A NOVEL PROTOCOL ARCHITECTURE FOR IOT: EFFICIENCY THROUGH DATA AND FUNCTIONALITY SHARING ACROSS LAYERS

VINÍCIUS GALVÃO GUIMARÃES

TESE DE DOUTORADO EM ENGENHARIA DE
SISTEMAS ELETRÔNICOS E DE AUTOMAÇÃO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

A NOVEL PROTOCOL ARCHITECTURE FOR IOT: EFFICIENCY THROUGH DATA AND FUNCTIONALITY SHARING ACROSS LAYERS

VINÍCIUS GALVÃO GUIMARÃES

ORIENTADOR: RENATO MARIZ DE MORAES
COORIENTADOR: ADOLFO BAUCHSPIESS
COORIENTADORA: KATIA OBRACZKA

TESE DE DOUTORADO EM ENGENHARIA DE
SISTEMAS ELETRÔNICOS E DE AUTOMAÇÃO

PUBLICAÇÃO: PPGEA.TD-143/2019
BRASÍLIA/DF: JUNHO - 2019

A NOVEL PROTOCOL ARCHITECTURE FOR IOT: EFFICIENCY THROUGH DATA AND FUNCTIONALITY SHARING ACROSS LAYERS

VINÍCIUS GALVÃO GUIMARÃES

TESE DE DOUTORADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR.

APROVADA POR:

[illegible]

Prof.ª Dr.ª Flavia Maria Guerra de Sousa Aranha Oliveira– ENE/UnB
Presidente da Banca

[illegible]

Prof. Dr. Paulo Roberto de Lira Gondim– ENE/UnB
Membro Interno - ENE/UnB

[illegible]

Prof. Dr. Jacir Luiz Bordim– CIC/UnB
Membro Interno

[illegible]

Prof.ª Dr.ª Cíntia Borges Margi– PCS/EPUSP
Membro Externo

BRASÍLIA, 28 DE JUNHO DE 2019.

FICHA CATALOGRÁFICA

GUIMARÃES, VINÍCIUS GALVÃO
A Novel Protocol Architecture for IoT: Efficiency Through Data and Functionality Sharing Across Layers [Distrito Federal] 2019.
xxiii, 140p., 210 x 297 mm (ENE/FT/UnB, Doutor, Engenharia de Sistemas Eletrônicos e de Automação, 2019).
Tese de doutorado – Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Comunicação Sem Fio	2. Internet das Coisas
3. Eficiência Energética	4. Pilha de Protocolos
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

GUIMARÃES, V. G. (2019). A Novel Protocol Architecture for IoT: Efficiency Through Data and Functionality Sharing Across Layers. Tese de doutorado em Engenharia de Sistemas Eletrônicos e de Automação, Publicação PPGEA.TD-143/2019, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 140p.

CESSÃO DE DIREITOS

AUTOR: Vinícius Galvão Guimarães
TÍTULO: A Novel Protocol Architecture for IoT: Efficiency Through Data and Functionality Sharing Across Layers.
GRAU: Doutor ANO: 2019

Vinícius Galvão

Vinícius Galvão Guimarães

Departamento de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

To Aída, Roberto, Ana Paula, Geneviève, and family.

ACKNOWLEDGMENTS

At first, I would like to thank God, that provided this great adventure and strength. Thank you for everything I am.

Right after, I thank my parents that always guided me along the way, believed in my potential, and encouraged me to move on. My sister, who is and always will be my friend and companion. My partner Genevieve, for supporting and standing next to me in these moments. My family, who have always cheered for my success along the way. My friends, for their credibility, patience, and support.

My advisor Prof. Renato Mariz de Moraes and co-advisor Prof. Adolfo Bauchspiess, who believed in my work and shared their knowledge for the development of this project. Their dedication and support will always be carried with me, and it will be an inspiration of professionalism, conduct, and yet friendship that I shall return to whoever works with me.

My co-advisor Prof. Katia Obraczka of the University of California Santa Cruz, who held me as her student during an amazing exchange research experience. Also, transmitted her knowledge and helped me improving this work to a new higher level.

For all of my friends and colleagues who were present on this journey and made it remarkable and priceless. Definitely, about friendship, I could not be luckier. I thank you all (Obrigado, Gracias, Merci, Dhan'yavāda, Arigato gozaimashita).

The LARA laboratory and its students who supported me and made it an ideal environment to work. A small place in space, but one of the greatest centers of intelligence and technical quality of UnB.

For their financial support to this thesis, I would also like to thank The Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

The Faculty of Technology and the University of Brasília for giving me this opportunity to get these names on my resume.

ABSTRACT

Title: A Novel Protocol Architecture for IoT: Efficiency Through Data and Functionality Sharing Across Layers
Author: Vinícius Galvão Guimarães
Supervisor: Renato Mariz de Moraes
Co-Supervisor: Adolfo Bauchspiess
Co-Supervisor: Katia Obraczka
Graduate Program in Electronic and Automation Systems Engineering
Brasília, June 28, 2019

TCP/IP is a standard stack for communication networks and present in many communication systems. However, in the Internet of Things (IoT) applications, many works propose cross-layer designs or even very different architectures to improve energy efficiency. Motivated by the need to accommodate IoT devices with limited power, processing, storage, and communication capabilities, this work introduces the IoT Unified Services, or IoTUS, a novel network protocol architecture that targets energy efficiency and compact memory footprint. IoTUS uses an extensible *service layer* that facilitates cross-layer sharing. It promotes sharing of both network control information (e.g., number of transmissions, receptions, collisions at the data-link layer) and functionality (e.g., neighbor discovery, aggregation) by different layers of the protocol stack. Additionally, IoTUS can be used by existing network stacks without having to modify the basic operation of their protocols. We implemented IoTUS on the Cooja-Contiki network simulator/emulator. Our theoretical and simulation results were similar and coherent. Extended simulations showed that IoTUS framework attained up to 76.83% less energy consumption than adapted ContikiOS stack in a monitoring application, with a linear topology of 10 nodes. For a 44 nodes tree topology, IoTUS got a network average of 42.33% less energy consumption. Consequently, IoTUS reached a network lifetime of 43 days, while adapted ContikiOS got up to 24 days. IoTUS used approximately 4 kbytes more of flash memory than adapted ContikiOS stack, but reduce up to 31.31% of the RAM usage. Also, network overhead in IoTUS resulted in approximately 81.3% in a 44 nodes tree topology, while adapted ContikiOS attained 87.3%. Our theoretical and simulation results showed improved performance, better energy efficiency, a more extended network lifetime, and more compact memory footprint when compared to current IoT protocol architectures.

Keywords: Wireless communication, Internet of Things, Energy Efficiency, Protocol Stack.

RESUMO

Título: Uma Nova Arquitetura de Protocolos para IoT: Eficiência Através do Compartilhamento de Dados e de Funcionalidades entre Camadas
Autor: Vinícius Galvão Guimarães
Orientador: Renato Mariz de Moraes
Coorientador: Adolfo Bauchspiess
Coorientadora: Katia Obraczka
Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos e de Automação
Brasília, 28 de junho de 2019

A pilha TCP/IP é um padrão para redes e, portanto, está presente em muitos sistemas de comunicação. No entanto, em aplicações com *Internet of Things* (IoT), muitos trabalhos propõem um design que infringem a restrição de acesso entre camadas não adjacentes ou, até mesmo, novas arquiteturas de protocolos para melhorar a eficiência energética. Motivado pela necessidade de acomodar dispositivos IoT com recursos limitados de energia, processamento, armazenamento e comunicação, este trabalho apresenta o *IoT Unified Services*, ou IoTUS, uma nova arquitetura de protocolos de rede voltada para eficiência de energia e compacto uso de memória. O IoTUS usa uma camada de serviços extensível que facilita o compartilhamento entre camadas. Promove também o compartilhamento das informações de controle de rede (por exemplo, número de transmissões, recepções, colisões na camada de enlace de dados) e funcionalidades (por exemplo, descoberta de vizinhos, agregação de pacotes) para diferentes camadas da pilha de protocolos. Além disso, o IoTUS pode ser usado por arquiteturas de protocolos já existentes, sem ter que modificar a proposta de seus protocolos já desenvolvidos. O IoTUS foi implementado no simulador/emulador de rede Cooja-Contiki. Os resultados teóricos e de simulação foram semelhantes e coerentes. As simulações estendidas mostraram que o IoTUS consumiu $76,83\%$ menos energia comparado à pilha de comunicação adaptada do ContikiOS em uma aplicação de monitoramento, com uma topologia linear de 10 nós. Para uma topologia de 44 nós, a IoTUS obteve uma média de $42,33\%$ menos consumo de energia. Consequentemente, o IoTUS atingiu uma vida útil de 43 dias, enquanto a pilha ContikiOS adaptada chegou a 24 dias. O IoTUS usou aproximadamente 4 kbytes a mais de memória flash do que a pilha ContikiOS adaptada, mas reduziu em até $31,31\%$ o uso de RAM. Além disso, o excesso de cabeçalhos na rede IoTUS foi de aproximadamente $81,3\%$ com uma topologia em árvore de 44 nós, enquanto o ContikiOS adaptado resultou em $87,3\%$. Portanto, os resultados teóricos e de simulação mostraram melhor desempenho do IoTUS, melhor eficiência energética, maior vida útil da rede e um compacto uso de memória, quando comparado às atuais arquiteturas de protocolos de IoT.

Palavras-chave: Comunicação Sem Fio, Internet das Coisas, Eficiência Energética, Pilha de Protocolos.

SUMMARY

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$App_{Payload}$	Application payload	[bytes]
App_{msg}	A single application message in Rime stack	[bytes]
B_{Events}	Broadcast events	
Bat	Battery energy available for one node	[J]
$Clock_{Res}$	Clock resolution of the device	[units/s]
DL_Ans	Amount of bytes in a DL's answer frame with Rime stack	[bytes]
DL_Reg	Amount of bytes in a DL's register frame with Rime stack	[bytes]
DL_Req	Amount of bytes in a DL's request frame with Rime stack	[bytes]
DL_{Beacon}	Amount of bytes in a DL beacon frame with Rime stack	[bytes]
E_{Ack}	Energy consumed with for one ACK	[J]
E_{CCA}	Energy consumed with one CCA process	[J]
E_{DSleep}	Energy consumed with radio's deep-sleep state (Lowest consumption)	[µJ]
$E_{First\ tx}$	Energy consumed for a ContikiMAC's first transmission	[J]
E_{Idle}	Energy consumed with radio's idle state	[mJ]
$E_{Other\ tx}$	Energy consumed for a ContikiMAC's transmission	[J]
E_{Pkt}	Energy consumed with for one packet with ContikiMAC	[J]
E_{Rx}	Energy consumed with radio's reception	[mJ]
$E_{Sending\,node}$	Final energy consumed of a transmitting node with ContikiMAC	[J]
E_{Sleep}	Energy consumed with radio's sleep state	[mJ]
E_{State}	Accumulated energy consumed in a state	[J]
E_{Tx}	Energy consumed with radio's transmission	[mJ]
$E_{Wake\,up}$	Energy consumed with one reception event (radio at Rx state)	[J]
Ev_{report}	Number of measuring reports of an experiment	
I_{Idle}	Drawn current with radio's idle state	[A]
I_{Sleep}	Drawn current with radio's sleep state	[A]
I_{State}	Drawn current with a radio's state	[A]
I_{Tx}	Drawn current with radio's transmission state	[A]
$IoTUS\_App_{msg}$	A single application message with IoTUS	[bytes]
$IoTUS\_KA_{Lin\_Conn}$	IoTUS's expected amount of bytes with Network's KA packets in a linear topology	[bytes]

$IoTUS\_ND_{Lin\_BC}$	IoTUS's expected amount of bytes with broadcast events in a linear topology	[bytes]
$IoTUS_{2\,nodes\,Sen\&KA}$	IoTUS's expected amount of bytes during the sensing and KA stage for two nodes	[bytes]
$IoTUS_{2\,nodes\,overhead}$	IoTUS's expected overhead for two nodes	[bytes]
$IoTUS_{Ans}$	Amount of bytes in IoTUS's answer packet with ND service	[bytes]
$IoTUS_{Beacon}$	Amount of bytes in IoTUS's beacon packet with ND service	[bytes]
$IoTUS_{DACK}$	Amount of bytes in IoTUS's DAO-ACK packet with ND service	[bytes]
$IoTUS_{DAO}$	Amount of bytes in IoTUS's DAO packet with ND service	[bytes]
$IoTUS_{Lin\_Ctrls}$	IoTUS's expected amount of bytes used with control packets in a linear topology	[bytes]
$IoTUS_{Lin\_Overhead}$	IoTUS's expected final overhead in a linear topology	[bytes]
$IoTUS_{Lin\_POverhead}$	IoTUS's expected *pure header overhead* in a linear topology	[bytes]
$IoTUS_{Lin\_RCtrls}$	IoTUS's expected amount of bytes used with control packets for the remaining duration of the experiment in a linear topology	[bytes]
$IoTUS_{Overhead}$	Expected overhead in IoTUS	[bytes]
$IoTUS_{Pure\,overhead}$	Expected *pure header overhead* in IoTUS	[bytes]
$IoTUS_{Reg}$	Amount of bytes in IoTUS's register packet with ND service	[bytes]
$IoTUS_{Req}$	Amount of bytes in IoTUS's request packet with ND service	[bytes]
$IoTUS_{Sen\&KA}$	IoTUS's expected amount of bytes during the sensing and KA stage	[bytes]
$IoTUS_{Single\,conn}$	Amount of bytes used for one association process in IoTUS	[bytes]
$IoTUS_{Total\,conn}$	Total amount of bytes used for during the association process in IoTUS	[bytes]
NL_{Beacon}	Amount of bytes in a Network's DIO packet with Rime stack	[bytes]
NL_{DAO_ACK}	Amount of bytes in a Network's DAO-ACK packet with Rime stack	[bytes]
NL_{DAO}	Amount of bytes in a Network's DAO packet with Rime stack	[bytes]
NL_{DIO}	Amount of bytes in a Network's DIO packet with Rime stack	[bytes]
NL_{DIS}	Amount of bytes in a Network's DIS packet with Rime stack	[bytes]
$Node_{Power}$	Average power consumption of a node	[W]
Num_{Tx}	Number of transmissions	
N	Total number of motes of a network	

$PB_{App\&DAO}$	*Piggyback Service*'s DAO and application message in IoTUS	[bytes]
RCE_{Events}	Reception event representing the radio hardware in the reception state during a short time	
$Rime\_DL_{Lin\_BC}$	Rime's expected amount of bytes with DL's broadcast events in a linear topology	[bytes]
$Rime\_KA_{Lin\_Conn}$	Rime's expected amount of bytes with Network's KA packets in a linear topology	[bytes]
$Rime\_NL_{Lin\_BC}$	Rime's expected amount of bytes with Network's broadcast events in a linear topology	[bytes]
$Rime_{2\,nodes\,Sen\&KA}$	Rime's expected amount of bytes during the sensing and KA stage for two nodes	[bytes]
$Rime_{2\,nodes\,overhead}$	Rime's expected overhead for two nodes	[bytes]
$Rime_{DL\,conn}$	Amount of bytes used for a DL's association process in Rime stack	[bytes]
$Rime_{Lin\_Ctrls}$	Rime's expected amount of bytes used with control packets in a linear topology	[bytes]
$Rime_{Lin\_Overhead}$	Rime's expected final overhead in a linear topology	[bytes]
$Rime_{Lin\_POverhead}$	Rime's expected *pure header overhead* in a linear topology	[bytes]
$Rime_{Lin\_RCtrls}$	Rime's expected amount of bytes used with control packets for the remaining duration of the experiment in a linear topology	[bytes]
$Rime_{NL\,conn}$	Amount of bytes used for a Network's association process in Rime stack	[bytes]
$Rime_{Overhead}$	Expected overhead in Rime stack	[bytes]
$Rime_{Pure\,overhead}$	Expected *pure header overhead* in Rime stack	[bytes]
$Rime_{Sen\&KA}$	Rime's expected amount of bytes during the sensing and KA stage	[bytes]
$Rime_{Single\,conn}$	Amount of bytes used for one association process in Rime stack	[bytes]
$Rime_{Total\,conn}$	Total amount of bytes used for during the association process in Rime stack	[bytes]
$Tdev'_{State}$	Accumulated value of a state in a device's timing system	
Tot_B_{Events}	Total broadcast events in a simulation	
V_{PP}	Battery voltage	[V]
t_{AStage}	Average duration to complete all network association	

t_{Beacon}	Period of beacons in IEEE 802.15.4 MAC	[ms]
t_{Idle}	Time elapsed with radio's idle state	[µs]
$t_{SF\,base}$	aBaseSuperFrameDuration parameter in IEEE 802.15.4 MAC	[ms]
t_{Sleep}	Time elapsed with radio's sleep state	[µs]
t_{State}	Time elapsed with a radio's state	[µs]
$t_{Superframe}$	Superframe period in IEEE 802.15.4 MAC	[ms]
$t_{one\,conn}$	Average duration of one complete node association	
$\lfloor x \rfloor$	The floor function, which outputs the greatest integer less than or equal to x	

LIST OF ACRONYMS AND ABBREVIATIONS

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks. 2, 8, 18, 19, 22
6TiSCH	IPv6 over the TSCH mode of IEEE 802.15.4e. 8, 22
ACK	Acknowledgment. 10, 13, 18, 28, 35, 52, 55–57, 66, 85
API	Application Programming Interface. 31, 33
APT	Advanced Packaging Tool. 24
BO	Beacon Order. 14
CCA	Clear Channel Assessment. 10, 46, 55–57
CLAMP	Cross-Layer Management Plane. 3, 8, 19, 22
ContikiOS	Contiki Operational system. 9, 10, 13, 19, 20, 22, 30, 35, 43–45, 49, 79, 104, 105
CPU	Central Process Unit. 46, 47
CSMA	Carrier Sense Multiple Access. 85
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance. 10
CTS	Clear to Send. 10
DAO	DODAG Advertisement Object. 18, 40, 52, 54, 64–71
DIO	DODAG Information Object. 17, 18, 49, 52, 54, 66–69
DIS	DODAG Information Solicitation. 17, 52, 54
DL	Data Link. 4, 5, 8–10, 13, 16, 18, 20, 25, 27, 28, 49, 52, 55, 57, 59–64, 66–69, 78, 84, 85, 88, 92, 93, 104
DODAG	Destination Oriented Directed Acyclic Graph. 17, 52
EMP	Energy Management Plane. 19
FCS	Frame Check Sequence. 15
FLIP	Flexible Interconnection Protocol. 7
GCRAD	Geographic Cross-layer Routing Adapted for Disaster. 8, 22
GNRC	Riot's GENERIC Stack. 10, 22

HDr	Header. 40
ICMPv6	internet control message protocol for IPv6. 17, 64
ID	Identification Number. 12–14, 21, 47
IDE	Integrated Development Environment. 44
IEEE	Institute of Electrical and Electronic Engineers. 2, 14
IoT	Internet of Things. 1, 2, 4, 5, 7–10, 16, 18, 19, 21, 23, 24, 34, 38, 44, 45, 50
IoTUS	IoT Unified Services framework. 4–6, 23–35, 37–39, 43–45, 49–54, 61–64, 66–74, 78–90, 92–96, 98–100, 102–105, 128
IP	Internet Protocol. 2
IPv4	Internet Protocol version 4. 7, 22
IPv6	Internet Protocol version 6. 2, 7, 8, 16–18, 22, 27, 105
ISO	International Organization for Standardization. 2
IWSN	Industrial Wireless Sensor Networks. 7
KA	Keep Alive. 25, 27, 49, 56, 57, 59, 65, 66, 68, 78, 84, 85
LLN	Low-power Lossy Networks. 2, 21, 22, 24
LPL	Low Power Listening. 12, 13
LPM3	Low Power Mode 3. 46, 47
LT	Lifetime. 82
MAC	Medium Access Control. 9, 20, 45, 49
MCU	Microcontroller Unit. 46, 59
ND	Neighbor Discovery. 2, 16, 18, 28, 42, 52
NDCC	Neighbor Discovery Control Command. 42
OFM	Open Framework Middleware. 9, 22
OS	Operational System. 9–11, 22, 45, 47, 48
OSI	Open System Interconnection. 2, 10, 19–21, 23
PAN	Personal Area Network. 15
PBp	Piggyback piece. 39, 40, 64

QoS	Quality of Service. 3, 18
RAM	Random Access Memory. 24, 83, 100, 102, 105
RDC	Radio-Duty Cycle. 9, 20, 44
RDML	Reducing Delay and Maximizing Lifetime. 7, 22
RPL	Routing Protocol for Low-Power and Lossy Networks. 2, 7, 8, 16, 17, 22, 27, 40, 43, 49, 54, 67, 104, 105
RTS	Request to Send. 10
SFD	Start of Frame delimiter. 15
SN	Sequence Number. 15, 47
SO	Superframe Order. 14
SRH	Source Route Header. 17, 18
TCP	Transmission Control Protocol. 2
TinyOS	Tiny Operational System. 9, 10
TSCH	Time-Slotted Channel Hopping. 8, 22
TSTP	Trustful Space-Time Protocol. 9, 22
UCLEAH	Uniform Clustering with Low Energy Adaptive Hierarchy. 8, 22
UDGM	Unit Disk Graph Medium. 48, 49
WSAN	Wireless Sensor and Actuator Networks. 9, 14
WSN	Wireless Sensor Networks. 9, 11, 22, 24
XLM	Unified Cross-Layer Module. 8, 22

1 INTRODUCTION

This chapter contextualizes the importance of new designs to the traditional communication stack for IoT applications. It also presents the problems in the current stacks, as well as the objectives and contributions of this work.

Internet usage initially grew with the number of computer users. With the development of new devices, the internet reached even further users as well as applications and functionalities. Therefore, a new paradigm to connect devices was created, the Internet of Things (IoT). Figure 1.1 illustrates some examples of IoT applications.

With technological development, new IoT devices are being integrated into our daily life, e.g., smart watches, autonomous cars, etc. Some innovative devices can harvest their energy from the environment [1, 2]. However, for most applications, energy is critical as the amount of transmissions/receptions is supplied by batteries.

With technology advances, devices' size and energy consumption tend to reduce. However, according to [3], IoT devices are likely to stay resource-constrained in the near future; hence, the relevance of the present work, targeting energy efficiency through an innovative protocol stack.

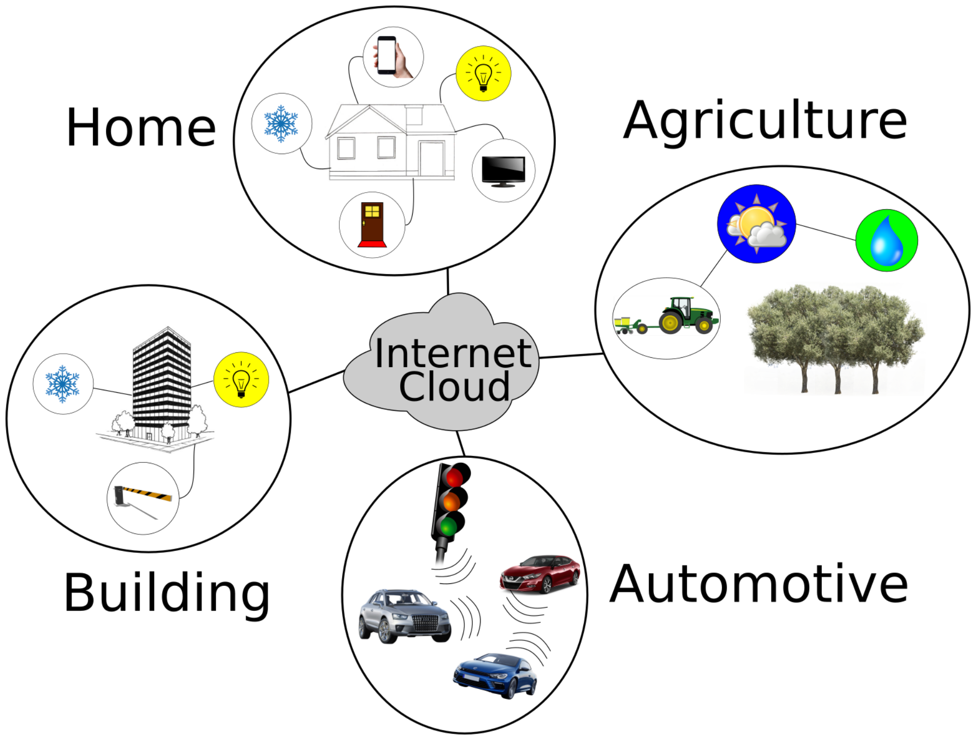


Figure 1.1 – Internet of Things devices with illustrative applications.

Similarly to the ISO/OSI (Open System Interconnection) communication standard [4], the design of the Internet's TCP/IP protocol architecture followed the principles of *layered system design*. As such, the functions performed by the TCP/IP protocol suite are implemented at different protocol layers, where each layer provides a specific set of services to the layer above through a well-defined interface. Using this interface, data being received or sent is passed through the stack. Accessing services provided by layer i through a well-defined interface shields layer $i + 1$ from the implementation details of layer i, simplifying each layer's design and improving overall's modularity, maintainability, and extensibility. However, the layered design approach can increase overhead, as each layer incurs additional communication (e.g., additional header fields) and processing costs. Furthermore, limiting the flow between layers restricts sharing of functionality across layers and leads to duplicated functions at different layers.

Motivated by the emergence of wireless networks, the networking research community devoted considerable attention to cross-layer approaches as a way to circumvent the limitations imposed by the traditional TCP/IP layered protocol architecture. Accordingly, a wide range of techniques that use cross-layer information aiming to improve performance was proposed [5, 6]. Even standard protocols like IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [7], developed by Institute of Electrical and Electronic Engineers (IEEE), use cross-layer operations to accommodate Internet Protocol version 6 (IPv6) [8] into Low-power Lossy Networks (LLN).

1.1 MOTIVATION

- Increasing number of IoT applications;
- Energy efficiency in wireless communications;
- Cross-layer designs' benefits and challenges.

1.2 PROBLEM DESCRIPTION

Protocols like IEEE 802.15.4 MAC [9], 6LoWPAN [7], IPv6 [8], and RPL [10] are standards in IoT systems. However, many of their procedures are duplicated on different levels (Figure 1.2). As discussed by [11], e.g., the Neighbor Discovery (ND) protocol of 6LoWPAN has to consider the designs of IPv6-ND [12] and RPL's ND.

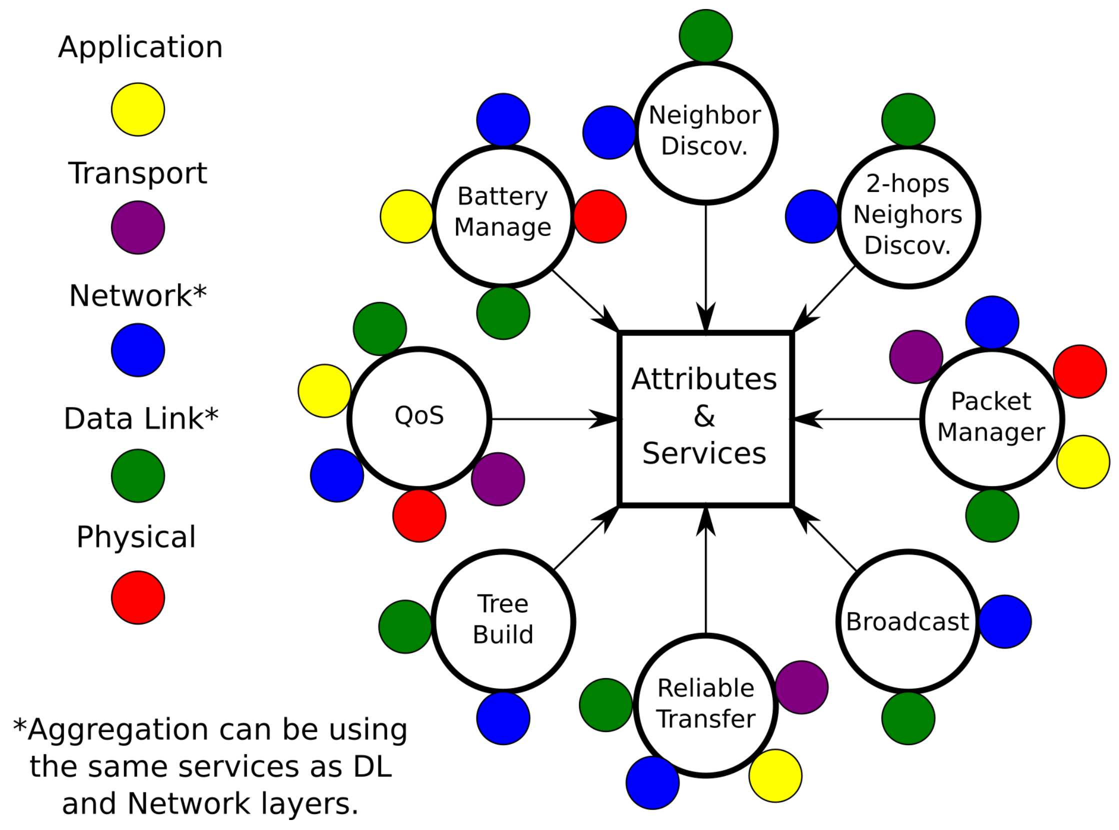


Figure 1.2 – Common procedures sorted by layers.

Many works have proposed better compatibility and optimization using a cross-layer approach. However, as pointed out by [5], there is not a wide accepted standard yet. As described in more detail in Chapter 2, several approaches have tried to bridge this gap. Notable examples include i) Using an adaptation layer that translates/optimizes header fields of standard protocols so they can run on capability-challenging devices [7], leading to monolithic protocol stacks [13]. ii) Developing more effective ways for cross-layer information sharing, e.g., CLAMP [14] and Rime [15]. However, most proposals to-date have either developed modules that require protocol redesign or use traditional cross-layer data exchange.

Data exchange between protocols of different layers can reduce memory usage and facilitate cross-layer operations, but protocols are not aware of each other procedures. Therefore, unless a set of protocols are designed to operate in the same stack, their procedures can be redundant and impact in the number of packets exchanged and energy consumed. In the layered OSI stacks, the impact of reduced information exchange between layers can be more significant, since the header size of packets can increase with the encapsulation process, which one protocol can not necessarily understand another protocol's header.

1.3 OBJECTIVES

1.3.1 General objectives

- Improve energy efficiency in IoT networks.

1.3.2 Specific objectives

- Identify opportunities from the state of the art of IoT;
- Propose and develop a framework to address cross-layer, aggregation, and functions for energy saving;
- Reduce the overall number of messages exchanged while improving performance, especially non-application control packets;
- Compare its performance with the state-of-the-art IoT stack.

1.4 CONTRIBUTIONS

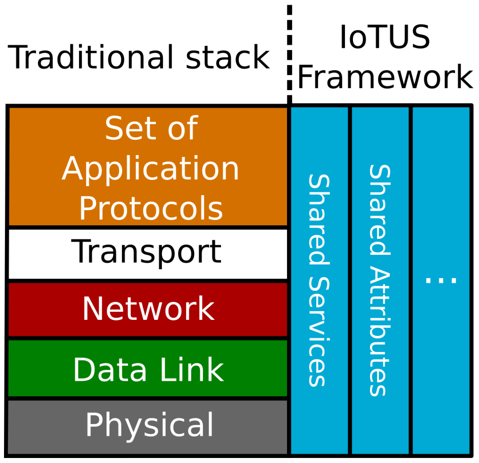


Figure 1.3 – IoTUS's extensible cross-layer sharing framework.

IoT Unified Services framework (IoTUS) proposes an extensible *service layer* that facilitates cross-layer sharing of not only control plane information, or *attributes* (e.g., number of transmissions, receptions, collisions at the Data Link layer) but also *services* (e.g., neighbor discovery, data aggregation), as shown in Figure 1.3. IoTUS can be used by existing protocol stacks allowing information and functionalities sharing among layers without requiring changes to these protocols. With this framework, protocols are aware of each other procedures and therefore, able to optimize their task (increase stack's synergy). Our evaluation by theoretical and extensive simulations shows that IoTUS can achieve significant energy savings, as well as use a smaller memory footprint when compared to the existing IoT stacks, in particular, the stack used in ContikiOS [16].

IoTUS's main contributions are:

- A systematic approach to cross-layer information sharing that yields both energy and storage efficiency;
- A modular and extensible service layer that enables sharing common functionality by different protocol layers;
- A framework that can be used by existing protocol stacks without having to modify their design.

Although IoTUS is available for every layer in the protocol stack, and cross-layer benefits can be obtained by every layer, in this work, we evaluate mainly the procedures operated by the Physical, Data Link, Network, and Application layer.

1.5 OUTLINE

The remainder of this work is divided as follow:

- **Chapter 2** discusses related works, their pros and cons, and how they pushed the development of better solutions;
- **Chapter 3** introduces the new framework IoTUS. It gives the reader an overall understanding of how IoTUS works, and what to expect of it;
- **Chapter 4** explains the details about the IoTUS. It describes the procedures and structures used to develop IoTUS;
- **Chapter 5** describes how this work was evaluated, the simulations proposed, and the methods to obtain the results;
- **Chapter 6** shows the development of theoretical models applied in different scenarios to evaluate and validate IoTUS framework;

- **Chapter 7** presents the simulation results obtained within different scenarios and parameters simulated. Every simulation is compared to the standard stack with equal parameters;
- **Chapter 8** concludes this work, with perspectives for future developments and experiments, followed by the References;
- **Appendix A** has an extended abstract written in Portuguese;
- **Appendix B** shows the main structs and fields used by modules of IoTUS framework;
- **Appendix C** present instructions for obtaining IoTUS's open-source code;
- **Appendix D** presents the Python scripts of the theoretical model used to calculate the expected results with a single node network;
- **Appendix E** presents the Python scripts of the extended theoretical model used to calculate the expected results for a linear topology;
- **Appendix F** lists the published works.

2 RELATED WORK

In this chapter, we present an overview of the current state of the art of protocol stacks and architectures that try to accommodate devices with constrained power, computation, storage, and communication capabilities. Detailed information over these works is further presented.

By design, the standard layered stack protocols encapsulate their information into headers. Thus, the abstraction and hidden data of each layer can cause undesirable effects, such as disconnections, delays, overhead, retransmissions, etc. Many works proposed cross-layer protocols to improve the efficiency of IoT devices [5, 6, 17], and other works also proposed modifications to the standard stack to facilitate this exchange of information between layers [15, 18, 14, 19, 20, 13]. As pointed out in [21], cross-layer design can eliminate a large number of control packets while improving decision making within protocols. On the other hand, layered stack allows loosely coupled modular protocols.

Through experiments using the Cooja-Contiki simulator [22, 16], the work reported in [23] shows that the control overhead in Routing Protocol for Low-Power and Lossy Networks (RPL) [10] can represent about 25% of the overall traffic in a 20-node network and can go up to 75% with 100 nodes. According to [23], it is worth noting that IPv6 [8] and RPL [10] use similar control packets for neighbor discovery, which could generate additional control overhead.

The Flexible Interconnection Protocol (FLIP) [24] uses flexible headers to interconnect heterogeneous devices while accommodating their different capabilities. The overhead incurred by FLIP's header will depend on the capabilities of the device running FLIP and the needed functionality. For low-power devices like scalar sensors (e.g., temperature, humidity, etc.), which typically need to send out their readings periodically, FLIP provides considerable savings when compared to fixed-length protocols like Internet Protocol version 4 (IPv4) [25] and IPv6 [8].

RAWMAC [26] is based on ContikiMAC [27] and uses cross-layer access on RPL to align the radio's wake-up schedule. Therefore, similar to D-MAC [28], RAWMAC reduced upward packet delay for monitoring systems.

In Reducing Delay and Maximizing Lifetime (RDML) [29], it maximizes network lifetime while reducing the message delay and the number of retransmissions for Industrial Wireless Sensor Networks (IWSN). This work assumes that energy efficiency is already optimized, and so, increasing the duty-cycle of nodes with residual energy will not impact on network lifetime. For that, cross-layer operations are required since position and proximity to the sink node are used in different layers.

Other efforts towards efficient protocol architectures specifically targeting networks of low power, low capability devices (e.g., wireless sensor networks) have used a “monolithic" approach, i.e., combining the functionality of multiple layers into a single layer. A notable example is the Unified Cross-Layer Module (XLM) [13] and Uniform Clustering with Low Energy Adaptive Hierarchy (UCLEAH) [19].

Adaptation-layer protocols like IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [7] have also been proposed as a way to translate protocol fields and optimize headers in order to be able to adapt standards such as IPv6 [8] and RPL [10] to run on top of IEEE 802.15.4 [30] in IoT devices. 6LoWPAN, IPv6, IEEE 802.15.4, and RPL can be considered the *de facto* protocols for the IoT stack, according to [31].

Other works like Time-Slotted Channel Hopping (TSCH) of IEEE 802.15.4e [9] have improved the design of IEEE 802.15.4 MAC, using channel hopping and better synchronization techniques to improve the protocol’s efficiency, latency, and reliability,especially for industrial applications. Also, with similar goals as 6LoWPAN, IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) [32] adaptation-layer protocol was proposed. Moreover, updates and improvements are still being developed, as presented in [33]. Finally, in [34], the 6TiSCH performance discussed. 6TiSCH makes use of cross-layer operations to optimize its performance, but its design does also consider the traditional layered protocol stack.

Cross-layer operations can be presented in most layers of the stack [35]. However, Physical, Data Link (DL), and Network layers can provide more impacting cross-layer optimization toward energy efficiency, e.g., Geographic Cross-layer Routing Adapted for Disaster (GCRAD) [36] improves end-to-end delay and energy efficiency compared with other state-of-the-art geographic routing methods. Another example is Leach-CLO [37], which is a clustering routing protocol and proposed a cross-layer optimization model involving Physical, Data Link (DL), and Network layers. Leach-CLO outperforms other traditional layered routing protocols for underwater communication, like Leach-L [38].

Other approaches try to improve IoT protocol stack efficiency through new architectures facilitating cross-layer information sharing. For example, Cross-Layer Management Plane (CLAMP) [14] uses a publish/subscribe/update/query system that allows protocols at different layers to share information. In this work, many other sub-layers were developed to support the entire stack and are discussed further in this chapter.

Another solution for cross-layer sharing is proposed by TinyXXL [20] which provides the

Tiny Operational System (TinyOS) [39] embedded operating system with more efficient data storage, as well as a generic interface for data exchange. TinyXXL is part of TinyCubus [40], a new cross-layer based protocol stack for sensor networks. Moreover, TinyCubus includes TinyAdapt [41], which provides an easier selection of protocols/parameters for a given set of application requirements.

In [18], an Open Framework Middleware (OFM) is developed for Wireless Sensor and Actuator Networks (WSAN). However, since shared information between layers is necessary, the standard layered protocol stack restricts the network software modularity and limits network optimal performance. Therefore, OFM also proposed a new stack consisting of modules (e.g., routing, security and communication services) and abstraction layer to facilitate information access to high layers as OFM's middleware.

Cross-layer information sharing in the Contiki Operational system (ContikiOS) [16], which includes Rime stack [15], works as follows. Information produced by different protocol layers is stored as attribute-value pairs and is accessible by protocols at different layers. Shared information is used in building packet headers with protocols at different layers. ContikiOS uses a Radio-Duty Cycle (RDC) layer separate from the Medium Access Control (MAC) layer. This may cause portability issues as most implementations handle both layers (RDC and MAC) as a single layer [42]. Rime [15] is a collection of sub-layers mainly at Network layer that provides modular network functionalities to the stack. According to [43], using COOJA/MSPSim hardware emulation tool (which includes TMote Sky [44]) enables accurate energy consumption evaluation compared with testbed results, resulting in a maximum difference of $0.6\,\mu\text{W}$ (running up to 15 nodes).

HARE [17] provides a set of protocols (from the Data Link layer until the Transport layer) to optimize uplink multi-hop communication using an adaptive transmission power level. HARE was implemented in real motes and uses ContikiOS [16] and Cooja [22] as its Operational System and simulator, respectively. Because of it, HARE also considers an RDC layer and a MAC layer. The design of many protocols in different layers permits easier cross-layer access and better-optimized operations.

Trustful Space-Time Protocol (TSTP) [21] is an application-oriented, cross-layer protocol for Wireless Sensor Networks (WSN) and IoT which proposed solutions for the entire stack (from the application to the Medium Access Control layer). In this work, a template metaprogramming technique is used to avoid monolithic, tightly-coupled software. Thus, although cross-layer is used, modular programming is still available within its stack design. TSTP improves the network procedures with its combined set of protocols and brings to the application a complete communication solution.

It is interesting to point out that Riot-OS [45], a more recently developed operating system for IoT devices, can use different protocol stacks, i.e., the Operational System (OS) design facilitates the exchange of communication stacks. Nevertheless, the Riot's default stack is named GENERIC (GNRC) and does not make use of any cross-layer sharing. In [46], Riot is compared with other stacks, including ContikiOS, where it shows that packet building time is slightly higher due to driver-processing and its linked lists packet management. Riot is an open-source project [47] and can be implemented in devices also compatible with Cooja [22].

The remainder of this chapter contains detailed information over some reviewed works, which is divided into Data Link layer protocols, Network layer protocols, cross-layer protocols, and other protocol stacks. In the end, we conclude this chapter discussing the gap our work intends to fill and a summarized view of all reviewed works.

2.1 DATA LINK LAYER

In general, Data Link layer can improve energy saving depending on the protocol's design and application's requirement. This is determined by the control on the radio's transmission events. Radio communication can represent the majority of a mote's consumption. Therefore, the group of protocols further described limit their procedures to the Data Link attributions, according to the Open System Interconnection (OSI) stack design.

2.1.1 S-MAC

Sensor-MAC [48], also known as S-MAC, is a consolidated approach in terms of energy-saving system that has been implemented on real motes using TinyOS [39] and Rene Motes [49]. This protocol works only with channel sensing for contention detection, which is similar to Clear Channel Assessment (CCA). Other features of this protocol are:

- **Activation and sleep cycles** [50]: allowing the system to perform greater energy savings. However, this feature may generate delays of transmissions (increased latency).
- **Packets for traffic control**: similar to the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) [51] protocol, e.g., Request to Send (RTS), Clear to Send (CTS), and Acknowledgment (ACK).

An overview of the operation mode of this protocol can be seen in Figure 2.1.

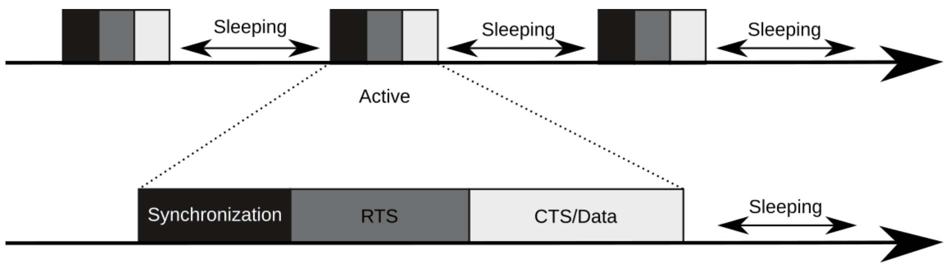


Figure 2.1 – Operating time diagram of S-MAC. Adapted from [48].

Table 2.1 – Implementation size of B-MAC and S-MAC protocols.

Protocol	**ROM (bytes)**	**RAM (bytes)**
B-MAC	3046	166
B-MAC with ACK	3340	168
B-MAC with LPL	4092	170
B-MAC with LPL, and ACK	4386	172
B-MAC with LPL, ACK, and RTS-CTS	4616	277
S-MAC	6274	516

According to [48], S-MAC was developed to meet the need for energy savings at the cost of latency. Moreover, latency over multiple hops was improved in [52]. Therefore, Figure 2.2 illustrates a temporal diagram of S-MAC-AL (adaptive listening).

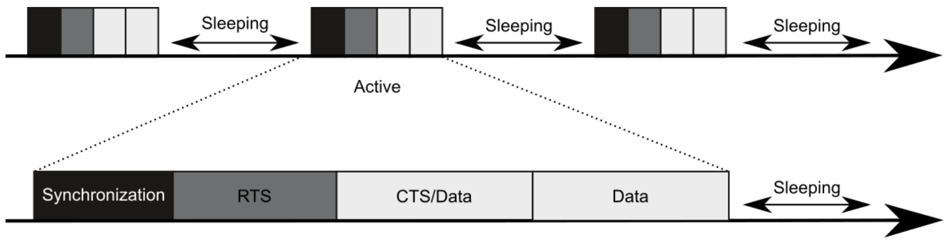


Figure 2.2 – Operating time diagram of S-MAC-AL. Adapted from [52].

Although S-MAC-AL improved latency over S-MAC, its design is not as efficient for larger networks. Nevertheless, S-MAC is still a standard for WSN environments and is present in many recent OSs.

