FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Modeling the Processing Delays of Internet of Things Nodes in the ns-3 Network Simulator

Guilherme Daniel Gonçalves Ferreira

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Manuel Alberto Pereira Ricardo

External Supervisor: Pedro Filipe Cruz Pinto

January 27, 2017

© Guilherme Ferreira, 2017

Resumo

A Internet of Things (IoT) está a emergir como uma arquitetura de informação global que permite a objetos do dia-a-dia estarem ligados à Internet. A IoT pode ser concretizada pelo uso de Low-Rate Wireless Personal Area Networks (LR-WPANs), i.e. por redes normalmente compostas por nós sensores com memória e capacidade de processamento limitados. Estas restrições permitem maior longevidade na duração das baterias deste nós, contudo, implicam um aumento dos tempos relacionados com as tarefas de processamento. Esta desvantagem é particularmente relevante quando se avaliam os atrasos destes nós. Os atrasos de processamento que são geralmente negligenciados em ambientes de simulação tornam-se relevantes devido a estas condições.

As redes orientadas à IoT são projectadas para suportar a instalação em grande escala e assim, um número massivo de nós sensores tem de ser testado e instalado. Antes da instalação destas redes é necessário testa-las para garantir que todos os nós funcionam como esperado. Portanto, simulações precisas tem de ser implementadas visto assumirem relevancia quando simulam redes de grande escala. Portanto, é obrigatório um simulador de redes permitir simulações rigorosas para estas redes orientadas à IoT.

Atualmente, o simulador ns-3 não tem em consideração atrasos de processamento em sensores. Esta dissertação tem como objetivo aprimorar este simulador, fornecendo-lhe a capacidade de simular os tempos de processamento de redes de sensores orientadas a IoT. Consequentemente, esta trabalho produziu duas contribuições. A primeira é uma análise experimental de um ambiente orientado para o IoT, fornecendo modelos que simulam os atrasos de processamento das redes desta variedade. E o segundo um módulo de simulação ns-3 que permitirá que o ns-3 também simule os atrasos de processamento em nós de sensores de uma LR-WPAN.

Abstract

The Internet of Things (IoT) is emerging as a global information architecture that allows everyday objects to be connected to the Internet. The IoT is facilitated by the use of Low-Rate Wireless Personal Area Networks (LR-WPANs) which are composed of sensors with limited memory and processing capabilities. These constraints allow the nodes to extend their battery lifetime, however, they also impose higher data processing times. This downside is particularly relevant when assessing these nodes' delays. Processing delays that are usually neglected in simulation environments become relevant due to these conditions.

IoT oriented networks are designed to support large scale deployment and thus, massive number of sensor nodes have to be tested and deployed. Prior to the deployment of such networks, their testing is necessary in order to guarantee that all the nodes have the expected behaviour. Therefore, precise simulations must be implemented since they assume particular relevance when simulating large scale networks. Thus, network simulators should provide accurate simulations for these IoT oriented networks.

Currently the ns-3 simulator does not take into account the processing delays on a sensor node. This dissertation aims at enhancing this simulator by providing the ability to simulate the processing times of IoT oriented sensor networks. Current work has two main contributions. First, it provides an experimental analysis of an IoT oriented environment, providing models that simulate the processing delays of networks. And second, it proposes a ns-3 simulation module that will enable ns-3 to also include processing delays in sensor nodes when simulating a LR-WPAN.

Contents

List of Figures

List of Tables

List of Abbreviations

6LowPAN	IPv6 over Low Power Wireless Personal Network
AP	Access Point
ARP	Address Resolution Protocol
APS	APplication Support sub-layer
CSMA	Carrier Sense Multiple Access
EXTREME	EXperimental Testbedfor ResEarch on Mobile nEtworks
FCS	Frame Check Sequence
FFD	Full-Function Device
GTS	Guaranteed Time Slot
ID	Identifier
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
INESC-TEC	Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência
IoT	Internet of Things
IP	Internet Protocol
LR-WPAN	Low-Rate Wireless Personal Area Network
MAC	Media Access Control
MCU	MicroController Unit
MTU	Maximum Transmission Unit
NDP	Neighbor Discovery Protocol
NLDE	Network Layer Data Entity
NLME	Network Layer Management Entity
OMNet++	Objective Modular Network Testbed in C++
OS	Operating System
PAN	Personal Area Network
PSDU	Physical Service Data Unit
RAM	Random Access Memory
RFC	Request For Comments
RFD	Reduced-Function Device
ROM	Read-Only Memory
RPL	Routing Protocol for Low power and Lossy Networks
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WPAN	Low-rate Wireless Personal Area Network
μIp	Micro IP
ZDO	ZigBee Device Objects

Chapter 1

Introduction

The Internet of Things (IoT) is emerging as a global information architecture that allows everyday objects to be connected to the Internet. Being interconnected, these objects can collaborate, coordinate actions, and exchange useful information in order to create smart environments with benefits to the human being. This is achieved by embedding this everyday objects with electronics and network connectivity in order to gather and send processing relevant data, such as temperature, humidity, moisture, pressure, among others.

Since nodes in these networks are constantly monitoring the surrounding environment it is required that their power consumptions are constrained, in order to extend the nodes' battery lifetime. This is achieved by reducing processing and memory capabilities in these sensors which, in turn, also demands low footprint software requirements, e.g. the Contiki Operating System (OS) [2], which is specifically designed for memory restrained units.

Constrained memory and processing capabilities of these nodes impacts directly in the amount of time it takes for these systems to process information. Larger processing delays are expected and they will be noticeable in all the actions performed by these nodes.

1.1 Motivation

In a communication scenario where a packet is generated and sent from one node to another, a delay can be accounted from source to destination. This delay represents the elapsed time since data is generated up to the moment the data is received in the destination node, and can be obtained by summing delay components such as processing, queuing, transmission and propagation. Although in some cases processing delays are negligible when compared to the other delays, in particular scenarios they can be relevant to take into account. In networks with nodes using the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 Standard [3], where simple Linux-based operating systems and constrained hardware capabilities are implemented, it is expected that the processing delays associated with these devices to be in the millisecond range, which should not be disregarded when using time sensitive applications.

Multiple hardware and software solutions can be found that are designed specifically for the IoT. At a preliminary stage prior to deployment, network simulators, such as the ns-3, allow testing without the need of real testbeds, a huge advantage when large scale devices and networks are assumed. In this context, there is a high demand for reliability and precision of IoT hardware/software network simulators.

1.2 Problem Statement

The use of network simulators is encouraged and it is also important to provide developers simulators with high levels of reliability and close-to-real results. As stated in the previous section, processing delays are relevant when assuming the restrained capabilities of the nodes used in IoT networks. In case developers choose to test an IoT deployment in a network simulator, this should take into account all the delay components.

Network simulators such as Cooja [4] or ns-3 [5] can be used to test the scenarios intended to be deployed. Regarding the Cooja simulating environment, real and emulated scenarios may present different results in terms of timing inaccuracies as highlighted in [6]. Concerning ns-3, it can be verified that it disregards processing delays of any kind and thus, timing inaccuracies in this simulator are expected.

The main problem this dissertation addresses is the ns-3 simulator's inaccuracy when it is used to test nodes for the Internet of Things based networks. It is relevant to improve the precision of this simulator since it allows simulation of large scale networks, integrated with other devices without the heavy processing requirements necessary when the nodes are emulated (e.g. as they are in Cooja simulator).

1.3 Objectives

The major goal of this dissertation is to improve the precision of the ns-3 network simulation tool, in particular when simulating nodes in Low-Rate Wireless Personal Area Networks (LR-WPANs). This is achieved by testing the Tmote Sky wireless sensor node running the Contiki OS, allowing the collection of the real elapsed times in multiple processing stages within each node on a LR-WPAN. Subsequently the gathered times are modeled and added to the ns-3 network simulator structure. Therefore this dissertation has two specific goals:

- To study and model the processing delays in a LR-WPAN, using different combinations of communication protocols;
- To implement the processing delays in ns-3 allowing the modeled delay times to be accounted when running a simulation with specific communication protocols.

1.4 Document Structure

The structure of this dissertation is as follows. Chapter 2 provides a background review of protocols and technologies, including details of relevant software and hardware for WSN, the IEEE 802.15.4 standard, and network simulators. Chapter 3 details the tests conveyed to gather real delays using a specific mote with the Contiki OS and a set of protocols in the network stack. Chapter 4 describes the methods applied when creating the ns-3 module. Finally chapter 5 presents the conclusion of this work and outlooks possible future work.

Chapter 2

Background

This chapter presents a review of the background technologies and protocols used in this dissertation. In Section 2.1 the IEEE 802.15.4 Standard is reviewed. In Sections 2.2 and 2.3 specific software and hardware solutions oriented towards IoT are detailed. The Section 2.4 presents an overview on simulation techniques allowing a greater understanding of network simulators and also provides analysis of some discrete-event network simulators.

2.1 IEEE 802.15.4 Standard

The IoT concept may assume that nodes are restrained in their energy and transmission, and thus, these nodes form networks which are usually deployed as short range communication personal networks, or just Personal Area Network (PAN). The standards for Wireless Personal Area Networks (WPAN) are developed by the IEEE 802.15 Working Group, and one of the most relevant standards for current context is the IEEE 802.15.4 which defines both physical and data link layers in the context of a LR-WPAN, which consists of networks with nodes using low data rates and low power consumptions.

The IEEE 802.15.4 standard specifies two different device types that can participate in the network: a Full-Function Device (FFD), that is capable of communicating with every node in the system and supports the full protocol and can consequently serve as a coordinator, and a Reduced-Function Device (RFD), that does not need to send large amounts of data at a time and therefore only communicates with one FFD at a time. Every node is attributed two addresses, a 64 bit length global IDentifier (ID) and a 16 bit length Personal Area Network (PAN) specific address attributed by the PAN coordinator when the device connects to the network.

The IEEE 802.15.4 standard defines four frame types as presented in Fig. 2.1. The Data frame that conveys the data payload. The Acknowledgment frame is used to convey that the reception of another frame was a success. The Beacon frame which is used by the coordinator to transmit beacons and coordinate the network. And the Command frame is used to set up and configure the clients [3].

Data frame format					
Octets: 2	1	variable	0/5/6/10/14	variable	2
Frame control	Sequence number	Addressing Fields	Auxiliary Security Header	Data Payload	Frame Check Sequence(FCS)

Acknowledgment frame format		
Octets: 2	1	2
Frame control	Sequence number	FCS

Beacon frame format									
Octets: 2	1	4/10	0/5/6/10/14	5	2	variable	variable	variable	2
Frame control	Sequence number	Addressing Fields	Auxiliary Security Header	Superframe Specification	Guaranteed Time Slot (GTS) Fields	Frame control	Pending Address Fields	Beacon Payload	FCS

Command frame format						
Octets: 2	1	4/10	0/5/6/10/14	1	variable	2
Frame control	Sequence number	Addressing Fields	Auxiliary Security Header	Command Frame Identifier	Command Payload	FCS

Figure 2.1: MAC frame formats

2.1.1 Internet Protocol (IP) stacks

The common TCP/IP stack has heavy code footprint in order to be used in memory-constrained devices and, in order to tackle this problem some lightweight alternatives were designed such as the Micro IP (μIP) and the (μIPv6).

