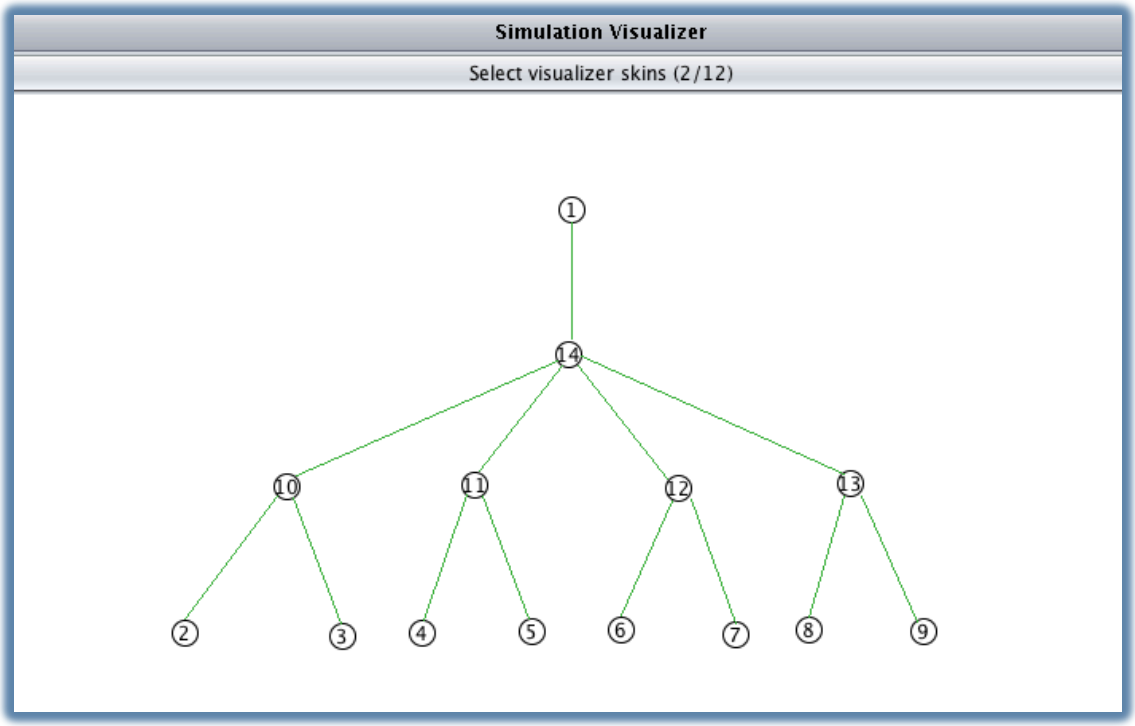

Figure 7: Network Topology

The network presents 8 sensor nodes (clients) that sent packets toward the root.
In fact, all the application traffic is destined to the node 1, since this mote acts as a router to forward the data to Internet.
Each couple of clients has a common parent that delivers the packets to the bottleneck mote; then, this node consigns the data to the sink node.
This network topology has been chosen to analyse the reception losses for the *relay* node 14.

At this point the user can choose the desired metrics to test the Test_RPL application through the metrics collection tool.

To start the plugin, the user has to click on the item *6LoWPAN metrics* in the drop-down menu of *Plugins* and consequently the tool appears in the COOJA window.

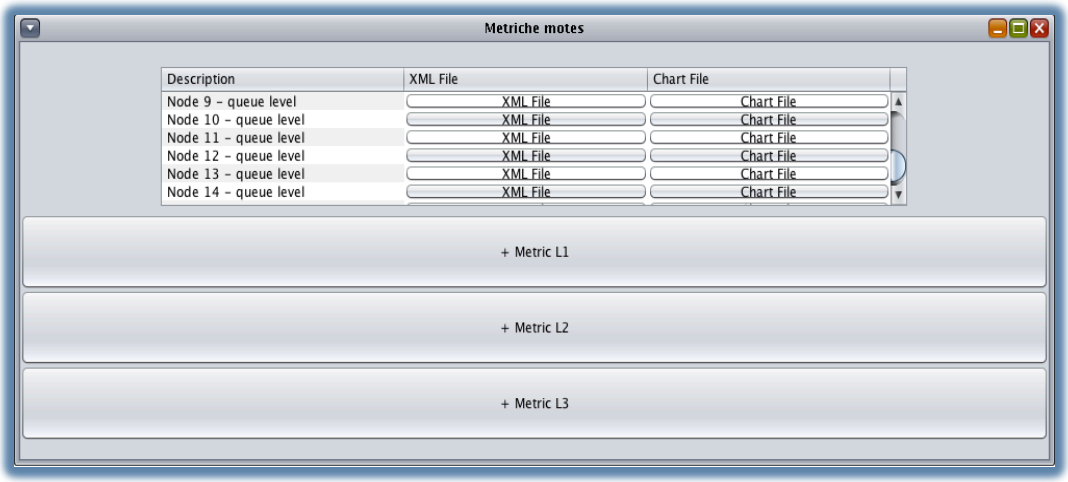

Figure 8: Graphical Interface of the Metrics Collection Tool

The plugin consists of three buttons and a table, containing the metrics selected by the user. In order to insert a new metric, the user chooses one of the three buttons, according to the IP-Stack layer. Once the user clicks on a button, the following frame appears in the window.

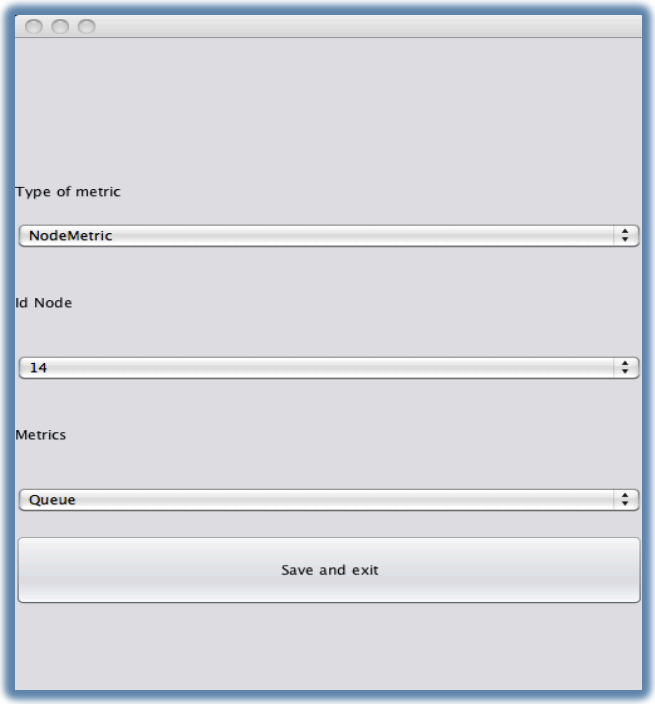

Figure 9: Graphical interface to select a new Metric

Concluded the metrics selection phase, the user can start the simulation, in order to analyse the metrics.

4.2 Analysis Metrics – Physical Layer

This section presents the metrics graphics for the first IP stack layer, obtained analysing each radio packet exchanged on the DGRM radio medium, as shown in the previous paragraph.

The charts have been calculated for the following metrics:

- Number of connections successfully sent – Node 1
- Number of connections successfully sent – Node 10
- Number of connections successfully sent – Node 11
- Number of connections successfully sent – Node 12
- Number of connections successfully sent – Node 13
- Number of connections successfully received – Node 14
- Number of connections not successfully received (collision) – Node 14

To analyse these metrics, the testing phase considers only the data-links between the parent nodes (plus root) and the node with ID 14, with the aim to show the radio collisions for the transceiver incorporated in the bottleneck node.

Firstly, the testing phase studies the first five metrics. The obtained charts are the following:

