AN NS-3 MODULE FOR SIMULATION OF SIMULTANEOUS WIRELESS INFORMATION AND POWER TRANSFER (SWIPT) OVER IEEE 802.11 NETWORKS

JOSÉ ANTÔNIO DE FRANÇA JUNIOR

DISSERTAÇÃO DE MESTRADO
EM ENGENHARIA ELÉTRICA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

AN NS-3 MODULE FOR SIMULATION OF SIMULTANEOUS WIRELESS INFORMATION AND POWER TRANSFER (SWIPT) OVER IEEE 802.11 NETWORKS

UM MÓDULO DO NS-3 PARA SIMULAÇÃO DE TRANSFERÊNCIA SIMULTÂNEA DE INFORMAÇÃO E ENERGIA SEM FIO (SWIPT) EM REDES IEEE 802.11

JOSÉ ANTÔNIO DE FRANÇA JUNIOR

ORIENTADOR: MARCELO MENEZES DE CARVALHO, DR.

DISSERTAÇÃO DE MESTRADO
EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGEE.DM-807/23
BRASÍLIA/DF: NOVEMBRO - 2023

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

An NS-3 Module for Simulation of Simultaneous Wireless Information and Power Transfer (SWIPT) over IEEE 802.11 Networks

José Antônio de França Junior

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:

Prof. Marcelo Menezes de Carvalho, Doutor (ENE/UnB)
(Orientador)

Prof. Hugerles Sales Silva, Doutor (ENE/UnB)
(Examinador Interno)

Prof. Marcos Augusto Menezes Vieira, Doutor (DCC/UFMG)
(Examinador Externo)

Brasília/DF, novembro de 2023.

FICHA CATALOGRÁFICA

FRANÇA JR., JOSÉ ANTÔNIO DE

An NS-3 Module for Simulation of Simultaneous Wireless Information and Power Transfer (SWIPT) over IEEE 802.11 Networks. [Brasília/DF] 2023.

xxx, 331p., 210 x 297 mm (ENE/FT/UnB, Mestre, Dissertação de Mestrado, 2023).

Universidade de Brasília, Faculdade de Tecnologia, Departamento de Engenharia Elétrica.

Departamento de Engenharia Elétrica

1. SWIPT	2. Energy Harvester
3. Coletor de Energia	4. Sustentabilidade Energética
5. WiFi	6. IEEE 802.11ah
7. Internet das Coisas	8. NS-3
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

FRANÇA JR., JOSÉ ANTÔNIO DE (2023). An NS-3 Module for Simulation of Simultaneous Wireless Information and Power Transfer (SWIPT) over IEEE 802.11 Networks. Dissertação de Mestrado, Publicação PPGEE.807/2023, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 331p.

CESSÃO DE DIREITOS

AUTOR: José Antônio de França Jr.

TÍTULO: An NS-3 Module for Simulation of Simultaneous Wireless Information and Power Transfer (SWIPT) over IEEE 802.11 Networks.

GRAU: Mestre ANO: 2023

José Antônio de França Jr.

Universidade de Brasília (UnB)

Campus Darcy Ribeiro

Faculdade de Tecnologia - FT

Departamento de Engenharia Elétrica (ENE)

Brasília - DF CEP 70919-970

Dedico este trabalho a você Max, por me ensinar o caminho da humildade, da paciência, da compreensão e por servir de exemplo para mim sem sequer proferir uma só palavra. Obrigado!

José Antônio de França Junior

AGRADECIMENTOS

Agradeço em primeiro lugar àquele que deu sua vida em meu lugar para a redenção dos meus pecados, a quem eu devo tudo que sou, tudo o que tenho e o que vier a ter, obrigado Deus! Agradeço a minha esposa Joyce por me auxiliar nesta jornada e por tudo que ela tem proporcionado em minha vida. Agradeço à Senhora Ana Maria da Silva França que desde minha infância sempre me estimulou ao conhecimento da ciências exatas. Agradeço ao meu melhor amigo, Max, pela amizade incondicional. Agradeço ao Professor Marcelo Carvalho pelos ensinamentos, por ter me aceito como seu orientando e por toda a paciência que tem tido comigo desde o início. Agradeço ao Professor Paulo Gondim por ter me acolhido e me orientado no início desta jornada. Agradeço a todos os Professores do PPGEE da UnB pelos ensinamentos. Agradeço a todos os servidores da Anatel que me ajudaram para que eu pudesse ingressar nesta jornada. Em especial agradeço ao Sr. Vinícius Oliveira Caram Guimarães e ao Sr. Davison Gonzaga da Silva por terem me concedido esta oportunidade. Agradeço ao Sr. Leonardo Marques Campos por me impulsionar a ser melhor a cada dia. Agradeço ao Sr. Thiago Barçante pelo companheirismo e por ter me auxiliado em tudo. Agradeço à Sra. Walkiria Cassimiro Pereira por todo o apoio. Agradeço à Professora Naja pelos ensinamentos que eu não poderia obter em outra fonte.

ABSTRACT

Our contemporary interconnected world faces a significant challenge due to the escalating energy requirements demanded by the proliferation of Internet of Things (IoT) devices projected to saturate the market in the coming years. In response to this challenge, Simultaneous Wireless Information and Power Transfer (SWIPT) emerges as a compelling solution, allowing devices to be powered by received signals intended for information transfer. Despite the significance of SWIPT, the majority of scientific research conducted on this subject has predominantly fixated on the hardware aspects of the Physical (PHY) layer, with limited exploration of SWIPT in broader contexts of standard network simulation deployments. In light of this, our primary research objective was to developed an SWIPT module using the power splitting (PS) technique, specifically tailored for the integration into the Network Simulator 3 (NS-3), in order to enable us to simulate SWIPT within IEEE 802.11 networks. Thus, in the following pages, we provide a thorough explanation of the theoretical foundations that support our research, including the fundamental principles of SWIPT, IEEE 802.11ah and NS-3. Then, we present the outcomes of an extensive survey we conducted to evaluate the current state of scientific knowledge regarding the simulation of SWIPT within standardized network architectures. Following this, we present the conceptual framework of our design, which closely aligns with the split architecture of the SWIPT receiver, incorporating the PS technique. After that, we delve into the technical details of the implementation of the SWIPT module into NS-3, including the key classes and other major dependencies. Subsequently, we present the results obtained from extensive simulations we conducted of our implementation on the NS-3, encompassing both single-link and network scenarios, using key metrics such as Signal-to-Noise and Interference Ratio (SNIR), throughput, frame loss, total harvested power, total harvested energy, and sustainability. In conclusion, we summarize the outcomes achieved through our research and outline the future directions we aspire to explore.

Keywords: IoT, SWIPT, NS-3, IEEE 802.11ah, Energy Sustainability.

RESUMO

No contexto atual de um mundo cada vez mais interconectado, surgem grandes desafios relacionados à demanda energética decorrente das crescentes projeções sobre a utilização de milhares de dispositivos de Internet das Coisas (do Inglês: *Internet of Things* - IoT). Em resposta, a tecnologia de Transferência Simultânea de Informação e Energia sem Fio (do Inglês: *Simultaneous Wireless Information and Power Transfer* - SWIPT) apresenta-se como alternativa promissora, em razão do seu potencial para fornecer energia elétrica extraída a partir de sinais eletromagnéticos intencionalmente recebidos. Contudo, a maior parte das pesquisas sobre SWIPT tem concentrado esforços no desenvolvimento de soluções relacionadas à camada física, com pouca atenção voltada à investigação do SWIPT em ambientes de simulação de redes. Diante disso, a presente pesquisa tem como objetivo principal o desenvolvimento de um módulo de SWIPT, utilizando a técnica de divisão de potência, sobre a plataforma do Simulador de Redes 3 (do Inglês: *Network Simulator 3*, NS-3), de modo a promover simulações do SWIPT em redes IEEE 802.11. Assim, nas páginas a seguir, discorrem-se sobre os fundamentos teóricos que subsidiam o tema desta pesquisa, incluindo-se conceitos sobre SWIPT, IEEE 802.11ah e NS-3. Em seguida, relatam-se os resultados de uma extensa pesquisa realizada para avaliar o estado atual do conhecimento científico sobre a simulação do SWIPT em plataformas computacionais. Após isso, apontam-se os aspectos conceituais aplicados ao desenvolvimento do módulo SWIPT. Adiante, passa-se a expor o detalhamento referente à implementação do módulo SWIPT no NS-3, incluindo as principais classes e outras dependências relevantes. Subsequentemente, apresentam-se os resultados das métricas de relação sinal-ruído e interferência (SNIR), vazão, perda de quadros, potência total coletada, energia total coletada e sustentabilidade energética, obtidos a partir de simulações realizadas em cenários de enlace único e de rede. Por fim, exprimem-se as conclusões sobre os resultados alcançados e delineiam-se as futuras direções a serem exploradas para a evolução da investigação sobre o tema.

Palavras-chave: IoT, SWIPT, NS-3, IEEE 802.11ah, Sustentabilidade Energética.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

α	Time-switching coefficient
β	DC-DC conversion efficiency factor
η	AC-DC conversion efficiency factor
θ_s	Power split factor

GLOSSARY

AC	Alternating Current
AP	Access Point
API	Application Programming Interface
AWGN	Additive White Gaussian Noise
BSS	Basic Service Set
BER	Bit Error Rate
CSB	Cross Slot Boundary
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
dBm	Decibel-milliwatts
DC	Direct Current
DTIM	Delivery Traffic Indication Map
ED Threshold	Energy Detection Threshold
EH	Energy Harvester
GI	Guard Intervals
HFC	Hybrid Fiber-Coax
ID	Information Decoder
IoT	Internet of Things
MAC	Medium Access Control Sub-layer
MCS	Modulation and Coding Schemes
MSDU	MAC Service Data Unit aggregation

MPDU	MAC Protocol Data Unit aggregation
NSS	Number of Spatial Streams
PER	Peak Error Rate
PHY	Physical Layer
PLCP	Physical Layer Convergence Procedure
PS	Power Splitting
PSDU	PLCP Service Data Unit
PPDU	PLCP Protocol Data Unit
PSTN	Public Switched Telephone Networks
RAW	Restricted Access Window
RF	Radio Frequency
S1G	Sub 1 GHz
SNIR	Signal-to-Noise-plus-Interference Ratio
SNR	Signal-to-Noise Ratio
SWIPT	Simultaneous Wireless Information and Power Transfer
TIM	Traffic Indication Map
TS	Time Switching
TWT	Target Wake-up Time
VHT	Very High Throughput
WET	Wireless Energy Transfer
WIT	Wireless Information Transfer
WPT	Wireless Power Transfer
NI Changes	Network Interface Changes
RNG	Random Number Generator

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Our technology-interconnected world faces a challenge driven by the ever-increasing energy demands for powering a multitude of Internet of Things (IoT) devices, as described in the work of Khairy *et al.* (2019). This challenge is compounded by the deployment of IoT devices deployed in remote and hard-to-reach locations, creating logistical complexity for maintenance and battery replacements, as discussed by (AL-SARAWI *et al.*, 2020).

In response to this challenge, Simultaneous Wireless Information and Power Transfer (SWIPT) represents a promising solution to enable devices to achieve energy self-sustainability. It allows this by capturing energy from incoming electromagnetic signals originally intended for data transmission. The integration of SWIPT acts as a prospective solution to bridge the gap between the challenge posed by the increasing energy demands and the possibility of providing sustainable energy sources.

The inaugural study on SWIPT, introduced by Varshney (2008), envisioned its applicability in scenarios involving power-constrained devices that could receive both energy and information simultaneously. Remarkably, as time has passed since SWIPT was first coined, its relevance has never been more apparent than in the present day. Today, SWIPT finds itself most urgently applicable to provide energy to tangible IoT network devices operating in real world applications. Furthermore, SWIPT has the potential to revolutionize traditional paradigms of wireless communication and power provisioning by replacing batteries with supercapacitors, as emphasized by (ROSA *et al.*, 2018).

Therefore, this dissertation embarks on a comprehensive exploration of SWIPT using the power splitting technique, which we will further describe in details, delving into its foundational principles, technological aspects, and implementation into the Network Simulator 3 (NS-3),

which is platform capable resembling real-world network scenarios. Throughout the following chapters, we will unravel the details of SWIPT, examine the development of SWIPT as an abstraction into NS-3, and showcase the results obtained from its simulations. By the end of this journey, we aim to provide the conclusions of our SWIPT implementation, its capabilities, limitations, and the possibilities it may offer for future investigations.

1.2 MOTIVATION AND CHALLENGES

The primary driving force that led us to undertake the scientific investigation was the unavailability of research dedicated to exploring SWIPT within real-word network standards using simulation platforms. This gap in the literature is particularly apparent as the predominant focus of the existing research on SWIPT primarily centers around the hardware of physical layer (PHY).

The challenges we faced towards accomplishing the objective of developing an SWIPT object model are related to thoroughly examining the code that models the features and functions of the existing classes in the NS-3 library. This in-depth exploration was essential to acquire the expertise needed to understand and manipulate its operations effectively.

Additionally, we faced challenges related to implementing our SWIPT abstraction within the NS-3 library. This involved dealing with various modules, including those related to WiFi classes and the energy framework of the NS-3 library. We also had to make decisions regarding the most suitable location for the implementation to accurately represent SWIPT operations.

In summary, our motivation for this investigation originated from the apparent scarcity of research focused on characterizing SWIPT within computational simulation platforms. This challenge led us to embark on this inquiry, starting with a comprehensive examination of the operations of preexisting classes, followed by the implementation of our model.

1.3 RESEARCH AIMS AND OBJECTIVES

The primary objective of this research was to develop an SWIPT object model, featuring the power splitting technique, into the NS-3.23 library, in order to further enable the conduction

of simulations of this solution within the available WiFi standards already integrated into NS-3.

Therefore, we aimed to design the implementation of the SWIPT object model with user-customizable attributes, facilitating the exploration of SWIPT under diverse network configurations, scenarios, and parameters. Additionally, we aimed to streamline the usage of our SWIPT implementation by providing a convenient helper function within configuration scripts.

Subsequently, we aimed to simulate our SWIPT implementation in two different scenarios. The first consisted of a single-link network architecture, in order to allow us to initially characterize the behavior of our SWIPT implementation and identify primary constraints that could potentially impact network performance and the capacity of the SWIPT object model to harvest energy.

Following this, our aim was to enhance the complexity of the testing scenario and perform additional simulations, in order to thoroughly characterize the overall behaviour of our SWIPT implementation in a network environment where multiple nodes contend for medium access, while considering other commonly encountered network characteristics.

1.4 CONTRIBUTIONS

The major contributions provided by this research are summarized as follows:

1. We present a novel contribution to the research community in the form of an extension to the NS-3.23 library allowing the simulation of SWIPT into a wide range of wireless network standards.

2. Our implementation can be deployed over any network scenarios that comply with the IEEE 802.11 standards available on NS-3.23, such as the IEEE-802.11a/b/g/n/ah. This is attributed to the development of our implementation as an independent extension of the energy framework of NS-3. Therefore, it can serve as a tool for other researches that may need to simulate SWIPT using these standards.

3. The development of the library is based on the extended energy framework presented in (TAPPARELLO *et al.*, 2014a; TAPPARELLO *et al.*, 2014b), which enables interoperability between the SWIPT harvester and future energy source models, such as super capacitors;

4. We have extended the PHY listener object resulting in a more flexible SWIPT harvester operation that could be deployed for harvesting energy in different states of the PHY;

5. The SWIPT harvester architecture is based on adjustable attributes of power splitting factor (θ_s), energy harvester efficiency factor (η) and battery conversion efficiency factor (β) to enable the exploration of SWIPT under various configurations and settings.

6. Our implementation includes the energy harvesting of interfering frames, which may overlap in time with the reception of an intended frame during the same RX state.

1.5 DISSERTATION ORGANIZATION

This dissertation is structured as follows:

In Chapter 2, we delve into the foundational aspects of SWIPT, elucidating the two most relevant techniques that have played a crucial role in its development. Additionally, we explore the important features of the IEEE 802.11ah, aligning them with our research objectives, presenting a comprehensive examination of the PHY, with a specific focus on the protocol data unit, and providing a description of the most relevant technical features related to the Sub 1 GHz PHY. We also provide a brief overview of the Restricted Access Window (RAW) and Traffic Indication Map (TIM) features of the IEEE 802.11ah, which are integral components of our simulation setup. Subsequently, we offer a succinct overview of NS-3, outlining the configuration scripts employed for running simulations and the Waf program, which is a python-based build tool. Our exploration then delves into the significant entities within the WiFi model library and the energy framework. Concluding the chapter, we provide a concise summary of our implementation on the IEEE 802.11ah NS-3 simulation module, aligning it with the theoretical foundations introduced earlier in the chapter.

After that, in Chapter 3, we present the findings of an in-depth survey conducted to assess the current scientific knowledge related to simulating SWIPT on computational platforms. The goal is to establish a common basis for comparison with our work, providing a clear understanding of the existing state of research in this area.

Following this, in Chapter 4 we outline the conceptual framework of the design of the SWIPT

module, which closely aligns with the split architecture of the SWIPT receiver, incorporating the power splitting technique. The design of our SWIPT module is presented under two complementary perspectives. The first perspective is illustrated as a block diagram, which is intended to provide a comprehensive overview of how the components function as a unified system. Meanwhile, the second perspective shows the integration of our design within the architecture of the wireless interface controller, namely, `WifiNetDevice`, in order to provide a clear understanding of how the concept of our SWIPT module is intended to fit into the NS-3 library.

Further ahead, in Chapter 5, we present a detailed overview of our implementation of SWIPT into NS-3. Our main objective is to explain how we enabled the simulation of this technology within the context of IEEE 802.11ah networks . We approach this exploration by showcasing its connections to primary classes and illustrating its dependencies within objects of the NS-3 library.

Additionally, in Chapter 6, we present the results obtained from simulating our implementation in two specific scenarios. The first scenario involved simulating SWIPT within an IEEE 802.11ah network architecture featuring a single link. Our primary objective in this scenario was to identify major constraints and explore potential impacts on both network performance and the energy harvesting capabilities of our implementation. In the second scenario, we examined the results of simulating SWIPT within a network environment consisting of RAW groups of SWIPT-enabled nodes, facing constraints typically encountered in IEEE 802.11ah networks. Our goal in this scenario was also to characterize major constraints and investigate potential impacts on both network performance and the energy harvesting capabilities of our implementation.

Finally, in Chapter 7, we provide the conclusions drawn from our research and engage in a discussion about future work we intend to explore for further enhancements of our implementation. Moreover, we have presented the following paper as a result of our investigation at the Toll Hall of the XLI Brazilian Symposium on Computer Networks and Distributed Systems (SBRC 2023):

JUNIOR, J. A. de F.; CARVALHO, M. M. Uma extensão do ns-3 para simulação de transferência simultânea de informação e energia sem fio (SWIPT) em redes IEEE 802.11. In: SBC. Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. [S.l.], 2023. p. 48–55.

CHAPTER 2

THEORETICAL BACKGROUND

In this chapter, we introduce the theoretical foundations supporting our research through a comprehensive exploration of key concepts. This includes an in-depth examination of SWIPT, relevant features of the IEEE 802.11ah standard, essential topics regarding NS-3, and an overview of the major aspects of our implementation. This coverage serves as the essential framework upon which our research discourse is constructed.

2.1 SWIPT BACKGROUND

In this section, we delve into the fundamental technical aspects of SWIPT. Our exploration starts with a brief introduction about the origins and inception of SWIPT. Building upon this foundation, we proceed to elucidate the crucial concepts and mechanisms that support this technology. By doing so, we aim to provide a comprehensive understanding of the complex interaction between wireless data communication and energy harvesting.

Historically, the fields of study related to the transmission of information and energy transferring have been explored as separate and distinct disciplines. This segregation has played a crucial role in fostering advancements and innovations within each specialized domain. As a consequence, it led to the formal division of electrical engineering into two discrete sub-fields: electric power engineering and communication engineering, as chronicled in (VARSHNEY, 2008) and corroborated by (CHEN, 2019).

However, in certain network scenarios, it became apparent that consolidating technologies from both areas of knowledge together within a single transmission medium was a viable approach. Notable examples include the Public Switched Telephone Networks (PSTN), where landline phones are powered by signals that travel in the same network medium that carries voice calls; and the Hybrid Fiber-Coax (HFC) networks, where data signals and electrical power

can be transmitted downstream to provide services to consumers and, at the same time, energy to power the amplifiers that boost the signals which travel along the coaxial cables of the network, respectively. This kind of integration aims to achieve cost reduction and enhance efficiency. Similarly, SWIPT, a contemporary wireless communication technology, combines the characteristics of Wireless Information Transfer (WIT) and Wireless Energy Transfer (WET) over the same electromagnetic signal, allowing for information decoding and energy harvesting to be achieved simultaneously at the receiver.

These considerations serve as the foundation for tackling one of the most significant challenges in SWIPT receivers design: finding the optimal trade-off between information transmission rate and energy harvesting, often referred to as the R-E (Rate-Energy) region, as elaborated in (CLERCKX *et al.*, 2019). This trade-off can be viewed from our perspective as a commitment to uphold a set of imperatives, including: maintaining precision in information decoding, ensuring that the harvested energy attains sufficient levels to provide devices with self-sustaining power autonomy, all the while upholding network performance standards without compromising energy harvesting efficiency.

Toward these objectives, two fundamental techniques come into play. The first technique is known as Time Switching (TS), which splits the received electromagnetic signal in the time domain. The second, Power Splitting (PS), splits the signal in the power domain, as elaborated in (CHEN, 2019).

It is also worth noting that certain SWIPT designs may incorporate a combination of both TS and PS techniques to enhance network performance and energy harvesting, as expounded in references (JAMEEL *et al.*, 2016), (JIANG *et al.*, 2019) and (WANG *et al.*, 2021).

Therefore, in the upcoming subsections, we first provide an overview of TS and then delve deeper into PS, as it is the technology employed in our project.

2.1.1 Time Switching

The Time-Switching (TS) technique employs a time-division strategy to distribute incoming power within a frame across the inputs of the information decoder and the energy harvester during distinct time intervals. This approach requires the division of the transmitted signal

into two segments: the information component and the energy component. The specific timing at which the information segment of the signal is directed to the information decoder and the energy segment to the energy harvester is determined by a parameter referred to as the time-switching coefficient, denoted by α, where $0 \leq \alpha \leq 1$.

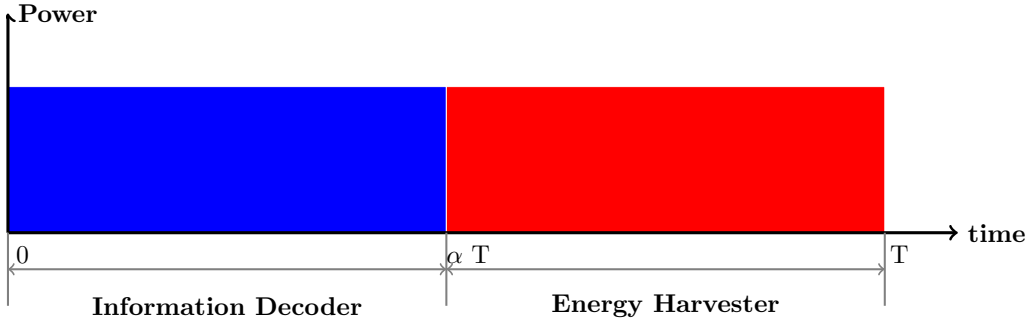
According to Chen (2019), the TS technique can be characterized as follows. First, the total duration of an incoming frame, consisting of an information portion and an energy portion is assumed to be T seconds. Consequently, the fraction of time corresponding to the information part can be expressed as αT and also the energy part as $(1-\alpha)T$, which leads to

$$y(t) = \begin{cases} h\sqrt{P_s}s(t) + n(t), & 0 \leq t \leq \alpha T, \\ h\sqrt{P_s}s(t) + n(t), & \alpha T \leq t \leq T, \end{cases} \tag{2.1}$$

where P_s is the transmission power, $s(t)$ is the transmission symbol, h is the complex channel gain and $n(t)$ is the additive white Gaussian noise (AWGN) with zero mean and variance $2\sigma^2$.

The TS technique can be visualized as illustrated in Fig. 2.1. This diagram provides a time-domain representation, demonstrating how a received frame is divided into two time intervals, with one allocated for the information decoder and the other designated for the energy harvester.

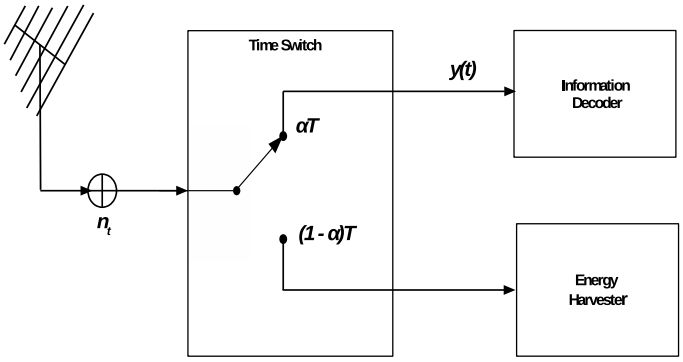
Figure 2.1. TS Allocation of a Frame in the Time-Domain.



Source: (CHEN, 2019)

Fig. 2.2 provides a visual representation of an SWIPT receiver that utilizes the TS technique. This SWIPT receiver includes both an information decoder and an energy harvester, both located within the same node. In this setup, the TS SWIPT receiver alternates its operational mode periodically, switching between decoding information during the time slot designated for data transmission and harvesting energy during the time slot allocated for power transfer.

Figure 2.2. TS SWIPT Receiver Block Diagram.



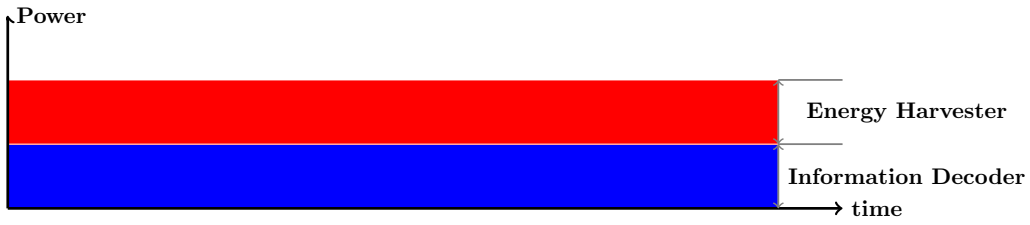
Source: (CHEN, 2019)

2.1.2 Power Splitting

The Power Splitting (PS) technique is a method used for splitting an incoming RF signal into two separate power components. This division is achieved by allocating a portion of the power of the signal to each component, and this allocation is controlled by a parameter called the power splitting factor (θ_s), where $0 \leq \theta_s \leq 1$.

In practical terms, this means that when an RF signal is received, it is divided into two parts: one part is sent to an information decoder and the other part is directed to an energy harvester. This process is visually represented in Fig. 2.3.

Figure 2.3. PS Allocation of a Frame in the Power-Domain.

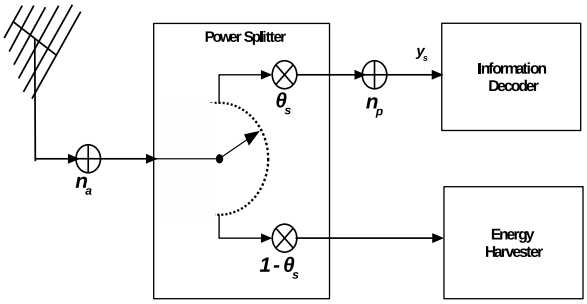


Source: (CHEN, 2019)

For this reason the majority of SWIPT receivers adopt a split architecture, segregating the information decoder from the energy harvester, as depicted in Fig. 2.4. The internal

configuration of information decoders is tied to the chosen technology. Nevertheless, certain internal mechanisms of the information decoder play a pivotal role in the operations of the energy harvesters, namely SWIPT harvester, thus rendering information decoders an indispensable component of SWIPT receivers.

Figure 2.4. PS SWIPT Receiver Block Diagram.



Source: (CHEN, 2019)

Whereas, SWIPT harvesters generally comprise the subsequent components: an antenna to capture electromagnetic signals, a matching circuit for ensuring impedance compatibility, a power splitter to perform θ_s, a rectifier responsible for converting electromagnetic signals alternating current (AC) levels into direct current (DC), a Power Management Unit (PMU) overseeing power distribution and synchronization with the information decoder, a DC-DC converter for optimizing the efficiency of the delivered DC voltage to the energy source, and an energy storage element for accumulating the harvested energy.

To model the operation of PS SWIPT, we extend the signal model initially proposed by Luo *et al.* (2021) to accommodate the path loss models specifically adopted for IEEE 802.11ah networks in (HAZMI *et al.*, 2012). As a result, the voltage level y_s that is forwarded to the input of the information decoder is given by

$$y_s = \sqrt{\theta_s}\left(\frac{h\sqrt{G_t G_r P_t}}{\sqrt{L(d)}}x_s + n_a\right) + n_p, \tag{2.2}$$

where θ_s is the PS factor, P_t is the source transmit power, x_s is the transmit symbol with unit

power, h is the channel gain due to small-scale fading, n_a is the noise at the antenna unit, and n_p is the noise at the signal processing unit, both modelled as Gaussian distributions, denoted as $\mathrm{N}(0, \sigma_a^2)$ and $\mathrm{N}(0, \sigma_b^2)$, respectively.

For the large-scale path loss $L(d)$ between transmitter and receiver, we consider the two outdoor scenario models adopted for IEEE 802.11ah networks in (HAZMI *et al.*, 2012): for the macro deployment path loss model, the antenna height is assumed to be 15 m above rooftop, whereas the pico or hot zone deployment path loss model assumes the antenna at roof top level. In both cases, the path loss (in dB) is given by

$$PL(d) = \xi + 37.6 \log_{10}(d) + 21 \log_{10}\left(\frac{f}{900\mathrm{MHz}}\right), \tag{2.3}$$

where f is the carrier frequency, d is the distance in meters, $\xi = 8$ for macro deployment, and $\xi = 23.3$ in the pico scenario.

For indoor scenarios, the path loss model for IEEE 802.11ah networks is derived by adjusting the frequency operation of the IEEE 802.11n model in a manner that scales it down. The IEEE 802.11n model is based on the Friis space loss up to a specific breakpoint distance d_{BP} and exhibits a slope of 3.5 after surpassing that distance. Consequently, the path loss model for IEEE 802.11ah networks can be described by

$$PL(d) = \begin{cases} L_{FS}, & \text{if } d \leq d_{BP} \\ L_{FS} + 3.5 \log_{10}(d/d_{BP}), & \text{if } d > d_{BP} \end{cases} \tag{2.4}$$

where $L_{FS} = 20 \log_{10}(4\pi f d/c)$, f is the carrier frequency, and c is the speed of light.