The μIP [7, 8] stack implements the main TCP/IP protocols utilizing IPv4. It includes an implementation of the main TCP/IP stack protocols, such as the Address Resolution Protocol (ARP), the Internet Control Message Protocol (ICMP), the Internet Protocol (IP), the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

IoT networks' commonly assume large scale deployment and thus, the μIP addressing support is insufficient since it inherited the IPv4's restricted addressing space. Therefore, the use of IPv6 protocol and its bigger addressing space led to the creation of the μIPv6 [9]. This stack is built on top of the Contiki OS, is consequently fully integrated in that software system, and it only implements main protocols that were seen as relevant when designing LR-WPANs. The μIPv6 uses low memory requirements as it requires a ROM size of about 11488 Bytes and a RAM of 1748 Bytes. The μIPv6 implements the IPv6, the ICMPv6, the Neighbor Discovery Protocol (NDP), the TCP and the UDP.

2.1.2 6LowPAN

The Maximum Transmission Unit (MTU) size for IPv6 packets is 1280 Bytes, and many IPv6 larger size packets do not fit in an IEEE 802.15.4 frame which has a maximum size of 127 Bytes. Therefore in order to transport packets using both IPv6 and IEEE 802.15.4 fragmentation and

reassembly of the packets is required before they are sent to lower layers. This led to creation of the IPv6 over Low power Wireless Personal Area Networks (6LowPAN) an adaptation layer between the MAC and the network layers, which allows the transmission and reception of IPv6 packets over the IEEE 802.15.4 standard, by performing packet the needed fragmentation and reassembly. When the packet to transmit is larger than the IEEE 802.15.4 frame payload then this packet is fragmented at the source, and reassembled at the destination.

As presented in Fig. 2.2, the data payload of a frame transmitted over the IEEE 802.15.4 standard can assume sizes from 88 Bytes to 188 Bytes. Also, this frame payload needs to include the IPv6 header, with a size of 40 Bytes, and the forth layer protocol header, UDP or TCP, with a size of 8 Bytes and 20 Bytes, respectively. Therefore, in the worst case scenario which can be seen in Fig. 2.3 the frame payload will account an header of 60 Bytes, leaving the application data with only 28 Bytes. This led to the inclusion of an header compressing feature in the 6LowPAN adaptive layer. 6LowPAN is capable of performing header compression for the IPv6, the ICMP and the UDP headers as standardized in [10, 11].

Maximum Header Size Data Frame					
Octets: 2	1	20	14	88	2
Frame control	Sequence number	Addressing Fields	Auxiliary Security Header	Data Payload	FCS

Minimum Header Size Data Frame				
Octets: 2	1	4	118	2
Frame control	Sequence number	Addressing Fields	Data Payload	FCS

Figure 2.2: Maximum and minimum header sizer for 802.15.4 frame

2.1.3 ZigBee

The ZigBee is a IEEE 802.15.4 standard based specification that is used to create PANs for small scale projects with the need of a wireless connection. It is a technology that employs low data rate, low cost and low power consumption. Its data rate can reach 250kbps and the distance ranges from 10m to 70m. The common power consumption of Zigbee-based devices is close to 30mA during standby mode. The ZigBee specification defines a particular stack architecture [12] that can be seen in figure 2.4. The ZigBee MAC and physical layers structure is the same as the IEEE 802.15.4 standard.

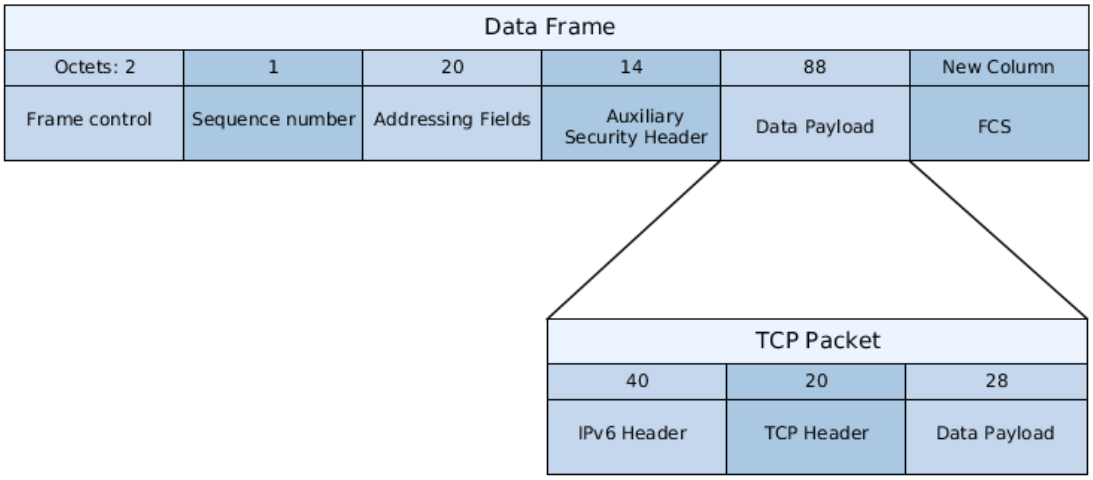

Figure 2.3: 802.15.4 frame containing an IPv6 and TCP packet

The physical layer consists of a 32 bit preamble that synchronizes the system. The first 8 bits of the frame mark the end of the preamble. The frame length is 8 bits and the Physical Service Data Unit (PSDU) has 127 Bytes in order to handle and manage data received from the previous layer [13]. There are two physical layers defined by this standard that operate in two different frequency ranges one at 868/915 MHz and the other at 2.4 GHz. The lower frequency one covers the European band, which is 868 MHz and the band used in the United States of America and Australia, the 915 MHz. The layer with the higher frequency is used worldwide.

The MAC layer was already presented in section2.1.

The ZigBee network layer works as an interface between the application and the IEEE 802.15.4 MAC layer. To do this it contains two service entities: the Network Layer Data Entity (NLDE), which provides the data transmission service, and the Network Layer Management Entity (NLME), which provides a management service. The NLME uses the NLDE in order to achieve its management tasks and also maintains a database of all the controlled objects, the network information base.

The application layer is composed of an APplication Support sub-layer (APS) and the Zigbee Device Objects (ZDO). The APS provides an interface between the application layer and the network layer through a set of services that are used by both the ZDO and the manufacturer defined objects.

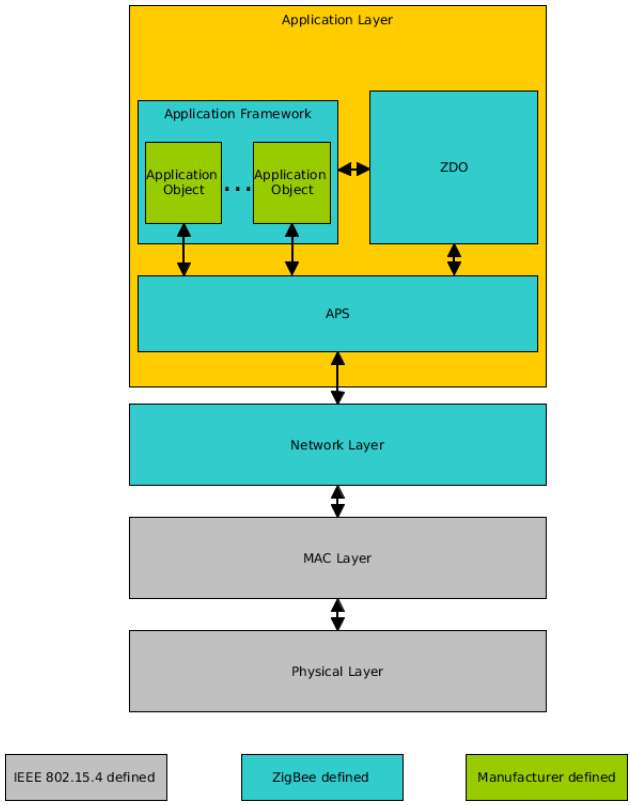

Figure 2.4: ZigBee Stack Architecture

2.2 Software

Since the sensor nodes in an IoT implementation may assume limited energy, communication and processing capabilities, the software needs to have low code footprint. A wide range of operating systems for IoT are available such as TinyOS, the Contiki OS [2], the RIOT OS[14], the Lite OS [15] and the FreeRTOS[16]. Following, details on Contiki OS are presented.

Contiki OS is an operating system, designed for memory constrained systems. It was created by Adam Dunkels and then further developed by a wide number of developers worldwide that still provide support and new features to the software. As reported in [17], the Contiki OS has been ported to multiple microcontroller architectures, including the Texas Instruments MSP430, the Atmel AVR, the Hitachi SH3, and the Zilog Z80. The AVR port was developed by the creators in only a few hours and the Zilog port was done by an exterior party to the Contiki OS development team and was done in a single day. The Contiki OS memory footprint when using full IPv6 networking and Routing Protocol for Low power and Lossy Networks (RPL) is of 10 kBytes of Random Access Memory (RAM) and of 30 kBytes of Read-Only Memory (ROM), which represents 100% of available RAM and 62.5% of available ROM of Tmote Sky sensor node [18].

The Contiki OS includes three network stacks that are specifically created to be used by IoT oriented devices: Rime, μIPv4 and μIPv6. Rime [19] is a lightweight layered communication stack designed for sensor networks. The Communication primitives in this stack are based on typical sensor network protocols use. Therefore applications and protocols running on top of it attach at any layer and use any of those primitives. Additional protocols not available in the stack can be developed if implemented directly on top of communication primitives already in the stack. The code memory footprint of the individual Rime modules are aimed to be lightweight. The current total footprint of the stack is less than two kiloBytes but more features can cause these to increase.

The μIPv4 and μIPv6 stacks are direct protocol implementations of the stack presented in section 2.1.1. And as stated μIPv6 was built directly on top of the Contiki OS.

2.3 Hardware

Concerning IoT oriented hardware, a wide range of products can be used including the Seed-Eye [20], the WiSMote [21], the Z1 [22], the MICAz [23] and the Tmote Sky [18]. All of these products are compatible with the IEEE 802.15.4 Standard and Contiki OS. Also, they can use ContikiOS oriented simulator Cooja.

The Tmote Sky includes a TI MSP430F1611 MicroController Unit (MCU),a TI CC2420 radio chip, 10 kBytes of RAM, 48 kBytes of ROM (flash), and an external flash of 1024 kBytes. It's MCU is presented as part of an ultra low power microcontroller family, the MSP430. In specific the MSP430F1611 is announced as having a power consumption of 330 μA (at 1 MHz using 2.2 V), 1.1 μA in standby mode and 0.2 μA in Off Mode (RAM retention) [24]. It's processor runs at 8 MHz, which is, as expected, much less than an average computer. The CC2420 is compliant with the IEEE 802.15.4 standard, and is designed for low power, draining 18.8 mA when receiving, and 17.4 mA when transmitting. It operates at 2.4 GHz, and achieves data rates of 250 kbit/s. The reduced processing power of the board increases the time it takes to execute programming routines and the reduced data rate of the radio chip increases the time taken to transmit a packet, and research efforts such as [25] explore these conclusions.

2.4 Network Simulators

Prior to IoT devices deployment the developer's intermediate steps may include an implementation of a testbed and the use of a network simulator to test and validate results. The latest is usually the best option when developers need to test topologies and communications in large scale scenarios where large hundred or even thousands of devices are to be deployed, for instance, as assumed by authors in [26]. Network simulators are not only employed in the testing and validation of network systems, but sometimes they can also be used in learning environments [27] or to address improvements in data center [28].

Though, many researchers rely on simulators to validate their work, even when the creation of a testbed is feasible, such as [29] and [30]. Consequently the validity of the networks simulator results is highly dependant on the reliability of the simulator used. Many network simulators exist providing advantages and disadvantages facing each other.

A myriad of network simulators are available such as ns-2 [31], ns-3 [5], Objective Modular Network Testbed in C++ (OMNet++) [32], SimPy [33], Cooja [4], GreenSim [34], Idea1 [35] and CupCarbon [36].