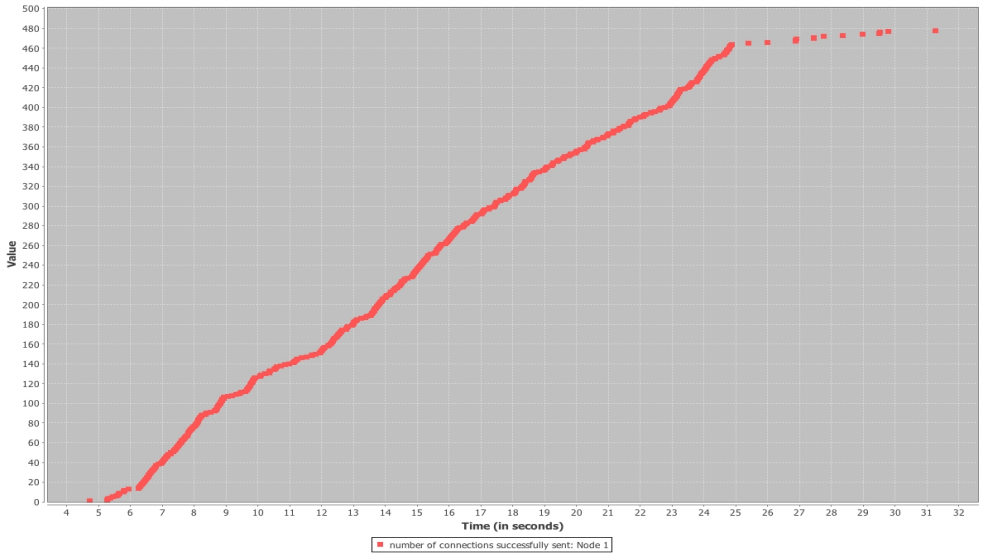

Figure 10: Number of radio connections successfully sent by the node 1

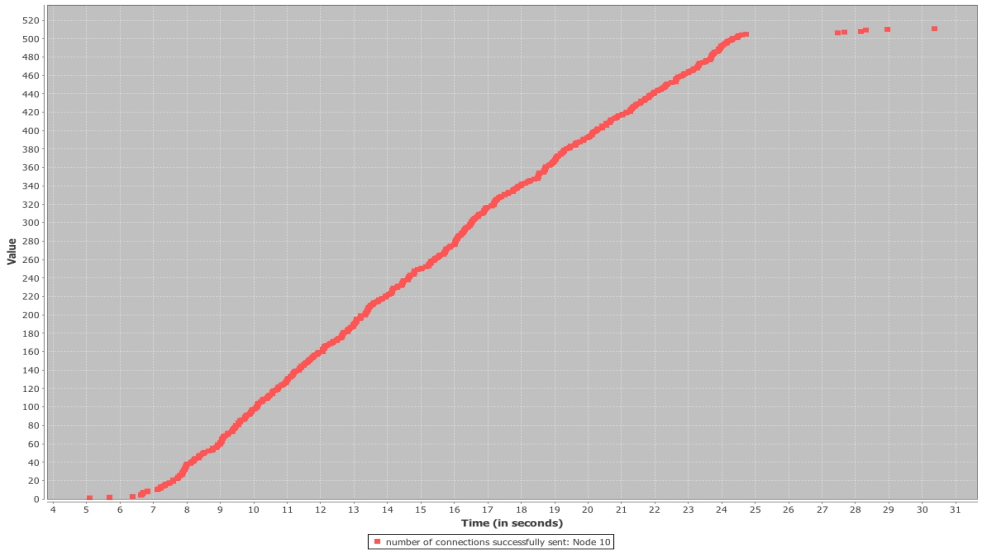

Figure 11: Number of radio connections successfully sent by the node 10

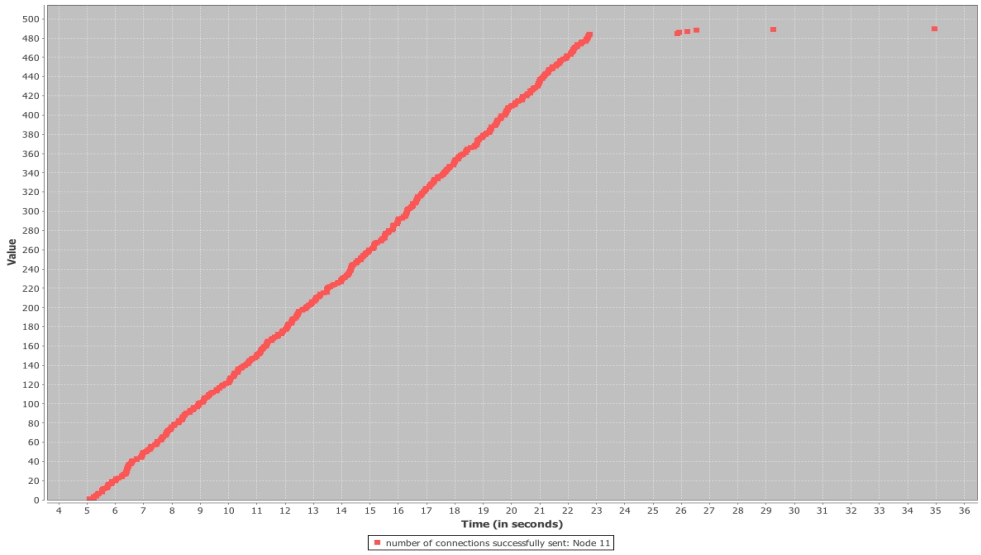

Figure 12: Number of radio connections successfully sent by the node 11

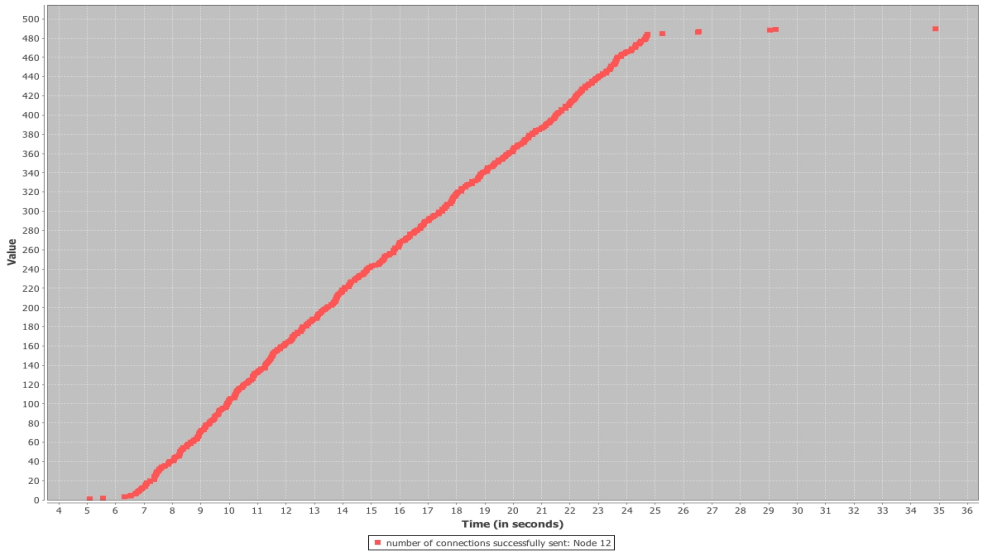

Figure 13: Number of connections successfully sent by the node 12

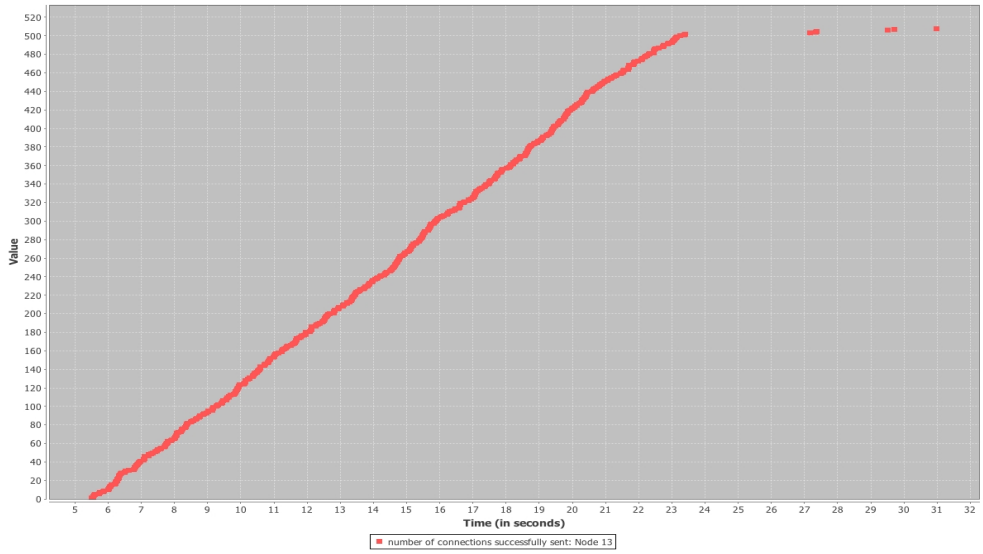

Figure 14: Number of radio connections successfully sent by the node 13

These charts define the number of radio connections sent by the nodes *1*, *10*, *11*, *12* and *13*. A radio connection has, like destination, all the motes reachable through the edges connected to the source node. Moreover, all the radio packets are equal, without distinction between DATA and ACK messages.

In details, the simulation presents the following transmissions:

- the node *1* transmits 478 radio connections to the mote *14*;
- the node *10* transmits 511packets to the nodes *2*, *3* and *14*;
- the node *11* sends 490 packets to the motes *4*,*5* and *14*;
- the node *12* sends 490 packets to the nodes *6*, *7* and *14*;
- the node *13* delivers 508 radio connections to the motes *8*, *9* and *14*.

Summing the radio connections sent by each node, in total 2477 radio packets are sent on the radio channel.

Each transmission has at least the mote *14* as destination node of the radio connection; therefore, this node is also called bottleneck, since it *hears* each radio packet sent on the radio medium, becoming thus interfered.

In order to study the interference level of the bottleneck node, the charts of the fifth and the sixth metric are reported below.

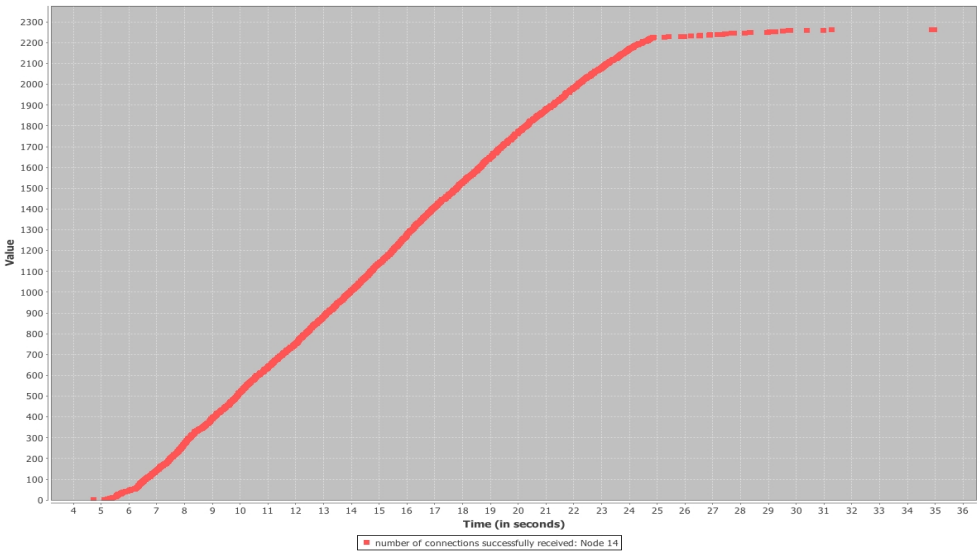

Figure 15: Number of radio connections successfully received by the node 14.

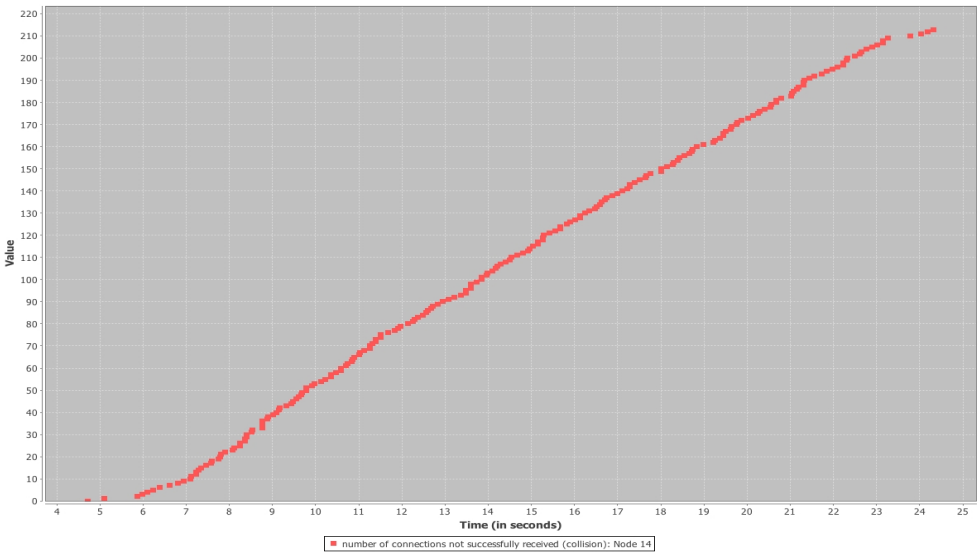

Figure 16: Number of radio connections not successfully received by the node 14 due to radio collisions.

The bottleneck receives correctly 2264 radio connections by the nodes, linked to its same edges. Actually, the node *14* would receive 2477 packets (2477 coincides with the amount of connections sent), but 213 of those fail during the reception phase due to collision errors, which are originated mainly by radio interferences.