A fading margin M (expressed in dB) is usually ensured to maintain a certain minimum received power level $P_{\min}$. This is done to account for the impact of propagation effects like shadowing and small-scale fading. Hazmi *et al.* (2012) have furnished typical values for the fading margin in outdoor scenarios.

Consequently, the linear path loss $L(d)$ to be employed in Eq. (2.2) can be expressed in a general form as

$$L(d) = 10^{\frac{PL(d)}{10}} = \kappa 10^{\gamma} d^{\alpha}, \tag{2.5}$$

where $\kappa = (f/9 \times 10^8)^{2.1}$, $\alpha = 3.76$, $\gamma = 0.8$ (macro) or $\gamma = 2.33$ (pico). For indoor scenarios, $\kappa = (4\pi f/c)^2$, $\gamma = 0.1M$, and $\alpha = 2$ if $d \leq d_{BP}$. If $d > d_{BP}$, $\kappa = (4\pi f/c)^2 (1/d_{BP})^{0.35}$, and $\alpha = 2.35$.

From Eq. (2.2), we also extend the the harvested energy model, E_b, delivered to the battery or supercapacitor upon receiving a symbol, which was initially proposed by Luo *et al.* (2021), in order to accommodate the path loss models of the IEEE 802.11ah standard proposed by Hazmi *et al.* (2012) as follows

$$E_b = \beta\eta(1-\theta_s)\left(\frac{|h|^2 G_t G_r P_t}{L(d)} + \sigma_a^2\right) T_s, \tag{2.6}$$

where T_s is the symbol duration, η is the AC/DC efficiency conversion factor of the energy harvester, and β is the DC/DC efficiency conversion factor of the battery. Since σ_a^2 is usually very small, we assume $\sigma_a^2 = 0$ (LUO *et al.*, 2021).

2.2 IEEE 802.11AH BACKGROUND

Within this section, we undertake on an exploration of the fundamental concepts tied to the IEEE 802.11ah standard, renowned as 'WiFi Halow', since the results presented in Chapter 6 were derived from simulations conducted on our SWIPT implementation using this technology and its key features discussed in the following subsections.

The term 'WiFi Halow' is the trademark for products that adhere to the specifications of the IEEE 802.11ah standard. These products capitalize on the global acceptance of earlier versions of the IEEE 802.11 standard to extend communication capabilities into the realm of IoT devices.

As per the specifications outlined in Amendment 2 of the IEEE 802.11-16 standard (IEEE, 2017a), the protocol stack of IEEE 802.11ah is characterized as an integrated collection of enhancements built upon the foundations of earlier versions of the IEEE 802.11. These improvements encompass both the Physical layer (PHY) and the Medium Access Control sublayer (MAC).

The IEEE 802.11ah standard offers several prominent features, such as the deployment of as many as 8,192 wireless devices within a single basic service set (BSS). This deployment occurs within license-exempt radio frequency (RF) bands below 1 GHz (S1G), enabling effective coverage of distances spanning up to 1 km. Other important features include the Restricted Access Window (RAW), Traffic Indication Map (TIM). Remarkably, all these capabilities can be seamlessly integrated into compact, miniature devices powered by small coin-shaped batteries.

2.2.1 The Physical Layer

This subsection provides an overview of the crucial components and functions of the PHY to facilitate the explanation of our implementation in the upcoming chapters.

We commence by highlighting the division of the PHY into two distinct sub-layers: the upper sub-layer is known as the Physical Layer Convergence Procedure (PLCP), while the lower sub-layer is the Physical Medium Dependent (PMD) sub-layer.

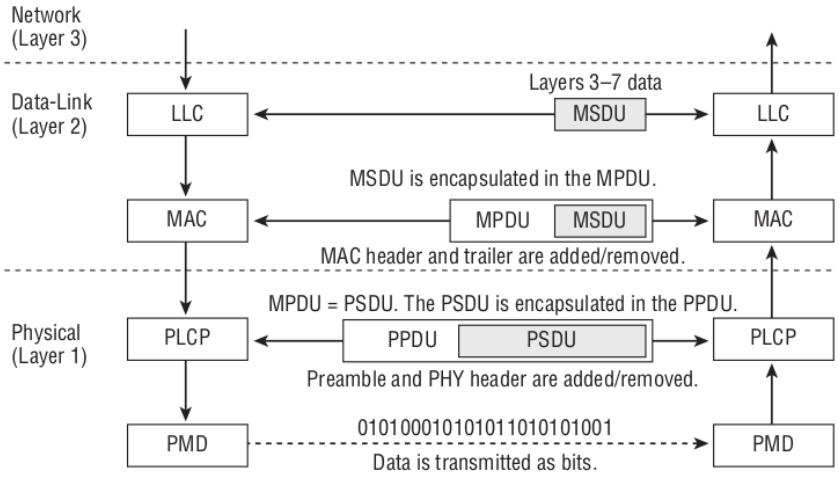
The PLCP sub-layer is responsible for handling the operations of transmission and reception of data related to the MAC sub-layer. The PLCP transmission process involves encapsulating the data received from the MAC, namely, MAC Protocol Data Unit (MPDU), into a PLCP Protocol Data Unit (PPDU). This is accomplished by adding a preamble and a header to the MPDU, which at this level is mapped into the PLCP Service Data Unit (PSDU). Then, the PPDU is passed down to the PMD sub-layer, which takes care of encoding, modulation and transmission of the PPDU as bits in the wireless channel, as illustrated on Fig. 2.5.

Nonetheless, before proceeding, it is important to emphasize that the preamble and header of the PPDU are transmitted at lower data rates defined by the Modulation and Coding Schemes (MCS) while the PSDU is transmitted at higher data rates. This distinction is deliberately designed because transmitting the preamble and header at lower data rates offers several advantages for ensuring flawless frame reception, including higher levels of Received Signal Strength Indicator (RSSI), improved Signal-to-Noise Ratio (SNR), lower Bit Error Rate (BER), and more.

As a result, this approach may contribute to the improvement in the reception of preamble synchronization sequences, control information fields and other frame-related parameters by

nodes within a network. This initial step is crucial for establishing a robust foundation prior to transmitting substantial volumes of payload data at greater speeds, which may lead to higher throughput levels, as described in (WESTCOTT *et al.*, 2011).

Figure 2.5. Encapsulation of the Data-link and Physical Layers Protocol Data Units



Source: (WESTCOTT *et al.*, 2011).

2.2.1.1 Sub 1 GHz PHY

Regarding the Sub 1 GHz PHY, our focus is on providing critical technical details specified in the IEEE Std 802.11ah-2016, as referenced in (IEEE, 2017b). Our goal is to align this information with the parameters utilized in our simulations.

Specifically for the IEEE 802.11ah, as documented in (IEEE, 2017b), the PHY is referred to as the Sub 1 GHz (S1G) PHY. This designation reflects its operation in frequency bands below 1 GHz, including the 900 MHz frequency band within the Industrial, Scientific, and Medical (ISM) band.

The S1G PHY is designed to support communication through frequency channels with multiple bandwidth options, including 1 MHz, 2 MHz, 4 MHz, 8 MHz, and 16 MHz. Furthermore, it can utilize different combinations of Modulation and Coding Schemes (MCS), along with varying Number of Spatial Streams (NSS) and Guard Interval durations (GI).

As an illustrative example, Table 2.1 highlights the MSC provided by the IEEE 802.11ah for channel bandwidths of 1 MHz and 2 MHz. This presentation specifically details information

when the NSS is configured to 1, and the GI is set to 8 μs, since these parameters were configured in our simulations, as previously mentioned.

Table 2.1. Modulation and Coding Schemes for IEEE 802.11ah using NSS=1 and GI=8 μs

MCS Index	Modulation	Coding Rate	Data Rate (kbps)	
			1 MHz	2 MHz
0	BPSK	1/2	300	650
1	QPSK	1/2	600	1300
2	QPSK	3/4	900	1950
3	16-QAM	1/2	1200	2600
4	16-QAM	3/4	1800	3900
5	64-QAM	2/3	2400	5200
6	64-QAM	3/4	2700	5850
7	64-QAM	5/6	3000	6500
8	256-QAM	3/4	3600	7800
9	256-QAM	5/6	4000	-
10	BPSK	1/2 Rep-2	150	-

Source: (TIAN *et al.*, 2016)

In Table 2.1, the coding rate of 1/2 Rep-2 for the MCS 10 signifies that the 6 information bits used for each OFDM symbol are encoded with a coding rate of a 1/2. Then, the resulting 12 encoded bits in each OFDM symbol are block-wise repeated twice.

The specific operational characteristics, which involve utilizing narrow bandwidths, employing S1G frequency bands, and selecting MCS index 10 from Table 2.1, collectively enable coverage extension up to 1 km. Furthermore, the S1G PHY inherits mechanisms and attributes from the Very High Throughput (VHT) PHY as specified in the IEEE 802.11ac standard. Additionally, it provides the MAC layer with the following services:

1. Encapsulation of MPDUs into PPDUs for facilitating the transmission and reception operations between nodes within a network.

2. Handling the transmission and reception of Orthogonal Frequency Division Multiplexing (OFDM) signals in the wireless channel.

2.2.2 RAW Mechanism

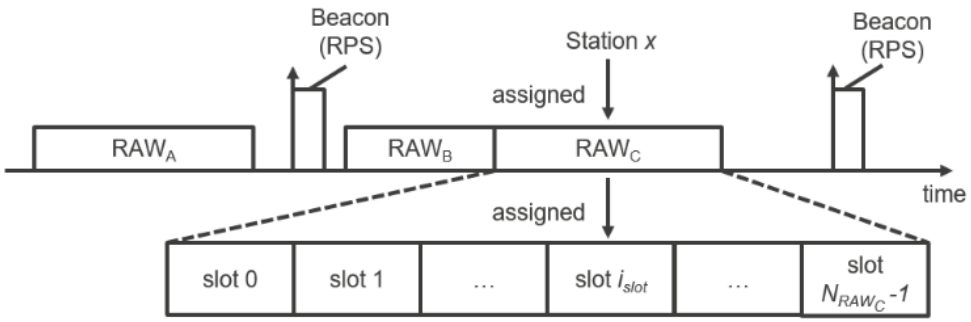
The primary objective of the RAW mechanism is to reduce the collision domain of the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol of the MAC layer. This is accomplished by dividing or the RF channel in the time domain into distinct time-slots, which are assigned to specific groups of stations. Therefore, as the mane implies, the RAW mechanism restricts stations organized in groups, namely RAW groups, to content for channel access only during specific windows of time, denominated RAW slots.

Ultimately, this feature was designed with the objective of increasing the probability that a transmitted frame is successfully received at the destination, reducing the likelihood of collisions and avoiding re-transmissions. Additionally, outside the period of their RAW slot assignment, stations are allowed to enter into sleeping mode in order to save energy.

The information regarding which RAW group stations belong to and which RAW slot they are allowed to use is conveyed inside the RAW Parameter Settings (RPS), which is broadcast by the Access Point (AP) in beacon frames at specific periods of time.

An illustrative example of the RAW feature is depicted in Figure 2.6. This diagram focuses on the details of the assignment of time-slot i to station x, which belongs to RAW group c, by the RPS information conveyed into the first beacon frame depicted. In the overall illustration of the diagram, beacon frames divide the RF channel into periodic time intervals. These beacon frames broadcast RPS information regarding different RAW groups, ranging from RAW Group$_A$ to RAW Group$_C$. The channel is then further subdivided into time slots, labeled slot$_0$ to slot$_{N-1}$, which are specifically allocated to the station of RAW Group$_C$.

Figure 2.6. RAW Mechanism



Source: (ŠLJIVO *et al.*, 2018).

As specified in the IEEE Std 802.11ah-2016 of reference (IEEE, 2017a), the duration of each RAW slot is defined as D_{SLOT}, according to the RAW Slot Definition sub-field of the RPS element and is given by

$$D_{\mathrm{SLOT}} = 500\mu s + C_{\mathrm{SLOT}} \times 120\mu s, \tag{2.7}$$

where C_{SLOT} is the value of the Slot Duration Count sub-field.

Another relevant sub-field of the RPS element that should be mentioned at this point refers to the Cross Slot Boundary sub-field (CSB), which indicates whether nodes are allowed to transmit after the duration of their RAW slot assigned.

Additionally, the RAW duration, denoted as D_{RAW}, expressed in units of μs, indicates the duration of the restricted medium access assigned to RAW. Moreover, D_{RAW} should be less or equal to the beacon interval. D_{RAW} is given by

$$D_{RAW} = D_{\mathrm{SLOT}} \times N_{RAW}, \tag{2.8}$$

where N_{RAW} is the number of RAW slots.

2.2.3 TIM Segmentation

This feature can be configured to stations that have established an association with the AP, denominated TIM stations. However, an exception arises in the context of non-associated stations within the IEEE 802.11ah standard, which are identified as non-TIM stations. Non-TIM stations may establish communications within the network only when the AP authorizes them to do so by the transmission of an association grant message.

The TIM feature serves to inform TIM stations if they have frames stored in the AP buffer to be received during their RAW slot assignment. Therefore, TIM stations are informed about downlink data through beacon frames. Then, during their RAW slot they must wake-up in order to receive downlink data. Otherwise, they may resume sleep.

Furthermore, TIM segmentation shortens beacon length in dense networks, where a non-segmented TIM would be overly long. This not only saves energy but also allows connecting up to 8,191 stations due to the hierarchical grouping.

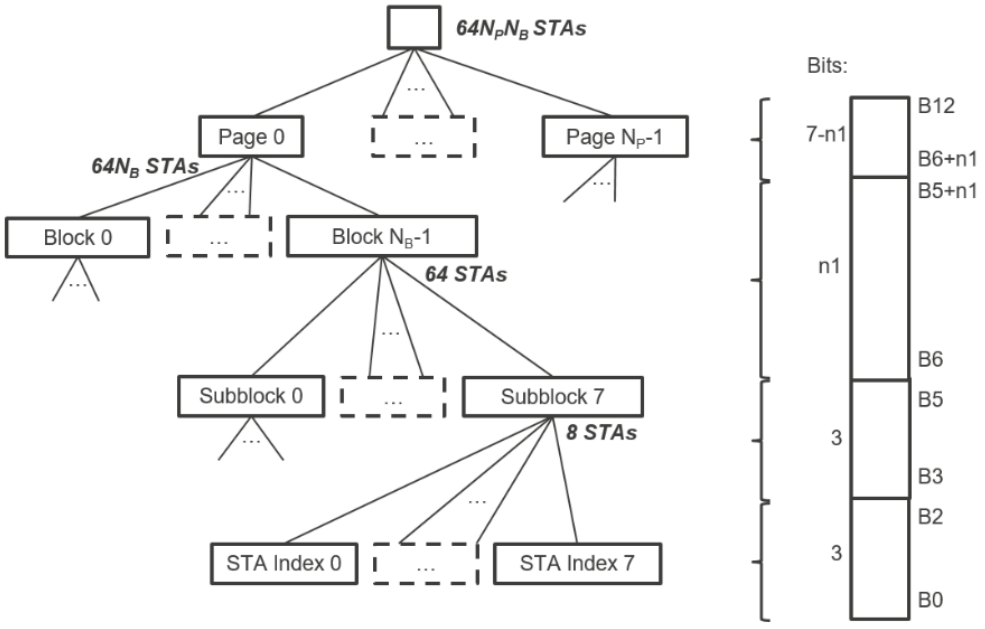
The TIM segmentation feature, as specified in the IEEE 802,11ah standard, allows stations to be organized in a three-level hierarchical structure, where each station is assigned a unique 13-bit Association Identification (AID). This hierarchy lets the AP indicate in a TIM bitmap whether stations have pending downlink data at multiple levels. Therefore, stations wake up only for their TIM group, conserving energy and reducing contention. The AID consists of the following fields, as illustrated on Fig. 2.7.

1. Page ID: The total number of TIM stations allocated in a page are divided into page slices, identified by the page ID number. Through this attribute the AP indicates only a portion of TIM stations within the rage of the page slice that might be served during the beacon intervals.

2. Block: This field further subdivides the page slice into blocks, indicating if the subset of stations are served during the beacon intervals. Each block may have up to 8 sub-blocks.

3. Sub-block: Each sub-block contains the index of up to eight stations.

Figure 2.7. TIM Segmentation



Source: (ŠLJIVO *et al.*, 2018).

2.3 NETWORK SIMULATOR 3 BACKGROUND

In this section, we provide a concise overview of the Network Simulator 3 (NS-3). This platform serves as the chosen network simulation environment for the development of our project. Specifically, the version of the simulator used in our project is the NS-3.23, as our implementation was built upon the IEEE 802.11ah simulation module presented in (TIAN *et al.*, 2016) and (TIAN *et al.*, 2018). Consequently, the subsequent paragraphs will refer to objects, models, classes and other data types compatible with this specific version.

NS-3 is a widely used open-source discrete-event network simulation framework. It is designed to model and simulate a vast range of communication networks and protocols, allowing researchers and developers to analyze and evaluate network behavior, performance, and characteristics in a controlled virtual environment. NS-3 is actively developed and maintained by a collaborative community of researchers, educators, and industry professionals.

The discrete nature of NS-3 refers to its modeling approach, where events in the simulated network may occur at specific points in time rather than in a continuous time flow. The points in time represent a series of discrete moments. At each of these moments, events can occur that may cause changes in the system being simulated. These events are processed one by one, as the simulation progresses in time steps.

The simulation process in NS-3 involves defining the network topology, setting parameters, configuring protocol behavior, and running simulations to collect data. NS-3 can output various metrics, such as packet loss rates, throughput, latency, and more, to help researchers assess the performance of their network designs and protocols.

Moreover, the NS-3 project is structured within a C++ namespace called "ns3". This namespace serves as an encapsulated environment for all NS-3 declarations, effectively segregating the NS-3 realm from the global C++ namespace. This design is deliberate, aiming to simplify integration with other codebases.

In summary, the following subsections describe how NS-3 simulation scripts function and the Waf tool employed for running simulations. Subsequently, we delve into a discussion of two crucial library models necessary for the development of our implementation in NS-3, namely the WiFi Model Library and the Energy Framework.

2.3.1 Simulation Scripts

In the realm of NS-3, simulation scripts, also denoted configuration scripts, typically encompass programs written either in C++ or Python. These scripts play a crucial role in streamlining the network configuration process, running simulations, and providing output. Hence, simulation scripts are typically organized into three essential phases:

1. **Configuration:** The configuration phase commences with the first line of code within the script and extends until the point where the command to initiate the simulation is encountered.
2. **Simulation:** The segment of code spanning from the initiation of the simulation to its conclusion is known as the simulation part. Throughout this phase, methods, functions, and other data structures can be tailored to collect specific information about the simulation.
3. **Destruction:** The destruction phase refers to the cleanup section, which is essential for releasing the computer memory resources utilized during the simulations.

Additionally, despite the fact that the Python programming language could also be used to craft simulation scripts in NS-3, this approach falls beyond the scope of our project.

2.3.2 Waf Configuration and Build System

Waf is the program used in various activities related to NS-3.23, such as installing NS-3 itself, configuring network simulation scenarios, testing the library models and so on.

The following paragraph outlines the results obtained from running the help function of Waf, as discussed in the manual presented in Appendix A, as follows:

```
  ~ ./waf --help
waf [commands] [options]

Main commands (example: ./waf build -j4)
build    : executes the build
check    : run the equivalent of the old ns-3 unit tests using test.py
clean    : cleans the project
configure: configures the project
```

```
dist     : makes a tarball for redistributing the sources
distcheck: checks if the project compiles (tarball from 'dist')
docs     : build all the documentation: doxygen, manual, tutorial, models
doxygen  : do a full build, generate the introspected doxygen
install  : installs the targets on the system
list     : lists the targets to execute
shell    : run a shell with an environment suitably modified
sphinx   : build the Sphinx documentation: manual, tutorial, models
step     : executes tasks in a step-by-step fashion, for debugging
uninstall: removes the targets installed

...
```

2.3.3 WiFi Model Library

The WiFi model library of NS-3 comprises a collection of objects that simulate the attributes and behavior of WiFi interface cards, closely resembling the flexible architecture found in real computer systems. It also works as a completely independent object of the other model libraries. Some of its major objects related to our research are described as follows.

2.3.3.1 WifiNetDevice

The `WifiNetDevice` object models the wireless network interface controller, which conforms to the specifications of the IEEE 802.11ah, IEEE 802.11a, IEEE 802.11b, IEEE 802.11g and IEEE 802.11n standards. It also comprises four distinct levels of models, which support the following features:

1. MAC high models:
    - Beacon generation, probing, and association state machines.
2. MAC low models:
    - Distributed Coordination Function (DCF) for IEEE 802.11 infrastructure and adhoc modes.

- Support for MAC Service Data Unit aggregation (MSDU) and MAC Protocol Data Unit aggregation (MPDU).
- Quality of Service (QoS) based Enhanced Distributed Channel Access (EDCA) and queueing extensions.

3. Rate control algorithms:
   - Including : Aarf, Arf, Cara, Onoe, Rraa, ConstantRate, and Minstrel.
4. PHY layer models.
   - Support for the IEEE 802.11ah, IEEE 802.11a, IEEE 802.11b, IEEE 802.11g, and IEEE 802.11n standards.
   - Deployment of various propagation loss models and propagation delay models.
   - Operation on 900 MHz, 2.4 GHz and 5 GHz frequency bands

These features provide a comprehensive toolkit for configuring and simulating IEEE 802.11 wireless networks across various network scenarios. Consequently, in the following sub-subsection, we will explore some of the most crucial models and classes for the development of our project.

2.3.3.2 WiFi Channel Class

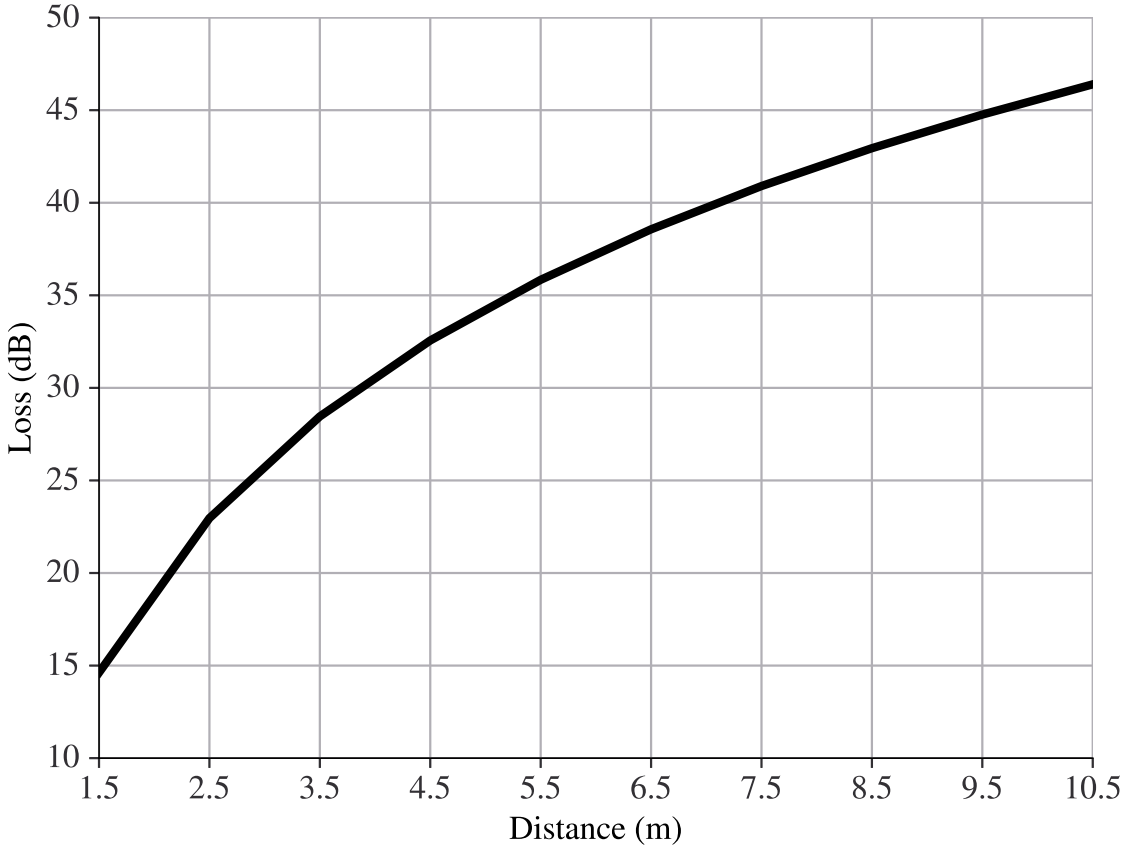
The WiFi Model Library features the `YansWifiChannel` class as the modelling object of the WiFi channel. This class provides a variety of methods for configuring propagation loss and propagation delay models, managing connections, and facilitating the exchange of frames between nodes through a wireless channel.

In the context of IEEE 802.11ah networks, simulations can be configured to mimic the propagation loss models described in (HAZMI *et al.*, 2012). These models encompass the macro deployment model and the pico or hotzone model tailored for outdoor environments, as defined by Equation (2.3). Additionally, they incorporate the free space loss model designed for indoor environments, as defined by Eq. (2.4).

To illustrate the path loss response obtained from the macro deployment model in the single-link scenario presented in Chapter 6, a graph is provided in Fig. 2.8. This graph displays

path loss values in dB, as a function of the input distance represented on the x-axis.

Figure 2.8. Macro Deployment Path Loss Model



Source: Own authorship

2.3.3.3 WiFi PHY Class

The WiFi PHY Class, namely `YansWifiPhy`, represents a PHY model that encompasses the standards-specific operational aspects of many IEEE 802.11 standards.

Nonetheless, it is important to note that these operations are limited to functions provided by the upper portion of the PHY, specifically the PLCP sub-layer, since the PMD sub-layer is not implemented in NS-3. Therefore, the PHY in NS-3 operates only in a per-packet basis, as described in the work of Lacage & Henderson (2006), where the acronym YANS (Yet Another Network Simulator) was first coined, and further elucidated in the study carried out by Fuxjaeger & Ruehrup (2015). Therefore, NS-3 does not model the processes of encoding, modulation and

transmission of RF signals into wireless channels.

Within the `YansWifiPhy` class, its responsibilities include both the transmission and reception of PPDUs. Regarding the reception of PPDUs, each received PPDU undergoes a probabilistic evaluation to determine its successful reception, factoring in parameters, such as RSSI and SNR.

2.3.3.4 The PHY State Machine

The `WifiPhyStateHelper` is a helper class that the `YansWifiPhy` class uses for controlling the PHY state machine. The PHY state machine is characterized by six distinct states that collectively define the operational behavior of the PHY within the IEEE 802.11 standards:

- **TX (Transmitting):** The PHY is actively transmitting a frame on behalf of its associated MAC.
- **RX (Receiving):** The PHY is synchronized with a frame and is awaiting the reception of its final bit before forwarding it to the MAC.
- **IDLE:** The PHY is in an idle state, waiting for one of the following states to occur: TX, RX, or CCA_Busy.
- **CCA_Busy (Clear Channel Assessment Busy):** The PHY is engaged in a clear channel assessment (CCA) process. During this state, it is neither in TX nor RX mode, but the measured energy levels at the receiver input are higher than the energy detection threshold.
- **SLEEP:** The PHY is in a power-saving mode and is incapable of sending or receiving frames.
- **SWITCHING:** The PHY is in the midst of a channel-switching process, transitioning from the previously used RF channel to another.

2.3.3.5 The PHY Frame Reception Operation

The following procedure outlines the principles that govern the operation of the PHY upon the reception of the first bit of an incoming frame, hereafter referred to as '$frame_k$':

1. If the PHY is found operating in the TX, SLEEP or SWITCHING states, $frame_k$ is dropped.

2. However, if the PHY is in either the IDLE or CCA_Busy states, it then evaluates the power of the first bit of $frame_k$. The power level of $frame_k$ is denoted as $S_k(t)$, where t represents the time interval for the reception of $frame_k$. $S_k(t)$ is assumed to be zero outside the reception interval t and remains constant during this interval. The evaluation involves comparing $S_k(t)$ with a predefined threshold, referred to as the Energy Detection (ED) threshold, and this comparison is expressed as follows:

   (a) If $S_k(t)$ exceeds the ED threshold, the PHY switches to the RX state. Additionally, it schedules an event for the reception of the last bit of $frame_k$, in order to evaluate the probability that $frame_k$ has been received with errors, denoted as $P_{\text{error}}(k)$. This transition to the RX state indicates the readiness of the PHY to receive and decode the remaining parts of $frame_k$.

   (b) However, if $S_k(t)$ falls below the ED threshold, the PHY remains either in the IDLE or CCA_Busy states and discards $frame_k$.

3. Once the PHY has received its last bit of $frame_k$, it must evaluate if $frame_k$ has been received successfully or not. To make this assessment, the PHY generates a pseudo-random number from a uniform distribution random function and compares if it is greater than $P_{\text{error}}(k)$.

   (a) To calculate $P_{\text{error}}(k)$, NS-3 implements the concept of additive interference power accumulation, as validated in (PEI; HENDERSON, 2010) and (MILLER, 2003), which treats the power levels of any other frames received on the same channel, that may overlap in time with the reception interval t of $frame_k$, as additional thermal noise. The sum of the power levels of all other overlapping frames received on the same channel is represented by $N_i(k,t)$, which is also considered as the interference noise and is given by

$$N_i(k,t) = \sum_{m \neq k} S(m,t), \tag{2.9}$$

where, $S(m,t)$ is the power level of the m-th overlapping frame received on the same channel during the t reception interval of $frame_k$.

(b) Then a piecewise linear function, illustrated in Fig. 2.9, and detailed shortly after its presentation, is employed to calculate the Signal-to-Noise-plus-Interference Ratio over time, SNIR(k,t), for $frame_k$ during its t reception interval, as follows

$$\text{SNIR}(k,t) = \frac{S_k(t)}{N_i(k,t) + N_f}, \tag{2.10}$$

where N_f is the noise floor corresponding to the thermal noise and non-idealities of the receiver, which is given by

$$N_f = k290B_W, \tag{2.11}$$

where k is the Boltzmann constant corresponding to 1.380649×10^{-23} (jK^{-1}); and B_W is the channel bandwidth in Hz.