In the current work, two specific simulators were studied and used: Cooja, a simulator built for the simulation of specific IoT nodes running the Contiki operating system; and ns-3, a network topology and devices simulator oriented to wireless and wired networks.

2.4.1 Simulation Methods

In order to better comprehend network simulation tools, three simulation methods are distinguished: discrete event, continuous and discrete rate.

Discrete event simulation consists in the use of events, discrete moments in time, to represent state changes to the system. This type of simulation comprises a list of future events and temporal evolution is obtain by transitioning to the next event on the list, and with each state change timers, variables and future events are updated. This type of simulation leads to a discrete temporal evolution since state changes are only taken into account for the instant in time represented by an event [37].

Continuous simulation consists in the creation of an equational model were the state variable changes between states in a continuous and smooth way.

The discrete rate simulation models the behaviour of linear continuous systems, hybrid systems, and any high speed/large volume system involving the rate-based movement or flow of material from one location to another. The flow movement speed during a simulation run is called the effective rate, this value stays constant between events. Calculation happens at events were the effective rate suffers alterations. There isn't a concept of time continuity in this type of simulation since the simulation clock only updates when events occur. An event calendar is maintained with a list of future events, the predicted time of events may be altered due to other system events rescheduling the calendar in the process [38]. It is a combination of continuous and discrete simulations, in the fact that it recalculates flow rates, which are continuous variables but it is also event based.

A wide range of network simulators use discrete-event simulation since networks can be simulated by way of events. This is due to the fact that only specific moments in a network need to be taken into account in order for it to be properly simulated, mainly the reception or transmission of packets.

2.4.2 Cooja

Cooja is a Java-based simulator designed for the simulation of network sensors running the Contiki OS [4]. It presents itself as a cross-level simulator, a new type of simulation that enables simulations at different levels, the network level, the operating system level, and the machine code instruction set level.

Cooja allows the emulation of the full source code in specific network nodes such as the WiSMote, the Z1, the MICAz, and the Tmote Sky. Although Cooja simulator presents high levels of accuracy, authors in [6] refer that this simulator presents errors between the real and simulated transmission related delays. These errors are minor when emulating the SkyMote, but large in case it is used the Zolertia Z1. These inaccuracies are dependant of the received packet size. In particular, when running the Contiki OS on a Sky hardware platform differences of around 80μs are expected when loading a packet using Cooja versus a live scenario. The same contribution also reveals that Cooja is capable of running other operating systems in the simulated nodes, and the study also evaluated the use of RiotOS. The delay inaccuracies verified for this situation were worse, and could not be disregarded when using the Z1 hardware platform. This leads to the assumption that even though other operating systems can be used in the Cooja network simulator, their results are less reliable.

2.4.3 Ns-3

Ns-3 is a free open source discrete event network simulator. It was created with the purpose of replacing the older network simulator ns-2, even thought this older version is still used. Ns-3 is not backwards compatible since the developers lacked resources to maintain this compatibility. Therefore it was written from scratch in C++, even though it is capable of running Phyton programs. It is mostly developed through volunteer contributions by developers all over the world.

As stated in [39], few studies were done regarding the validation of the timing accuracy of ns-3. Although testing and validation of older modules are not easy to find, a lot of developers are constantly proposing the creation of new modules for the simulator. The authors in [40] proposed a new module that allows for multi-channel Wi-Fi scanning and Access Point(AP) selection. The model is validated and tested to improve its accuracy. In [41] a resource-constrained software router module is created deriving from a ns-3 module that previously existed. Authors in [42] have assessed and validated the IEEE 802.11 MAC Model module. Using the EXperimental Testbed for ResEarch on Mobile nEtworks (EXTREME) testbed [43] the results obtained proved that even though overall the module had satisfactory responses, in several cases there are noticeable differences between the simulated and testbed values.

Up to the moment of writing this dissertation, ns-3 accounts for fourty three modules, and some of these modules are relevant for the current context, such as the 6LowPAN module which implements the adaptive layer with the same name. It complies with Request for Comments (RFC) 4944 [10] and RFC 6282 [11] except in the lack of mesh-under architecture [44]. The LR-WPAN module which is compliant with the IEEE 802.15.4 Standard. The ns-3 also possesses an IPv6

[45] implementation which was created adapting the previously existing and still functioning IPv4 implementation.

Unlike Cooja, ns-3 assumes all nodes are simple a version of the real hardware, and therefore it does not take into account the different processing capabilities of each sensor. Further exploration of the simulator revealed that it does not take into consideration processing delays of any kind. Since none of the analysed modules had processing delay accountability.

Chapter 3

Experimental Analysis

This chapter presents the experimental analysis that was conducted in order to collect and analyze the real processing delays verified in the nodes of a LR-WPAN. The hardware/software pair used for the experiments consisted of a Tmote Sky sensor with Contiki OS. In the previous work [25] the authors have chosen Contiki and collected the processing delays when using the Contiki 2.5. Since then, newer versions of Contiki have been launched. The actual version of Contiki is the 3.0 with major differences in code and programming flow, and thus, in this in this work the process to collect delays and order to analyse the delays using the Contiki 3.0 version , current work to apply real delay processing values of the actual Contiki OS stack to ns-3 simulations, new simulations and real experiments were performed.

Using the Cooja simulator, a study of the Contiki network stack was conducted in order to assess if it had suffered significant changes compared with older stack versions, where relevant modifications were verified and accounted.

Later on, it was implemented a testbed based on Tmote sky real hardware nodes.This hardware platform is commonly used for IoT implementations and a number of motes were already available in Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência (INESC-TEC) Research Institute.

3.1 Simulated Scenario

Prior to deployment in order to gather real nodes' processing delays, the Cooja network simulation tool was used to test the code to be used in the live testbed. As of Contiki 3.0 only two types of network stacks are implemented for the Sky type processing unit: Rime and μIpv6. This was verified by reviewing the sky platform specific files, which lacked a μIPv4 implementation. Since the ns-3 simulator does not currently possess an implementation of the Rime network stack only the μIpv6 stack was used in the tests.

With the purpose of maximizing the amount of scenarios that could be simulated in ns-3, the following four schemes were defined employing different protocol combinations.

- UDP communication with 6LowPAN header compression

- UDP communication without 6LowPAN header compression
- TCP communication with 6LowPAN header compression
- TCP communication without 6LowPAN header compression

Work was done with the goal of having those four schemes fully tested and its processing delays modeled.

Two programs were initially created in the Sky motes using Contiki OS, each one composed of client and server parts: the first performed the transfer of packets using UDP protocol and the second using TCP protocol, in the transport layer. After first experiments, a forwarder node was added to the UDP scenario, which routed packets sent from the client to the server, by using the RPL protocol. For both UDP and TCP scenarios the activation and deactivation of 6LowPAN header compression is possible and can be achieved simply by altering the minimum packet size for 6LowPAN header compression to be enabled. Which can be changed in a library called contiki-conf.h in the appropriated platform type, in this case named as "sky".

The topology is presented in Figure 3.1.

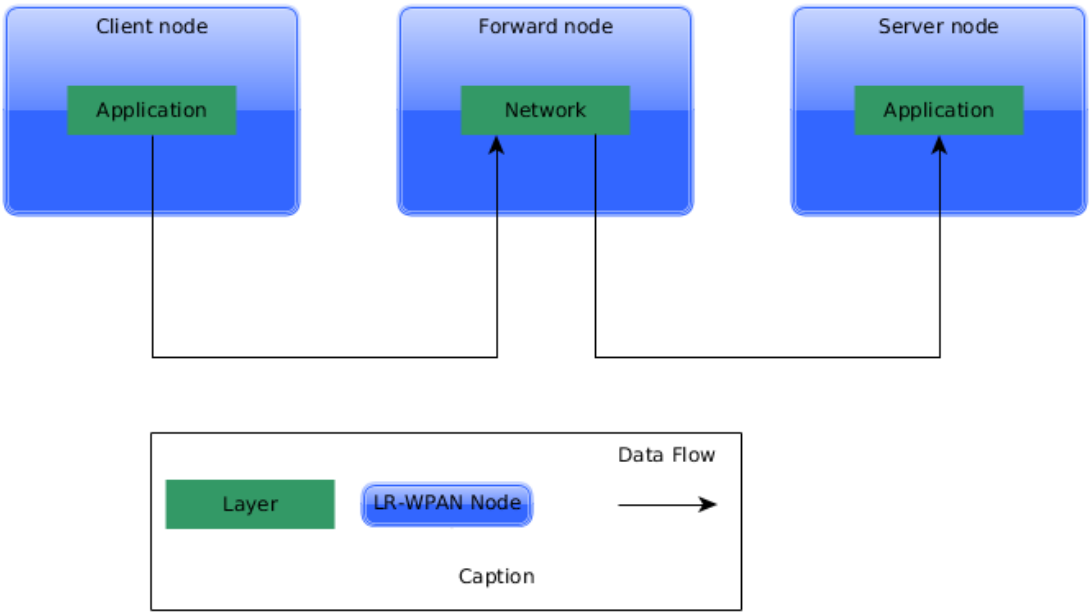

Figure 3.1: Topology

The timing tools built in the Contiki OS were tested in order to assess their precision levels. Since the Tmote Sky has an 8 MHz processor it should allow 125 nanosecond readings at best. In the tests conducted the most precise timer already comprised in the Contiki OS, the "rtimer",

oscillated every 33 μsecond, approximately. Therefore since the desired precision is on the millisecond level an error of $\frac{33\mu s}{1000\mu s} = 0.033 = 3.3\%$ is expected, which was considered acceptable for the intended purposes.

One of the considered options was to measure the processing delays of each individual layer. This was achieved by using same methodology as in [25] that used timers that measure delays between labels inserted into parts of the code where the data goes through, as presented in Figure 3.2. The Figure displays also the file and the function within file were each label was added. The timers start and stop in the depicted layers and, e.g. in the client the timer stops when the frame is inserted into the Carrier Sense Multiple Access (CSMA) queue, which enables to account only the processing delay (and not other delays such as queue and transmission delays).

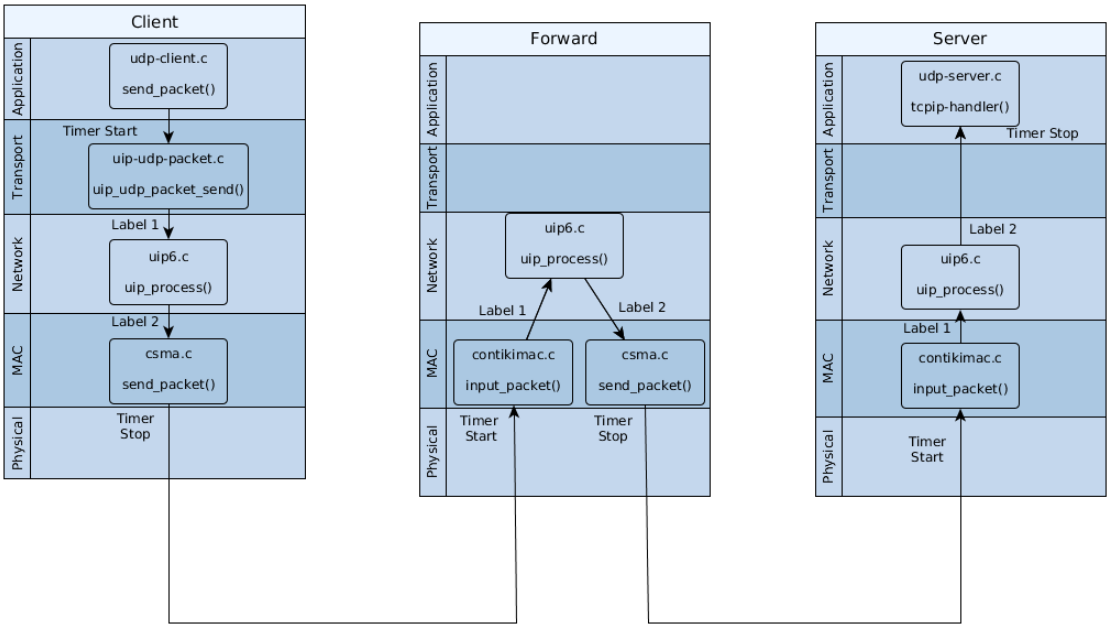

Figure 3.2: Contiki network stack scope

Although this was the considered approach, later efforts to simplify this layer by layer delay collection, led to account multiple layers processing in a single timer, grouping all processing delays. Therefore, the labels were disregarded and only the total time from timer start to timer stop was taken into account.