2.1.2 B-MAC

Developed in Berkeley, B-MAC [53] was developed aiming at environment monitoring applications with varying traffic rates. Therefore, instead of synchronization periods, B-MAC uses preambles large enough to be detected by the receiving node. Consequently, energy saving is obtained by reducing idle listening (event in which a device has its radio on but receives no packet). Also, B-MAC developed the Low Power Listening (LPL), which reduces idle listening even more. The implementation size of B-MAC in the MICA2 [54] modules is presented in Table 2.1.

2.1.3 X-MAC

Based on B-MAC [53] and WiseMAC [55], the X-MAC [56] approach significantly improves energy efficiency with idle listening. It proposes small periodic preambles capable of being sensed by the receiver node (Figure 2.3).

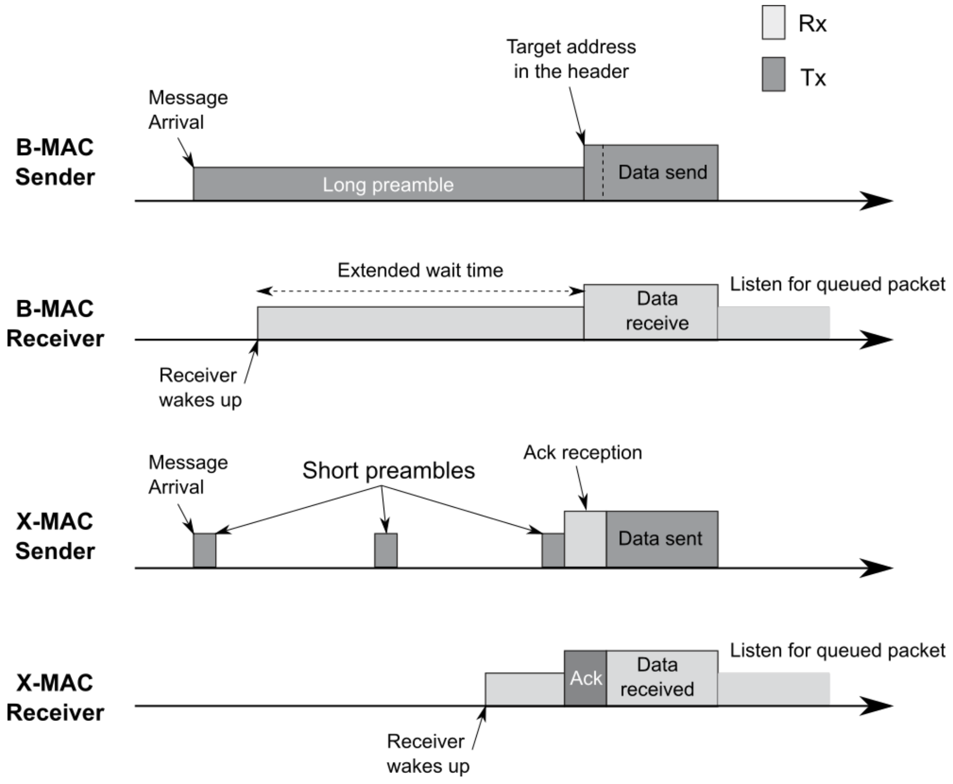


Figure 2.3 – Comparative time diagram of B-MAC and X-MAC. Adapted from [56].

Hence, X-MAC energy efficiency is given not only with its shorter active times but also with its identified preambles. Therefore, the destination node Identification Number (ID) is sent inside the preamble, and thus, other nodes go faster to sleep. Consequently, X-MAC works better in denser networks than B-MAC and WiseMAC.

2.1.4 ContikiMAC

ContikiMAC [27] is implemented in ContikiOS [16], and its design improved the LPL mechanism done in X-MAC [56]. Moreover, it is the *de facto* standard low-power DL protocol used by ContikiOS.

In ContikiMAC, the preamble sent to initiate communication is the packet itself (with its destination ID and payload). Also, Phase-lock mechanism (present in WiseMAC [55]) allows the sender to record the wake-up schedule of a neighbor receiver and uses fewer preambles for the next communications. Meanwhile, the receiver node uses periodical wake-ups to listen for the packet; thus, it sends an ACK frame once a reception occurs.

Figure 2.4 illustrates the communication process in ContikiMAC in which repeated frames already containing the message payload are sent as a preamble until the ACK is confirmed. However, as shown in Figure 2.5, broadcasts are not confirmed, but instead, they are transmitted for the whole period inter-sampling (wake-ups).

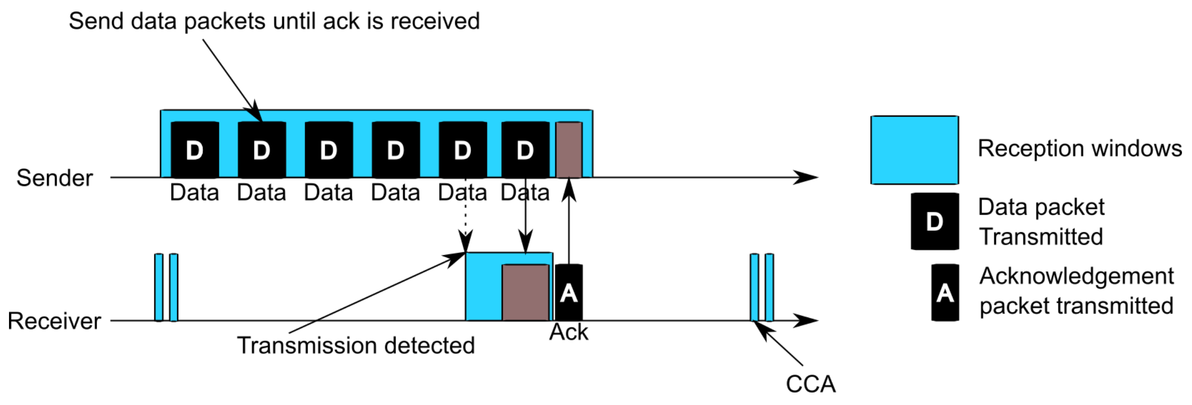


Figure 2.4 – ContikiMAC unicast transmission example.

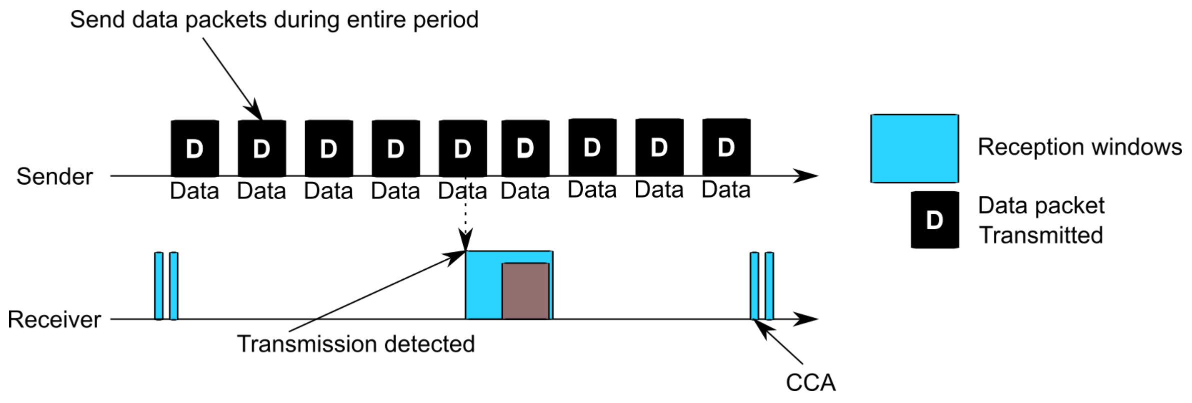


Figure 2.5 – ContikiMAC broadcast transmission example. Adapted from [27].

2.1.5 IEEE 802.15.4 MAC

IEEE 802.15.4 [30] was developed by the IEEE and is a standard for WSAN. It has several modes of operation. Therefore, some examples of parameters to be configured are:

- Maximum active time;
- Beacon network;
- Time between synchronization signals (beacons);
- Nodes' ID of 16-bit or 64-bit.

IEEE 802.15.4 MAC employs *Superframes*, which are communication slots; hence, devices can communicate by either contention processes or reserved slots. During the *Superframe* (Figure 2.6) beacons are transmitted, informing network parameters, and communication slots are available for connected nodes.

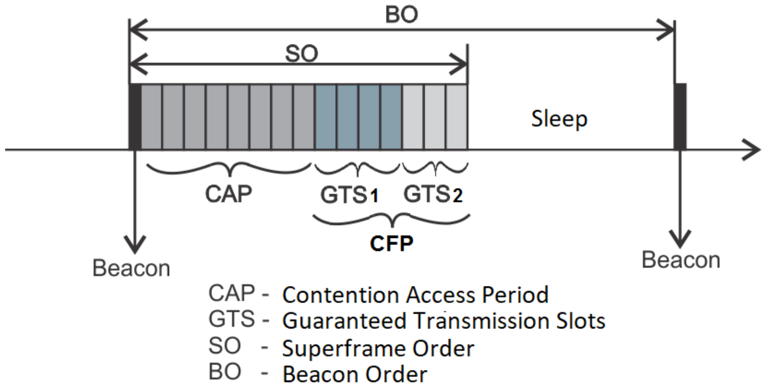


Figure 2.6 – Timing diagram of *Superframe* by the IEEE 802.15.4 MAC protocol. Adapted from [30].

As illustrated in Figure 2.6, the definition of Beacon Order (BO) and Superframe Order (SO) determine the total time of the period between beacons and the maximum active time of the network. Hence, these times are given by

$$t_{Superframe} = t_{SF\,base}.2^{SO}, \tag{2.1}$$

and

$$t_{Beacon} = t_{SF\,base}.2^{BO}, \tag{2.2}$$

where $t_{SF\,base}$ is the aBaseSuperFrameDuration parameter in IEEE 802.15.4 MAC, e.g., $16.36\,\text{ms}$.

Furthermore, IEEE 802.15.4 MAC has a registering process (Figure 2.7) between coordinators and other devices (an end-device or a router node). Thus, its association process takes four main steps for the coordinator: sending beacons, processing an association request (MLME_ASSOCIATE), waiting for a data request, and answering the association response. Meanwhile, the end-device has five other steps: scanning (MLME_SCAN), choosing a coordinator (Personal Area Network (PAN) selection), requesting to associate, requiring for an answer (data request), and processing the answer. Because of this association process, IEEE 802.15.4 MAC can have independent groups coexisting, with a network formed by a general coordinator, routers, and end-devices.

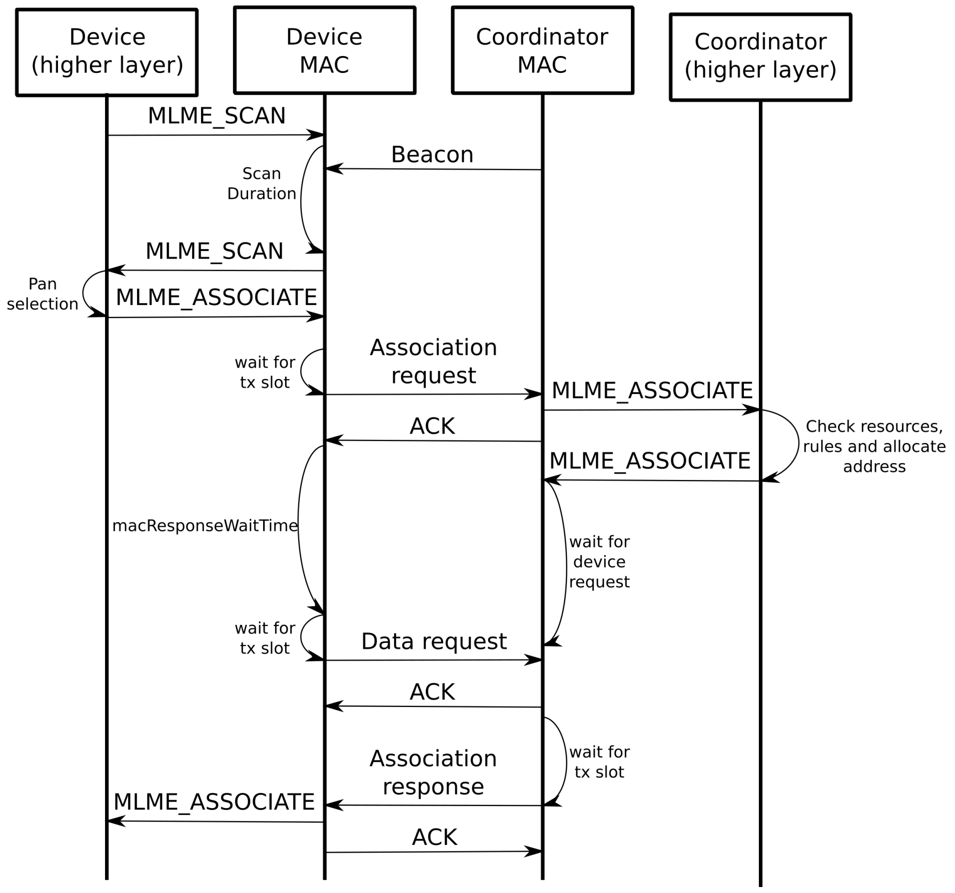


Figure 2.7 – Standard IEEE 802.15.4 MAC association procedure. Adapted from [57].

Figure 2.8 shows an example of the header that can be used by IEEE 802.15.4, with fields like Start of Frame delimiter (SFD), Frame Check Sequence (FCS), Sequence Number (SN), and FCS.

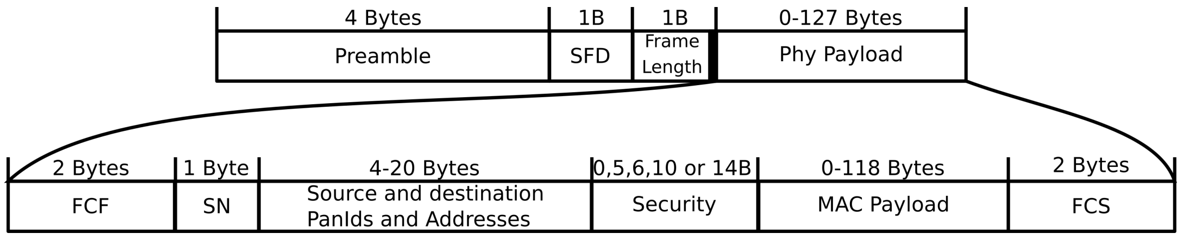


Figure 2.8 – IEEE 802.15.4 Physical and MAC headers. Adapted from [30].

2.2 ROUTING LAYER

The third layer or Network layer has many procedures capable of great impact on energy consumption; thus, Network protocols should also be energy-aware. Moreover, some procedure in the third layer may be redundant with the DL layer, e.g., Neighbor Discovery (ND).

2.2.1 IPv6

Internet Protocol version 6 (IPv6) [8] is the standard for IoT devices in the Network layer since its design allows big address numbers (16 bytes), flexibility, and security. Thus, Many computers, smartphone, and tablets are already supporting the IPv6. However, the address field can be redundant, since Data Link addresses are generally used. Moreover, this standard also defines a, ND procedure [12] that can be redundant with RPL [10].

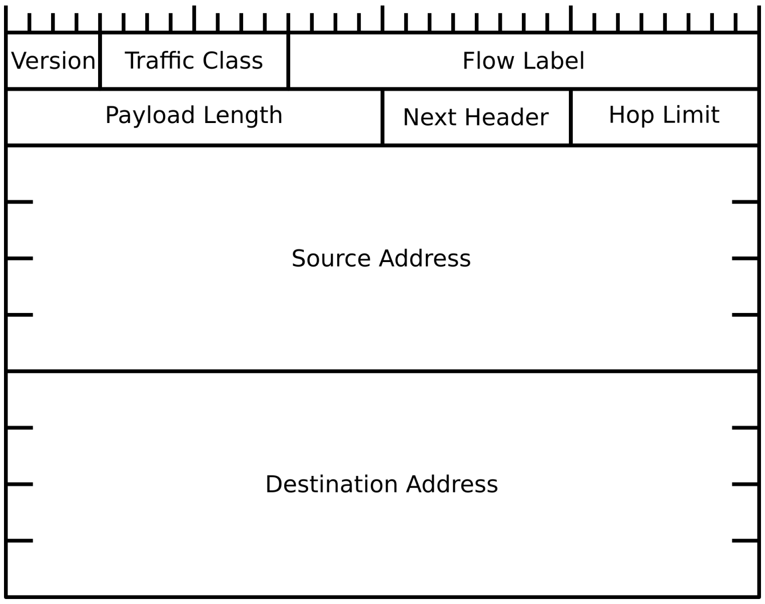


Figure 2.9 – General IPv6 header. Adapted from [8].

Figure 2.9 presents a general IPv6 header. IPv6 headers are too big (40 bytes) to be used by devices developed to transmit at most 127 bytes. Furthermore, this header is not yet final and can increase even more due to its flexibility ("Next header" field). An example of "Next header" field is the option to use Source Route Header (SRH) [58] (Figure 2.10). Another flexibility option is the internet control message protocol for IPv6 (ICMPv6) [59] (Figure 2.11), and so it is used to exchange control packets in the network.

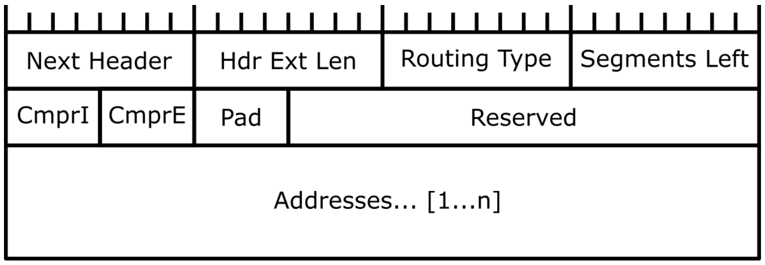


Figure 2.10 – IPv6 Source Routing Header (SRH). Adapted from [58].

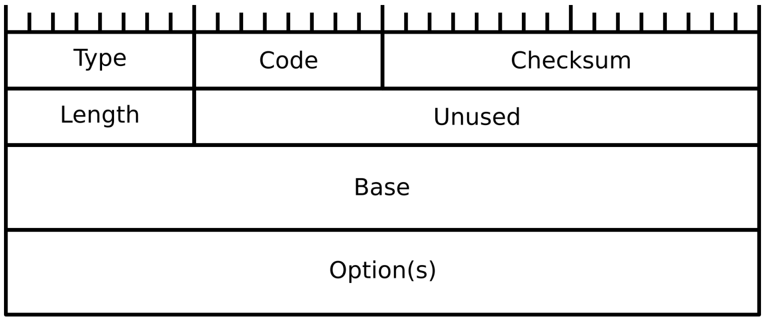


Figure 2.11 – ICMPv6 header. Adapted from [59].

2.2.2 RPL

RPL [10] is another standard routing protocol for constrained devices due to its flexibility, reliability, and reasonable complexity. Its main feature is building a tree topology or Destination Oriented Directed Acyclic Graph (DODAG). The topology consists of one data sink node per instance, using an objective function to determine the network goal and defining which paths to create. However, the instances feature is not always implemented [23]; indeed, most implementations restrict to only one instance at a time.

Therefore, RPL specifies a set of new ICMPv6 control messages to exchange network information:

- **DODAG Information Solicitation (DIS)**: used by the nodes to proactively solicit graph information which is answered with a DODAG Information Object (DIO) packet.

Its functions are fundamental to the building process of the tree, initiating the path upwards to the sink node;

- **DODAG Information Object (DIO)**: the answer of the neighbor device to a DIO packet. It is responsible for building the path upwards, towards the sink node;
- **DODAG Advertisement Object (DAO)**: this frame is responsible for creating and maintaining the path downwards to the end-devices. When this control frame is being relayed, the router device can either store its information or just relay (SRH would then be necessary for downwards packets);
- **DAO_ACK**: an ACK packet for a DODAG Advertisement Object (DAO).

2.3 CROSS LAYER PROTOCOLS

Some works tried to improve energy consumption with optimization over more than one layer. Thus, better energy efficiency was attained, but at the cost of interoperability and/or modularity.

2.3.1 D-MAC

D-MAC [28] improves energy efficiency by reducing idle listening using the Network layer's wake-up schedule. D-MAC main benefits are low latency while keeping energy savings. Also, its design corroborates in networks of a single destination (data sink).

Figure 2.12 illustrates D-MAC's operation. It can be observed that devices optimize their DL wake-up schedule to start transmitting right before its father node, thus reducing idle listening and energy consumption. However, the downwards packets do not have the same benefit; instead, they may experience relatively higher delays. Nevertheless, if necessary, nodes can work more often to maintain Quality of Service (QoS) and ensure efficiency.

2.3.2 6LoWPAN

6LoWPAN [7] was developed to accommodate IEEE 802.15.4 [30] and IPv6 [8] in constrained IoT devices. It reduces the amount of overhead and optimizes redundant functions and field between these protocols, e.g., duplicated addresses and Neighbor Discovery.

Similar to IPv6, 6LoWPAN has flexible headers and uses information in the DL layer

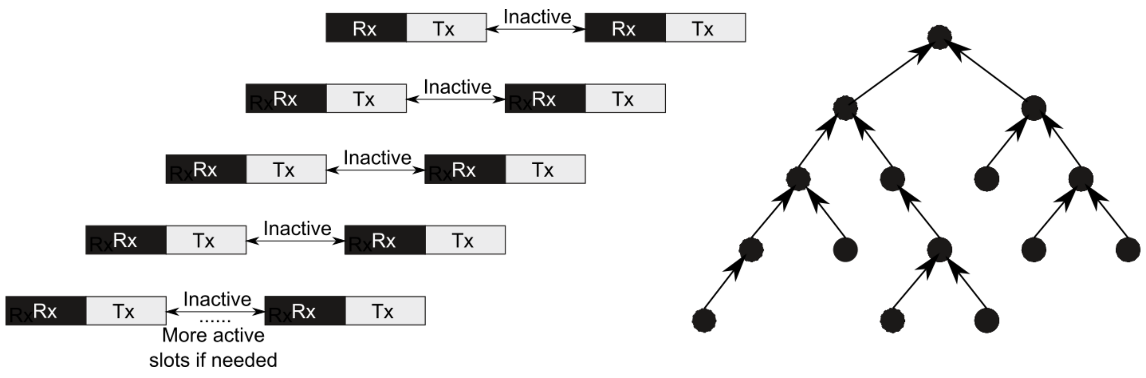


Figure 2.12 – Timing diagram of D-MAC routing operation. Adapted from [28].

to reduce the header size. Also, by doing cross-layer access, 6LoWPAN can improve the addressing procedure. Figure 2.13 presents a general header used by 6LoWPAN.

1 Byte	1 Byte	1 Byte	2 bytes	N Bytes
ESC	EET	EDP	LOWPAN_IPHC	Payload

Figure 2.13 – General 6LoWPAN header. Adapted from [7].

2.4 CROSS LAYER STACKS

This section presents a group of works that changed the standard OSI protocol stack.

2.4.1 CLAMP

CLAMP [14] refers to Cross-Layer Management Plane, which is only one of the layers proposed by this work. Many new interfaces and sub-layers were proposed as well. Figure 2.14 presents the stack and its submodules.

CLAMP also presented a publish/subscribe/update/query system, where protocols can share any information, and other protocols can access it by querying or subscribing to this data. However, some proposed modules have independent operations (set by parameters), e.g., Energy Management Plane (EMP), which is responsible for the radio duty cycle.

2.4.2 Rime stack

ContikiOS [16] is a largely used system for IoT devices; thus, its protocol stack (including Rime [15]) can be considered a standard. Also, the packet managing system (*Packetbuf*)

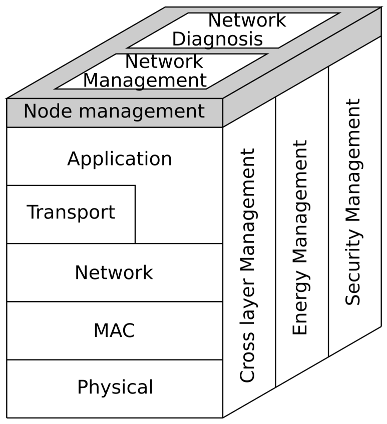


Figure 2.14 – CLAMP stack. Adapted from [14].

allows a shared manner to build packets in the stack.

However, although ContikiOS stack is similar to OSI stack, it splits the DL layer into MAC and RDC (Figure 2.15), with a framer sub-layer to translate *Packetbuf* fields and the radio's headers. Moreover, although protocols use a shared service to build packets, they are still not aware of each other's procedures and, therefore, cannot optimize their task without recurring to cross-layer information access.

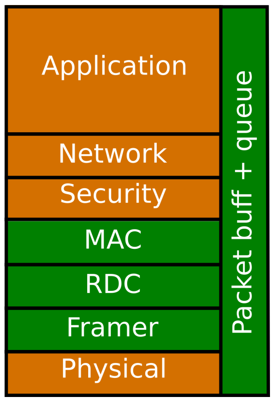


Figure 2.15 – ContikiOS stack. Adapted from [16].

ContikiOS and Cooja [22] form a powerful development environment, but RDC and MAC division along with Rime do not provide a good stack for some protocol implementations (e.g., the IEEE 802.15.4 MAC). In this work, we adapted ContikiOS's protocol stack into a traditional stack composed of only Physical layer (and its framer sub-layer), MAC, Network, Transport, and Application layer. The ContikiOS's *Packetbuf* is still used for packet managing. Therefore, for convenience, this adapted traditional protocol stack in ContikiOS is also called Rime stack for the remaining of this text. The adapted stack accepts any protocol in its layers, as the ContikiMAC, RPL, IEEE 802.15.4 MAC, etc.

2.5 CHAPTER'S CONCLUSION

We observed that cross-layer designs successfully attempted to solve specific problems for specific applications, especially in LLN. For example, many solutions have being proposed for applications with large throughput demand [35]. For many applications, the cross-layer results are usually superior to the traditional layered stacks, either using monolithic approaches or complete new designed stacks. However, the methods to enable cross-layer development are mainly through non-modular protocols stacks, data sharing tools, or architecture modifications. Although works like [21] developed means to keep modular development with cross-layer design, the optimized procedures between protocols are mostly given by their implementation assuming a specific setup of protocols.

Therefore, there is still a gap between traditional layered standard protocols and cross-layer modular designs that will allow both methods to communicate between themselves and still benefit from their qualities. In this way, our work proposes a cross-layer facilitator, with vertical modules/services that allow data exchange and functionality sharing to traditional protocol stacks. Our proposal has a design that does not require the stack to be changed. Hence, this allows a smoother transition from layered stacks to cross-layer designs for IoT devices.

To facilitate comparison, we summarized the reviewed protocols and stacks in Table 2.2, Table 2.3, Table 2.4, and Table 2.5. Thus, we split the work between groups, describing their main contributions, and their restriction to the OSI design, i.e., having cross-layer access.

Status options are:

Tbd — Testbed or real device;

Sim — Simulated Only;

N/A — Not available

Table 2.2 – Reviewed layer-independent Data Link protocols.

Name	Status	Improvements	Year
S-MAC [48]	Tbd	Group wake-up schedule	2002
B-MAC [53]	Tbd	Preambles and idle listening	2004
X-MAC [56]	Tbd	Shorter preambles with ID included	2006
ContikiMAC [27]	Tbd	Payload in the preamble and Phase-lock	2011
IEEE 802.15.4 MAC [30]	Tbd	Versatility and bigger networks	2006
IEEE 802.15.4e [9]	Tbd	More Versatility and channel hopping	2016

Table 2.3 – Reviewed layer-independent Network protocols.

Name	Status	Improvements	Year
IPv4 [25]	Tbd	Internet standard	1981
IPv6 [8]	Tbd	New Internet standard	1998
RPL [10]	Tbd	Routing protocol for LLN	2012

Table 2.4 – Reviewed Cross-layer protocols.

Name	Status	Improvements	Year
RAWMAC [26]	Sim	Aligned wake-up and improved latency	2014
FLIP [24]	Tbd	Dynamic headers	2004
D-MAC [28]	Sim	Improved upwards latency	2004
6LoWPAN [7]	Tbd	Accommodate IPv6 and RPL in LLN	2017
6TiSCH [34]	Tbd	Accommodate IPv6 and RPL with TSCH	2019
GCRAD [36]	Sim	Lower end-to-end delay and lower energy consumption	2017
Leach-CLO [37]	Tbd	Efficiently balance the energy consumption in WSN	2018
RDML [29]	Sim	Maximize nodes' duty-cycle and network lifetime	2018
HARE [17]	Tbd	Improved uplink transmission and energy consumption	2018

Table 2.5 – Reviewed cross-layer communication stacks.

Name	Status	Improvements	Year
UCLEAH [19]	Sim	Optimum hop distance	2017
XLM [13]	Sim	Unified cross-layering stack	2006
CLAMP [14]	Sim	Data sharing with a subscription system	2007
TinyXXL [20]	N/A	Cross-layer Data exchange for tinyOS	2007
OFM [18]	N/A	Abstract functionalities for cross-layer	2010
ContikiOS Stack [15]	Tbd	Shared packet building	2007
GNRC [47][1]	Tbd	Switchable stack in the OS	2018
TSTP [21]	Tbd	Shared metadata-enriched zero-copy buffers	2018

[1]GNRC does not have cross-layer, but Riot allows communication stack exchange and multiple stacks running simultaneously.

3 IOTUS - IOT UNIFIED SERVICES

The new IoTUS framework for IoT stacks is introduced in this chapter. It is based on modules to provide shared data and functionalities; thus, a general description and examples are used to explain IoTUS's design.

IoTUS promotes information and functionalities sharing across protocol layers while maintaining most benefits of a layered design. Through efficient cross-layer sharing, IoTUS's main goal is to achieve energy efficiency, as well as a more compact memory footprint, both of which are important to accommodate IoT devices with limited capabilities. Similar to other proposals [14, 15, 20], IoTUS provides modules to standardize the way information is accommodated in packets. However, differently from other reviewed works [14, 15, 20, 21, 24], IoTUS not only provides modular cross-layer benefits but also allows protocols (in the traditional OSI-like stack) to be aware of each other procedures and optimize their tasks through a subscription process. For example, when protocols share their behavior with the stack (e.g., periodic packets and expected procedures), other protocols are aware of possible aggregations and can optimize their parameters accordingly. As illustrated in Figure 3.1(b), IoTUS is designed as a collection of service layers (or modules) which can be used directly by existing stacks, without modifications of these protocols.

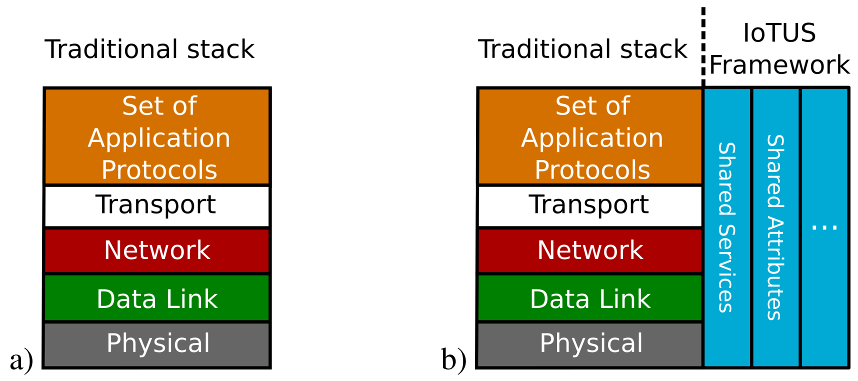


Figure 3.1 – (a) Traditional network stack; (b) Aggregating extensible IoTUS's modules.

In cross-layer architectures, centralized data storage facilitates the information exchange between layers, e.g., TinyXXL [20] uses such a storage system, and Rime [16] provides a shared packet building module. In IoTUS, a data exchange system was also implemented, which includes a packet building module.

Other IoT protocols [14, 18, 20] have some independent modules to share specific functions, e.g., *duty cycling* is often a separate service in the stack; *neighbor discovery* has its separate implementation, providing service to other layers; *topology building* can be an independent service that provides connection to other protocols; etc.

In IoTUS, the framework functions are an additional feature to the traditional layered stack (Figure 3.1). Therefore, no design change is required in the stack. IoTUS can operate alongside with off-the-shelf protocols. The process of sharing functionalities is one of the main contributions of this framework. Also, IoTUS's modules are not independent; instead, they are controlled by the protocols, i.e., a module has no access to send a packet itself, but can build them on demand.

IoTUS framework was developed for WSN and IoT networks to provide both with an architecture capable of improving protocols' performance. Energy efficiency is mostly obtained by the optimizations that protocols can achieve with IoTUS, reducing the number of packets exchanged in the network while keeping their requirements. Flash memory has an initial increment with the installation size of the framework (less than 5 kbytes), but it provides an improved Random Access Memory (RAM) usage due to the centralized non-redundant storage. Our future goal also includes optimizing Internet protocol procedures for LLN with this framework cross-layer benefits.

IoTUS's main module is called *IoTUS-Core*. It is required during compilation and runtime. For the compilation process, this core module processes each protocol in the stack and includes the other requested modules as needed.

3.1 COMPILATION STAGE

The IoTUS framework has an important process during compilation, controlled by the *IoTUS-Core*, which prepares the framework to process its services. The framework compiling directives (*Makefiles*) parses all protocols used in the stack and sets IoTUS's code to be operated according to the selected modules.

IoTUS modules are similar to dependencies in Linux's Advanced Packaging Tool (APT) [60]. They can rely on other IoTUS services, i.e., services can be split into smaller services and protocols can select exactly which one(s) to use. *IoTUS-Core* also takes care of this dependence management.

IoTUS actual implementation is composed of the following modules, classified as:

- **Mandatory modules:** includes the IoTUS-Core, Node Manager, Packet Manager, Task Manager, Network Attributes, and Events Register;

- **Optional modules**: Piggyback Service, Neighbor Discovery, and Tree Manager.

Node Manager maintains information about the node's neighbors such as addresses, ranking in the routing tree, link layer sequence number, link quality (RSSI), etc. The *Packet Manager* module provides functions to build packets; these functions can be used by any layer in the stack when protocols are adding, removing, or changing fields in their respective transmission units (e.g., packets at the Network layer, frames at the data link layer, etc). The *Task Manager* assigns the control of a module to protocols, thus ensuring synchronization between procedures and protocols.

Other modules are optional, and their utilization will depend on the protocol's functional needs. For example, the *Piggyback Service* module provides a data aggregation mechanism that can be used by protocols at different layers. Piggybacking helps achieve energy efficiency by "packaging" as much information as possible into a packet. *Network Attributes and Event Register* module concentrates information about many general network values, e.g., number of transmissions, connection quality, package drop rate, and others.

3.2 RUNTIME STAGE

At the start of a node's operation, IoTUS framework interacts with the protocols that are using its modules. This is possible because the compilation stage already recognized which protocol was available to use this framework. Thus, *IoTUS-Core* is again responsible for starting every module.

As the modules are receiving the start signal from the *IoTUS-Core*, the *Task Manager* initiates its subscription process that will assign the modules' tasks to the requesting protocols, e.g., which protocol will control the *Neighbor Discovery* task, using the shared features provided by this module.

To illustrate IoTUS's operation, let us consider an environmental monitoring application (Figure 3.2), where nodes use a basic network protocol stack composed of a Data Link protocol (e.g., ContikiMAC [27]), a static routing protocol, and the application layer protocol (which sends periodic data from sensing nodes), thus forming a tree rooted at the data sink. In addition to application-layer messages, Keep Alive (KA) control messages are periodically generated by nodes to the data sink. In this topology, application data is flowing to the sink device, since it is connected to the main controller/supervisor system.

In most existing networks stacks, packets are built based on an array of bytes, in which

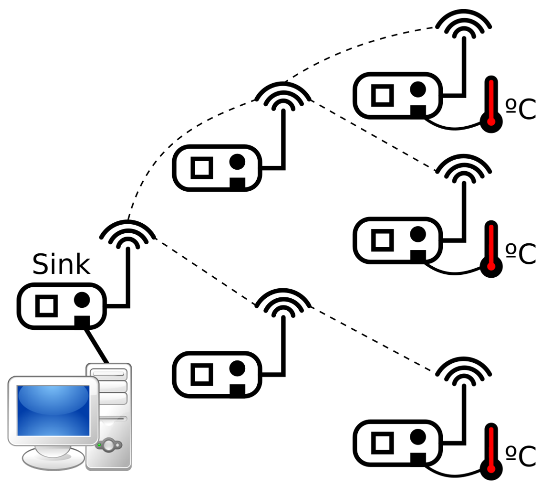


Figure 3.2 – Example of an IoT network for temperature monitoring environment.

headers are added to the payload as each layer processes packets on their way down in the stack. In IoTUS, a protocol uses information maintained by *Node Manager* to build a packet using the *Packet Manager*. During this step, additional information can also be added to the packet, such as timeout, priority, fragmentation/aggregation, etc. After the protocol finishes handling this packet, it sends a signal to the next layer, containing the reference to the packet's metadata (memory block containing the structure holding shared information, e.g., payload, destination node, timeout, etc.). In this way, every field added to the packet has a globally known format, readable across different protocols.

In the example shown in Figure 3.3, the application layer starts to build a packet and will signal the Network layer when it is done. The Network layer evaluates the information already inserted in this packet block created by the application and inserts more processed data and header, sending, by its turn, a signal to the next layer.

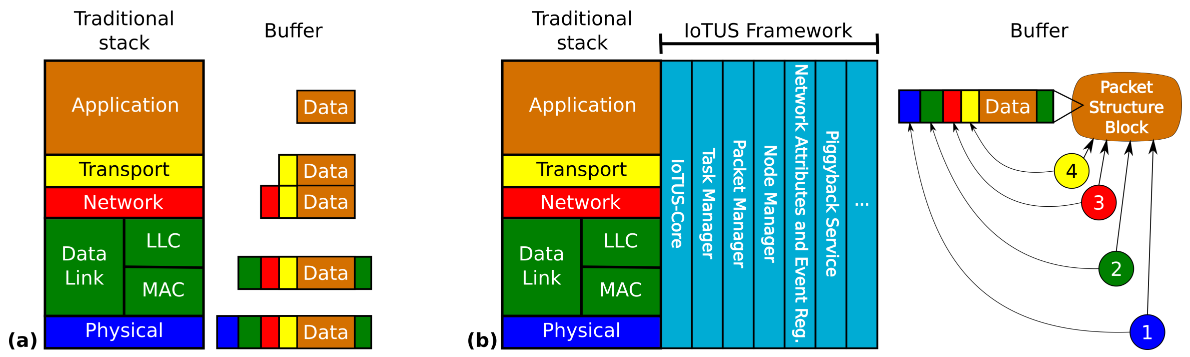


Figure 3.3 – (a) Traditional packet building; (b) IoTUS service modules - packet building.

Since the process of managing packets across layers is done using a centralized module (i.e., the *Packet Manager*), other modules such as *Piggyback Service* can use the outgoing packet to aggregate information from other layers. For example, in the case of an application message being built to be transmitted, a Keep Alive (KA) control packet can be piggybacked, which results in improved network efficiency.

As previously discussed, besides facilitating information sharing across layers, IoTUS also allows sharing of services, e.g., *Neighbor discovery* and *Tree Manager* services, which are responsible for discovering information about a node's neighbors and maintaining a routing tree, respectively. Thus, other protocols aware of these modules' operations can aggregate their requests, reducing overhead and/or improving connection speed.

To illustrate the process of sharing a service module in IoTUS (like *Neighbor Discovery*, connecting motes as shown in Figure 3.4), consider two protocols in different layers that need to find near nodes, e.g., RPL [10] in the Network layer and IEEE 802.15.4 MAC [30] in Data Link layer. In this case, the network protocol needs neighbor information to determine the best path to send/forward messages. On the Data Link layer, its protocol requires association with a coordinator device; thus, there is a registering protocol. Other protocols could be added to this example (e.g., IPv6 [8]), but for simplicity consider only RPL and IEEE 802.15.4 MAC.

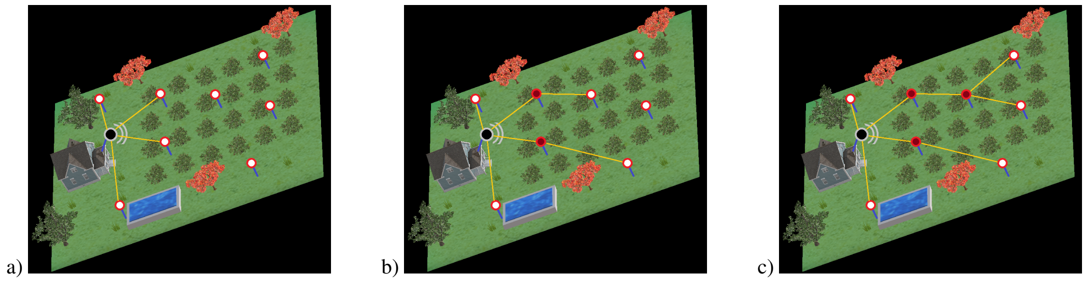


Figure 3.4 – Network protocol creating a routing path through a Neighbor Discovery (ND) procedure: a) First stage of ND; b) Second stage of ND; c) Third stage of ND.

For these two protocols doing ND procedures, consider as well that *Task Manager* already processed their requests (to control the *Neighbor Discovery* module) and assigned the lower layer IEEE 802.15.4 MAC to control it. Therefore, the Data Link protocol will determine the time for each frame exchange, as well as which type of message will be used, e.g., periodic broadcasts (beacons), register request, register answer, and others. In that way, the Network protocol will be aware of which type of packet it can optimize in its discovery protocol, by aggregating with the Data Link layer.

The *Neighbor Discovery* and *Tree Manager* modules have previously determined the type of messages that are generally used by protocols; therefore, this framework allows protocols from different layers to aggregate their messages into one single packet, since association exchanges are usually similar to each other. In this work, only one *Neighbor Discovery* module was implemented, but other association strategies can be implemented by other modules and executed along with this one.

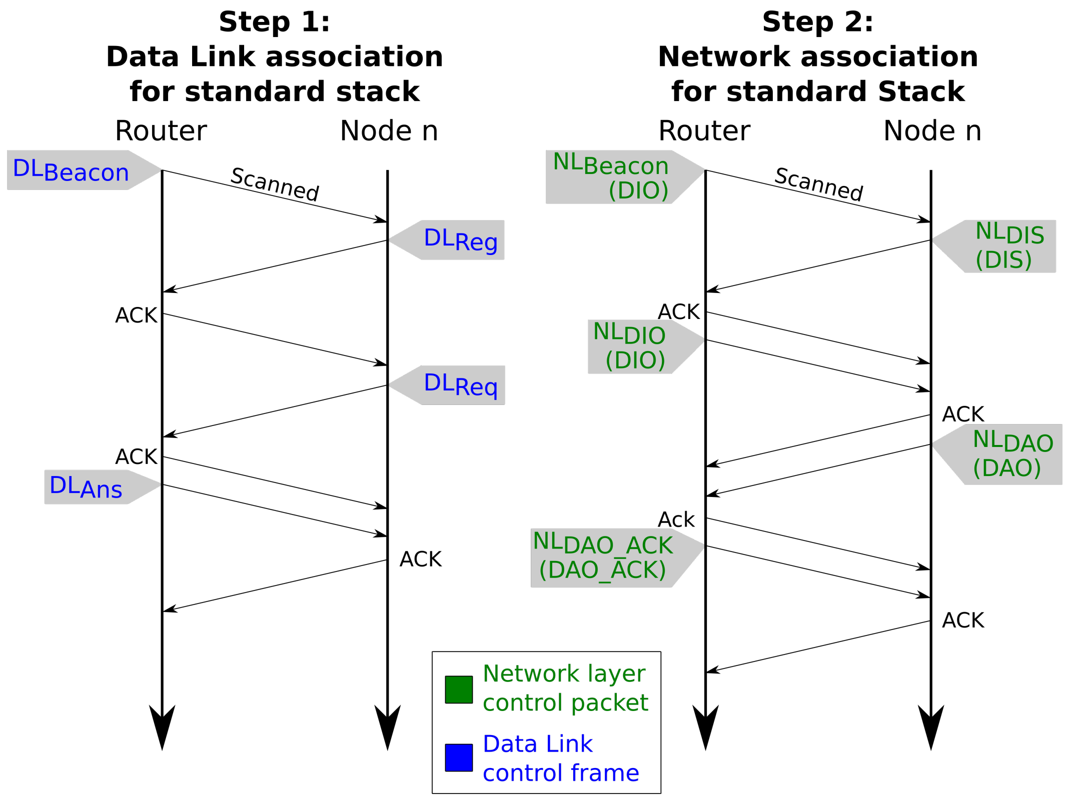


Figure 3.5 – Time diagram of exchanged control packets in a standard stack registering (2nd layer) and neighbor discovery (3rd layer) process.