(c) Then, NS-3 uses the `ErrorRateModel` to obtain the Bit Error Rate (BER), $P_{\text{BER}}(l)$, from SNIR(k,t), using Eq.(2.12) for BPSK modulation and Eqs. (2.13, 2.14) for QAM modulation, as described in the work of Lacage & Henderson (2006).

$$\text{BER}(k,t) = \frac{1}{2} erfc\left(\sqrt{\frac{E_b}{N_o}(k,t)}\right), \tag{2.12}$$

$$\text{BER}(k,t)) = 1 - (1 - P_{\sqrt{M}}(k,t))^2, \tag{2.13}$$

$$P_{\sqrt{M}}(k,t) = \left(1 - \frac{1}{\sqrt{M}}\right) erfc\left(\sqrt{\frac{1.5}{M-1} log_2 M \frac{E_b}{N_o}(k,t)}\right), \tag{2.14}$$

(d) Following that, in the parts of interfering frames that may overlap in time with $frame_k$, defined as chunks, are used to calculate the chunk success rate, denoted as $P_{csr}(l)$, given as follows

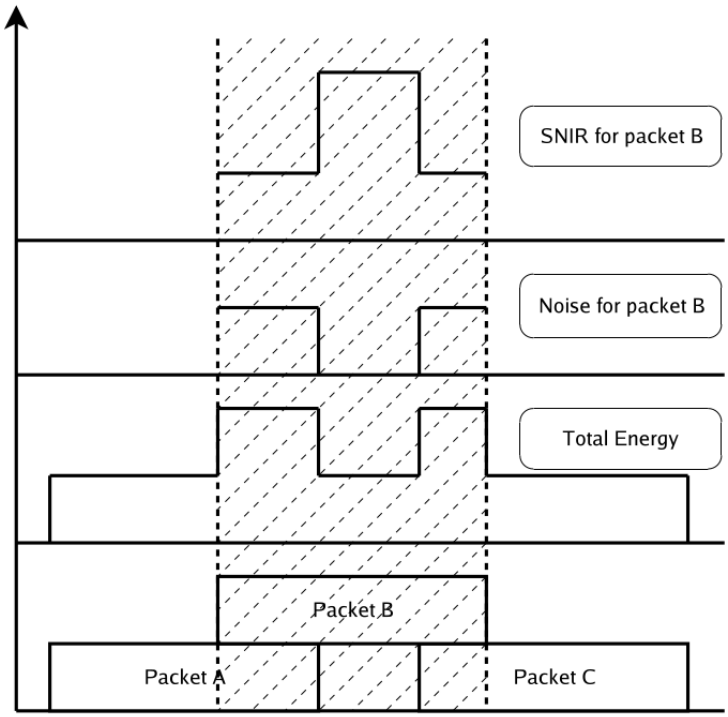
$$P_{csr}(l) = (1 - P_{\text{BER}}(l))^n, \tag{2.15}$$

where l is the length of n bits defined as a chunk.

(e) Finally, the probability that $frame_k$ has been received with errors, $P_{\text{error}}(k)$, is given by

$$P_{\text{error}}(k) = 1 - P_{csr}(l), \tag{2.16}$$

Figure 2.9. SNIR Function over Time



Source: (LACAGE; HENDERSON, 2006).

A brief additional clarification is needed for the illustration depicted in Figure 2.9. The central focus of this illustration is on determining the SNIR during the reception of packet B. As mentioned earlier, this emphasis is due to the assumption that power levels outside the time interval allocated to the reception of packet B, highlighted by the diagonal parallel dashed lines, are considered as zero. Consequently, the illustration concentrates solely on the SNIR calculations within the specific time interval designated for the reception of packet B.

In the depicted scenario, packets A and C overlap in time during the dedicated interval for the reception of packet B. To elaborate further, upon analyzing the illustration from bottom to top, the lower section of the graph reveals that packet A overlaps in time with the beginning of the reception of packet B. And packet C is observed to overlap towards the end of packet B.

In the second graph from bottom to top, the representation illustrates the total energy levels

accumulation over time. Notably, in the chunk where packet A overlaps in time with packet B, the energy level is higher compared to the subsequent section where the energy is solely attributed to packet B. This pattern is repeated in the chunk where packet C overlaps in time with packet B. The graph visually captures how the overlapping chunks contribute to variations in the total energy levels.

Continuing the explanation, the third graph from bottom to top illustrates the interference noise levels over time. It is notable that the segments representing the noise for packet B are specifically depicted only in the regions where the overlaps with packets A and C occur. The calculation of interference noise levels can be accomplished by applying Eq. (2.9).

Finally, the graph at the top of the illustration displays the SNIR function over time. Here, a step response is evident, with the peak centered during the interval without overlaps, and lower values can be observed in the chunks where overlaps with packets A and C occur. This aligns with the piecewise linear behavior as defined in Eq. (2.10).

2.3.4 The Energy Framework

The energy framework of NS-3 offers support for simulating energy sources, energy consumption of devices and energy harvesting, which are represented by three major objects: Energy Source objects, Device Energy Model objects, and Energy Harvester objects.

2.3.4.1 Energy Source

The Energy Source object represents the power supply of nodes. The primary function of the Energy Source is to supply energy to nodes. Each node can have one or more energy sources, and each energy source can be linked to multiple device energy models. Connecting an energy source to a device energy model implies that the associated device consumes power from that source.

2.3.4.2 Device Energy Model

The Device Energy Model object represents the energy consumption model of each node. It follows the PHY state machine model, where each device is assumed to operate in multiple states, and each state is associated with a specific parameter of electric DC current drawn. Whenever the state of the device changes, the corresponding Device Energy Model notifies the Energy Source of the new current drawn. The Energy Source then calculates the energy consumed at that particular state and adjusts the remaining energy accordingly.

2.3.4.3 Energy Harvester

The energy harvester object represents the component responsible for harvesting energy from the environment and replenishing the energy levels of the connected Energy Source. They include the complete modelling of energy harvesting devices and the corresponding attributes, such as solar panels, whose environmental attributes include solar radiation. This means that when implementing an energy harvester, it is essential to model both the energy contribution from the environment and the additional features of the energy harvesting device, as part of the collective modeling process.

2.4 SWIPT IMPLEMENTATION

In this section we provide a concise overview of our implementation over the Extension of the IEEE 802.11ah NS-3 Simulation Module presented in (TIAN *et al.*, 2018).

The entire codebase of our implementation was developed using C++ programming language according to the NS-3 namespace and following the concept of the extended energy framework delineated in references (TAPPARELLO *et al.*, 2014a) and (TAPPARELLO *et al.*, 2014b). For this reason, our implementation is compatible with various WiFi standards in NS-3.23, such as the IEEE 802.11a/b/g/n/ah standards. Furthermore, it facilitates interoperability with existing and forthcoming energy source model releases built upon the same framework, ensuring flexibility and compatibility with a wide range of configurations.

This achievement was made possible through the usage of a technique referred to as "Scaf-

folding," as detailed in (NS-3 Project, 2015a). This methodology consists of creating first a skeleton of the intended classes, in order to ensure compatibility within the existing models of the NS-3 library. And then, fill in the details of the desired features using member-functions, data types and others, as the design matures. By doing so it helped us to reduce dependencies between classes, enhance the reuse of our code and broaden the range of functionalities.

Hence, to complete the skeleton with the desired features, we conducted a comprehensive exploration of the intricacies surrounding the classes responsible for managing the transmission and the reception of frames within the WiFi model library. We meticulously dissected the underlying mechanisms of the operations of essential classes to discern how they work and how the energy consumption manifested at every juncture.

The driving force behind this investigation was originated from the need to gain a comprehensive understanding of the foundational mechanisms that support the operations of the WiFi PHY model and the Energy Framework, which further allowed us to acquire the necessary expertise for the development of our model.

Moreover, our implementation is based on the concept of the SWIPT receiver with a split architecture deploying the PS technique, with some additional features:

- A member function was introduced to model the connection between the antenna and the SWIPT harvester
- A listener object was introduced to replicate the mechanism of synchronization carried out by the PMU.
- A set of class attributes were introduced to represent various parameters, including θ_s, the combined efficiency of the matching circuit with the AC-DC rectifier, and the efficiency of DC-DC conversion.

Additionally, to facilitate the configuration of simulation scripts, we developed an SWIPT helper class. This class automates the process of configuring the SWIPT harvester object in the simulation script, establishing the necessary connections to the energy source, and offering an user-friendly interface for parameter customization.

Finally, the outcome of this research yielded the creation of the blueprint presented from various perspectives in the next Chapter, in which we aim to deliver a comprehensive understanding

of our implementation.

2.5 SUMMARY

In this Chapter we covered the theoretical foundations that constitute the essential framework of our research.

We started by covering the theory related to SWIPT, describing the major techniques used for simultaneously extraction of energy from incoming electromagnetic signals, namely the TS and PS techniques.

Then, we provided an overview of the IEEE 802.11ah standard, focusing on the main attributes related to our research, such as the S1G PHY, the RAW and TIM mechanisms.

After that, we covered some theoretical aspects related to NS-3, describing the major components of configuration scripts, the Waf system, and two major libraries that were crucial for our implementation: the WiFi model and the Energy framework.

Following that we introduce some information about our SWIPT implementation, describing the concepts we used as references, the adopted scaffolding technique, the some additional features that will be described in detail in the next Chapter.

CHAPTER 3

RELATED WORK

In this chapter, we present the outcomes of an extensive survey conducted to evaluate the current state of scientific knowledge regarding the simulation of SWIPT within standardized network architectures using computational platforms. Despite the significant attention SWIPT has received within the research community, it is crucial to note that the emphasis has predominantly been on the hardware aspects of the PHY layer, as evident in references such as (JIANG; HUANG, 2019), (JU; ZHANG, 2014), and (SHARMA *et al.*, 2022). This focus has resulted in limited exploration of SWIPT simulations within standardized network architectures. To address this hiatus and establish a common foundation for comparison with our work, we offer a comprehensive overview of research efforts that may approximate to our work by integrating various types of energy harvesting techniques. These efforts encompass both simulation-based studies and practical prototypes, providing valuable insights into common features used in our simulations. The following paragraphs commence with the presentation of a study investigating a practical prototype, followed by the description of a study involving the deployment of an energy harvester in a practical network setup. Subsequently, the remaining of the presented research focuses on implementations of energy harvesters into simulation platforms. To conclude this Chapter, we provide a table summarizing the distinctive features of each study, facilitating a convenient comparison with our own implementation.

In the study conducted by Taris *et al.* (2012), the authors introduce a prototype that uses the Wireless Energy Transfer (WET) technology, operating in the 900 MHz frequency band, assembled using commercially available off-the-shelf components. This prototype incorporates basic elements, including an antenna, a matching network, a rectifier, and an energy storage component. The study presents measured results of rectified voltage levels obtained within an indoor environment, utilizing an RF source with a transmitting power level of 16.8 dBm. The distance from the prototype to the RF source varied between 0.3 to 3 meters. These

measured results were subsequently compared to the results obtained from simulations and theoretical calculations. To illustrate the decline in the curve of the reported results obtained from measurements, at a distance of 0.3 meters, the measured rectified output voltage reached 2.1 V, while at 0.5 meters, it reached 1.25 V. Additionally, at a distance of 1.5 meters from the source, the measured rectified output voltage reached 500 mV. These findings were also presented as a comparison with the reported results of similar studies. Hence, from the information presented in this paper, it is essential to note some distinctions when establishing a comparison to our work. Firstly, it is important to note that the solution presented in this paper primarily centers on the hardware. In contrast, our implementation is centered on purpose of network simulations. Secondly, the type of energy harvester investigated in this work, WET, differs from the SWIPT adopted in our implementation. Lastly, the paper does not provide information about investigating their proposed solution within a network scenario, as our work does. However, it is noteworthy that, despite the dissimilarities, their solution employs approximate values for the same parameters we used in our simulations, such as the frequency of operation and the distance range, yielding the results displayed in Chapter 6.

In the study conducted by Lee *et al.* (2016), an investigation into the deployment of SWIPT over a practical IEEE 802.11 network setup operating in the 2.4 GHz frequency band is presented. The proposed SWIPT model in this study introduces a novel approach for allocating the resources of a received frame between the inputs of the energy harvester and the information decoder based on the destination address of the frame. Specifically, they introduced an independent scheme to decode the header of incoming frames. This process involves assessing whether the destination address field matches the MAC address of the node. When the verification succeeds, the frame is entirely directed to the input of the information decoder. On the contrary, if the destination address differs from the MAC address of the node, the entire frame is allocated to the energy harvester. Additionally, when the frame is allocated to the energy harvester, their model conducts a second assessment before initiating the energy harvesting process. This involves verifying whether the RSSI of frames meet a specified ED threshold. This threshold is configured to only allow frames with RSSI high enough to be routed to the energy harvester, in order to ensure that the harvested energy levels surpass the energy expenditure of the node, thereby promoting the energetic sustainability of the model. Furthermore, the paper reports on practical experiments conducted using commercially available devices, where the

trade-off between network throughput and harvested energy is assessed as a function of the number of nodes within the network. Therefore, we can conclude that this paper highlights some characteristics of their proposed model that diverge and converge with our work. As an example of a divergent aspect, the strategy used for determining whether frames are directed to the information decoder or to the energy harvester seems to deviate from the concept of the simultaneous execution of information decoding and energy harvesting from both Time Switching (TS) and Power Splitting (PS) techniques of SWIPT, as described in Chapter 2. This deviation is attributed to the fact that the PHY machine state may operate in some states other than RX when the energy of a frame is being harvested. Nonetheless, this strategy also displays an opportunistic approach: if a frame is going to be dropped due to not being addressed to the node while the wireless channel remains occupied for the rest of the duration of the frame, it might as well be used for energy harvesting purposes. This study aligns with our work concerning the attribute of energetic sustainability of the proposed model. Although not fully elaborated in this paper, we also consider sustainability as a metric for the results obtained from our simulations, as presented in Chapter 6.

In the work of (ALSADER; SAVVARIS, 2017), a solution that combines hardware simulation of a piezoelectric energy harvester with the concurrent simulation of a Wireless Sensor Network (WSN) is presented. They achieve this by using the software LTSpice to model and simulate the electronic circuit of the piezoelectric energy harvester, while NS-3 is used to simulate the WSN. To interconnect these two platforms, the authors employ Linux containers, with each software running on its respective operating system. Additionally, the study utilizes the Transmission Control Protocol (TCP) to transmit information about the outputs of the piezoelectric energy harvester to the WSN. Furthermore, their implementation in NS-3 incorporates a fuzzy-based PMU for the piezoelectric energy harvester. The study reports the following results after 10 minutes of simulation: minimum, average, and maximum values of the energy harvested are 3.236360x10^{-11}J, 4.74617x10^{-5}J, and 1.000928x10^{-4}J, respectively. They also report an efficiency level of 85% energy expenditure by the WSN from the total harvested energy level. However, it is crucial to note that a comparison between this paper and our implementation might reveal some dissimilarities. These differences start with the type of energy harvester used and its respective modeling and simulation, as our approach does not model the piezoelectric type nor involves its hardware simulation. Nevertheless, a notable similarity between this study

and our work can be identified through the metric denoted as energy utilization efficiency (EUE), which signifies the ratio of the total energy expenditure of the WSN to the total harvested energy. Moreover, while each study takes a unique approach in certain aspects, the shared foundation within the energy framework underscores the diversity in modeling approaches NS-3 can provide.

In reference to (BENIGNO *et al.*, 2015), the authors proposed the implementation of an extension involving the utilization of solar power through photovoltaic panels within the energy framework of NS-3. This study integrates geographic coordinates to enhance the accuracy of device representation, considering panel dimensions and mechanical tilt in the modeling process. The results displayed from simulations are presented across various time scales and aim to provide a comprehensive performance analysis. However, this study does not provide information regarding which type of network was configured for running the simulations, since their major focus is on accurately modeling the realistic behaviour of solar energy harvesting systems. Furthermore, there are clear differences between their approach and ours, specifically in the type of energy harvester chosen. Despite these distinctions, when it comes to the implementation methodology within NS-3, there are shared aspects between their approach and ours. Both studies are conducted within the NS-3 energy framework, with subtle variations. Notably, our work builds upon this foundation by incorporating the WiFi model. To elaborate, while their research introduces a novel class, `SunEnergyHarvester`, for modeling solar energy harvesting, our study implements an energy harvester named `SwiptHarvester` within the NS-3 energy framework. The difference in the names assigned to the energy harvesters underscores the versatility and flexibility inherent in implementing various types of energy harvester objects within a shared foundational structure. This illustrates the diverse approaches that can be employed to achieve similar goals while adapting to specific requirements or preferences in the naming conventions of the components within NS-3.

In (ZHONG *et al.*, 2017), the authors present a solution involving a mobile charger node for a multi-hop sensor network implemented in NS-3, with the mobile charger node dynamically adjusting its position to reach nodes with low battery levels and recharge them. In their proposal, the operational management of the charger involves a charging request queue to record requests from nodes. The priority for recharging is determined by considering factors such as residual

energy level, energy consumption rate, and the distance between the mobile charger and nodes. It is worth noting that this paper differs from our work in some aspects. Particularly, the energy harvesting technology employed in their approach, WET, differs from what we have implemented. Despite these differences, both our work and theirs share a common foundation within the energy framework of NS-3. In their implementation, a new class, `WET EnergyHarvester`, is introduced, while our work implements the `SwiptHarvester` using the same foundational principles. This signifies that, despite variations in specific methodologies, both studies leverage the established energy framework of NS-3 for their respective implementations.

In the research conducted by Rehman *et al.* (2018), they delved into the realm of SWIPT using the LabVIEW software. Their experimental setup involved a network architecture consisting of a single transmitter and a receiver. The transmitter, operating at a radio frequency band of 1 GHz, employed BPSK modulation. The receiver, comprising an information decoder and an energy harvester, played a crucial role in processing the transmitted signals. The experimental conditions included a transmitter output power of 0 dBm, a θ_s value of 0.8, a rectifier threshold set at 0.01V, and an efficiency level of 0.5. The reported output results fell within the range of 0.071V to 0.075V. However, the paper does not provide additional insights into how the model performs within the broader context of existing network standards. This absence of information poses a challenge when attempting to assess the real-world applicability of their findings, as it leaves uncertainties regarding the scalability and compatibility of their approach beyond the specific LabVIEW-based setup presented in the study.

In the work of Xu *et al.* (2019) the usage of SPICE as a platform for simulating the electronic circuit of an energy harvester is proposed. They also mention the assembly of a prototype device designed to validate the results obtained from simulation through experimentation. The proposed energy harvester model comprises a discrete-component matching network (MN), a cross-coupled rectifier, and a PMU. The intended network architecture setup involves smart sensors operating at 2.45 GHz. The reported results demonstrate a peak power harvester efficiency (PHE) of 48.35%, utilizing an RF input signal power of -3 dBm. Additionally, the paper includes a comparison of the performance of their proposed model with six other similar projects. It is worth noting a few differences between this paper and our work. Firstly, their focus is primarily on PHY aspects of wireless energy harvesting. Secondly, the type of energy

harvester used in their approach is different from what we have implemented. Lastly, similar to previous studies, this work does not provide information about simulation results within existing network standards, which would allow for the evaluation of the practical applicability of their findings beyond the specific laboratory environment presented in the study.

In (LEE *et al.*, 2018), the authors propose a novel MAC protocol called W^{2}P-MAC to address the issue of co-channel interference resulting from Power Beacons (PBs) used for recharging node batteries in IEEE 802.11 networks. This scheme is designed to ensure that when WiFi nodes transmit frames to the Access Point (AP), their energy levels are reported alongside. The AP then serves as a central hub for gathering energy level information, calculating the average energy level, and relaying this information to the PBs. The decision for a PB to become active is based on notifications received from the AP regarding the average energy value. When a PB becomes active, it employs the Clear Channel Assessment (CCA) mechanism and a random back-off counter before transmitting Energy Request to Send (ERTS) messages across the network. Once an ERTS message is sent, the PB awaits an Energy Clear to Send (ECTS) reply before commencing the energy transfer to nodes. When the energy transfer concludes, the AP broadcasts an Energy Acknowledgment (EACK) to the network. It is worth highlighting the distinctions between this paper and our proposed work. Specifically, their focus lies in proposing a MAC protocol for the coexistence between PBs and WiFi nodes. In contrast, our SWIPT harvester module is implemented in the PLCP sub-layer. Despite these differences, it is noteworthy that both studies share a common goal of implementing solutions for WiFi networks, aligning with the broader objectives of our work.

In the work of Zhao *et al.* (2018) an extended version of the CSMA/CA protocol for WET within a Wireless Local Area Network (WLAN) context is proposed. Their model introduces the concept of allowing frames sent from the AP to an individual node to also be used for the purpose of energy harvesting by other nodes in the network. To achieve this, a node is required to check if the back-off counter has reached zero and if the voltage level of a supercapacitor surpasses a specified threshold before initiating transmission. If the second condition is not met, the node suspends its back-off timer and commences an energy harvesting procedure to extract energy until the voltage level of the supercapacitor exceeds a certain voltage threshold. The reported results indicate that the throughput values in the downlink and uplink directions

tend to converge to the same level, as the number of nodes in the network increases. Therefore, it is worth noting that this study differs from our work in certain aspects. Specifically, the type of energy harvester employed and the method for extracting energy from frames differ from our approach. Nevertheless, there are similarities between this study and our work. Our implementation also shares the capability of extracting energy from frames addressed to other nodes, given that the frames meet the operational conditions of the PHY or, in the last case, can be interpreted as interfering frames. These functionalities are detailed in Chapter 5.

In the study of Iqbal & Lee (2021), the authors introduce a novel MAC protocol based on frame-slotted Advocates of Linux Open-source Hawaii Association (ALOHA) to enhance resource allocation efficiency in spatio-temporal node diversity within wireless powered IoT networks. In their scheme, IoT nodes compete for medium access and randomly select transmitting slots within a frame. Any unused slots in a frame can be utilized for energy harvesting. Nodes may also harvest varying amounts of energy due to their spatial diversity relative to transmitted PBs and the differing quantities of harvesting slots available to them. Additionally, nodes can use the randomly selected slots to transmit data. The reported results indicate improvements in throughput of up to 12.5% and energy efficiency gains of up to 20% compared to the traditional ALOHA scheme. This paper takes a different approach from our proposed implementation. The main difference is that it does not align with the SWIPT paradigm we focused on. It also introduces additional infrastructure requirements for using PBs and employs a distinct MAC protocol, frame-slotted ALOHA, for medium access contention.

In the study conducted by Khairy *et al.* (2019), the authors present a solution for IoT data communications integrated with WET using WiFi technology. Their solution is centered on the observation that the highest energy consumption of a node can be attributed to frame collisions that may occur in random channel access procedures, leading to frame retransmissions. In response, they developed an analytical model to characterize the energy consumption and the RF energy harvesting of IoT devices in a CSMA/CA context. Through the analysis of the analytical model, they reported being able to find the necessary and sufficient conditions for the frequency of beacons broadcast from the AP and the charging period of IoT devices that allow a network with a random topology to reach long-term energy sustainability. Subsequently, they proposed a distributed Energy-Sustainable, Throughput-Optimal (ESTO) algorithm to

select the charging period of IoT devices and maximize the total network throughput. They also reported that the performance evaluation carried out using NS-3 was able to demonstrate the achievement of the expected results. Therefore, this study exhibits both contrasts and similarities with our work. The contrasts primarily revolve around the type of energy harvester used, the PHY state when energy harvesting occurs, the specific algorithms they implemented to reach the optimal conditions, as well as the analytical model they developed. However, the similarity lies in the demonstration of the energy sustainability of their solution, which is also a metric we used for presenting the results of our implementation in Chapter 6.

In the research conducted by Sousa *et al.* (2017), the authors introduce a wireless power transfer (WPT) simulator based on resonant coupling transfer. This simulator can be used either as a standalone C++ package or as an NS-3 module. The motivation behind their work stems from a perceived gap between WPT and wireless network simulations. To bridge this gap, the authors propose an analytical model based on the circuit analysis of a resistor-capacitor-inductor (RLC) circuit employing *Kirchhoff Laws* to represent WPT operations. Then, MATLAB was used to provide the calculations for the proposed analytical model and for assessing the output. Furthermore, the authors implemented the WPT model consisting of two key classes: `GlobalCoupler` and `Resonator`. The simulation results were then compared with existing works to evaluate the efficacy of their solution. Thus, we can conclude that this study also presents both contrasts and similarities with our work. The primary contrast regards the type of energy harvester employed and the second the development of an analytical model. Notably, their solution offers one key advantage of functioning as a standalone C++ package. Conversely, the similarity lies in the optional deployment on NS-3.

3.1 SUMMARY

Table 3.1 summarizes the information presented at this Chapter

Table 3.1. Related Work Summary

Reference	EH Type	Radio Frequency	Network		Simulation Platform	Sustain. Displayed
			Type	Standard		
(TARIS *et al.*, 2012)	WET	902 MHz	-	-	-	-
(LEE *et al.*, 2016)	SWIPT	2.4 GHz	WiFi	802.11b/g/n	-	✓
(ALSADER; SAVVARIS, 2017)	Piezo	-	WSN	-	Spice, NS-3	✓
(BENIGNO *et al.*, 2015)	Solar	-	M2M	-	NS-3	-
(ZHONG *et al.*, 2017)	WET	-	WSN	-	NS-3	-
(REHMAN *et al.*, 2018)	SWIPT	1 GHz	-	-	LabVIEW	-
(XU *et al.*, 2019)	SWIPT	2.45 GHz	WiFi	802.11b/g/n	Spice	-
(LEE *et al.*, 2018)	WET	-	WiFi	-	-	-
(ZHAO *et al.*, 2018)	WET	2.4 GHz	WiFi	-	Monte Carlo	-
(IQBAL; LEE, 2021)	WET	-	IoT	-	-	-
(KHAIRY *et al.*, 2019)	WET	-	WiFi	-	NS-3	✓
(SOUSA *et al.*, 2017)	Resonant Coupling	4 MHz	WSN, IoT	-	NS-3, C++	-
Our work	SWIPT	900 MHz	WiFi	802.11a/b/g/n/ah	NS-3	✓

Source: Own authorship.

CHAPTER 4

THE DESIGN OF THE SWIPT MODULE

This chapter outlines the design of the SWIPT module implemented into NS-3. The conceptual framework of this design closely aligns with the split architecture of the PS SWIPT receiver, as expounded in Chapter 2.

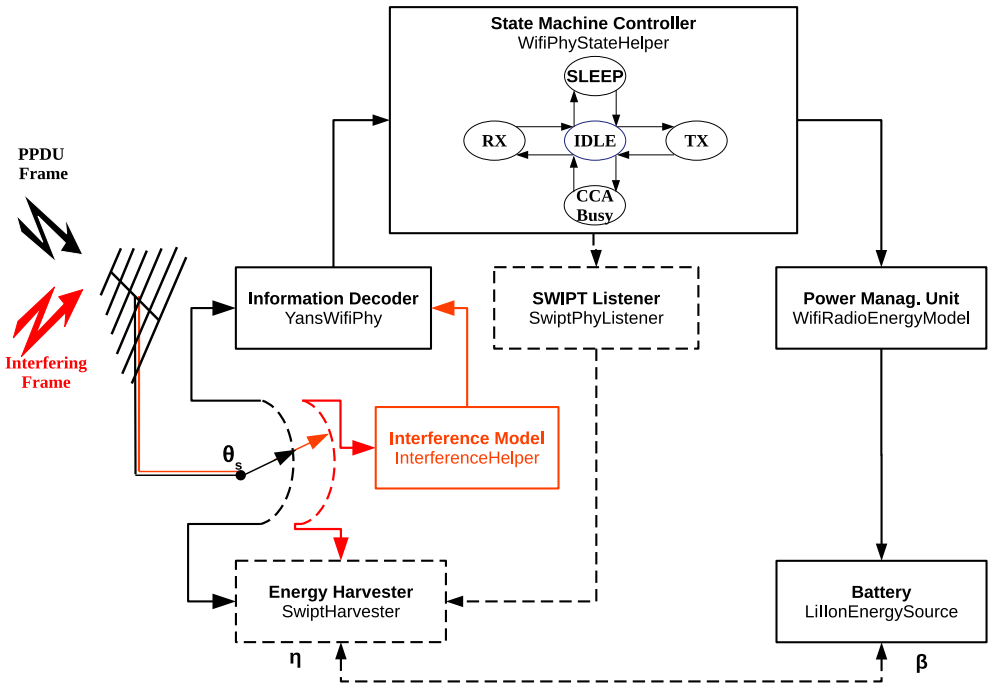
Therefore, in the following sections, we initiate our discussion by presenting the design of our implementation under two complementary perspectives. The first perspective is illustrated by the block diagram presented in Fig. 4.1, which is intended to provide a comprehensive overview of how the components in our design function as a unified system. Meanwhile, the second perspective is depicted in Fig. 4.2 through the integration of our design within the architecture of the `WifiNetDevice`, in order to provide a clear understanding of how the concept of our SWIPT module is intended to fit into the models of the NS-3 library.

Before we proceed, it is important to reiterate that in NS-3, the IEEE 802.11 PHY layer operates on a per-frame basis, as previously explained in Chapter 2. Hence, in the following paragraphs, we consistently refer to the protocol data unit of the PHY as PPDU or 'frames', rather than utilizing terms such as 'bits' or 'Radio Frequency (RF) signals', since NS-3 does not model the PMD sub-layer.

4.1 THE SWIPT BLOCK DIAGRAM

This section provides a perspective of the design of our implementation, offering a comprehensive explanation of the block diagram depicted in Fig. 4.1. This diagram illustrates the split architecture of the SWIPT receiver using the PS technique, as discussed in Chapter 2.

Figure 4.1. The SWIPT Receiver Block Diagram



Source: Own authorship.

Our objective is to present the concept of the design of our implementation as a dynamic system. This will be achieved by elucidating the operations of each component in terms of its inputs, processing, and outputs. In doing so, we will trace the journey of a PPDU through various paths and blocks depicted in the diagram and also explain the effects interfering frames might have on the system. We will begin with the PPDU reception, guide you through intermediate stages where relevant parameters are exchanged between blocks, and ultimately describe its transformation into energy for the purpose of recharging the battery. In this description we also utilize concepts such as signals to streamline the understanding of parameter exchanges between classes.

Before we delve into the explanation of the block diagram, it is important to establish a clear understanding of its graphical elements, as follows:

- Each block encompasses a bold description of its function according to the components of the split architecture of the SWIPT receiver, as outlined in Chapter 2, and is associated with its respective key class, whose details will be covered in-depth in Chapter 5. Other components, such as, AC-DC rectifier and the DC-DC converter are represented in terms

of their attributes of efficiency by the η and β variables, respectively. Whereas, the impedance matching circuit is not represented in the block diagram, due to the frame-basis operation of NS-3, as previously mentioned.

- The state machine controller block functions as the manager responsible for supervising the operational states of the information decoder. This oversight is encapsulated by the state diagram within the state machine controller block, ensuring synchronization of operational states across all blocks in conjunction with the information decoder. Furthermore, the block plays a crucial role in disseminating notifications to listener blocks whenever state transitions occur, as previously detailed.

- Blocks encased by dashed lines denote newly developed classes specifically for our implementation. In contrast, blocks enclosed by solid lines represent preexisting classes from the NS-3 library. Moreover, the dashed lines connecting these blocks symbolize the creation of new functions designed to enable the seamless exchange of parameters and data between these classes. This distinction helps clarify the architecture and integration of our implementation into the NS-3 framework.