3.2 Implemented Testbed

The created testbed was similar to the one used in the simulation work. It was composed of three sensors: a client, a forwarder and a server node. The client sends a packet with a variable size to

an intermediary sensor. This node was used to obtain the delay a packet requires when it follows down the Contiki stack. The forward node preforms the routing of a packet from the client to the server, and it was used to obtain the delays a packet spends when it enters a node up until the third layer, and then leaves the node. Finally, the server node receives the packet from the forwarding node, allowing the collection of the processing delay when a packet follows the path from layer 1 to the top layer.

In order to collect the real delays, the hardware nodes were connected to a computer that received the processing delays as they were obtained by the sensors, as presented in Figure 3.3. Since Tmote Sky's range being of about 50 meters and, at normal power transmission the forwarding node was being bypassed and the client packets were sent directly to the server, this experiment to be conducted in a small area and the client nodes' transmission power was reduced to a minimum.

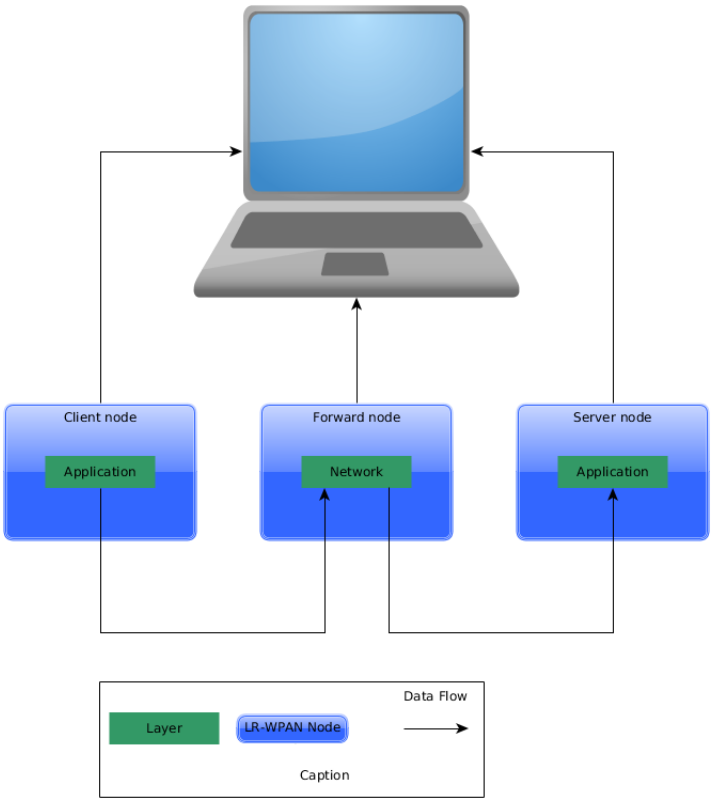

Figure 3.3: Live Testbed Topology

3.3 Simulation versus Testbed Results

The Cooja simulation tool has an in-built emulation model for the sky platform, and this model in theory should produce extremely precise tests that would output values equal to the live tests. It

was prepared a set of experiments in Cooja and in the real tests using all four protocol combinations, and using five different payloads for each.

Figure 3.4 presents the results in the client node for the real test and the Cooja emulation. The Figure 3.5 compares the server results for the same protocol combination. These figures assume the use of TCP since it was the scenario with the highest differences between real and simulated scenarios. The difference between the average values obtained was always below 33 μs, meaning that the difference was always under the clock precision and was therefore considered irrelevant.

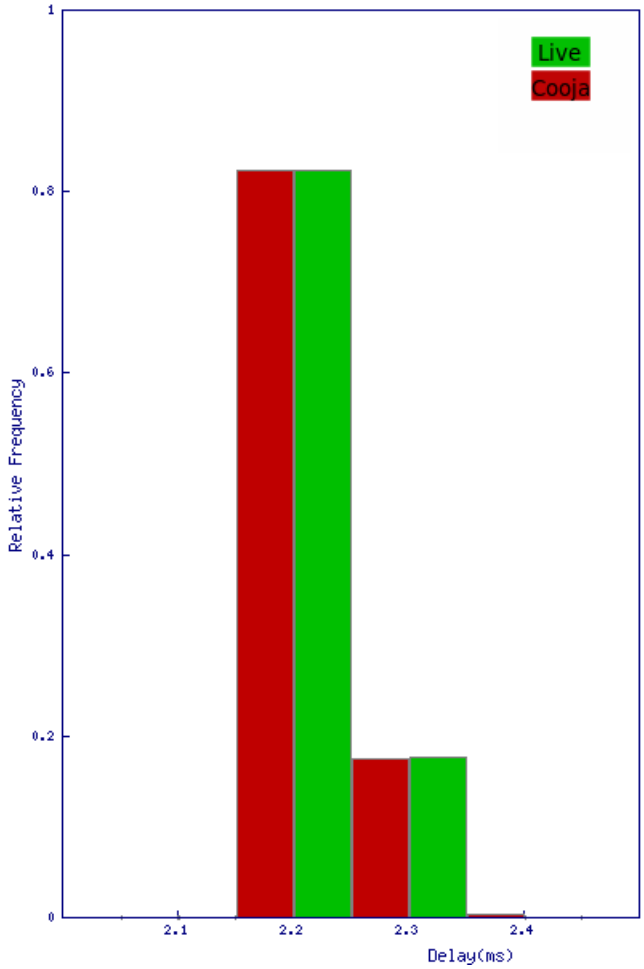

Figure 3.4: TCP with 6LowPAN compression 25 Bytes payload client

The tests using the real mote eproved to be very time consuming and Cooja has a very high level of precision. Similar work can be done in future work in order to check if this also occurs for other hardware platforms.

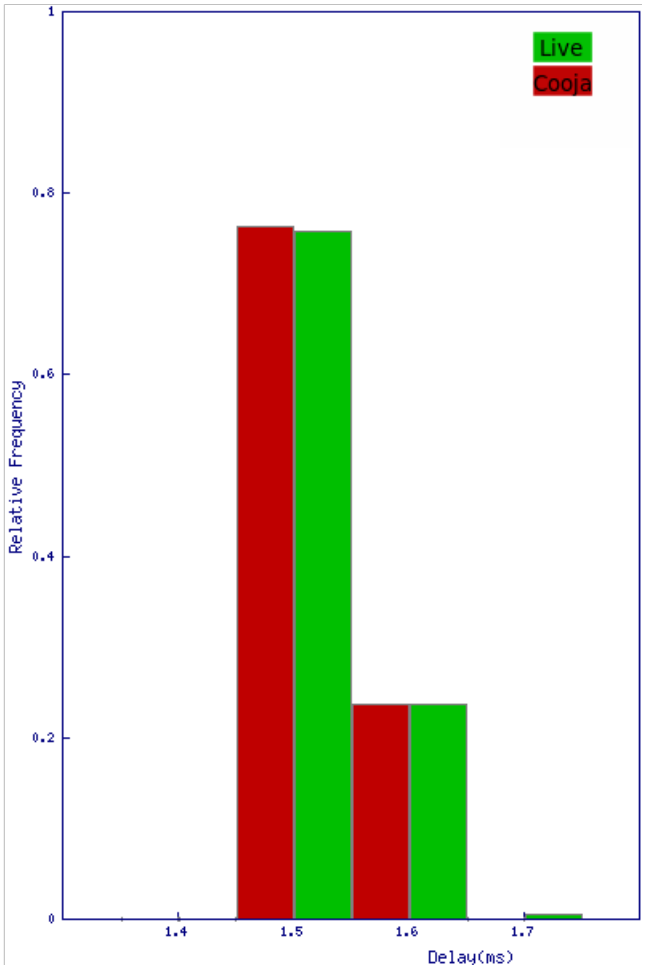

Figure 3.5: TCP with 6LowPAN compression 25 Bytes payload server

3.4 Collecting and Modeling Processing Delays

In order to collect the real processing delays using the real hardware nodes, the experiments were setup as follows: in each experiment around 700 packets were sent from client to server. The packet payload sizes were increased by 5 Bytes from the minimum size, which is zero, to the maximum size, which is variable according to the used protocols. The maximum sizes used in each experiment are presented in Table 3.1. This set up was repeated three times for each size. Further repetitions of these experiments were not necessary since all experiments presented the exact same results.

The collected delays revealed that multiple simulations with the same conditions would lead to the same values, which differs at maximum of 150 μs. Consequently the approximation by statistical model seemed unnecessary and thus, a simple average of the collected values was performed.

Protocol Combination	Maximum data payload size in Bytes
UDP with 6LowPAN header compression	71
UDP	47
TCP with 6LowPAN header compression	67
TCP	43

Table 3.1: Payload values

Figures 3.6, 3.7 and 3.8 present the results obtained when using UDP protocol with 6LowPAN header compression enabled. Each point represents the average processing delay obtained for transmission, showing the processing delay and the correspondent payload used during the test. After careful analysis of these results, it can be assumed that they can be modelled with a simple linear regression. Thus, the plotted function in those graphs represents the associated linear regression using the Perl function Regression from its statistics library. This function in turn uses the statistical standard Gentleman's algorithm [46] for linear regression which is as follows:

$$Slope = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

$$Y - intercept = \frac{(\sum y)(\sum x^2) - (\sum x)(\sum xy)}{n(\sum x^2) - (\sum x)^2}$$

All of the data was modeled with this algorithm which created a satisfactory approximation.

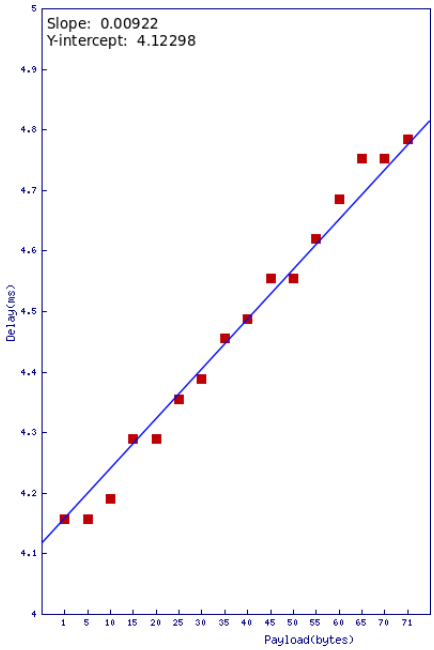

Figure 3.6: UDP Client with 6LowPAN compression

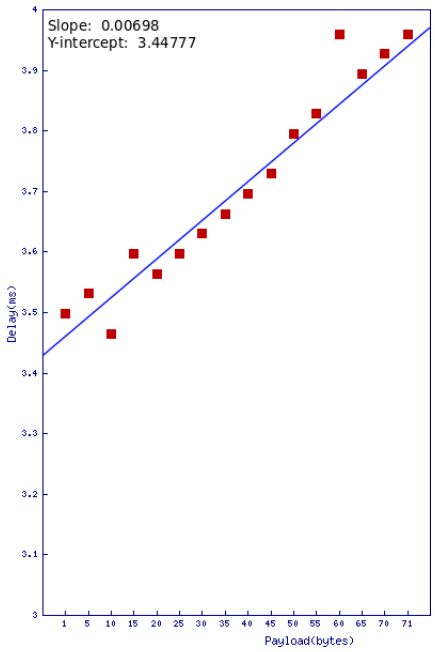

Figure 3.7: UDP Forward with 6LowPAN compression

In Table 3.2 the slope and Y-intercept values for all the experiments are presented.