Figure 3.5 shows how two nodes would behave inside the network. First, an exchange[1] of Data Link control packets would happen: router node sends periodic broadcasts (DL_{Beacon}) followed by the connecting node starting the association with a DL_{Reg} and polling the answer (DL_{Ans}) with DL_{Req}. The second step of exchanges is given by the Network's ND control packets: broadcasts from the router‘s Network layer (NL_{Beacon}) followed by the connecting node starting network layer's association (NL_{DIS}); router replies with its network information (NL_{DIO}), and the connecting node sends its second step of association packets (NL_{DAO}) followed by the router's answer ($NL_{DAO-ACK}$). This process is repeated for every device association.

On the other hand, using the IoTUS framework, where the registering/neighbor discovery procedure can be aggregated with its *Neighbor Discovery* shared service, those control frames could be put into one single message (as long as the total packet length is within

[1]Packets are replied with an ACK by the Data Link layer

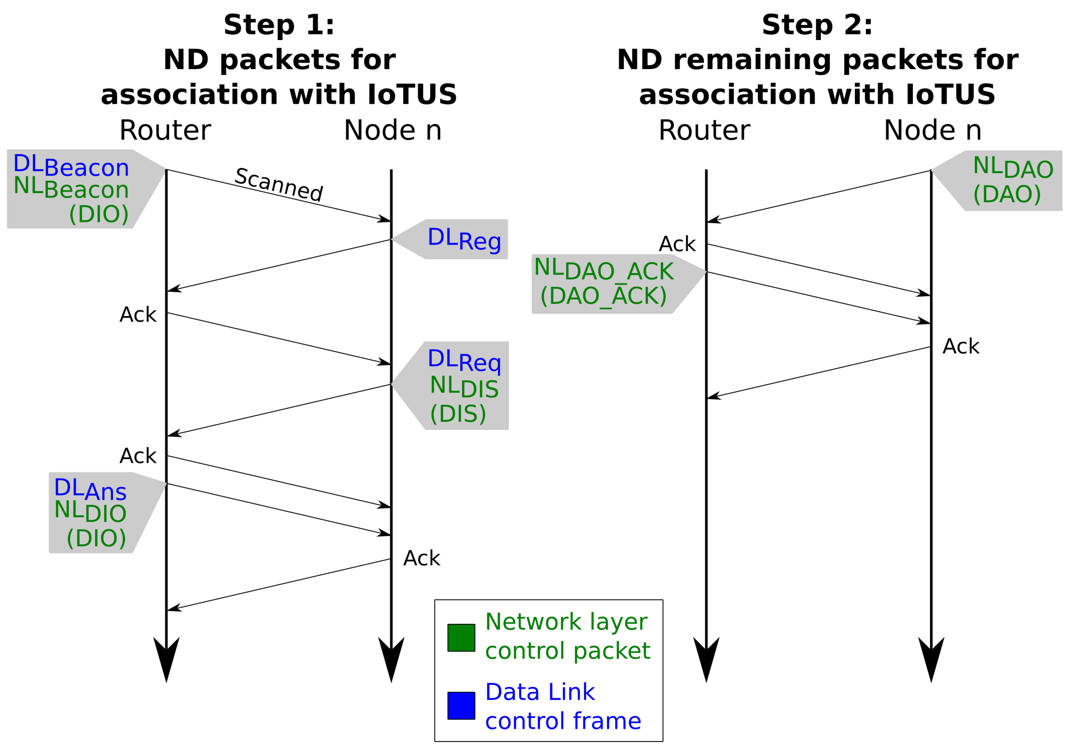


Figure 3.6 – Time diagram of control packets exchanged with the new IoTUS framework in a registering (2nd layer) and neighbor discovery (3rd layer) process.

maximum size). In this way, Figure 3.6 shows the same process done between two nodes connecting to each other and using the IoTUS framework.

3.3 CHAPTER'S CONCLUSION

The IoTUS framework is presented in this chapter using examples of its features. Also, the interaction demonstrates how this framework attains energy efficiency with data and functionality sharing. For that, a monitoring application is exemplified to compare the standard stack procedures with protocols of a stack using IoTUS.

IoTUS was developed to improve energy consumption by allowing protocols to synchronize and/or aggregate their procedures. Hence, with more protocols and more complex tasks, better memory usage, code reduction, and improved network lifetime, in general, are expected. However, IoTUS comes at the cost of processing time and initial additional code. This processing cost caused by IoTUS framework can increase CPU consumption, but the energy saved in radio operations is expected to have more impact on the overall network consumption.

4 IOTUS: DETAILED DESCRIPTION

In this chapter, the new framework is explained in details, and implementation aspects are exposed.

IoTUS framework was developed to provide better energy efficiency and lower memory usage in low computational power devices. The actual implementation of IoTUS was done in ContikiOS; therefore, it is in C Language. In this system, *Makefiles* are files used by the compiler, which contains instructions for the compilation process. Since IoTUS has a compilation stage, it uses *Makefiles* to configure its code according to the protocols in use by the stack. Thus, for each protocol, two files are expected: *Makefile* and *Dependencies*.

For example, consider a stack being added by IoTUS. If the X protocol was implemented to use IoTUS framework, then this protocol will provide (in its folder) the files, "Makefile.X_v1.0.0" and "Dependencies.X_v1.0.0". Versioning for IoTUS is not yet implemented. Hence, versions of modules and protocols are managed only by their name.

After the compilation stage, the device is ready to execute the code during its *runtime* stage, as shown in Figure 4.1. However, many modules will be requested by protocols at this stage, and so a synchronization system has to be used. In this case, IoTUS framework has the *Task Manager* modules receiving protocol‘s requests and assigning the task.

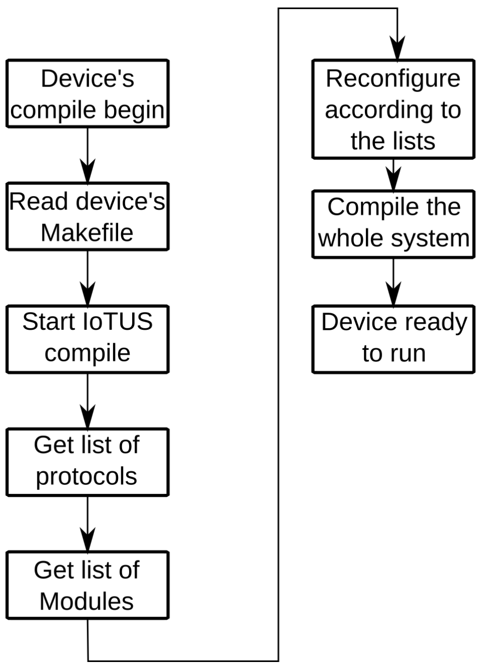


Figure 4.1 – Compiling process for IoTUS framework.

Hence, for a better comprehension of IoTUS framework, the remainder of this chapter describes in details all modules that have been currently implemented, plus some additional tools, e.g., *Safe-printer*, *Addresses manager*, etc. More details over the implemented structures can also be found in Appendix B.

4.1 IOTUS-CORE

This module can be considered the main service in *IoTUS-Core* because it has functions during the compilation stage and at runtime. Therefore, this section is divided into two, explaining this module's operation for both of these stages.

4.1.1 Compilation stage

As previously mentioned, the compilation stage is composed of *IoTUS-Core* reading each protocol's *Makefile* and *Dependencies*. The protocols without these files should not provide information to IoTUS framework, although they still have read access to the framework.

Each module/service also has its own *Makefile*, which can request another module in the framework to be installed, creating the dependency system. This process of reading *Makefiles* both from protocols and requested modules can be seen in Figure 4.2.

This way, as the protocols know exactly which module they are using, the interface functions and Application Programming Interface (API) are known, allowing for every information in that module to be understood by protocols across the stack using the same module.

The *Makefiles* read by IoTUS contain codes as shown in Listing 4.1, which informs of its name, as well as of the possible subfolders that have to be compiled.

Similarly, the *Dependencies* files are also processed by *IoTUS-Core*, which provides information to the framework in the format shown in Listing 4.2.

With all files processed, the framework can then reconfigure its code to correctly operate all the requested modules, allowing the system to continue to the next stage.

```
IOTUS_PROTOCOL_NAME = protocol_X

# If more folders need to be compiled, just add them to the following line, like
# THIS_SUB_MODULES := $(THIS_MODULE_FOLDER)/<NEW FOLDER>
```

Listing 4.1 – Makefile used by *IoTUS-Core* during compilation stage.

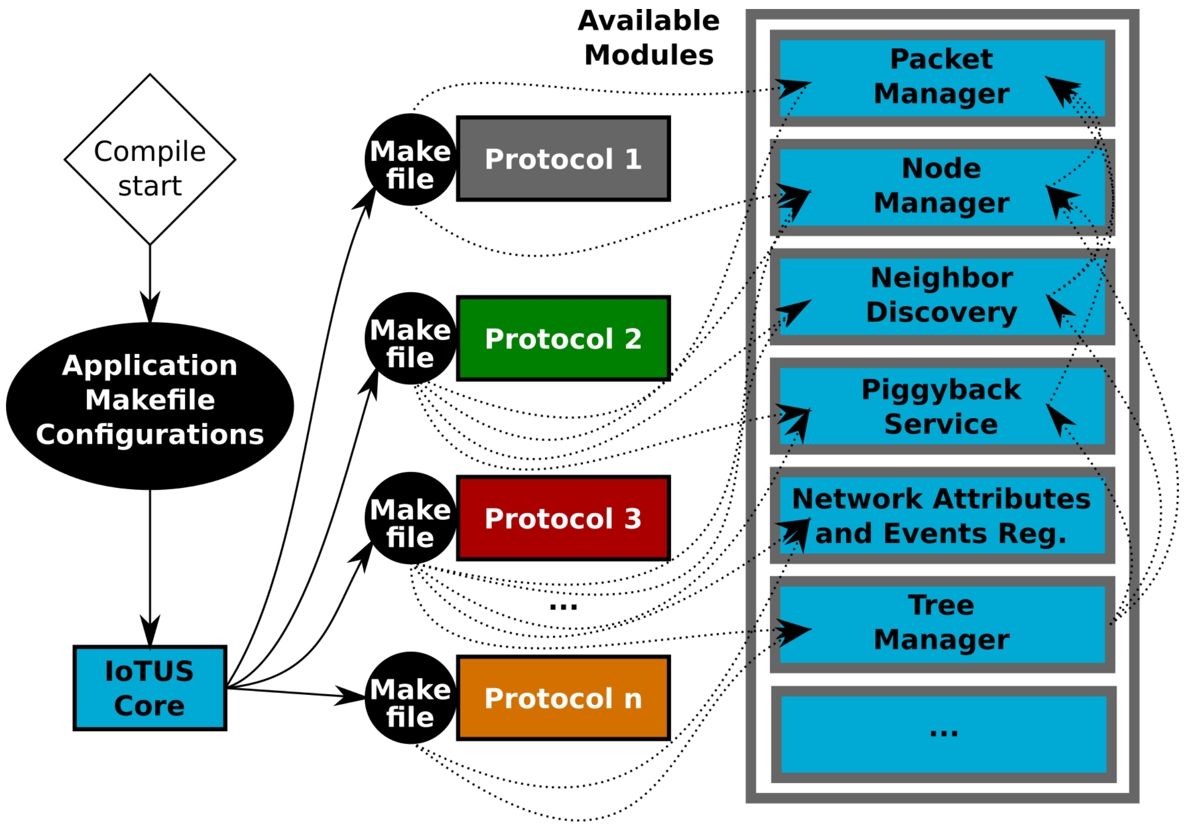


Figure 4.2 – Example of *IoTUS-Core* installing requested modules from different protocols.

```
# This file should contain all the dependencies that this
# service requires to work.

IOTUS_SERVICE_DEPENDENCIES_LIST  = packet_v1.0.0 nodes_v1.0.0 piggyback_v1.0.0
IOTUS_SERVICE_DEPENDENCIES_LIST += neighbor_discovery_v1.0.0
```

Listing 4.2 – Example of a *Dependencies* file used by *IoTUS-Core* during the compilation stage.

4.1.2 Runtime stage

In this stage, stack, framework, and modules are being executed together. Thus, synchronization is necessary, which is done by the *IoTUS-Core* in association with the *Task Manager* (further explained below). Since *IoTUS-Core* is responsible for starting all installed modules at runtime, it is expected that the main application's code calls *IoTUS-Core* at the beginning of a device's operation.

Within the initialization, the protocols using this framework must wait for two signals from *IoTUS-Core*. The first signal is the *initializing* command, where protocols will initiate their setups and requests with modules, e.g., subscribing to control some task in *Task Manager*. The second signal is the *start* command, where protocols can poll for results in modules and/or start their own task, e.g., polling the subscription process done in *Task Manager*.

This initializing procedure done by *IoTUS-Core* in association with the *Task Manager* can be seen in Figure 4.3.

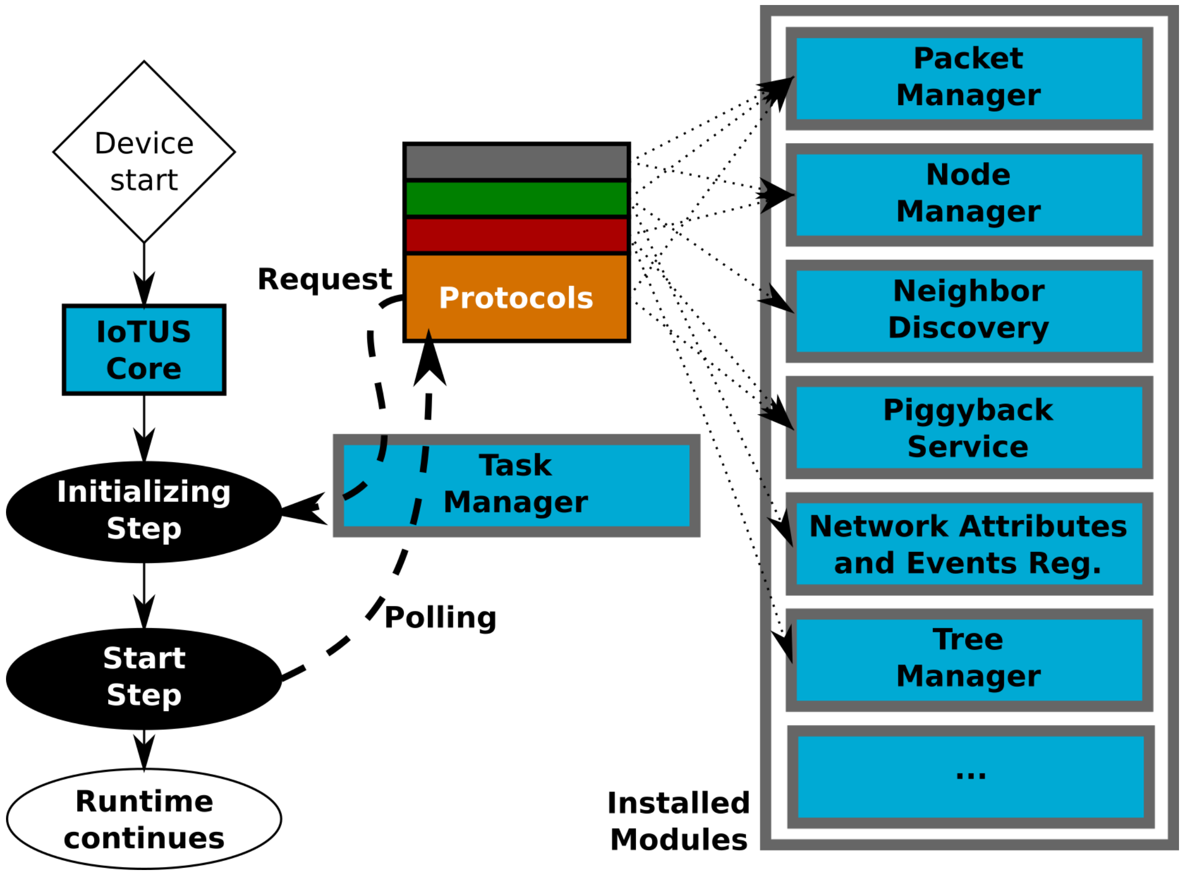


Figure 4.3 – *IoTUS-Core* initializing and starting protocols and their requested modules.

At runtime stage, *IoTUS-Core* provides the $Demanding\_Period()$ functionality to the stack with IoTUS, where its protocols can share their time-critical tasks with each other, e.g., real-time tasks as a periodic coordinator broadcast. This is necessary for low power computational devices without multi-task capacity.

One example of time demanding task is writing on the serial port (*print* function). Therefore, another tool provided by IoTUS called *Safe-Printer* can store messages in memory and print them during a non-critical period, using the $Demanding\_Period()$ function. In the IoTUS implementation, the *Safe-Printer* tool has only a few bytes of storage, using a circular buffer ring to store data to be printed at a safe time. Its use is not mandatory, but it is convenient, as a debugging tool.

As described in other modules, *IoTUS-Core* also simplifies API for some operations done by other mandatory modules, e.g., *IoTUS-Core* provides functions for the application layer that integrates *Packet Manager* and *Node Manager*.

4.2 NODE MANAGER

Many information extracted by different layers can be attached to a given neighbor node; however, protocols usually would not share this data. This module centralizes the data gathered on neighbors. It creates a standardized way to share a determined set of information, thus allowing shared neighbor data to be stored in structures of blocks, retrieving its block by reference pointers or search by address number.

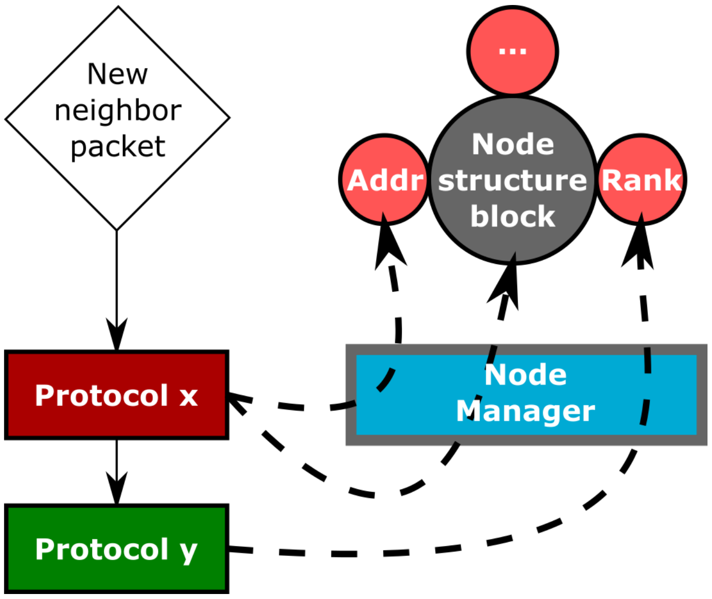


Figure 4.4 – Protocols receiving a packet from a new neighbor, parsing, and sharing its information.

As seen in Figure 4.4, considering that a new neighbor packet is received, protocols can share its data using the *Nodes Manager* while reading and extracting the packet's content. This module has most of the fields generally expected in an IoT environment, e.g., the link quality extracted by the physical layer can be attached to the node structure block, along with the address given by the second layer and the rank given by the third layer (how distant, in number of hops, the node is from a sink node). Such an approach reduces memory usage, redundancy, and improves cross-layer decisions.

4.3 PACKET MANAGER

Similar to *Node Manager*, *Packet Manager* is responsible for concentrating most of the packet information into one structure, as represented by the small circles being added to the big packet block in Figure 4.5. This means shared data is located in a single place, where every layer can understand what is being added to the header. Moreover, the *Packet Manager* works on top of the *Node Manager* and saves memory space by pointing to its structure block instead of copying all information into each packet block.

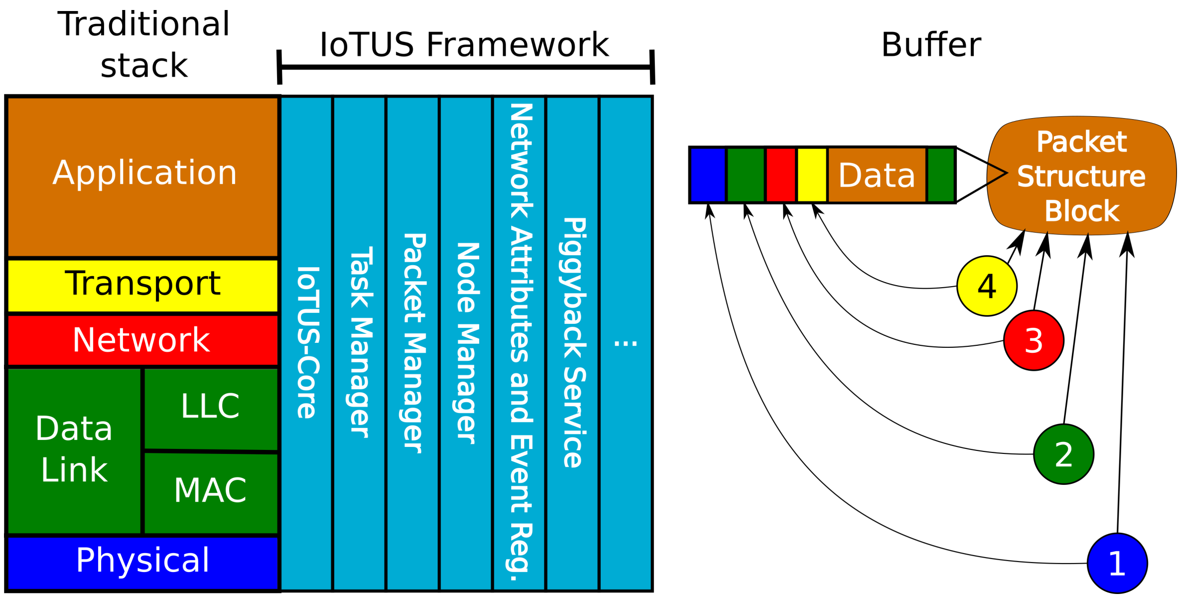


Figure 4.5 – IoTUS *Packet Manager* used to build an application message.

Many packet parameters, like source and destination addresses, are available in a standardized manner. This information is also available across the layers when the packet is being built. IoTUS framework allocates the whole collection of possible data dynamically – allocating only the necessary memory for the information attached through linked lists – whereas Rime/ContikiOS only operates statically.

Thus, some of the standard fields held by this packet block are:

- **Data**: The buffer carrying the packet payload;
- **Parameters**: A binary set of defined flags. Some of them are:
  - Wait for ACK: if the packet needs an acknowledgment after transmission;
  - Aggregation: if the packet supports the *Piggyback Service* functions;
  - Fragmentation: if the packet can be split and transmitted;

- **Timeout**: The maximum delay the packet can wait until transmission;
- **Final destination node**: The final node for which the packet should be sent to;
- **Next destination node**: The next hop that the packet should take until its final node;
- **Previous source node**: The last neighbor node that sent the packet;
- **Additional information list**: A generic linked list to hold a block of information. With this field, protocols can share data that they extract from the packet. These blocks have a defined header so that other protocols can also interpret the information.

As can be seen in the default fields in the packet block, along with the *Node Manager* module, the *Packet Manager* references the source and destination nodes using this new system; thus, it creates an integration that facilitates other services' operations, like aggregation of packets (further explained in the section *Piggyback Service*).

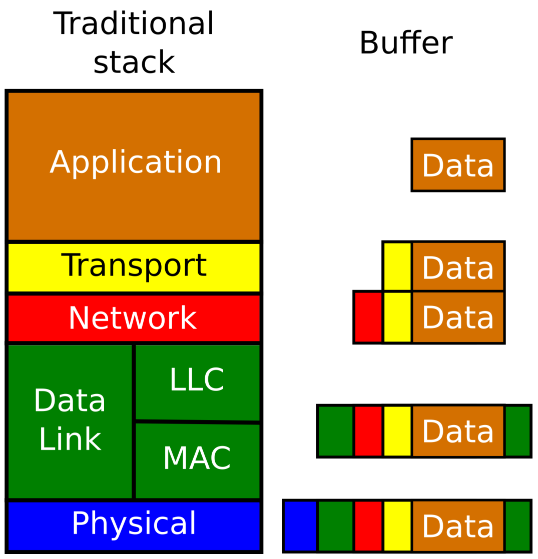


Figure 4.6 – Traditional network stack building an application message through encapsulation.

A side-by-side comparison between a traditional layered stack and the same stack extended with the IoTUS framework will be made next. As in the traditional way illustrated in Figure 4.6, messages are sent down the stack using abstract functions and encapsulating headers in the buffer. In Figure 4.5, the same stack with IoTUS framework builds the message using dynamic packet structure blocks. The *Packet Manager* creates a block containing the buffer that can hold headers and a small amount of information attached to it. Each block contains well defined and known information, which represents the packet fields.

Hence, after the application layer signals the messaging procedure in the shared layer, the packet structure block (buffered) is reserved, and a signal is sent to the lower layer. Layers below will set structure blocks and attach them to the packet structure block along with their headers. This process is represented by the circles numbered with the layer rank (4 to 1). At the physical layer, all information attached is readable, and the header is ready.

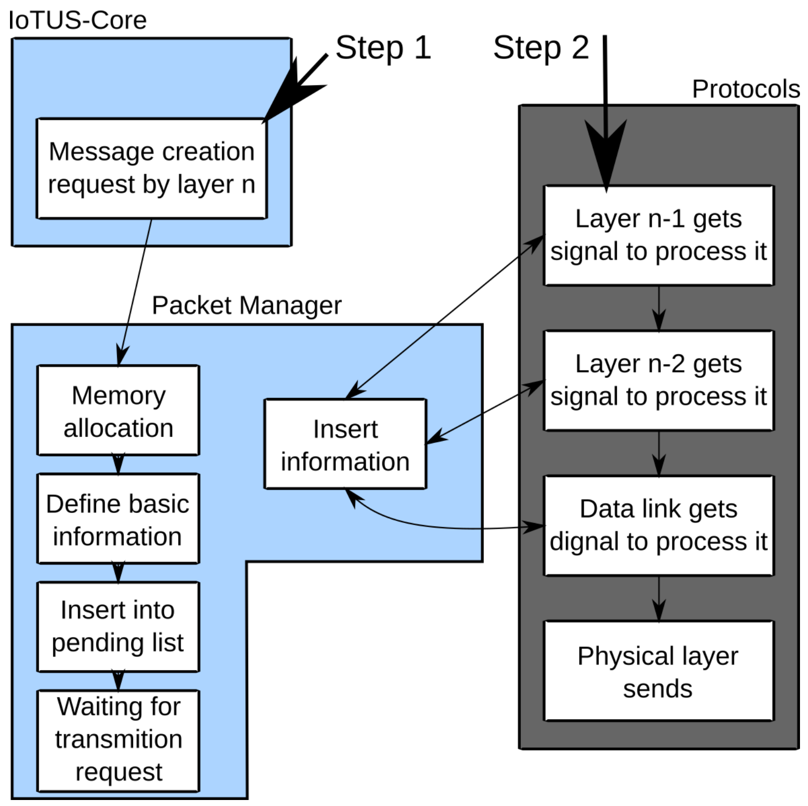

IoTUS-Core
Step 1
Step 2
Protocols
Message creation request by layer n
Layer n-1 gets signal to process it
Packet Manager
Layer n-2 gets signal to process it
Memory allocation
Insert information
Define basic information
Data link gets dignal to process it
Insert into pending list
Physical layer sends
Waiting for transmition request

Figure 4.7 – Message construction with IoTUS.

The creation of a packet container in the IoTUS system can be seen with more details in Figure 4.7, where Step 1 represents a calling from a protocol to the message creating function, defining some basic information (payload, destination node, parameters, timeout, and others). Continuing, lower layers will get information (Step 2) to process this packet. This way, packets are always stored in a list of dynamic containers to which all layers have access. IoTUS's services generally use pointer references to other services' block structures, which helps to keep the information up to date.

4.4 TASK MANAGER

This manager module assigns tasks to each running protocol. Since IoTUS framework proposes shared services and functionality, it would be possible that two or more protocols using the same service would request it to start a procedure more than once; thus, synchronization is necessary. However, it does not stop any protocol from doing a redundant operation by itself; instead, it simply informs all layers which service will be controlled by each protocol, allowing the redundancy if necessary.

The process of requesting a task is done at the start-up of the device by each protocol using this framework. As illustrated in Figure 4.3, *IoTUS-Core* provides two steps for the protocols using this framework; they can then subscribe and poll for the assigned process provided by the *Task Manager*. Inside this module, priorities are usually given to the protocols located in lower layers, i.e., protocols in physical and data link layers have higher priority than network and application layers. This also solves some issues with addressing, packet aggregation, and other tasks.

In the case of addressing (a procedure which recognizes the neighbor's addresses and/or sets its address) most of the data link protocols have their own methods. Thus, they already have to use the packet header to insert these addresses. A network layer that checks that the data link layer is already addressing can use this feature to synthesize their addressing system accordingly. In the traditional layered stack, this process is done through cross-layer access, e.g., 6LoWPan protocol [7] is generally used with IEEE 802.15.4 [30] and uses its addressing system to generate the IPv6 [8] address.

Another example of a task is the aggregation service. In many cases, it is done by the network layer, but if the data link layer has some procedures that would benefit from aggregating their control packets, its protocol can use the shared service and optimize its control packets as well.

As mentioned previously, at initialization time, every protocol and service can request the core to be in charge of some of these chores, avoiding that more than one module replicates the same job. This list can also be extended according to the application necessity; therefore, it is a layer synchronization approach that can serve many cases.

For future protocols' implementations using IoTUS's enhancements, it is expected that overhead can be reduced by a better synergy through layers, i.e., reducing header sizes and avoiding duplicated packet fields. For this reason, *Task Manager* provides other IoT task assignments, for example:

- Insert the source address into the packet header;
- Insert the final destination address into the packet header;
- Insert the previous sender address into the packet header;
- Insert the next receiver address into the packet header;
- Fragment packets;
- Control the *Neighbor Discovery* service;
- Control the *Tree Manager* service.

4.5 PIGGYBACK SERVICE

IoTUS provides data aggregation through its *Piggyback Service* module, which is very important to reduce energy consumption by reducing network traffic. Protocols can create piggyback pieces and set defined parameters like timeout, destination address, and others. Furthermore, if the timeout expires, *Piggyback Service* will signal the callback function of the block creator, so that the protocol sends that data as soon as possible. However, if conditions are matched, the piggyback container will be aggregated (with small compact headers) into the outgoing packet.

Piggyback Service uses *Node Manager* and *Packet Manager*. The main conditions to insert a piggyback block into a packet are:

- There is an outgoing packet;
- The packet has to be flagged as allowing aggregation by its creator protocol;
- The packet is addressed to the same next router or same final destination node.

Data inserted into *Piggyback Service* to be aggregated become a Piggyback piece (PBp) and will wait a maximum amount of time equivalent to the piece's selected timeout. The timeout can vary as intend by a protocol which is able to poll shared information of other protocols in the stack using IoTUS's modules. The delay over hops is impacted only by the router's processing time and their duty-cycling. Therefore, even if intermediate routers increase a packet size with more piggyback pieces, the relayed packets are not held by this module and thus do not impact on latency.

This service creates control headers into the packet that allow separate transmissions to have shared destinations in the network. Thus, many layers can rely on it to have their control packet optimized, as is the case of DAO packet in RPL.

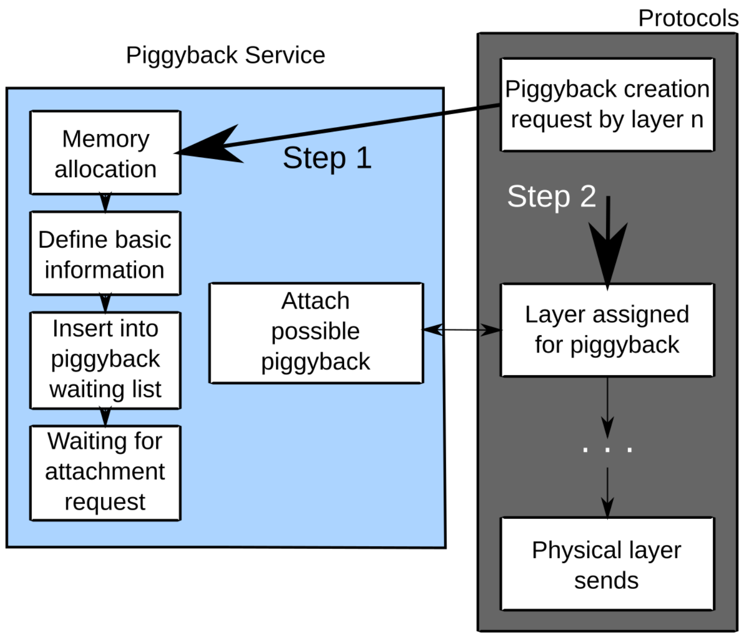

Protocols
Piggyback Service
Memory allocation
Step 1
Piggyback creation request by layer n
Step 2
Define basic information
Insert into piggyback waiting list
Attach possible piggyback
Layer assigned for piggyback
Waiting for attachment request
. . .
Physical layer sends

Figure 4.8 – IoTUS Piggyback Service.

The process of creating a piggyback block is given in Figure 4.8. Protocols would start by a similar process as the *Packet Manager*, allocating resources with specific functions of the module (Step 1 of Figure 4.8). Later, if any protocol was previously assigned by the *Task Manager*, it will try to attach possible piggyback blocks into a packet being built, aggregating its information (Step 2 of Figure 4.8).

Figure 4.9 illustrates the Piggyback piece (PBp) being created by "Protocol y", which represents Step 1 in Figure 4.8. When “Protocol x" creates a packet, Headers (HDrs) are attached. At some assigned layer, *Piggyback Service* is called and attaches the pieces that match this condition, representing Step 2 in Figure 4.8. Bellow the dashed line, the process is represented by a node using this procedure and delivering the packet to its destination, where the piggyback piece is detached again and delivered to the target node’s protocol.

The same process of "Protocol x" creating a packet that adds the piggyback piece of "Protocol y" can be seen in Figure 4.10. In this case, the piggyback pieces and the packet blocks are represented inside the list that each service holds, the dashed line represents the timeline, and the squares on bottom describe the events following the sequence of circles aligned above them.

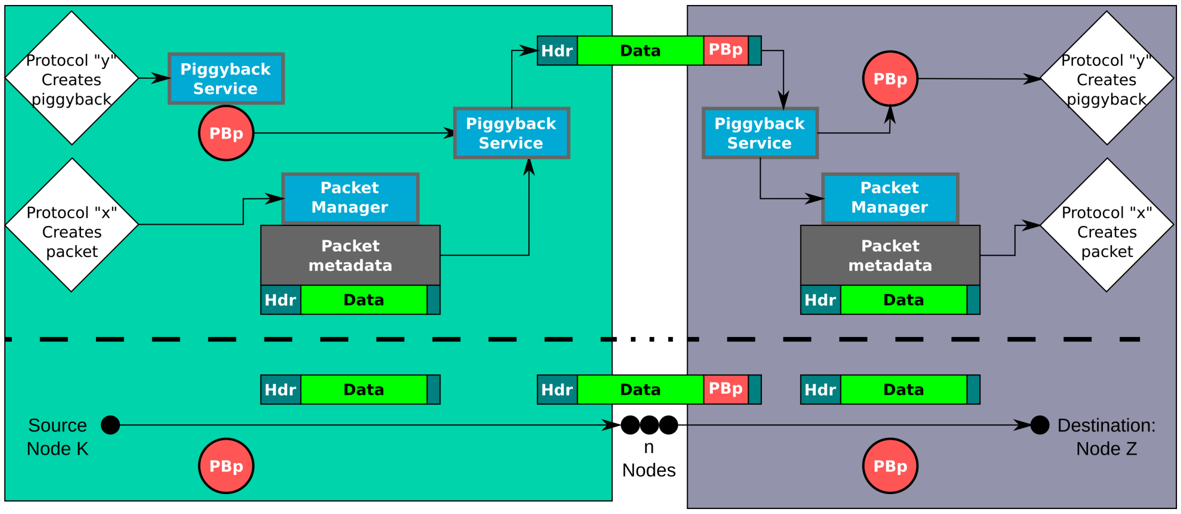


Figure 4.9 – Piggyback assembly, transmission, and dis-assembly.

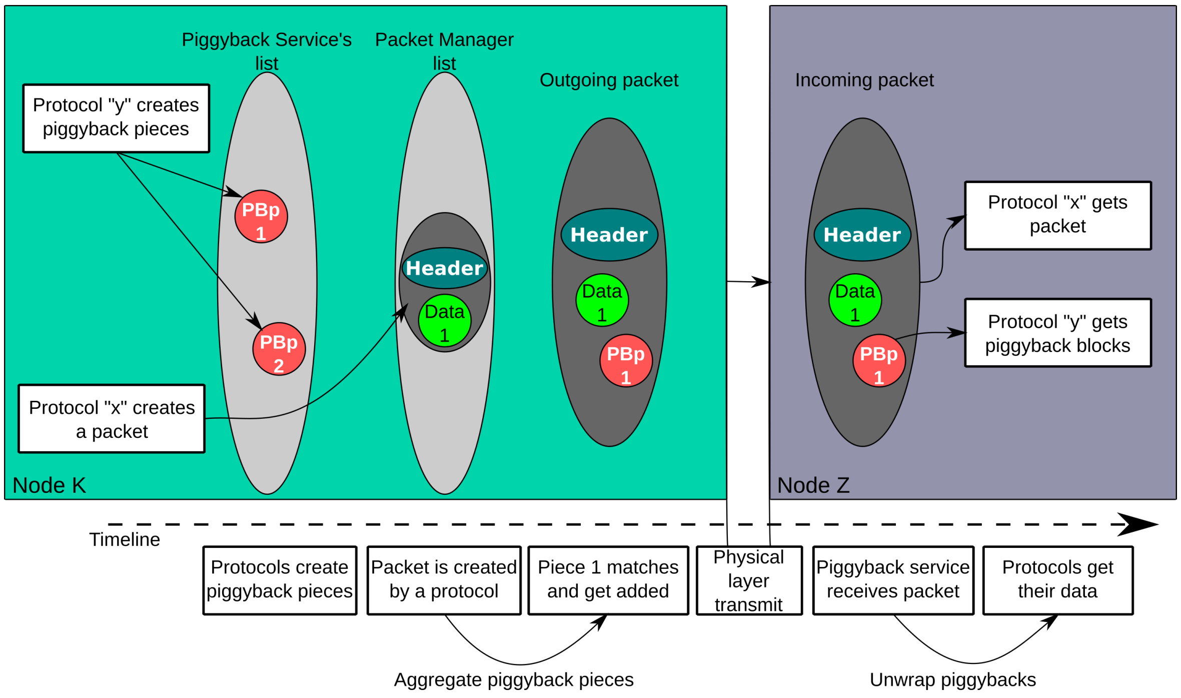

Piggyback Service's list
Packet Manager list
Outgoing packet
Incoming packet
Protocol "y" creates piggyback pieces
PBp 1
PBp 2
Header
Data 1
Header
Data 1
PBp 1
Header
Data 1
PBp 1
Protocol "x" gets packet
Protocol "y" gets piggyback blocks
Protocol "x" creates a packet
Node K
Node Z
Timeline
Protocols create piggyback pieces
Packet is created by a protocol
Piece 1 matches and get added
Physical layer transmit
Piggyback service receives packet
Protocols get their data
Aggregate piggyback pieces
Unwrap piggybacks

Figure 4.10 – Piggyback Service used by different protocols to improve transmission overhead.

4.6 NEIGHBOR DISCOVERY

The Neighbor Discovery service provides the aggregation of messages usually exchanged between devices in their association procedure. It is composed of 5 types of Neighbor Discovery Control Command (NDCC): ND-beacons (broadcasts), ND-Request, ND-Poll, ND-Answer, and ND-Confirm. Each type of message represents a step of this association process that can be used by different layers simultaneously. Some examples of standard protocols that have similar messages are IEEE 802.15.4 MAC [30] and RPL [10].

The ND module's control is assigned by the *Task Manager*; thus, only one protocol will define when to send these ND messages. The controlling protocol can choose which NDCC it will need. Other unused NDCCs can still be used by other protocols, as needed.

In the *Neighbor Discovery* service, protocols have a function that allows their piece of control message to be aggregated into one of those 5 types of NDCC. The pieces are aggregated in such a way that they can be delivered and correspondingly restored at the destination node.

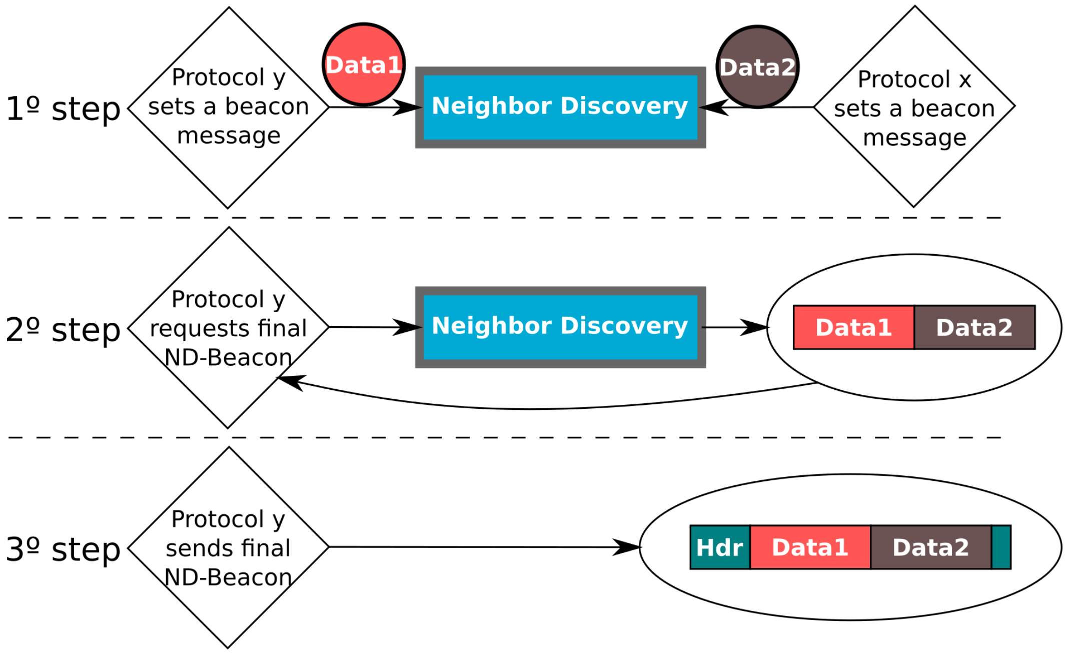

1º step
Protocol y sets a beacon message
Data1
Neighbor Discovery
Data2
Protocol x sets a beacon message
2º step
Protocol y requests final ND-Beacon
Neighbor Discovery
Data1
Data2
3º step
Protocol y sends final ND-Beacon
Hdr
Data1
Data2

Figure 4.11 – *Neighbor Discovery* service: beacon message creation and transmission.

To exemplify these operations, consider Figure 4.11, where two protocols, "x" and "y", need to send periodic broadcast packets (beacons) to their neighbors. Hence, both of them will set their data (first step) into the ND module. If "Protocol y" was assigned by the *Task Manager* to control the *Neighbor Discovery*, then it will request the final aggregated ND-

Beacon message to be sent as a beacon. The final beacon can, therefore, contain more than one data or control command from different protocols.

This process of aggregating control message pieces and delivering them at the destination node is similar to the *Piggyback Service*. As such, it can improve network energy efficiency by reducing network traffic. Furthermore, *Neighbor Discovery* can generally shorten the association time between devices.

4.7 TREE MANAGER

The tree building process is usually present in the third layer. However, the re-configuration of a network topology can impact other layers too, causing disconnections and inefficiency. Therefore, sharing these alterations through the stack can help other protocols to improve their procedures, mainly if they use the topology to optimize medium access.

Tree Manager provides functions to store and share data about the tree topology and its connections. For example, it can store which node is the root (sink), the network connection status, and so on.

The control packets of *Tree Manager* are based on those existent in the *Neighbor Discovery* module. The main goal of this module is to provide tools that allow RPL [10] to share its information with other protocols.