4.3 Analysis Metrics – Data-Link Layer

This section analyses the frame packets exchanged among the same nodes of the previous paragraph, with the aim to detect eventual losses in the 802.15.4 links (each link coincides to a edge of the topology network).

To reach this objective ten metrics are test:

- Number of frames successfully sent by the node 1 and received without errors by the node 14;
- Number of frames successfully sent by the node 1 and not received due to collisions by the node 14;
- Number of frames successfully sent by the node 10 and received without errors by the node 14;
- Number of frames successfully sent by the node 10 and not received due to collisions by the node 14;
- Number of frames successfully sent by the node 11 and received without errors by the node 14;
- Number of frames successfully sent by the node 11 and not received due to collisions by the node 14;
- Number of frames successfully sent by the node 12 and received without errors by the node 14;
- Number of frames successfully sent by the node 12 and not received due to collisions by the node 14;
- Number of frames successfully sent by the node 13 and received without errors by the node 14;
- Number of frames successfully sent by the node 13 and not received due to collisions by the node 14.

Firstly, the link between the node *1* and the node *14* is analysed, considering the number of frames sent and received. The related graphs are reported below.

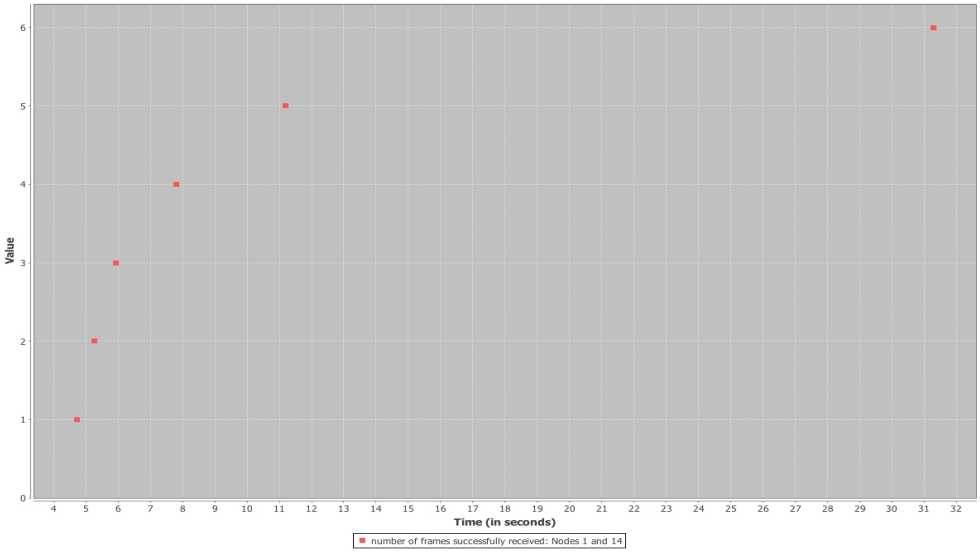

Figure 17: Number of frames sent by the node 1 to the bottleneck without errors

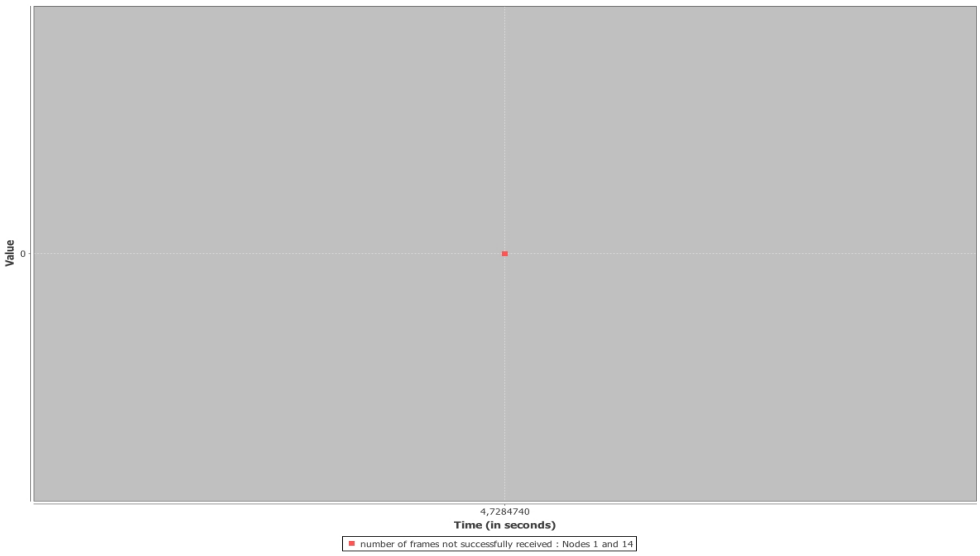

Figure 18: Number of frames sent by the node 1 to the bottleneck with reception errors

The root sends only 6 frames, during the length of the experiment. All the messages arrive at their destination without reception errors. Comparing the number of frames sent (see Figure 17) to the number of radio packets transmitted (see Figure 10), a considerable gap emerges, given the high number of connections sent on the radio medium by the sink node. In fact the root has to transmit to the *relay* node *14* a radio packet (ACK message) for each message sent by the application clients.

Secondly, the edge between the node *10* and the bottleneck is analysed with the following graphics.

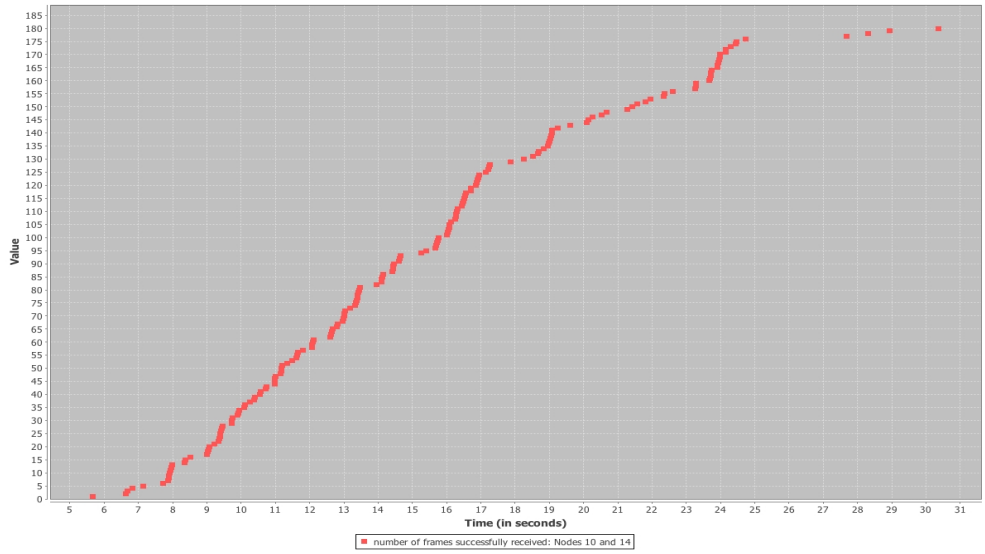

Figure 19: Number of frames sent by the node 10 to the bottleneck without errors

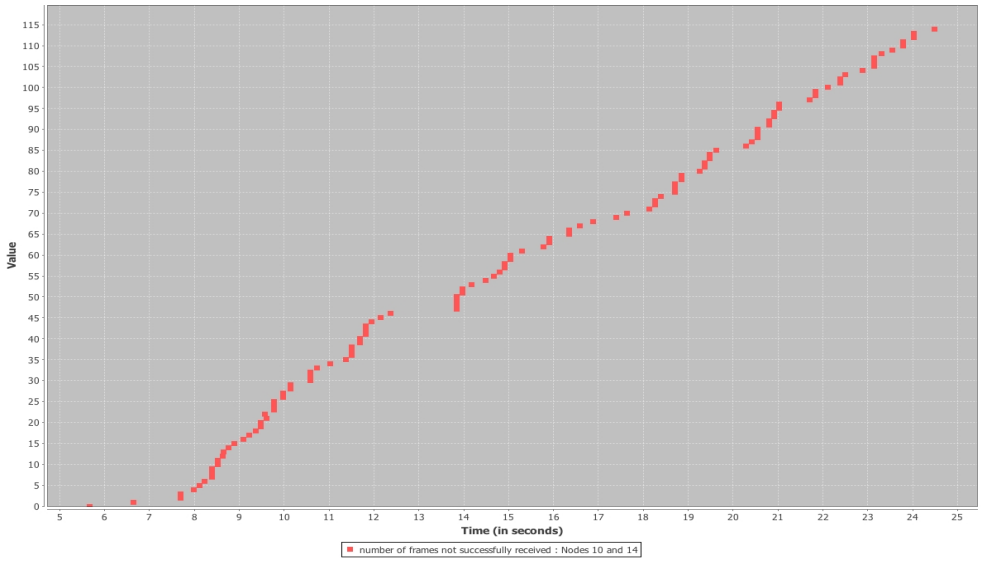

Figure 20: Number of frames sent by the node 10 to the bottleneck node with collision errors.

In this case the *relay* node *10* forwards 294 frames to the bottleneck (180 received correctly and 114 lost for collisions), sent by the client nodes *2* and *3*.

The figure 11 has showed 511 radio connections transmitted by the node *10*; this discrepancy (511 - 294) is due to the ACK messages, sent by the *relay* node to its children nodes.

Given the two L2 metrics, the link 10 – 14 presents a frame loss rate of the 39%.

Thirdly, the link 11-14 is studied through the charts, reported below.