Protocol Combination	Node type	Slope	Y-intercept
UDP with 6LowPAN header compression	Client	0.00922	4.12298
	Forward	0.00698	3.44777
	Server	0.00529	1.41624
UDP	Client	0.01040	4.07067
	Forward	0.00736	2.99739
	Server	0.00401	1.27440
TCP with 6LowPAN header compression	Client	0.00922	1.98783
	Server	0.00593	1.19984
TCP	Client	0.00963	1.19389
	Server	0.00366	1.45453

Table 3.2: Results

3.5 Discussion

Since the processing delay is obtained by running a fixed code, there is no variable when it comes to these delays. Therefore, processing delays obtained in each individual experiment had little fluctuation, at most of about 100 μseconds, which when dealing with values in the millisecond order is not very relevant. The results showed that the packet size was the main factor when it

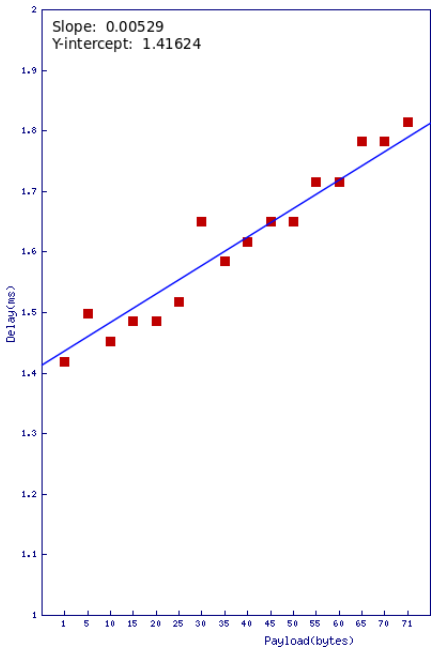

Figure 3.8: UDP Server with 6LowPAN compression

comes to processing delays. The growth of the size was directly proportional to the processing delay growth. Thus, these delays were modelled by a simple linear regression varying with the packet size. These regressions provided the values that are later inserted into the ns-3 module.

It was concluded by data analysis that receiving a packet takes a considerably less amount of time than sending one. This is expected since the actions needed to create a packet and a frame are considerably more time exhausting. Also, as expected, packets sent with 6LowPAN header compression generated a higher constant processing delay, although in most cases the increase was not substantial, going as low as adding only 50 μseconds which was an increase of about 1%. At most of 400 μseconds which is an addition of about 27%.

The TCP processing delay was considerably lower than the UDP processing delay, since the architecture used in TCP scenario does not include a forwarder node. Also, the fact that the RPL protocol is active in the UDP test imposes larger processing delays, when compared with the TCP scenario where RPL was not enabled.

The collected data presented in Table 3.2 indicates that processing times can be as high as almost 5 milliseconds, and never lower than 1 millisecond, which in networks based on the 802.15.4 IEEE Standard may be highly relevant. Since ns-3 does not take into account any processing delays, this leads to relevant delay inaccuracies when simulating these networks.

Chapter 4

Ns-3 Developments

This chapter details the methodology used to implement the modelled processing delays in the ns-3 simulator. At first the global approach is explained and then, the details of the final implementation and its results are presented.

4.1 Ns-3 Integration Alternatives

In order to implement the modelled processing delays in ns-3 two approaches could be followed: one being the creation of a ns-3 module from scratch and the other being the adaptation of an existing ns-3 module.

The option was to reduce changes to the ns-3 existing files to the minimum required, in order to preserve its structure and integrity, and at the same time follow the procedures used by the ns-3 developers when creating or changing its source files. Therefore, a standalone module was the chosen approach; it was a less invasive, despite having the downside of being more time consuming, since it required more than simply adding a new class to an existing model.

After reviewing in detail the ns-3 build it was clear that there was no chance to change events after their creation and, therefore, two possible solutions to include the delay in the stack were studied: either changing the scheduled events before their creation, or creating a new event that introduces delays while the packet follows through the stack. The second solution was selected mainly because joining, for instance, the processing delays with the queueing delays would end up leaving the solution with a lack of capacity to module the processing delays for each layer. This happens because the queueing delays are all scheduled only once and the processing delays would be limited the same way and, even though individual layer processing delays are not yet modeled, the module was created with the intent of being expanded later. And thus restrictive solutions like this one were discarded in order to make the end result as adaptive as possible.

As explained above, the implemented module simulates the processing delays as only one time for all the layers, thus, the actual implementation does not simulate delays of individual layers. Therefore there was no need to alter more than one layer. Since the forwarding processing delay is accounted in two of the modeled scenarios only the adaptation of either the second or

the third layer made sense, since data that is going to be forwarded does not go higher than the third layer and the first layer delays are not accounted for. It was opted to alter the network layer because research revealed three different module implementations of layer 3 and nineteen of layer 2, therefore the layer which required less code alteration was selected.

The number of module implementations for the data link layer can be seen as children of the ns3::NetDevice class in Figure4.1.

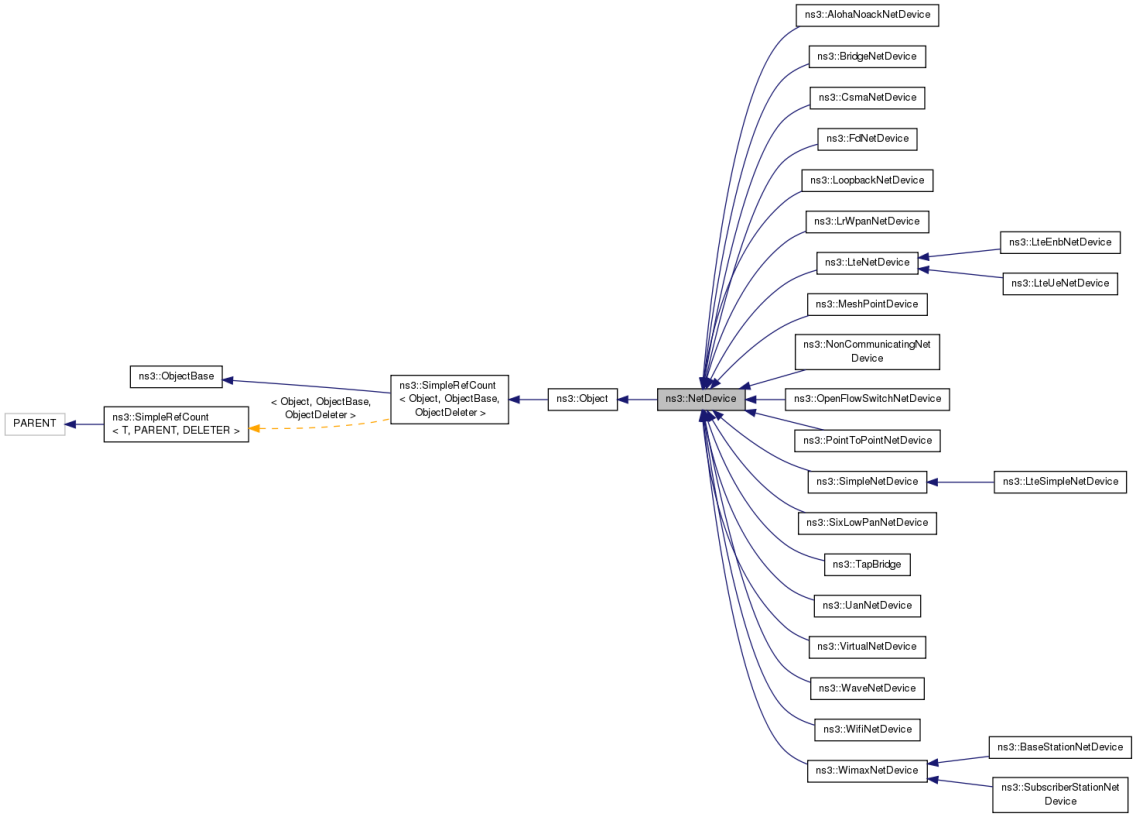

Figure 4.1: ns3::NetDevice Inheritance Diagram [1]

As presented in Figure 4.2 the IPv6 layer 3 protocol is defined by only one class, which can be assessed due to the fact that the IPv6 class is parent only to it. Therefore this class controls all of the IPv6 packets when they go through this layer of the stack.

As it can be seen in Figure 4.3 the IPv4 class is the parent of two different classes. The Ipv4L3Protocol which was the starting point for the Ipv6L3Protocol, and much like that class it handles almost all of the packets that reach the network layer going up or down the stack. However it does not handle packets when the Ipv4L3ClickProtocol is installed. This class is created to be used only by click nodes which are created to behave like configurable routers [47]. This class was disregarded since after a brief analysis it did not seem relevant to be used in conjunction

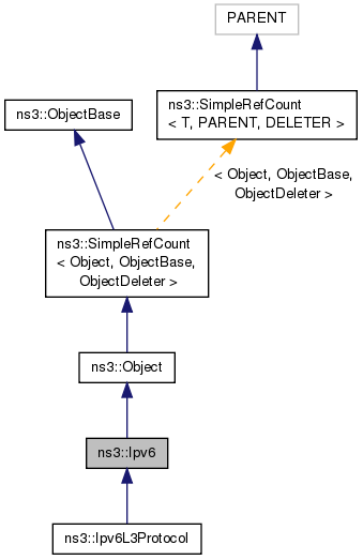

Figure 4.2: ns3::IPv6 Inheritance Diagram [1]

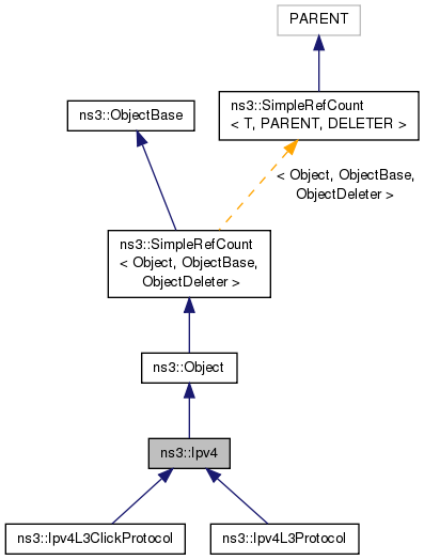

Figure 4.3: ns3::IPv4 Inheritance Diagram [1]

with the created work. However, since it is an adaptation that shares much of its code with the Ipv4L3Protocol class, porting the created class seems to be a fairly fast procedure.

4.2 Implementation

The final implementation is composed of a class A.1, a helper A.3 and their respective libraries A.4 A.2, which were made from scratch. Also, the following two classes were changed:

- Ipv4L3Protocol
- Ipv6L3Protocol

Since these classes' structure is fairly similar, only the Ipv6L3Protocol changes are presented, the alterations done to the other class followed similar changes. Since processing delay times are obtained for three different packet handling situations: when the packet is being received, sent or routed to another node, three functions from the class were modified. The Receive function for a node receiving a packet. The Sent function for a node sending a packet. The IpForward function for a node routing a packet.