We initiate our explanation by directing our attention to the antenna. After the PPDU is captured by the antenna, it then proceeds along a path within the SWIPT receiver circuitry until it reaches a crucial juncture, where a switch element is featured, symbolizing the power split factor denoted by the variable θ_s. The operation of the power split factor dictates the division of the power of the PPDU between two distinct blocks: the upper block, signifying the information decoder and the lower block, representing the energy harvester. Consequently, a specific fraction of the power is directed towards the input of the energy harvester, this allocation is described as

$$P_{EH} = P_{RX} G_{Antenna} (1 - \theta_s), \tag{4.1}$$

where P_{EH} represents the fraction of power of the PPDU directed to the input of the energy harvester, expressed in Watts. P_{RX} refers to the power of the received frame, expressed in Watts. $G_{Antenna}$ is the antenna linear gain and θ_s corresponds to the power split factor, in linear scale.

Whereas, the portion of power assigned to the input of the information decoder block is

given by

$$P_{ID} = P_{RX} G_{Antenna} \theta_s, \tag{4.2}$$

where P_{ID} represents the fraction of power of the PPDU at the input of the information decoder, expressed in Watts.

Another critical aspect to emphasize regarding the power split factor is its role in enabling the simultaneous execution of information decoding and energy harvesting. This is achieved by concurrently transmitting the structure of the PPDU to the input of both blocks. From this point forward, we will denote the structure of the PPDU directed to the input of the energy harvester with a power level of P_{EH}, as $PPDU_{EH}$, and the same PPDU structure routed to the input of the information decoder, but with a power level of P_{ID}, as the $PPDU_{ID}$.

Similarly, any interfering frame arriving at the antenna concurrently with the reception of the PPDU will experience a power division by the same factor θ_s. This division allocates a portion of power to both the information decoder and the energy harvester, as illustrated by the highlighted elements in red in Fig. 4.1. Specifically, the power fraction destined for the information decoder undergoes initial processing by the Interference Model block. Subsequently, the outcomes of this processing are directed to the information decoder block. In contrast, the power fraction associated with the interfering frame is directly channeled to the energy harvester.

Once the $PPDU_{ID}$ enters the information decoder block, it undergoes a sequence of assessments to determine its eligibility for further decoding. These assessments involve checking whether the PHY state machine is in either the IDLE or CCA_Busy states, and verifying if P_{ID} from Eq. (4.2) exceeds the ED threshold. If all these evaluations prove successful, the information decoder transmits a signal to the state machine controller, signaling the authorization to transition into the RX state for the resumption of $PPDU_{ID}$ reception. However, in case of any unsuccessful evaluations, the state of the information decoder may remain unchanged, contingent upon the assessment of the state machine controller.

At the same time that the $PPDU_{ID}$ enters the information decoder block, the $PPDU_{EH}$ is also directed into the energy harvester block. However, the $PPDU_{EH}$ is temporarily stored in a

buffer until it receives a signal indicating the state machine controller has transition into the RX state. In the event that the signal indicates a transition to any state other than RX, the energy harvester promptly discards the $PPDU_{EH}$.

Following the receipt of the signal from the information decoder, the state machine controller initiates the transition of the operational state moving it from IDLE or CCA_Busy to RX. Subsequently, the state machine controller dispatches a broadcast notification to both the SWIPT listener and the Power Management Unit (PMU), signaling the occurrence of the RX event.

Upon receiving the RX notification, the SWIPT listener transmits a signal to the energy harvester block in order to enable the start of the energy harvesting procedure. The amount of energy harvested can be expressed as follows

$$E_{EH} = \eta G_{Antenna}(1-\theta_s)(P_{RX}T_{PPDU_{EH}} + P_{Interference}T_{Interference}), \tag{4.3}$$

where E_{EH} represents the harvested energy, quantified in Joule, η corresponds the AC-DC conversion efficiency factor, according to (ZHANG *et al.*, 2015) and (ZHANG; HO, 2013), as cited by (LUO *et al.*, 2021); $T_{PPDU_{EH}}$ designates the duration for which the $PPDU_{EH}$ is received; $T_{Interference}$ is the duration for which the interfering frame overlaps in time with the $PPDU_{EH}$; and $P_{Interference}$ is the power of the interfering frame in Watts.

Likewise, when a notification of a state transition is received, the Power Management Unit (PMU) commences the necessary procedures to calculate the energy consumed in current state E_n, (J), as follows

$$E_n = V_{Bat}I_{state}\Delta t_{state}, \tag{4.4}$$

where V_{Bat} signifies the voltage level of the battery during the current state; I_{state} is the electric current drawn from the battery during the current state, defined in Amperes; and Δt_{state} denotes the duration of the current state, specified in seconds.

The immediate cumulative level of energy consumption after the conclusion of each state,

represented as E_{Acc}, (J), is calculated as

$$E_{Acc} = \sum_{s=1}^{n-1} E_s + E_n \tag{4.5}$$

where s denotes the previous state of the state machine controller has gone through and E_s denotes the energy consumed at the corresponding previous state.

At the conclusion of the simulation period, the total energy consumption, represented as E_{total} (J) for the SWIPT receiver, which is equivalent to E_{Acc}, can also be calculated as follows

$$E_{total} = \sum_{i=1}^{I} E_{RX_i} + \sum_{j=1}^{J} E_{TX_j} + \sum_{l=1}^{L} E_{IDLE_l} + \sum_{m=1}^{M} E_{CCA_m} + \sum_{n=1}^{N} E_{SLEEP_n} + \sum_{k=1}^{K} E_{SWITCH_k}, \tag{4.6}$$

where I is the total number the RX states during simulation. E_{RX_i} corresponds to the energy expenditure during the i-th RX state. J is the total number the TX states during simulation. E_{TX_j} corresponds to the energy expenditure during the j-th TX state. L is the total number the IDLE states during simulation. E_{IDLE_l} corresponds to the energy expenditure during the l-th IDLE state. M is the total number the CCA_BUSY states during simulation. E_{CCA_m} corresponds to the energy expenditure during the m-th CCA_BUSY state. N is the total number the SLEEP states during simulation. E_{SLEEP_n} corresponds to the energy expenditure during the n-th SLEEP state. K is the total number the SWITCHING states during simulation. E_{SWITCH_k} corresponds to the energy expenditure during the k-th CCA_BUSY state. And all energy values are expressed in Joules.

Subsequently, at the end of each state, the PMU transmits a signal to the battery, notifying it of the occurrence of a new event. This triggers the reduction of the energy stored in the battery by the quantity of energy consumed during that specific state. Nonetheless, in the context of RX events, the energy harvested during this state, as indicated in Eq. (4.3), is directed from the output of the energy harvester block to the battery block to facilitate its recharging. The quantity of energy utilized for recharging the battery $E_{Recharge}(J)$ is given by

$$E_{Recharge} = \beta\eta T_{PPDU_{EH}} P_{RX} G_{Antenna}(1 - \theta_s), \tag{4.7}$$

where β denotes the DC-DC conversion efficiency, which according to (IDOTA *et al.*, 1997) and (VELLACHERI *et al.*, 2014), as cited by (LUO *et al.*, 2021), for Lithium-ion batteries and supercapacitor ranges from 0.9 to 0.95.

The initial energy stored in the battery subtracted by the total energy consumed during all states, then added to the total energy harvested during all RX states, leads to the remaining level of energy stored in the battery, denoted as $E_{Remaining}$, which is given by

$$E_{Remaining} = E_{initial} - \sum_{s=1}^{n} E_s + \sum_{i=1}^{I} E_{Recharge_i} \tag{4.8}$$

where $E_{initial}$ denotes the initial level of energy stored in the battery; $\sum_{s=1}^{n} E_s$ corresponds to the cumulative level of energy consumption; I represents the number of RX events and $\sum_{i=1}^{I} E_{Recharge_i}$ represents the sum of the energy used for recharging the battery during the I events. All values are expressed in Joule.

4.1.1 Sustainability of the SWIPT Receiver

In the study conducted by Luo *et al.* (2021), a definition to assess the sustainability of SWIPT systems is proposed. In short, this proposition suggests that sustainability of an SWIPT system is achieved when the expected harvested energy is greater than or equal to the average energy consumption.

Adapting this to the current context implies that the total energy used for recharging the battery must be equal to or greater than the total energy consumption of the SWIPT receiver. Subsequently, in order to achieve sustainability, the $E_{Remaining}$ term should be equal to $E_{Initial}$ in Eq. (4.8). By rearranging this equation, we obtain

$$\sum_{s=1}^{n} E_s = \sum_{i=1}^{I} E_{Recharge_i} \tag{4.9}$$

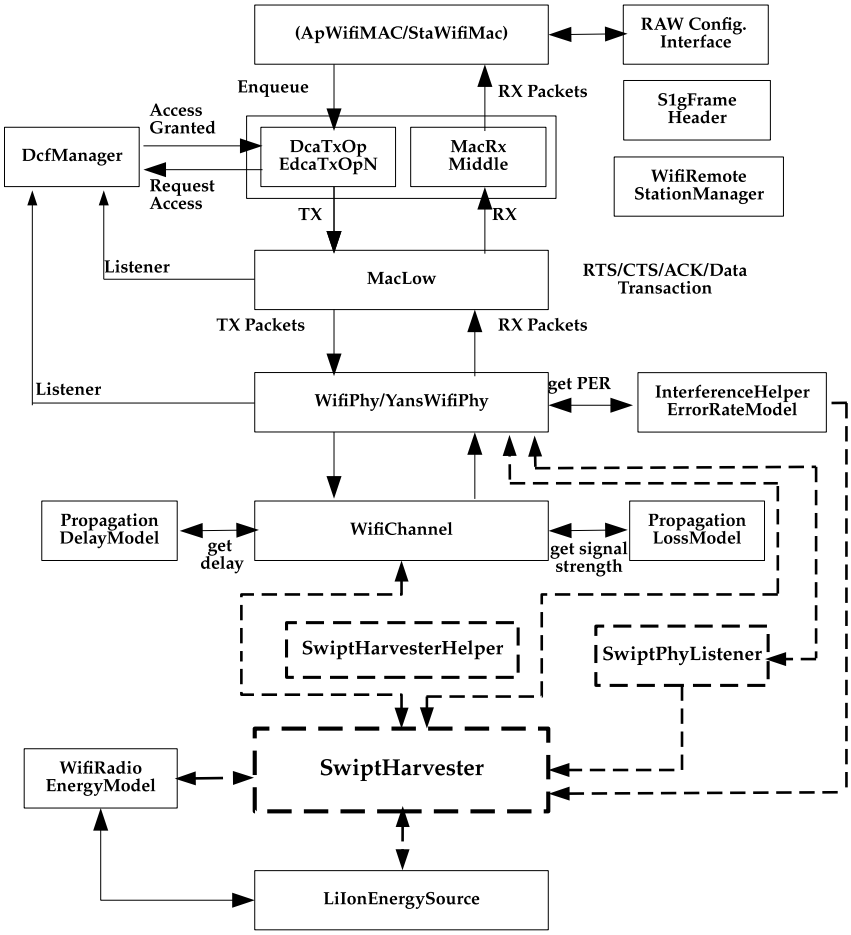
where $\sum_{s=1}^{n} E_s$ corresponds to the cumulative level of energy consumption and $\sum_{i=1}^{I} E_{Recharge_i}$ refers to the cumulative level of energy used for recharging the battery.

The assessment of the sustainability of our implementation will be expounded in Chapter 6.

4.2 INTEGRATION DESIGN OF THE SWIPT MODULE INTO THE WIFINETDEVICE ARCHITECTURE

In this section, we offer an overview of the design of the integration of the SWIPT module into the architecture of the `WifiNetDevice` object of NS-3.23, as illustrated in Fig. 4.2, which also encompass some major objects that belong to the energy framework.

Figure 4.2. SWIPT Integration to the WifiNetDevice Architecture



Source: Adapted from Fig. 2 in Ref. (TIAN *et al.*, 2018).

Before delving further, it is important to underscore that this section is primarily dedicated to elucidating the formulation of how the key mechanisms of the SWIPT module are meant to integrate into the architecture of the `WifiNetDevice`. This intentional focus aims to establish a clear link between the positioning of the blocks presented in the previous section and the crucial classes, which will be further detailed in Chapter 5.

Our discussion starts with a focus on the significance of the `WifiNetDevice` object, which, as outlined in (NS-3 Project, 2015b) serves to model the wireless network interface controller in alignment with the specifications presented in (IEEE, 2007). It also encompasses four distinct levels of models, namely MAC high models, MAC low models, rate control algorithms, and PHY layer models, as previously mentioned in Chapter 2.

Therefore, it is noteworthy that all classes belonging to our design connect to the blocks corresponding to PHY layer objects within the architecture of the `WifiNetDevice`. None of these classes exhibit any dependencies with the MAC high or MAC low models.

This can be observed through the presence of dashed lines connecting to the `SwiptHarvester` block, which is one of the major classes of our design, to other PHY layer components. It can also be verified in a lesser degree in the case of the `SwiptPhyListener` class, which only presents links connecting the `YansWifiPhy` to the `SwiptHarvester` classes.

A notable exception pertains to the `SwiptHarvesterHelper` class, which exhibits no connections to other classes with the architecture of the `WifiNetDevice` object. This is due to the fact that the `SwiptHarvesterHelper` class functions as a helper class, offering an optional feature to aid users in configuring the `SwiptHarvester` object within the simulation script of NS-3.

Furthermore, it is essential to note that the elements within our design establish connections with the energy framework. This is clearly depicted through the connections originating from `SwiptHarvester` and extending to both `WifiRadioEnergyModel` and `LiIonEnergySource`, as depicted in the lower section of Fig. 4.2.

All the depicted placements and interconnections, meticulously outlined thus far, serve as critical components that collectively support and prescribe the comprehensive functionality and behavior of the SWIPT receiver.

4.3 SUMMARY

In this chapter, we have provided an extensive explanation of the design of the SWIPT module into the energy framework and WiFi model library of NS-3. Our approach involved gradually presenting our design, beginning with a fundamental concept of the SWIPT receiver in the form of a block diagram. Then, we showcased how our design is intended to integrate

with the `WifiNetDevice` architecture, with an additional focus on significant dependencies with the energy framework of NS-3.

Throughout our explanation, we have referenced various classes, each possessing unique attributes and features within its respective contexts, which will be further explained in Chapter 5.

CHAPTER 5

THE IMPLEMENTATION OF SWIPT INTO NS-3

This chapter presents our implementation of the SWIPT module into NS-3, with the purpose of enabling the simulation of this technology over IEEE 802.11ah networks. In this context, we delve into the crucial technical details of this implementation, including the relevant classes, sub-classes, helper classes and others that have been integrated into NS-3.

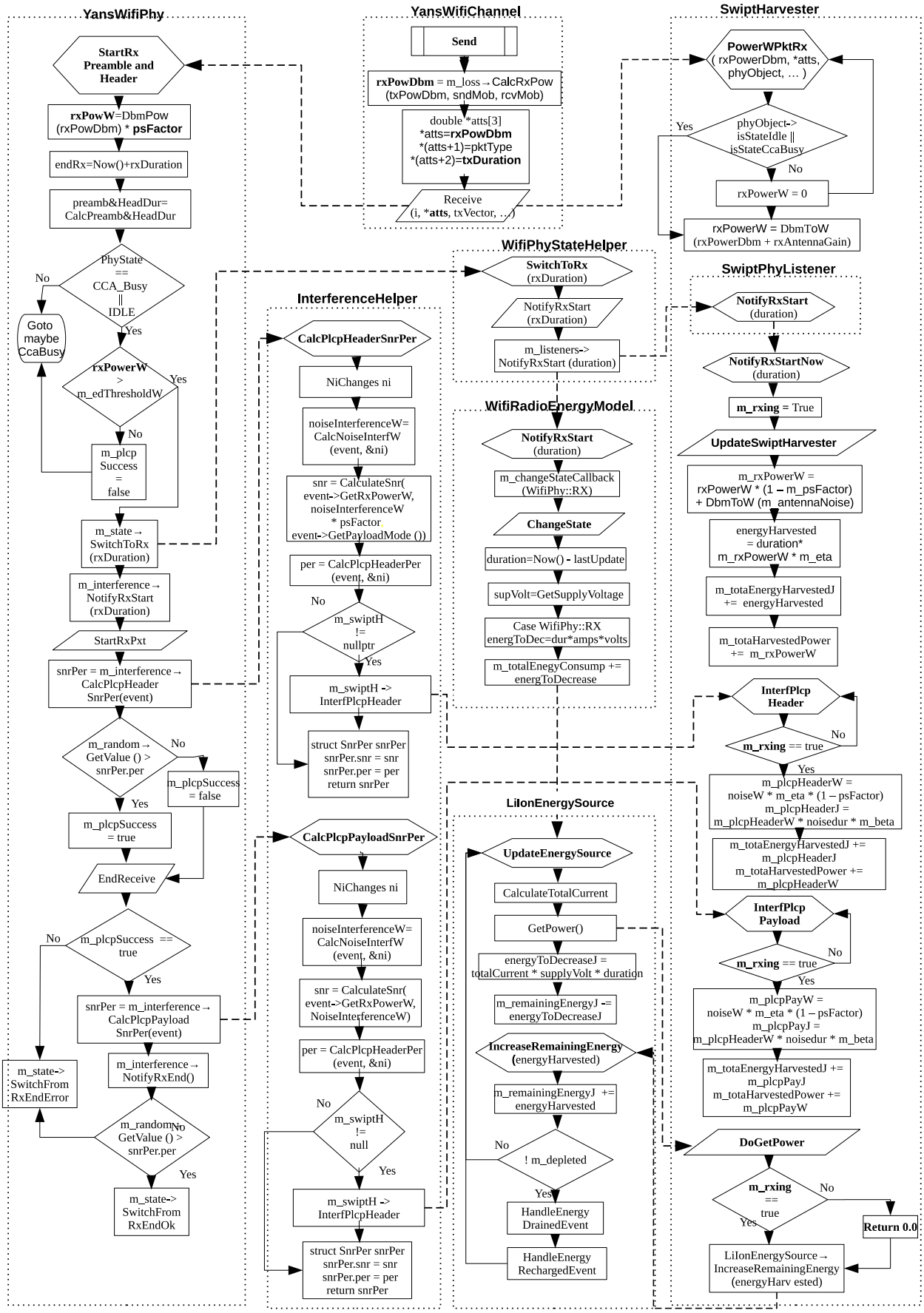
As mentioned in Chapter 2, our implementation has been developed within the IEEE 802.11ah module of NS-3 version 3.23, as detailed in (TIAN *et al.*, 2018). It also features as an independent component built upon the concept of the energy framework, as detailed in references (TAPPARELLO *et al.*, 2014a) and (TAPPARELLO *et al.*, 2014b). Consequently, it seamlessly integrates with various WiFi standards available in the IEEE 802.11ah module of NS-3.23, encompassing IEEE 802.11a, 802.11b, 802.11g, 802.11n, and 802.11ah. Furthermore, it offers interoperability with existing and future energy source models that adhere to the same framework, ensuring versatility and compatibility across a wide range of configurations.

Therefore, in the following section and subsections we enrich our discussion by presenting our implementation in a flowchart format, showcasing their features and dependencies within NS-3 objects, as depicted on Fig. 5.1.

5.1 ESSENTIAL CLASSES OF THE SWIPT IMPLEMENTATION

In this section, we will delve into the mechanisms underlying the key classes that are the cornerstone of our SWIPT implementation. Our objective is to provide an in-depth exploration of the most critical sections of the codebase from the perspective of the SWIPT receiver, emphasizing the essential features of both the existing classes within the NS-3 library and the new mechanisms we have incorporated to align with the blueprint presented in the flowchart depicted in Fig. 5.1.

Figure 5.1. Flowchart of Key Classes



Source: Own authorship.

It is important to acknowledge that providing a comprehensive description encompassing all functionalities and features across all classes within the codebase is an impractical endeavor. Hence, the flowchart illustrated in Fig. 5.1 succinctly presents the critical relationships, dependencies and attributes of the essential classes of our study. These essential classes are as follows:

- `YansWifiChannel`.
- `YansWifiPhy`.
- `WifiPhyStateHelper`.
- `InterferenceHelper`.
- `WifiRadioEnergyModel`.
- `LiIonEnergySource`.
- `SwiptHarvester`.
- `SwiptPhyListerner`.

5.1.1 `YansWifiChannel`

As its name implies, the `YansWifiChannel` is predefined class within the NS-3 library dedicated to modeling the characteristics of the WiFi channel within the WiFi model library of NS-3. As such, it oversees communications between nodes over the wireless channel and provides member functions for configuring a range of propagation loss models, propagation delay models, and other related propagation effects.

We begin our exploration of this class by focusing on the upper-middle section of the flowchart of Fig. 5.1, where the 'Send' block is situated within the `YansWifiChannel` class. The member function 'Send' is responsible for transmitting PPDUs over the WiFi channel. It also comprehends the two subsequent blocks underneath.

The first block showcases the code snippet responsible for computing the power level of the frame at the receiving end. This calculation is performed by utilizing a pointer to the chosen class of the propagation loss model to invoke a member function named 'CalcRxPow'. This member function receives parameters related to the power level of the transmitted frame and

the mobility models of both the sending and receiving nodes. As a result, the member function returns the power level of the frame at the receiving end, expressed in decibel-milliwatts (dBm), which is stored in the 'rxPowDbm' variable.

This approach might seem unconventional in this context, especially considering the expected association of the 'Send' member function with transmission-related matters. However, the availability of class-defined attributes related to the propagation loss and of the propagation delay models, in addition to the parameters related to mobility models and power levels of the transmitted frames offer a unique opportunity for reducing the complexity that would otherwise be introduced if this operation was carried out individually at each node of the network.

Moving forward, in the next block we come across a data structure declared within the NS-3 namespace used for grouping several related variables into one place, which is declared as 'atts'. It stores numerical values in the format of a 'Double' data type, including: the power level of the received frame, the frame type and the duration of the frame transmission.

It is important to note that the duration of frame transmission is identical to the duration of frame reception, even though the transmission of the preamble and header of a PPDU may use a different MCS from the transmission of the PSDU, as elaborated in Chapter 3. This consistency can be verified by examining the duration values stored in the variables 'txDuration,' 'rxDuration,' and 'duration', used by the member functions: 'Send' of the `YansWifiChannel` class, 'StartRx-PreambleAndHeader' of the `YansWifiPhy` class and 'NotifyRxStart' of the `SwiptPhyListener`, respectively. This uniformity is of paramount importance for carrying out a range of tasks distributed throughout the codebase, including scheduling the simulation of future events, synchronizing the processes of information decoding and energy harvesting, calculating the energy level extracted from each frame within the `SwiptHarvester` class and so on.

In the last stage, the member functions responsible for the reception of the frame within the adjacent classes, namely 'StartRxPreambleandHeader' in the `YansWifiPhy` class and the 'PowerWPktRx' in the `SwiptHarvester`, are invoked simultaneously by the 'Receive' member function, which passes the 'atts' struct, the value of the θ_s and other pertinent information for further processing.

5.1.2 `YansWifiPhy`

The characteristics and functionalities of the `YansWifiPhy` class are thoroughly documented in the section that describes the PHY model of the reference (LACAGE; HENDERSON, 2006). Therefore, as previously mentioned, our objective in this subsection is to present the most relevant features for the development of our implementation.

The initial member function triggered inside the `YansWifiPhy` class is the 'StartRX Preamble and Header'. This member function is responsible for performing a series of evaluations upon the reception of the first bit of an incoming frame, including verifying the necessary PHY state for frame reception, checking the compliance of the received power level with the sensitivity threshold and validating the preamble of the PPDU. These assessments are essential to determine the eligibility for receiving the remaining part of the frame.

In the next block, the task at hand is to ascertain the portion of power allocated for information decoding. This is accomplished by converting the received power level from dBm to Watts using the 'DbmPow()' member function. Then, the power level in Watts is multiplied by the value stored in the 'psFactor' variable, which corresponds to the θ_s parameter. The result of this operation is stored in the 'rxPowW' variable.

The operations carried out in the two subsequent blocks are relatively straightforward to grasp, as they refer to determining timing attributes used for scheduling the simulation of future events. The code in the first block determines the expected point in time for the frame reception to end, which is accomplished by adding the current simulation time to the duration of the PPDU and storing the result in the 'endRx' variable. The second block determines the specific duration of the preamble and header of the PLCP frame, storing the result in the 'preambleAndHeaderDur' variable for later processing.

This brings us to one of the most crucial junctures of our design, since the initial prerequisite for enabling the reception of any incoming frame is the accurate assessment of which state the PHY is operating on. Therefore, if the PHY state machine is found operating either on CCA_BUSY or IDLE states, the routine proceeds. If the PHY is found to be in the RX state, the frame is treated as an interfering frame, which will be explained further ahead. However, if the PHY machine state is in any other state, the frame is promptly discarded and the PHY

state machine may remain either on IDLE or CCA_BUSY states.

Following this, the value previously stored in the 'rxPowerW' variable is checked for compliance, according to the ED threshold, as specified in Table 23-31 of the (IEEE, 2017a). Thus, if the power level of the received frame complies with the specified ED threshold, the PHY state machine will be prompted to switch to the RX state. However, if the power level falls below this threshold, a flag denoting the successful reception of the PLCP frame will be set to false and the PHY state machine might transition to the CCA_BUSY state.

A brief explanation is warranted here. It is worth mentioning that some manufacturers have achieved significant technological advancements, allowing them to manufacture devices with substantially lower ED thresholds, as exemplified in (IMEC, 2018). Consequently, in such scenarios, sensitivity thresholds as low as -104 dBm may be adopted instead of the -96 dBm threshold specified in the IEEE 802.11ah standard, which might lead to a further reduction in the power allocated for information decoding and an increase in the power available for energy harvesting.

Once the frame has successfully passed the two preceding verification procedures, the PHY state machine is granted permission to transition to the RX state. This transition is coordinated by the `WifiPhyStateHelper`, which not only handles the PHY state machine operations, but also supervises the execution of various other procedures, including the broadcast of notifications to other classes. The classes which receive notifications are referred to as 'listener' classes. The operations carried out by listener classes are critical, especially when it comes to simultaneously decoding the remaining portions of the frame and conducting the energy harvesting processes.

In the subsequent block, the 'm_interference' pointer to the `InterferenceHelper` class invokes the member function 'NotifyRxStart', passing the frame duration as an argument, in order to notify that a RX event has happened for further processing. Next, the parallelogram that encompasses the member function 'StartRxPkt' represents the reception of the preamble and header of the PLCP frame. It is followed by the section of code responsible for calculating the SNR and the PER of the preamble and header. This is done by invoking the 'CalcPlcpHeaderSnrPer' member function within the `InterferenceHelper` class and passing the required parameters. The results are stored in the 'snrPer' struct.

In the next decision point, the probability of encountering errors during the reception of the

preamble and header is evaluated. This assessment is carried out by drawing a random number from a uniform distribution Random Number Generator (RNG), using the m_random pointer to function to call the 'GetValue' member function. Subsequently, this randomly generated number is compared to the PER value stored in the 'snrPer' struct. If the generated random number is greater than the PER value, the 'm_pclpSuccess' flag is set to 'true,' indicating successful reception. Conversely, if the random number is less than or equal to the PER value, the flag is set to 'false', signifying the presence of errors in the reception process.

In the subsequent phase, the 'EndReceive' member function is called to manage the reception of the PSDU. In the initial segment of code within this member function, a decision element verifies whether the preamble and header have been received without errors, as indicated by the 'm_pclpSuccess' boolean flag. If this condition is met, the member function continues. Otherwise, the 'm_state' pointer to the `WifiPhyStateHelper` invokes the 'SwitchFromRxEndError' member function to terminate the process and discard the PLCP frame.

In the next two blocks the 'm_interference' pointer to the `InterferenceHelper` class calls two member functions. The first member function, 'CalcPlcpPayloadSnrPer', is used for calculating the SNR and PER of the PSDU. The second member function, 'NotifyRxEnd', is called to indicate that the PSDU is in the process of being received.

Once again, an evaluation for the presence of errors is conducted, this time focusing on the PSDU. The assessment is carried out in the same fashion as before. A pseudo-random number is drawn from a uniform distribution RNG and compared to the PER value stored in the 'snrPer' struct. If the assessment succeeds, the 'm_interference' pointer connected to the `InterferenceHelper` class calls the 'SwitchFromRxEndOk' member function. In case of failure, it invokes the 'SwitchFromRxEndError' member function to terminate the process and discard the PLCP frame.

A final point regarding the drop of frames due to error detection must be clarified. From the perspective of the energy harvesting process, the discard of frames do not interrupt the energy harvesting operation. This can be attributed to the fact that the SWIPT harvester solely focuses on extracting energy from incoming frames and does not perform any information decoding operation. As a result, the presence of errors does not affect the continuity of the process of energy scavenging, as power reception continues to be received until the last bit of the

frame arrives. Nonetheless, it is worth noting that the level of harvested power may fluctuate due to interference from other frames, as previously described on Chapter 2.

5.1.3 `WifiPhyStateHelper`

The `WifiPhyStateHelper` is a helper class available in the NS-3 library that is crucial for managing the operations of the PHY state machine, broadcasting information about state transitions and serving various functions within the codebase presented in the flowchart of Fig. 5.1.

The initial block within the the `WifiPhyStateHelper` class represents the 'SwitchToRx' member function. It is invoked by the 'StartRxPreambleAndHeader' member function of the `YansWifiPhy` class when the assessments concerning the operating state of the PHY and the ED threshold are successfully completed.

The 'SwitchToRx' member function also receives the value of the frame duration as an argument upon its calling. As previously mentioned, the frame duration is vital for keeping the synchronization of the information decoding and energy harvesting processes and so on.

Then, in the following block, the 'NotifyRxStart' member function broadcasts notifications to all listener classes about the transition of the PHY to the RX state. In order to receive notifications about the PHY state machine events, a listener class must be registered with the `YansWifiPhy` class, which keeps the records of all registered listener classes in a vector of pointers declared as 'm_listerners'. It is worth noting that the process of broadcasting and listening to notifications is confined inside each node of the network. Hence, listeners inside one node do not receive notifications sent from other nodes.

In the final block, a loop iterates through each listener class registered in the 'm_listeners' vector. For each registered listener, it accesses the pointer to send the notification about the RX transition that has just occurred.

5.1.4 `InterferenceHelper`

This class is a preexisting component in the NS-3 library, responsible for conducting operations associated with the determination of PER and SNR, as detailed in Chapter 2. To enhance

its functionality for harvesting the energy of interfering frames, we have implemented specific modifications.

It is worth clarifying that this class employs a function denominated NI changes to represent all relevant network interface changes that should be monitored during the reception of the PLCP. The monitoring of NI changes is carried at two specific phases: during the reception of the preamble and header of the PLCP, and during the reception of the PSDU. This is performed to ensure the correct assessment of the BER according to their respective MCS, which are usually different from one another.

The initial member function within this class, which is called by the `YansWifiPhy` class, is 'CalcPlcpHeaderSnrPer'. It receives the 'event' denoting the preamble and header of the PLCP frame, along with associated attributes, such as the preamble and header duration.