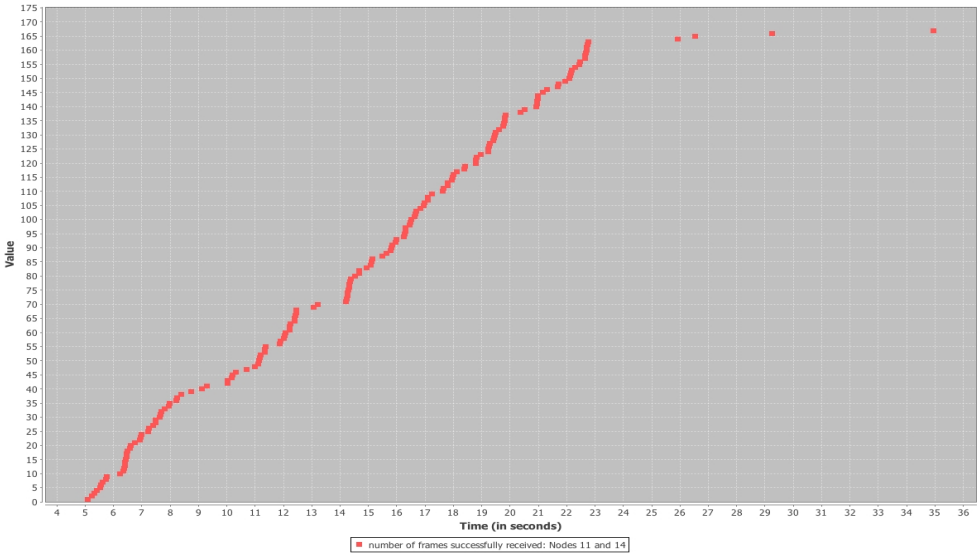

Figure 21: Number of frames sent by the node 11 to the bottleneck without errors

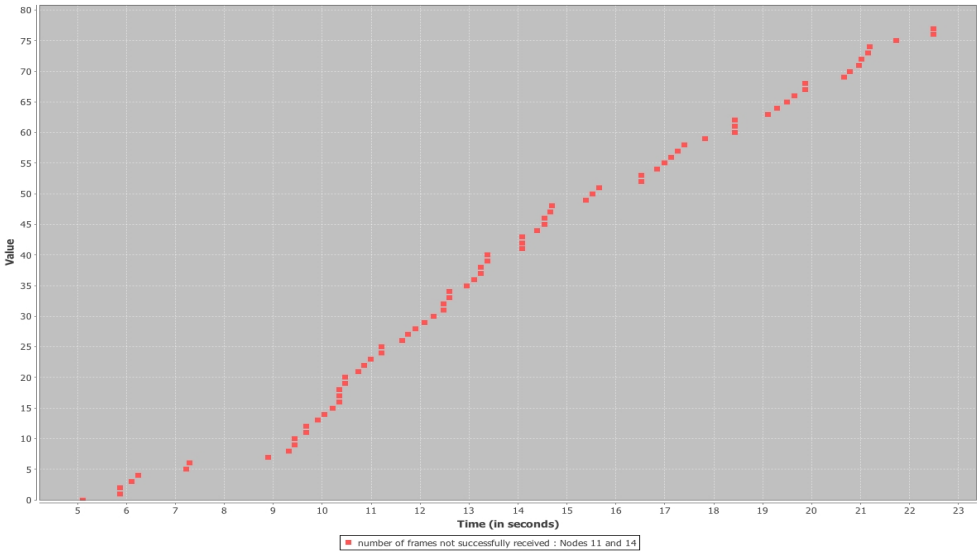

Figure 22: Number of frames sent by the node 11 to the bottleneck with collisions

The node *11* sends 244 frames to the node *14*, but 77 of those (see figure 22) fail due to the radio collisions. Therefore, the 32% of the frames is discarded.

The difference between the number of frames sent and the number of packets transmitted on the radio medium (see figure 12) is due to the same causes, discussed for the node *10* and this observation will be omitted for the next L2 analyses.

Fourthly, the subsequent graphs provide more information for the data-link that connects the *relay* node *12* with the node *14*.

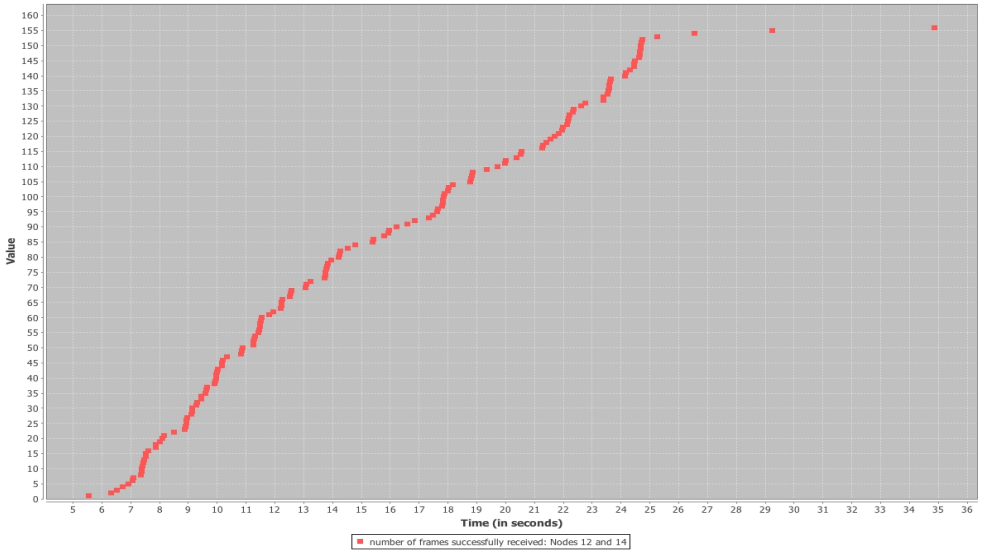

Figure 23: Number of frames sent by the node 12 to the bottleneck without errors

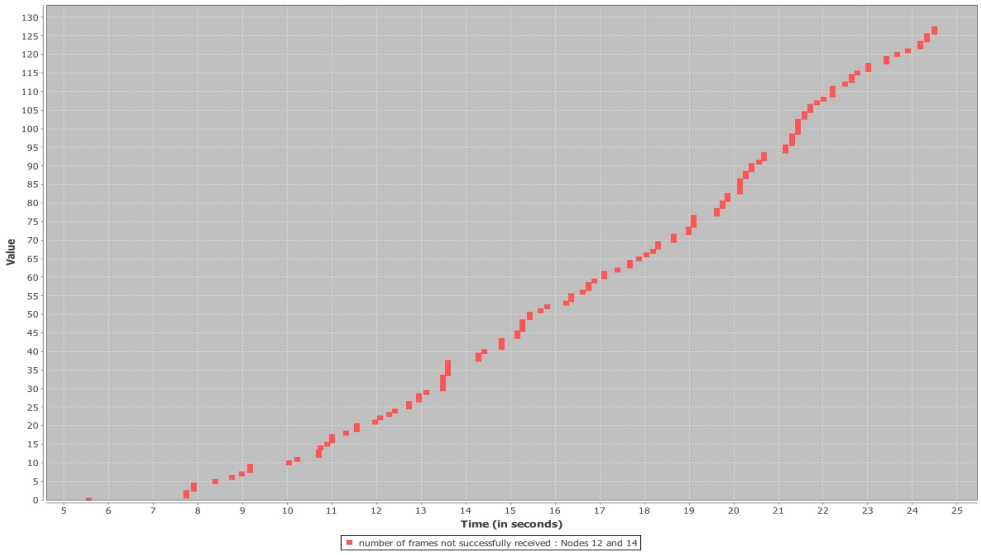

Figure 24: Number of frames sent by the node 12 to the bottleneck with collisions.

The *relay* node with ID *12* forwards 283 frames to the *router* node. 180 arrive correctly at the destination, while the remaining frames are lost in the link due to radio collisions. Thus, in this case, the frame loss rate reaches the 45%.

Finally, the testing phase focuses on the edge with the node *13* and the node *14* as endpoints.

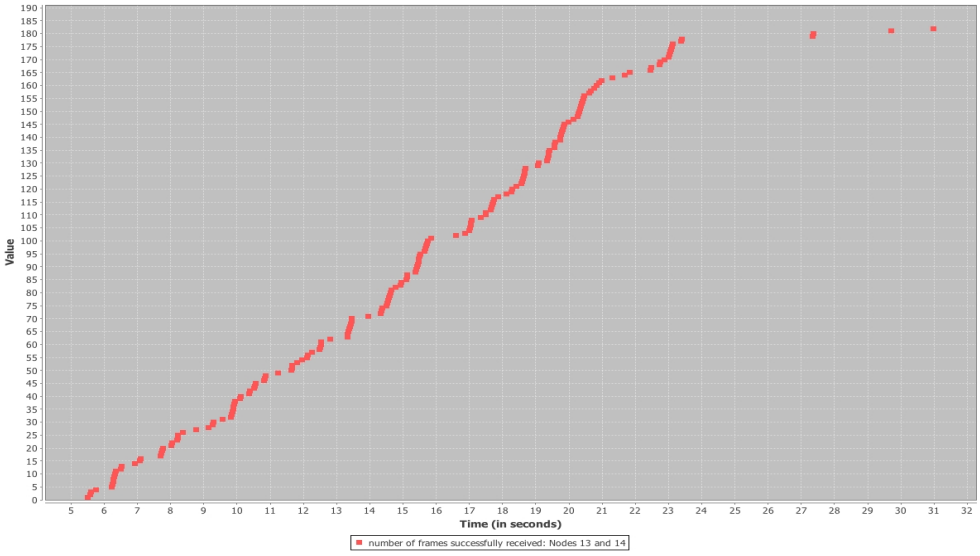

Figure 25: Number of frames sent by the node 13 to the bottleneck without errors

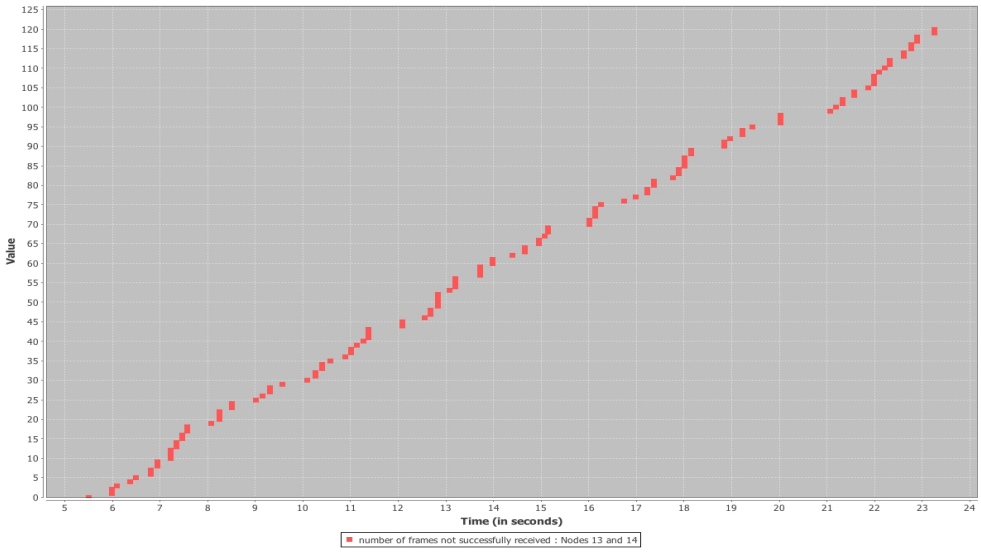

Figure 26: Number of frames sent by the node 13 to the bottleneck with collision errors

The *relay* node sends 302 frames to the bottleneck node, which are destined for the root. 120 frames fail during the reception phase of the node *14*; instead, most L2 packets reach the destination without errors. So the percentage of lost frames is equal to the 40%.

In order to verify the correctness of the frame lost rates calculated in this section by the metrics collection tool, the testing phase illustrates other two metric graphics:

- Number of frames successfully received – Node 14;
- Number of frames not successfully received – Node 14.