4.8 CHAPTER'S CONCLUSION

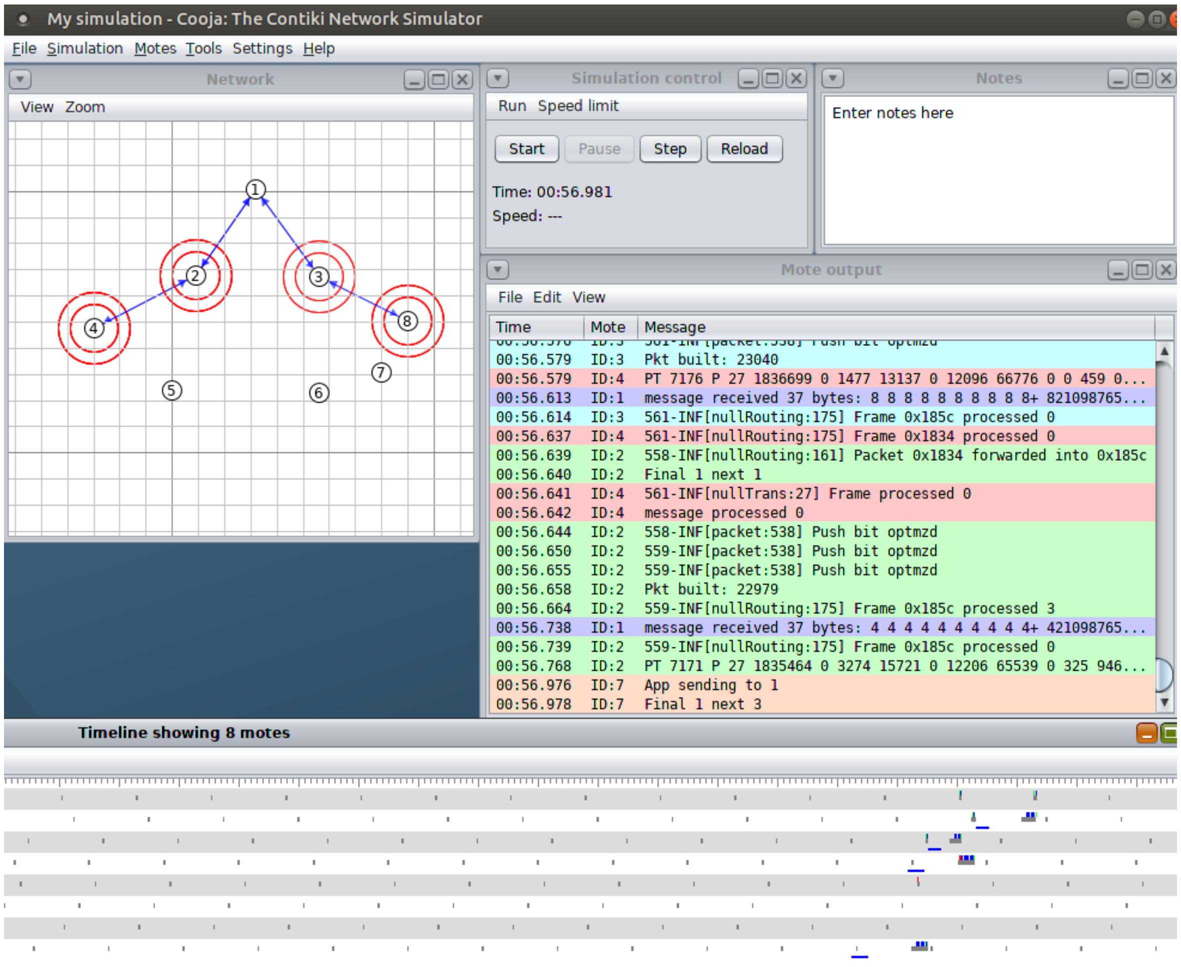
In this chapter, IoTUS is described in details, presenting its developed implementation and design. The modules like *Packet Manager*, *Node Manager*, and *Tree Manager* allow data sharing by protocols in a stack. Meanwhile, modules like *Piggyback Service*, *Neighbor Discovery*, and *Tree Manager* allow functionality sharing in the stack. Hence, the description of their procedures is fundamental to understand how they interact with protocols and the stack.

IoTUS was implemented in ContikiOS using the C language. As well, it is open-source, and its code is available online; thus, Appendix C contains details about the software and how to access it.

5 METHODOLOGY

This chapter exposes each procedure to evaluate IoTUS framework, as we selected a simulator, defined test scenarios, performed simulations, and obtained measurements.

We evaluated IoTUS using the Cooja-ContikiOS [22, 16] simulation/emulation platform. We used Cooja (Figure 5.1) for the following reasons: first, it provides an experimental platform specifically designed for wireless networks, well suited for capability-constrained wireless sensor networks and IoT networks. The advantage of using the simulation for a new design is to run a controlled environment and so obtain reproducible experiments, besides a helpful Integrated Development Environment (IDE). Cooja conveniently includes an implementation of the ContikiOS [16] and its Rime stack [15].



```
Time       Mote  Message
00:56.579  ID:3  Pkt built: 23040
00:56.579  ID:4  PT 7176 P 27 1836699 0 1477 13137 0 12096 66776 0 0 459 0...
00:56.613  ID:1  message received 37 bytes: 8 8 8 8 8 8 8 8 8 8+ 821098765...
00:56.614  ID:3  561-INF[nullRouting:175] Frame 0x185c processed 0
00:56.637  ID:4  561-INF[nullRouting:175] Frame 0x1834 processed 0
00:56.639  ID:2  558-INF[nullRouting:161] Packet 0x1834 forwarded into 0x185c
00:56.640  ID:2  Final 1 next 1
00:56.641  ID:4  561-INF[nullTrans:27] Frame processed 0
00:56.642  ID:4  message processed 0
00:56.644  ID:2  558-INF[packet:538] Push bit optmzd
00:56.650  ID:2  559-INF[packet:538] Push bit optmzd
00:56.655  ID:2  559-INF[packet:538] Push bit optmzd
00:56.658  ID:2  Pkt built: 22979
00:56.664  ID:2  559-INF[nullRouting:175] Frame 0x185c processed 3
00:56.738  ID:1  message received 37 bytes: 4 4 4 4 4 4 4 4 4 4+ 421098765...
00:56.739  ID:2  559-INF[nullRouting:175] Frame 0x185c processed 0
00:56.768  ID:2  PT 7171 P 27 1835464 0 3274 15721 0 12206 65539 0 325 946...
00:56.976  ID:7  App sending to 1
00:56.978  ID:7  Final 1 next 3
```

Figure 5.1 – Illustrative Cooja simulator IDE screen.

ContikiOS's protocol stack already includes cross-layer operations (*Packetbuf*) and a non-standard Radio-Duty Cycle (RDC) layer, which is not always easily portable to other

MAC protocols (e.g., IEEE 805.15.4 MAC [30]). However, new IoT stacks still maintain the traditional layered design, like Riot [47]. Using the traditional layered protocol stack not only makes it easier to be analyzed but also easier to be reproduced in other Operational Systems. Therefore, we focused the comparison of a traditional stack with its layered operations and IoTUS framework used in the same stack.

In this work, we adapted ContikiOS's protocol stack into a traditional stack composed of only Physical layer (and its framer sub-layer), MAC layer, Network layer, and Application layer. Transport layer protocols, although necessary in IoT systems, were not yet implemented. The ContikiOS *Packetbuf* was kept only in the traditional stack running without IoTUS but removed when using IoTUS framework. For convenience, the adapted traditional protocol stack is also called Rime stack, although the original Rime consists of many sub-layers and network functionalities. This adapted stack accepts any protocol in its layers, as the ContikiMAC, RPL, IEEE 802.15.4 MAC, etc.

Hence, we used the adapted Rime stack as a reference to compare with IoTUS. Additionally, codes that run on Cooja-ContikiOS can be directly ported to real devices running ContikiOS. Real testbeds accessible to the research community, e.g., FlockLab [61], can also be used to test our code. We developed IoTUS[1] under ContikiOS [16] for the TMote Sky [44] device emulated within Cooja, as shown in Figure 5.1.

5.1 ENERGY CONSUMPTION VALIDATION

To validate the results obtained using the simulation tool, we developed theoretical models that describe the energy consumption of a TMote Sky device. The theoretical model considers the parameters and timings resulted from the implemented version of its respective protocol, but all theoretical results are calculated independently from any simulation, using only the equations developed in Chapter 6.

Initially, a single node was modeled considering its power consumption specification, which is summarized in Table 5.1.

In the sequence, we increased the theoretical model complexity by adding more devices and considering them executing the ContikiMAC [27] protocol. This extended model is then used to calculate the performance of a linear topology (Figure 5.2), where the first node is the data sink, the last node generates messages, and intermediate devices only relay this

[1]IoTUS is available at <`https://github.com/Vinggui/contiki-IoTUS.git`>.

Table 5.1 – Energy consumption by states of TMote Sky.

State	Current
micro-controller Active (No Radio)	$1.8\,\mathrm{mA}$ @$1\,\mathrm{MHz}$, $3\,\mathrm{V}$
micro-controller sleep (No Radio)	$5.4\,\mu\mathrm{A}$
Reception	$18.8\,\mathrm{mA}$
Transmission	$17.4\,\mathrm{mA}$ ($0\,\mathrm{dBm}$)
Idle	$18.8\,\mathrm{mA}$
Sleep	$0.426\,\mathrm{mA}$

message towards the sink node. This topology is important to evaluate performance over multiple hops.

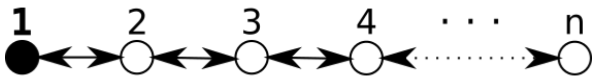


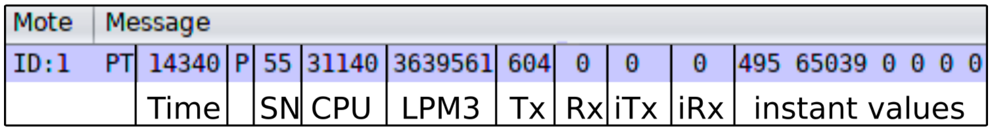
Figure 5.2 – Linear topology considered for theoretical model validation and simulations.

With the theoretically estimated energy consumption of each node, we evaluate the results obtained with the same configuration executed in the Cooja simulator. Note that Cooja's emulation of TMote Sky provides four different power modes[2] for the radio, namely $Reception$, $Transmission$, $Idle$, and $Sleep$, and two for the micro-controller, $Active$ and $Sleep$.

In the simulation environment, energy consumption measurements were provided by two different Cooja-ContikiOS tools, namely: *PowerTrace* and *PowerTracker*. *PowerTrace* is a ContikiOS tool, which periodically reports energy consumption through its serial port. *PowerTrace* reports time spent in each state (transmitting, receiving, idle, or sleep for the radio, and active or idle for CPU). *PowerTracker* comes with the Cooja simulator and provides power consumption measurements of the radio. However, *PowerTrace* is executed in the Microcontroller Unit (MCU) code and does not compute short radio's state transitions (e.g., the Rx state for CCA before transmitting). On the other hand, *PowerTracker* can detect radio's transitions and therefore provides more accurate power consumption measurements than *PowerTrace*. Consequently, we use *PowerTracker* to measure the energy consumed by the radio while CPU consumption is still extracted from the *PowerTrace* tool.

The report sent by *PowerTrace* ("Message" field of Figure 5.3) is split into fields contained each consumption state of the device. The values in these fields are represented as

[2]For this implementation, radio uses only the sleep state (power down is not used), and the main Central Process Unit (CPU) switches between active and Low Power Mode 3 (LPM3) states.



Mote	Message									
ID:1 PT	14340	P	55	31140	3639561	604	0	0	0	495 65039 0 0 0 0
	Time		SN	CPU	LPM3	Tx	Rx	iTx	iRx	instant values

Figure 5.3 – Report example of a node with *PowerTrace*.

a counter variable that depends on the CPU clock speed. Therefore, the fields correspond to SN, total accumulated value of CPU, sleeping state in LPM3, transmitting (Tx), reception (Rx), idle transmitting[3] (iTx), and idle reception (iRx). The instant field represents the difference accumulated from the previous report, and thus, is useful to calculate the instant power consumption, given the report period.

Hence, the equation to get the accumulated energy consumption of each device using the reports provided by *PowerTrace* is given by

$$E_{State} = \frac{V_{PP} * I_{State} * Tdev'_{State}}{Clock_{Res}}, \tag{5.1}$$

where E_{State} is the accumulated energy (given in Joules) consumed in the state until the report, V_{PP} is the battery voltage (Volts), I_{State} is the current drawn (in amperes) in the state measured, $Tdev'_{State}$ is the accumulated time of a state (CPU, Tx, Rx, etc.) in a device's timing system, and $Clock_{Res}$ is the clock's frequency ($32{,}768\,\mathrm{Hz}$). However, the "Time" field in the report is given by the OS's timer, which has a scaling factor of 256; thus, the OS's timer has a frequency of $128\,\mathrm{Hz}$. For example, as shown in Figure 5.3,

$$\frac{Time}{128\,\mathrm{Hz}} \approx \frac{CPU + LPM3}{32{,}768\,\mathrm{Hz}}. \tag{5.2}$$

The reports provided by *PowerTracker* are generated by the script running in the simulator. Thus, differently from using the serial peripheral in devices, *PowerTracker* does not impact in the nodes procedures. By default, the reports provided by this tool have the format shown in Listing 5.1, in which nodes have their names with their ID number followed by the state (ON, TX, RX, and INT), the accumulated time on that state, and the percentage it represents to the total time accumulated (MONITORED). Also, an average (AVG) of all nodes is automatically provided by *PowerTracker*. The "INT" represents the radio's interference, i.e., when it receives packets that were not destined to that node, while "ON" state represents

[3]Idle transmitting (iTx) is a specific state that can considered only in a few cases.

```
All nodes connected. Consumption:
AVG ON 11672752 us 11,70 %
AVG TX 123433 us 0,12 %
AVG RX 14224 us 0,01 %
AVG INT 4393 us 0,00 %
Sky_1 MONITORED 49890931 us
Sky_1 ON 472638 us 0,95 %
Sky_1 TX 44358 us 0,09 %
Sky_1 RX 6794 us 0,01 %
Sky_1 INT 3249 us 0,01 %
Sky_99 MONITORED 49891251 us
Sky_99 ON 11200114 us 22,45 %
Sky_99 TX 79075 us 0,16 %
Sky_99 RX 7430 us 0,01 %
Sky_99 INT 1144 us 0,00 %
```

Listing 5.1 – Powertracker report example for two nodes.

all radio's active states. Therefore, we have that

$$t_{Idle} = ON - (TX + RX), \tag{5.3}$$

and

$$t_{Sleep} = MONITORED - ON, \tag{5.4}$$

where t_{Idle} and t_{Sleep} represents the accumulated time on idle and sleep states, respectively.

Hence, the equation to calculate the energy consumed with *PowerTracker* is similar to Equation (5.1), but using the accumulated time instead, given in micro seconds.

5.2 IOTUS FRAMEWORK PERFORMANCE

To guide our experiments, we consider a general environmental monitoring application (e.g., Figure 5.4) in which sensing values of the environment are deployed periodically (e.g., temperature, humidity, etc.) to a data sink. Figure 5.5 shows a 44-node tree topology used in our simulations. The tree is rooted at the data sink (node 1); all intermediate and leaf nodes are sensing nodes. Note that intermediate tree nodes act both, as traffic generators as well as forwarders.

Table 5.2 shows the main simulation parameters used in our tests. We used the simulator set with a random mote startup delay and a radio medium model as Unit Disk Graph Medium (UDGM): Distance Loss. All other parameters, especially those related to the OS, Contiki-MAC, Physical layer, and TMote sky were kept as they are in the original implementation

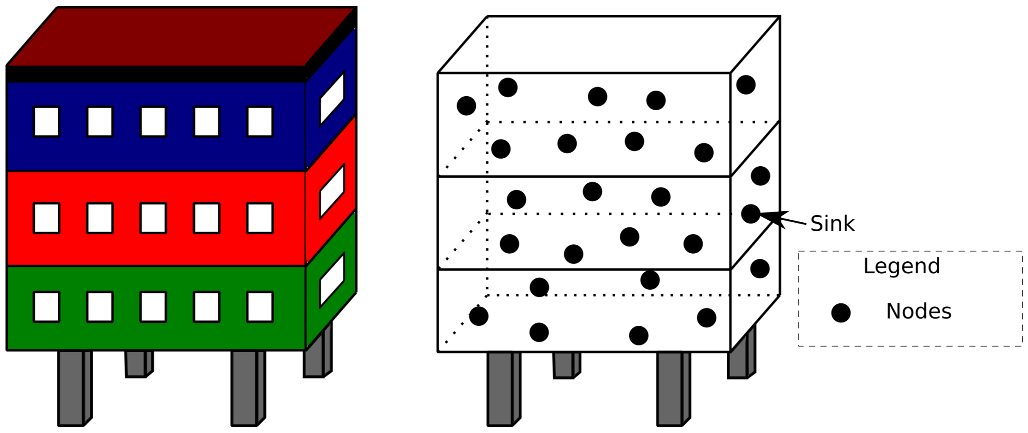


Figure 5.4 – Example of a general environmental monitoring application with nodes.

Table 5.2 – Simulation Parameters.

Parameter	Value
Sensing rate (application packets)	30 seconds
Random delays for application packets	15 seconds
Application piggyback timeout	29 seconds
Application payload size	20 bytes
ContikiMAC's RDC wake-up period	125 milliseconds
Data Link Beacon period	4 seconds
Network's DIO rate (broadcast packet)	4 seconds
Neighbor discovery scanning duration	5 seconds
Keep Alive control data size	12 bytes
Keep Alive transmission rate	30 seconds
Data Link backoff	2 seconds
Cooja's radio medium	UDGM: Distance Loss
Cooja's mote startup delay	1,000 milliseconds
Cooja's mote transmission range	50 meters

of ContikiOS. The *sensing rate* and *application payload size* have been previously used in the literature (e.g., [62]). Keep Alive messages are control messages periodically generated by the network layer and are transmitted through the tree toward the root; thus, we set their size and frequency based on ContikiOS's implementation of RPL. ContikiMAC has a radio wake-up schedule to receive packets, and its default value of 125 ms was kept for these simulations. Application packets can have a random delay of up to 15 seconds to reduce network packet collision. Also, when using IoTUS *Piggyback Service*, application packets can wait a timeout of 29 seconds. Finally, MAC layer adds a backoff period of up to 2 seconds when necessary.

IoTUS framework was compared to the ContikiOS's adapted stack [15] because the latter

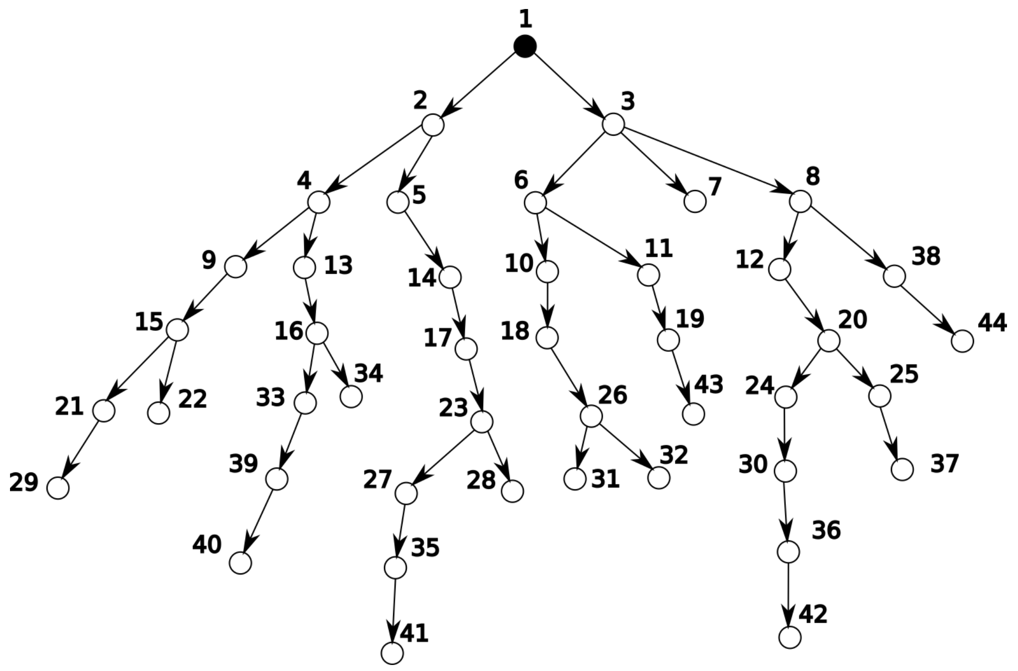


Figure 5.5 – Tree topology considered for environmental monitoring.

is the wide accepted architecture available with ContikiOS for IoT applications. Also, that adapted Rime still uses some optimized features for IoT protocols, e.g., the shared tool for building packets inside the stack (*Packetbuf*).

It is important to advance that, to obtain stochastic relevance, results reported in Chapter 7 were obtained by averaging over 10 runs using random seeds. Thus, this process provides a confidence interval of 95%, when assuming a Student or a t-distribution [63, p. 432].

5.2.1 Overhead validation

Another theoretical model was developed to evaluate the overhead of IoTUS framework compared to the adapted Rime stack. Here, a star topology (Figure 5.6) and a linear topology (Figure 5.2) were used, both with 10 nodes, which results already show large differences and characteristics of IoTUS framework under these topologies. The theoretical results are then compared with the measurements obtained by the simulations.

With the star topology, every node sends application messages and are one hop away from the sink. This topology is interesting to evaluate the performance for long-range radios.

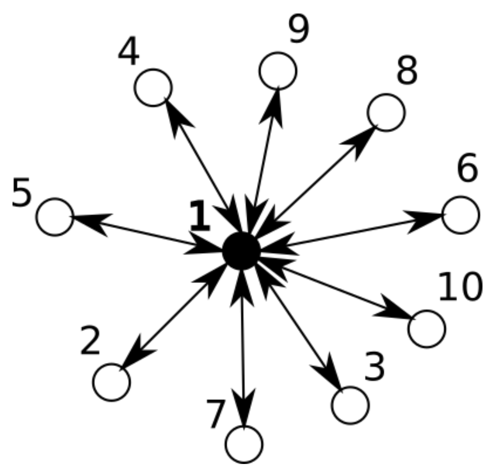


Figure 5.6 – Star-like topology used for IoTUS validation.

5.2.2 IoTUS simulated results

As mentioned in Chapter 3, IoTUS framework brings more advantage when more complex applications are tackled, i.e., IoTUS provides better results when more protocols and/or more procedures are being executed in the stack. Therefore, we developed two simulation setups that use different protocols and procedures. These setups operate with the star, linear, and tree topologies.

With both setup's results, it is possible to glimpse over the performance of IoTUS framework with protocols exchange. The metrics considered to evaluate this simulation were:

- Memory usage;
- Energy consumption;
- Overhead;
- Network lifetime.

Static routes setup

This setup consists of the original implementation of ContikiMAC [27] and a static routing protocol. Therefore, this network protocol requires a pre-defined routing table, i.e., paths already set.

Thus, the main procedures executed in this setup is:

- Application messages with sensed data (payload);

- Keep Alive packets.

It is expected that the main advantage of IoTUS will be the *Piggyback Service*, which aggregates packets towards the sink.

Neighbor discovery setup

For this setup, both Data Link and network layers have protocols with more functionalities. Thus, for the DL layer, we modified ContikiMAC [27] and called it ContikiMAC802. The modified DL protocol in this setup uses an adaptation of the registering process of IEEE 802.15.4 MAC [30]. For the network layer, we created a simplified version of RPL [10] protocol, called RPL-like, which performs similar Neighbor Discovery procedures, and uses this information to create paths to the sink, forming the tree topology.

ContikiMAC802 has all features of ContikiMAC plus an association procedure, and it is composed of 5 control frames for commands: Beacons, Register request (Reg), Data request (Req), register answer (Ans), and acknowledgment (ACK). The RPL-Like implementation creates its routing table and DODAG, and it has another 5 control packets: DIO, DAO, DODAG Information Solicitation (DIS), and DAO-ACK.

The protocols implemented for adapted Rime stack form three layers: physical, DL, and Network. Each layer adds a minimum header with a fixed size (base). Also, each protocol may have specific commands, which is fixed and named "base". The association of these headers and a command base forms then a control frame/packet. For example, a DL Beacon control frame is formed by a Phy header, a DL header, and a Beacon command, which has a total size of 20 bytes. Table 5.3 shows, therefore, all possible combinations of messages implemented for these protocols.

Table 5.3 – Headers, payload, and control commands' sizes in adapted Rime stack.

Packet type	Base [bytes]	Frame/Packet [bytes]
Phy header	6	—
Data link header	11	—
Network header	1	—
Checksum header	2	—
DL_{Beacon} (Beacon)	1	20
DL_Reg (Register)	8	27
DL_Req (Request)	6	25
DL_Ans (Answer)	4	23
Acknowledgment (ACK)	0	11
NL_{DIO} (DIO)	12	33
NL_{DIS} (DIS)	4	25
NL_{DAO} (DAO)	4	25
NL_{DAO_ACK} (DAO-ACK)	4	25
App_{msg} (App. Message)	20	41

5.3 CHAPTER'S CONCLUSION

This chapter presents the methods and parameters used in this work to validate and obtain results that compare IoTUS framework to the adapted Rime stack. Therefore, scenarios were proposed in such a way that performance is evaluated with different complexity, i.e., the stack with IoTUS is compared with Rime stack using varying network size, topologies, and protocols procedures. The results are compared with memory usage, energy consumption, overhead, average power consumption, and network lifetime.

6 THEORETICAL RESULTS

This chapter presents theoretical models, equations, and expected results for the simulation experiments developed.

The energy consumption evaluation starts with simple scenarios: one or two motes, transmitting without collisions. Later, the complexity of these scenarios is increased step by step, so that expected results can be compared. Consequently, these energy estimation procedures can be validated, and more complex topologies can be analyzed.

For overhead evaluation, we compared IoTUS framework with the adapted Rime stack [16] using theoretical models based on the proposed linear and star-like topologies. Also, in these scenarios, we used modified protocol versions called ContikiMAC802 and RPL-like; thus, many parameters could be obtained by the original ContikiMAC [27] and RPL [10] implementations.

Initially, a two-mote scenario is evaluated with the association and periodic transmission events. Later, this logic is extended to an N nodes network for both star and linear topologies. The notation N represents the total number of motes in a scenario.

In this work, overhead is defined as the percentage of header and control bytes per total number of bytes transmitted in the network. However, our theoretical model considers an environment without packet collision or repeated transmissions, i.e., an ideal environment without packet loss.

Hence, for evaluation purpose, we split the overhead into the following terms:

- **Pure header**: The header bytes of any protocol inserted in the packet fields, usually informing properties or details other than instruction or payload, e.g., packet size, addresses, checksum, packet type, sequence number, etc;
- **Control data**: The bytes of a frame or packet containing protocols' control commands, e.g., the commands of a mote's association or the RPL‘s packets (DIO, DAO, DIS, and others).

Thus, a *pure header overhead* is defined as the percentage of pure header bytes over the total number of bytes successfully transmitted.

6.1 ENERGY CONSUMPTION EVALUATION

The motes considered in this work have well-defined states of operation. However, the transitions between those states can produce transients not easily measured/accounted. Thus, it is common to accept an average power consumption per state, i.e., an idle state uses a fixed value of current and when transmitting, it uses a higher, also fixed, value. In this work, TMote Sky [44] was used for simulation; thus, the theoretical model will consider the values shown in Table 5.1. Besides, motes are normally powered with a battery, providing $3\,\mathrm{V}$ of voltage (V_{PP}).

The energy consumption of a node in a given state is given by

$$E_{State} = V_{PP} * I_{State} * t_{State}, \tag{6.1}$$

where I_{State} corresponds to the current, and t_{State} is the elapsed time.

6.1.1 Scenario 1: single node transmitting broadcast packets

If this mote starts transmitting packets, then the energy calculation considers the transmission current and the transmission time (proportional to the packet's length – 49 bytes with this scenario). As TMote Sky's radio technology uses a baud rate[1] of $250\,\mathrm{kbps}$, the transmitting state will last $1.568\,\mathrm{ms}$. Also, because ContikiMAC [27] is used at the DL layer, its implementation requests a CCA procedure before the transmission, along with a wait for ACK. As shown in Figure 6.1, to initiate a transmission, the mote leaves the sleep state, goes to reception/CCA, and then starts transmitting; hence, it takes a total of 2.2 ms.

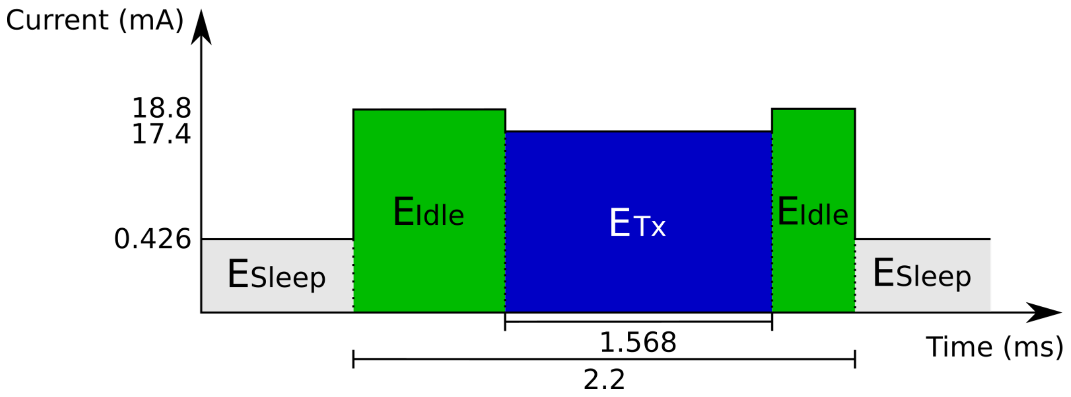


Figure 6.1 – Time diagram of a transmitted packet.

Therefore, consider a scenario that lasts 30 minutes, in which motes transmit every 30

[1]The baud rate is the rate at which symbols are transferred in a communication channel.

seconds. Motes will have a number of transmissions (Num_{Tx}) equal to

$$\begin{aligned} Num_{Tx} &= \frac{30 * 60\ s}{30\ s/transmission} \\ &= 60\ transmissions, \end{aligned} \tag{6.2}$$

and so, the total energy consumption per state is given by

$$E_{Idle} = V_{PP} * (2.2\ ms - 1.568\ ms) * I_{Idle} * Num_{Tx}, \tag{6.3}$$

$$E_{Tx} = V_{PP} * 1.568\ ms * I_{Tx} * Num_{Tx}, \tag{6.4}$$

$$E_{Sleep} = V_{PP} * (30\ s * 1000\ ms - 2.2\ ms) * I_{Sleep} * Num_{Tx}. \tag{6.5}$$

Hence, for this single mote, the expected energy consumption is shown in Table 6.1. It is important to point out that, in this work, we consider idle consumption as the event of being in the reception state without receiving any message. Appendix D contains the script used to obtain these results.

Table 6.1 – Energy consumption expected for one node simulation during 30 minutes.

State	Energy (mJ)
E_{Tx}	4.9
E_{Rx}	0.0
E_{Idle}	2.138
E_{Sleep}	2,300.2

6.1.2 Scenario 2: one node transmitting, and other node receiving

Increasing the complexity, consider a receiver node introduced to the vicinity of this transmitting mote. Also, consider that nodes are using ContikiMAC [27] to access the medium. The new node will be receiving packets from the previous mote. Moreover, the energy-saving mechanisms of this protocol is activated to preserve consumption.

ContikiMAC protocol adds many procedures to save energy, e.g., duty cycle reception, preambled packets, neighbor phase detection, burst transmission, etc. Therefore, the default implementation parameters and also the average values of events will be considered for this analysis, e.g., the number of CCAs before transmission, the quantity of preamble before a reception, and so on.

Moreover, the receiver acknowledges with a 5 bytes frame (ACK is expected to spend 0.32 ms), and the transmitter sends application messages and network Keep Alive (KA).

Table 6.2 – ContikiMAC parameters and average attributes.

Attribute	Value
Time between receptions event	125 ms
Duration of CCA	0.44 ms
Number of CCA in reception event	2
Time gap between CCA in reception event	0.74 ms
Number of CCA before a transmission	6
Time gap between CCA before a transmission	0.6
Number of preambles before finding the neighbor	27.5
Number of preambles after finding the neighbor	6
Time gap between preambles	0.84 ms
Time to wait for a burst transmission	0.25 ms

Table 6.2 presents the parameters for this new setup of two nodes using ContikiMAC. This DL protocol works with preambles, and hence, some procedures are multiplied by its average number. The expected energy consumption for the first transmission is given by

$$E_{First\ tx} = 6 * E_{CCA} + (27.5) * E_{Pkt} + E_{Ack}, \tag{6.6}$$

where E_{CCA} is the consumption for one CCA procedure, E_{Pkt} is the consumption for one packet (preamble), and E_{Ack} is the consumption for one ACK.

After finding the neighbor's wake up schedule, the subsequent messages in ContikiMAC are optimized. Hence, preambles average number reduces from 27.5 to 4.5, and so each transmission after neighbor's information is given by

$$E_{Other\ tx} = 6 * E_{CCA} + (4.5) * E_{Pkt} + E_{Ack}. \tag{6.7}$$

Thus, in the 30 minutes scenario, a sending mote would generate Num_{Tx} of application messages and Network KA, which results in a total of 120 packets. As well, all motes would have periodical reception events (receiver node puts its radio to reception state waiting for a message) given by

$$\begin{aligned} RCE_{Events} &= \frac{30 * 60 * 1000\ ms}{125\ ms/Receptions} \\ &= 14400\ Receptions. \end{aligned} \tag{6.8}$$

Therefore, the final consumption equation can be expressed as

$$E_{Sending\,node} = (E_{First\ tx} + 119 * E_{Other\ tx}) + RCE_{Events} * E_{Wake\,up} + E_{Sleep}, \quad (6.9)$$

where $E_{Wake\,up}$ is the total energy consumption caused by the reception events, and E_{Sleep} is the consumption spent with sleep state. However, part of the wake-up events do not receive any packet, being accounted as E_{Idle}, and the other part is accounted as E_{Rx}. Therefore, assuming $14,400$ wake-up events as E_{Idle} gives an upper bound of the final mote consumption.

With these equations, the two nodes would have an expected energy consumption as given in Table 6.3.

Table 6.3 – Theoretic energy consumption of one sink node and one transmitting (application and network packets) node during 30 minutes.

Nodes	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)
1 (sink)	2.2	10.6	723.0	2,284.3
2 (app.)	46.0	2.4	762.5	2,281.9

6.1.3 Scenario 3: 6 nodes, linear topology, and only end node sends data

This model was used in a linear topology of six motes, in which only the end node (number 6) generates application messages. However, the Network layer does not generate any packet. Table 6.4 presents the expected energy consumption for this configuration.

Table 6.4 – Expected energy consumption in a linear topology with only one node generating messages (application layer) during 30 minutes.

Nodes	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)
1 (sink)	1.1	5.3	719.7	2,284.2
2	25.0	6.5	743.3	2,283.1
3	25.0	6.5	743.3	2,283.1
4	25.0	6.5	743.3	2,283.1
5	25.0	6.5	743.3	2,283.1
6 (app.)	23.9	1.2	740.0	2,283.0

6.1.4 Scenario 4: 6 nodes, linear topology, and all but sink nodes transmit data

We increased the number of packets generated in the network by making motes (from number 2 to 6) create both application messages and network control packets, i.e., sensed data reports and Keep Alive. Table 6.5 presents the expected energy consumption for this configuration. Appendix E contains the script used to obtain these results for linear topology.

Table 6.5 – Expected energy consumption in a linear topology with all nodes generating messages (application and Network layer) during 30 minutes.

Nodes	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)	E_{DSleep} (µJ)
1 (sink)	11.0	53.1	749.9	2,284.8	5.36
2 (app.)	231.7	54.4	969.9	2,273.8	5.34
3 (app.)	185.3	41.4	918.1	2,275.8	5.34
4 (app.)	138.8	28.4	866.2	2,279.9	5.35
5 (app.)	92.4	15.4	814.4	2,279.9	5.35
6 (app.)	46.0	2.4	762.5	2,281.9	5.36

The TMote Sky model was used in the Contiki/Cooja[2] simulation with its default radio configuration, so the radio does not use the lowest energy mode available. Moreover, the sleep state described in the fifth column (E_{Sleep}) of Table 6.5 refers to the third most energy-saving mode available for this mote (I_{Sleep} equals to $426\,\mu A$).

The TMote Sky can operate in "Deep Sleep" modes [44]. MCU idle and Radio off operate with only $54.4\,\mu A$ nominal current, and by putting the MCU in standby, only $5.1\,\mu A$ is required. For that, a mote has to use the radio's lowest energy consumption mode for the sleep periods, which would consume $1\,\mu A$. It is important to note that the different levels of sleep mode can impact on the transition times of the radio, i.e., to get ready to transmit/receive. Here, as the simulated application is very coarse in time, deep sleep modes can easily be accommodated.

Thus, using the *Deep Sleep mode*, the expected results for the linear topology with all nodes sending KA and application messages would be as shown in the last column (E_{DSleep}) of Table 6.5, in which radio's sleep state would have almost no impact on the energy consumption.

6.2 OVERHEAD EVALUATION

In this section, the overhead of the different scenarios will be theoretically analyzed. We skip scenarios 1 to 4, to keep the text short, and focus on scenarios with neighbor discovery. Scenario 5 consists of a Data Link layer with default ContikiMAC protocol, and a Network layer with a static routing protocol. Scenario 5 is skipped in our theoretical models since the

[2]Contiki/Cooja was used as a state simulator, hence average transitions times are incorporated in simulation.

procedures in it are a subset of the procedures in Scenario 6.

6.2.1 Scenario 6: 2 nodes and neighbor discovery procedure

Association stage

Initially, consider a connection between two motes in the standard stack. In this case, because of the selected protocols, DL would be the first layer to start its procedures. ContikiMAC802 association protocol is based on the standard IEEE 802.15.4 MAC, which exchanges at least 5 types of command frames. ContikiMAC802 implemented frames are described in Table 5.3.

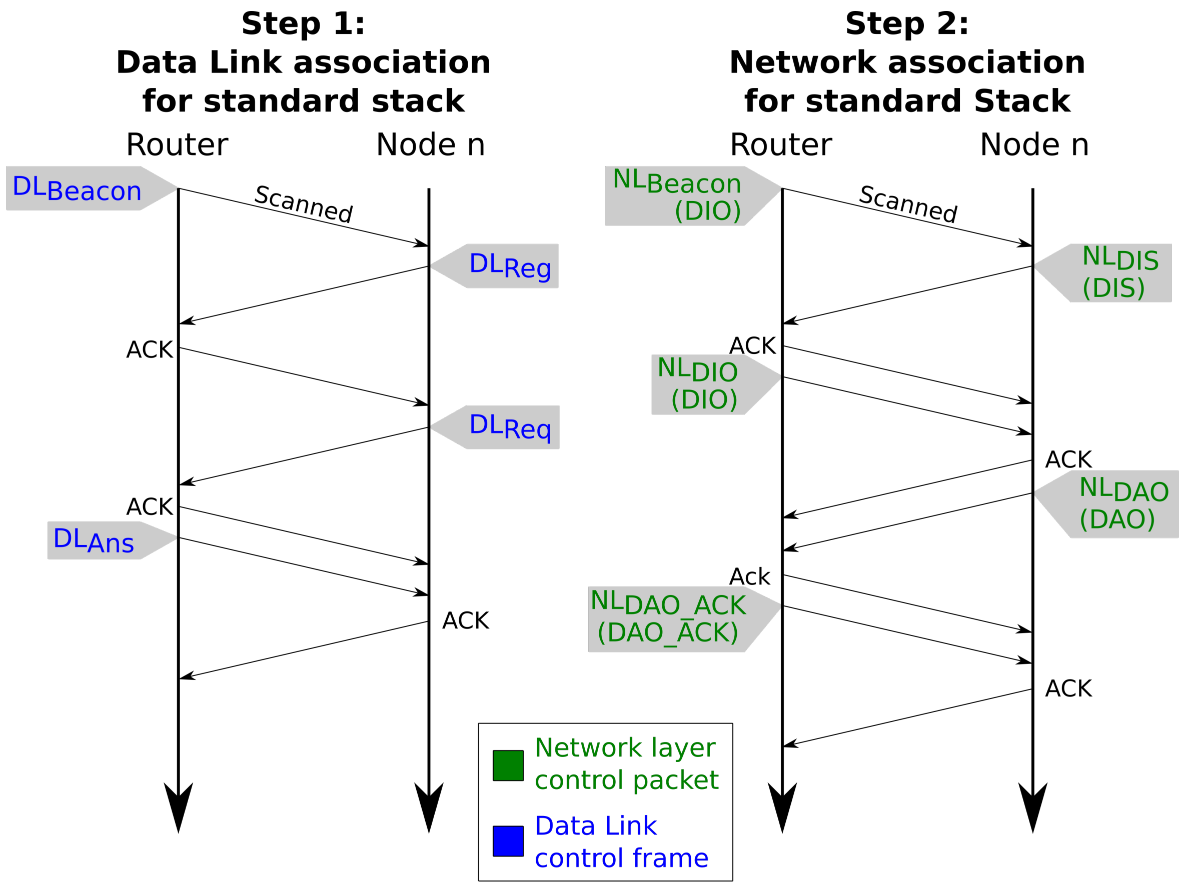


Figure 6.2 – Time diagram of control packets exchanged in a standard stack registering (2nd layer) and neighbor discovery (3rd layer) process.

Figure 6.2 illustrates the Data Link and Network association process of two nodes, where RPL-Like packets are represented with green color, while ContikiMAC802 is the blue color. Hence, in Step 1, after the scanning step (in this case listening for DL beacons), Rime stack would spend

$$\begin{aligned} Rime_{DL\,conn} &= DL_{Beacon} + DL\_Reg + DL\_Req + DL\_Ans + 3 * ACK \\ &= 128\,bytes, \end{aligned} \tag{6.10}$$

where DL_{Beacon} is the router's beacon to be answered followed by IEEE 802.15.4 MAC command types. DL_Reg, DL_Req, and DL_Ans correspond to the total size of DL associations frames specified in Table 5.3.

In the adapted Rime stack [16], since little information is shared between layers, the network layer has to wait for the DL to be ready to send packets. Also, another scanning procedure is necessary, so that the network layer recognizes its neighbors. In the RPL-Like protocol, it has another 4 commands type (Table 5.3). After a second scanning procedure, the **Network layer** association would be given by

$$\begin{aligned} Rime_{NL\,conn} &= NL_{Beacon} + NL_{DIS} + NL_{DIO} + NL_{DAO} \\ &\quad + NL_{DAO\_ACK} + 4 * ACK \\ &= 185\,bytes, \end{aligned} \tag{6.11}$$

where NL_{DIS}, NL_{DIO}, NL_{DAO}, and NL_{DAO_ACK} correspond to the total size of network control packets, as specified in Table 5.3.

Thus, the association procedure of a single mote in adapted Rime stack results

$$\begin{aligned} Rime_{Single\,conn} &= Rime_{DL\,conn} + Rime_{NL\,conn} \\ &= 313\,bytes. \end{aligned} \tag{6.12}$$

On the other hand, with IoTUS, *Neighbor Discovery* could benefit from aggregation. This procedure adds 1 byte to the header and 2 bytes per layer command (two layers in this case). Therefore, as shown in Figure 6.3, some control commands could be aggregated. The packets' total sizes are described in Table 6.6.

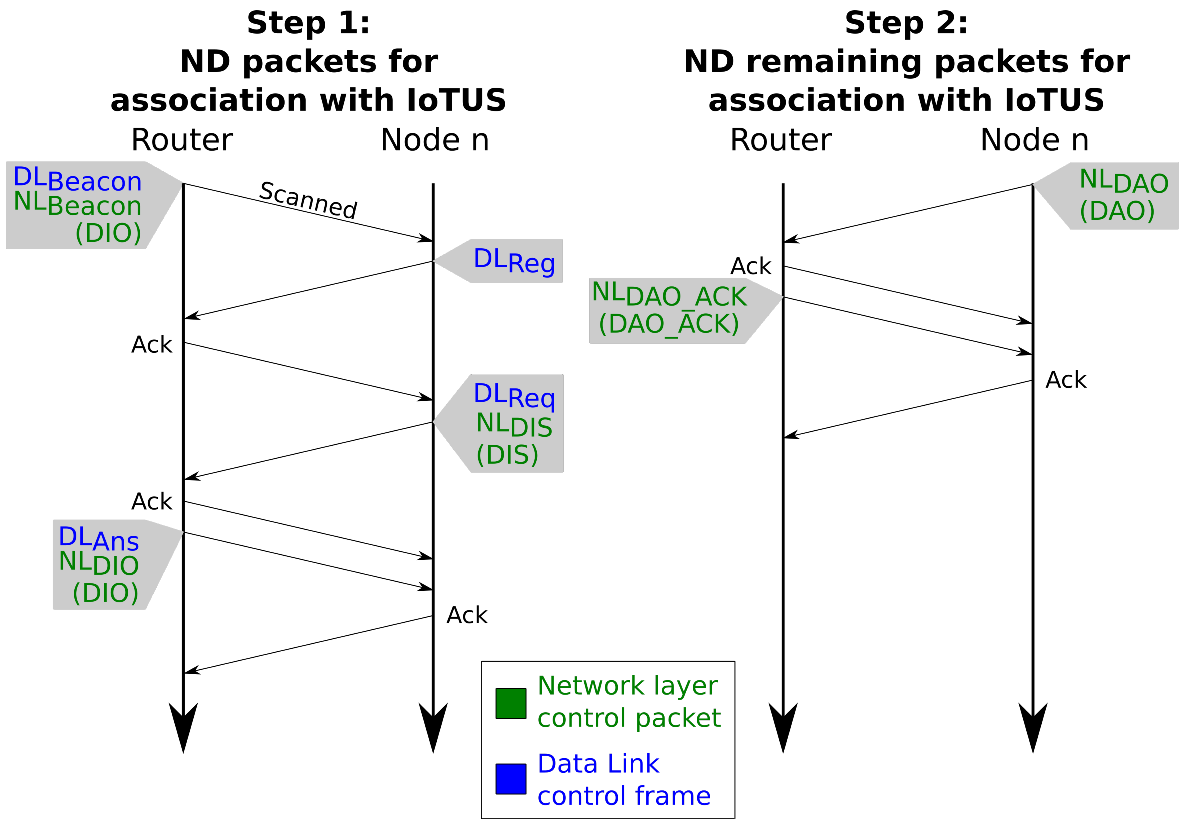


Figure 6.3 – Time diagram of control packets exchanged with the new IoTUS framework in a registering (2nd layer) and neighbor discovery (3rd layer) process.