In order to create the delays an event was created, using the Schedule function, from the Scheduler module. This function creates a future event in a given time that executes another function which is also given. This method was used in all three situations when the function is about to call the next function. Using this procedure, the function execution is halted for the desired amount of time and therefore, the processing simulation delay is inserted into the stack navigation. This allows the insertion of the desired delays with limited code alteration of the classes at hand. A flowchart of the algorithm is presented in Figure 4.4.

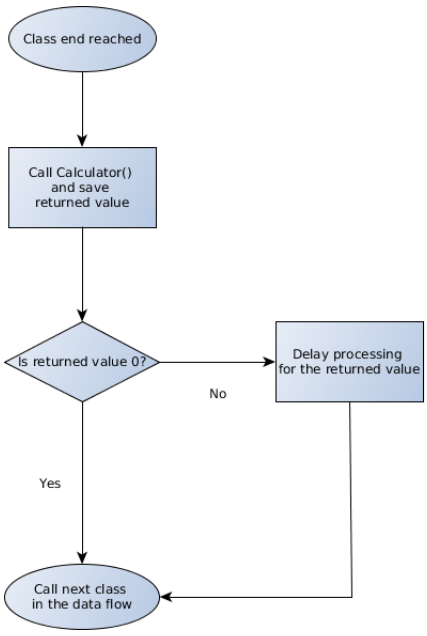

Figure 4.4: Flowchart of the used algorithm

4.2.1 Helper

In ns-3, the nodes are instances of class ns3::Node and the user can aggregate objects to it. This objects define the various protocols, applications and network devices that create each nodes' stack. As a way of simplifying the aggregations of these different objects, ns-3 uses helpers in order to ease the users work and thus, an helper was created for the conceived module. This helper installs the processing delay accountability into each node and makes it so that all communication of this node account these delays.

The created helper is composed of the following functions:

ProcessingHelper(): The Constructor of the class which is executed whenever we create a new object of this class;

SetAttribute(): This function assigns a provided value to an attribute of the ns3::Processing class. It is used to provide the type of hardware or software systems used in the simulation;

Install(): Installs the processing delay accountability in a provided node;

InstallAll(): Provides the same functionality of the previous function but for all existing nodes in the simulation.

4.2.2 Module

The created module was named Processing and it is composed of the following functions:

Processing(): The Constructor of the module which is executed whenever we create a new object of that class;

~Processing(): The Destructor of the module which is executed whenever an object of this class goes out of scope or whenever the delete expression is applied to a pointer to one of the objects;

GetTtpeId(): This is the function which defines the type ID of the class and returns said ID when called;

Calculator(): This function is called in order to obtain the processing delays. For it to be called properly it is required to fill the following parameters:

- Node type parameter which describes how the packet is going through the stack, it can be received, sent or forwarded.
- The fourth layer protocol parameter which describes the protocol used in the transport layer, either TCP or UDP.
- The third layer protocol parameter which describes the protocol used in this layer, either IPv6 and IPv4.

- The NetDevice type parameter which is an object associated with the node that is handling the packet. The Netdevice allows to check if SixLowPAN compression is active.
- And the size of the packet received, since it is the main parameter to calculate the delay.

The private definitions are the m_hardware variable which receives the type of sensor being used in the simulation, where current work assumed only the Tmote Sky as active since it was the only sensor used. The m_software variable which receives the type of operating system being used, as only Contiki OS was activated.

Using the Calculator function parameters plus the hardware and software types that were provided by the helper, it is possible to simulate the processing delay that the packet expends while following the stack.

Figure 4.5 provides a collaboration diagram for the created class, a fairly independent and highly adaptable module was obtained. This works in hand with the desired goal of adapting the procedure to be used in all layers of the ns-3 stack, and provide processing delays for each individual layer.

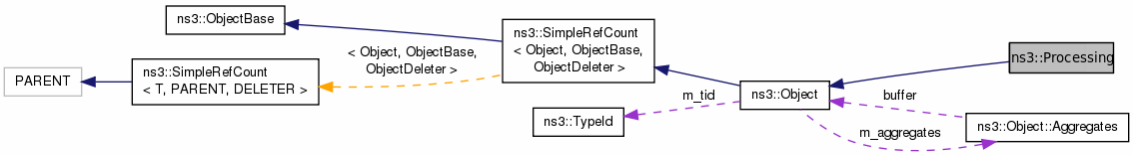

Figure 4.5: ns3::Processing Collaboration Diagram

4.3 Results

In order to assess the results of the created module, a ns-3 simulation was created by changing a ns-3 pre-existing example. This simulation consists of a LR-WPAN with two nodes, a client and a server. The simulated scheme used is UDP with the 6LowPAN header compression activated. The source code to this simulation can be found in A.5.

Two tests were performed: the first one where the created module was OFF in all the nodes; and the second one where the created module was ON. The results of these tests are presented in Figure 4.6 where each point represents the obtained average packet end-to-end delay when running this simulation, exchanging 700 packets ten times for each payload.

From the results obtained it is noticeable that, when the module is ON, the total delay increases, indicating that processing delays are being accounted. Also, these results show the importance of

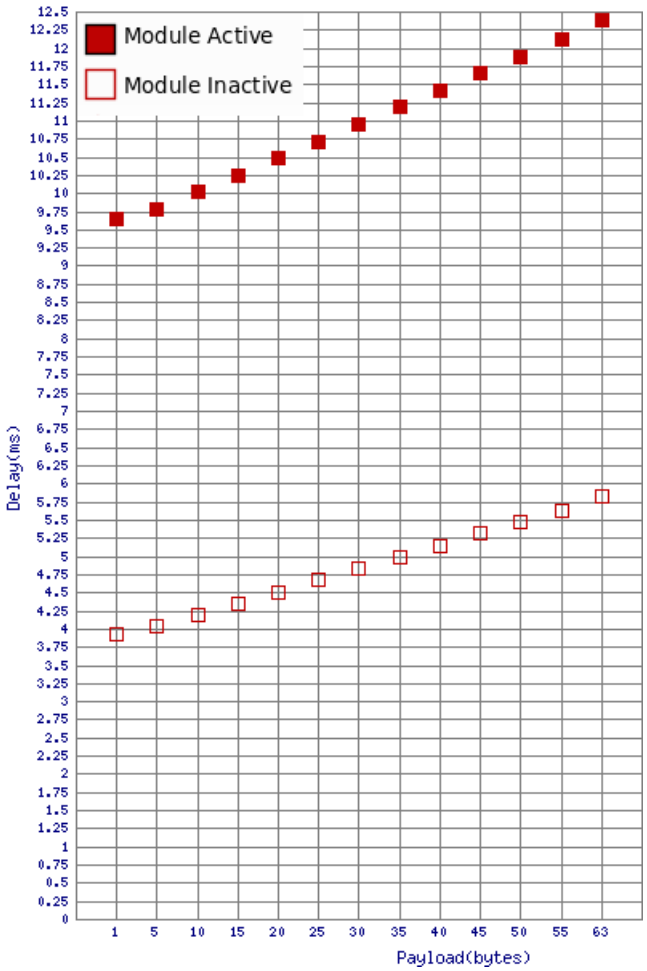

Figure 4.6: Testing Simulation Result

accounting processing delays, since these processing delays represent more than 100% of initial end-to-end delay.

In Figure 4.7 it is presented the difference between the delays seen in Figure 4.6 and also a line which is a simple linear regression of the results obtained by using the testbed. The results of both the modeled values and the processing delays obtained in the ns-3 are similar.

4.4 Discussion

The proposed ns-3 implementation consists of a new module complete with helper, that was placed inside the /src/internet folder. The helper is used in accordance with the ns-3 policy and can be used to quickly install into a node the processing delay accountability provided that the user defines the hardware and software systems being used. The class detects if the node has been installed with the processing helper and if so, automatically detects the protocols that are being used in the

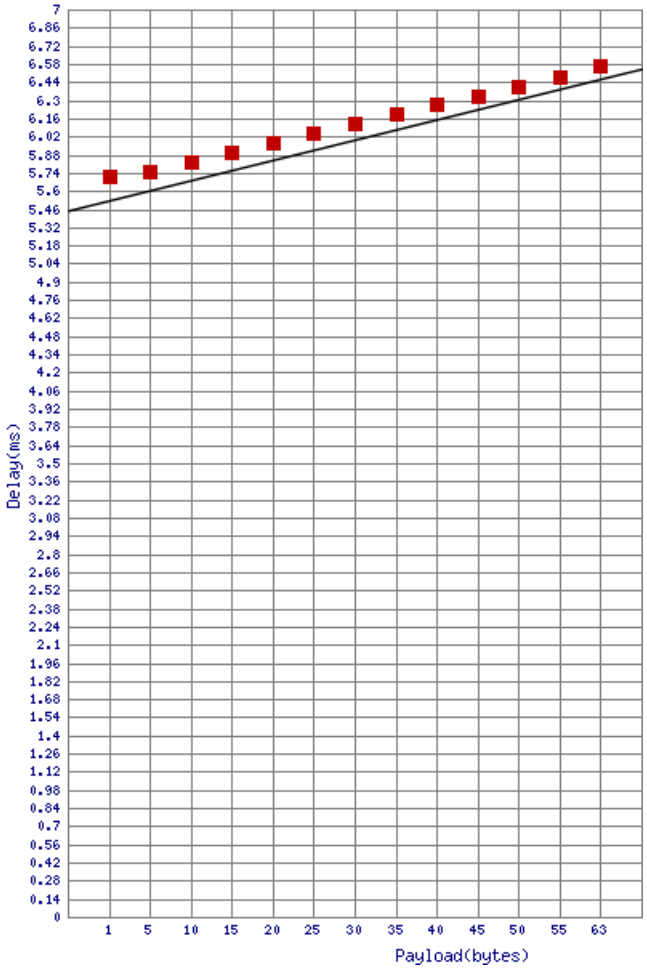

Figure 4.7: Comparison between modeled delay and ns-3 delay

program and adds the delays accordingly, only in the case those delays are modeled and activated in the class. If they are not modeled the program will run normally as if the Processing class was not installed at all. The files created were tested by running a simulation that recreates the exact same conditions in which the delays were obtained.

Chapter 5

Conclusions and Future Work

This chapter reviews the work done in order to collect, model and include processing delays in ns-3 simulations. The conclusions of these accomplishments are also presented and possible future work is depicted.

5.1 Work Review

In the first part of this dissertation, the background technologies and protocols used are detailed and a survey of the related work is also presented.

A study of the Contiki network stack was conducted in order to determine the logical placement of time labels that would allow the processing delay calculation. Several testbed schemes were implemented with the Contiki operating system and combined with the Tmote Sky sensor node while using TCP communication and also UDP communication, both using the real sensors and using the Cooja simulation tool. An extensive number of simulations were run using a chosen testbed with the purpose of collecting data to create a valid delay modeling. Perl scripts were created in order to gather the collected data and modeled it with the creation of histograms and graphs to allow the data analysis. This work in simulation and testbed environments created a processing delay measuring set up that can easily allow the implementation of the same testbed for other Contiki supported hardware systems. Also, the accuracy of the Cooja simulator was tested when it comes to the Sky processing unit and its emulation values were proved to be very accurate for the ran simulations.

An evaluation and analysis of the resulting data allowed to define the main altering factors when it comes to processing delay times in the same hardware system. The processing delays were modeled for TCP and UDP communication with both 6LowPAN activated and deactivated, TCP without forward.