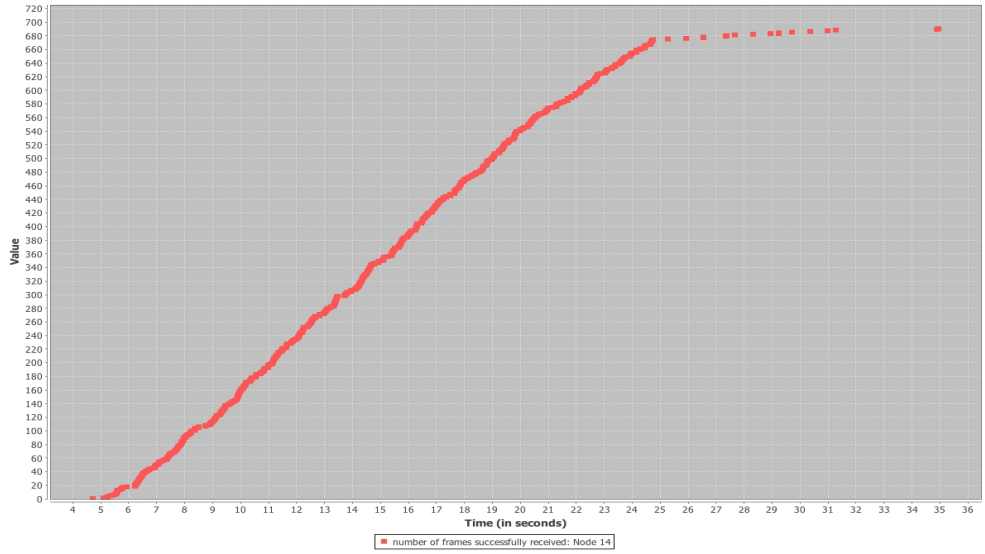

Figure 27: Number of frames received by the bottleneck without errors

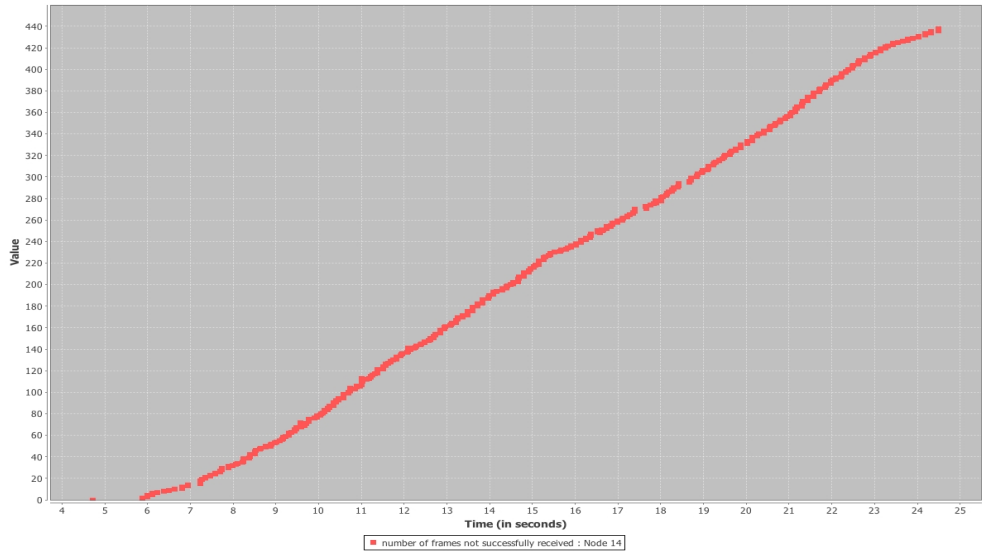

Figure 28: Number of frames not received by the bottleneck

The figure 27 shows that the frames, received correctly by the node *14*, are 691, while 438 are the L2 packets, lost during the reception phase.

If you sum the number of the frames sent by the *relay* nodes and received without errors, that is 6 (1 -> 14) + 180 (10 -> 14) + 167 (11 -> 14) + 156 (12 -> 14) + 182 (13 -> 14), the obtained result is exactly 691.

Instead, if you sum the number of the frames sent by the *relay* nodes and not received correctly by the node *14*, that is 0 (1 -> 14) + 114 (10 -> 14) + 77 (11 -> 14) + 127 (12 -> 14) + 120 (13 -> 14), the obtained result is exactly 438.

Therefore, the frames loss average rate is equal to 39%, highlighting the high loss probability due to the node *14*, that is the last *router* for each packet sent by the application clients.

4.4 Analysis Metrics – Networking Layer

As discussed in the paragraph **3.8**, the metrics for the third IP Stack layer are two: datagrams queue and delay end-to-end.

The datagrams queue is a Node Metric and it applied to the *relay* nodes in order to analyse eventual congestion states.

The charts below, which represent the queue for the nodes 13 and 14, verify if the graphic trend respects the maximum threshold of datagrams supported in the queue of the node. This value is set in the TinyOS application, running on the node. For the simulations analysed in this paragraph, the maximum number of packets that the queue of a node can contain is set to 12.

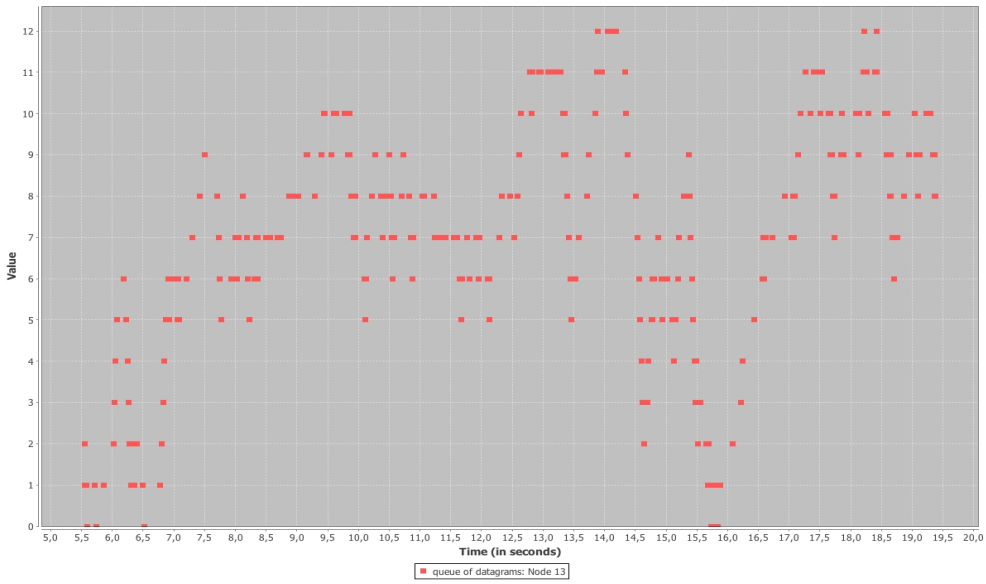

Figure 29: Queue Evolution for the relay node 13

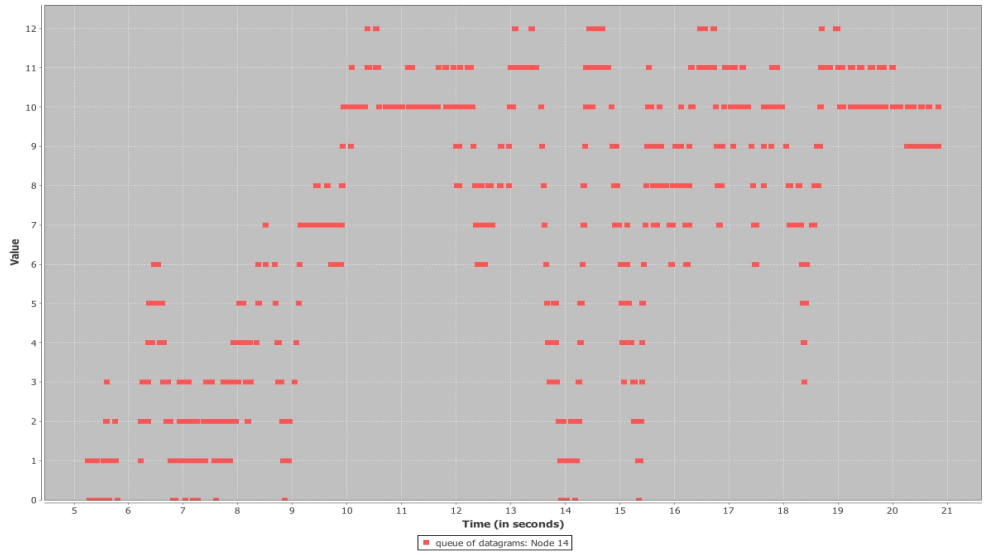

Figure 30: Queue Evolution for the relay node 14

The node *13* has the function to route the application traffic, generated by the nodes *8* and *9*, toward the bottleneck node *14*. Its queue, represented in the figure 29, reaches the threshold of the 12 datagrams for 6 times, while the queue average value is about of 6,84 packets.

Instead, the figure 30 illustrates the queue trend for the node *14*. In this case, the queue attains the maximum value of 39 times with an average value of 7,5 packets.
These results are justified by the role of the node within the routing tree; in fact the node *14* has the function to route all the traffic created by the application clients to the root. Therefore, the maximum threshold of the queue is reached many times, since the node *14* will process more datagrams than the node *13*.

The delay end-to-end represents the necessary time to send a datagram from a client node to the application server.
In order to calculate this metric, the testing phase regards the datagrams sent by the nodes *4* toward the sink node. The chart is the following:

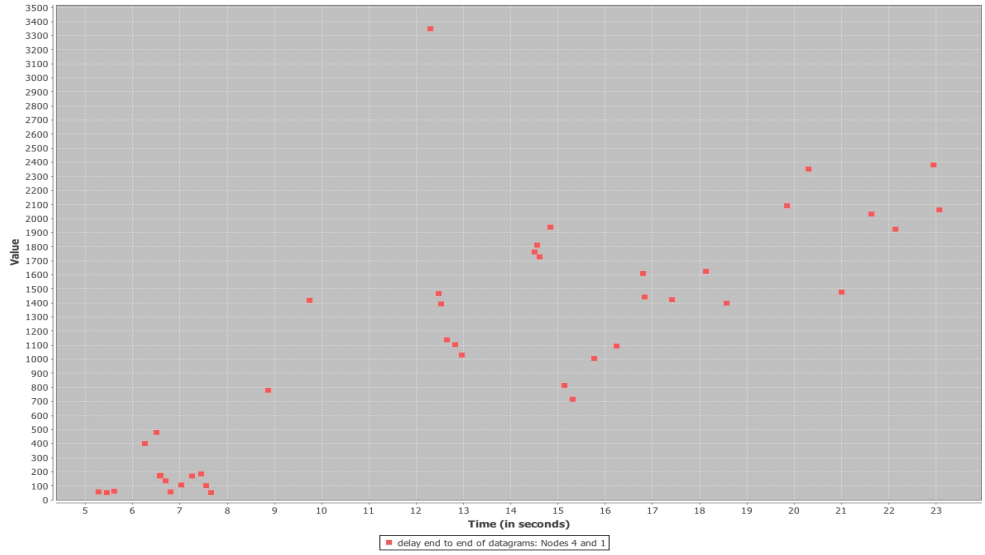

Figure 31: Delay end-to-end – Source 4 – Destination 1

The delay end-to-end considers the sum between the transmission delay and the queuing delay. By increasing the time, the application traffic grows and therefore also the time taken for a packet to be transmitted across a network grows too. In the figure 31, the metric value is expressed in ms.