The registering process (Figure 6.3) between two motes using IoTUS framework spends

$$\begin{aligned} IoTUS_{Single\,conn} &= IoTUS_{Beacon} + IoTUS_{Reg} + IoTUS_{Req} + IoTUS_{Ans} + IoTUS_{DAO} \\ &\quad + IoTUS_{DACK} + 5 * ACK \\ &= 255\,bytes, \end{aligned} \tag{6.13}$$

where $IoTUS_{Beacon}$ is the DL and network packets aggregated with *Neighbor Discovery* module, $IoTUS_{Req}$ is the aggregated request packet, and $IoTUS_{Ans}$ is the aggregated answer packet for this association process. Also, all values correspond to the complete size specified in Table 6.6.

Hence, analyzing the association process of only two nodes illustrated in Figure 6.3, IoTUS would reduce 58 bytes per connection, i.e., 18.53% of association bytes reduced.

However, the final impact of IoTUS framework in the system is not only in the number of bytes exchanged during the association but also in its duration. As the association

Table 6.6 – Size of aggregated commands with IoTUS.

Field	Aggregated size [bytes]
DL Beacon + DIO ($IoTUS_{Beacon}$)	38
DL Register ($IoTUS_{Reg}$)	32
DL Request + DIS ($IoTUS_{Req}$)	36
DL Answer + DIO ($IoTUS_{Ans}$)	42
DAO ($IoTUS_{DAO}$)	26
DAO-ACK ($IoTUS_{DACK}$)	26

time reduces, less periodic messages would be transmitted, i.e., fewer events of the periodic broadcast from DL layer and Network layer will occur, impacting on the scenario overhead measurement.

Due to the parameters in Table 5.2, it was observed that the Rime stack association process takes an average total of 13 DL_{Beacon} plus 13 NL_{Beacon} until the association process is complete. Therefore, the final overhead measurement for Rime stack considers the events that occurred during the association process and is given by

$$\begin{aligned} Rime_{Total\,conn} &= Rime_{Single\,conn} \\ &\quad + 12 * DL_{Beacon} + 12 * NL_{Beacon} \\ &= 949\,bytes. \end{aligned} \tag{6.14}$$

Meanwhile, the system using IoTUS takes less time for the association, since both protocols were aware of each other's procedures. These protocols will have an average of 5 broadcast events during the association. Hence, IoTUS final overhead measurement is given by

$$\begin{aligned} IoTUS_{Total\,conn} &= IoTUS_{Single\,conn} + 4 * IoTUS_{Beacon} \\ &= 407\,bytes. \end{aligned} \tag{6.15}$$

Therefore, for the connection procedure of two nodes, it is expected that Rime stack will have a *pure header overhead* of 77.76% which is given by

$$Rime_{Pure\,overhead} = 1 - \frac{BASE(Rime_{Total\,conn})}{Rime_{Total\,conn}}, \tag{6.16}$$

where the function $BASE()$ means extracting only the base values of the respective packets specified in Table 6.6.

Meanwhile, IoTUS system will result in a *pure header overhead* of 73.71% which is obtained from

$$IoTUS_{Pure\,overhead} = 1 - \frac{BASE(IoTUS_{Total\,conn})}{IoTUS_{Total\,conn}}. \tag{6.17}$$

Therefore, using IoTUS resulted in a *pure header overhead* reduction of 4.05% per node connection. It is important to remind that the default implementation of IEEE 802.15.4 MAC [9] and RPL [10] has even bigger headers, and also uses 6LoWPAN [7] and ICMPv6 [59] protocols. Thus, IoTUS is expected to facilitate these protocols to reduce even more network overhead.

Periodic sensing and keep alive stage

After the association stage, motes are able to communicate at the application layer. Consequently, messages from application, network, and DL layer are expected to coexist in this environment.

The stack using IoTUS framework can benefit from the functions provided by *Piggyback Service*. In this case, depending on the parameters used when a packet is created, this module can optimize the network by aggregating control packets with messages, messages with other messages, and so on. *Piggyback Service* module adds 1 byte of header, plus 1 or 2 bytes per piece of message (PBp).

Therefore, the size of packets exchanged with IoTUS is not constant but dependent on the application parameters. IoTUS would have an overhead result depending on how often it produces aggregated packets. For this scenario, Table 6.7 presents possible aggregation sizes.

The scenario considered here monitors environmental temperature, with no stringent time requirements. Although *Piggyback Service* may increase delay (depending on the timeout parameter), this delay is normally acceptable for this application. Therefore, in this example, the application layer will create piggyback pieces (the block of memory used by *Piggyback Service* described in Chapter 4), as expressed in Table 6.7.

Since the selected generation rate of DAOs packets and application messages have the same value (Table 5.2), the analysis of the average case, given by events of "*DAO + 1 Application message*", can be used as the theoretical model of this scenario. Note that when packets are not aggregated (e.g., timeout expires), the node will transmit a "Single application message" plus, eventually, a "Single DAO command". *Piggyback Service* can also

Table 6.7 – IoTUS Aggregated size for some typical aggregation conditions.

Aggregation Condition	Aggregated size [bytes]
Single application message in the packet ($IoTUS\_App_{msg}$)	42
2 Application messages	64
3 Application messages	86
4 Application messages	108
5 Application messages	130 (exceeds frame)
Single DAO command in the packet ($IoTUS_{DAO}$)	26
DAO + 1 Application message ($PB_{App\&DAO}$)	48
DAO + 2 Application messages	70
DAO + 3 Application messages	92
DAO + 4 Application messages	114
DAO + 5 Application messages	136 (exceeds frame)

aggregate more than one packet (e.g., "DAO + 2 Application messages"), so the event "DAO + 1 Application messages" represents the network expected case.

Thus, according to the parameters for this scenario and the observed average broadcast events (B_{Events}) per connection, Rime stack uses 13 B_{Events} per mote association, which represents an average of 52 seconds of association stage ($t_{one\,conn}$) for two nodes. Since the whole scenario lasts 30 minutes and each mote reports every 30 seconds, all nodes would have

$$Ev_{report} = \left\lfloor \frac{30\,\text{min} * 60\,\text{s} - t_{one\,conn}}{30\,\text{s}} \right\rfloor, \tag{6.18}$$

where $t_{one\,conn}$ is the average duration of one node's complete association. In Rime stack, 58 report events (Ev_{report}) are then expected with 58 network's DAO packets generated.

The broadcast parameter in Table 5.2 allows determining the amount of Tot_B_{Events} during the 30 minutes of simulation, which is given by

$$Tot\_B_{Events} = \left\lfloor \frac{30\,\text{min} * 60\,\text{s}}{4\,\text{s}} \right\rfloor \qquad = 450\,broadcast\ events. \tag{6.19}$$

Then, according to Table 5.3, Rime stack in the sensing and KA stage spend an amount of bytes equivalent to

$$\begin{aligned} Rime_{2\,nodes\,Sen\&KA} = {} & Ev_{report} * (App_{msg} + NL_{DAO} + ACK * 2) \\ & + Tot\_B_{Events} * (DL_{Beacon} + NL_{DIO}), \end{aligned} \tag{6.20}$$

where App_{msg} is the application's complete size message (41 bytes).

The expected Rime stack's overhead for the whole scenario is 96.03% which is given by

$$Rime_{2\,nodes\,overhead} = 1 - \frac{Ev_{report} * App_{Payload}}{Rime_{Single\,conn} + Rime_{2\,nodes\,Sen\&KA}}, \tag{6.21}$$

where $App_{Payload}$ is the application's payload only (20 bytes of data).

On the other hand, IoTUS framework would be able to intrinsically aggregate these packets, spending 59 bytes (ACK included) for every 20 bytes of payload. Also, the association time is shorter, since this system takes an average of 5 B_{Events}, which means 20 seconds of association stage ($t_{one\,conn}$). Therefore, the system with IoTUS can generate 59 report events (Ev_{report}). IoTUS framework is expected to produce an amount of bytes during its sensing and KA stage equivalent to

$$\begin{aligned} IoTUS_{2\,nodes\,Sen\&KA} = {} & Ev_{report} * (PB_{App\&DAO} + ACK) \\ & + Tot\_B_{Events} * (IoTUS_{Beacon}), \end{aligned} \tag{6.22}$$

where $PB_{App\&DAO}$ is the full size of an event of application message aggregated with network's DAO packet, $IoTUS_{Beacon}$ is the DL beacon and network's DIO aggregated with the *Neighbor Discovery* module of IoTUS.

For this scenario, IoTUS framework is expected to have an overhead of 94.34% which is given by

$$IoTUS_{2\,nodes\,overhead} = 1 - \frac{Ev_{report} * App_{Payload}}{IoTUS_{Single\,conn} + IoTUS_{2\,nodes\,Sen\&KA}}, \tag{6.23}$$

where $PB_{App\&DAO}$ is the full size of an event of application message aggregated with network's DAO packet, $IoTUS_{Beacon}$ is the DL beacon and network's DIO aggregated with the *Neighbor Discovery* module of IoTUS.

In IoTUS system worst case, no packet is aggregated by *Piggyback Service*. Thus, the term "$PB_{App\&DAO} + ACK$" in Equation (6.23) would be substituted by "$IoTUS\_App_{msg} + IoTUS_{DAO} + 2 * ACK$", which results in an overhead of 94.8%. It means that IoTUS system still held a better result for the duration of this scenario.

6.2.2 Scenario 7: 10 nodes star topology, neighbor discovery procedure

The star topology analysis (Scenario 7) is an extension of the 2 nodes topology analyzed in the previous scenario. However, with more nodes, some collisions may occur and therefore cause some delay at the connection process, while the coordinator (mote 1) will be broadcasting its DL's beacons and RPL's DIO packets during the whole scenario, every 4 seconds (Table 5.2).

For these parameters in Table 5.2 and the current DL protocol, reception slots occur every $125\text{m}s$. Also, because the beacon period is 4 seconds (Data link and Network layer), the nodes have 32 communication slots between broadcasts, which is enough for 10 nodes to proceed with their association.

The overhead measurement of this scenario is a proportion between the connection stage and the sensing and keep alive stage of 9 sensing motes and 1 sink coordinator. Therefore, for the 30 minutes of scenario, 9 motes are expected to generate two types of transmissions: 1 DAO packet and 1 application message every 30 seconds. Consequently, after the connection stage, each mote will generate about 58 transmissions (Ev_{report}) of each type during the whole scenario according to Equation (6.18). Also during this period, the coordinator mote would send 450 broadcasts of DL's beacons and 450 broadcasts of RPL's DIO, according to Equation (6.19).

The association stage for a star topology is similar to the 2 nodes scenario. However, collision and backoff events can delay this procedure. Still, because of the small number of motes compared to the number of available transmission slots, this effect does not significantly impact the results.

Hence, using the values calculated for two nodes, the number of bytes exchanged for the sensing and keep-alive stage ($Rime_{Sen\&KA}$) in the adapted Rime is given by

$$\begin{aligned} Rime_{Sen\&KA} =& (N-1) * Ev_{report} * (App_{msg} + NL_{DAO} + 2 * ACK) \\ & + Tot\_B_{Events} * (DL_{Beacon} + NL_{DIO}), \end{aligned} \tag{6.24}$$

where App_{msg} is the full size of an application message, NL_{DAO} is the network keep alive, ACK is the acknowledge between transmissions, DL_{Beacon} is the DL beacon, and NL_{DIO} is the RPL's DIO.

Thus, the total overhead in Rime stack corresponds to 85.62% which is obtained from

$$Rime_{Overhead} = 1 - \frac{(N-1) * Ev_{report} * App_{Payload}}{(N-1) * Rime_{Single\,conn} + Rime_{Sen\&KA}}, \tag{6.25}$$

where $App_{Payload}$ is the application's payload only (20 bytes of data), and $Rime_{Single\,conn}$ is the process of a single connection discussed in Equation (6.12).

Analogously, in the system using IoTUS framework and considering the average transmitting event (consisting of 1 application message aggregated with 1 piggyback piece), IoTUS would have its $IoTUS_{Sen\&KA}$ stage give by

$$\begin{aligned} IoTUS_{Sen\&KA} =& (N-1) * Ev_{report} * (PB_{App\&DAO} + ACK) \\ & + Tot\_B_{Events} * IoTUS_{Beacon}, \end{aligned} \tag{6.26}$$

where $PB_{App\&DAO}$ is the full size of an event of application message aggregated with KA packet, $IoTUS_{Beacon}$ is the DL beacon, and RPL's DIO integrated with the neighbor discovery service of IoTUS.

Thus, the total overhead in IoTUS system corresponds to 79.06% which is given by

$$IoTUS_{Overhead} = 1 - \frac{(N-1) * Ev_{report} * App_{Payload}}{(N-1) * IoTUS_{Single\,conn} + IoTUS_{Sen\&KA}}. \tag{6.27}$$

Thus, for the star topology scenario, IoTUS should obtain 6.5% less overhead than the adapted Rime stack.

6.2.3 Scenario 8: 10 nodes linear topology, neighbor discovery procedure

Differently from the star topology, the linear connection causes delays at the nodes further away from the coordinator. Also, for this topology, router nodes will be sending periodic broadcasts of both DL beacons and network's DIO packets, plus network's DAO packets upwards the sink. This increases overhead since many control data is exchanged until the last node is finally connected. It is important to remind that, differently from the star topology scenario, now only the last mote is sending periodic application messages after its connection.

For the 10 nodes linear topology (Figure 5.2), the third mote (therefore having rank 3) will only start its connection after the second node is ready. Consequently, the whole association stage is considered complete after the last mote gets the connected status. Also, every byte exchanged for nodes will be counted for the final overhead measurement.

For example, using Rime stack, the first two connections (sink and second nodes) will proceed as described in Equation (6.14). The association of the third node will have its connection bytes ($Rime_{Total\,conn}$), broadcast events of the previous nodes, plus DAO packets sent by the second node. In sequence, the association of the fourth node will have its $Rime_{Total\,conn}$ bytes, broadcast events (sink, second node, and third node), plus DAO packets (second and third nodes).

Therefore, according to Table 5.2, for every 4 seconds of network runtime, coordinator and router nodes will generate broadcast packet, e.g., DL beacons and Network's DIO packets, besides upward DAO packets every 30 seconds. For the same reasons in the star topology, the DL protocol association using Rime stack is expected to take an average of 5 broadcast events (B_{Events}) to be complete, while the network protocol is expected to take 8 more B_{Events}. Meanwhile, using IoTUS framework, the association of both DL and Network layers will occur simultaneously, and they are expected to take a total of 5 B_{Events} to be done.

Linear topology: association stage

In Rime stack, the DL protocol initiates its connection before the Network layer. Also, the DL layer takes only an average of 20 seconds (5 B_{Events}) to be completed, letting the network available to do its own connections, which takes an average of 52 seconds (13 B_{Events}). Therefore, the analysis of this topology is done separately in layers.

Hence, for Rime stack, on average, the last node to connect will take 468 seconds (t_{AStage}). It means that, during the association stage, the sink node will have this time to generate broadcasts (DL_{Beacon}). Consequently, the second node will have the same duration, minus the time it took to do its own connection (52 seconds). Therefore, the total amount of bytes generated with DL broadcasts during the association stage by each node in the network is given by

$$Rime\_DL_{Lin\_BC} = \sum_{n=1}^{(N-1)} \left(\frac{t_{AStage} - [(n-1) * t_{one\,conn}]}{4} * DL_{Beacon} \right), \tag{6.28}$$

where n is the node position in the linear topology, equivalent to its rank.

With the same structure, the total amount of broadcasts done by Rime's Network layer is given by

$$Rime\_NL_{Lin\_BC} = \sum_{n=1}^{(N-1)} \left(\frac{t_{AStage} - [(n-1) * t_{one\,conn}]}{4} * NL_{Beacon} \right). \tag{6.29}$$

Also, in Rime stack, Network layer starts transmitting NL_{DAO} packets after its association process is complete. However, not only NL_{DAO} generation rate is different, but also its packets are relayed until the sink. Hence, the total amount of bytes transmitted with network's DAO packets during the association stage is given by

$$Rime\_KA_{Lin\_Conn} = \sum_{n=1}^{N} \left(\left\lfloor \frac{t_{AStage} - [(n-1) * t_{one\,conn}]}{30} \right\rfloor * (n-1) * (NL_{DAO} + ACK) \right). \tag{6.30}$$

Considering

$$Rime_{Lin\_Ctrls} = Rime\_DL_{Lin\_BC} + Rime\_NL_{Lin\_BC} + Rime\_KA_{Lin\_Conn}, \tag{6.31}$$

in Rime stack, the *pure header overhead* during the association stage of 10 nodes results in 80.28% which is obtained by

$$Rime_{Lin\_POverhead} = 1 - \frac{(N-1) * BASE(Rime_{Single\,conn}) + BASE(Rime_{Lin\_Ctrls})}{(N-1) * Rime_{Single\,conn} + Rime_{Lin\_Ctrls}}. \tag{6.32}$$

Analogously, IoTUS framework would have its broadcast packets aggregated by *Neighbor Discovery*. The average duration expected for the association stage of 9 motes is 180 seconds (t_{AStage}). Thus, the IoTUS broadcast events would result in an amount of byte given by

$$IoTUS\_ND_{Lin\_BC} = \sum_{n=1}^{(N-1)} \left(\left\lfloor \frac{t_{AStage} - [(n-1) * t_{one\,conn}]}{4} \right\rfloor * (IoTUS_{Beacon}) \right). \tag{6.33}$$

Also, the Network layer would generate $IoTUS_{DAO}$ packets after its complete association. Hence, similar to the standard stack, the expected amount of bytes is given by

$$IoTUS\_KA_{Lin\_Conn} = \sum_{n=1}^{N} \left(\left\lfloor \frac{t_{AStage} - [(n-1) * t_{one\,conn}]}{30} \right\rfloor * (n-1) * (IoTUS_{DAO} + ACK) \right). \tag{6.34}$$

Considering

$$IoTUS_{Lin\_Ctrls} = IoTUS\_ND_{Lin\_BC} + IoTUS\_KA_{Lin\_Conn}, \tag{6.35}$$

then, IoTUS framework *pure header overhead* during the association stage of 10 nodes results in 73.45% which is given by

$$IoTUS_{Lin\_POverhead} = 1 - \frac{(N-1) * BASE(IoTUS_{Single\,conn}) + BASE(IoTUS_{Lin\_Ctrls})}{(N-1) * IoTUS_{Single\,conn} + IoTUS_{Lin\_Ctrls}}. \tag{6.36}$$

Linear topology: Periodic sensing and keep alive stage

Since only the last node in the linear topology will be generating application messages, then all other motes (routers) will be doing broadcasts and DAO packets after the association stage. Also, in the Rime stack, it is expected that the previous stage takes an average of 462 seconds to be completed (all nodes connected). Hence, for this scenario with 30 minutes duration, the remaining time allows Rime nodes to generate 44 report events (Application message and DAO packets– Ev_{report}), plus 333 broadcast events (Tot_B_{Events}). Thus, the additional bytes spent at this remaining stage can be calculated as

$$\begin{aligned} Rime_{Lin\_RCtrls} = & \sum_{n=1}^{N} [Ev_{report} * (NL_{DAO} + ACK) * (n-1)] \\ & + (N-1) * Ev_{report} * App_{msg} + Rime_{Lin\_Ctrls} \\ & + (N-1) * Ev_{report} * (DL_{Beacon} + NL_{Beacon}). \end{aligned} \tag{6.37}$$

Finally, Rime stack overhead for the linear topology results in 97.33% and can be expressed as

$$Rime_{Lin\_Overhead} = 1 - \frac{(N-1) * Ev_{report} * App_{Payload}}{(N-1) * Rime_{Single\,conn} + Rime_{Lin\_RCtrls}}. \tag{6.38}$$

Analogously, IoTUS framework association stage took 180 seconds on average, which leaves time for 54 report events (Application message and DAO packets), plus 405 broadcast events (Tot_B_{Events}). Thus, the additional bytes spent at this remaining stage can be calculated as

$$\begin{aligned} IoTUS_{Lin\_RCtrls} = & \sum_{n=1}^{(N-1)} [Ev_{report} * (IoTUS_{DAO} + ACK) * (n-1)] \\ & + (N-1) * Ev_{report} * PB_{App\&DAO} + IoTUS_{Lin\_Ctrls} \\ & + (N-1) * Tot\_B_{Events} * IoTUS_{Beacon}. \end{aligned} \tag{6.39}$$

Therefore, in the linear topology, IoTUS framework is expected to have an overhead of 95.71% and is obtained by

$$IoTUS_{Lin\_Overhead} = 1 - \frac{(N-1) * Ev_{report} * App_{Payload}}{(N-1) * IoTUS_{Single\,conn} + IoTUS_{Lin\_RCtrls}}. \tag{6.40}$$

6.3 THEORETICAL RESULTS' CONCLUSION

In this chapter, we developed theoretical models to evaluate IoTUS framework and Rime stack. Moreover, energy and overhead model were used to estimate the improvement of stacks using the new framework compared to the standard stack.

The theoretical models also allow evaluating the measuring tools created to calculate the simulation results. Therefore, the results obtained in this chapter will be compared with simulations in Chapter 7. A summary of the theoretical results is presented in Table 6.8 and Table 6.9.

Also, we recapitulate here, for convenience, the defined scenarios:

- **Scenario 1**: single node transmitting broadcast packets;
- **Scenario 2**: one node transmitting, and other node receiving;
- **Scenario 3**: 6 nodes, linear topology, and only end node transmits;
- **Scenario 4**: 6 nodes, linear topology, and 5 nodes transmitting to sink;
- **Scenario 5**: Only discussed in Chapter 7;
- **Scenario 6**: 2 nodes and neighbor discovery procedure;
- **Scenario 7**: 10 nodes, neighbor discovery procedure, and star topology;
- **Scenario 8**: 10 nodes, neighbor discovery procedure, and linear topology.

Table 6.8 – Theoretical energy results for different scenarios.

Scenario	Node ID	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)	E_{DSleep} (μJ)
1	1	4.9	0.0	2.138	2,300.2	—
2	1 (sink)	2.2	10.6	723.0	2,284.3	—
	2 (app.)	46.0	2.4	762.5	2,281.9	—
3	1 (sink)	1.1	5.3	719.7	2,284.2	—
	2	25.0	6.5	743.3	2,283.1	—
	3	25.0	6.5	743.3	2,283.1	—
	4	25.0	6.5	743.3	2,283.1	—
	5	25.0	6.5	743.3	2,283.1	—
	6 (app.)	23.9	1.2	740.0	2,283.0	—
4	1 (sink)	11.0	53.1	749.9	2,284.8	5.36
	2 (app.)	231.7	54.4	969.9	2,273.8	5.34
	3 (app.)	185.3	41.4	918.1	2,275.8	5.34
	4 (app.)	138.8	28.4	866.2	2,279.9	5.35
	5 (app.)	92.4	15.4	814.4	2,279.9	5.35
	6 (app.)	46.0	2.4	762.5	2,281.9	5.36

Table 6.9 – Theoretical results of IoTUS framework and adapted Rime stack.

Parameter	Scenario	IoTUS Framework	Rime stack	Gain
Pure header overhead	6	73.7%	77.8%	$\approx 5.26\%$
	8	73.5%	80.3%	$\approx 8.46\%$
Overhead	6	94.3%	95.8%	$\approx 1.56\%$
	7	79.1%	85.6%	$\approx 7.59\%$
	8	95.7%	97.3%	$\approx 1.64\%$
Network association time	8	180 s	468 s	$\approx 61.53\%$ faster

7 SIMULATION RESULTS

This chapter presents the simulation results obtained in this work. Therefore, we compare to the theoretical models and expand the topology to a higher number of devices.

The simulation results in this chapter present the performance of IoTUS framework for different stack complexities. Thus, it is important to remember that results reported here were obtained by averaging over 10 runs using random seeds with a 95% confidence interval. As a baseline, we use ContikiOS's adapted Rime stack, which is a simplified traditional stack with ContikiOS's cross-layer *Packetbuf*.

The simulations were divided into the scenarios described in Chapter 6, plus other scenarios expanded by the simulation environment. Therefore, the scenarios are:

- Energy consumption simulation validation:
  - **Scenario 1**: single node transmitting broadcast packets;
  - **Scenario 2**: one node transmitting, and other node receiving;
  - **Scenario 3**: 6 nodes, linear topology, and only end node transmits;
  - **Scenario 4**: 6 nodes, linear topology, and 5 nodes transmitting to sink;
- IoTUS performance compared to adapted Rime stack [16]:
  - **Scenario 5**: 10 nodes, static routing network, and tree topology;
  - Overhead simulation validation:
    - **Scenario 6**: 2 nodes and neighbor discovery procedure;
    - **Scenario 7**: 10 nodes, neighbor discovery procedure, and star topology;
    - **Scenario 8**: 10 nodes, neighbor discovery procedure, and linear topology;
  - Energy consumption, memory usage, lifetime, etc.:
    - **Scenario 7**: 10 nodes, neighbor discovery procedure, and star topology;
    - **Scenario 8**: 10 nodes, neighbor discovery procedure, and linear topology;
    - **Scenario 9**: 44 nodes, neighbor discovery procedure, and tree topology;
    - **Scenario 10**: 44 nodes, neighbor discovery procedure, tree topology, and radio's deep sleep.

7.1 ENERGY CONSUMPTION SIMULATION VALIDATION

All the measurements obtained in this work were obtained with Cooja [22] simulator. Cooja provides different ways to measure energy: Over the serial output of each device, using state tracker of each device (*PowerTracker*), and by means of the message output log. In this work, we use more precise Cooja build in (*PowerTracker*) and compare the simulation results with the theoretical results developed in Chapter 6.

Thus, the first simulation (Scenario 1) considers only one device, running a task that turns the radio on and sends a single packet (a broadcast) periodically. This procedure is illustrated in Figure 7.1. Therefore, Table 7.1 compares the expected results of the theoretical model and the simulation.

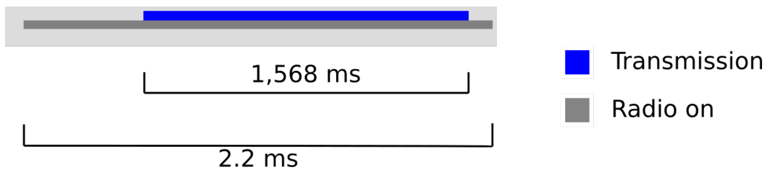


Figure 7.1 – Single transmission simulation in Cooja, repeated 60 times over $30min$ simulation.

Table 7.1 – Energy consumption of Scenario 1: one device compared between the theoretical model and simulation. 30 minutes simulation.

Method	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)
Theoretical model	4.9	0.0	2.138	2,300.2
Simulated results	4.8	0.0	2.293	2,300.6

As observed, both simulation and theoretical results were quite similar. Hence, we increased the complexity of this scenario by adding a receiver node (Scenario 2). In this case, only one device transmits packets to the data sink (node 1). Also, nodes are using ContikiMAC [27] protocol features to save energy. Therefore, Table 7.2 compares the expected results of the theoretical model and the simulation.

Table 7.2 – Energy consumption of Scenario 2: one mote sending (application messages and KA) and other receiving. 30 minutes simulation.

Node ID	Method	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)
1	Theoretical model	2.2	10.6	723.0	2,284.3
	Simulated results	2.0	11.2	738.8	2,283.8
2	Theoretical model	46.0	2.4	762.5	2,281.9
	Simulated results	47.9	2.1	780.1	2,281.9

Once more, we increased the complexity by putting six nodes in a linear topology (Figure 5.2), corresponding to Scenario 3. In this case, only the **last device** transmits packets to the data sink (node 1), while the other nodes relay these packets. Therefore, Table 7.3 compares the expected results of the theoretical model and the simulation. These results can also be visualized in Figure 7.2.

Table 7.3 – Energy consumption of Scenario 3: six devices and only one sending messages in a linear topology. 30 minutes simulation.

Node ID	Method	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)
1 (sink)	Theoretical model	1.1	5.3	719.7	2,284.2
	Simulated results	1.0	5.6	731.0	2,284.5
2	Theoretical model	25.0	6.5	743.3	2,283.1
	Simulated results	25.9	6.7	760.0	2,283.2
3	Theoretical model	25.0	6.5	743.3	2,283.1
	Simulated results	26.7	8.0	761.3	2,283.1
4	Theoretical model	25.0	6.5	743.3	2,283.1
	Simulated results	25.7	8.8	762.2	2,283.1
5	Theoretical model	25.0	6.5	743.3	2,283.1
	Simulated results	26.4	8.1	763.0	2,283.1
6 (app.)	Theoretical model	23.9	1.2	740.0	2,283.0
	Simulated results	24.8	1.2	752.0	2,283.5

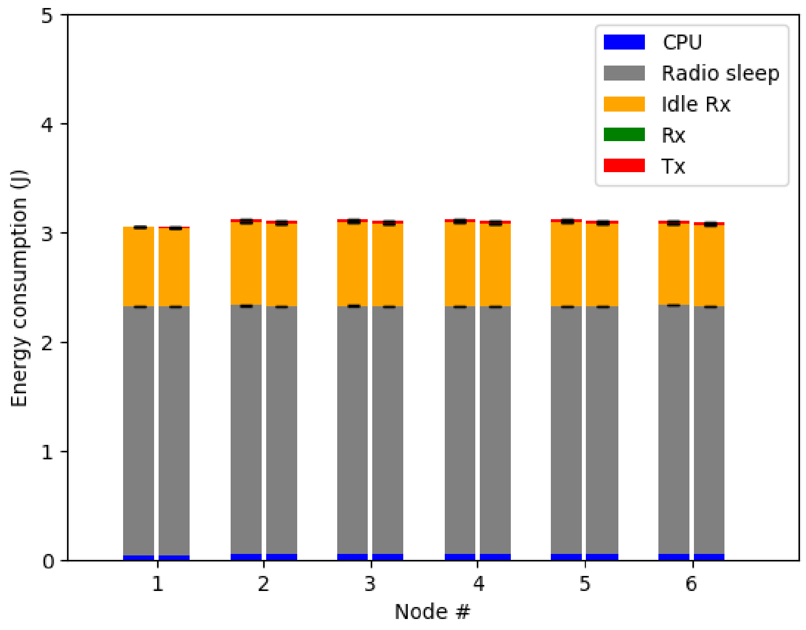


Figure 7.2 – Scenario 3: energy consumption in linear topology with only node six generating messages.

Finally, we increased the number of packets generated in the network simulated. In this case, nodes create application messages and network control frames every 30 seconds and send them to the data sink device (Scenario 4). Therefore, Table 7.4 compares the expected results of the theoretical model and the simulation. Once again, these results can be visualized in Figure 7.3.

Table 7.4 – Energy consumption of Scenario 4: five devices sending messages in a linear topology. 30 minutes simulation.

Node ID	Method	E_{Tx} (mJ)	E_{Rx} (mJ)	E_{Idle} (mJ)	E_{Sleep} (mJ)
1 (sink)	Theoretical model	11.0	53.1	749.9	2,284.8
	Simulated results	9.5	53.0	800.4	2,281.7
2 (app.)	Theoretical model	231.7	54.4	969.9	2,273.8
	Simulated results	232.9	53.3	1,056.6	2,270.3
3 (app.)	Theoretical model	185.3	41.4	918.1	2,275.8
	Simulated results	185.3	60.5	1,005.3	2,272.5
4 (app.)	Theoretical model	138.8	28.4	866.2	2,279.9
	Simulated results	142.4	28.2	920.1	2,276.2
5 (app.)	Theoretical model	92.4	15.4	814.4	2,279.9
	Simulated results	94.9	20.2	866.8	2,278.8
6 (app.)	Theoretical model	46.0	2.4	762.5	2,281.9
	Simulated results	48.6	2.2	780.4	2,282.2

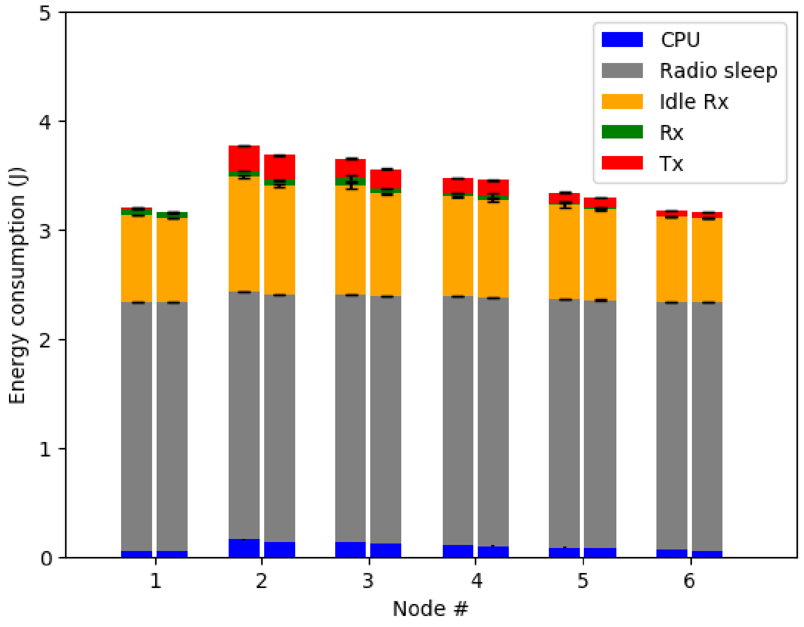


Figure 7.3 – Scenario 4: energy consumption in linear topology with sink (node 1) and all five nodes generating messages.

The differences observed on the bars from Figure 7.2 and Figure 7.3 indicates that the closer a router is to the data sink, the higher is its energy consumption. This behavior is expected since devices closer to the sink in a linear or tree topology are more likely to relay packets than further away nodes.

Thus, the results presented in Table 7.3 and Table 7.4 confirm that the method to measure energy consumption in the simulator is coherent to the theoretical model; thus, it can be used for further simulations.

Furthermore, we observed that the amount of energy spent on the sleep state is higher than other states. However, this is coherent, since the used sleep mode consumed $426\,\mu A$, the duty cycle obtained in these simulations was between 0.3% to 1.3%, and simulations were 30 minutes long.

7.2 IOTUS PERFORMANCE COMPARED TO RIME STACK

7.2.1 Scenario 5: 10 nodes, static routing network, and tree topology

In this scenario, we evaluated IoTUS in terms of its energy efficiency and memory footprint. Moreover, its setup is composed as:

- **Topology**: tree with a single root node (data sink);
- **Data Link Layer**:
  - *Protocol*: ContikiMAC;
  - *Events*: *Piggyback Service* may aggregate packets flowing to the same destination;
- **Network layer**:
  - *Protocol*: static routing table;
  - *Events*: Keep Alive packets generation in all devices every 30 seconds;
- **Application layer**:
  - *Protocol*: None;
  - *Events*: message generation in all devices every 30 seconds;

While latency is an important performance metric for delay-sensitive applications, it is not considered to be critical for environmental monitoring (an important motivation application for this thesis). Though we do not report latency results in this work, we note that additional processing incurred by IoTUS (e.g., packet construction, data aggregation) may increase latency. For example, the duration between building a message at the application layer and finally transmitting it was on average $4.1\,\mathrm{ms}$ with IoTUS, while the Rime stack processed the same request in $1.3\,\mathrm{ms}$. Depending on the end-to-end propagation delay, this difference may be disregarded. According to [46], Riot [45] and TinyOS [39] took around $3\,\mathrm{ms}$ to build a transmitting packet, while ContikiOS [16] resulted in around $1.5\,\mathrm{ms}$. ContikiOS (and the adapted Rime stack defined in this work) uses a static centralized array buffer to store packets (*Packetbuf*), while Riot and TinyOS use dynamically allocated memory and linked list to store their packets. IoTUS not only uses a dynamic linked list to store its packets but also adds more processing with its management; thus, the longer building time of a transmitting packet is caused by the additional IoTUS's modules processing. As part of our future work, we will be evaluating IoTUS impact on latency.

Figure 7.4 plots energy consumption averaged over all 44 network nodes using the topology illustrated in Fig. 5.5 for both IoTUS and Rime over time. The shaded areas represent the confidence interval of each line according to their colors.

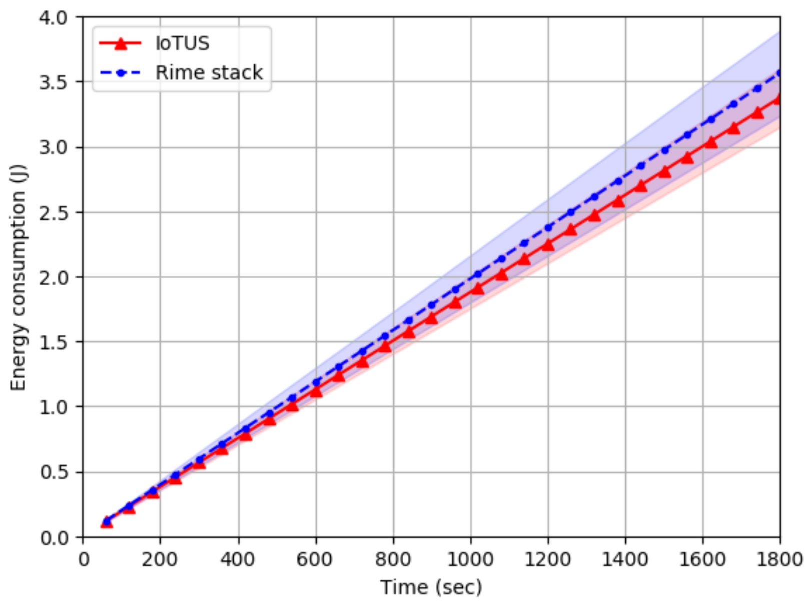


Figure 7.4 – Scenario 5: average energy consumption per node in a 44-node tree network.

We ran the simulations for 30 minutes to allow enough time for the system to reach steady state. Although IoTUS's average energy consumption gain was about 5.33%, as shown in

Figure 7.6, nodes 2 and 3 experienced energy consumption gains of 13% each. Note that these are the closest nodes to the root of the tree and, thus, are the ones that need to forward the highest number of packets on their way to the sink.

Figure 7.6, which plots consumption by power state for selected nodes in the network, shows that most of nodes' 2 and 3 energy consumption gains by IoTUS come from the radio, more specifically by spending less time in transmission mode when compared to the Rime stack. This is mainly due to IoTUS's aggregation feature provided by the *Packet Manager* and *Piggyback Service*. For comparison purposes, node 43 (a leaf node) had the overall minimum consumption. Figure 7.5 also shows energy consumption for other nodes, which are a mix of intermediate and leaf nodes.

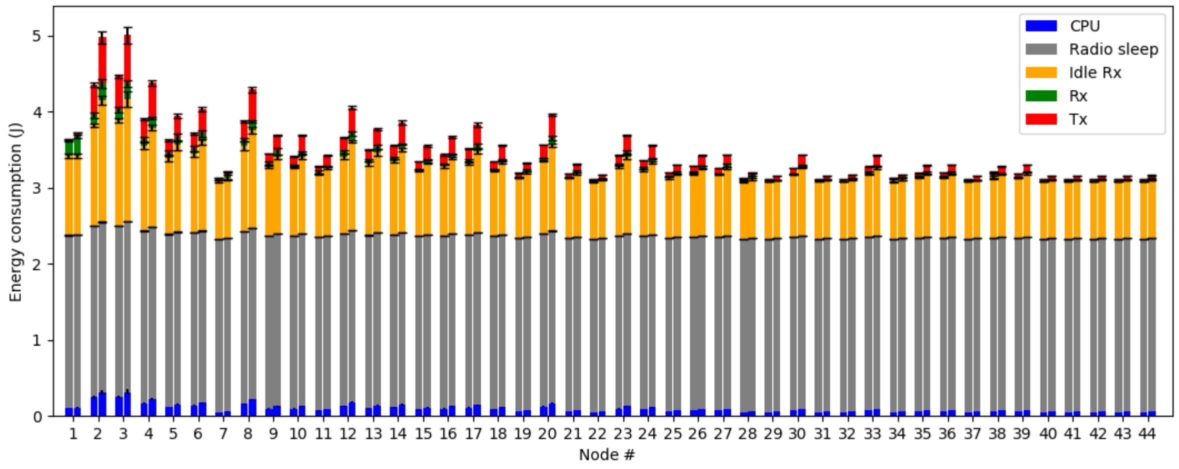


Figure 7.5 – Scenario 5: total energy consumption by states of all 44 nodes network. The left side bar of each node describes the IoTUS framework consumption. The right side bar indicates the consumption of the node for the Rime stack.

As expected, for this simulation, radio functions are by far the most energy consuming. Even though active states can consume thousands more than sleep state and up to 10 times more than CPU, Figure 7.6 reveals that nodes' radios spent most of the time in either idle or sleep, and these two states contributed the most to the overall energy consumed by nodes.

This scenario was also simulated for small trees. Thus, Figure 7.7 shows the maximum and minimum energy consumption gains attained for networks of varying size. These results were obtained by using trees of sizes 2, 8, 14, 20 nodes, etc. where each tree corresponds to the tree shown in Figure 5.5 but only including nodes up to node id 2, 8, 14, 20, etc., respectively.

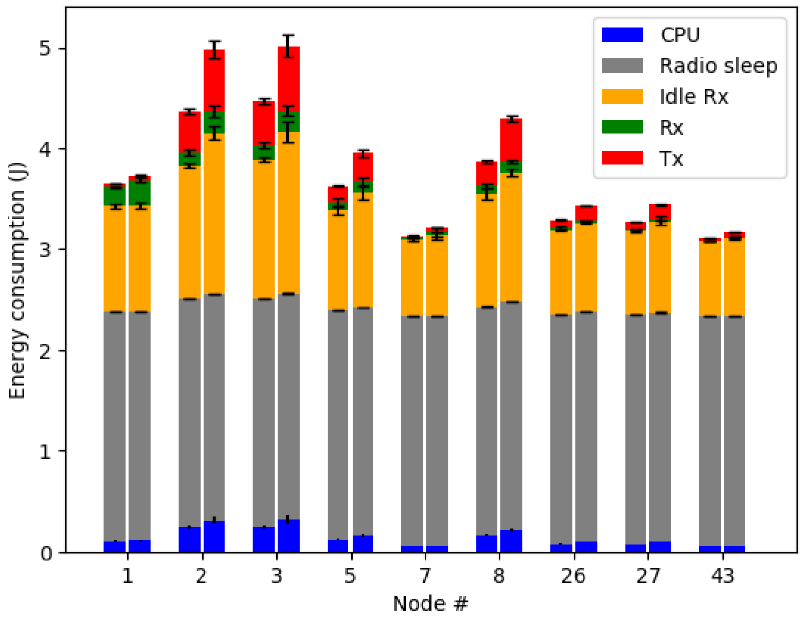


Figure 7.6 – Scenario 5: total consumption by states of selected nodes from Figure 7.5.