Concerning the implementation of the modeled delays in ns-3, multiple studies were conducted in order to follow an optimal approach to include the processing delays accountability

feature in this network simulator. A ns-3 module that allows the accountability of the processing delays of LR-WPAN sensor nodes was created, along with a necessary helper. Therefore, all submitted changes to ns-3 were made in coherence with the simulator developing architecture.

5.2 Future Work

As it stands the ns-3 module is perfectly functional when it comes to UDP with or without 6LowPAN and is semi functional for TCP, which lacks the modulation for the forwarding node.

As future work TCP modeling can be addressed, in order to fully model the TCP delays in the stack. This would complete all the processing delays that can be modeled while using the hardware/software pair utilized in this dissertation.

Also, as future work, other hardware and software combinations can be tested in order to provide ns-3 users the choice of other hardware solutions beyond the one used in the current work.

The adaptation of the ns-3 module can also be addressed in future work in order to allow modulation of the processing times for individual layers, which can extend later the use of this module in order to include other stack combinations easily.

As future work, the ns-3 module created can be submitted as a new module to be release in the main ns-3.

Appendix A

Ns-3 Source Code

This appendix comprises the source code used in the creation of the ns-3 module.

A.1 processing.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2017 FEUP
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Author: Guilherme Ferreira
 */

#include <fstream>
#include "processing.h"

#ifndef CONTIKI
```

```
#define CONTIKI    0
#endif

#ifndef SKY
#define SKY     0
#endif
int n=0;

namespace ns3
{
NS_LOG_COMPONENT_DEFINE ("Processing");

NS_OBJECT_ENSURE_REGISTERED (Processing);

TypeId Processing::GetTypeId ()
  {
    static TypeId tid = TypeId ("ns3::Processing")
    .SetParent<Object> ()
    .SetGroupName ("Internet")
    .AddConstructor<Processing> ()
    .AddAttribute ("Hardware",
                   "The hardware type to use in this simulation",
                   UintegerValue (0),
                   MakeUintegerAccessor (&Processing::m_hardware),
                   MakeUintegerChecker<uint8_t> ())
    .AddAttribute ("Software", "The software type to use in this simulation",
                   UintegerValue (0),
                   MakeUintegerAccessor (&Processing::m_software),
                   MakeUintegerChecker<uint8_t> ())
  ;
  return tid;
  }

Processing::Processing ()
  {
  }

Processing::~Processing ()
  {
  }
```

```
float Processing :: Calculator ( uint16_t Node,uint16_t L4, uint16_t L3, std :: string
    Compress,uint16_t size )
 {
  std :: string type;
  if (m_hardware==CONTIKI)
    {
      type+= "Contiki ";
     }


  if (m_software==SKY)
    {
      type+= "Sky ";
     }



  if (Node==1)
     {
       type+= "Client ";
     }
  if (Node==2)
    {
       type+= "Forward ";
     }
  if (Node==3)
    {
       type+= "Server ";
     }



  if (L3==1)
    {
       type+= "Ipv6 ";
     }
  if (L3==2)
     {
       type+= "Ipv4 ";
```

```
    }


if (L4==17)
  {n++;
      type+= "UDP ";
      size=size-8;
    }
if (L4==6)
  {
      type+= "TCP ";
      size=size-20;
    }


if (!Compress.compare("ns3::SixLowPanNetDevice"))
  {
      type+= "Sixlowpan";
    }


if ( size==0)
  {
    return 0;
  }

  // std :: cout << type <<"      "<<Simulator::Now()<<" count "<<n<<" "<<size <<"
       client    "<<4.12298+size*0.00922<<"server"<< 1.41624+size*0.00529<<std::endl;
if (!type.compare("Contiki Sky Client Ipv6 UDP Sixlowpan"))
  {
    return (4.12298+(size*0.00922))*0.001;
    }
 if (!type.compare("Contiki Sky Server Ipv6 UDP Sixlowpan"))
  {
    return (1.41624+(size*0.00529))*0.001;
    }
 if (!type.compare("Contiki Sky Forward Ipv6 UDP Sixlowpan"))
  {
    return 3.44777+(size*0.00698)*0.001;
    }
```

```
        if (!type.compare("Contiki Sky Client Ipv6 UDP "))
       {
         return 0.01040+(size*4.07067)*0.001;
        }
      if (!type.compare("Contiki Sky Server Ipv6 UDP "))
       {
         return 1.27440+(size*0.00401)*0.001;
        }
      if (!type.compare("Contiki Sky Forward Ipv6 UDP "))
       {
         return 2.99739+(size*0.00736)*0.001;
        }
            if (!type.compare("Contiki Sky Client Ipv6 TCP Sixlowpan"))
       {
         return 1.98783+(size*0.0922)*0.001;
        }
      if (!type.compare("Contiki Sky Server Ipv6 TCP Sixlowpan"))
       {
         return 1.19984+(size*0.00593)*0.001;
        }
      if (!type.compare("Contiki Sky Client Ipv6 TCP "))
       {
         return 1.19389+(size*0.00963)*0.001;
        }
      if (!type.compare("Contiki Sky Server Ipv6 TCP "))
       {
         return 1.45453+(size*0.00366)*0.001;
        }

      return 0;
   }
}
```

A.2 processing.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2017 FEUP
```

```
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Author: Guilherme Ferreira
 */

#ifndef PROCESSING_H
#define PROCESSING_H

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/sixlowpan-module.h"


#ifndef UIPV6_UDP_CLIENT
#define UIPV6_UDP_CLIENT 0
#endif

namespace ns3{

class Processing : public Object
{
  public:

  /**
   * \brief Constructor.
   */
  Processing();
```

```
  /**
   * \brief Destructor.
   */
  virtual ~Processing();

   /**
   * \brief Get the type ID of this class.
   * \return type ID
   */

  static TypeId GetTypeId();


  float Calculator(uint16_t Node,uint16_t L4, uint16_t L3, std::string Compress,
      uint16_t size);

  private:
  /**
   * \brief
   */
    uint8_t m_hardware;

  /**
   * \brief
   */
    uint8_t m_software;

};

}
#endif
```

A.3 processing-helper.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2017 FEUP
 *
```

```
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Author: Guilherme Ferreira
 */


#include "processing-helper.h"

namespace ns3
{

NS_LOG_COMPONENT_DEFINE ("ProcessingHelper");

ProcessingHelper::ProcessingHelper ()
{
  m_ProcessingFactory.SetTypeId ("ns3::Processing");
}

void
ProcessingHelper::SetAttribute (std::string n1, const AttributeValue &v1)
{
  m_ProcessingFactory.Set (n1, v1);
}

void
ProcessingHelper::Install (Ptr<Node> node)
{
  Ptr<Object> process = m_ProcessingFactory.Create <Object> ();
  node->AggregateObject (process);
```

```
}


void
ProcessingHelper :: Install (NodeContainer nodes)
{
  for (NodeContainer:: Iterator i = nodes.Begin (); i != nodes.End (); ++i)
    {
      Ptr<Node> node = *i;
      if (node->GetObject<Ipv4L3Protocol> () || node->GetObject<Ipv6L3Protocol> ())
        {
          Install (node);
        }
    }
}

void
ProcessingHelper :: InstallAll ()
{
  for (NodeList:: Iterator i = NodeList::Begin (); i != NodeList::End (); ++i)
    {
      Ptr<Node> node = *i;
      if (node->GetObject<Ipv4L3Protocol> () || node->GetObject<Ipv6L3Protocol> ())
        {
          Install (node);
        }
    }
}


} /* namespace ns3 */
```

A.4 processing-helper.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2017 FEUP
 *
 * This program is free software; you can redistribute it and/or modify
```

```
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Author: Guilherme Ferreira
 */

#ifndef PROCESSING_HELPER_H
#define PROCESSING_HELPER_H

#include <string>
#include <vector>

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/sixlowpan-module.h"

namespace ns3 {

class ProcessingHelper
{
public:

  ProcessingHelper();
  virtual ~ProcessingHelper() {}

  void SetAttribute (std::string n1, const AttributeValue &v1);

  void CreateAndAggregateObjectFromTypeId (Ptr<Node> node);

  void Install (Ptr<Node> node);
```

```
  void Install (const NodeContainer nodes);

  void InstallAll ();

private:

  ObjectFactory m_ProcessingFactory;

};

} /* namespace ns3 */

#endif
```

A.5 6lowpan-test.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
 /*
 * Copyright (c) 2013 Universita' di Firenze, Italy
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Author: Tommaso Pecorella <tommaso.pecorella@unifi.it>
 */

#include <fstream>
#include "ns3/netanim-module.h"
```

```
#include "ns3/core-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/csma-module.h"
#include "ns3/internet-apps-module.h"
#include "ns3/ipv6-static-routing-helper.h"
#include "ns3/ipv6-routing-table-entry.h"
#include "ns3/sixlowpan-module.h"
#include "ns3/lr-wpan-module.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/mobility-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("ExampleSixlowpan");

int main (int argc, char** argv)
{
 bool verbose = false;
 Address serverAddress;
 // LogComponentEnable ("SixLowPanNetDevice", LOG_LEVEL_ALL);
 CommandLine cmd;
 cmd.AddValue ("verbose", "turn on some relevant log components", verbose);
 cmd.Parse (argc, argv);

 Packet::EnablePrinting ();
 Packet::EnableChecking ();

 NS_LOG_INFO ("Create nodes.");
 NodeContainer n;
 n.Create (3);

 MobilityHelper mobility;
 mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                "MinX", DoubleValue (0.0),
                                "MinY", DoubleValue (0.0),
                                "DeltaX", DoubleValue (10),
                                "DeltaY", DoubleValue (10),
                                "GridWidth", UintegerValue (3),
                                "LayoutType", StringValue ("RowFirst"));
```

```
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobility.Install(n);

NS_LOG_INFO("Create IPv6 Internet Stack");
InternetStackHelper internetv6;
internetv6.Install(n);

NS_LOG_INFO("Create channels.");


LrWpanHelper lrWpanHelper;
NetDeviceContainer lr = lrWpanHelper.Install(n);
lrWpanHelper.AssociateToPan(lr, 0);

SixLowPanHelper sixlowpan;
NetDeviceContainer six = sixlowpan.Install(lr);

Ipv6AddressHelper ipv6;
ipv6.SetBase("2001:1::", Ipv6Prefix(64));
Ipv6InterfaceContainer i = ipv6.Assign(six);
serverAddress = Address(i.GetAddress(0,1));


i.SetForwarding(1, true);
i.SetDefaultRouteInAllNodes(1);
/*
ProcessingHelper process;
process.SetAttribute("Hardware", UintegerValue(0));
process.SetAttribute("Software", UintegerValue(0));
process.Install(n);*/


uint16_t port = 4000;  // well-known echo port number
UdpServerHelper server(port);
ApplicationContainer apps = server.Install(n.Get(0));
apps.Start(Seconds(1.0));
apps.Stop(Seconds(1050.0));

uint32_t packetSize =12+1;//12 is the header size so payload is the value added to
    12
```

```
uint32_t maxPacketCount = 700;
Time interPacketInterval = Seconds (1);
UdpClientHelper client (serverAddress, port);
client . SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));
client . SetAttribute (" Interval ", TimeValue ( interPacketInterval ));
client . SetAttribute ("PacketSize", UintegerValue (packetSize));
apps = client . Install (n.Get (2));
apps. Start (Seconds (2.0));
apps.Stop (Seconds (1000.0)); //


AsciiTraceHelper ascii ;
lrWpanHelper.EnableAsciiAll ( ascii . CreateFileStream ("sixlowpan. tr "));
lrWpanHelper.EnablePcapAll (std :: string ("6lowpan"), true);