4.5 Statistics

The metrics, analysed in the previous paragraphs, allow you to study the data communication from the viewpoint of the nodes.

The purpose of this section is to analyse overall the 6LoWPAN network, with the aim to measure the speed, the reliability and the capacity of the 802.15.4 links.

To reach this goal, the following statistics are calculated: average delay end-to-end, packet loss rate and throughput in transmission and in reception.

1. The average delay end-to-end is obtained by the single delays end-to-end, calculated for each datagram sent by a client.

Therefore, the metrics collection tool processes the individual delays end-to-end, calculating their average.

2. The packet loss rate (PLR) indicates the percentage of data packets, sent by the client nodes that do not reach the root of the 6LoWPAN network.

The packet loss can be caused by the following factors:

- the queue of the *relay* nodes is congested;

- radio collisions caused mainly by the radio interference;
- random errors on the radio channel.

The statistic is calculated by the metric collection tool through the formula reported below:

$$PLR = 1 - \frac{packets\ received}{packets\ sent}$$

Where

- *packets received* is the number of datagrams received correctly by the sink node;
- *packets sent* is the amount of datagrams sent by the clients.

3. The throughput in reception is the average rate of successful message delivery over a communication channel; it is measured in bits per second.

In order to determine this statistic, the metrics collection tool calculates the throughput of each application client.

The throughput of a client is obtained calculating the ratio between the data amount sent by the node and the total time interval.

The total time interval is determined summing, for each datagram which will reach the destination, the time that intervenes between the instant in which the current datagram has been sent and the instant in which the previous datagram has been transmitted.

Once the throughput of each sensor node has been defined, the system throughput in reception is obtained summing each single throughput.

4. The throughput in transmission, called also the maximum throughput, is calculated with the same procedure of the previous point. The only difference regards those datagrams that the metrics collection tool has to consider. In this case the plugin considers all the datagrams sent by the clients, also those that do not reach the destination.

Consequently, the throughput in reception considers the packets loss in the 802.15.4 links, while the throughput in transmission calculates the network capacity in the optimal situation. Therefore, the relation between the throughputs is

Throughput_TX ≅ Throughput_RX * (1+PLR).

4.6 Testing Multiple Simulations

In the previous paragraphs, the testing phase has considered single simulations, carried out on the *Test_RPL* application. In order to determine the statistics illustrated in the paragraph **4.5** that measure overall the reliability, the speed and the capacity of the 6LoWPAN links, this section presents the way to execute repeatedly the COOJA metrics collection tool according to the changing of the data traffic and of the seed of the simulation [14].

To reach this goal, the testing phase uses the metrics collection tool through the shell, without making use of the GUI. In this case, to launch the COOJA simulations, the *ant* command to type is *ant run_nogui –Dargs=/path/fils.csc*.

Unlike the GUI simulations, the simulation configuration file (*.csc*) is required as parameter of the *ant* command.

This file is written in XML and it contains the structure of the simulation that the metrics collection tool will test. In other words, the *csc* document defines:

- the radio medium used and the related links between the motes;
- the program binary image running on the nodes;
- the nodes position in the network;
- the COOJA metrics collection tool executing during the simulation;
- the javascript program to manage the duration of the simulation and to run the methods of the plugin in order to calculate metrics and statistics.

Practically, the *ant* command mentioned above, will be launched a number of times equal to the data traffic transmission rates to test. This procedure will be repeated three times on the basis of the changing of the simulation seed, in order to mediate the statistics values, deleting false values.

The data transmission rates vary from 1 packets/s to 11 packets/s, incrementing each time of 0,5 packets/s.

[14] seed determines a fixed starting point for the sequence of random values that are used in a simulation

In order to change the data rate, the metrics collection tool will modify the constant PACKET_INTERVAL (see **3.1**), defined in the *Makefile* of the Test_RPL application.
Instead, to manage the amount of traffic generated by the application clients, the testing phase requires each sensor node to send 500 datagrams to the root. To set this value, the metrics collection tool will modify the constant PACKET_NUMBER (see **3.1**), placed in the same file of PACKET_INTERVAL.

The paragraph below shows the statistics charts, calculated through the procedure just described. The related values are reported in the section **E** of the appendix.