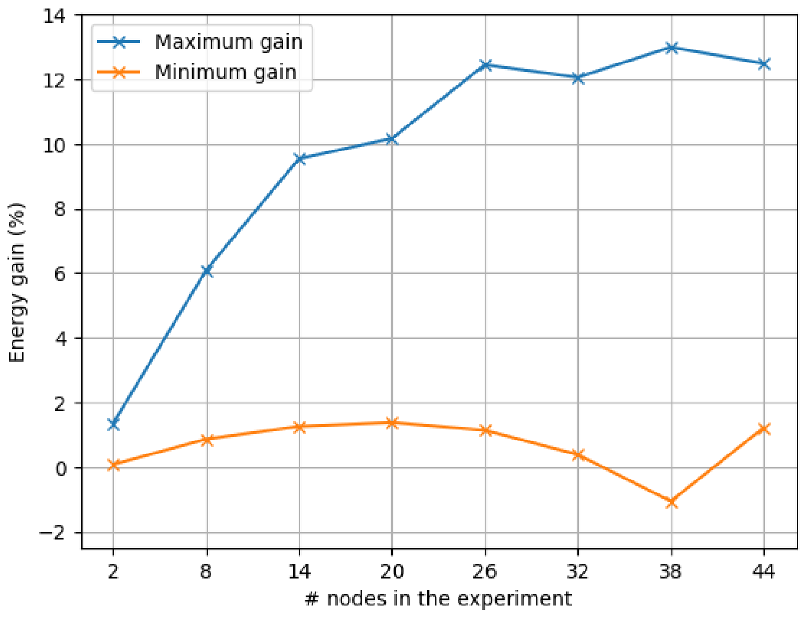


Figure 7.7 – Scenario 5: maximum and minimum radio energy gain per nodes in the network (over 10 simulation runs).

We observe that, as the network increases in size, so does the energy consumption gains obtained using IoTUS. As previously discussed, the highest gains were observed by nodes 2 and 3, which also had the highest energy consumption, according to Figure 7.6. Since these nodes are the most likely to be the first to have their battery depleted, guaranteeing them the highest energy savings results in extending the network's overall lifetime.

To calculate lifetime, we consider that each device is supplied with a NiMH battery [64] (e.g., 2000 mAh at 3 volts) that would provide up to

$$\begin{aligned} Bat &= 2000\,\text{mA} * V_{PP} * 60\,\text{min} * 60\,\text{s} \\ &= 2\,\text{A} * 3\,\text{V} * 60\,\text{min} * 60\,\text{s} \\ &= 21.6\,\text{kJ}. \end{aligned} \tag{7.1}$$

Thus, a node's Lifetime (LT) is obtained by

$$Lifetime(LT) = \frac{Bat}{Node_{Power}}, \tag{7.2}$$

where $Node_{Power}$ is the average power consumption of a device.

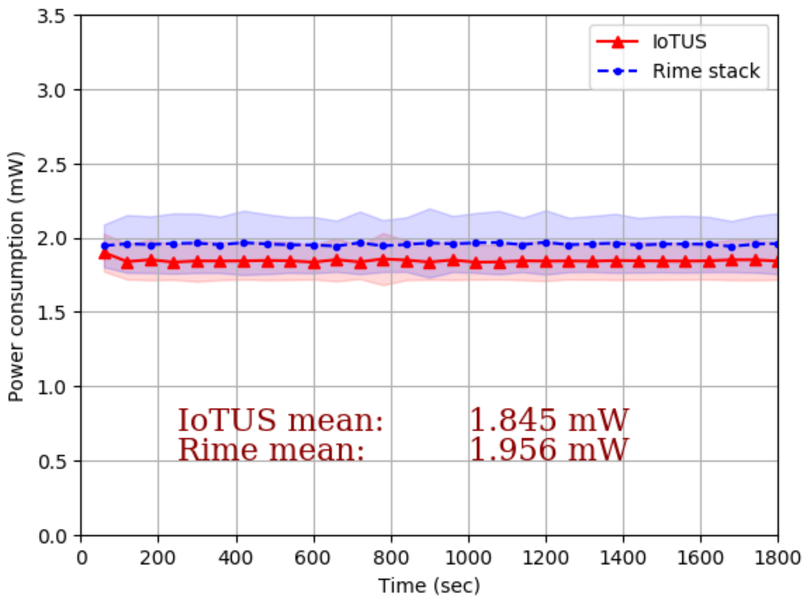


Figure 7.8 – Scenario 5: average power consumption of the network.

Figure 7.8 plots power consumption averaged over all 44 network nodes, and Figure 7.9 presents results for expected network lifetime for tree topologies of varying sizes. Network lifetime is defined as the time between the start of the simulation until any one of the nodes' battery is depleted.

Considering this scenario, power modes and the fact that nodes 2 and 3 did exhibit the highest energy consumption, IoTUS achieves a lifetime of $8,700,322$ s ($\approx$ 100 days), representing an improvement of 12.11% over Rime ($7,760,324$ s$\approx$ 89 days). Furthermore, for

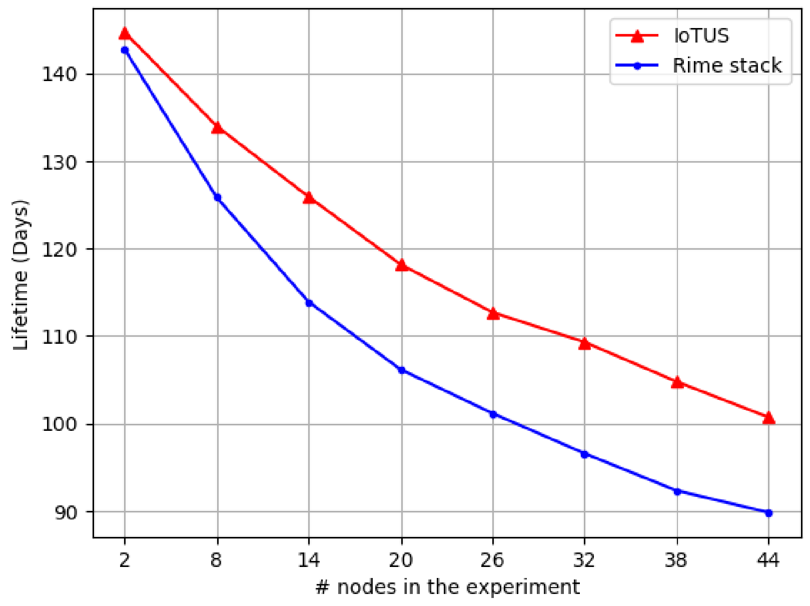


Figure 7.9 – Scenario 5: maximum lifetime per number of nodes in the network.

this particular topology, once nodes 2 and 3 deplete their batteries, the rest of the network gets disconnected from the sink and therefore is no longer able to perform its monitoring task.

We should point out that, for the static routing setup of Scenario 5, we were not able to run simulations using trees with more than 44 nodes. Since the code is compiled for all nodes at the same time, the static routing table saved in each device gets bigger with the network size. The RAM size needed by routing table in the adapted Rime stack exceeded the 10 KByte of TMote Sky's RAM capacity. On the other hand, larger tree topologies could still be emulated when using IoTUS, which attests to its efficient memory footprint. IoTUS offers additional tools for protocols in the stack and impacts on memory usage, but network scalability is mostly limited by the protocols' design. In this way, IoTUS requires an initial flash memory to be installed, but it reduces the amount of used RAM and allows more neighbors to be stored; thus, it possibly allows larger networks compared to the adapted Rime stack.

Figure 7.10 shows a comparative memory footprint characterization between IoTUS and Rime as packet buffer size increases. While IoTUS requires additional flash space (less than 5 kbytes, or 18% more than Rime), it saves RAM storage through its ability to share information across layers and avoids information duplication. As shown in Figure 7.10, IoTUS's memory footprint savings increases with the size of the packet buffer. In this simulation, memory saving reaches 23.63% for a packet buffer size of 15 packets.

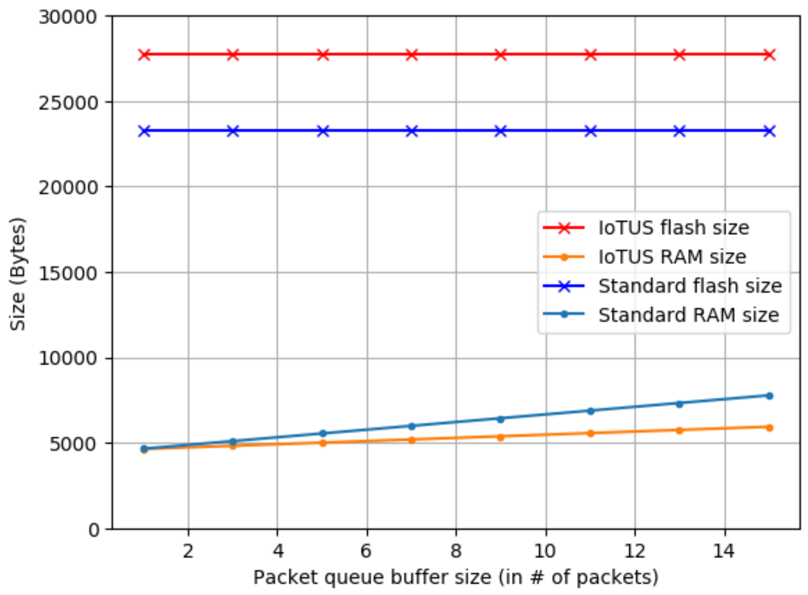


Figure 7.10 – Scenario 5: memory usage when increasing packet buffer capacity.

7.2.2 Overhead simulation validation

The simulation of Scenarios 6, 7, 8, 9, and 10 will be using the *Neighbor Discovery* module of IoTUS framework. Therefore, the routing table of the Network layer will be generated at runtime. Also, the Data Link layer and Network layer will be executing association procedures along with Keep Alive packet and application messages (after connected). Hence, the configuration of this scenario is:

- **Topology**: 2 nodes, star, and linear;
- **Data Link Layer**:
  - *Protocol*: ContikiMAC802 (has association procedure and periodic broadcasts);
  - *Events*:
    - *Neighbor Discovery* aggregates the association frames and helps the connection process;
    - After connected, router nodes will be broadcasting DL beacon frames;
    - *Piggyback Service* may aggregate packets flowing to the same destination;
- **Network layer**:
  - *Protocol*: RPL-Like;
  - *Events*:

        * *Neighbor Discovery* aggregates the association frames and helps the connection process;
        * After connected, router nodes will be broadcasting DL beacon frames;
        * Keep Alive packets generation in all devices every 30 seconds;

- **Application layer**:
    - *Protocol*: None;
    - *Events*: After connected, there will be message generation in all devices every 30 seconds;

Cooja simulator [22] can provide the log of communication of a simulation. Therefore, this log contains every packet transmitted during the simulation, including collided packets, retransmissions, ACKs, etc. However, according to the definition of overhead adopted in Chapter 6, collided packets and retransmissions should not happen.

Hence, to validate our developed overhead measuring tool, we do not consider the collided and/or retransmitted packets. Consequently, only the successful transmitted packets are accounted for overhead. This filtering process also improves the comparison analysis between IoTUS framework and Rime stack, since the influence of protocols' design (e.g., ContikiMAC and Carrier Sense Multiple Access) will be reduced. Note that we always use the same configuration, parameters, and filtering process for both systems: the stack using IoTUS framework and for Rime stack.

Scenario 6: 2 nodes and neighbor discovery procedure

The overhead measuring tool parses and averages the whole log files of the 10 runs, counting every byte in all the messages. However, as discussed before, we account only the successfully delivered packets. When the network is completely connected (routing paths are established), we calculate the *pure header overhead* (defined in Chapter 6). Finally, with the complete 30 minutes log, we calculate the final average overhead.

Scenario 6 contains only one transmitting node and one receiver (data sink). Therefore, Figure 7.11 presents the results of *Pure header overhead*, in which DL and Network layers are discriminated. Also, the final overhead calculated with the 30 minutes log files is presented in Figure 7.12, when events of application message are already generated. Finally, Table 7.5 compares these results with the theoretically expected values.

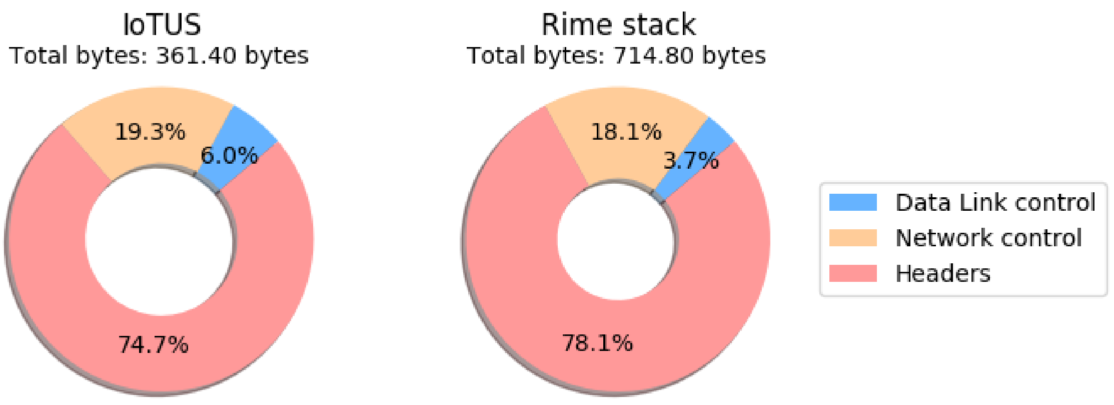


Figure 7.11 – Scenario 6: *Pure header overhead* comparison for two nodes.

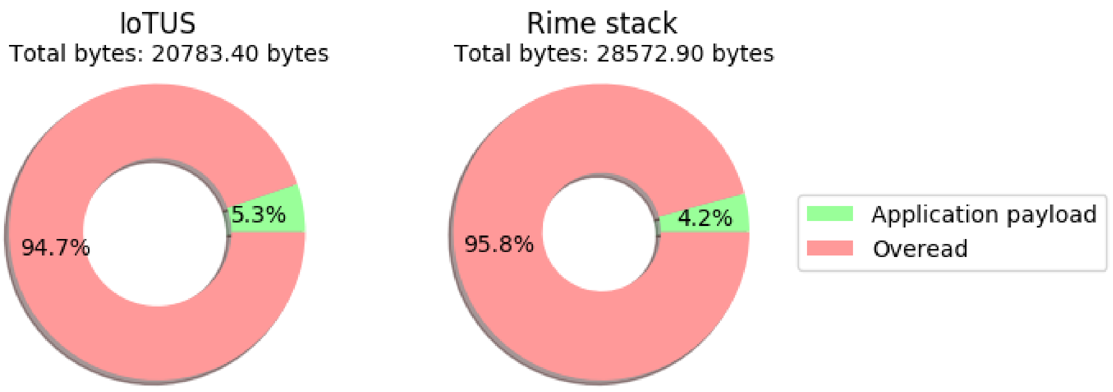


Figure 7.12 – Scenario 6: overhead comparison for two nodes.

Table 7.5 – Scenario 6: *Pure header overhead* and Overhead validation.

Type	Method	IoTUS	Rime stack
Pure header overhead	Theoretical model	73.7%	77.8%
	Simulated results	74.7%	78.1%
Overhead	Theoretical model	94.3%	96.0%
	Simulated results	94.7%	95.8%

As observed, the expected results were quite close to the simulations. Also, it can be seen that the Network layer had a greater amount of command bytes exchanged, which is coherent to the packet sizes presented in Table 5.3 and Table 6.6. Moreover, in general, IoTUS framework spends less total bytes than Rime stack to do the same procedure.

With these results, we also obtained the goodput for this network, which is defined as the amount of application payload bytes per second received at the sink node. Thus, IoTUS and Rime stack had a similar final goodput of 0.6 bytes/s.

Scenario 7: 10 nodes, neighbor discovery procedure, and star topology

With the overhead measuring tool validated for two nodes, we expanded the network up to 10 devices in Scenario 7, but in a star topology. In this case, the network nodes compete for the transmitting slot, but since many slots are available and collisions are not accounted, results are an extension of the Scenario 6. Therefore, the overhead calculated with the 30 minutes log files is presented in Figure 7.13, and Table 7.6 compares these results with the theoretically expected values.

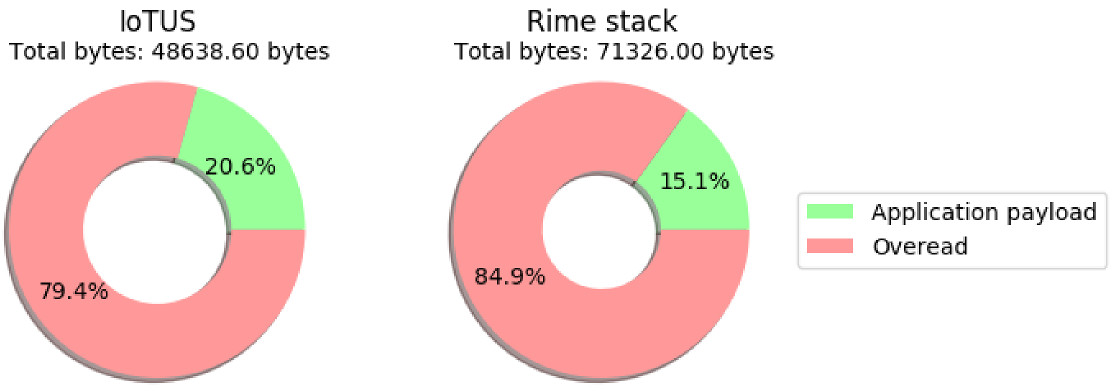


Figure 7.13 – Scenario 7: overhead comparison for 10 nodes and star topology.

Table 7.6 – Overhead validation in Scenario 7.

Type	Method	IoTUS	Rime stack
Overhead	Theoretical model	79.1%	85.6%
	Simulated results	79.4%	84.9%

In terms of the average amount of header bytes exchanged, IoTUS spent approximately $38,619$ bytes (79.4%) during the 30 minutes of the simulation, while Rime stack spent $60,555$ bytes (84.9%). Meanwhile, the goodput measured in IoTUS was 5.49 bytes/s, while rime stack had 5.88 bytes/s. Thus, for a similar goodput, IoTUS used fewer bytes, a difference of $21,936$ bytes and equivalent to 36.22%.

Scenario 8: 10 nodes, neighbor discovery procedure, and linear topology

The last topology analyzed with the theoretical model is given in Scenario 8, which has a linear topology. In this case, only the last node transmits application messages after connected by DL layer and Network layer.

Table 7.7 – Scenario 8: *Pure header overhead* and Overhead validation.

Type	Method	IoTUS	Rime stack
Pure header overhead	Theoretical model	73.5%	80.3%
	Simulated results	73.3%	80.8%
Overhead	Theoretical model	95.7%	97.3%
	Simulated results	96.5%	97.3%

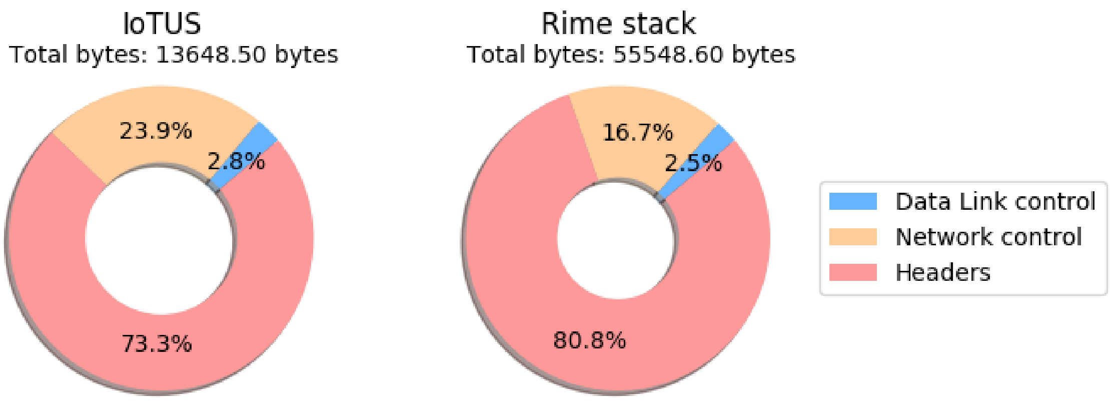


Figure 7.14 – Scenario 8: *Pure header overhead* comparison for 10 nodes and linear topology.

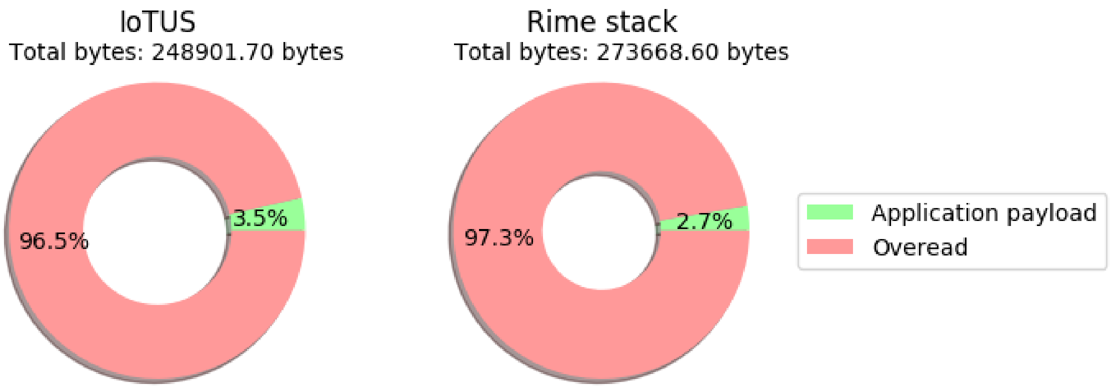


Figure 7.15 – Scenario 8: overhead comparison for 10 nodes and linear topology.

Therefore, Figure 7.14 presents the results of *Pure header overhead*. Also, the final overhead calculated with the 30 minutes log files is presented in Figure 7.15. Finally, Table 7.7 compares these results with the theoretically expected values.

In the linear topology, node "n+1" has to wait for the node "n" to be connected, and so the association process takes longer than the star topology. Therefore, more periodic control frames are transmitted during the association stage.

Once again, in terms of the average amount of header bytes exchanged, IoTUS spent approximately $240,190$ bytes (96.5%) during the 30 minutes of the simulation, while Rime stack spent $266,279$ bytes (97.3%). Meanwhile, the goodput measured in IoTUS was 0.53 bytes/s, while rime stack had 0.44 bytes/s. Thus, IoTUS used fewer bytes with a better goodput, a difference of $26,089$ bytes and equivalent to 9.79% improvement.

7.2.3 Energy consumption, memory usage, lifetime, etc.:

Scenario 7: 10 nodes, neighbor discovery procedure, and star topology

Figure 7.16 plots energy consumption averaged over all 10 network nodes using the star topology illustrated in Fig. 5.6. Also, after 30 minutes of simulation, Figure 7.17 presents the energy consumption per state of each node.

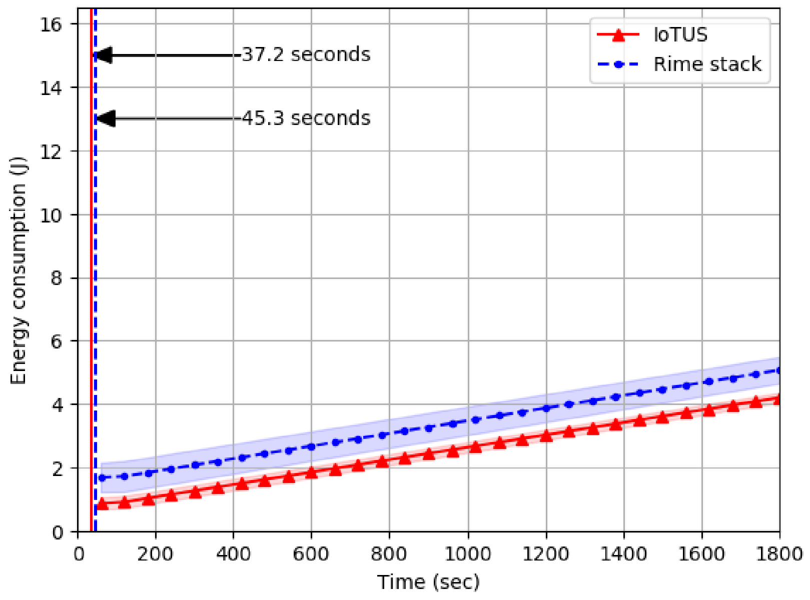


Figure 7.16 – Scenario 7: average energy consumption per node in a 10-node star network.

According to Figure 7.17, for IoTUS and Rime stack, all transmitting devices but the sink

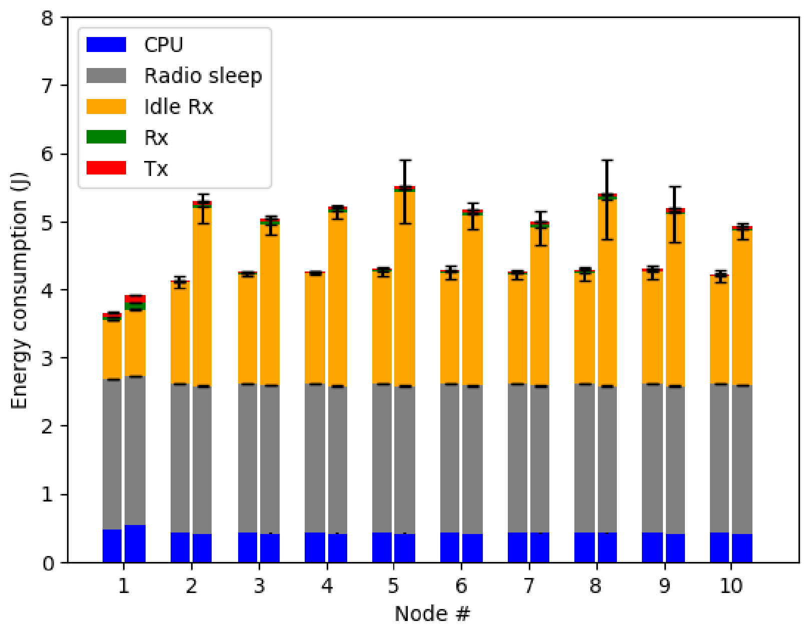


Figure 7.17 – Scenario 7: total energy consumption by states of all 10 nodes network. The left side bar of each node describes the IoTUS framework consumption. The right side bar indicates the consumption of the node for the Rime stack.

node had similar energy consumption. However, this occurred because all devices but sink had the same function (leaf nodes) and the same number of hops from the data sink. Still, IoTUS framework obtained an average gain of approximately 19.51% compared to the Rime stack. Moreover, the average network association time, illustrated in Figure 7.16 as vertical lines, was faster in the environment using IoTUS ($37.2\,\mathrm{s}$) than in the Rime Stack ($45.3\,\mathrm{s}$).

Scenario 7 was also simulated for a small number of nodes. Thus, Figure 7.18 illustrates the final energy consumption averaged over the N nodes in each size of the network. Also, Figure 7.19 shows the maximum and minimum energy consumption gains attained for networks of varying size.

According to Figure 7.19, for the star topology in Scenario 7, IoTUS attained a maximum gain over Rime in the network with three nodes (25.07%). However, due to the number of communication slots per message transmission period available in this scenario, the energy consumption per number of nodes in the network does not increase significantly, as shown in Figure 7.18.

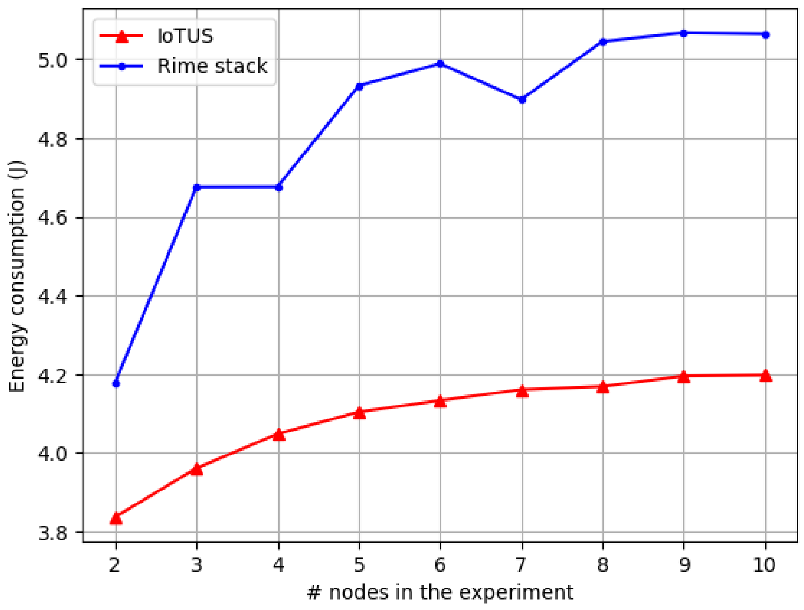


Figure 7.18 – Scenario 7: final average energy consumption.

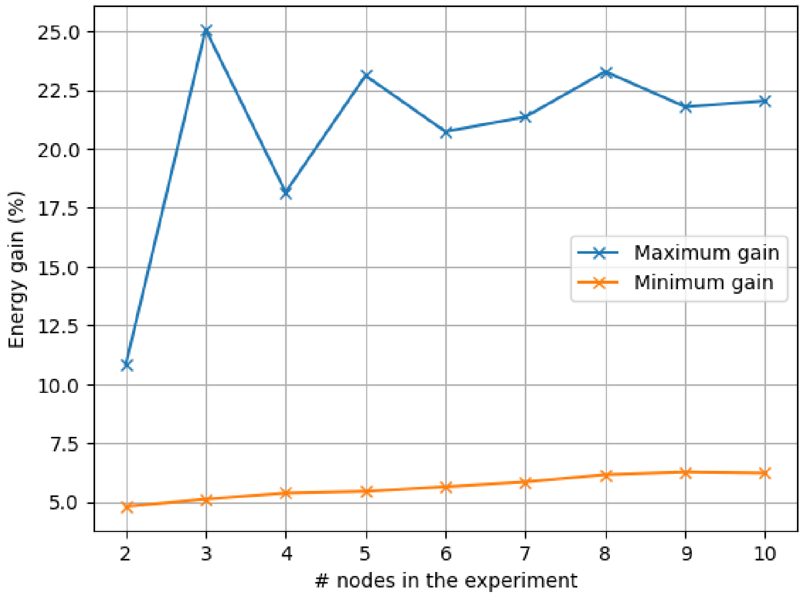


Figure 7.19 – Scenario 7: maximum and minimum radio energy gain of IoTUS over Rime, per nodes in the network.

Scenario 8: 10 nodes, neighbor discovery procedure, and linear topology

In this scenario, the benefits of sharing procedures and other information between layers is shown by the impact on energy consumption and association time. Therefore, Figure 7.20

plots energy consumption averaged over all 10 network nodes using the linear topology illustrated in Figure 5.2. Also, after 30 minutes of the simulation, Figure 7.21 presents the energy consumption per state of each node. Note that only the end node (device 10) transmits application message for this linear topology.

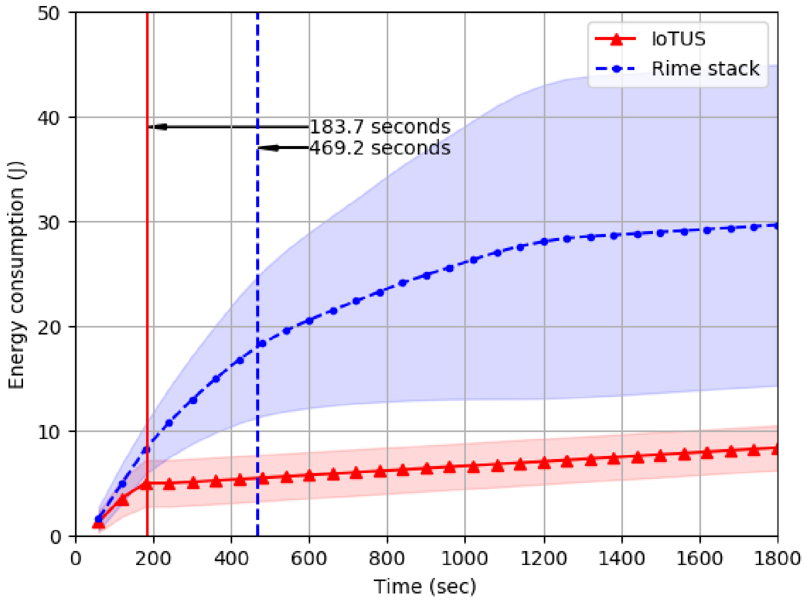


Figure 7.20 – Scenario 8: average energy consumption per node in a 10-node linear network.

It can be observed that both stacks with and without IoTUS framework had a big increment of energy consumption. On IoTUS, after 30 minutes of simulation, the final energy consumption was 8.35 J, while in Rime stack it reached 29.64 J. These values are very high compared to the other scenarios because many disconnections and scanning procedures occur, which also increase confidence interval. In IoTUS, these association of both layers is enhanced by *Neighbor Discovery* service; therefore, IoTUS provided fewer disconnections and more stable association through the network. Also, IoTUS had an average gain of 71.81%.

In this Scenario 8, the devices have to associate to two layers before being ready to broadcast and let the next node connect in sequence. Since the neighbor discovery process of the second and third layers are independent, many disconnections occur. For example, consider node "n-1" already connected in both layers, then node "n" will scan "n-1" broadcast frames and start its DL layer association. After node "n" gets connected in DL, it will start broadcast DL frames and will start an association process for its Network layer. However, the node "n+1" recognizes node "n" and starts another DL association too. Hence, node "n+1" DL connection can slow node "n" Network layer connection. Consequently, Figure 7.21

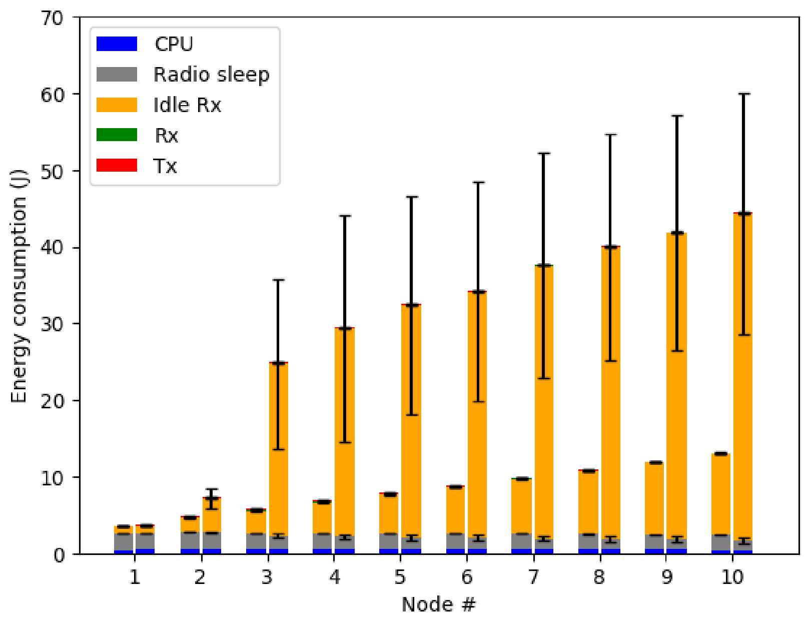


Figure 7.21 – Scenario 8: total energy consumption by states of all 10 nodes network. The left side bar of each node describes the IoTUS framework consumption. The right side bar indicates the consumption of the node for the Rime stack.

show the big increment of energy consumption from node 2 to node 3.

Another important event in this Scenario 8 is that by the time node 3 gets its network association, all other nodes (4 to 10) are already connected within the Data Link layer. Therefore, the step of energy consumption is smaller from node 3 and beyond. Consequently, the whole process of network connection in Rime stack takes 469.2 s (Theoretical model estimated 468 s). Meanwhile, in IoTUS, *Neighbor Discovery* module improves the association procedure of both layers. Therefore, IoTUS had a smoother step between nodes in a linear topology and also provided a faster network connection of 183.7 s (Theoretical model estimated 180 s).

Scenario 8 was also simulated for a small number of nodes. Figure 7.22 illustrates the final energy consumption averaged over the N nodes in each size of the network. Figure 7.23 shows the maximum and minimum energy consumption gains attained for networks of varying size.

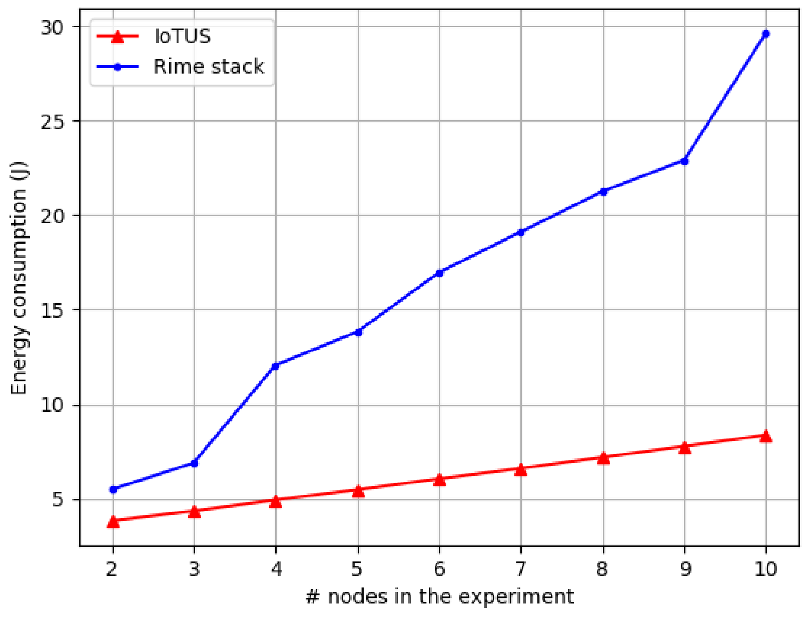


Figure 7.22 – Scenario 8: final average energy consumption.

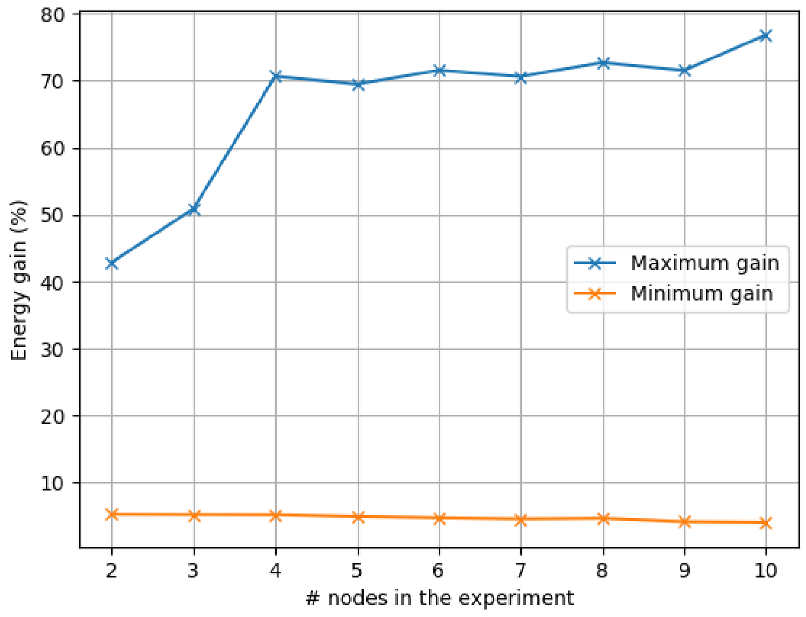


Figure 7.23 – Scenario 8: maximum and minimum radio energy gain per nodes in the network.

Therefore, according to Figure 7.23, for the star topology in Scenario 8, IoTUS attained a maximum gain over Rime in the network with nine nodes (76.83%). Moreover, Figure 7.22 shows the impact of multiple hops when using IoTUS framework and Rime stack.

Scenario 9: 44 nodes, neighbor discovery procedure, and tree topology

Scenario 9 is composed of 44 nodes in a tree topology (Figure 5.5), which has mixed similarities of the star and linear topologies. Thus, in this case, nodes will compete for transmission slots and will have to wait for their predecessor node to get connected first.

Therefore, Scenario 9 also had an overhead evaluation, and so Figure 7.24 presents the *Pure header overhead* obtained in these simulations, as well as Figure 7.25 presents the final average overhead after 30 minutes of simulations.

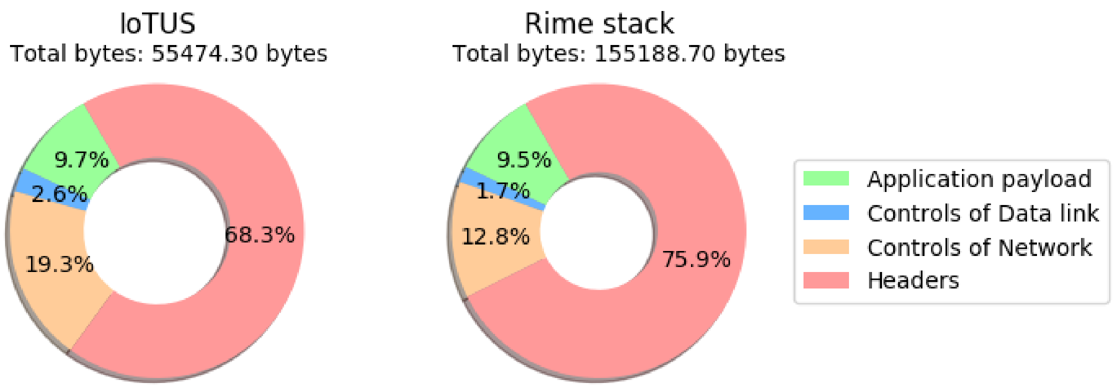


Figure 7.24 – Scenario 9: *Pure header overhead* comparison for 44 nodes and tree topology.

Thus, with only $37,888$ bytes (68.3%), IoTUS framework spent fewer header bytes to complete the network connection than Rime stack ($117,788$ bytes or 75.9%). Therefore, in the association stage for the tree topology, IoTUS obtained a gain of 67.83%.

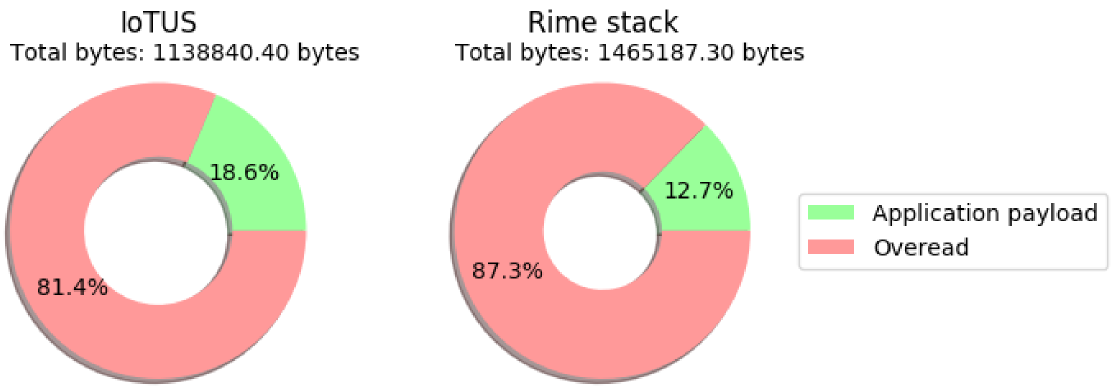


Figure 7.25 – Scenario 9: overhead comparison for 44 nodes and tree topology.

Meanwhile, regarding the average amount of header bytes exchanged after 30 minutes of the simulation, IoTUS spent approximately $927,023$ bytes (81.4%), while Rime stack spent

$1,279,108$ bytes (87.3%). Also, the goodput measured in IoTUS was 26.94 B/s, while Rime stack had 15.30 B/s. Thus, IoTUS used fewer bytes while keeping an improved goodput, resulting in a difference of $352,085$ bytes or a gain of 27.52%.

For energy evaluation, Figure 7.26 plots the energy consumption averaged over all 44 network nodes for both IoTUS and Rime over time. Therefore, IoTUS's average energy consumption gain was approximately 34.88%. Moreover, Figure 7.27 plots consumption by power state for selected nodes in the network.

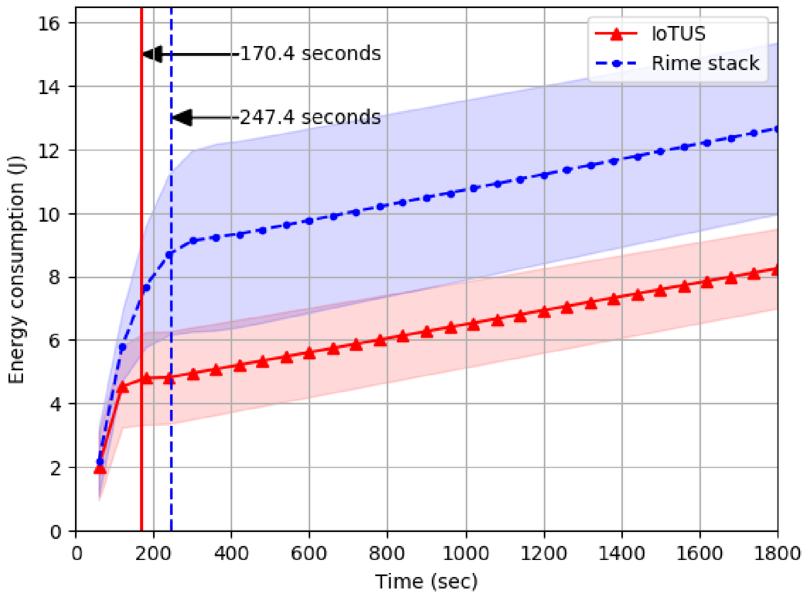


Figure 7.26 – Scenario 9: average energy consumption per node in a 44-node tree network.