Subsequently, within this member function, the NI changes iterates through any pertinent chunks of n bits that may overlap with the preamble and header of the PLCP frame. This process yields the calculation of the power level in watts for the interfering chunk, which gets stored in the 'noiseInterferenceW' variable, as depicted in the code snippet of the second block.

Additionally, the 'CalculateSnr' method is called to provide the SNR of the corresponding chunk. This calculation relies on parameters such as the power level of the event, the value stored in 'noiseInterferenceW', the 'psFactor', and the MCS of the event. The outcome of this computation is stored in the 'snr' variable.

It is worth mentioning that the value of psFactor is used by the 'CalculateSnr' method, since the SNR results should also reflect the fractions of power of the received frame and noise interference allocated to the `YansWifiPhy` class. Whereas, the value of (1 - psFactor) is used allocate the fraction of power of the incoming frame and also of the noise interference to the `SwiptHarvester` class. Therefore, due to this reason it is important to mention that the Eq. (2.10), has been modified to reflect this, as follows

$$SNIR(k,t,\theta_s) = \frac{S_k(t) \times \theta_s}{N_i(k,t) \times \theta_s + N_f}, \tag{5.1}$$

where $S_k(t)$ is the power of the received frame, in Watts; θ_s is the power split factor, in linear scale; $N_i(k,t)$ is the noise interference, in Watts; and N_f is the noise floor, in Watts.

Furthermore, the value of PER is ascertained through the 'CalcPlcpHeaderPer' method, employing the attributes of the event and relevant parameters on the chunk under inspection. Then, in the next block in the flowchart, a verification routine is carried out in order to ensure that only SWIPT-enabled nodes are called by the pointer to class code snippet. This is done through the verification of the value stored in the pointer to the `SwiptHarvester` object does not hold a 'NULL' value, since this condition represents only the AP, which does not possess energy harvesting capabilities.

Next, in the subsequent block of the flowchart, a verification routine is conducted to ensure that only SWIPT-enabled nodes are invoked by the pointer to class code snippet. This verification is achieved by checking that the value stored in the pointer to the `SwiptHarvester` object does not hold a 'NULL' value. This condition is essential as it represents only the AP, which does not possess energy harvesting capabilities.

In the final stage, the 'SnrPer' struct is employed to store the values of SNR and PER, which are returned to the `YansWifiPhy` class. The second member function illustrated in the section of the flowchart is the 'CalcPlcpPayloadSnrPer'. It performs the same operation for the PSDU for the reasons explained earlier.

5.1.5 `WifiRadioEnergyModel`

This class is an integral part of the Energy Framework of NS-3, serving as a repository for the attributes and variables essential for calculating energy consumption associated with each state of the PHY. While our implementation has not introduced any modifications to this class, its importance cannot be overstated.

Also, it contains attributes related to parameters such as electrical current draw, state durations, and battery voltage levels, which are either derived from neighboring classes or configured as attributes, in alignment with the values specified in the reference (IMEC, 2018). These attributes are of paramount importance for accurately modeling energy consumption, particularly for nodes adhering to the IEEE 802.11ah standard.

The first member function featured within this class is the 'NotifyRxStart', which is also triggered by the 'm_listeners' vector of the `WifiPhyStateHelper` class when a notification is

broadcast informing the transition of the PHY state machine to RX, including the argument about the duration of the frame.

The next block displays a callback function designed to notify the base object class about state transitions. The reason behind this approach is to update the remaining energy stored in the `LiIonEnergySource` object installed in each node and to manage energy depletion and recovery events.

Following this, the 'ChangeState' member function, responsible for determining the energy expenditure at each state, is invoked. Afterward, the code snippets displayed in the next two blocks calculate, at first, the duration of the present operating state by subtracting the time of the last update from the current simulation timestamp, and storing the result in the 'duration' variable. Secondly, the 'GetSupplyVoltage' member function is invoked to read the voltage level of the energy source and the result is stored in the 'supVolt' variable.

Next, we encounter another decision symbol, which represents a switch statement. This switch statement tests the current state of the PHY to calculate its corresponding level of energy consumption. Specifically, the block displayed in the flowchart illustrates the code snippet for the energy expenditure at the RX state. In this context, the previous values stored in the variables 'duration' and 'supVolt' are multiplied by the value of the 'amps' variable, which corresponds to the current drawn at the RX state. The result of this operation is stored in the 'energToDec' variable.

In the final block, the 'm_totalEnergyConsump' trace source, representing the total energy consumed by the node, has its value increased by the value of the 'energToDec' variable. This action signals all connected trace sinks that an event corresponding to an increase in the total energy consumption has just occurred.

5.1.6 `LiIonEnergySource`

This class is also predefined within the Energy Framework of NS-3. It models the operation of Lithium-Ion (Li-Ion) batteries by configuring the necessary parameters. By default, it is set up to match the attributes of the Panasonic CGR18650DA Li-Ion Battery, as referenced in (PANASONIC, 2010), which served as the basis for the configuration of our simulations.

The first member function invoked in the `LiIonEnergySource` class is the 'UpdateEnergySource'. This is done to ensure that the amount of energy consumed at the end of each state of the PHY is accurately decremented from the remaining energy levels of the battery.

To accomplish this, in the next block, the amount of current drawn at the end of each state is obtained by the 'CalculateTotalCurrent' member function. The result of this operation is used further ahead in the flowchart.

Then, the `LiIonEnergySource` class utilizes the 'GetPower' member function in order to access the level of energy extracted from the received frame, which, in turn, calls the 'DoGetPower' member function of the `SwiptHarvester` class. And the result is stored in the 'totalEH' variable.

At this point, it is important to note that the amount of energy returned by the 'DoGetPower' member function depends on which state the PHY is operating on. Therefore, if the PHY state machine is found operating in the RX state, the 'DoGetPower' member function will return the amount of energy that the `SwiptHarvester` class has been capable of scavenging during the same RX state. However, if the PHY is in any state other than RX, it will return zero. Additional mechanisms involved in this procedure will be explained in the subsection related to the `SwiptHarvester` class.

Next, similarly to the 'Change State' member function in the `WifiRadioEnergyModel` class, the subsequent block calculates the energy consumed at the end of each state and stores the result in the 'energyToDecrease' variable. However, its purpose differs slightly from the 'Change State' member function, as it utilizes the value stored in the 'energyToDecrease' variable to decrement the remaining energy level of the battery recorded in the 'm_remainingEnergyJ' trace source.

Furthermore, 'IncreaseRemainingEnergy' member function is invoked by the `SwiptHarvester` class, passing as an argument the value stored in the 'energyHarvested' variable. Then, the 'm_remaining' energy trace source at this turn gets its value incremented by the amount stored in the 'energyHarvested' variable.

Thus far, we presented two major member functions that manage the remaining energy levels of the battery. The first, 'UpdateEnergySource,' is responsible for decrementing the remaining energy level, according to the energy expenditure at each state. The second, 'IncreaseRemain-

ingEnergy', performs the opposite operation, receiving the amount of energy provided by the `SwiptHarvester` to recharge the battery.

It is important to note that the 'IncreaseRemainingEnergy' member function was introduced in our implementation to model the battery recharging process while ensuring that energy consumption in the RX state and recharging occur within the same time frame, while the PHY remains in the RX state. To achieve this, a double-checking procedure was implemented to guarantee that the battery recharging process aligns with the energy harvesting process. This procedure can be thought of as a two-way handshake, where the `LiIonEnergySource` object requests the `SwiptHarvester` to return the harvested energy levels, and the `SwiptHarvester` verifies if the PHY is in a suitable state, before returning the corresponding levels of harvested energy.

Following this, the battery is checked for energy depletion. Therefore, if the test succeeds, the routine continues as normal. However, if the test fails and the battery is found to be depleted, a procedure to restore suitable energy levels is executed by invoking the 'HandleEnergyDrainedEvent' member function. This procedure is responsible for instructing the `WifiRadioEnergyModel` to shut down all node operations and take appropriate measures to restore energy levels. Once the battery is replenished, the 'HandleEnergyRecharged' member function is called to resume the operations of the node.

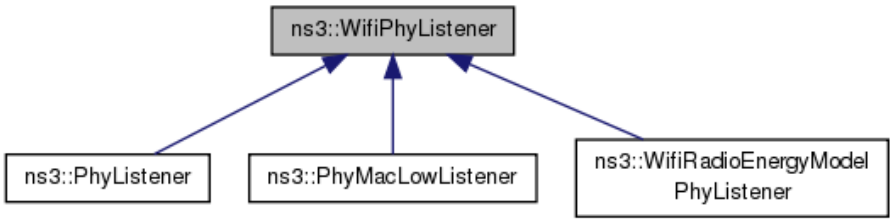
It is important to note that the procedures involved in the 'HandleEnergyDrainedEvent' and 'HandleEnergyRecharged' member functions have been developed for our implementation but have not been fully tested, because our focus has primarily been on presenting the necessary configurations for maintaining sustainable energy levels through the harvested energy, as discussed in the next Chapter. In our future work, we plan to evaluate the relevance of further testing and incorporating these procedures into our implementation.

5.1.7 `SwiptPhyListerner`

The `SwiptPhyListerner` is a derived class from the base object class `WifiPhyListener` specifically developed for the implementation of our model. Therefore, it inherits the same attributes and special properties found in the `WifiRadioEnergyModelPhyListener`, `PhyListener`

and PhyMacLowListener classes, as depicted in Fig. 5.2.

Figure 5.2. WifiPhyListener



Source: (NS-3 Project, 2015b).

The SwiptPhyListerner class is represented by the box enclosed by a dotted line inside the SwiptHarvester class. For each PHY state discussed in Chapter 3, the SwiptPhyListerner class includes a corresponding member function to receive notifications and to invoke another related member function within the SwiptHarvester class. These invoked member functions within the SwiptHarvester class, among other features, set control flags, which are subsequently checked to enable or disable the energy harvesting process.

In order to fulfill the requirement of being registered with the YansWifiPhy class previously explained, the SetUpSwiptPhyListener member function of the SwiptHarvester class invokes the 'RegisterListener' member function of the YansWifiPhy class passing the pointer to the SwiptPhyListener object. This operation ensures proper synchronization between the SwiptPhyListener object and the WifiStateHelper class, enabling seamless operation when a notification about a new state is broadcast.

Specifically, in the case of the RX event, the enclosed block within the SwiptPhyListerner class contains the member function that is invoked by the WifiPhyStateHelper 'm_listeners' vector. This member function is responsible for receiving notifications regarding the PHY transition to the RX state and obtaining the duration of the frame. Once this member function is called, it triggers another member function within the SwiptHarvester class to initiate the necessary steps for enabling the energy harvesting process. However, if the SwiptPhyListerner class receives notifications for events other than RX, the flag indicating the occurrence of the receiving process is set to false, and the energy harvesting process is halted.

5.1.8 `SwiptHarvester`

The `SwiptHarvester` class is one of the most important classes we have introduced in our implementation. Its operation is aligned with the theoretical foundation of the SWIPT split architecture using the Power Splitting (PS) technique outlined in Chapter 3.

This class is defined within the namespace of NS-3 as a sub-class derived from the 'Energy-Harvester' base object class inheriting its attributes and defining some of its own, including the following.

1. The power split factor (θ_s) represented by the 'm_psFactor' variable.
2. The antenna noise represented by the 'm_antennaNoise' variable.
3. The SWIPT harvester AC-DC conversion efficiency (η) represented by the 'm_eta' variable.
4. The DC-DC conversion efficiency (β) represented by the 'm_beta' variable.

The `SwiptHarvester` class also features two trace sources that signal events that happen during simulation to provide information about changes in the values of the following variables:

1. The total amount of harvested power represented by the 'm_totalHarvestedPower' variable.
2. The total amount of harvested energy represented by the 'm_totalEnergyHarvestedr' variable.

At the start of the `SwiptHarvester` class, the 'PowerWPktRx' member function is invoked by the 'Receive' member function of the `YansWifiChannel` class, which passes various attributes about the received frame as arguments, including the received power level in dBm, frame duration, frame type and more. These attributes are simultaneously received by both the `YansWifiPhy` class and the `SwiptHarvester` class, ensuring the concurrent operation of the SWIPT receiver, as previously described.

Following this, the PHY state machine undergoes a verification process to ensure it is functioning within the necessary states for frame reception, specifically, 'IDLE' or 'CCA_Busy' states. This verification is represented by a code snippet that contains the 'phyObject', which is a pointer to the `YansWifiPhy` object, that calls two member functions: 'IsStateIdle' and

'IsStateCcaBusy,' both of which return boolean values. If this verification is successful, the power received by the node, expressed in Watts, is determined for further energy harvesting procedures. This is done by adding the power of the received frame, expressed in units of dBm, to the gain of the antenna, expressed in dBi. The result of this addition is converted from dBm to Watts by the 'DbmToW' member function and then stored in the 'm_rxPowerW' variable. However, in the event of a failed verification, the variable 'm_rxPowerW' is set to zero, and the 'PowerWPktRx' member function is called again to await the reception of the next frame.

Next the box enclosed with dotted lines inside the `SwiptHarvester` class represents the code of the `SwiptPhyListerner` class that is declared within the source file of the `SwiptHarvester` class. The next block displays the 'NotifyRxStart' member function, which is responsible for receiving notifications regarding the transition of the PHY state machine into the RX state, as previously described. This illustration of the 'NotifyRxStart' member function was chosen, because in this context we centralize our attention on the processes related to the frame reception, as previously stated. However, it is crucial to note that the `SwiptPhyListerner` class comprises routines corresponding to each possible state of the PHY, each handling notifications specific to its corresponding state. All the major details about its features have been already covered in the previous subsection that describes the `SwiptPhyListerner` class.

Upon receiving a notification indicating the transition of the PHY state machine to the RX state, the 'NotifyRxStart' member function calls the 'NotifyRxStartNow' member class of the `SwiptHarvester` class. This function also passes the duration of the frame as an argument. In the following block, the 'm_rxing' boolean flag is set to true, which enables the energy harvesting procedure to be executed, while the PHY state machine remains in the RX state.

As a consequence of that, the 'UpdateSwiptHarvester' member function is called to compute the total amount of energy harvested during the simulation interval. In order to accomplish that, first the share of power that must be directed to the inputs of the `SwiptHarvester` class is calculated. This calculation consists of multiplying the value stored in the 'rxPowerW' variable to $(1 - \theta_s)$ and then adding the level of noise attributed to the antenna that is converted from dBm into Watts by the 'DbmToW' member function. The result is stored in the 'm_rxPowerW' variable.

After this, the level of energy harvested from a frame is determined by multiplying three

variables together: the duration of the frame, the value stored in the 'm_rxPowerW' variable, and the value stored in the 'm_eta' variable, which corresponds to the efficiency of the AC-DC conversion. The result of this operation is stored in the 'energyHarvested' variable.

The values stored in the 'energyHarvested' variable, obtained each time the PHY transition of the RX state, are accumulated into the 'm_totalEnergyHarvestedJ' variable, which represents the total amount of energy harvested during the simulation period. The 'm_totalEnergyHarvestedJ' variable serves also as a trace source for signaling each increment in the total amount of energy harvested. This process ensures the continuous monitoring and recording of the harvested energy and power throughout the simulation interval.

Next, the member function denominated 'InterfPlcpHeader' gets called by the pointer to function 'm_swiptH' of the `InterferenceHelper` class, passing the necessary arguments for the computation of the harvested power and energy from the fraction of power determined by the psFactor from the chunks of interfering frames that may have overlapped with the preamble and header of the PLCP.

In order for the extraction of power and energy from these chunks to take place, the PHY must be operating at the RX state, otherwise the 'InterfPlcpHeader' awaits for the next call.

In the following blocks the code snipped representing the level of the power and energy harvested from the chunks of interfering frames adopts the same parameters used in the 'UpdateSwiptHarvester' member function to extract the power and energy from the PLCP. The results are used to recharge the battery and added to the 'm_totalHarvestedPower' and 'm_totalEnergyHarvestedJ' variables, respectively.

It is worth mentioning that, as explained in the subsection regarding the `InterferenceHelper` class, for each member function that calculates the SNR and PER of the preamble and header of the PPDU and for the PSDU, there is a corresponding member function in the `SwiptHarvester` class to extract their energy, respectively. Therefore, the next member function executes the same operation mentioned on the previous paragraph, but for the chunks found in the PSDU.

Finally, when the `LiIonenergySource` calls upon the `SwiptHarvester` to transfer the extracted energy for recharging, the 'DoGetPower' member function assesses whether the 'm_rxing' flag is set to true. If this verification succeeds, the `SwiptHarvester` triggers the

'IncreaseRemainingEnergy' member function of the `LiIonenergySource`, passing along the amount of extracted energy in the 'energyHarvested' variable an argument. However, if the verification fails, the 'DoGetPower' member function returns a value equals to zero. As previously explained, this mechanism ensures that energy used for the battery recharging corresponds to the same energy extracted from the frame received during the same time-frame while the PHY remained in the RX state.

5.2 SUMMARY

In this chapter, we have provided an extensive explanation of the integration of our SWIPT implementation into the energy and WiFi modules of NS-3. Our approach involved providing an illustrative blueprint of our implementation in the form of a flowchart. This flowchart highlights the essential classes and their interdependencies, offering a comprehensive understanding of our implementation.

Throughout our explanation, we have referenced various classes, each possessing unique attributes and features within its respective contexts. In the following list, we provide a brief summary of the attributes of each class discussed in this chapter:

- `YansWifiChannel`: This class is a predefined element in the NS-3 library, specifically designed to model the characteristics of the WiFi channel within the WiFi model library of NS-3.
- `YansWifiPhy`: This class models the PHY layer of the IEEE 802.11a/b/g/n/ah standards present in the version 3.23 of NS-3. It is also denoted as the WiFi information decoder.
- `WifiPhyStateHelper`: This is a vital helper class found in the NS-3 library. Its primary role involves managing the operations of the PHY state machine and disseminating information about state transitions.
- `InterferenceHelper`: This class is an existing component within the NS-3 library, tasked with performing operations related to determining the values of PER and SNR.
- `WifiRadioEnergyModel`: This is another class already packed with the NS-3 distribution

that stores variables of each state of the state machine of the PHY for power consumption calculations.

- `WifiPhyListener`: This is an already existing class of the NS-3 library from which the `SwiptPhyListener` inherits its attributes from.
- `LiIonEnergySource`: This class is predefined within the Energy Framework of NS-3, specifically designed to model the operation of Lithium-Ion (Li-Ion) batteries.
- `SwiptHarvester`: This class is part of our implementation over the NS-3 energy framework in order to enable the simulation of SWIPT harvester.
- `SwiptPhyListener`: This class is also part of the of our implementation. It keep the synchronicity of the simultaneous operation of SWIPT by forwarding information about the transition of states of the PHY state machine.

CHAPTER 6

SIMULATION RESULTS

In this chapter, we present the outcomes derived from the extensive simulations we conducted of our implementation on the NS-3 platform. Our main objective is to thoroughly characterize how our implementation behaves across different scenarios within the IEEE 802.11ah network standard. Additionally, we aim to identify constraints that could potentially impact the overall performance of the network or hinder the energy harvesting operations incorporated in our SWIPT implementation.

Therefore, to further explore these aspects and provide a clearer understanding of the functionalities of our implementation, we showcase the outcomes obtained using seven metrics. Some of these metrics may be more suitable for a specific scenario, including SNIR, frame loss, UDP packet loss, throughput, harvested power, harvested energy, and energy sustainability.

The first scenario corresponds to the network architecture outlined in our study presented at the Toll Hall of the XLI Brazilian Symposium on Computer Networks and Distributed Systems (SBRC 2023), detailed in (JUNIOR; CARVALHO, 2023). In this specific scenario, our main emphasis was on examining how our implementation behaves within the context of an IEEE 802.11ah network architecture. This architecture is characterized by a single-link configuration, which involves one SWIPT-enabled node and one AP.

Whereas, in the second scenario, which constitutes the primary focus of our ongoing work, we delve into a detailed examination of groups of SWIPT-enabled nodes operating within a network environment, facing constraints typically encountered in real IEEE 802.11ah networks. Our goal is to characterize the behaviour of our implementation and discern the critical effects on both network performance and the capabilities of energy harvesting.

The aforementioned metrics we used for obtaining the results of the simulations carried out in both scenarios are detailed as follows:

- SNIR (db): the results of this metric are determined by the `InterferenceHelper` class, as described in the Chapter 5, using Eq. (5.1).

- Frame loss (%): the results of this metric are calculated at the end of the simulation, as follows

$$FrameLoss(\%) = \left(1 - \frac{Frames_{RX}}{Frames_{TX}}\right) \times 100\%, \tag{6.1}$$

where $Frames_{RX}$ is the number of frames received by a node during the simulation interval and $Frames_{TX}$ is the number of frames transmitted by the AP during the simulation interval. They are provided by the methods 'MonitorRx' and 'MonitorTx', respectively, configured in the simulation script presented in Appendix B.

- UDP packet loss (%): The results of this metric are given by

$$UDPLoss(\%) = \left(1 - \frac{UDP_{RX}}{UDP_{TX}}\right) \times 100\%, \tag{6.2}$$

where UDP_{RX} represents the count of UDP packets received by the UDP server over the duration of the simulation interval and UDP_{TX} are the number of UDP packets transmitted by the UDP clients during the same simulation interval. They are provided by the 'udpPacketReceivedAtServer' and 'OnUdpPacketSent' trace sources of the `UdpServer` and `UdpClient` classes, respectively.

- Throughput: The results of this metric are obtained as follows

$$\text{Throughput} = \frac{UDP_{RX} \times \text{Payload Size} \times 8}{\Delta T}, \tag{6.3}$$

where Payload Size is a configuration parameter presented in Table 6.1 and in Table 6.2, and ΔT is the simulation interval.

- Harvested Power: is the total amount of power an SWIPT-enabled node is capable to extract from the $PPDU_{EH}$ received during the simulation interval, using Eq.4.1.

- Harvested Energy: is the total amount of energy an SWIPT-enabled node is capable to extract from the $PPDU_{EH}$ received during the simulation interval, using Eq.4.3.

- Energy Sustainability: refers to a condition in which the energy used for recharging the battery, sourced from the energy harvester, is either equal to or exceeds the energy expenditure of an SWIPT-enabled node, as elaborated in Chapter 5.

Nonetheless, before we proceed, it is noteworthy that as our research progressed through time, shifting our investigative focus from the initial scenario to the second, the degree of complexity increased to the point that it challenged us to delve deeper into exploring new solutions within the context of this research to ensure that our implementation could work and closely mirror real-world case scenarios. Consequently, certain parameters that were tailored to the specific conditions of the first scenario had to be replaced with more suitable approaches in the second scenario. The consequence of these adaptations on the information presented in this chapter is that the results obtained for metrics under the same name in both scenarios might not yield directly comparable results. Additionally, the subsequent characteristics are applicable to both scenarios:

1. The battery used in our setup is the Lithium-Ion model integrated into the energy module of NS-3, described in (NSNAM, 2010), and its specific parameters are detailed in (PANASONIC, 2010), due to $0.9 \leq \beta \leq 0.95$.
2. The sensitivity of the SWIPT harvester is presumed to be equivalent to the ED threshold of -104 dBm, which is configured for the information decoder. This assumption is made due to the limited distance range under investigation, facilitating this simplification.
3. The device energy model has been set up using the parameters as outlined in (IMEC, 2018), in addition to the supplementary parameters detailed in (LEE *et al.*, 2021).
4. The deployment of the SWIPT harvester is exclusive to network nodes, as the AP does not offer support for the installation of the SWIPT harvester.

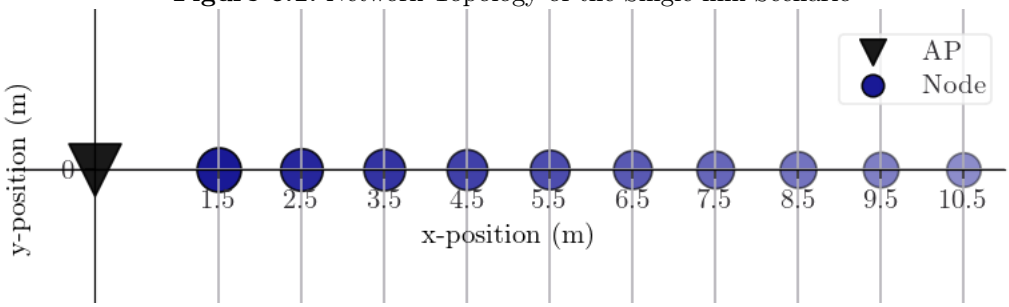
6.1 SINGLE LINK SCENARIO

As previously mentioned, this scenario consists of an IEEE 802.11ah network architecture featuring one SWIPT-enabled node and one AP. We deliberately chose this specific network architecture to conduct the initial characterizations of our implementation, due to the fact that it could enabled us to isolate the subject of our investigation from external influences, such as frame collisions induced by hidden nodes, which could introduce unforeseen outcomes. This approach allowed us to evaluate the presence of only internal factors that might potentially constrain the operation of our implementation independently.

6.1.1 Simulation Setup

The setup for this scenario included deploying a UDP server at the SWIPT-enabled node and a UDP client at the AP. This configuration aimed to establish a saturated traffic flow, employing a UDP payload size of 1024 bytes. The traffic was generated at regular intervals of 54.62 milliseconds, as detailed in Table 6.1. The primary goal of this configuration was to characterize our implementation and identify potential constraints.

Figure 6.1. Network Topology of the Single-link Scenario



Source: Own authorship

Regarding the configuration of the RAW mechanism deployed in this scenario, according to Chapter 2, the RPS element was set to be broadcast inside beacon frames, which are sent at each interval of 102400 μs, as specified in Table.6.1. The RPS contains information about only one RAW Group which includes the SWIPT-enabled node with the ID number of 0. Therefore, the N_{RAW} parameter of Eq. (2.8) was set to 1 RAW slot. Then, the C_{SLOT} parameter of Eq. (2.7) was set to the value of 849, resulting in the value of D_{SLOT} equals to 102380 μs, which accounts for the same overall value of D_{RAW}, which is less than the beacon interval.

Also, in order to assess both network and energy harvesting performance at various levels of received power, we manipulated the distance between the node and the AP by incrementally increasing it from 1.5 m to 10.5 m, with steps of 1.0 m. For each specified increment, we adjusted the values of θ_s within a range from 0 to 1.0, with steps of 0.1.

The decision to set the lower limit value within the distance range at 1.5 m was intentional and based on the reference distance specified for the macro deployment propagation loss model, as outlined in Table 6.1, which by default is set to 1 m. Consequently, values where the distance was less than or equal to the reference distance were deliberately excluded to prevent unforeseen

outcomes. The diagram shown in Fig. 6.1 illustrates the single-link scenario. In this scenario, an SWIPT-enabled node is depicted at various distances within the specified range. Also, the configuration of other essential parameters for this simulation scenario is presented in Table 6.1.

Table 6.1. Single-Link Scenario Configuration Parameters

Parameter	**SWIPT-enabled Node**	**Access Point**
Radio Frequency	900 MHz	900 MHz
Beacon interval	-	102400μs
Beacon Frame Duration	-	5600μs
Transmission Power	0 dBm	30 dBm
Antenna Gain	6 dBi	6 dBi
Short Guard Enabled	False	False
Channel Width	1 MHz	1 MHz
Energy Detection Threshold	-104 dBm	-104 dBm
CCA Mode1 Threshold	-107 dBm	-107 dBm
LDPC Enabled	False	False
S1G Short Field Enabled	False	False
S1G 1M Field Enabled	True	True
Noise Figure	6.8	6.8
Propagation Loss Model	Macro deployment	Macro deployment
Data Mode	150 kbps	150 kbps
Control Mode	150 kbps	150 kbps
PLCP Preamble and Header Bit Rate	150 kbps	150 kbps
UDP payload size	–	1024 bytes
UDP packet interval	–	54.613 ms

Source: Own authorship

6.1.2 Results

In this subsection, we present the outcomes derived from simulations of our implementation within the single-link scenario. The results are structured based on commonly used metrics, including SNIR, frame loss, throughput, total power harvested, total energy harvested, and sustainability. While the sustainability metric is not conventionally employed in network scenarios, it has been utilized in some related works, as illustrated in Table 3.1.

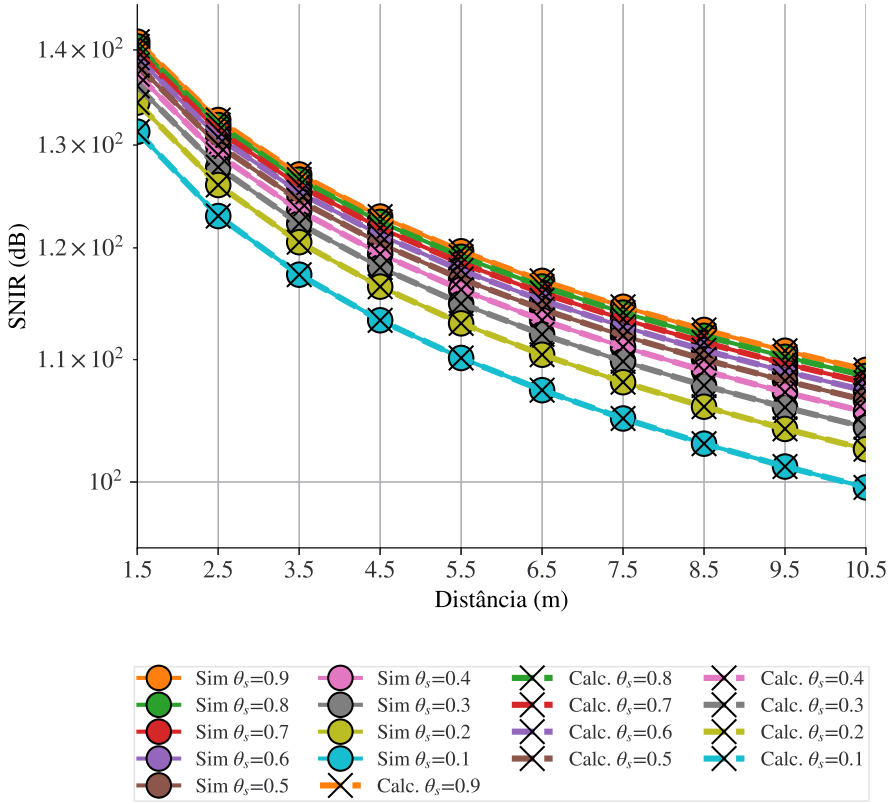
Moreover, it is important to mention that the results presented in this subsection resemble the outputs that could have been observed from a network with a deterministic behavior. This is attributed to the fact that contention for medium access in this scenario is minimum and the downlink traffic consists of a saturated flow of data sent from the AP to the node, resulting in a predictable and consistent data transfer pattern.