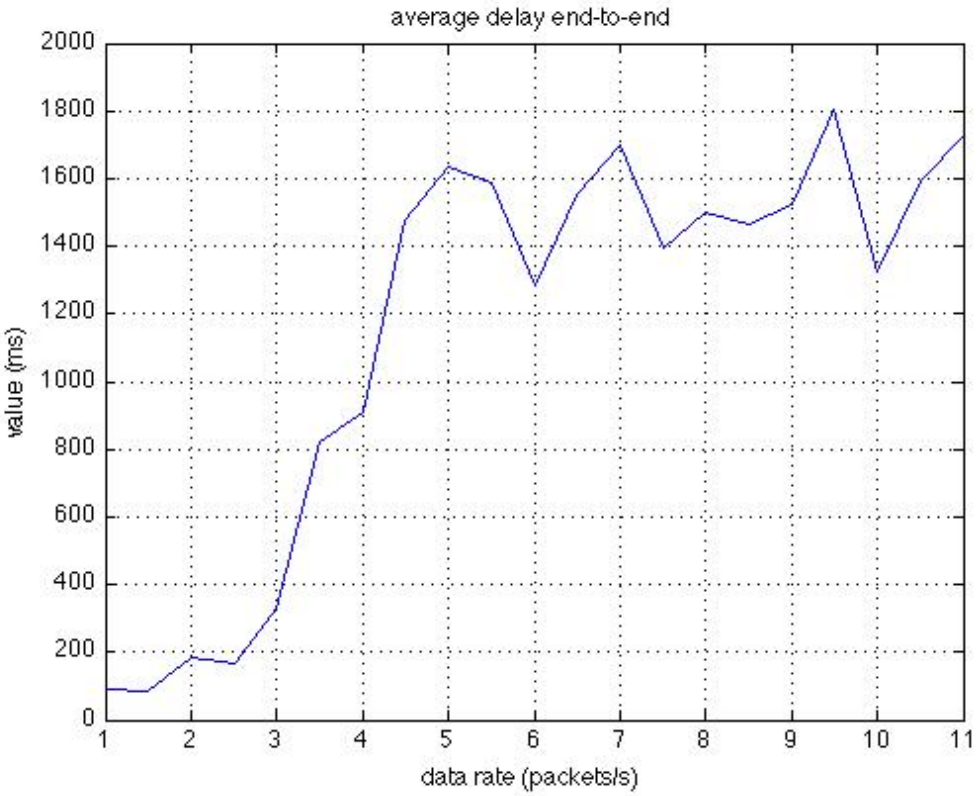

Figure 32: Average delay end-to-end – realized with MATLAB

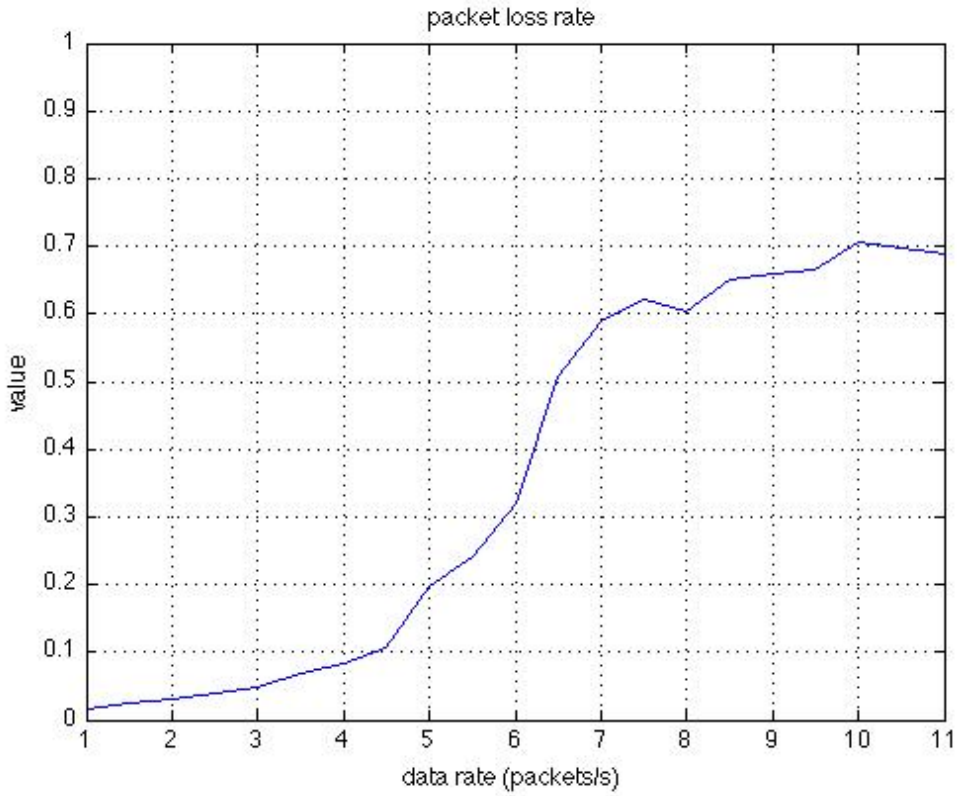

Figure 33: Packet loss rate – realized with MATLAB

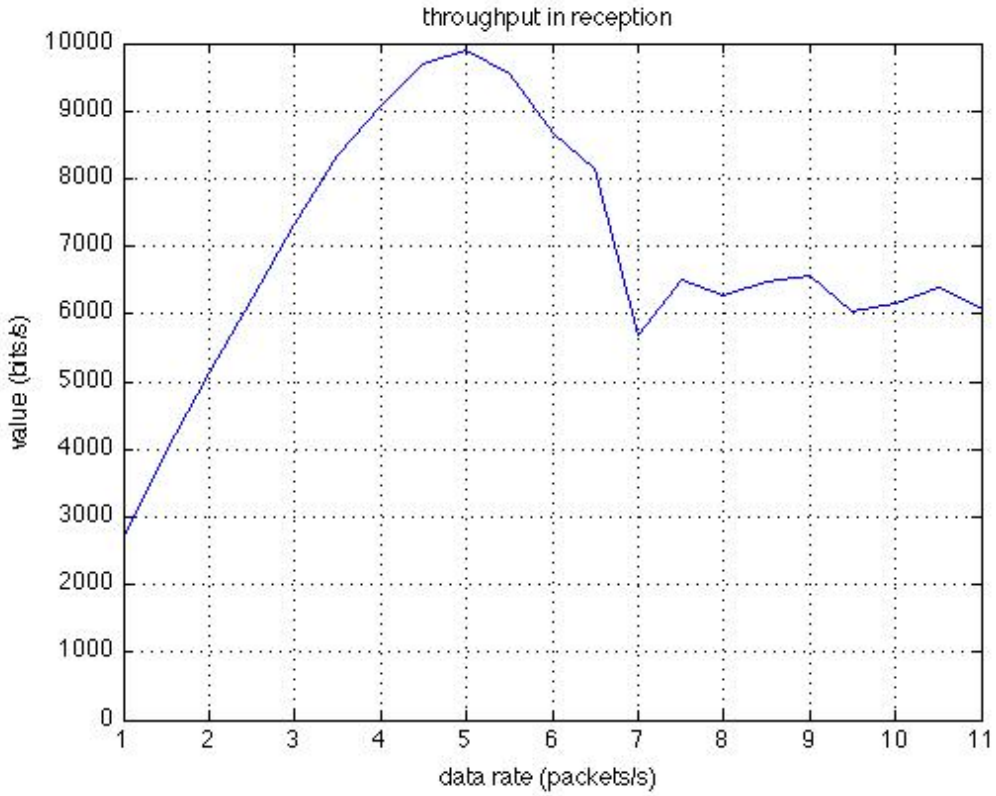

Figure 34: Throughput in reception – realized with MATLAB

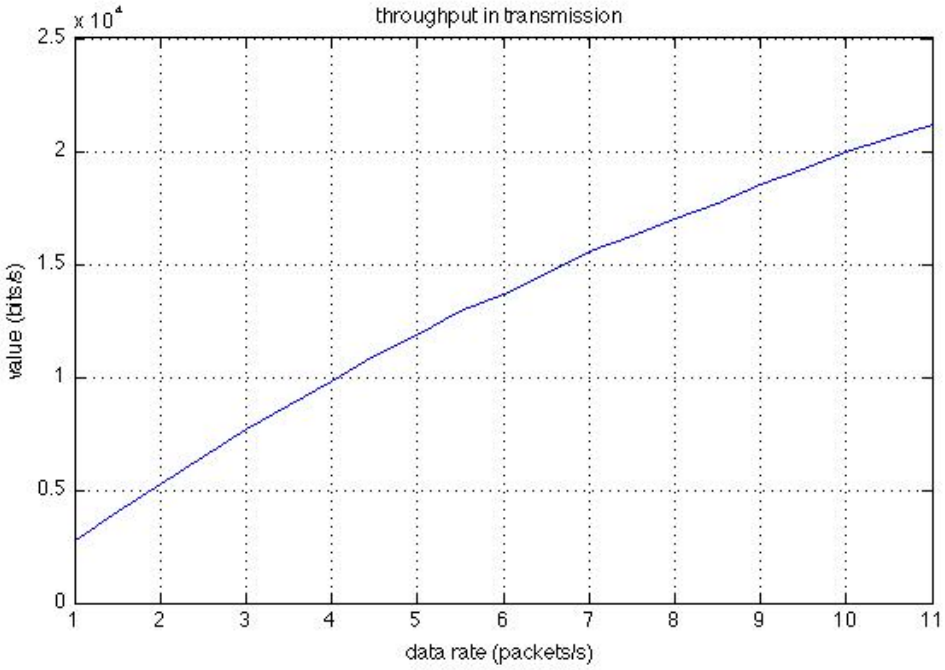

Figure 35: Throughput in transmission – realized with MATLAB

The figure 32 represents the distribution of the average delay end-to-end according to the change of the *data rate*. Increasing the number of packets generated in a second by the client nodes, the *relay* nodes have to process more datagrams and therefore the packets remain in queue for a longer time, increasing the queuing delay. If the queuing delay rises, also the delay end-to-end does.

The second graphic (figure 33) illustrates the upward trend of the packet loss rate. Incrementing the data rate, the frequency of packets exchanged on the network will grow considerably, and proportionally also the amount of datagrams lost due to the queues congestion of the *relay* nodes and to the radio collisions.

The figure 34 shows the evolution of the throughput in reception. With the data rate between 1 and 5 the function grows proportionally and then it decreases, stabilizing itself to a value of 6100 bits/s. This trend is due to the growth of the data traffic in the network, that leads to an increase of the packet loss rate. Therefore, the application clients decrease the packets transmission rate, lowering the throughput.

The figure 35 represents the throughput in transmission. The curve maintains an upward trend, unlike the previous graphic, since this statistic does not consider the queues congestion and the packet loss rate caused by the increment of the data rate. In fact this statistic represents the networks capacity in optimal conditions.

In order to verify the correctness of the graphics, the paragraph considers some particular data, taken from the section A of the appendix:

- rate 1 – packet loss rate = 0,0156 – throughput in reception = 2719 – throughput in transmission = 2772
- rate 3 – packet loss rate = 0,0485 – throughput in reception = 7310 – throughput in transmission = 7667
- rate 5 – packet loss rate = 0,1971– throughput in reception = 9888 – throughput in transmission = 11862

The aim is to prove the validity of the relation Throughput_TX ≅ Throughput_RX * (1+PLR).

Rate 1 -> 2719 * 1,0156 = 2762 ≅ 2772

Rate 3 -> 7310 * 1,0485 = 7665 ≅ 7667

Rate 5 -> 9888 * 1,1971 = 11837 ≅ 11862

The correctness of the graphics is proved by the following results.
This means that the metrics collection tool correctly captures the radio packets exchanged on the radio medium, determining metrics and statistics that allow the COOJA users to know the features of the low-power and lossy networks.

5. Conclusion

The dissertation presents the development and the testing of a metrics collection tool for 6LoWPAN networks.

The 6LoWPAN networks are made up of the sensor nodes that have the feature to send data to Internet through the networking protocol IPv6.

The IPv6 packets travel over IEEE 802.15.4 based networks, where IEEE 802.15.4 is a standard, which specifies the physical layer and the media access control for low-power and lossy networks.

To test the data communication between sensors of a 6LoWPAN network, COOJA is a java simulator widely used, which operates on three different levels: the *networking* level, the *operating system* level and the *machine code instruction set* level. COOJA allows you to build in a simple way the network topology to test, choosing the desired radio medium. Once the network is set, the simulator is able to calculate the radio power level for each mote of the simulation and to detect each radio connection sent on the radio medium.

However, COOJA does not dispose of tools capable to process metrics or capable to detect statistics for the 6LoWPAN links. Therefore, this work develops a new plugin for the COOJA simulator that permits to cover the current deficiencies. The metrics collection tool is composed of java components, which capture the radio connections sent on the radio medium, in order to detect the necessary data, required by the metrics chosen by the users.

Then, the metrics charts allow you to highlight criticalities in the network links, such as:

- the high number of radio collisions originated from the *bottleneck* node that forwards all the clients' packets to the sink node,
- the queue congestion for the *relay* nodes,
- the delay end-to-end between a source and the root.

The metrics calculation allows the metrics collection tool to detect important statistics. The average delay end-to-end, the packet loss rate and the throughput

measure overall the functioning of a 6LoWPAN network, allowing to check the speed, the reliability and the capacity of the links between the motes.
Increasing the data rate of the clients, the packet loss rate raises considerably, since the queue of the *relay* nodes reaches the maximum threshold, causing the deletion of the datagrams in excess. Therefore, the throughput in reception, after an initial climb phase, decreases and stabilizes itself, due to the high amount of packets lost. Instead, the throughput in transmission does not consider the packet loss rate and thus the graphic curve grows constantly.
If the throughput in reception is proportional to the product between the throughput in transmission and the packet loss rate, then the metrics collection tool has correctly captured and elaborated the radio packets of the radio medium. In the simulations of this work project, the relation between the statistics attests this and therefore the new plugin can be used to identify networking metrics for 6LoWPAN networks that use the RPL protocol to route the data traffic.

Future work on the metrics collection tool will concern:

- the realization of statistics according to the change of the hop count [15];
- the interaction with the routing protocol CTP [23];
- the implementation of the plugin in other simulators, such as NS 2 and TOSSIM;
- a comparison between the metrics collection tool and other similar tools of different simulators.

[15] *Hop count* refers to the number of routers through which data must pass between source and destination.

Appendix

Section A – Components of TestRPLC

The section illustrates the main components of the module *TestRPLC*.

define RPL_ROOT_ADDR 1	The node 1 is the routing tree root
define UDP_PORT 5678	The application traffic uses the UDP port 5678 for the data transport
event void Boot.booted()	- The event sets the multicast address used by RPL messages - it sets the routing tree root through RPL_ROOT_ADDR (*RootControl.setRoot()*); - it starts the routing process (*RoutingControl.start()*,*SplitControl.start()*); - it connects the application traffic to UDP_PORT (*RPLUDP.bind(UDP_PORT)*)
event void RPLUDP.recvfrom()	The event manages the reception of an application packet.
event void Timer.fired()	It defines the length of the experiment. If the TinyOS code is running by the root, the duration is equal to the EXPERIMENT_DURATION (*MilliTimer.startOneShot(EXPERIMENT_DURATION)).* Instead if the program is running on other motes, the event establishes the duration to send a single packet (*MilliTimer.startOneShot(PACKET_INTERVAL*)).
task void sendTask()	The task has the function to manage the sending of UDP packets from the sensor nodes toward the sink node (*RPLUDP.sendto()),* using RPL (*RPLRoute.getDodagId()).* The destination port is UDP_PORT (*dest.sin6_port = htons(UDP_PORT))*
event void MilliTimer.fired()	This event is linked to previous Timer.fired(). If the simulation life is finished, also the experiment is finished too; otherwise if the nodes have not finished the sending of their packets, the time simulation is extended in order to deliver the remaining messages (*MilliTimer.startOneShot(PACKET_INTERVAL + (call Random.rand16() % 100)), post sendTask*)
event void SplitControl.stopDone()	When the simulation is finished, the root shows the results of the experiment.

Section B – Details Generation XML

The section presents the structure of the methods that have the function to create the XML files, from the metrics data. The metrics data are placed in the vector defined in line 3.

```
1 If (id_metric = 1)
2         <Root>
3         for i=1 to vector(Storing).size
4                   if (vector[i] contains id_metric)
5                             if (NodeMetric)
6                                       <ITEM Mote=X Time=Y Value=Z>
7                                       </ITEM>
8                             else if (CoupleMetric)
9                                       <ITEM Source = X Destination = Y Time = Z Value = A>
10                                      </ITEM>
11        </Root>
      else if id_metric = 2
            ……...
      else if id_metric = 3
            .....
```

The change from the metrics data to XML is made possible by the classes of the packet *org.jdom.output* [16].

Section C – Details Radio Power Level in UDGM

The section illustrates how the radio power level is calculated for the motes of the radio connection, generated by the UDGM.