In Scenario 9, the addition of an association process and neighbor discovery procedure altered the energy results of Figure 7.27 when compared to Scenario 5 (static routing tree topology). Hence, in Scenario 9, further distant nodes presented the highest consumption (e.g., node 42). Consequently, the node with the highest gain in this 44 devices tree topology was node 32 (43.25% gain compared to Rime stack).

For this scenario, the gains obtained by IoTUS come mostly from the radio, more specifically by spending less time in transmission mode when compared to the Rime stack. IoTUS's aggregation feature provided by the *Packet Manager*, *Piggyback Service*, and *Neighbor Discovery* module caused most of the gains compared to Rime stack. These IoTUS's modules also improved association time, which is illustrated by vertical lines in Figure 7.26. Thus, the new framework completed the network connection in 170.4 s, while Rime stack had an average of 247.4 s.

Scenario 9 was also simulated for a small number of nodes; thus, Figure 7.28 presents

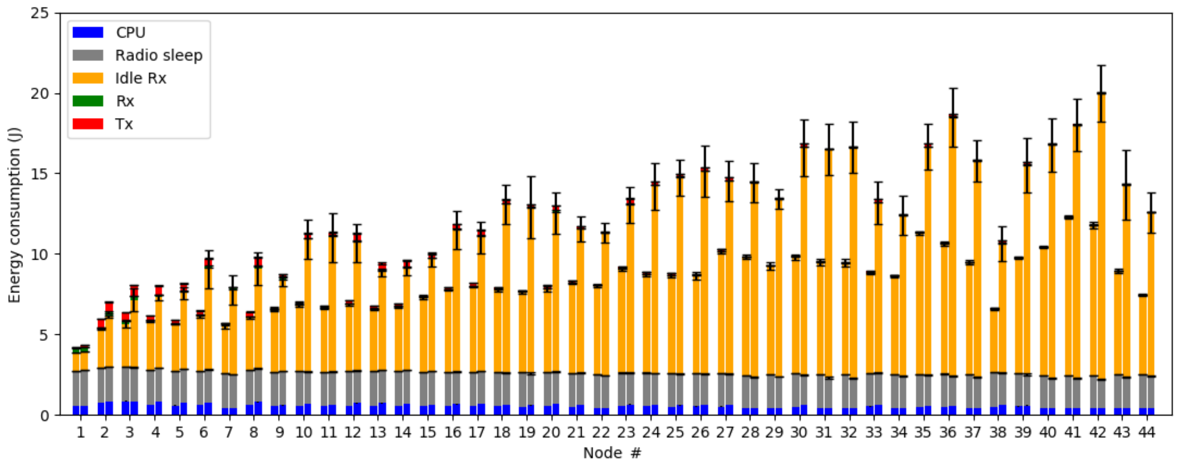


Figure 7.27 – Scenario 9: total energy consumption by states of all 44 nodes network. The left side bar of each node describes the IoTUS framework consumption. The right side bar indicates the consumption of the node for the Rime stack.

the final energy consumption averaged over the N nodes in each network size. Also, Figure 7.29 shows the maximum and minimum energy consumption gains attained for networks of varying size.

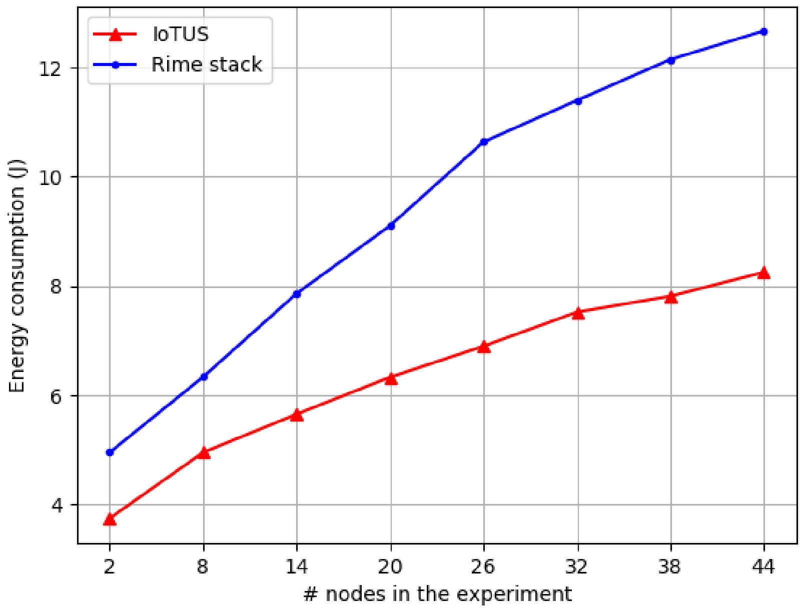


Figure 7.28 – Scenario 9: final average energy consumption with 44 nodes, neighbor discovery, and tree topology.

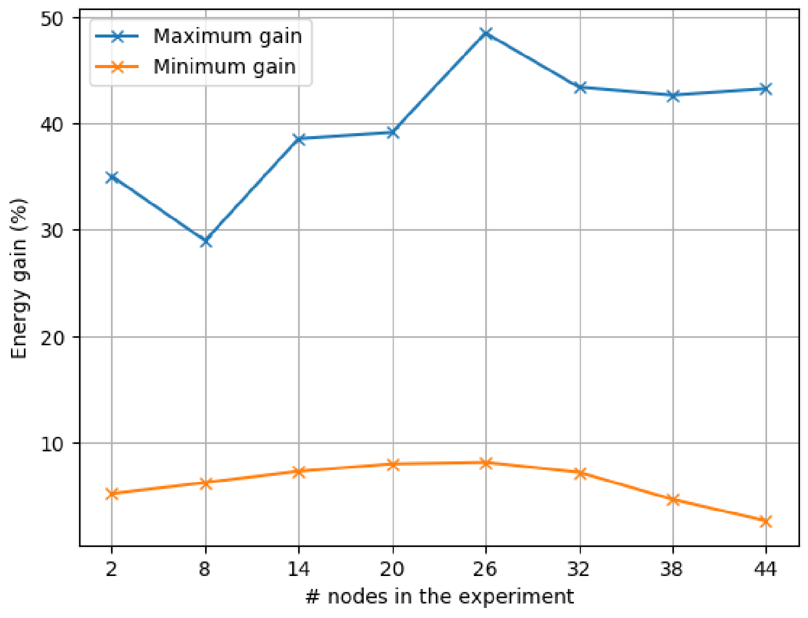


Figure 7.29 – Scenario 9: maximum and minimum radio energy gain per nodes in the network with 44 nodes, neighbor discovery, and tree topology.

Therefore, according to Figure 7.28, increasing the tree topology number of nodes drastically impacts on the final energy consumption. However, maximum nodes' gain over Rime stack does not have the same rate, and thus, the simulation with 26 nodes obtained the highest gain, i.e., IoTUS framework consumed a total of 7.479 J, while Rime resulted in 14.522 J (a gain of 48.50%).

Figure 7.30 presents the average network power consumption, which also illustrates the moment a network achieves steady state. Hence, it can be seen that Rime stack takes longer to reach steady state, and after that, it still spends more power.

Figure 7.31 presents results for expected network lifetime for tree topologies of varying sizes. Hence, IoTUS achieves a lifetime of $3,161,695$ s ($\approx$ 36 days), representing an improvement of 62.84% over Rime ($1,941,592$ s$\approx$ 22 days).

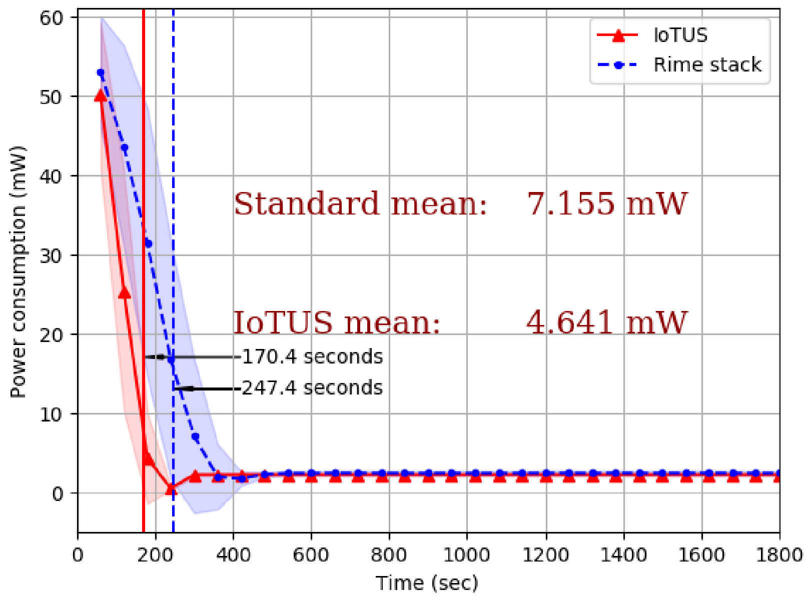


Figure 7.30 – Scenario 9: average power consumption of the network with 44 nodes, neighbor discovery, and tree topology.

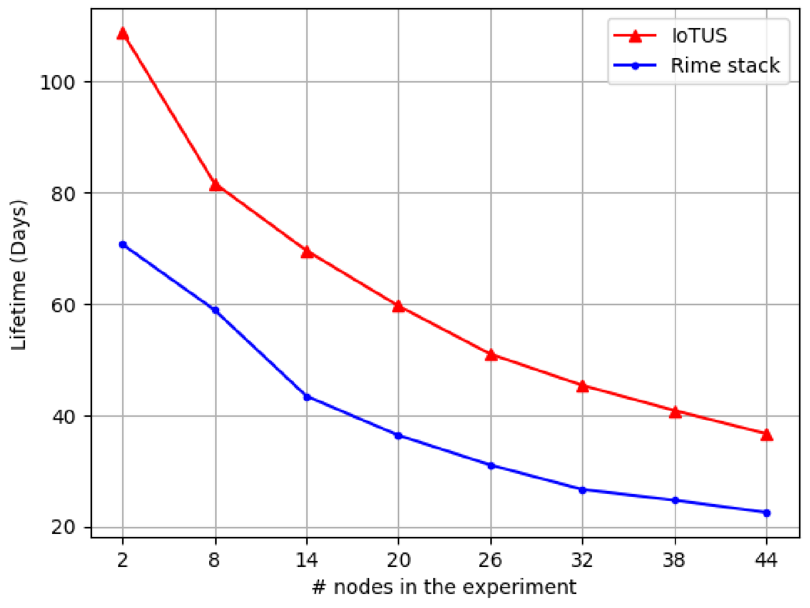


Figure 7.31 – Scenario 9: maximum lifetime per number of nodes in the network with 44 nodes, neighbor discovery, and tree topology.

Figure 7.32 shows a comparative memory footprint characterization between IoTUS and Rime as packet buffer size increases. In this scenario, IoTUS implementation plus the addition of protocols procedures (e.g., neighbor discovery, association, etc.) consumed a flash space of $30,462$ bytes, while Rime stack consumed $26,504$ bytes (less than 4 kbytes, or

14.9% more than Rime). Also, the RAM can consume up to $4,084$ bytes in IoTUS, while Rime consumed up to $5,946$ bytes. Therefore, it means that IoTUS consumed 31.31% less RAM for a packet buffer size of 15 packets.

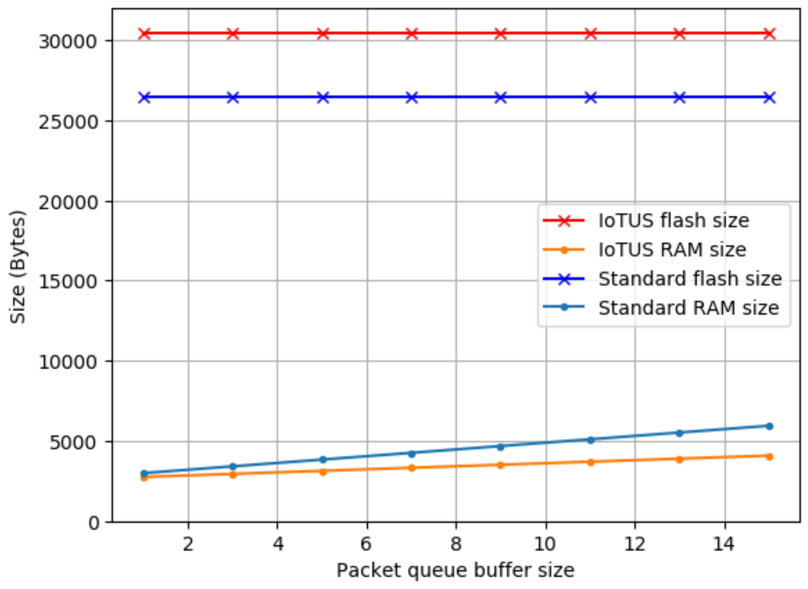


Figure 7.32 – Scenario 9: memory usage when increasing packet buffer capacity.

7.2.4 Scenario 10: 44 nodes, neighbor discovery procedure, tree topology, and radio's deep sleep

We simulated all the 9 scenarios using the default sleep mode ($0.426\,\mathrm{mA}$) in TMote Sky's implementation. However, regarding the environment sensing application, the deep sleep mode ($1\,\mu\mathrm{A}$) could be applied. Therefore, in Scenario 10, we estimate the energy results considering the radio's deep sleep mode instead. Therefore, Figure 7.33 plots the energy consumption averaged over all 44 network nodes for both IoTUS and Rime over time.

Since the sleep state is drastically reduced in deep sleep mode, IoTUS's average energy consumption gain was approximately 42.33%. Also, for a detailed analysis, Figure 7.34 plots consumption by power state for selected nodes in the network.

Furthermore, IoTUS in Scenario 10 obtained a lifetime of $3,774,280$ s ($\approx$ 45 days), while Rime stack achieved $2,135,360$ s ($\approx$ 24 days). Therefore, it represents 76.75% of lifetime improvement.

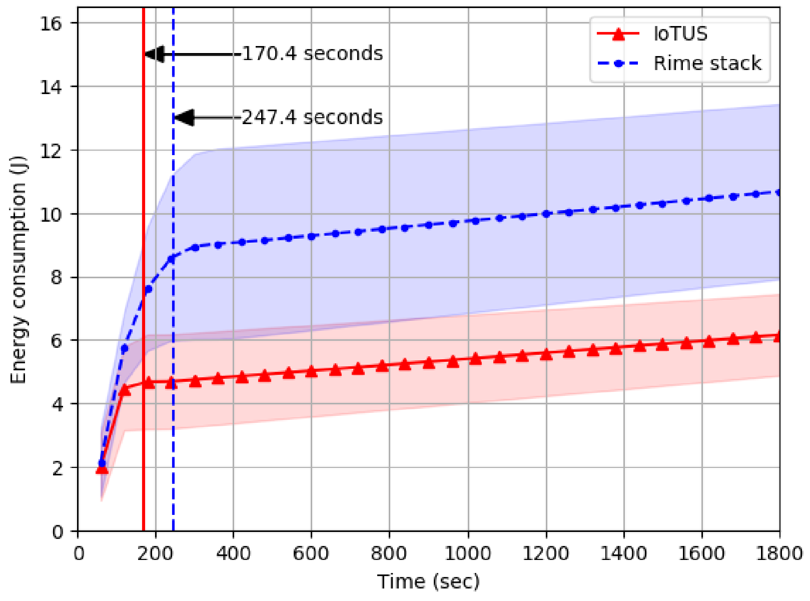


Figure 7.33 – Scenario 10: average energy consumption per node in a 44-node tree network.

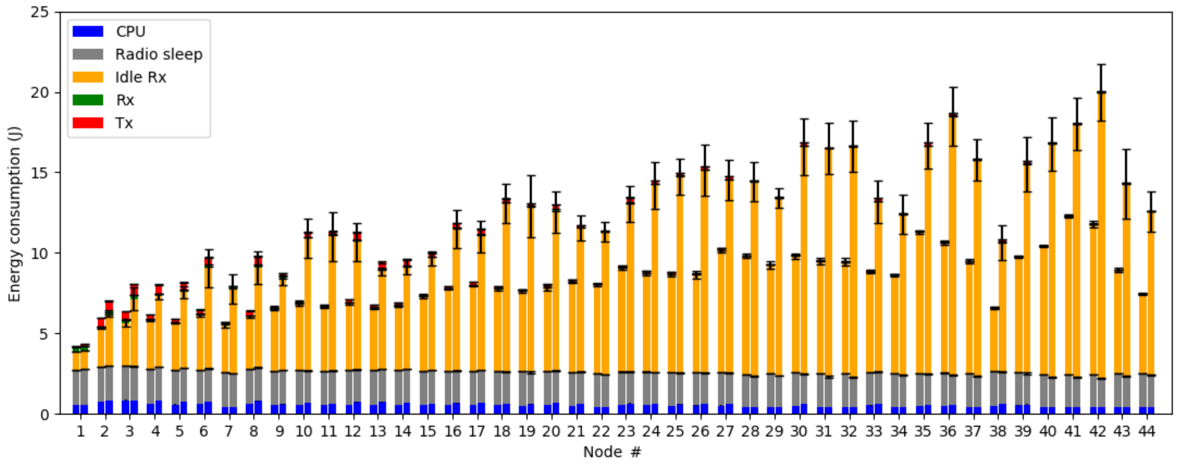


Figure 7.34 – Scenario 10: total energy consumption by states of all 44 nodes network. The left side bar of each node describes the IoTUS framework consumption. The right side bar indicates the consumption of the node for the Rime stack.

7.3 SIMULATION RESULTS' CONCLUSION

The simulated results comparing IoTUS framework and Rime stack [15] in this chapter are summarized into Table 7.8.

Therefore, compared to the static routing tree (Scenario 5), IoTUS framework improved energy efficiency over Rime while reducing its memory footprint (from 18% to 9.9%) and increasing the RAM efficiency (from 23.63% to 31.31%). Hence, with more protocols procedures in the stack, IoTUS improved memory usage.

These results confirm that the new IoTUS framework can improve network energy efficiency.

Summarizing the results in Table 7.8, we recapitulate here, for convenience, all scenarios described in this thesis:

- **Scenario 1**: single node transmitting broadcast packets;
- **Scenario 2**: one node transmitting, and other node receiving;
- **Scenario 3**: 6 nodes, linear topology, and only end node transmits app. message;
- **Scenario 4**: 6 nodes, linear topology, and 5 nodes transmitting app. message to sink;
- **Scenario 5**: 10 nodes, static routing network, and tree topology;
- **Scenario 6**: 2 nodes and neighbor discovery procedure;
- **Scenario 7**: 10 nodes, neighbor discovery procedure, and star topology;
- **Scenario 8**: 10 nodes, neighbor discovery procedure, linear topology, and only end node transmits app. message;
- **Scenario 9**: 44 nodes, neighbor discovery procedure, and tree topology;
- **Scenario 10**: 44 nodes, neighbor discovery procedure, tree topology, and radio's deep sleep.

Table 7.8 – Compared simulated results of IoTUS framework and adapted Rime stack.

Parameter	Scenario	IoTUS Framework	Rime stack	Approximated gain[1]
Packet building time	2	3.60 ms	1.28 ms	-81.25%
	5–9	4.12 ms	1.29 ms	-119.37%
Average final energy consumption	5	3.3742 J	3.5642 J	5.33%
	7	4.1976 J	5.0647 J	19.51%
	8	7.7666 J	22.9122 J	71.81%
	9	8.2454 J	12.6620 J	34.88%
	10	6.1535 J	10.6707 J	42.33%
Maximum energy consumption gain	5	4.1902 J	4.8155 J	13.0% at 38 nodes
	7	4.1419 J	5.5279 J	25.07% at 3 nodes
	8	5.7794 J	24.9426 J	76.83% at 10 nodes
	9	7.479 J	14.522 J	48.50% at 26 nodes
Network lifetime	5	8,700,322 s	7,760,324 s	12.11%, $\approx$11days
	9	3,161,695 s	1,941,592 s	62.84%, $\approx$14days
	10	3,774,280 s	2,135,360 s	76.75%, $\approx$19days
Memory usage	5	Flash: 27,712 B	Flash: 23,302 B	-18.92%
		RAM: 5,932 B	RAM: 7,768 B	23.63%
	9	Flash: 30,462 B	Flash: 26,504 B	-14.9%
		RAM: 4,084 B	RAM: 5,946 B	31.31%
Pure header overhead	6	74.7%	78.1%	4.35%
	7	79.2%	81.6%	2.94%
	8	73.3%	80.8%	9.28%
	9	68.3%	75.9%	10.01%
Overhead	6	94.7%	95.8%	1.14%
	7	79.4%	84.9%	6.47%
	8	96.5%	97.3%	0.82%
	9	81.4%	87.3%	6.75%
Goodput[2]	6	0.60 B/s	0.62 B/s	96.77%
	7	5.49 B/s	5.88 B/s	93.36%
	8	0.53 B/s	0.44 B/s	120.45%
	9	26.94 B/s	15.30 B/s	176.07%
Network association time	7	37.2 s	45.3 s	17.88%
	8	183.7 s	469.2 s	60.84%
	9	170.4 s	247.4 s	31.12%
Energy efficiency[3]	9	38.9 $\mu J/B$	68.4 $\mu J/B$	43.13%

[1]The gain column is given by "1-(IoTUS/Rime)".

[2]The gain equation for goodput is given by (IoTUS/Rime).

[3]Average final energy consumption per number of delivered application payload bytes (Figure 7.25).

8 CONCLUSION

In this paper, a new framework called IoT Unified Services framework, or IoTUS, was introduced. Its main goal is to facilitate sharing across protocol layers while preserving the benefits of layered protocol architectures, in particular, modularity and portability. To this end, IoTUS proposes an extensible *service layer*, that allows sharing of control plane information (e.g., collisions at the data-link layer, number of transmissions/receptions, radio packet size, ID address size), as well as sharing of services (e.g., neighbor discovery, network events log, data aggregation). Additionally, IoTUS can be used by existing network stacks without having to modify the basic operation of their protocols.

Since the framework is an addition to the general stack, we used proposed scenarios to evaluate its performance, which varies with complexity and number of motes. A traditional layered stack, which is a modification of ContikiOS's stack [16] (conveniently called adapted Rime), was used for comparison. Therefore, we evaluated Data Link (DL) and Network protocols implemented with and without IoTUS.

For the DL protocols, it was used ContikiMAC [27] and a modified version of it to fit IoTUS's design. Moreover, a ContikiMAC802 protocol was developed using features of ContikiMAC and the association process of IEEE 802.15.4 MAC [30]. For the Network layer, it was used a static routing protocol, and an RPL-Like protocol (developed using the association process of RPL [10]).

We implemented IoTUS on ContikiOS[16] and evaluated its performance using ContikiOS's Cooja network simulator. Also, we developed a theoretical model for energy and overhead measurements as a validation method. These models were applied to the star and linear topologies from 1 to 10 nodes. The theoretical results were then compared to the simulation results, which resulted in coherent measurements between both methods.

The network evaluated had sizes of two motes up to 44 nodes (limited by the static routing table in memory). These nodes formed the star and linear topologies, providing an environment similar to long-range applications and multiple-hops applications, respectively. The tree topology was also evaluated, which represents a general case of monitoring applications. For all scenarios, the monitoring application parameter was considered.

The results showed that the stack with IoTUS framework attained up to 76.83% less energy consumption than adapted Rime stack in a monitoring application, with a linear topology of 10 nodes. For a 44 nodes tree topology using radio's deep sleep mode, IoTUS got a network average of 42.33% less energy consumption. Consequently, IoTUS reached a network lifetime of 43 days, while adapted Rime got up to 24 days.

Moreover, IoTUS used approximately 4 kbytes more of flash memory than adapted Rime, but reduce up to 31.31% of the RAM usage. Also, network overhead in IoTUS resulted in approximately 81.3% in a 44 nodes tree topology, while adapted Rime attained 87.3%.

Therefore, our results demonstrate that IoTUS is able to achieve energy efficiency, as well as more compact memory footprint when compared to adapted Rime, an OSI-like layered stack adapted from ContikiOS's stack.

8.1 FUTURE DEVELOPMENTS

Directions for future work include:

- Implementing protocol standards like RPL and IPv6 using IoTUS;
- Exploring more layers' procedures, like the Transport layer;
- Simulating scenarios with a larger number of devices;
- Deploying and evaluating IoTUS in real-world testbeds;
- Evaluating IoTUS latency performance;
- Extending the experiments for the different types of applications.

BIBLIOGRAPHY

[1] S. Beeby and N. White, *Energy harvesting for autonomous systems*. Norwood, MA: Artech House, 2010.

[2] W. K. G. Seah, A. E. Zhi, and H. P. Tan, "Wireless Sensor Networks Powered by Ambient Energy Harvesting (WSN-HEAP) - Survey and challenges," in *Proceedings of the 2009 1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace and Electronic Systems Technology, Wireless VITAE 2009*, Aalborg, Denmark, 2009, pp. 1–5.

[3] O. Iova, P. Picco, T. Istomin, and C. Kiraly, "RPL: The Routing Standard for the Internet of Things... Or Is It?" *IEEE Communications Magazine*, vol. 54, no. 12, pp. 16–22, dec 2016.

[4] Webopedia, "The 7 layers of the OSI Model," accessed 2019-02-01. [Online]. Available: https://www.webopedia.com/quick_ref/OSI_Layers.asp

[5] B. Fu, Y. Xiao, H. J. Deng, and H. Zeng, "A survey of cross-layer designs in wireless networks," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 110–126, 2014.

[6] L. D. Mendes and J. J. Rodrigues, "A survey on cross-layer solutions for wireless sensor networks," *Journal of Network and Computer Applications*, vol. 34, no. 2, pp. 523–534, 2011.

[7] S. Chakrabarti, G. Montenegro, R. Droms, and J. Woodyatt, "IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) ESC Dispatch Code Points and Guidelines," February 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8066

[8] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," December 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2460

[9] IEEE, "IEEE Standard for Low-Rate Wireless Networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, apr 2016.

[10] T. Winter, Ed., P. Thubert, Ed., A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RPL: IPv6 Routing

Protocol for Low-Power and Lossy Networks," March 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6550

[11] J. P. Vasseur, N. Agarwal, J. Hui, Z. Shelby, P. Bertrand, and C. Chauvenet, "RPL: The IP routing protocol designed for low power and lossy networks," *In Internet Protocol for Smart Objects (IPSO) Alliance*, no. April, p. 20, 2011.

[12] Z. Shelby, Ed., S. Chakrabarti, E. Nordmark, and C. Bormann, "Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)," November 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6775

[13] I. Akyildiz, M. Vuran, and O. Akan, "A Cross-Layer Protocol for Wireless Sensor Networks," in *Proceedings of the 2006 40th Annual Conference on Information Sciences and Systems*. Princeton, NJ, USA: IEEE, mar 2006, pp. 1102–1107.

[14] S. A. Madani, S. Mahlknecht, and J. Glaser, "A Step towards Standardization of Wireless Sensor Networks: A Layered Protocol Architecture Perspective," in *Proceedings of the 2007 International Conference on Sensor Technologies and Applications (SENSORCOMM 2007)*. Valencia, Spain: IEEE, oct 2007, pp. 82–87.

[15] A. Dunkels, F. Österlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *Proceedings of the 5th international conference on Embedded networked sensor systems - SenSys '07*. Sydney, Australia: ACM Press, 2007, pp. 335–349.

[16] Contiki, "The Open Source OS for the Internet of Things," accessed 2016-07-22. [Online]. Available: http://www.contiki-os.org/

[17] T. A. Vázquez, S. Barrachina-Muñoz, B. Bellalta, and A. Bel, "HARE: Supporting efficient uplink multi-hop communications in self-organizing LPWANs," *Sensors (Switzerland)*, vol. 18, no. 1, 2018.

[18] M. S. Aslam, S. Rea, and D. Pesch, "A Vision for Wireless Sensor Networks: Hybrid Architecture, Model Framework and Service based Systems," in *Proceedings of the 2010 Fifth International Conference on Digital Information Management (ICDIM)*. Thunder Bay, ON, Canada: IEEE, 2010, pp. 353–358.

[19] K. Babber and R. Randhawa, "A Cross-Layer Optimization Framework for Energy Efficiency in Wireless Sensor Networks," *Wireless Sensor Network*, vol. 09, no. 06, pp. 189–203, 2017.

[20] A. Lachenmann, P. J. Marrón, D. Minder, M. Gauger, O. Saukh, and K. Rothermel, "TinyXXL: Language and runtime support for cross-layer interactions," in *Proceedings of the 2006 3rd Annual IEEE Communications Society on Sensor and Adhoc Communications and Networks, Secon 2006*, vol. 1, Reston, VA, USA, 2007, pp. 178–187.

[21] D. Resner, G. Medeiros de Araujo, and A. A. Fröhlich, "Design and implementation of a cross-layer IoT protocol," *Science of Computer Programming*, vol. 165, pp. 24–37, 2018.

[22] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with COOJA," in *Proceedings of the Conference on Local Computer Networks, LCN*. Tampa, FL, USA: IEEE, 2006, pp. 641–648.

[23] A. Parasuram, D. Culler, and R. Katz, "An Analysis of the RPL Routing Standard for Low Power and Lossy Networks," *Technical Report No. UCB/EECS-2016-106*, p. 98, 2016.

[24] I. Solis and K. Obraczka, "FLIP: A Flexible Interconnection Protocol for heterogeneous internetworking," *Mobile Networks and Applications*, vol. 9, no. 4, pp. 347–361, 2004.

[25] J. Postel, "Internet Protocol," September 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc791

[26] P. Gonizzi, P. Medagliani, G. Ferrari, and J. Leguay, "RAWMAC: A routing aware wave-based MAC protocol for WSNs," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2014 IEEE 10th International Conference on*. Larnaca, Cyprus: IEEE, 2014, pp. 205–212.

[27] A. Dunkels, "The contikimac radio duty cycling protocol," *SICS Technical Report T2011:13*, 2011.

[28] G. Lu, B. Krishnamachari, and C. S. Raghavendra, "An adaptive energy-efficient and low-latency MAC for data gathering in wireless sensor networks," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, vol. 00, no. C, Santa Fe, NM, USA, 2004, pp. 224–231.

[29] J. Tan, A. Liu, M. Zhao, H. Shen, and M. Ma, "Cross-layer design for reducing delay and maximizing lifetime in industrial wireless sensor networks," *Eurasip Journal on Wireless Communications and Networking*, vol. 2018, no. 1, 2018.

[30] IEEE, "IEEE 802.15.4-2006 - IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs)," *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pp. 1–320, Sept 2006.

[31] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized protocol stack for the internet of (important) things," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 3, pp. 1389–1406, 2013.

[32] T. Watteyne, Ed., M. Palattella, and L. Grieco, "Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement," May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7554

[33] Q. Wang, Ed., X. Vilajosana, and T. Watteyne, "6TiSCH Operation Sublayer (6top) Protocol (6P)," November 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8480

[34] X. Vilajosana, T. Watteyne, M. Vucinic, T. Chang, and K. S. J. Pister, "6TiSCH: Industrial Performance for IPv6 Internet-of-Things Networks," *Proceedings of the IEEE*, vol. 107, no. 6, pp. 1153–1165, jun 2019.

[35] H. Khattak, Z. Ameer, U. Din, and M. Khan, "Cross-layer design and optimization techniques in wireless multimedia sensor networks for smart cities," *Computer Science and Information Systems*, vol. 16, no. 1, pp. 1–17, 2019.

[36] Z. Niroumand and H. S. Aghdasi, "A geographic cross-layer routing adapted for disaster relief operations in wireless sensor networks," *Computers and Electrical Engineering*, vol. 64, pp. 395–406, 2017.

[37] W. Zhang, X. Wei, G. Han, and X. Tan, "An Energy-Efficient Ring Cross-Layer Optimization Algorithm for Wireless Sensor Networks," *IEEE Access*, vol. 6, pp. 16 588–16 598, 2018.

[38] X. Li, S. L. Fang, and Y. C. Zhang, "The study on clustering algorithm of the underwater acoustic sensor networks," in *Proceedings 14th International Conference on Mechatronics and Machine Vision in Practice, M2VIP2007.* Xiamen, China: IEEE, 2007, pp. 78–81.

[39] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and Others, "TinyOS: An operating system for sensor networks,"

in *Weber W., Rabaey J.M., Aarts E. (eds) Ambient intelligence*. Berlin, Heidelberg: Springer, 2005, pp. 115–148.

[40] P. J. Marrón, D. Minder, A. Lachenmann, and K. Rothermel, "TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks (TinyCubus: Ein Adaptives Cross-Layer Framework für Sensornetze)," *it - Information Technology*, vol. 47, no. 2, jan 2005.

[41] D. Minder, M. Handte, and P. J. Marrón, "TinyAdapt: An adaptation framework for sensor networks," in *Proceedings of the INSS 2010 - 7th International Conference on Networked Sensing Systems*, Kassel, Germany, 2010, pp. 253–256.

[42] K. Roussel and Y.-q. Song, "A critical analysis of Contiki's network stack for integrating new MAC protocols," Ph.D. dissertation, INRIA Nancy, 2015.

[43] J. Eriksson, F. Österlind, N. Finne, A. Dunkels, N. Tsiftes, and T. Voigt, "Accurate Network-Scale Power Profiling for Sensor Network Simulators," in *Roedig U., Sreenan C.J. (eds) Wireless Sensor Networks. EWSN 2009. Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer, 2009, vol. 5432 LNCS, pp. 312–326.

[44] Moteiv, "Tmote sky - Low Power Wireless Sensor Module," accessed 2018-06-11. [Online]. Available: http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf

[45] E. Baccelli, C. Gundogan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wahlisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[46] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," *arXiv:1801.02833 [cs.NI]*, 2018.

[47] Riot, "The friendly Operating System for the Internet of Things," accessed 2019-01-15. [Online]. Available: https://riot-os.org/

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

[49] www.xbow.com, “Wireless motes Rene,” accessed 2014-05-01. [Online]. Available: www.xbow.com

[50] K. Han, J. Luo, Y. Liu, and A. Vasilakos, “Algorithm design for data communications in duty-cycled wireless sensor networks: A survey,” *IEEE Communications Magazine*, vol. 51, no. 7, pp. 107–113, jul 2013.

[51] R. M. de Moraes and H. Sadjadpour, “Wireless Network Protocols,” in *Jerry D. Gibson. (Org.). Mobile Communications Handbook*, 3rd ed. CRC press, 2012, pp. 603–614.

[52] W. Ye, J. Heidemann, and D. Estrin, “Medium access control with coordinated adaptive sleeping for wireless sensor networks,” *IEEE/ACM Transactions on Networking*, vol. 12, no. 3, pp. 493–506, 2004.

[53] J. Polastre, J. Hill, and D. Culler, “Versatile Low Power Media Access for Wireless Sensor Networks,” in *Proceedings of the 2nd ACM Conferences on Embedded Networked Sensor Systems*, Baltimore, MD, USA, 2004, pp. 95–107.

[54] B. University of California, “Mica2 Wireless mote,” accessed 2016-07-07. [Online]. Available: http://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf

[55] A. El-Hoiydi and J.-d. Decotignie, “WiseMAC : An Ultra Low Power MAC Protocol for Multi-hop Wireless Sensor Networks,” *Power*, vol. 3121, no. 5005, pp. 18–31, 2004.

[56] M. Buettner, G. V. Yee, E. Anderson, and R. Han, “X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks,” in *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys 2006)*, Boulder, Colorado, USA, 2006, pp. 307–320.

[57] L. Constante, J. Lau, R. Moraes, G. Araujo, C. Montez, and E. Leao, “Enhanced association mechanism for IEEE 802.15.4 networks,” in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Limassol, Cyprus: IEEE, sep 2017, pp. 1–8.

[58] J. Hui, J. Vasseur, D. Culler, and V. Manral, “An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL),” March 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6554

[59] R. Bonica, D. Gan, D. Tappan, and C. Pignataro, “Extended ICMP to Support Multi-Part Messages,” April 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4884

[60] Wikipedia, “Advanced Packaging Tool,” accessed 2019-02-02. [Online]. Available: https://pt.wikipedia.org/wiki/Advanced_Packaging_Tool

[61] FlockLab, “Flocklab,” accessed 2016-10-21. [Online]. Available: https://www.flocklab.ethz.ch/wiki/

[62] F. Osterlind, E. Pramsten, D. Roberthson, J. Eriksson, N. Finne, and T. Voigt, “Integrating building automation systems and wireless sensor networks,” in *Proceedings of the 2007 IEEE Conference on Emerging Technologies & Factory Automation (EFTA 2007)*, Patras, Greece, 2007, pp. 1376–1379.

[63] B. R. Haverkort, *Performance of Computer Communication Systems: A Model-Based Approach*, 1st ed. Chichester, England: John Wiley & Sons, Inc., 1998.

[64] Wikipedia, “AA battery,” accessed 2019-02-02. [Online]. Available: https://en.wikipedia.org/wiki/AA_battery

APPENDIX

A RESUMO ESTENDIDO EM LÍNGUA PORTUGUESA

Título: Uma Nova Arquitetura de Protocolos para IoT: Eficiência Através do Compartilhamento de Dados e de Funcionalidades entre Camadas
Autor: Vinícius Galvão Guimarães
Orientador: Renato Mariz de Moraes
Coorientador: Adolfo Bauchspiess
Coorientadora: Katia Obraczka
Programa de Pós-Graduação em Engenharia de Sistemas Eletrônicos e de Automação
Brasília, 28 de junho de 2019

Palavras-chave: Comunicação Sem Fio, Internet das Coisas, Eficiência Energética, Pilha de Protocolos.

Contextualização

A pilha TCP/IP é um padrão para redes e, portanto, está presente em muitos sistemas de comunicação. No entanto, para aplicações com *Internet of Things* (IoT), como os da Figura A.1, muitos trabalhos propõem designs que infringem a restrição de acesso entre camadas não adjacentes ou, até mesmo, novas arquiteturas de protocolos para melhorar a eficiência energética.

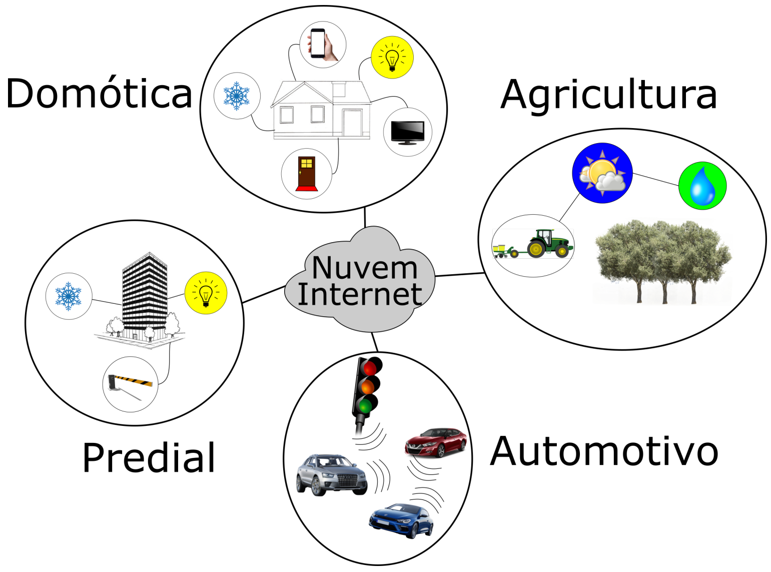


Figura A.1 – Dispositivos da Internet das Coisas com aplicações ilustrativas.

Motivado pela necessidade de acomodar dispositivos IoT com recursos limitados de energia, processamento, armazenamento e comunicação, este trabalho apresenta o *IoT Unified Services*, ou IoTUS, uma nova arquitetura de protocolos de rede voltada para eficiência de energia e compacto uso de memória.

IoTUS - IoT Unified Services

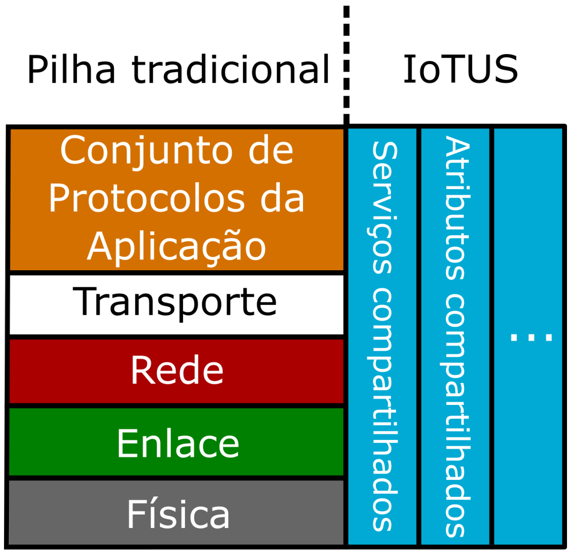


Figura A.2 – Camada de serviços extensível do IoTUS adicional à uma pilha tradicional.

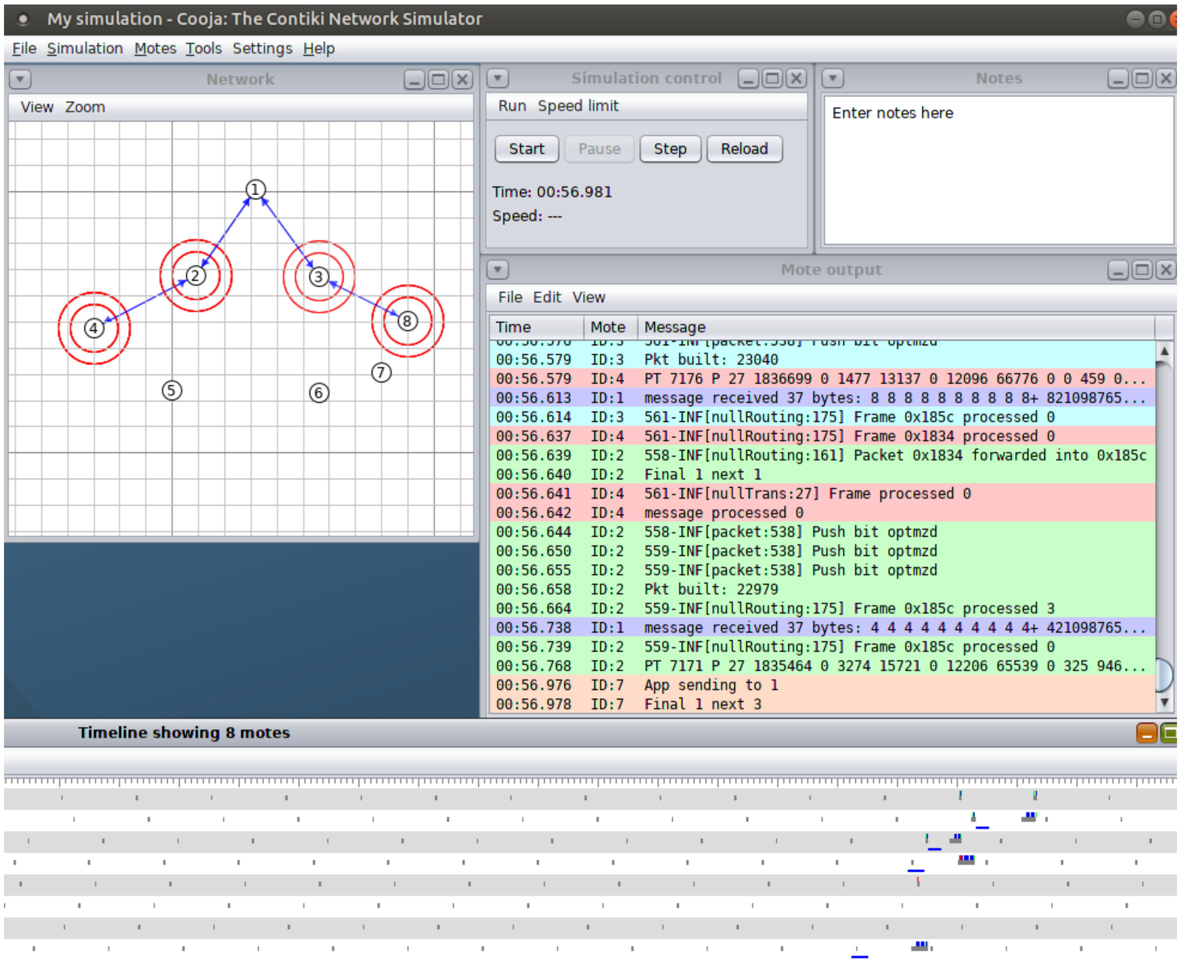
O IoTUS usa uma camada de serviços extensível que facilita o compartilhamento entre camadas (Figura A.2). Promove também o compartilhamento das informações de controle de rede (por exemplo, número de transmissões, recepções, colisões na camada de enlace de dados) e funcionalidades (por exemplo, descoberta de vizinhos, agregação de pacotes) para diferentes camadas da pilha de protocolos.