Furthermore, it is noteworthy to highlight a general behavior identified in our implementation, as observed from the simulation results, especially regarding both the lower and upper limit values of the θ_s range. When θ_s is set to 0, the power of received frames directed to the input of the information decoder falls below the ED threshold, rendering the energy harvesting mechanism ineffective. Conversely, setting θ_s to 1 results in all power of received frames being exclusively allocated to the information decoder, resulting in harvested energy levels equal to 0. Therefore, to ensure meaningful results, the values of 0 and 1 were intentionally excluded from the θ_s range, making the effective range for θ_s to fall between 0.1 and 0.9, which is used as input for the results presented in the following metrics.

6.1.2.1 Signal-to-Noise-plus-Interference Ratio (SNIR)

Fig. 6.2 provides a graphical representation of the simulated and calculated results related to the SNIR metric, expressed in dB. The SNIR curves exhibit a decreasing trend as the distance between the SWIPT-enabled node and the AP increases. The x-axis spans from 1.5 meters to 10.5 meters in 1-meter increments, outlining the varying distances. Additionally, for each distance increment, the curves correspond to systematically increased values of θ_s, ranging from 0.1 to 0.9, in 0.1 intervals.

Figure 6.2. SNIR Simulated vs Calculated Results



Source: Own authorship

As previously indicated, the levels of SNIR displayed in the graoh are calculated by the `InterferenceHelper` class using Eq. (5.1), as detailed in Chapter 5. Therefore, it is important to note that the input value of $N_i(k,t)$, which represents the cumulative effect of additive interference power in Eq. (5.1) is assumed to be zero in this specific scenario.

Consequently, the substantial SNIR levels displayed on the graph of Fig. 6.2, can be attributed to the power levels of received frames, which resulted from the propagation over short distances using the path loss values obtained from the macro deployment propagation loss model, as displayed on Fig. 2.8, and noise floor levels, N_f, as low as -113.98 dBm, determined according to Eq. (2.11).

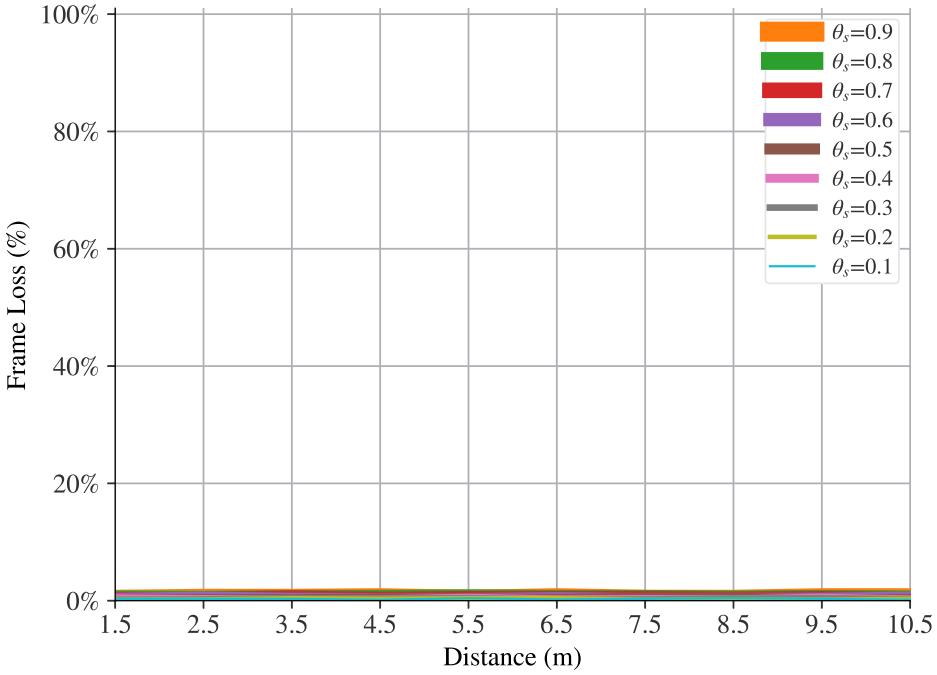
Hence, by assuming that the results obtained from the simulation of this scenario closely approximates to the case of a network with deterministic behaviour due to the reasons previously described, we can also employ Eq. (5.1) to perform the theoretical calculations of SNIR to

validate the results of this metric. Further conclusions regarding these results are elaborated in Chapter 7.

6.1.2.2 Frame Loss

Fig. 6.3 illustrates a graph depicting frame loss percentages. The outcomes are represented by flat lines parallel to the x-axis, with each line corresponding to a different value of θ_s within the specified range. The x-axis displays the predefined distance values.

Figure 6.3. Frame Loss %



Source: Own authorship

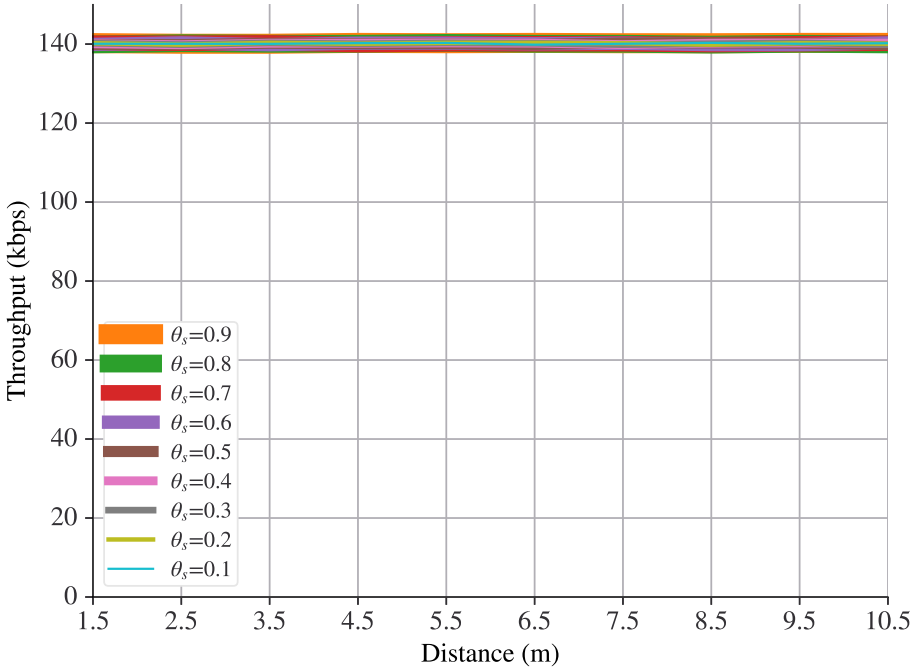
It is noteworthy that the results displayed in this graph, represent a zero percentage of frame loss, even when only 10% of the power of incoming frames is allocated to the input of the information decoder, with θ_s set to 0.1.

Furthermore, this illustration highlights that as the channel usage approaches full capacity, due to the saturated traffic flow, the frame loss remains consistently at zero percent throughout the graph. This behaviour can be attributed, in part, to the substantial levels of SNIR, as discussed earlier.

6.1.2.3 Throughput

The graph of Fig. 6.4 presents the results related to the throughput metric, in kbps. The horizontal lines displayed on the graph represent the throughput results for each specific value of θ_s. Whereas, the x-axis represents the predefined range of values for the distance parameter.

Figure 6.4. Throughput



Source: Own authorship

First, it is important to highlight that the results displayed on the graph represent a consistent pattern, where, regardless of the value set for θ_s, there is still a uniform superposition of parallel flat lines indicating the maximum throughput level has been reached.

Also, these results closely approximate to the transmission data rate parameter outlined in Table 6.1, which corresponds to the intended setup of a saturated channel. This close approximation is attributed to the existing method implemented for throughput calculation within the NS-3 library, which primarily considers the payload size of the successfully received UDP packets, as indicated in Table 6.1, while overlooking the UDP header and the additional overhead introduced by lower-layer protocols, as expresses in Eq. (6.3).

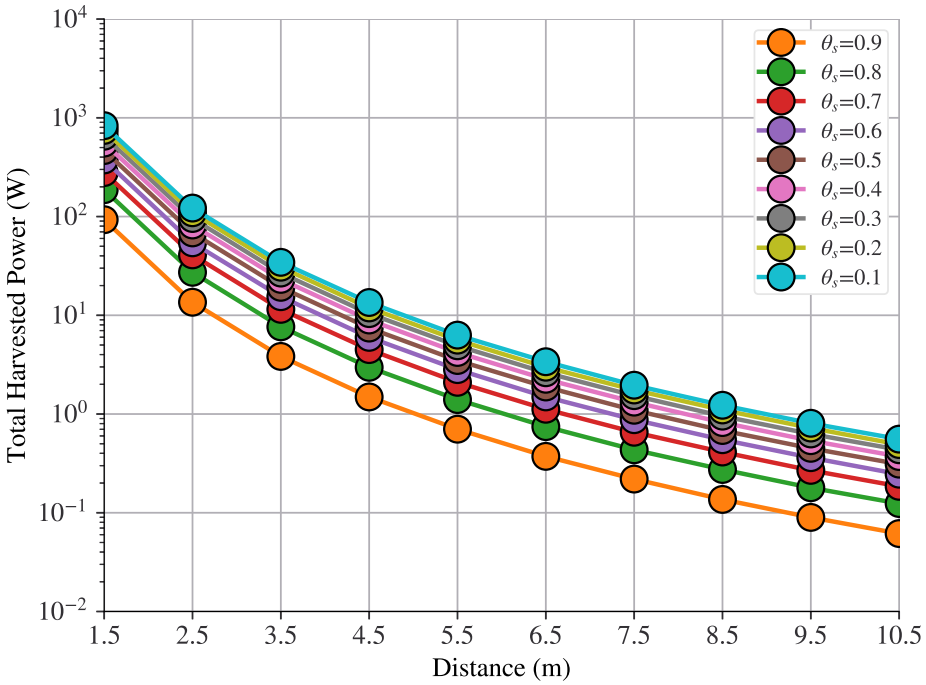
However, the outcomes obtained for this metric indicate that our implementation can behave

in a consistent manner withstanding a data rate close to the maximum channel capacity across the ranges of the specified input parameters, while carrying out other tasks related to energy harvesting, which will be analyzed in the following sections.

6.1.2.4 Total Power Harvested

The graph depicted in Fig. 6.5 showcases the cumulative harvested power results over the simulation interval, expressed in Watts. The results are presented in relation to the distance between the node and the AP, shown on the x-axis. Additionally, each curve in the graph corresponds to a specific value of θ_s within the predefined range, which decay as the distance of the SWIPT-enabled node and the AP increases, providing a comprehensive representation of the relationship between total harvested power, distance, and θ_s values.

Figure 6.5. Total Power Harvested



Source: Own authorship

Initially, from the results presented in this graph, we can observe an inverse relationship between the total power harvested and the values of θ_s. This can be attributed to the fact that the counterpart of θ_s, which is $(1 - \theta_s)$, is the relevant variable in the context of the energy harvester procedures, since it is used to determine the most of the results obtained from its

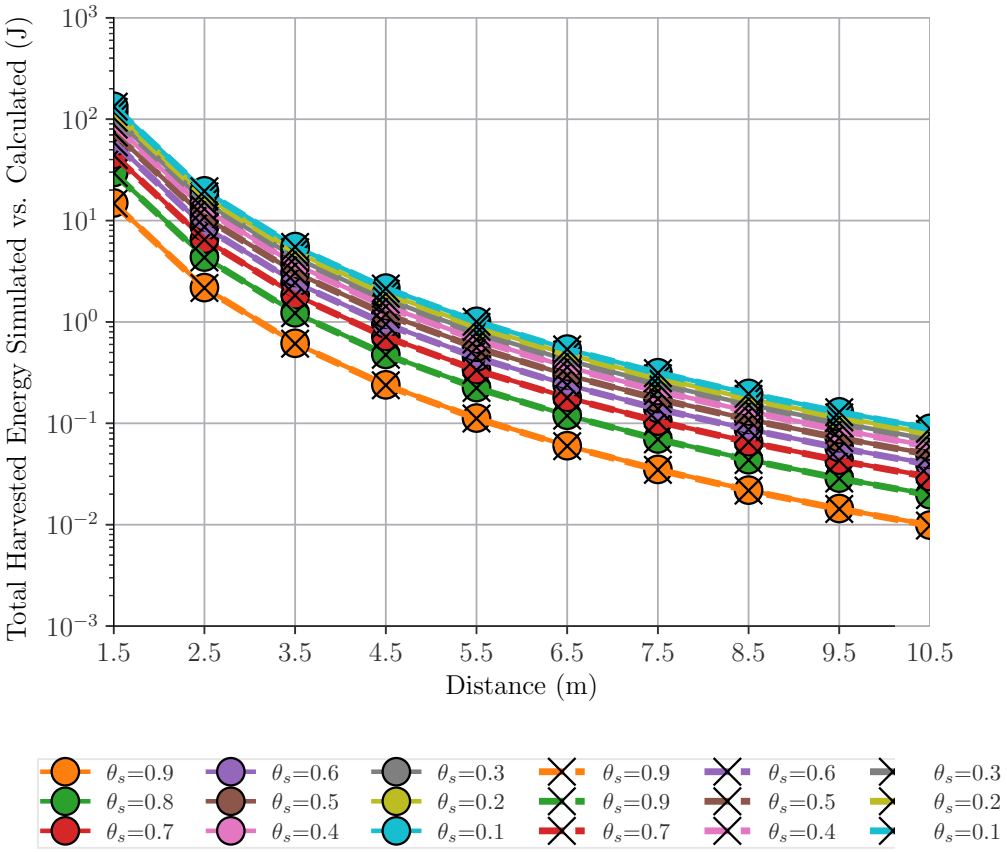
operations, as exemplified in Eq. (4.1).

Furthermore, the significant levels of total harvested power shown in this graph serve to characterize the behavior of the SWIPT harvester. It represents the ability of the SWIPT harvester to cumulatively extract the total power amount depicted in the graph by the conclusion of the simulation interval. Additionally, this result acts as a stock variable sink from a trace source attribute of the `SwiptHarvester` class. This variable can be employed to trace the behavior of the SWIPT harvester in real-time during the simulation.

6.1.2.5 Total Energy Harvested

Similarly to the results of the previous metric, the curves depicted in the graph of Fig. 6.6 correspond, from top to bottom, to the simulated and calculated results of total energy harvested for each value of θ_s ranging from 0.1 to 0.9, respectively. Each curve illustrates the decline of the total energy harvested, expressed in Joules, as the distance between the node and the AP increases in the x-axis.

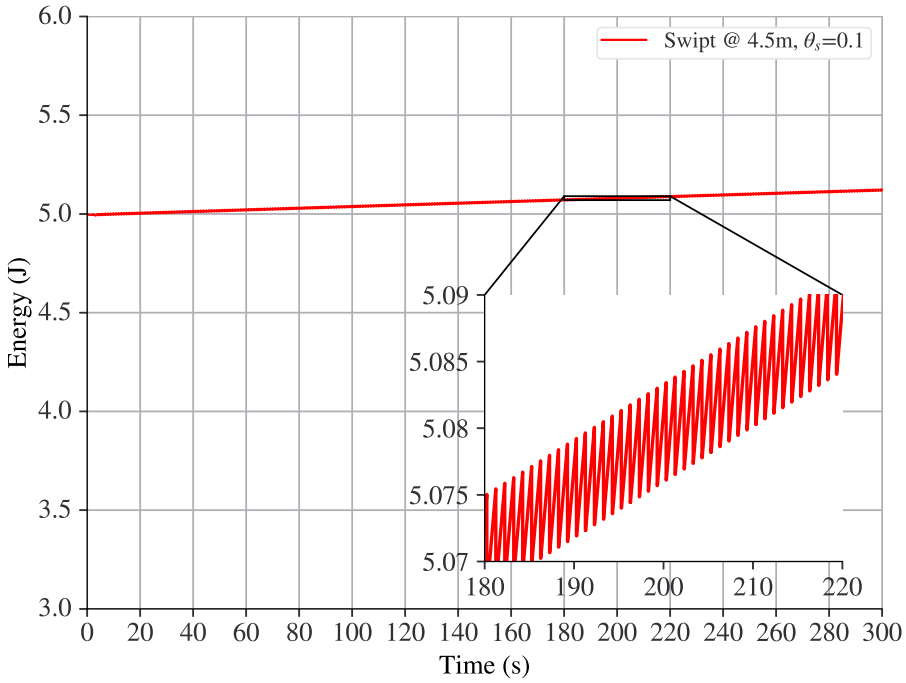
Figure 6.6. Total Harvested Energy Simulated vs. Calculated Results



Source: Own authorship

Firstly, it is important to note that each increment of energy added to the cumulative levels of total energy harvested presented in the graph of Fig. 6.6, corresponds to a level of energy that the SWIPT harvester was able to extract from the portion of the power of the incoming frame allocated to its input. Therefore, in practical terms, these increments of energy correspond to the energy level the SWIPT harvester provides, at each RX state, to recharge the battery. These recharging energy levels associated with the levels of energy expenditure at each state compose periodic cycles of charging and discharging that characterize the fluctuations of the remaining energy levels of the battery, as depicted in Fig. 6.7.

Figure 6.7. Battery Energy Levels Fluctuations During Simulation



Source: Own authorship

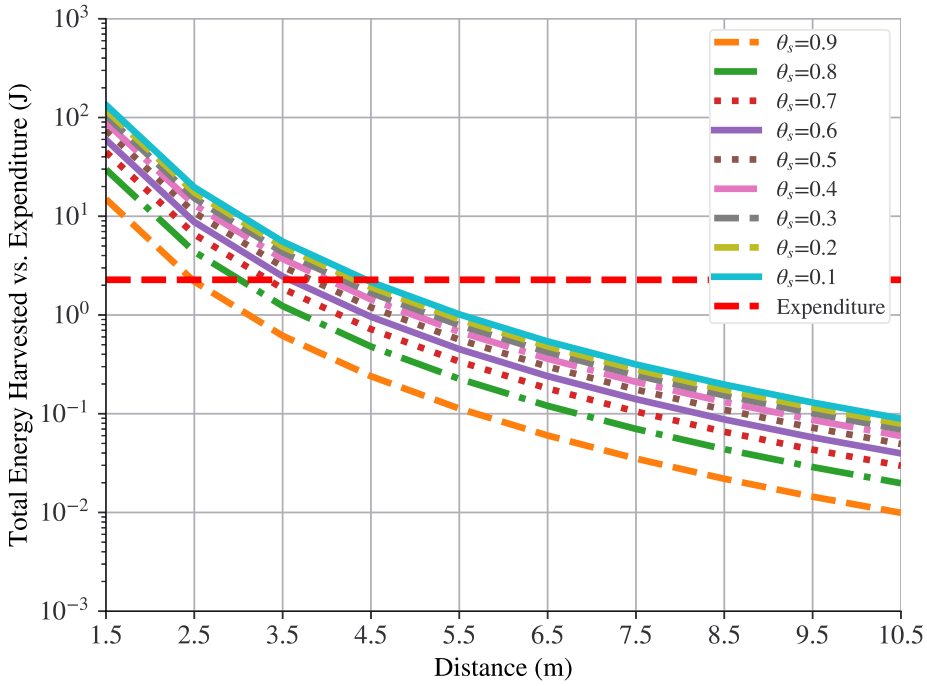
The depiction in Fig. 6.7 illustrates the remaining energy levels of the battery of an SWIPT-enabled node over time. In this graph, the SWIPT-enabled node is assumed to be located 4.5 meters away from the AP, with a θ_s value set to 0.1. The overall view of the graph shows a rising trend in the line representing the remaining energy level of the battery. It starts at an initial value of 5 J at $t = 0$ and, at 300s, displays a remaining energy level of approximately 5.1 J. The zoomed-in view of the inclined line highlights the periodic cycles of charging and discharging of the battery, as mentioned earlier.

Moreover, within this scenario, it is reasonable to assume that the results obtained for this metric can be also approximated to the outputs that could have been obtained from a system with a deterministic behaviour. As such, we can deploy Eq. (4.3) to calculate the results displayed in the graph of Fig. 6.6. Therefore, the simulated and calculated curves of total harvested energy displayed in this graph closely superpose one another, indicating a close approximation between the two methods, thus validating the results obtained from simulations.

6.1.2.6 Sustainability

The graph depicted in Fig. 6.8 conveys the same data as presented in the results of the previous metric. Furthermore, it introduces a noteworthy addition of a horizontal red dashed line that symbolizes the energy consumption of the SWIPT-enabled node. Also, the distance and θ_s parameters are used as inputs each with its predefined values.

Figure 6.8. Sustainability



Source: Own authorship

Therefore, by examining the graph in Fig. 6.8, we can notice that the points where the red dashed line intersects the curves of total energy harvested represent the break-even points.

These break-even points are consistent with the model outlined in Eq. (4.9) and signify that the total energy harvested, used for recharging the battery, is equals or exceeds the total energy consumption of the SWIPT-enabled node.

Additionally, it is important to highlight that the curves corresponding to lower values of θ_s have break-even points situated at distances farther from the origin of the graph. Conversely, curves associated with higher θ_s values exhibit break-even points closer to the origin, indicating an overall reduction in the equilibrium distance.

Also,it is important to note that the SWIPT-enabled node spends most of its time in the RX state, actively receiving the saturated UDP traffic flow. The RX state ranks second-highest in terms of energy consumption. Conversely, the SWIPT-enabled node seldom enters the SLEEP state, where the energy expenditure could be reduced, potentially shifting the break-even point to the left, at distances greater than 4.5 meters away from the AP. These considerations will be further detailed in Chapter 7.

6.2 IEEE 802.11AH NETWORK SCENARIO

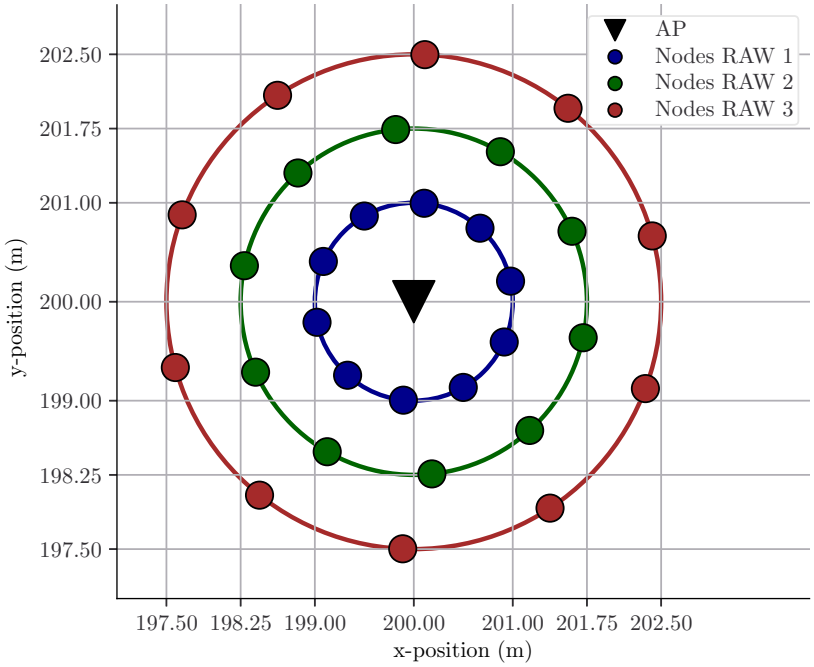
The design of this second scenario aimed to provide a setup to enable the characterization of our implementation and to identify potential constraints that could impact both network performance and the energy harvesting capacity of our SWIPT implementation. In this network scenario, nodes are configured to request authentication from the AP before gaining access to network resources.

Moreover, nodes in this setting must compete with their peers for access to the wireless channel. In this scenario, the existence of interfering frames could strain limited resources, possibly causing elevated energy consumption due to frame retransmissions. Consequently, simulating our implementation under these conditions becomes relevant for understanding its behavior in a network facing such challenges. This allows us to characterize its ability to sustain energy levels for the operations performed by network nodes and to identify potential constraints.

6.2.1 Simulation Setup

To replicate such an environment, we deployed an IEEE 802.11ah network architecture comprising a single AP and 30 SWIPT-enabled nodes. These SWIPT-enabled nodes are arranged into three RAW groups, each comprising 10 nodes. The RAW groups were strategically positioned in three concentric circles surrounding the AP with radii of 1m, 1.75m, and 2.5m, creating the geometric configuration depicted in Fig. 6.9. This arrangement was designed to ensure uniform conditions for all nodes within the same RAW group.

Figure 6.9. Network Topology of the Network Scenario



Source: Own authorship

The composition of these RAW groups are as follows: The first group encompasses nodes with ID numbers ranging from 0 to 9, located along the inner edge of a circle with a radius of 1 meter. The second group consists of nodes with ID numbers ranging from 10 to 19, positioned along the middle edge of a circle with a radius of 1.75 meters. Finally, the third group comprises nodes with IDs ranging from 20 to 29, strategically placed along the outer edge of a circle with a radius of 2.5 meters.

The AP was configured to broadcast beacon frames at each interval of 102400 μs, as specified

in Table 6.2. Also, using the parameters defined in Chapter 2 related to the RAW Mechanism, the N_{RAW} parameter of Eq. (2.8) was set to 3 RAW slots. Each RAW slot was assigned to the nodes within each specific RAW group. Then, the C_{SLOT} parameter of Eq. (2.7) was set to the value of 280, resulting in the value of D_{SLOT} equals to 33605 μs. Accounting for the overall value of D_{RAW} being set to 100815 μs, which is less than the beacon interval. The configuration of other key parameters employed for simulating this network scenario is provided in Table 6.2.

However, as mentioned at the beginning of this Chapter, certain parameters that were originally used in the first scenario had to be replaced to better align with the simulation requirements of the second scenario. These parameters encompass the following

1. Propagation loss model,
2. Data mode and control mode,
3. Gain of the AP antenna,
4. UDP payload size, and
5. UDP packet interval.

The propagation loss model has been transitioned from the macro deployment type to the pico or hot zone propagation loss model. This change was made to align with the environmental characteristics of the network architecture of the current scenario, as described in (BELLEKENS *et al.*, 2017). Also, the default value for antenna height assumes a rooftop placement, which further justifies this adjustment.

The adjustment made in the data mode parameter, increased the bit rate of transmitted data frames from 150 kbps to 600 kbps. This, in turn, allows data frames to be sent at a faster speeds, reducing time the PHY machine state would have to remain in the TX state. Consequently, this modification effectively reduced the energy consumption of the nodes when compared to the previous 150 kbps setting.

Moreover, recent advancements in antenna manufacturing have facilitated the modification of the antenna gain parameter for AP. These innovations have enabled the development of Wireless Local Area Network (WLAN) antennas operating at 900 MHz with a gain of 13 dBi, as exemplified in (L-COM, 2021). Consequently, this adjustment allows for the transmission of

frames at higher power levels, resulting in increased power levels to be allocated to both the information decoder and the energy harvester.

In order to harmonize the UDP packet payload size with the size commonly used in real IoT networks, a modification was introduced in this parameter. Specifically, the payload size of 64 bytes was selected in accordance with the work of Tian *et al.* (2017). This adjustment supports running simulations mirroring the characteristics of real IoT network traffic.

Furthermore, the UDP transmission interval was also adjusted to better reflect the characteristics of IoT applications where nodes send packets sporadically. As a result, the value of 3.072 seconds was adopted, which is significantly lower than the values suggested in the work of Bel *et al.* (2018). Nonetheless, this modification aligns with the event-driven nature of many IoT data transmissions.

Table 6.2. Network Configuration Parameters

Parameter	**Node**	**Access Point**
Radio Frequency	900 MHz	900 MHz
Beacon interval	-	102400μs
Beacon Frame Duration	-	2800μs
Transmission Power	0 dBm	30 dBm
Antenna Gain	6 dBi	13 dBi
Short Guard Enabled	False	False
Channel Width	1 MHz	1 MHz
Energy Detection Threshold	-110 dBm	-110 dBm
CCA Mode1 Threshold	-107 dBm	-107 dBm
LDPC Enabled	False	False
S1G Short Field Enabled	False	False
S1G 1M Field Enabled	True	True
Noise Figure	6.8	6.8
Propagation Loss Model	Outdoor Pico	Outdoor Pico
Data Mode	600 kbps	600 kbps
Control Mode	300 kbps	300 kbps
PLCP Preamble and Header TX Rate	150 kbps	150 kbps
UDP payload size	64 bytes	–
UDP packet interval	3.072 s	–

Source: Own authorship

6.2.2 Results

In this subsection, we showcase the outcomes derived from extensively running simulations of our implementation within the IEEE 802.11ah network scenario. The results are structured around five of the seven previously mentioned metrics, including: UDP packet loss, throughput, total power harvested, total energy harvested, and sustainability. The results obtained for these metrics are presented in terms of their average values. Hence, we performed simulations for our implementation in the current scenario at least 10 times. Each simulation extended over 4800-second intervals, utilizing unique pseudo-random seed values for statistical relevance. Following these simulations, we computed the averages of the results and presented them in the subsequent sections.

It is important to highlight that, in this scenario, the focus has shifted from the frame loss metric to the UDP packet loss metric. This deliberate modification was introduced to enable a more thorough characterization of our implementation specific to this network scenario.

Before proceeding, it is relevant to note that we could not validate the following results using other methods, such as an analytical model. This limitation stems from the absence of suitable models that could verify the outcomes presented here. Developing such models was not within the scope of our current research objectives, as it would require a significant investment of time and resources.

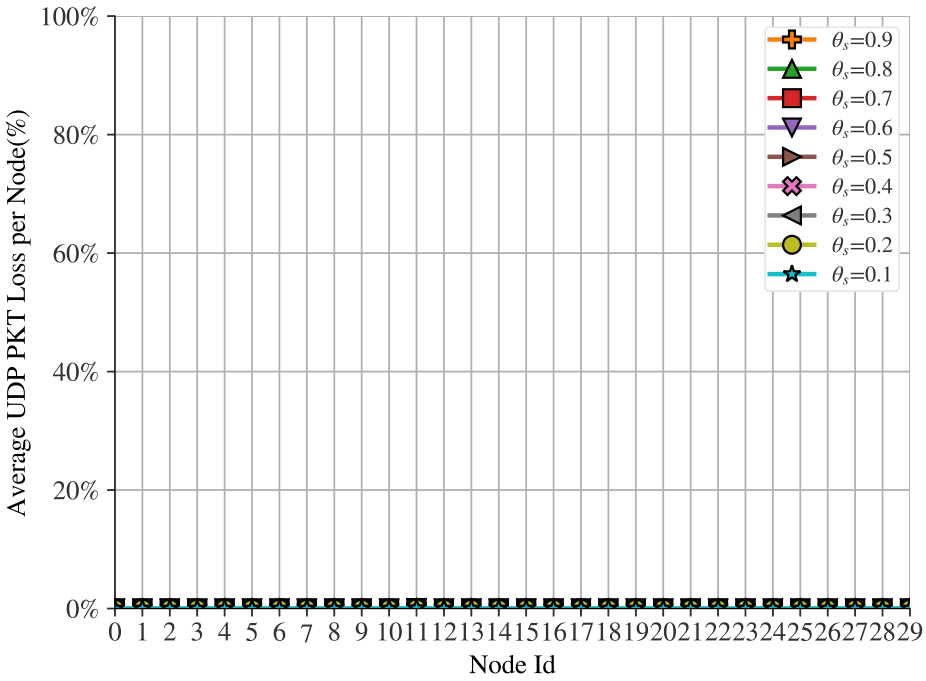
Also, to the best of our knowledge, based on our thorough survey to evaluate the current state of scientific knowledge concerning the simulation of SWIPT on computational platforms, as detailed in Chapter 3, no information was found regarding research dedicated to the development of analytical models in this context.