For a destination node the signal strength is obtained considering the following code line.

signalStrength = AbstractRadioMedium.SS_STRONG + distFactor * *(AbstractRadioMedium.SS_WEAK - AbstractRadioMedium.SS_STRONG)

[16] Java packet – Documentation in http://www.jdom.org/docs/apidocs/org/jdom/output/package-summary.html

where:

- *AbstractRadioMedium.SS_STRONG* is an *AbstractRadioMedium* constant, that indicates the maximum power level with value -10 dB;
- *distFactor* is equal to (distance between this node and source node of the connection)/(distance of the maximum transmission)
- *AbstractRadioMedium.SS_WEAK* is an *AbstractRadioMedium* constant, that represents a weak power level with value -95 dB;
- *AbstractRadioMedium.SS_STRONG* is an *AbstractRadioMedium* constant, that represents a switched-off power level with value -100 dB.

On the contrary, the signal strength for an interfered node is obtained through this code.

If (*distFactor* < 1)

signalStrength is the same value of the previous power level

else

signalStrength = AbstractRadioMedium.SS_WEAK

Section D - Installation Metrics Collection Tool In Cooja

Plugin Components

As mentioned in the section **3.2**, the plugin components are located in the directory *plugins*.

In *plugins* the directory *sixlowpan_metrics* will be created containing the following files:

- Capture.java
- Chart.java
- Create_Table.java
- FormatCode.java
- GUI_L1/SelectMetricL1.java
- GUI_L2/SelectMetricL2.java
- GUI_L3/SelectMetricL3.java
- Info_Metrics_Couple.java
- Info_Metrics_Node.java
- Info_Statistics.java

- JTableButtonMouseListener
- JTableButtonRenderer
- Metric_Table.java
- Plugin_Metrics_No_Gui.java
- Plugin_Metrics.java
- SaveChart.java
- Statistics.java
- Storing.java
- Throughput_Tx.java
- XML_Save.java

The extension to COOJA leads to the design of new classes in the directory *plugins*/*analyzers* and in the directory *cooja*.

In *analyzers* the following classes will be added:

- Analyzer_802154.java
- Analyzer_6LoWPAN.java

Instead, in *cooja* the directory *sixlowpan* is created, containing the java files reported below:

- generate_frame_datagram.java
- Datagrams/Datagram.java
- Datagrams/Frame.java
- Datagrams/Hop.java
- Datagrams/Headers/COAPHeader.java
- Datagrams/Headers/CTPHeader.java
- Datagrams/Headers/FrameHeader.java
- Datagrams/Headers/Header.java
- Datagrams/Headers/IPV6Header.java
- Datagrams/Headers/TCPHeader.java
- Datagrams/Headers/UDPHeader.java
- Nodes/Node.java
- Nodes/Queue.java
- Nodes/EndPoint/COAPEndPoints.java
- Nodes/EndPoint/CTPEndPoints.java
- Nodes/EndPoint/EndPoints.java
- Nodes/EndPoint/Frame154EndPoints.java
- Nodes/EndPoint/IPV6EndPoints.java

- Nodes/EndPoint/TCPEndPoints.java
- Nodes/EndPoint/UDPEndPoints.java

The work project, listed above, is maintained in a server of the Department of Information Engineering of the University of Padua.
The link to access is *https://telecom.dei.unipd.it/tlcrepos/castellani/iot/metricCooja/*.

Include the plugin in COOJA

Accede to the directory *config*, placed in the path *contiki*/*tools/cooja* and modify the files *cooja_applet.config* and *cooja_default.config*.
In the files you have to add the rows
se.sics.cooja.plugins.sixlowpan.metrics.Plugin_Metrics and
se.sics.cooja.plugins.sixlowpan.metrics.Plugin_Metrics_No_Gui in the section
se.sics.cooja.GUI.PLUGINS.
In this way, the plugin will be active in COOJA with or without GUI.

Section E – Statistics Values

This section contains the values of the statistics calculated by the metrics collection tool.

Average delay end-to-end

Data Rate (packets/s)	Value (ms)
1	91
1,5	83,71
2	183,50
2,5	169,512
3	328,10
3,5	821,74
4	907,12
4,5	1474,05
5	1632,74
5,5	1585,80
6	1284,53
6,5	1552,42
7	1701,46
7,5	1396,12
8	1499,63
8,5	1466,76
9	1524,02
9,5	1802,82
10	1325,44
10,5	1591,77
11	1728,13

Packet loss rate

Data Rate (packets/s)	Value
1	0,0156
1,5	0,0240
2	0,0293
2,5	0,0402
3	0,0485
3,5	0,0699
4	0,0831
4,5	0,1056
5	0,1971
5,5	0,2408
6	0,3191
6,5	0,5080
7	0,5905
7,5	0,6207
8	0,6039
8,5	0,6519
9	0,6601
9,5	0,6655
10	0,7072
10,5	0,6964
11	0,6872

Throughput in reception

Data Rate (packets/s)	Value (bits/s)
1	2719
1,5	3971
2	5140
2,5	6252
3	7310
3,5	8341
4	9078
4,5	9681
5	9888
5,5	9545
6	8656
6,5	8141
7	5691
7,5	6517
8	6260
8,5	6463
9	6566
9,5	6041
10	6163
10,5	6376
11	6077

Throughput in transmission

Data Rate (packets/s)	Value (bits/s)
1	2772
1,5	4062
2	5286
2,5	6508
3	7667
3,5	8772
4	9843
4,5	10935
5	11862
5,5	12923
6	13708
6,5	14566
7	15600
7,5	16253
8	17010
8,5	17715
9	18543
9,5	19169
10	19987
10,5	20558
11	21211

References

[1] JeongGil Ko and Andreas Terzis, Johns Hopkins University, Stephen Dawson, Haggerty and David E. Culler, University of California at Berkeley, Jonathan W. Hui, Cisco Systems, Inc. Philip Levis, Stanford University, "Connecting Low-Power and Lossy Networks to the Internet", IEEE Communications Magazine, April 2011.

[2] Carsten Bormann, JP Vasseur and Zack Shelby, "The Internet of Things", IETF Journal, vol. 6, no. 2, November 2010.

[3] IEEE standard 802.15.4, June 2006.

[4] T. Narten, IBM, E. Nordmark, Sun Microsystems, W. Simpson, Daydreamer, H. Soliman, Elevate Tehnologies, "Neighbor discovery for IP version 6 (IPv6)", IETF RFC 4861, September 2007.

[5] T. Winter, Ed., P. Thubert, Ed., Cisco Systems, A. Brandt, Sigma Designs, T. Clausen, LIX, Ecole Polytechnique, J. Hui, Arch Rock Corporation, R. Kelsey, Ember Corporation, P. Levis, Stanford University, K. Pister, Dust Networks, R. Struik, JP. Vasseur, Cisco Systems, "RPL: IPv6 Routing Protocol for Low power and Lossy Networks", draft-ietf-roll-rpl-19, March 2011.

[6] D. S. J. De Couto et al., "A High-Throughput Path Metric for Multi-Hop Wireless Routing", Proc. 9th ACM lnt'l. Conf. Mobile Computing and Networking, San Diego, California, USA, September 2003.

[7] TinyOS site, http://www.tinyos.net/.

[8] Martin Stehlìk, Master Thesis, Masaryk University, "Comparison of Simulators for Wireless Sensor Networks", Spring 2011.

[9] R. Droms, Ed., Cisco, J. Bound, Hewlett Packard, B.Volz, Ericsson, T. Lemon, Nominum, C.Perkins, Nokia Research Center, M. Carney, Sun Microsystems, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", IETF RFC 3315, July 2003.

[10] R. Hinden, Nokia, S. Deering, Cisco Systems, "IP Version 6 Addressing Architecture", IETF RFC 3513, February 2006.

[11] G. Montenegro, Microsoft Corporation, N. Kushalnagar, Intel Corp, J. Hui, D. Culler, Arch Rock Corp, "Transmission of IPv6 packets over IEEE 802.15.4 Networks", IETF RFC 4944, September 2007.

[12] Contiki site, http://www.contiki-os.org/.

[13] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, Thiemo Voigt, Swedish Institute of Computer Science, "Cross-Level Sensor Network Simulation with COOJA", In proceeding of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications, Tampa, Florida, USA, November 2006.

[14] G. Wittenburg and J. Schiller, "Running realworld software on simulated wireless sensor nodes", In Proc. of the ACM Workshop on Real-World Wireless Sensor Networks (ACM REALWSN'06), Uppsala, Sweden, June 2006.

[15] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications", In Proceedings of the first international conference on Embedded networked sensor systems, pages 126–137, New York, NY, USA, 2003.

[16] B. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing", In International Conference on Information Processing in Sensor Networks (IPSN), IEEE Press Piscataway, NJ, USA, 2005.

[17] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, Thiemo Voigt, Nicola Tsiftes, Swedish Institute of Computer Science, “Demo Abstract: MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards”, In Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN 2008), Bologna, Italy, January 2008.

[18] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Swedish Institute of Computer Science, Robert Sauter, Pedro José Marrón, University of Bonn and Fraunhofer IAIS, “Demo Abstract: Towards Interoperability Testing for Wireless Sensor Networks with COOJA/MSPSim”, In: 6th European Conference on Wireless Sensor Networks (EWSN), Cork, Ireland, 11-13 Feb 2009.

[19] Carlo Alberto Boano and Kay Römer, Universität zu Lübeck, Lübeck, Germany - Fredrik Österlind and Thiemo Voigt, Swedish Institute of Computer Science Kista, Stockholm, Sweden, “Demo Abstract: Realistic Simulation of Radio Interference in COOJA”, In European Conference on Wireless Sensor Networks (EWSN 2011), Bonn, Germany, February 2011.

[20] Fredrik Österlind, Joakim Eriksson, Adam Dunkels, Swedish Institute of Computer Science, “Demo Abstract: Cooja TimeLine: A Power Visualizer for Sensor Network Simulation”, In Proceeding of the 8th ACM Conference on Embedded Networked Sensor Systems, New York, NY, USA, 2010

[21] Eriksson, Joakim and Österlind, Fredrik and Voigt, Thiemo and Finne, Niclas and Raza, Shahid and Tsiftes, Nicolas and Dunkels, Adam, “Demo abstract: accurate power profiling of sensornets with the COOJA/MSPSim simulator”, In: Sixth IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2009), 12-15 Oct 2009, Macau SAR, P.R.C..

[22] Richard Huber, Philipp Sommer, and Roger Wattenhofer, Computer Engineering and Networks Laboratory ETH Zurich, Switzerland, “Demo Abstract: Debugging Wireless Sensor Network Simulations with YETI and COOJA”, In: 10th ACM/IEEE

International Conference on Information Processing in Sensor Networks (IPSN), Chicago, IL, USA, April 2011.

[23] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, Sukun Kim, Philip Levis, and Alec Woo, “tiny OS page for CTP”, http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html, 2006–2007.