Através do eficiente compartilhamento em camadas cruzadas, o principal objetivo do IoTUS é alcançar a eficiência energética, bem como um compacto uso de memória, ambos importantes para acomodar dispositivos IoT com recursos limitados. Semelhante às outras propostas [14, 15, 20], IoTUS fornece módulos para padronizar o modo como a informação é acomodada nos pacotes. No entanto, diferentemente de outras propostas [14, 15, 20, 21, 24], o IoTUS não apenas fornece benefícios de designs entre camadas, como também permite que os protocolos na pilha tradicional estejam cientes dos procedimentos uns dos outros para otimizar suas tarefas através de um processo de inscrição. Por exemplo, quando protocolos compartilham seu comportamento com os demais da pilha (pacotes periódicos e procedimentos esperados), outros protocolos estarão cientes de possíveis agregações e poderão otimizar seus parâmetros para operar de acordo.

Além disso, como ilustrado na Figura A.2, o IoTUS pode ser usado por arquiteturas de protocolos já existentes, sem ter que modificar a proposta de seus protocolos já desenvolvidos.

Metodologia

Avaliou-se o IoTUS usando a plataforma de simulação/emulação Cooja-ContikiOS. O simulador Cooja (Figura A.3) foi utilizado pelos seguintes motivos: primeiro, ele fornece uma plataforma experimental especificamente projetada para redes sem fio, adequada para redes de sensores sem fio com restrição de capacidade/processamento. A vantagem de usar a simulação para um novo projeto é executar um ambiente controlado e assim obter experimentos reprodutíveis.



Time: 00:56.981
Speed: ---

```
Time       Mote   Message
00:56.579  ID:3   Pkt built: 23040
00:56.579  ID:4   PT 7176 P 27 1836699 0 1477 13137 0 12096 66776 0 0 459 0...
00:56.613  ID:1   message received 37 bytes: 8 8 8 8 8 8 8 8 8 8+ 821098765...
00:56.614  ID:3   561-INF[nullRouting:175] Frame 0x185c processed 0
00:56.637  ID:4   561-INF[nullRouting:175] Frame 0x1834 processed 0
00:56.639  ID:2   558-INF[nullRouting:161] Packet 0x1834 forwarded into 0x185c
00:56.640  ID:2   Final 1 next 1
00:56.641  ID:4   561-INF[nullTrans:27] Frame processed 0
00:56.642  ID:4   message processed 0
00:56.644  ID:2   558-INF[packet:538] Push bit optmzd
00:56.650  ID:2   559-INF[packet:538] Push bit optmzd
00:56.655  ID:2   559-INF[packet:538] Push bit optmzd
00:56.658  ID:2   Pkt built: 22979
00:56.664  ID:2   559-INF[nullRouting:175] Frame 0x185c processed 3
00:56.738  ID:1   message received 37 bytes: 4 4 4 4 4 4 4 4 4 4+ 421098765...
00:56.739  ID:2   559-INF[nullRouting:175] Frame 0x185c processed 0
00:56.768  ID:2   PT 7171 P 27 1835464 0 3274 15721 0 12206 65539 0 325 946...
00:56.976  ID:7   App sending to 1
00:56.978  ID:7   Final 1 next 3
```

Figura A.3 – Tela ilustrativa do simulador Cooja.

A pilha de protocolos do ContikiOS já inclui operações cross-layer (*Packetbuf*) e uma camada RDC fora do padrão OSI, que nem sempre é facilmente portável para outros protocolos MAC. No entanto, novas pilhas IoT ainda mantêm o design tradicional em camadas, como no Riot. O uso da pilha tradicional de protocolos em camadas não apenas facilita a análise, mas também facilita a reprodução em outros sistemas operacionais. Portanto, o foco desta proposta foi a comparação do IoTUS com uma pilha tradicional e suas operações em camadas.

Neste trabalho, adaptou-se a pilha de protocolos do ContikiOS para uma pilha tradicional composta apenas de camada física (e sua subcamada *framer*), camada MAC, camada de rede e camada de aplicação. Os protocolos da camada de transporte, embora necessários em sistemas IoT, ainda não foram implementados. O *Packetbuf* do ContikiOS foi mantido apenas na pilha tradicional.

Para validar os resultados obtidos usando a ferramenta de simulação, foram desenvolvidos modelos teóricos, que descrevem o consumo de energia de um dispositivo TMote Sky. O modelo teórico considera os parâmetros e tempos resultantes da versão implementada de seus respectivos protocolos, mas todos os resultados teóricos são calculados independentemente de qualquer simulação.

No ambiente de simulação, as medições de consumo de energia foram fornecidas por duas ferramentas Cooja-ContikiOS diferentes: *PowerTrace* e *PowerTracker*. *PowerTrace* é uma ferramenta ContikiOS, que reporta periodicamente o consumo de energia através de sua porta serial. *PowerTrace* relata o tempo gasto em cada estado (transmissão, recepção, inatividade, e ativo ou inativo para o CPU). *PowerTracker* vem com o simulador Cooja e fornece medições de consumo de energia do rádio. No entanto, *PowerTrace* é executado no código do MCU e não calcula as curtas transições de estado do rádio (por exemplo, o estado Rx para o CCA antes da transmissão). Por outro lado, *PowerTracker* pode detectar transições de rádio e, portanto, fornece medições de consumo de energia mais precisas do que o *PowerTrace*. Consequentemente, usamos *PowerTracker* para medir a energia consumida pelo rádio, enquanto o consumo de CPU ainda é extraído da ferramenta *PowerTrace*.

Para guiar as simulações deste trabalho, considerou-se uma aplicação de monitoramento ambiental geral como na Figura A.4, em que os valores de medição do ambiente tais como temperatura e umidade são reportados periodicamente para um nó sumidouro, geralmente conectado à internet ou ao sistema de controle. A Figura A.5 mostra uma topologia de árvore de 44 nós usada nas simulações deste trabalho. A árvore está direcionada ao coletor de dados (nó 1); Todos os nós intermediários e folhas são nós sensores. Observe que os nós intermediários da árvore agem tanto como geradores de tráfego quanto como roteadores.

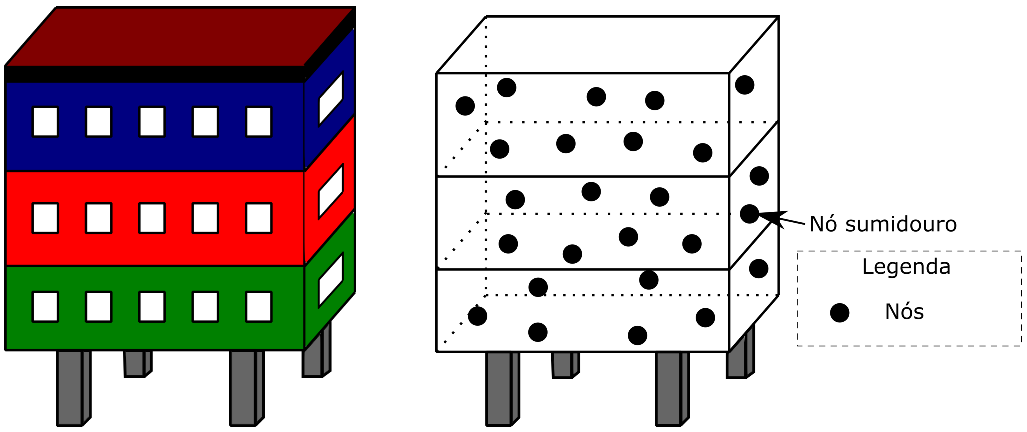


Figura A.4 – Exemplo de uma aplicação geral de monitoramento ambiental com nós.

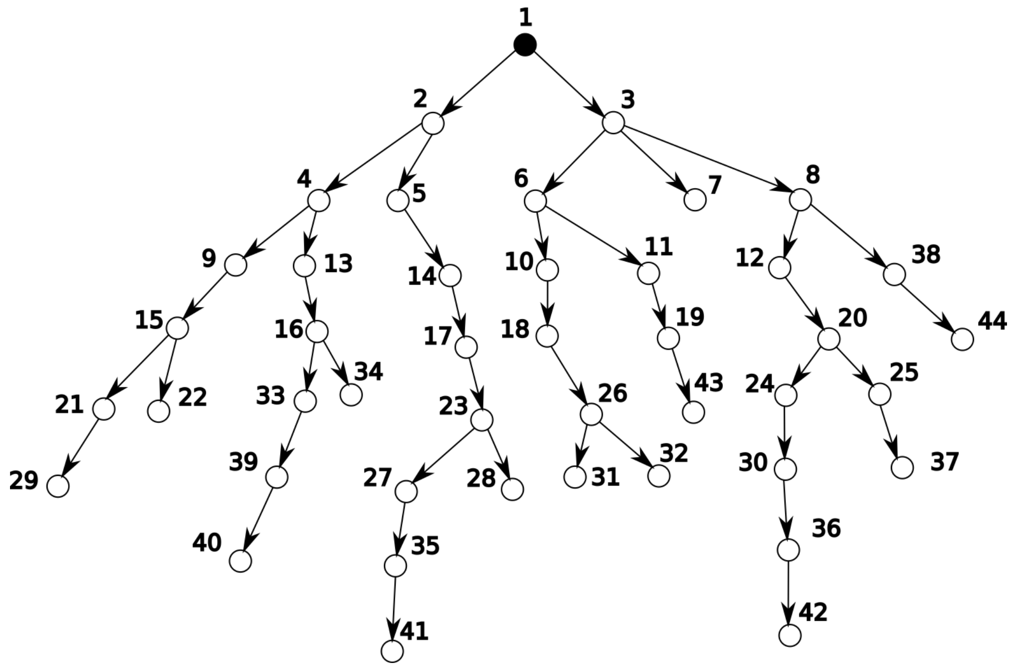


Figura A.5 – Topologia de árvore considerada para monitoramento ambiental.

Resultados

Os resultados teóricos e de simulação mostraram-se bastante parecidos e coerentes. Assim, nos resultados simulados, o IoTUS consumiu $76,83\%$ menos energia comparado à pilha de comunicação adaptada do ContikiOS em uma aplicação de monitoramento, com uma topologia linear de 10 nós (Figura A.6). Para uma topologia de 44 nós em árvore, o IoTUS obteve uma média de $42,33\%$ menos consumo de energia (Figura A.7).

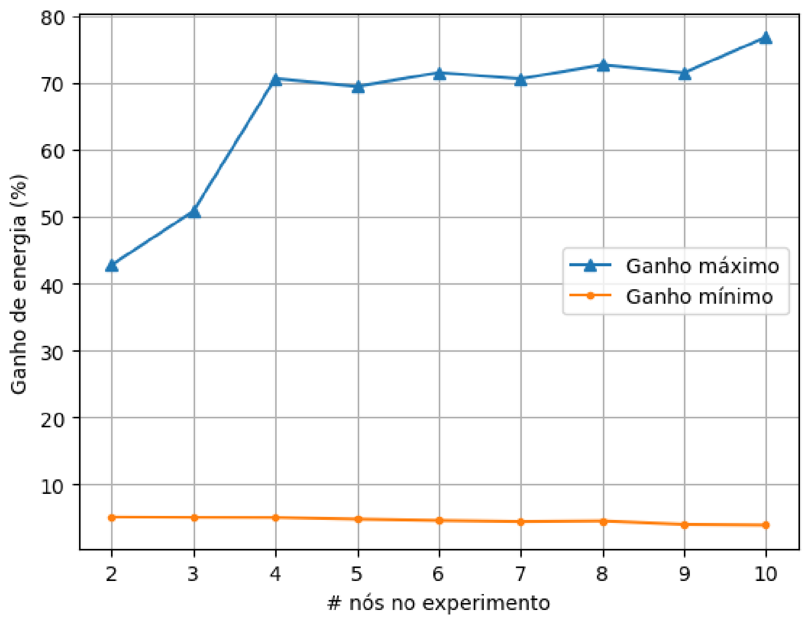


Figura A.6 – Máximo e mínimo ganho de energia de rádio por nós na rede com topologia linear.

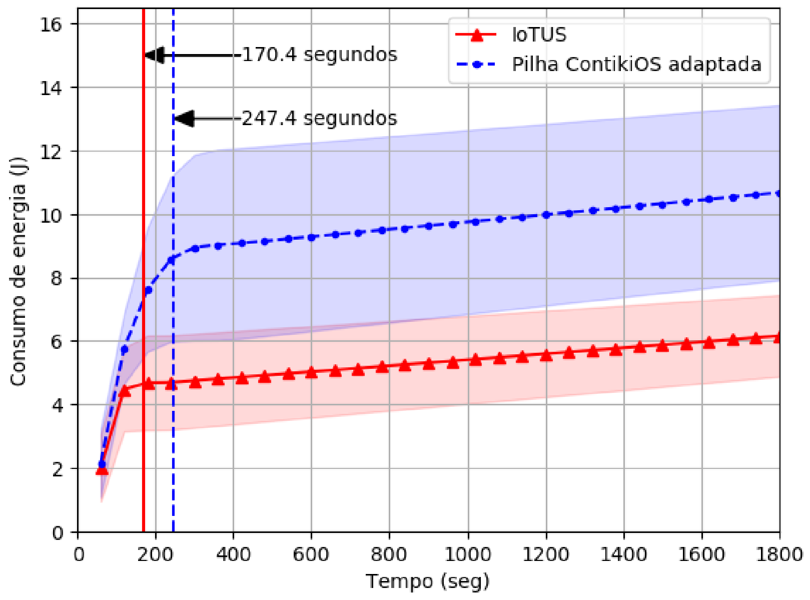


Figura A.7 – Máximo e mínimo ganho de energia de rádio por nós na rede com topologia em árvore.

Consequentemente, o IoTUS atingiu uma vida útil de 43 dias, enquanto a pilha ContikiOS adaptada chegou a 24 dias. Quanto à memória, o IoTUS usou aproximadamente 4 kbytes a mais de memória flash do que a pilha adaptada do ContikiOS, mas reduziu em até $31,31\%$ o uso de RAM (Figura A.8).

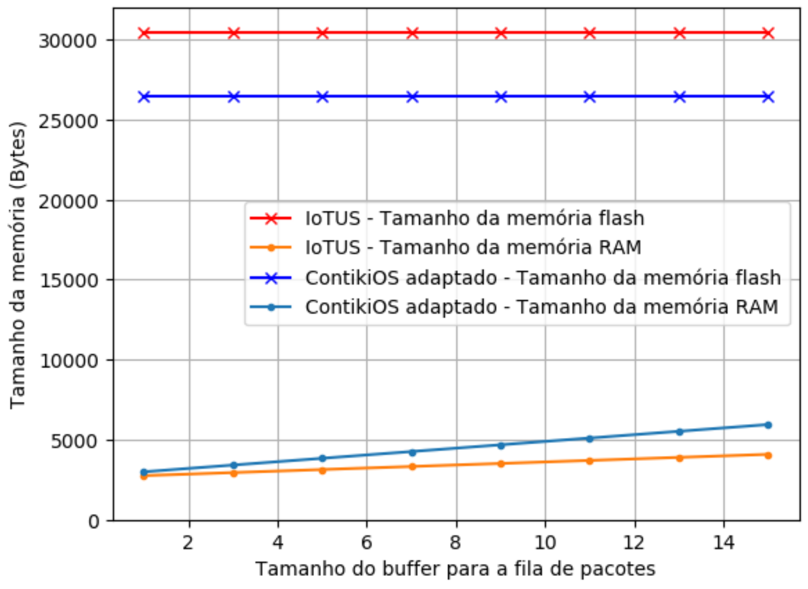


Figura A.8 – Uso de memória ao aumentar a capacidade do buffer de pacote.

Além disso, como apresentado na Figura A.9, o excesso de cabeçalhos na rede IoTUS foi de aproximadamente $81,3\%$ com uma topologia em árvore de 44 nós, enquanto o ContikiOS adaptado resultou em $87,3\%$.

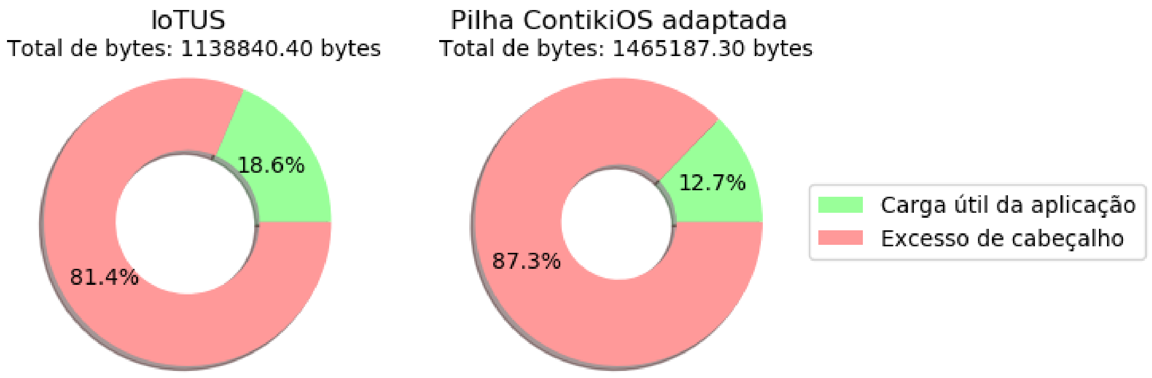


Figura A.9 – Comparação do excesso de cabeçalhos para 44 nós em uma topologia de árvore.

Conclusão

Os resultados teóricos e de simulação mostraram melhor desempenho do IoTUS, melhor eficiência energética, maior vida útil da rede e um compacto uso de memória, quando comparado às atuais arquiteturas de protocolos de IoT.

B C STRUCTS AND FUNCTIONS USED IN IOTUS IMPLEMENTATION

```c
/*
 * Main packet struct to hold most of its information
*/
typedef struct packet_piece {
  struct packet_piece *next;
  struct mmem data;
  struct ctimer transmit_timer;
  iotus_node_t *finalDestinationNode;
  iotus_node_t *nextDestinationNode;
  iotus_node_t *prevSourceNode;
  void (*confirm_cb)(struct packet_piece *packet, iotus_netstack_return
    returnAns);
  LIST_STRUCT(additionalInfoList);
  timestamp_t timeout;
  iotus_layer_priority priority;
  uint16_t firstHeaderBitSize;
  uint16_t lastHeaderSize;
  uint8_t params;
  uint8_t pktID;
  uint8_t type;
  uint8_t collisions;
  uint8_t transmissions;
} iotus_packet_t;

/*
 * List of additional options implemented
*/
enum packet_additionalInfoList_types {
  IOTUS_PACKET_INFO_TYPE_RESERVED = 0,

  IOTUS_PACKET_INFO_TYPE_P2P_ADDRESSES,
  IOTUS_PACKET_INFO_TYPE_SOURCE_ADDRESS,
  IOTUS_PACKET_INFO_TYPE_DEST_ADDRESS,
  IOTUS_PACKET_INFO_TYPE_SEQUENCE_NUMBER,
  IOTUS_PACKET_INFO_TYPE_RETX_ATTEMPTS_DID,

  //Values generally sensed and available from the radio after receiving
```

```
    a packet
  IOTUS_PACKET_INFO_TYPE_RADIO_RCV_BLCK,
  //Value generally expected by the radio to transmit a packet
  IOTUS_PACKET_INFO_TYPE_RADIO_TX_BLCK,

  IOTUS_PACKET_INFO_TYPE___N,
};

/*
 * Structs to hold additional options implemented
*/

/**
 * This struct is preceded by the type
    IOTUS_PACKET_INFO_TYPE_RADIO_RCV_BLCK
 */
typedef struct __attribute__ ((__packed__)) packet_rcv_additional_info {
  uint16_t networkID;
  int8_t linkQuality;
  int8_t rssi; //got from CCA
} packet_rcv_block_output_t;

/**
 * This struct is preceded by the type
    IOTUS_PACKET_INFO_TYPE_RADIO_TX_BLCK
 */
typedef struct __attribute__ ((__packed__)) packet_tx_additional_info {
  uint8_t txPower;
  uint8_t channel;
} packet_tx_block_input_t;

/*
 * This struct is preceded by the type
    IOTUS_PACKET_INFO_TYPE_P2P_ADDRESSES
 * Used to inform end to end communications
 */
typedef struct __attribute__ ((__packed__)) packet_e2e_nodes_addit_info {
  uint8_t *sourceNode;
  uint8_t *finaLNode;
} packet_addresses_block_t;

/*
 * List of available functions
*/
```

```
uint8_t
  packet_get_parameter(iotus_packet_t *packet_piece, uint8_t param);
void
  packet_set_parameter(iotus_packet_t *packet_piece, uint8_t param);
void
  packet_clear_parameter(iotus_packet_t *packet_piece, uint8_t param);
iotus_node_t *
  packet_get_final_destination(iotus_packet_t *packet_piece);
iotus_node_t *
  packet_get_prevSource_node(iotus_packet_t *packetPiece);
iotus_node_t *
  packet_get_next_destination(iotus_packet_t *packetPiece);
Status
  packet_set_next_destination(iotus_packet_t *packetPiece, iotus_node_t *
    node);
Status
  packet_set_final_destination(iotus_packet_t *packetPiece, iotus_node_t
    *node);
Boolean
  packet_holds_broadcast(iotus_packet_t *packetiece);
Status
  packet_set_tx_channel(iotus_packet_t *packetPiece, uint8_t channel);
int16_t
  packet_get_tx_channel(iotus_packet_t *packetPiece);
Status
  packet_set_tx_power(iotus_packet_t *packetPiece, int8_t power);
int8_t
  packet_get_tx_power(iotus_packet_t *packetPiece);
Status
  packet_set_sequence_number(iotus_packet_t *packetPiece, uint8_t
    sequence);
int16_t
  packet_get_sequence_number(iotus_packet_t *packetPiece);
Status
  packet_set_rx_netID(iotus_packet_t *packetPiece, uint16_t netID);
Status
  packet_set_rx_linkQuality_RSSI(iotus_packet_t *packetPiece, uint8_t
    linkQuality, uint8_t rssi);
packet_rcv_block_output_t *
  packet_get_rx_block(iotus_packet_t *packetPiece);
iotus_packet_t *
  packet_create_msg(uint16_t payloadSize, const uint8_t* payload,
     iotus_layer_priority priority, uint16_t timeout, Boolean
    AllowOptimization,
    iotus_node_t *finalDestination);
Boolean
  packet_destroy(iotus_packet_t *msgPiece);
unsigned int
  packet_get_size(iotus_packet_t *packet_piece);
uint16_t
  packet_push_bit_header(uint16_t bitSequenceSize, const uint8_t *bitSeq,
    iotus_packet_t *packet_piece);
uint16_t
  packet_append_last_header(uint16_t byteSequenceSize, const uint8_t *
   headerToAppend,
  iotus_packet_t *packet_piece);
uint8_t
  packet_read_byte(uint16_t bytePos, iotus_packet_t *packet_piece);
uint8_t
  packet_read_byte_backward(uint16_t bytePos, iotus_packet_t *
   packet_piece);
void
  packet_set_byte_backward(uint8_t byte, uint16_t bytePos, iotus_packet_t
    *packetPiece);
void
  packet_extract_data_bytes(uint8_t *buff, uint16_t bytePos, uint16_t
   size, iotus_packet_t *packetPiece);
Status
  packet_unwrap_appended_byte(iotus_packet_t *packetPiece, uint8_t *buf,
   uint16_t num);
uint8_t
  packet_unwrap_pushed_byte(iotus_packet_t *packetPiece);
uint32_t
  packet_unwrap_pushed_bit(iotus_packet_t *packetPiece, int8_t num);
uint16_t
  packet_get_payload_size(iotus_packet_t *packetPiece);
uint8_t *
  packet_get_payload_data(iotus_packet_t *packetPiece);
void
  packet_parse(iotus_packet_t *packetPiece);
Boolean
  packet_has_space(iotus_packet_t *packetPiece, uint16_t space);
void
  packet_set_confirmation_cb(iotus_packet_t *packet, packet_sent_cb
   func_cb);
void
  packet_confirm_transmission(iotus_packet_t *packet,
   iotus_netstack_return status);
```

```c
uint8_t
  packet_queue_size_by_node(iotus_node_t *node);
iotus_packet_t *
  packet_get_queue_by_node(iotus_node_t *node, iotus_packet_t *from);
/* This function provides the core access to basic operations into this
    service */
void
  iotus_signal_handler_packet(iotus_service_signal signal, void *data);
```

Listing B.1 – Main structs in language C for IoTUS's *Packet Manager*.

```c
/*
 * Main node struct
*/
typedef struct nodes {
  struct nodes *next;
  LIST_STRUCT(additionalInfoList);
  uint8_t params;
} iotus_node_t;

/*
 * List of additional options implemented
*/
enum nodes_additional_Info_types {
  IOTUS_ADDRESSES_TYPE_INCLUDE(NODES_ADD_INFO),
  IOTUS_NODES_ADD_INFO_TYPE_TOPOL_TREE_RANK,
  IOTUS_NODES_ADD_INFO_TYPE_LAST_SEQ_NUM,
  IOTUS_NODES_ADD_INFO_TYPE_NETID,
  IOTUS_NODES_ADD_INFO_TYPE_WAKEUP_PHASE,
  IOTUS_NODES_ADD_INFO_TYPE_NEXT_ADDR_TO_NODE,
  IOTUS_NODES_ADD_INFO_TYPE___N
};

/*
 * List of available functions
*/
uint8_t
  node_get_parameter(iotus_node_t *node, uint8_t param);
void
node_set_parameter(iotus_node_t *node, uint8_t param);
uint8_t *
  nodes_get_address(iotus_address_type addressType, iotus_node_t *node);
iotus_node_t *
```

```
  nodes_get_node_by_address(iotus_address_type addressType, uint8_t *
    address);
Boolean
  nodes_set_address(iotus_node_t *node, iotus_address_type addressType,
    uint8_t *address);
iotus_node_t *
  nodes_update_by_address(iotus_address_type addressType, uint8_t *
    address);
void
  nodes_destroy(iotus_node_t *node);
void
  iotus_signal_handler_nodes(iotus_service_signal signal, void *data);
```

Listing B.2 – Main structs in language C for IoTUS's *Node Manager*.

```
/*
 * Main node struct
*/
typedef struct piggyback_piece {
  struct piggyback_piece *next;
  struct mmem data;
  iotus_layer_priority priority;
  iotus_node_t *finalDestinationNode;
  timestamp_t timeout;
  uint8_t params;
  uint8_t pktID;
  uint8_t extendedSize;
} iotus_piggyback_t;


/*
 * List of additional options implemented
*/
enum nodes_additional_Info_types {
  IOTUS_ADDRESSES_TYPE_INCLUDE(NODES_ADD_INFO),
  IOTUS_NODES_ADD_INFO_TYPE_TOPOL_TREE_RANK,
  IOTUS_NODES_ADD_INFO_TYPE_LAST_SEQ_NUM,
  IOTUS_NODES_ADD_INFO_TYPE_NETID,
  IOTUS_NODES_ADD_INFO_TYPE_WAKEUP_PHASE,
  IOTUS_NODES_ADD_INFO_TYPE_NEXT_ADDR_TO_NODE,
  IOTUS_NODES_ADD_INFO_TYPE___N
};

```

```c
/*
 * List of available functions
*/
Boolean
  piggyback_destroy(iotus_piggyback_t *piece);
void
  piggyback_confirm_sent(iotus_packet_t *packet, uint8_t status);
iotus_piggyback_t *
  piggyback_create_piece(uint16_t headerSize, const uint8_t* headerData,
    iotus_layer_priority targetLayer, iotus_node_t *destinationNode,
   int16_t timeout);
void
  piggyback_unwrap_payload(iotus_packet_t *packet);
uint16_t
    piggyback_apply(iotus_packet_t *packet_piece, uint16_t availableSpace
   );
void
  piggyback_subscribe(iotus_layer_priority layer, piggy_cb_func *cbFunc);
void
  iotus_signal_handler_piggyback(iotus_service_signal signal, void *data)
   ;
```

Listing B.3 – Main structs in language C for IoTUS's *Piggyback Service*.

C IMPLEMENTATION SOURCE AND FOLDERS

The implemented framework IoTUS is open source and is available online at `https://github.com/Vinggui/contiki-IoTUS`

The framework code is inside the "iotus-arch" folder. Services, examples of protocols (including the ContikiMAC802 and RPL-Like used in this work), and the services/tools of IoTUS framework are all available in this online open and free repository.

Instructions are present in the main page of its repository, where user can understand how to install and executes its functions.

D PYTHON SCRIPT: THEORETICAL ENERGY CONSUMPTION OF A SINGLE NODE

Listing D.1 – Python script to calculate the theoretical energy consumption of a single node.

```python
import matplotlib.pyplot as plot
import matplotlib as mpl
import metrics_funcs_multiple_exp as metric
import numpy, math
import os

# Parameters
volt = 3.0
curr_tx = 0.0174
curr_rx = 0.0188
curr_idle_tx = 0
curr_idle_rx = curr_rx
curr_radio_sleep = 0.000426
curr_cpu = 0.0018
curr_lpm = 0.0000054

# Topology
num_nodes = 1

RADIO_ON_DUR = 2.2
RADIO_TX_DUR = 1.534

RADIO_REPORT_PERIOD = 30000


total_time = 30*60*1000
num_pkt_sent = total_time/RADIO_REPORT_PERIOD

radio_tx_cons_period = volt*curr_tx*(RADIO_TX_DUR)*num_pkt_sent
radio_idle_cons_period = volt*curr_idle_rx*(RADIO_ON_DUR-RADIO_TX_DUR)*
                                     num_pkt_sent
radio_sleep_cons_period = volt*curr_radio_sleep*(RADIO_REPORT_PERIOD-
                                     RADIO_ON_DUR)*num_pkt_sent
```

```python

# Results
print("Consumptions")
print("Tx: "+str(radio_tx_cons_period))
print("rx: "+str(radio_idle_cons_period))
print("sleep: "+str(radio_sleep_cons_period))
```

E PYTHON SCRIPT: THEORETICAL ENERGY CONSUMPTION OF SIX NODES IN A LINEAR TOPOLOGY

Listing E.1 – Python script to calculate the theoretical energy consumption of N nodes in a linear topology.

```python
import matplotlib.pyplot as plot
import matplotlib as mpl
import metrics_funcs_multiple_exp as metric
import numpy, math
import os


# Parameters
volt = 3.0
curr_tx = 0.0174
curr_rx = 0.0188
curr_idle_tx = 0
curr_idle_rx = curr_rx
curr_cpu = 0.0018
curr_lpm = 0.0000054

# Select the sleep mode consumption to use
#curr_radio_sleep = 0.00000002
curr_radio_sleep = 0.000426


# Linear topology, select number of nodes
#num_nodes = 2
num_nodes = 6

#-------------------------
# The node that is supposed to be generating messages,
# if 0 then all nodes will be considered generating (App and KA)
node_gen = 0
#-------------------------

# Wakeup in ms
```

```python
wake_up_period = 125

# Number of cca per wakeup
CCA_per_wakeup = 2

# Simulation duration in min
simulation_duration = 30

# Period of application msg in sec
application_period = 30

# Period of keep alive msg in sec, 0 means no service...
if node_gen == 0:
    keepAlive_period = 30
else:
    keepAlive_period = 0


def get_transmission_time(bytes):
    #Add 6 bytes of PHY layer preamble
    return 8*(bytes+6)/250

def lcm(a,b): return abs(a*b)/math.gcd(a,b) if a and b else 0

class consumption_states:
    def __init__(self, name):
        self.name = name
        self.tx = 0
        self.rx = 0
        self.idle_rx = 0
        self.sleep = 0
        self.duration = 0

    def __add__(self, other):
        new_result = consumption_states("("+self.name + " + "+other.name+
                                                                  ")")
        new_result.tx = self.tx + other.tx
        new_result.rx = self.rx + other.rx
        new_result.idle_rx = self.idle_rx + other.idle_rx
        new_result.sleep = self.sleep + other.sleep
        new_result.duration = self.duration + other.duration
        return new_result

    def __mul__(self, other):
        final_value = other
        new_result = consumption_states("("+self.name + " * "+str(
                                                     final_value)+")")
        new_result.tx = self.tx*final_value
        new_result.rx = self.rx*final_value
        new_result.idle_rx = self.idle_rx*final_value
        new_result.sleep = self.sleep*final_value
        new_result.duration = self.duration*final_value
        return new_result

def dump(obj):
    print(" ")
    print("Consumption of "+obj.name)
    print("%s = \t\t%s" % ("tx", obj.tx))
    print("%s = \t\t%s" % ("rx", obj.rx))
    print("%s = \t%s" % ("idle rx", obj.idle_rx))
    print("%s = \t%s" % ("sleep", obj.sleep))
    print("%s = \t%s" % ("duration", obj.duration))
    print("")

def dump_topology(topology):
    node_list = []
    for rank_list in topology:
        for device in rank_list:
            node_list.append(device)

    node_list.sort(key=lambda x: x.id)

    text_to_print = ""
    text_to_print += "Energy consumption of nodes\n"
    text_to_print += "Node ##      Tx         Rx        Idle      Sleep\n"
    for i in range(0, len(node_list)):
        text_to_print += "Node {:02}: {: >8.4f}  {: >8.4f}  {: >8.4f}  {:
                                              >9.4f}\n".format(i + 1, \
                                  node_list[i].consumption.tx/1000,\
                                  node_list[i].consumption.rx/1000, \
                                  node_list[i].consumption.idle_rx/1000,\
                                  node_list[i].consumption.sleep/1000)

    print(text_to_print)


# Times...
time_pkt_43b_transmission = get_transmission_time(43)
```

```python
time_pkt_5b_transmission = get_transmission_time(5)
gap_between_wakeup_CAA = 0.744
gap_between_attempts_CAA = 0.6
time_of_one_CAA = 0.441
gap_between_transmissions_attempts = 0.838
time_one_transmission_attempt = gap_between_transmissions_attempts +
                                     time_pkt_43b_transmission
idle_time_after_ack_received = 0.252

# Calculate how many transmissions could happen between wakeup pariods
gap_to_transmit_between_wake_up = wake_up_period - gap_between_wakeup_CAA
                                      - 2*time_of_one_CAA
max_transmission_between_wakeup = numpy.floor(
                                     gap_to_transmit_between_wake_up/
                                     time_one_transmission_attempt)
mean_transmissions_first_burst_communication =
                                     max_transmission_between_wakeup/2
print("First burst transmission mean attempts = "+\
          str(mean_transmissions_first_burst_communication))

#Calculate consumption for every event
consumption_one_pkt_attempt = consumption_states("one pkt sent attempt")
consumption_one_pkt_attempt.tx = volt*curr_tx*time_pkt_43b_transmission
consumption_one_pkt_attempt.idle_rx = volt*curr_idle_rx*
                                     gap_between_transmissions_attempts
consumption_one_pkt_attempt.duration = time_pkt_43b_transmission+
                                     gap_between_transmissions_attempts
dump(consumption_one_pkt_attempt)

consumption_one_pkt_attempt_received = consumption_states("one pkt
                                     received attempt")
consumption_one_pkt_attempt_received.rx = volt*curr_rx*
                                     time_pkt_43b_transmission
consumption_one_pkt_attempt_received.idle_rx = volt*curr_idle_rx*\
                                 (gap_between_transmissions_attempts+\
                                 time_pkt_43b_transmission/2)
consumption_one_pkt_attempt_received.duration = time_pkt_43b_transmission
                                     +\
                                    gap_between_transmissions_attempts+\
                                    time_pkt_43b_transmission/2
dump(consumption_one_pkt_attempt)

consumption_one_CCA_attempt = consumption_states("one CCA attempt")
consumption_one_CCA_attempt.idle_rx = volt*curr_idle_rx*time_of_one_CAA
consumption_one_CCA_attempt.sleep = volt*curr_radio_sleep*gap_between_attempts_CAA
consumption_one_CCA_attempt.duration = time_of_one_CAA+gap_between_attempts_CAA
dump(consumption_one_CCA_attempt)

consumption_one_wakeup = consumption_states("one wakeup")
consumption_one_wakeup.idle_rx = volt*curr_idle_rx*time_of_one_CAA*CCA_per_wakeup
consumption_one_wakeup.sleep = volt*curr_radio_sleep*gap_between_wakeup_CAA*(CCA_per_wakeup-1)
consumption_one_wakeup.duration = time_of_one_CAA*CCA_per_wakeup + gap_between_wakeup_CAA*(CCA_per_wakeup-1)
dump(consumption_one_wakeup)

consumption_one_ACK_sent = consumption_states("one ACK sent")
consumption_one_ACK_sent.idle_rx = volt*curr_idle_rx*idle_time_after_ack_received
consumption_one_ACK_sent.tx = volt*curr_tx*time_pkt_5b_transmission
consumption_one_ACK_sent.duration = idle_time_after_ack_received + time_pkt_5b_transmission
dump(consumption_one_ACK_sent)

consumption_one_ACK_received = consumption_states("one ACK received")
consumption_one_ACK_received.idle_rx = volt*curr_idle_rx*idle_time_after_ack_received
consumption_one_ACK_received.rx = volt*curr_rx*time_pkt_5b_transmission
consumption_one_ACK_received.duration = idle_time_after_ack_received + time_pkt_5b_transmission
dump(consumption_one_ACK_received)

#Calculate the consumption of the first burst attempt
consumption_first_burst_communication = consumption_one_CCA_attempt*6\
            +consumption_one_pkt_attempt*\
            (mean_transmissions_first_burst_communication+1.5)\
            +consumption_one_ACK_received
consumption_first_burst_communication.name = "First burst communication"
dump(consumption_first_burst_communication)

#Calculate the consumption of the later communicatin with known destination already saved into phase
consumption_later_communications = consumption_one_CCA_attempt*6\
                                    +consumption_one_pkt_attempt*4.5\
                                    +consumption_one_ACK_received
consumption_later_communications.name = "later communications"
dump(consumption_later_communications)

#Calculate the consumption of the receiving communicatin
consumption_received_communications =
                                      consumption_one_pkt_attempt_received
                                      \
                                    +consumption_one_ACK_sent
consumption_received_communications.name = "received communications"
dump(consumption_received_communications)

# check the number of receptions not matching
total_application_pkt_attempts = 0
if application_period > 0:
    total_application_pkt_attempts = simulation_duration*60/
                                          application_period

total_keepAlive_pkt_attempts = 0
if keepAlive_period > 0:
    total_keepAlive_pkt_attempts = simulation_duration*60/
                                          keepAlive_period

total_wakeup_done_for_simulation = 1000*simulation_duration*60/
                                      wake_up_period
total_time_remaining_in_sleep_per_period = simulation_duration*60*1000\
                        -consumption_first_burst_communication.duration\
                        -consumption_later_communications.duration*\
                        (total_application_pkt_attempts+\
                        total_keepAlive_pkt_attempts-1)\
                        -consumption_one_wakeup.duration*\
                        total_wakeup_done_for_simulation
print("Total remaining sleep time = "+\
    str(total_time_remaining_in_sleep_per_period))
consumption_total_remaining_sleep_leaf_node = consumption_states("
                                      remaining sleep")
consumption_total_remaining_sleep_leaf_node.sleep = volt*\
    curr_radio_sleep*total_time_remaining_in_sleep_per_period
consumption_total_remaining_sleep_leaf_node.duration =\
    total_time_remaining_in_sleep_per_period
dump(consumption_total_remaining_sleep_leaf_node)


```

```
#Final consumption for a leaf node without colision and a root node
consumption_final_node_leaf_rank2_noColision =\
    consumption_first_burst_communication\
    +consumption_later_communications*\
    (total_application_pkt_attempts+total_keepAlive_pkt_attempts-1)\
    +consumption_one_wakeup*total_wakeup_done_for_simulation\
    +consumption_total_remaining_sleep_leaf_node
consumption_final_node_leaf_rank2_noColision.name = "Final leaf node in a
                                                      duo exp"
dump(consumption_final_node_leaf_rank2_noColision)

#------------------------
#                        Topology Analisis
#------------------------

class node:
    def __init__(self, id, father_node, rank):
        self.id = id
        self.father_node = father_node
        self.consumption = consumption_states("cons of node {}".format(id
                                                      ))
        self.rank = rank
        self.total_wakeups = 0
        self.total_rx = 0
        self.total_first_tx_attempts = 0
        self.total_later_tx_attempts = 0
        self.total_possible_collisions = 0
        self.total_sleep_time = 0

    def relayed_msg_to_add(self, number_to_add):
        if self.rank == 0:
            return
        # transmit the values spent of this node to its father node
        self.father_node.total_rx += number_to_add
        self.father_node.total_wakeups -= number_to_add
        self.father_node.total_sleep_time -=\
                consumption_one_pkt_attempt_received.duration
        if self.rank > 1:
            if self.father_node.total_first_tx_attempts == 0:
                #This device will need to do one first attempt
                self.father_node.total_first_tx_attempts = 1
                self.father_node.total_later_tx_attempts += number_to_add
                                                    -1
                self.father_node.total_sleep_time -=\
                    consumption_first_burst_communication.duration
                self.father_node.total_sleep_time -=\
                    consumption_later_communications.duration*(
                                                                        number_to_add -
                                                                        1)
            else:
                self.father_node.total_later_tx_attempts += number_to_add
                self.father_node.total_sleep_time -=\
                        consumption_later_communications.duration *\
                        number_to_add

        self.father_node.relayed_msg_to_add(number_to_add)

    def recursive_relay_messages(self):
        self.relayed_msg_to_add(self.total_first_tx_attempts\
                                +self.total_later_tx_attempts)


node_father = node(1, None, 0)
topologia_linear = [[node_father]]
node_n = None
for node_id in range(2, num_nodes + 1):
    node_n = node(node_id, node_father, node_id-1)
    node_father = node_n
    topologia_linear.append([node_n])


#Calculate the energy spent by this topoology
topology = topologia_linear

rank_size = len(topology)
for rank_list in topology:
    for device in rank_list:
        if device.rank == 0:
            device.total_wakeups = total_wakeup_done_for_simulation
            device.total_sleep_time = simulation_duration * 60 * 1000
        else:
            device.total_wakeups = total_wakeup_done_for_simulation
            device.total_sleep_time = simulation_duration * 60 * 1000
            if node_gen == 0 or node_gen == device.id:
                device.total_first_tx_attempts = 1
                device.total_later_tx_attempts =\
                    total_application_pkt_attempts\
                    +total_keepAlive_pkt_attempts-1
                device.total_sleep_time -=\
                    (consumption_first_burst_communication.duration\
                    +consumption_later_communications.duration*\
                    (device.total_first_tx_attempts\
                    +device.total_later_tx_attempts))
                device.recursive_relay_messages()

for rank_list in topology:
    for device in rank_list:
        sleep_cons = consumption_states("remaining sleep")
        device.total_sleep_time -= consumption_one_wakeup.duration * device.total_wakeups
        sleep_cons.sleep = volt * curr_radio_sleep * device.total_sleep_time
        sleep_cons.duration = device.total_sleep_time

        device.consumption = consumption_first_burst_communication*device.total_first_tx_attempts\
                    +consumption_later_communications*\
                    (device.total_later_tx_attempts)\
                    +consumption_one_wakeup*device.total_wakeups\
                    +consumption_received_communications*device.total_rx\
                    +sleep_cons

        device.consumption.name = "Final node {}".format(device.id)


#Dump devices
dump_topology(topology)
```

F PUBLISHED WORKS

Some part of this thesis has already been accepted in the following conference:

- V. G. GUIMARAES, A. BAUCHSPIESS, K. OBRACZKA, and R. M. DE MORAES, "A Novel IoT Protocol Architecture: Efficiency Through Data and Functionality Sharing Across Layers," *accepted at the 28th International Conference on Computer Communications and Networks (ICCCN 2019)*, Valencia, Spain, jul 2019.

Other publications during the period of the thesis:

- V. G. GUIMARAES, A. BAUCHSPIESS, and R. M. DE MORAES, "Dynamic Timed Energy Efficient and Data Collision Free MAC Protocol for Wireless Sensor Networks," *Journal of IEEE América Latina*, vol. 13, pp. 416-421, 2015.
- V. G. GUIMARAES, A. BAUCHSPIESS, and R. M. DE MORAES, "Customized MAC Protocol for Small Properties with Wireless Irrigation Automation," *in XXI Congresso Brasileiro de Automática – CBA2016*, Vitória, ES, Brazil, oct 2016, pp. 2956-2961.