Nevertheless, as this scenario represents an evolution in our research compared to the initial phase, the results presented here are in line with our expectations, providing a generally sound basis for comparison with the outcomes observed in the first scenario. This suggests that the results may be reasonably considered correct.

6.2.2.1 Average UDP Packet Loss

The graph displayed in Fig. 6.10 illustrates the average percentage of UDP packet loss for each node in the network, organized according to the predefined distance for each RAW group and using the specified range of values for θ_s.

Figure 6.10. Average UDP Packet Loss %



Source: Own authorship

Firstly, a relevant aspect that may impact the behavior of our implementation, as characterized by the adoption of this metric, is its ability to eliminate the transient period when nodes are not yet associated with the AP. This phase, characterizes our implementation as the phase nodes experience the highest energy expenditure, as they cannot access network resources such as entering into SLEEP mode.

In this context, the ability to transmit and receive UDP packets implies that nodes are associated with the AP. This is precisely why we intentionally chose UDP packet loss as the initial metric for characterizing the behavior of our implementation, rather than frame loss. The rationale is that, during the non-associated period, the behavior of the nodes is beyond the control of our configuration script and may exhibit variations based on the network architecture,

potentially leading to unexpected results. Consequently, the frame loss metric does not align with the anticipated response of our intended setup.

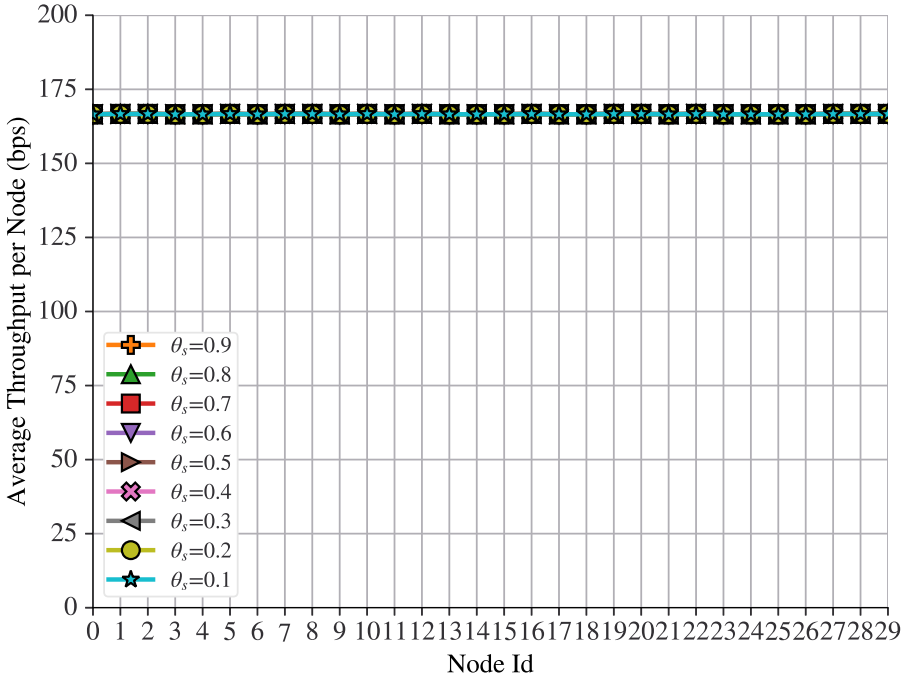
Moreover, in order to examine the behaviour of our implementation behind this metric, we monitored the packets received by the UDP server deployed at the AP. These packets were transmitted from UDP clients installed at each node of the network, operating within their designated RAW group, where they had to contended for wireless channel access during their assigned RAW slot, at the specified UDP packet interval for transmissions. Then, at the end of the simulation period, we examined the number of UDP packets received by the UDP server, matching them to the MAC address of each node. These readings were then compared to the number of UDP packets transmitted from each UDP client.

Considering that the assignment of the MAC address to the nodes in this scenario is statically configured and the number of received UDP packets filtered at the AP provides an exact correspondence to the number of UDP packets transmitted by each node, resulting in the superposition set of parallel flat lines displayed on the graph of Fig. 6.10 at the level 0% UDP packet loss, with each line corresponding to a specific value of θ_s within the predefined range of values, suggests to us that regarding the characterization of our implementation through the results obtained using the UDP packet loss metric, our implementation has performed without producing any negative impacts over network performance.

6.2.2.2 Average Throughput

Fig. 6.11 displays the results of the average throughput obtained from the simulation of our implementation within this network scenario, using as inputs the predefined range of values for distance and θ_s, where each node in the network is represented with its respective ID number in the x-axis and the y-axis corresponds to their respective level of average throughput, expressed in bits per second (bps).

Figure 6.11. Average Throughput



Source: Own authorship

Despite not reaching expressive levels of throughput, as observed in the results of this metric in the previous scenario, the average throughput displayed in Fig. 6.11 align with the anticipated behavior of nodes operating according to the intended setup, as previously described.

The results are characterized by a superposition of flat lines, where each line corresponding to a specific value of θ_s within the predefined range, indicating that the throughput remains consistently at its maximum level for all nodes across all RAW groups, which are assigned to different RAW slots in time, but all with the same duration.

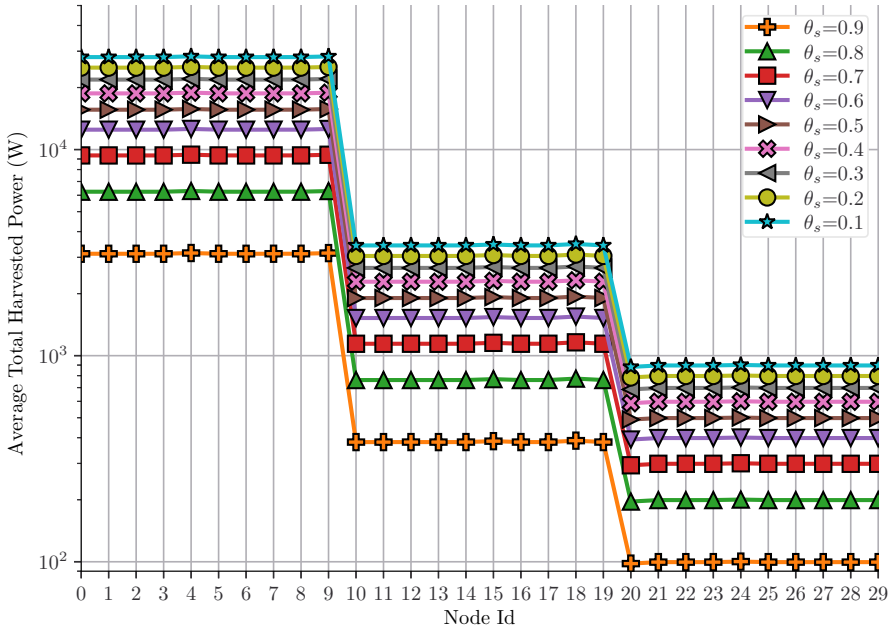
From these results, we can also deduce that our implementation does not produce any significant impacts on network performance. Had our implementation exhibited any constraints, we would have expected to notice fluctuations in the the results displayed in this graph.

6.2.2.3 Average Total Harvested Power

In Fig. 6.12, a graph is presented showcasing the results of the average total harvested power for each node within the network. These results are organized according to each value within

the range of θ_s and correspond to each node of the network individually, since their ID numbers are represented in the x-axis.

Figure 6.12. Average Total Harvested Power



Source: Own authorship

As we can observe from this graph, the values of the average total power harvested exhibit a stepwise decline for the nodes of each RAW group and for each value of θ_s accordingly. Where, the nodes of the first RAW group, positioned at the innermost edge of the concentric circle surrounding the AP, achieve the highest levels of average total power harvested. This can be attributed to the shorter distances, where path loss has a lesser impact on transmitted power levels, consequently leading to higher power levels to be received by these nodes.

Following this pattern, the nodes of the second RAW group, located at the edge of the circle in the middle, obtain lower levels of average total harvested power as compared to the first group. This is primarily attributed to the higher path loss, impacting the transmitted power levels and leading to a decrease in the power received by the nodes, as compared to the nodes in the first group.

Likewise, the nodes within the third group consistently exhibit the lowest levels of average total power harvested. This phenomenon can be attributed to the the higher levels of path loss

experienced in this outermost group, which results in the lowest levels of scavenged power by these nodes.

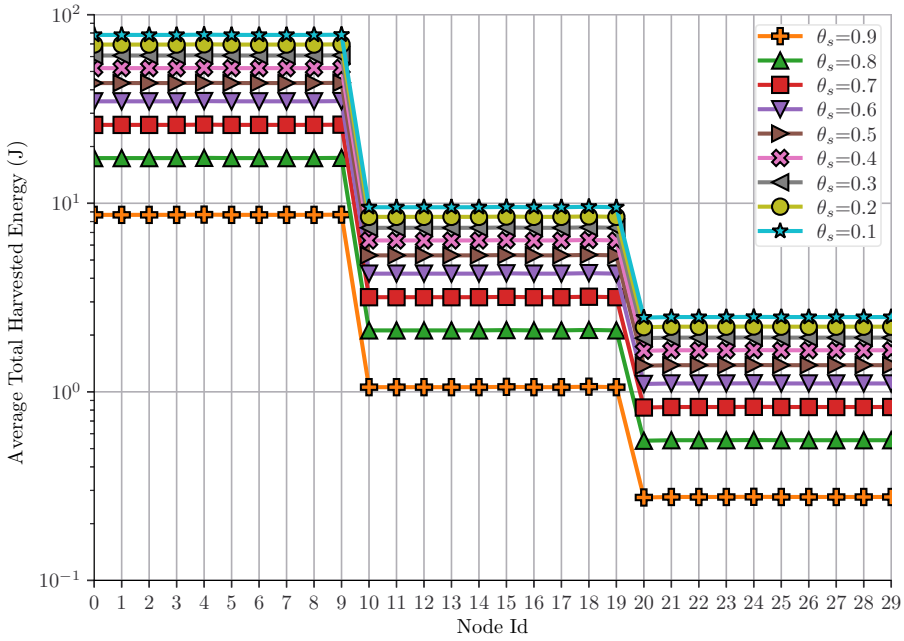
It must be noticed that in this configuration the average total harvested power levels displayed on the graph are the outcome of the power harvested from beacon frames and acknowledgement responses sent from the AP to nodes.

At this point, these results serve as an indication that, thus far, none of the outcomes from the previous metrics have shown any significant negative impacts resulting from our implementation. This absence of noticeable constraints may be attributed to the fact that, through the observation of the previously displayed results, we have not identified any substantial factors causing discernible negative impacts on the behavior of our implementation.

6.2.2.4 Average Total Harvested Energy

Fig. 6.13 presents a graph displaying the results of the average total harvested energy for each node in the network. The results are categorized based on the values within the θ_s range.

Figure 6.13. Average Total Harvested Energy



Source: Own authorship

The same stepwise decline observed in the average total harvested power, is depicted in this

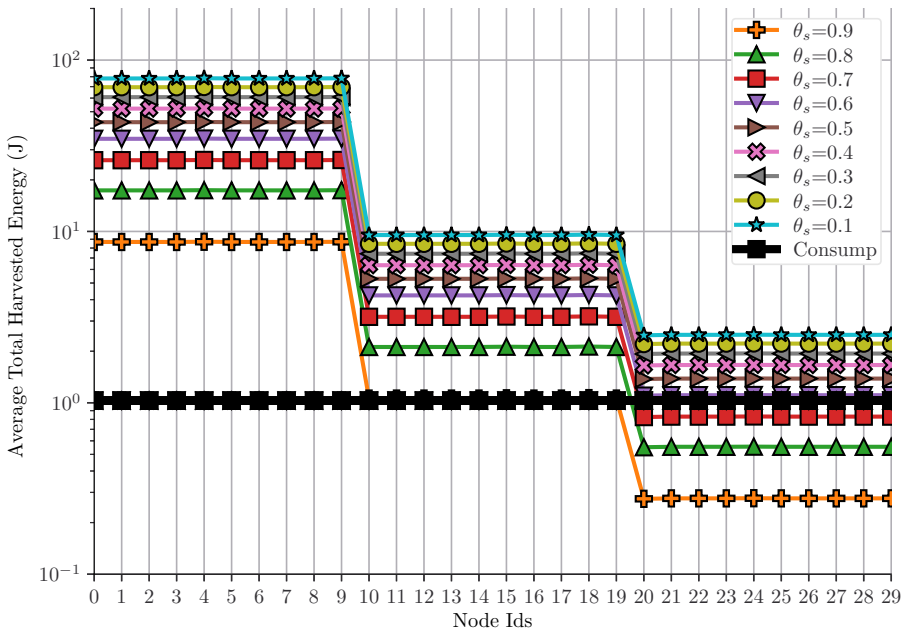
graph, corresponding to the same effects described earlier. This pattern can be attributed to the fact that the harvested energy level modeled in Eq. (4.3) is derived from Eq. (4.1). As a result, Fig. 6.13 exhibits a similar pattern in response to changes in θ_s and RAW group placement.

Furthermore, it must be noticed that these results correspond to the average total levels of energy used for recharging the battery, which encompass the overall effects introduced by a wide range of parameters, specially those related to η and β.

6.2.2.5 Sustainability

In Fig. 6.14, we provide a graph that displays the results of the average total harvested energy for each node in the network, similar to the previous graph. Additionally, this graph incorporates the average energy expenditure level of each node, represented by the black line.

Figure 6.14. Network Sustainability



Source: Own authorship

As previously mentioned, this metric allows the characterization of the sustainability attribute of our implementation, which, in other words, describes the behaviour of our implementation in performing the task of enabling devices to achieve self-sufficiency in energy provision for their

network operations.

The sustainability in this scenario can be verified by analyzing the position of the break-even points displayed in the graph depicted in Fig. 6.14. These points are located in the region where the black line, which represent the energy consumption of the nodes, intersects or remains bellow the stepwise declining lines representing the average total harvested energy.

Therefore, from the illustration of this graph, we can observe that the black line remains at the constant level of approximately $1J$ across the whole range of the values of θ_s for all nodes in the network. This signifies that disregard the value that θ_s might assume or the location of the nodes within the specified scenario, the values of energy expenditure of all nodes remains at constant level throughout the graph. This consistency serves as a representation of how our implementation behaves under the conditions outlined in this scenario.

The results of this metric regarding the nodes belonging to the first RAW group reveals that, the levels of the average total harvested energy of each stepwise declining lines are all above the black line. This means that, under this condition and configuration, our implementation may allow devices to become self-sufficient in terms of the amount of energy required for their deployment, and to also provide energy for other nodes.

A seconds analysis of the results based on this metric, with respect to the nodes that constitute the second RAW group, reveals that for the upper bound of the θ_s range, which correspond to 10% of the received power of incoming frames allocated for energy harvesting purposes, our implementation may allow devices deployed under these same conditions to achieve the break-even point for all values within the θ_s range. Specifically, for $\theta_s \leq 0.8$ our implementation may allow devices to receive a surplus of energy, similar to the nodes of the first group, but in a lesser degree.

A third analysis of the results obtained for this metric, considering the nodes of the third RAW group, displays that our implementation may allow the nodes of the third RAW group to achieve energy sustainability when $\theta_s \leq 0.6$. For nodes that may deploy other values θ_s the sustainability cannot be confirmed.

Another suggestion derived from the results obtained for this metric, indicates the investigation of a forth RAW group, which could be introduced by comprising 10 nodes, positioned

along a farther distant edge with a radius of 3.25 meters. In this hypothetical implementation we empirically estimate that the value of θ_s set to 0.1, may lead to achieving sustainability. A successful investigation of this hypothesis may lead to an alignment of the values of the distance parameter used in this second scenario to be approximated to the values of distance where sustainability was achieved in the first scenario.

6.3 SUMMARY

In this chapter, we have presented the results obtained from the extensive simulations of our implementation within two distinct network scenarios. The first consisted of an IEEE 802.11ah single-link architecture and the second of an IEEE 802.11ah network architecture composed of one AP and 30 SWIPT-enabled nodes, structured according to the 3 RAW Groups and other correlated mechanisms settings.

In the first scenario, we initially presented a simplified overview of our single-link SWIPT implementation in a paper for the XLI Brazilian Symposium on Computer Networks and Distributed Systems. This chapter expands on our research, offering a more comprehensive presentation of simulation results across various metrics, such as SNR, frame loss percentage, throughput, total harvested power, total harvested energy, and sustainability. The findings indicate that our implementation exhibit no significant constraints affecting network performance or the capabilities of the SWIPT harvester. Notably, certain input values introduced complexity in graphical representation and increased computational complexity in the simulation scripts, especially within the range $10^{-10} \leq \theta_s < 0.1$.

Subsequently, we advanced our research by investigating our implementation in a more challenging scenario, necessitating the adaptation of parameters from the first scenario to better represent real-world conditions. The results revealed distinctive behavior, particularly highlighting the transient period when network nodes seek association with the AP. This phase incurs the highest energy consumption as nodes cannot enter the SLEEP state, and the RAW mechanism settings do not reduce their collision domain. Notably, for the defined lower and upper bounds of the θ_s range, the observed behavior produced outputs deemed irrelevant for our investigation within the outlined metrics thus far.

Currently, our primary focus is set on finalizing the remaining details for the submission of a second research paper to major conferences of the field. And, for the future we set our sights to expand our research even further. The details about our future work will be presented in Chapter 7.

CHAPTER 7

CONCLUSIONS

In this dissertation, we presented the results of a comprehensive research we conducted with the primary objective of developing an SWIPT receiver object model with a split architecture, designed specifically for the power splitting technique, to be deployed over the IEEE 802.11ah NS-3 Simulation Module. Our investigation also focused on achieving a seamless integration of our SWIPT implementation into the NS-3 library. Additionally, we aimed to thoroughly characterize the behavior of our SWIPT implementation through extensive simulation runs across two distinct scenarios. These simulations intended to identify constraints that could potentially produce negative impact over the performance of the network or limit the capacity of the SWIPT harvester for harvesting energy.

During the preliminary stage of our research, we conducted a thorough survey in order to identify the current state of scientific research towards the subject of developing an SWIPT object model over computational platforms devoted for the purpose of enabling simulation of this technology within IEEE 802.11 standard networks, as described in Chapter 3. To the best of our knowledge, we have not identified any studies directly related to the primary objective of our research. The results of this survey are succinctly summarized in Table 3.1.

Following that, in the subsequent phase of our research, we directed our focus towards a comprehensive investigation of the essential classes related to the development of our SWIPT object model. In this endeavour we have dissected the major components of the key classes to uncover their operating principles, related attributes responsible for modelling the major features, and dependencies with other relevant classes within the context of the NS-3 library model. As a consequence of this thorough investigation, we acquired the necessary expertise that enable us to build our own model, as detailed in Chapter 5.

Further, we extensively simulated our implementation in two specific scenarios in order to characterize the behavior of our SWIPT object model and identify constraints that could

potentially limit network performance or negatively impact the operations of the SWIPT harvester. Therefore, in the first scenario, which corresponds to the first achievement of our research, we presented a paper describing our implementation of a single-link scenario, in a simplified overview, to the Extended Proceedings of the XLI Brazilian Symposium on Computer Networks and Distributed Systems. However, in Chapter 6 we provided a more comprehensive exploration of the results obtained from the simulation of this scenario, according to its relevant metrics, including: SNR, percentage of frame loss, throughput, total harvested power, total harvested energy and sustainability. All the results indicated that our implementation may not present thus far any significant constraints that could negatively impact either the network performance nor the capabilities of the SWIPT harvester. Also, in this context the results obtained from simulations, they allowed us to characterize the behaviour of our implementation according to the predefined range of inputs. Specifically, in the case of the input values where, $10^{-10} \leq \theta_s < 0.1$, we observed that this range of values did not yield proportionate benefits in terms of the total harvested energy magnitude as a system output, since this range only allows for marginal power increments at the input of the SWIPT harvester. Consequently, we have concluded that, for the purpose of simplifying the computational complexity of the simulation process and achieving significant improvements in terms of the SWIPT harvester output, the improved parameter range for θ_s, thus far and as long as the relevant metrics of this scenario are concerned, falls between 0 and 1.0.

Regarding the results related to the SNIR metric, which yielded significant levels of the SNIR presented in first scenario, they can be rationalized by the environmental characteristics inherent by the macro propagation loss model deployed, which suggests its usage in applications, such as rural areas, remote industrial plants, or critical mission applications involving sensor deployment in hard-to-access locations like oil extraction platforms and energy transmission towers. In these specific settings, it is presumed that adjacent and co-channel radio frequency interference is either nonexistent or effectively suppressed, thereby leaving thermal noise and non-idealities of the electronic components of the receiver circuitry as the primary sources of noise experienced at the receiver. Additionally, the frames transmitted from the AP may reach Effective Isotropic Radiated Power (EIRP) levels of up to 36 dBm. These levels of EIRP coupled with the low path loss derived from the macro deployment loss model significantly contribute to the substantial SNIR levels attained.

As a consequence, these substantial levels of SNIR exert a profound impact on the percentage of frame loss. This was evidenced through the consistent results displayed in the results regarding the metric of frame loss, which remained at a base level of zero across the entire spectrum of distances and values of θ_s. This was visually represented in the graph depicted on Fig. 6.3, as a superimposed set of flat lines, each aligning on top of one another.

A similar outcome was observed in the results displayed for the throughput metric, where they closely approximate the transmission data mode and remain constant throughout the graph displayed on Fig. 6.4, spanning the entire range of distances and values of θ_s.

Regarding the adoption of the macro deployment propagation loss model as the default choice for determining path loss values in large outdoor scenarios, especially in situations where antenna deployment at a height of 15 meters above rooftops is feasible, as mentioned earlier, this model has played an important role in providing the results obtained for the sustainability metric. This is evidenced by the results displayed in the graph of Fig.6.8 where the average total harvested energy levels reach the break-even point at the distance of 4.5m, in the curve corresponding to the θ_s value of 0.1 and providing energy surplus for distances lower than 4.5m, encompassing higher values of θ_s.

In summary, the simulations conducted within the single-link scenario have yielded results that support the characterization of the described behavior of our implementation and provides no indication of major constraints that could impact the performance of the network or limit the SWIPT harvester capacity as long as the relevant metrics are concerned.

In the second simulation scenario, our focus transitioned towards investigating our implementation within a more challenging scenario, encompassing the network architecture described in Chapter 6. This required adapting some parameters specifically tailored for the first scenario to better resemble a real-use case of the second scenario. The behaviour our implementation exhibited through the obtained results allowed us to characterize some relevant attributes, such as: the transient period in which the network nodes request association with the AP represents the highest phase in energy consumption, since nodes cannot enter into the SLEEP state and neither have their collision domain reduced by the RAW mechanism settings. Specifically, for the lower and upper bounds previously defined of the θ_s range, we discovered that the behaviour does not lead to significant outputs concerning our implementation, until the present moment,

as far as the outlined metrics are concerned.

Also, in the adaptation phase of the parameters of the second scenario, we were compelled to find a more suitable propagation loss model to better represent the deployment of the network architecture of the second scenario, since the macro deployment propagation loss model seems to fits best applications that require an antenna placement at a height of 15 meters above rooftop. Therefore, in order to better capture the propagation loss in environments characterized by higher node density in close proximity to the AP,, the pico deployment propagation loss model may represented a more realistic option.

As a consequence of this modification, the power levels received by the SWIPT-enabled nodes experienced a substantial reduction, which ultimately resulted in the in lower levels of energy harvested. This effect also caused the break-even points of sustainability to be shifted closer towards the AP.

Nevertheless, when considering network performance, it is noteworthy that despite the considerable reduction resulted from larger intervals set between UDP packet transmission with smaller payload sizes, the results exhibited a consistent throughput levels for all nodes. This consistency persisted across the entire range of θ_s values, as illustrated by a set of flat lines perfectly aligned on top one another, as illustrated in the graph of Fig. 6.11.

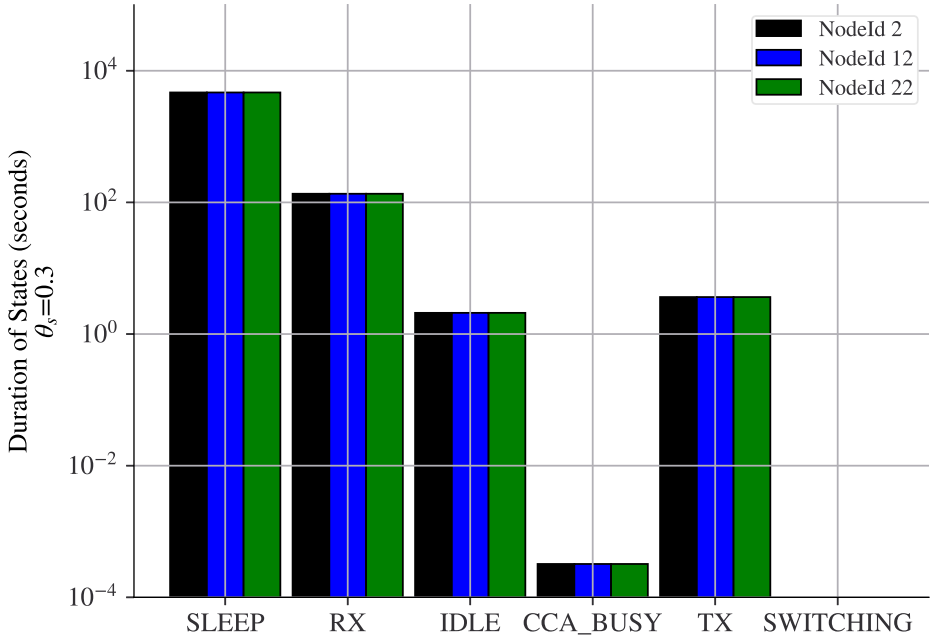
Similar findings extend to the metric related to the UDP packet loss, which is likewise depicted as a set of parallel flat lines for all nodes situated at varying distances and across the entire spectrum of θ_s values, as depicted in the graph of Fig. 6.10.

An important observation that demands our attention pertains to the average energy consumption levels of the nodes, as illustrated in Figure 6.14. The significance of this findings lies in the fact that if our simulations had resulted in frame collisions, necessitating retransmissions, or interference caused by hidden nodes, we would not have observed the consistent trend in the energy consumption levels.

However, for a more comprehensive conclusion of this energy consumption pattern regarding the second scenario, we describe the time allocation and energy consumption of nodes 2, 12, and 22. These nodes belong to distinct RAW groups, positioned at different distances, as previously described. Our short analysis encompassed only the relevant operating states of the PHY layer.

Our findings revealed that, in terms of average percentage of time spent, the PHY state machine allocates approximately: 97.15% of the time on SLEEP, 2.081% of the time on RX, 0.075% of the time on TX, 0.043% of the time on IDLE, 6.667^{-6} of the time on CCA_Busy, and 0% on Switching, as depicted on the graph of Fig. 7.1.

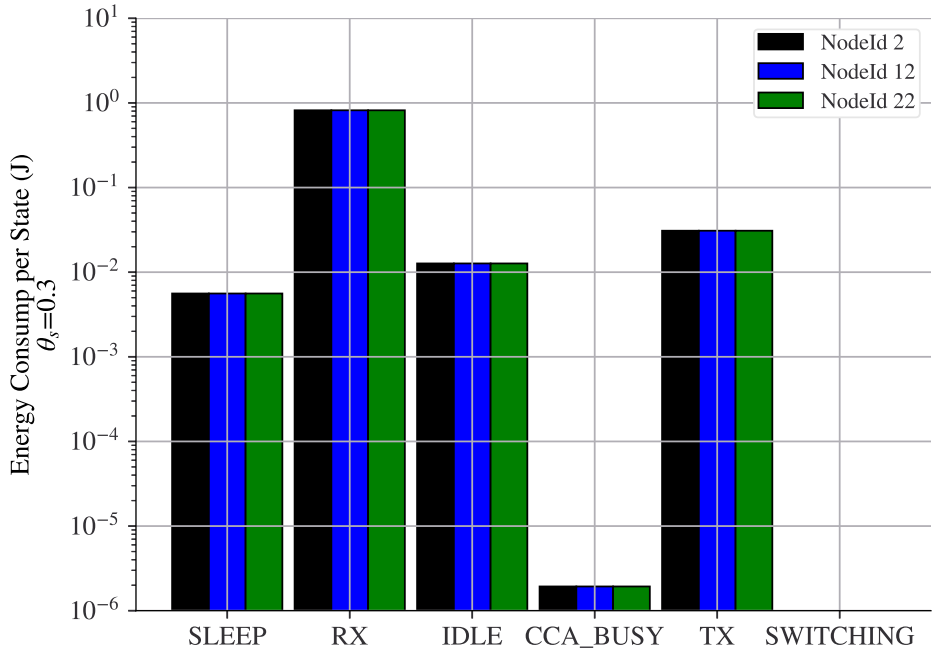
Figure 7.1. Average State Duration



Source: Own authorship

However, when considering the percentage of the average energy consumption, these states contributed to: 0.646% on SLEEP, 94.32% on RX, 3.56% on TX, 1.46% on IDLE, nearly 0% on CCA_Busy, and 0% on Switching, of the total energy expenditure of the node, as illustrated on Fig. 7.2.

Figure 7.2. Average State Energy Expenditure



Source: Own authorship

Hence, we can conclude that the peak energy consumption happens during the RX state, coinciding with the energy harvesting process. In this state, a significant portion of the received frames consists of beacon frames and acknowledgment responses originated from the AP. These frames serve primarily for dual purpose of controlling and managing the network. Nonetheless, in the context of our SWIPT implementation they also encompass the purpose of serving as a source of energy for all SWIPT-enabled nodes within the network.

Based on these findings, we can conclude that the PHY machine state exerts a notable impact on the sustainability of out implementation. As an illustrative example of this concept, the levels of harvested energy could potentially be enhanced by introducing an exceptionally low energy consumption state, such as the SLEEP state, which is described in the next Section.

The characterization of the behaviour of our implementation through all metrics used in both scenarios where our implementation has been thoroughly simulated enable us to conclude that our implementation has led us to achieve our primary objectives. This includes supporting the energetic sustainability of SWIPT-enabled nodes, thereby enabling future research to pursue

further advancements towards this subject.

7.1 FUTURE WORK

Based on the findings derived from our research, we have identified several avenues for improving our implementation, which are outlined below.

Initially, our intention is to explore appropriate methods that could aid in the validation of the results presented in Chapter 6. As previously mentioned, to the best of our knowledge, we have not encountered any dedicated research on the development of methods in this context.