Simulator :: Stop (Seconds (10000));
NS_LOG_INFO ("Run Simulation.");
FlowMonitorHelper flowmon;
Ptr<FlowMonitor> monitor = flowmon. InstallAll ();
monitor = flowmon.GetMonitor();

Simulator :: Run ();
monitor->CheckForLostPackets ();
Ptr< Ipv6FlowClassifier > classifier = DynamicCast<Ipv6FlowClassifier> (flowmon.
    GetClassifier ());
std :: map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();

uint32_t txPacketsum = 0;
uint32_t rxPacketsum = 0;
uint32_t DropPacketsum = 0;
uint32_t LostPacketsum = 0;
double Delaysum = 0;

for (std :: map<FlowId, FlowMonitor::FlowStats>:: const_iterator iter = stats .begin ();
     iter != stats .end (); ++iter)
   {
     txPacketsum += iter ->second.txPackets;
     rxPacketsum += iter ->second.rxPackets;
     LostPacketsum = txPacketsum-rxPacketsum;
```

```
        DropPacketsum += iter->second.packetsDropped.size();
        Delaysum += iter->second.delaySum.GetSeconds();
       }

  std::cout << "  All Tx Packets: " << txPacketsum << "\n";
  std::cout << "  All Rx Packets: " << rxPacketsum << "\n";
  std::cout << "  All Delay Per Packet: " << Delaysum / txPacketsum << "\n";
  std::cout << "  All Total Delay : " << Delaysum << "\n";
  std::cout << "  All Lost Packets: " << LostPacketsum << "\n";
  std::cout << "  All Drop Packets: " << DropPacketsum << "\n";
  std::cout << "  Packets Delivery Ratio: " << ((rxPacketsum * 100) / txPacketsum) <<
      "%" << "\n";
  std::cout << "  Packets Lost Ratio: " << ((LostPacketsum * 100) / txPacketsum) << "
      %" << "\n";
  monitor->SerializeToXmlFile(" serialize -stat.flowmon", true, true);

  Simulator::Destroy ();
  NS_LOG_INFO ("Done.");
}
```

References

[1] ns-3: ns-3 documentation [online]. URL: https://www.nsnam.org/doxygen/.

[2] Contiki: The open source operating system for the internet of things [online]. URL: http://www.contiki-os.org/.

[3] IEEE SA - 802.15.4-2011 - IEEE standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (LR-WPANs) [online]. URL: https://standards.ieee.org/findstds/standard/802.15.4-2011.html.

[4] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In Local computer networks, proceedings 2006 31st IEEE conference on, pages 641–648. IEEE, 2006. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4116633.

[5] ns-3 [online]. URL: https://www.nsnam.org/.

[6] Kévin Roussel, Ye-Qiong Song, and Olivier Zendra. Using cooja for WSN simulations: Some new uses and limits. In Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks, EWSN '16, pages 319–324. Junction Publishing, 2016. URL: http://dl.acm.org/citation.cfm?id=2893711.2893790.

[7] Adam Dunkels. Full TCP/IP for 8-bit architectures. In Proceedings of the 1st International Conference on Mobile Systems, Applications and Services, MobiSys '03, pages 85–98. ACM. URL: http://doi.acm.org/10.1145/1066116.1066118, doi:10.1145/1066116.1066118.

[8] Adam Dunkels, Juan Alonso, and Thiemo Voigt. Making TCP/IP Viable for Wireless Sensor Networks.

[9] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O'Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, and Adam Dunkels. Making sensor networks IPv6 ready. In Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys '08, pages 421–422. ACM. URL: http://doi.acm.org/10.1145/1460412.1460483, doi:10.1145/1460412.1460483.

[10] RFC 4944 [online]. URL: https://tools.ietf.org/html/rfc4944.

[11] RFC 6282 [online]. URL: https://tools.ietf.org/html/rfc6282.

[12] Standards: ZigBee specification | zigbee alliance [online]. URL: http://www.zigbee.org/download/standards-zigbee-specification/.

[13] Firdaus, E. Nugroho, and A. Sahroni. ZigBee and wifi network interface on wireless sensor networks. In 2014 Makassar International Conference on Electrical Engineering and Informatics (MICEEI), pages 54–58, 2014. doi:10.1109/MICEEI.2014.7067310.

[14] RIOT - the friendly operating system for the internet of things [online]. URL: https://www.riot-os.org/.

[15] The LiteOS operating system [online]. URL: http://www.liteos.net/.

[16] FreeRTOS - market leading RTOS (real time operating system) for embedded systems with internet of things extensions [online]. URL: http://www.freertos.org/.

[17] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, LCN '04, pages 455–462. IEEE Computer Society, 2004. URL: http://dx.doi.org/10.1109/LCN.2004.38, doi:10.1109/LCN.2004.38.

[18] Tmote sky schematics microsoft word - tmote-sky-datasheet-102.doc sentilla corporation [online]. URL: https://fccid.io/document.php?id=613132.

[19] Adam Dunkels. Poster abstract: Rime — a lightweight layered communication stack for sensor networks. URL: https://www.researchgate.net/publication/242380890_Poster_Abstract_Rime_-_A_Lightweight_Layered_Communication_Stack_for_Sensor_Networks.

[20] Seed eye | evidence [online]. URL: http://www.evidence.eu.com/products/seed-eye.html.

[21] Wismote [online]. URL: http://wismote.org/doku.php.

[22] Z1 platform | zolertia [online]. URL: http://zolertia.io/product/hardware/z1-platform.

[23] Micaz [online]. URL: http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf.

[24] Msp430f15x, msp430f16x and msp430f161x microcontroller datasheet [online]. URL: http://www.ti.com/lit/gpn/msp430f1611.

[25] Pedro Filipe Cruz Pinto. Admission Control based on End-to-end Delay Estimation to Enhance the Support of Real-Time Traffic in Wireless Sensor Networks. PhD thesis, 2015. URL: http://repositorio-aberto.up.pt/handle/10216/80645.

[26] H. Kdouh, H. Farhat, G. Zaharia, C. Brousseau, G. Grunfelder, and G. El Zein. Performance analysis of a hierarchical shipboard wireless sensor network. pages 765–770. IEEE, 2012. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6362886, doi:10.1109/PIMRC.2012.6362886.

[27] A. Wang and Z. Liu. Analysis and utilizing of the error models in network education with NS3. In 2010 Second International Workshop on Education Technology and Computer Science (ETCS), volume 1, pages 15–18, 2010. doi:10.1109/ETCS.2010.64.

[28] A. R. A. Kumar, S. V. Rao, and D. Goswami. NS3 simulator for a study of data center networks. In 2013 IEEE 12th International Symposium on Parallel and Distributed Computing, pages 224–231, 2013. doi:10.1109/ISPDC.2013.37.

[29] D. Carvin, P. Owezarski, and P. Berthou. A generalized distributed consensus algorithm for monitoring and decision making in the IoT. In 2014 International Conference on Smart Communications in Network Technologies (SaCoNeT), pages 1–6, 2014. doi:10.1109/SaCoNeT.2014.6867769.

[30] N. Boufares, I. Khoufi, P. Minet, L. Saidane, and Y. Ben Saied. Three dimensional mobile wireless sensor networks redeployment based on virtual forces. In 2015 International Wireless Communications and Mobile Computing Conference (IWCMC), pages 563–568, 2015. doi:10.1109/IWCMC.2015.7289145.

[31] The network simulator - ns-2 [online]. URL: http://www.isi.edu/nsnam/ns/.

[32] OMNeT++ discrete event simulator - home [online]. URL: https://omnetpp.org/.

[33] Overview — SimPy 3.0.9 documentation [online]. URL: https://simpy.readthedocs.io/en/latest/.

[34] C. Fogelberg and V. Palade. GreenSim: A network simulator for comprehensively validating and evaluating new machine learning techniques for network structural inference. In 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, volume 2, pages 225–230, 2010. doi:10.1109/ICTAI.2010.105.

[35] Wan Du. Modeling and simulation of wireless sensor networks. phdthesis, 2011. URL: https://tel.archives-ouvertes.fr/tel-00690466/document.

[36] Ahcène Bounceur. CupCarbon: A new platform for designing and simulating smart-city and IoT wireless sensor networks (SCI-WSN). pages 1–1. ACM Press, 2016. URL: http://dl.acm.org/citation.cfm?doid=2896387.2900336, doi:10.1145/2896387.2900336.

[37] S. Kapralov and V. Dyankova. Modeling a system with discrete events. In 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation (EMS), pages 167–172, 2012. doi:10.1109/EMS.2012.50.

[38] C. Damiron and D. Krahl. A global approach for discrete rate simulation. In Proceedings of the Winter Simulation Conference 2014, pages 2966–2977, 2014. doi:10.1109/WSC.2014.7020136.

[39] Guillaume Kremer, Denis Carvin, Pascal Berthou, and Philippe Owezarski. Configuration schemes and assessment of NS3 models using a wireless testbed. working paper or preprint, 2013. URL: https://hal.archives-ouvertes.fr/hal-00817453.

[40] J. B. Ernst, S. C. Kremer, and J. J. P. C. Rodrigues. A wi-fi simulation model which supports channel scanning across multiple non-overlapping channels in NS3. In 2014 IEEE 28th International Conference on Advanced Information Networking and Applications, pages 268–275, 2014. doi:10.1109/AINA.2014.36.

[41] T. Meyer, D. Raumer, F. Wohlfart, B. E. Wolfinger, and G. Carle. Low latency packet processing in software routers. In International Symposium on Performance Evaluation

of Computer and Telecommunication Systems (SPECTS 2014), pages 556–563, 2014. doi:10.1109/SPECTS.2014.6879993.

[42] Nicola Baldo, Manuel Requena-Esteso, José Núñez-Martínez, Marc Portolès-Comeras, Jaume Nin-Guerrero, Paolo Dini, and Josep Mangues-Bafalluy. Validation of the IEEE 802.11 MAC model in the ns3 simulator using the EXTREME testbed. In Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools '10, pages 64:1–64:9. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010. URL: http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8705, doi:10.4108/ICST.SIMUTOOLS2010.8705.

[43] EXTREME testbed® [online]. URL: http://networks.cttc.es/mobile-networks/extreme_testbed/.

[44] Aminul Haque Chowdhury, Muhammad Ikram, Hyon-Soo Cha, Hassen Redwan, S. M. Saif Shams, Ki-Hyung Kim, and Seung-Wha Yoo. Route-over vs mesh-under routing in 6lowpan. In ResearchGate, pages 1208–1212. URL: https://www.researchgate.net/publication/220762351_Route-over_vs_mesh-under_routing_in_6LoWPAN, doi:10.1145/1582379.1582643.

[45] Sébastien Vincent, Julien Montavont, and Nicolas Montavont. Implementation of an IPv6 stack for NS-3. In Proceedings of the 3rd International Conference on Performance Evaluation Methodologies and Tools, ValueTools '08, pages 75:1–75:9. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). URL: http://dx.doi.org/10.4108/ICST.VALUETOOLS2008.4374, doi:10.4108/ICST.VALUETOOLS2008.4374.

[46] W. Morven Gentleman. Algorithm AS 75: Basic procedures for large, sparse or weighted linear least problems. 23(3):448–454. URL: http://www.jstor.org/stable/2347147, doi:10.2307/2347147.

[47] NS-3-click: Click modular router integration for NS-3. In ResearchGate. URL: https://www.researchgate.net/publication/262425159_NS-3-Click_Click_Modular_Router_Integration_for_NS-3, doi:http://dx.doi.org/10.4108/icst.simutools.2011.245535.