Concerning the operations of the PHY state machine, it is notable that the IDLE state, serving as an intermediary state for transitions to and from other states, ranks as the third highest in terms of energy expenditure. Consequently, we regard the optimization of the time allocated at this state as a viable option for reducing energy consumption, potentially allowing for the deployment of nodes at greater distances.

A similar principle applies to the time allocation in the SLEEP state. Here, the potential for energy harvesting from any frames transmitted across the network could yield even more favorable outcomes. This improvement could be further be supported by employing θ_s values approaching zero, enabling the full power level of received frames to be allocated to the extraction of energy. As well as, the deployment of power beacons, which are frames transmitted with large dummy payloads for the purpose of recharging battery levels of the nodes. However, it is important to recognize that implementing this approach would necessitate a minor modification in the technology nomenclature to accommodate these new hybrid strategies

Moreover, as per the radio energy consumption model we employed, the electrical current drawn by the nodes during TX, RX, IDLE, and CCA_Busy states remains consistent for data rates of 150 kbps, 300 kbps, and 600 kbps. Therefore, another avenue for exploration could involve optimizing combinations of the data rates to achieve further reductions in energy consumption. This approach may help narrow the gap in our efforts to reach the break-even point of sustainability at larger areas, such as those mentioned for the deployment of the macro deployment propagation loss model.

Also, we have developed features that enable the simulation of a Power Beacon (PB) feature

into the AP forming a HAP but not yet fully tested it. Then in future research we plan to access the incorporation of this feature to support energy scavenging at other PHY states.

Our implementation also incorporates a non-linear energy harvesting approach that can be used for harvesting the power of interfering frames, which will fully access in future works.

We also developed a seventh state into the PHY state machine model to assist the node when battery is depleted. Likewise, we have not fully developed all necessary dependencies in order to present any results so far.

The success obtained on the performance assessment of the sustainability metric of the second scenario propel us to investigate the outcomes of the introduction of a fourth RAW group, that may enable the conduction of a thorough comparative investigation between the single link and the network scenario.

BIBLIOGRAPHY

AL-SARAWI, S.; ANBAR, M.; ABDULLAH, R.; HAWARI, A. B. A. Internet of things market analysis forecasts, 2020–2030. In: *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. [S.l.: s.n.], 2020. p. 449–453. Cited in page 1.

ALSADER, M.; SAVVARIS, A. Integrated ltspice and ns-3 power management simulation for energy harvesting. *International Journal of Applied Mathematics and Informatics*, v. 11, p. 120–125, 2017. Cited 2 times in pages 34 and 40.

BEL, A.; ADAME, T.; BELLALTA, B. An energy consumption model for ieee 802.11 ah wlans. *Ad Hoc Networks*, Elsevier, v. 72, p. 14–26, 2018. Cited in page 86.

BELLEKENS, B.; TIAN, L.; BOER, P.; WEYN, M.; FAMAEY, J. Outdoor ieee 802.11 ah range characterization using validated propagation models. In: IEEE. *GLOBECOM 2017-2017 IEEE Global Communications Conference*. [S.l.], 2017. p. 1–6. Cited in page 85.

BENIGNO, G.; BRIANTE, O.; RUGGERI, G. A sun energy harvester model for the network simulator 3 (ns-3). In: *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking - Workshops (SECON Workshops)*. [S.l.: s.n.], 2015. p. 1–6. Cited 2 times in pages 35 and 40.

CHEN, Y. *Energy harvesting communications: principles and theories*. [S.l.]: John Wiley & Sons, 2019. Cited 5 times in pages 6, 7, 8, 9, and 10.

CLERCKX, B.; ZHANG, R.; SCHOBER, R.; NG, D. W. K.; KIM, D. I.; POOR, H. V. Fundamentals of wireless information and power transfer: From rf energy harvester models to signal and system designs. *IEEE Journal on Selected Areas in Communications*, v. 37, n. 1, p. 4–33, 2019. Cited in page 7.

FUXJAEGER, P.; RUEHRUP, S. Validation of the ns-3 interference model for ieee802. 11 networks. In: IEEE. *2015 8th IFIP Wireless and Mobile Networking Conference (WMNC)*. [S.l.], 2015. p. 216–222. Cited in page 23.

HAZMI, A.; RINNE, J.; VALKAMA, M. Feasibility study of I 802.11ah radio technology for IoT and M2M use cases. In: *2012 IEEE Globecom Workshops*. [S.l.: s.n.], 2012. p. 1687–1692. Cited 4 times in pages 10, 11, 12, and 22.

IDOTA, Y.; KUBOTA, T.; MATSUFUJI, A.; MAEKAWA, Y.; MIYASAKA, T. Tin-based amorphous oxide: a high-capacity lithium-ion-storage material. *Science*, American Association for the Advancement of Science, v. 276, n. 5317, p. 1395–1397, 1997. Cited in page 46.

IEEE. *IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. 2007. 1-1076 p. Cited in page 49.

IEEE. *IEEE Standard for Information technology–Telecommunications and information exchange between systems - Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 2: Sub 1 GHz License Exempt Operation*. 2017. 1-594 p. Cited 3 times in pages 12, 17, and 56.

IEEE. *IEEE Standard for Information technology–Telecommunications and information exchange between systems - Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 2: Sub 1 GHz License Exempt Operation*. 2017. 154 p. Cited in page 14.

IMEC. *IEEE802.11ah Wi-Fi HaLoW Radio in TSMC 40nm CMOS*. 2018. Accessed January 20, 2023. Disponível em: <https://www.imec-int.com/drupal/sites/default/files/2018-11/IEEE802.11AH%20WIFI%20HALOW%20RADIO%20IN%20TSMC%2040NM%20CMOS_digital.pdf>. Cited 3 times in pages 56, 60, and 72.

IQBAL, A.; LEE, T.-J. Spatiotemporal medium access control for wireless powered iot networks. *IEEE Internet of Things Journal*, v. 8, n. 19, p. 14822–14834, 2021. Cited 2 times in pages 38 and 40.

JAMEEL, F.; ALI, A.; KHAN, R. Optimal time switching and power splitting in swipt. In: *2016 19th International Multi-Topic Conference (INMIC)*. [S.l.: s.n.], 2016. p. 1–5. Cited in page 7.

JIANG, R.; XIONG, K.; FAN, P.; ZHANG, Y.; ZHONG, Z. Power minimization in swipt networks with coexisting power-splitting and time-switching users under nonlinear eh model. *IEEE Internet of Things Journal*, v. 6, n. 5, p. 8853–8869, 2019. Cited in page 7.

JIANG, W.; HUANG, K. A joint beamforming design in heterogeneous swipt network with imperfect csi. In: *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*. [S.l.: s.n.], 2019. p. 853–858. Cited in page 32.

JU, H.; ZHANG, R. A novel mode switching scheme utilizing random beamforming for opportunistic energy harvesting. *IEEE Transactions on Wireless Communications*, v. 13, n. 4, p. 2150–2162, 2014. Cited in page 32.

JUNIOR, J. A. de F.; CARVALHO, M. M. Uma extensão do ns-3 para simulação de transferência simultânea de informação e energia sem fio (swipt) em redes ieee 802.11. In: SBC. *Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.], 2023. p. 48–55. Cited in page 70.

KHAIRY, S.; HAN, M.; CAI, L. X.; CHENG, Y. Sustainable wireless iot networks with rf energy charging over wi-fi (cowifi). *IEEE Internet of Things Journal*, v. 6, n. 6, p. 10205–10218, 2019. Cited 3 times in pages 1, 38, and 40.

L-COM. *HyperLink Wireless 900 MHz 13 dBi 120 Degree Sector Panel Antenna*. 2021. Accessed October 25, 2023. Disponível em: <https://www.l-com.com/Images/Downloadables/Datasheets/ds_HG913P-120.pdf>. Cited in page 85.

LACAGE, M.; HENDERSON, T. R. Yet another network simulator. In: *Proceedings of the 2006 Workshop on ns-3*. [S.l.: s.n.], 2006. p. 12–es. Cited 4 times in pages 23, 26, 27, and 55.

LEE, H.; KIM, Y.; AHN, J. H.; CHUNG, M. Y.; LEE, T.-J. Wi-fi and wireless power transfer live together. *IEEE Communications Letters*, v. 22, n. 3, p. 518–521, 2018. Cited 2 times in pages 37 and 40.

LEE, I.-G.; KIM, D. B.; CHOI, J.; PARK, H.; LEE, S.-K.; CHO, J.; YU, H. Wifi halow for long-range and low-power internet of things: System on chip development and performance evaluation. *IEEE Communications Magazine*, v. 59, n. 7, p. 101–107, 2021. Cited in page 72.

LEE, W. J.; SHAH, S. T.; MUNIR, D.; LEE, T.-J.; CHUNG, M. Y. A mechanism on energy harvesting and data communications in wi-fi network. In: *Proceedings of the 10th International Conference on Ubiquitous Information Management and Communication*. [S.l.: s.n.], 2016. p. 1–6. Cited 2 times in pages 33 and 40.

LUO, Y.; LUO, C.; MIN, G.; PARR, G.; MCCLEAN, S. On the study of sustainability and outage of SWIPT-enabled wireless communications. *IEEE Journal of Selected Topics in Signal Processing*, v. 15, n. 5, p. 1159–1168, 2021. Cited 5 times in pages 10, 12, 45, 46, and 47.

MILLER, L. E. Validation of 802.11 a/uwb coexistence simulation. *national institute of standards and technology (NIST), WCTG white paper*, 2003. Cited in page 25.

NS-3 Project. *NS-3 Manual Release ns-3.23*: Release ns-3.23. 2015. 128-154 p. September 16, 2023. Disponível em: <https://www.nsnam.org/docs/release/3.23/manual/ns-3-manual.pdf>. Cited in page 30.

NS-3 Project. *NS-3 Model Library Release ns-3.23*. 2015. 416-418 p. September 16, 2023. Disponível em: <https://www.nsnam.org/docs/release/3.23/models/ns-3-model-library.pdf>. Cited 2 times in pages 49 and 64.

NSNAM. *NS3 LiIon Model Fitting*. 2010. Accessed February 8, 2023. Disponível em: <https://www.nsnam.org/wiki/Li-Ion_model_fitting>. Cited in page 72.

PANASONIC. *Lithium Battery Cylindrical 3.6V 2.45Ah Primary*. 2010. Accessed February 8, 2023. Disponível em: <https://www.datasheets.com/en/part-details/cgr18650da-panasonic-31361608#datasheet>. Cited 2 times in pages 61 and 72.

PEI, G.; HENDERSON, T. R. Validation of ofdm error rate model in ns-3. *Boeing Research Technology*, p. 1–15, 2010. Cited in page 25.

REHMAN, M. R. ur; ZADEH, H. A.; ALI, I.; LEE, K.-Y. Labview based modeling of swipt system using bpsk modulation. In: IEEE. *2018 International Conference on Electronics, Information, and Communication (ICEIC)*. [S.l.], 2018. p. 1–4. Cited 2 times in pages 36 and 40.

ROSA, R. L.; ZOPPI, G.; DONATO, L. D.; SORBELLO, G.; CARLO, C. A. D.; LIVRERI, P. A battery-free smart sensor powered with rf energy. In: *2018 IEEE 4th International Forum on Research and Technology for Society and Industry (RTSI)*. [S.l.: s.n.], 2018. p. 1–4. Cited in page 1.

SHARMA, V.; YASWANTH, J.; SINGH, S. K.; BISWAS, S.; SINGH, K.; KHAN, F. A pricing-based approach for energy-efficiency maximization in ris-aided multi-user mimo swipt-enabled wireless networks. *IEEE Access*, v. 10, p. 29132–29148, 2022. Cited in page 32.

ŠLJIVO, A.; KERKHOVE, D.; TIAN, L.; FAMAEY, J.; MUNTEANU, A.; MOERMAN, I.; HOEBEKE, J.; POORTER, E. D. Performance evaluation of ieee 802.11 ah networks with high-throughput bidirectional traffic. *Sensors*, Mdpi, v. 18, n. 2, p. 325, 2018. Cited 2 times in pages 16 and 18.

SOUSA, A. D. de; VIEIRA, L. F.; VIEIRA, M. A. Modeling, analysis and simulation of wireless power transfer. p. 143–150, 2017. Cited 2 times in pages 39 and 40.

TAPPARELLO, C.; AYATOLLAHI, H.; HEINZELMAN, W. Energy harvesting framework for network simulator 3 (ns-3). In: *Proceedings of the 2nd International Workshop on Energy Neutral Sensing Systems*. [S.l.: s.n.], 2014. p. 37–42. Cited 3 times in pages 3, 29, and 51.

TAPPARELLO, C.; AYATOLLAHI, H.; HEINZELMAN, W. Extending the energy framework for network simulator 3 (ns-3). *arXiv preprint arXiv:1406.6265*, 2014. Cited 3 times in pages 3, 29, and 51.

TARIS, T.; VIGNERAS, V.; FADEL, L. A 900mhz rf energy harvesting module. In: IEEE. *10th IEEE International NEWCAS Conference*. [S.l.], 2012. p. 445–448. Cited 2 times in pages 32 and 40.

TIAN, L.; DERONNE, S.; LATRÉ, S.; FAMAEY, J. Implementation and validation of an ieee 802.11 ah module for ns-3. In: *Proceedings of the 2016 Workshop on ns-3*. [S.l.: s.n.], 2016. p. 49–56. Cited 2 times in pages 15 and 19.

TIAN, L.; SANTI, S.; LATRÉ, S.; FAMAEY, J. Accurate sensor traffic estimation for station grouping in highly dense ieee 802.11 ah networks. In: *Proceedings of the First ACM International Workshop on the Engineering of Reliable, Robust, and Secure Embedded Wireless Sensing Systems*. [S.l.: s.n.], 2017. p. 1–9. Cited in page 86.

TIAN, L.; ŠLJIVO, A.; SANTI, S.; POORTER, E. D.; HOEBEKE, J.; FAMAEY, J. Extension of the ieee 802.11 ah ns-3 simulation module. In: *Proceedings of the 2018 Workshop on ns-3*. [S.l.: s.n.], 2018. p. 53–60. Cited 4 times in pages 19, 29, 48, and 51.

VARSHNEY, L. R. Transporting information and energy simultaneously. In: *2008 IEEE International Symposium on Information Theory*. [S.l.: s.n.], 2008. p. 1612–1616. Cited 2 times in pages 1 and 6.

VELLACHERI, R.; AL-HADDAD, A.; ZHAO, H.; WANG, W.; WANG, C.; LEI, Y. High performance supercapacitor for efficient energy storage under extreme environmental temperatures. *Nano Energy*, Elsevier, v. 8, p. 231–237, 2014. Cited in page 46.

WANG, S.; MA, L.; WU, W. Joint ts beamforming and hybrid ts-ps receiving design for swipt systems. *IEEE Access*, v. 9, p. 50686–50699, 2021. Cited in page 7.

WESTCOTT, D. A.; COLEMAN, D. D.; MILLER, B.; MACKENZIE, P. *CWAP Certified Wireless Analysis Professional Official Study Guide: Exam PW0-270*. [S.l.]: John Wiley & Sons, 2011. Cited in page 14.

XU, P.; FLANDRE, D.; BOL, D. Analysis, modeling, and design of a 2.45-ghz rf energy harvester for swipt iot smart sensors. *IEEE Journal of Solid-State Circuits*, IEEE, v. 54, n. 10, p. 2717–2729, 2019. Cited 2 times in pages 36 and 40.

ZHANG, R.; HO, C. K. Mimo broadcasting for simultaneous wireless information and power transfer. *IEEE transactions on wireless communications*, IEEE, v. 12, n. 5, p. 1989–2001, 2013. Cited in page 45.

ZHANG, R.; MAUNDER, R. G.; HANZO, L. Wireless information and power transfer: From scientific hypothesis to engineering practice. *IEEE Communications Magazine*, IEEE, v. 53, n. 8, p. 99–105, 2015. Cited in page 45.

ZHAO, Y.; HU, J.; DIAO, Y.; YU, Q.; YANG, K. Modelling and performance analysis of wireless lan enabled by rf energy transfer. *IEEE Transactions on Communications*, v. 66, n. 11, p. 5756–5772, 2018. Cited 2 times in pages 37 and 40.

ZHONG, P.; LI, Y.; HUANG, W.; KUI, X.; ZHANG, Y.; CHEN, Y. An extension to ns-3 for simulating mobile charging with wireless energy transfer. In: SPRINGER. *Data Science: Third International Conference of Pioneering Computer Scientists, Engineers and Educators, ICPCSEE 2017, Changsha, China, September 22–24, 2017, Proceedings, Part II*. [S.l.], 2017. p. 256–270. Cited 2 times in pages 35 and 40.

APPENDIX A

SINGLE LINK MANUAL

This manual contains procedures for installing the software described in the article submitted to the Tools Hall of the XLI SBRC Symposium. It also contains instructions on the use of the main components to run the simulation and graphically visualize the results, namely:

1. NS-3 source code, version 3.23, changed to implement the IEEE 802.11ah standard [1];
2. Folder for storing simulation results in .csv format files;
3. JupyterLab Notebook with code in Python 3.8 language, for reading and graphical presentation of results, located in the parent directory of the previous item.

Therefore, there is no need to download any additional version of NS-3. The version mentioned in item 1, can be found at the following address: `<https://github.com/imec-idlab/IEEE-802.11ah-NS-3>`.

Finally, given that NS-3 is used, to a large extent, through line commands entered into the operating system's terminal, the following information in this manual follows the same pattern.

A.1 SIMULATION SCRIPTS

Simulation scripts are programs written in the C++ programming language, which generally have a code structure subdivided into 3 (three) main parts, namely:

1. Configuration;
2. Simulation;

[1] Le Tian, Amina Sljivo, Serena Santi, Eli De Poorter, Jeroen Hoebeke, Jeroen Famaey. Extension of the IEEE 802.11ah NS-3 Simulation Module. Workshop on NS-3 (WNS3), 2018.

3. Destruction.

The configuration begins with the first line of code found in script and covers the entire length up to the point where the command to start the simulation is found.

The section of code that covers the instructions from starting the simulation to ending it refers to the simulation part. In this range, methods, functions and other data structures can be configured to collect information that you want to observe about the simulation.

Finally, the destruction section corresponds to the remainder and is necessary due to the need to free up memory used during the simulations.

The Python programming language can also be used to configure simulation scripts in NS-3. However, this approach is not part of the scope of the tool in question.

A.1.1 Configuration and Construction System (Waf)

Waf is the program used for several NS-3 activities, for example, to install NS-3 itself and configure test scenarios

The following excerpt presents the main Waf commands and was obtained as a result of executing the Waf help function in the simulator covered in this manual, as follows:

```
 ./waf --help
waf [commands] [options]

Main commands (example: ./waf build -j4)
build    : executes the build
check    : run the equivalent of the old NS-3 unit tests using test.py
clean    : cleans the project
configure: configures the project
dist     : makes a tarball for redistributing the sources
distcheck: checks if the project compiles (tarball from 'dist')
docs     : build all the documentation: doxygen, manual, tutorial, models
doxygen  : do a full build, generate the introspected doxygen
install  : installs the targets on the system
list     : lists the targets to execute
shell    : run a shell with an environment suitably modified
sphinx   : build the Sphinx documentation: manual, tutorial, models
step     : executes tasks in a step-by-step fashion, for debugging
```

```
uninstall: removes the targets installed

...
```

The Waf help function can also be used to obtain information about the configuration options available through `scripts`, as shown in the following example:

```
 ./waf --run "scratch/singleLink --help"
Waf: Entering directory `/home/user/repos/SWIPT3/SWIPT/build'
Waf: Leaving directory `/home/user/repos/SWIPT3/SWIPT/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.066s)

singleLink [Program Arguments] [General Arguments]

Program Arguments:
--simulationTime:  Duração da Simulação [300]
--coolDown:        Tempo adicional para p/ esvaziamento dos buffers [5]
--payloadSize:     Tamanho da carga útil de um pacote UDP [1024]
--verbose:         Habilitar logs da simulação [false]
--tracing:         Habilitar geração de PCAP [false]

General Arguments:
--PrintGlobals:              Print the list of globals.
--PrintGroups:               Print the list of groups.
--PrintGroup=[group]:        Print all TypeIds of group.
--PrintTypeIds:              Print all TypeIds.
--PrintAttributes=[typeid]:  Print all attributes of typeid.
--PrintHelp:                 Print this help message.
```

A.1.2 Directory for Executing Simulation Scripts

The root directory of the NS-3 installation is the location where Waf routines should be run. Therefore, installing NS-3, setting attributes or running simulation scripts must always call `<./waf>` at the root of the NS-3 tree, which in the case of SWIPT.tar.gz is called `SWIPT`.

The results of the simulation scripts, error messages are also presented in the NS-3 root directory, in most cases.

A.1.3 Scripts Storage Subdirectory

The simulation script, like the other classes, have a specific storage location in the NS-3 subdirectory tree, whose name and path are `scratch` and `SWIPT/scratch`, respectively.

If there is more than one script in the `scratch` folder, when the Waf simulation command is executed, all `scripts` will be compiled for NS-3 consistency checking purposes. Furthermore, Waf does not allow the deliberate creation of subfolders or different types of files in this space.

Given the above information, it is clear that Waf must be executed at the highest level of the NS-3 directory tree structure, as explained in the next subsection.

A.1.4 Results Subdirectory

In addition to the information presented on the computer screen during simulation execution, the folder is also used by NS-3 to store values of the parameters of interest in .csv format files, as defined in the configuration.

These .csv files contain detailed information about the following quantities: remaining energy level in the node battery, total harvested energy, total harvested power, and so on.

A.2 TOOL INSTALLATION

A.2.1 Prerequisites for Installation

A.2.1.1 Operating System

It is recommended to use the Ubuntu 20.04 LTS (Focal Fossa) Operating System (OS), as some components of version 3.23 of NS-3 need to be run on Python2, such as the Waf installation system, which will be presented in the next chapter.

Furthermore, it is recommended to install OS Ubuntu 20.04 directly on a physical disk, without using disks or virtual machines. Therefore, the stability of the system was not evaluated in different test environments.

If it is necessary to install OS Ubuntu 20.04, we suggest adopting the procedure available at: Desktop Ubuntu Installation

A.2.1.2 Dependencies with Other Programs

Note, installation of the following programs requires knowledge of the user `root` password.

To make the installation less laborious, below we present the commands that can be used to install the various NS-3 dependencies. These programs are in accordance with what appears in the repository IEEE 802.11ah NS-3 with changes to the context of single link scenario.

The commands presented below are based on the use of Oh My Zsh `shell`. Thus, optionally, if the user is interested in downloading it, its installation procedure is available in the repository Oh My Zsh.

Furthermore, the following commands can be copied from this document and pasted in their entirety into the Ubuntu 20.04 Linux terminal screen, or typed manually, as appropriate.

```
sudo apt-get update
sudo apt-get -y install tar
sudo apt-get install -y bzip2
sudo apt-get -y install gcc g++ python
sudo apt-get -y install gcc g++ python python-dev
sudo apt-get -y install qt4-dev-tools libqt4-dev
sudo apt-get -y install mercurial
sudo apt-get -y install bzr
sudo apt-get -y install cmake libc6-dev libc6-dev-i386 g++-multilib
sudo apt-get -y install gdb valgrind
sudo apt-get -y install gsl-bin libgsl2 libgsl2:i386
sudo apt-get -y install flex bison libfl-dev
sudo apt-get -y install tcpdump
sudo apt-get -y install sqlite sqlite3 libsqlite3-dev
sudo apt-get -y install libxml2 libxml2-dev
sudo apt-get -y install libgtk2.0-0 libgtk2.0-dev
sudo apt-get -y install vtun lxc
```

A.2.1.3 JupyterLab Installation

To install Jupyter Lab, you must first make sure that Python3 is installed and working correctly on the system. According to the Ubuntu PortalUbuntu Python 3.8.2 is a package embedded in the Ubuntu 20.04 installation.

To install Jupyter Lab, enter the following command in the terminal:

```
cd /home/nomeDoUsuario
python3 -m pip install --upgrade pip
pip3 install jupyterlab
```

In case of problems, check the tool documentation available in Jupyter Lab.

Furthermore, the packages presented in the second cell of the Notebook `/SWIPT/ResultsingleLink/N` must be installed in Python 3.8, so it is suggested to insert the following command, if necessary]:

```
pip3 install argparse glob2 subprocess ipywidgets matplotlib numpy pandas
mpl_toolkits cycler
```

A.2.2 Tool Installation Location

Assuming that the SWIPT.tar.gz file has been downloaded to the `downloads` folder, create a new folder in your `/home/user` directory, transfer the SWIPT.tar.gz file inside of this new folder and unzip it using the command `tar xzf SWIPT.tar.gz`, as shown in the following example:

```
cd /home/user
mkdir repos
mv Downloads/SWIPT.tar.gz repos/
cd repos/
tar xzf SWIPT.tar.gz
cd SWIPT
ls -ld */
```

The last command `ls -ld */` is used to list only the directories in the SWIPT folder. Therefore, it must be checked whether the files presented in the user installation are in accordance

with the information presented below:

```
drwxr-xr-x  3 user user 131072 Jan  2 12:20 bindings/
drwxr-xr-x  7 user user 131072 Mar 19 00:12 build/
drwxr-xr-x  7 user user 131072 Jan  2 12:20 doc/
drwxr-xr-x 16 user user 131072 Jan  2 12:20 examples/
drwxr-xr-x  3 user user 131072 Mar 11 22:13 OptimalRawGroup/
drwxr-xr-x  2 user user 131072 Jan  2 12:20 __pycache__/
drwxr-xr-x  2 user user 131072 Mar 19 00:45 ResultsingleLink/
drwxr-xr-x  2 user user 131072 Mar 19 00:12 scratch/
drwxr-xr-x 45 user user 131072 Jan  2 12:20 src/
drwxr-xr-x  2 user user 131072 Jan  2 12:20 testpy-output/
drwxr-xr-x  3 user user 131072 Jan  2 12:20 utils/
drwxr-xr-x  2 user user 131072 Jan 18 21:02 waf-tools/
```

After that, configure the Waf installation system using the command shown below:

```
CXXFLAGS="-std=c++11" ./waf configure --disable-examples --disable-tests
```

If the system has been configured correctly and Waf finds no errors, the following information should be presented on the last line: `configure finished successfully (2.536s)`, as shown on the following screen.

```
---Summary of optional NS-3 features:
Build profile : debug
Build directory :
BRITE Integration : not enabled (BRITE not enabled (see option --with-brite))
DES Metrics event collection  : not enabled (defaults to disabled)
Emulation FdNetDevice : enabled
Examples : not enabled (option --disable-examples selected)
File descriptor NetDevice : enabled
GNU Scientific Library (GSL) : not enabled (GSL not found)
Gcrypt library : not enabled
GtkConfigStore: not enabled (library 'gtk+-2.0 >= 2.12' not found)
MPI Support: not enabled (option --enable-mpi not selected)
NS-3 Click Integration : not enabled
NS-3 OpenFlow Integration : not enabled (Required boost libraries not found)
Network Simulation Cradle: not enabled (NSC not found (see option --with-nsc))
PlanetLab FdNetDevice: not enabled
PyViz visualizer : not enabled (Python Bindings are needed but not enabled)
Python Bindings : not enabled (Python library or headers missing)
Real Time Simulator : enabled
```

```
SQlite stats data output : not enabled (library 'sqlite3' not found)
Tap Bridge : enabled
Tap FdNetDevice: enabled
Tests: not enabled (option --disable-tests selected)
Threading Primitives: enabled
Use sudo to set suid bit: not enabled (option --enable-sudo not selected)
XmlIo: not enabled (library 'libxml-2.0 >= 2.7' not found)
'configure' finished successfully (2.536s)
```

This ends the simulation environment preparation stage. Now the NS-3 source code must be compiled.

A.2.3 Compiling NS-3 Source Code

The NS-3 source code can be compiled by executing the following command:

```
./waf
```

If the installation is successful, a notification about the success of the operation will be displayed on the screen. However, in case of an error, Waf will be interrupted and the compiler will show the reasons for the interruption.

The time required to compile the NS-3 source code for the first time may take some time. If the user wishes to obtain additional proof of the success of the installation, the following command can be executed in the root directory `SWIPT`.

```
./waf --run scratch/hello-simulator
```

If the environment is working correctly, the following message will be displayed at the end of the routine compilation:

```
 ./waf --run scratch/hello-simulator
Waf: Entering directory `/home/user/repos/SWIPT3/SWIPT/build'
[ 847/1750] Compiling scratch/hello-simulator.cc
[ 848/1750] Compiling scratch/scratch-simulator.cc
[ 849/1750] Compiling scratch/RAW-generate.cc
```

```
[1736/1750] Linking build/scratch/scratch-simulator
[1739/1750] Linking build/scratch/hello-simulator
[1739/1750] Linking build/scratch/RAW-generate
Waf: Leaving directory `/home/user/repos/SWIPT3/SWIPT/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (6.268s)
Hello Simulator
        ...
```

A.3 SIMULATING THE SWIPT ENERGY HARVESTER

The energy harvester script is called `singleLink.cc`, its storage location is the `scratch` subdirectory. The full path from the run directory is `/SWIPT/scratch/singleLink.cc`. In summary:

```
Script de configuração: singleLink.cc
Localização: /scratch
Caminho: /SWIPT/scratch/singleLink.cc
```

A.3.1 Simulation script Configuration Options

To learn about the available configuration options for the simulation script, type the following command in the execution directory or root directory of the `SWIPT` installation:

```
./waf --run "scratch/singleLink --help"
```

As a result, NS-3 presents the following information:

```
Waf: Entering directory `/home/user/repos/SWIPT3/SWIPT/build'
Waf: Leaving directory `/home/user/repos/SWIPT3/SWIPT/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.093s)
Std: IEEE 802.11ah; simulationTime: 300s; coolDown: 5s;
```

```
Packet Size: 1024Bytes; verbose: 0; tracing: 0
(...)
singleLink [Program Arguments] [General Arguments]

Program Arguments:
--simulationTime:  Duração da Simulação [300]
--coolDown:        Tempo adicional para p/ esvaziamento dos buffers [5]
--payloadSize:     Tamanho da carga útil de um pacote UDP [1024]
--verbose:         Habilitar logs da simulação [false]
--tracing:         Habilitar geração de PCAP [false]

General Arguments:
--PrintGlobals:              Print the list of globals.
--PrintGroups:               Print the list of groups.
--PrintGroup=[group]:        Print all TypeIds of group.
--PrintTypeIds:              Print all TypeIds.
--PrintAttributes=[typeid]:  Print all attributes of typeid.
--PrintHelp:                 Print this help message.
```

The configuration parameters are presented after the section that reads:

`singleLink [Program Arguments] [General Arguments]`

Thus, the simulation script configuration options are subdivided into two categories: program arguments and general arguments.

The configuration options for the program arguments are:

```
Program Arguments:
--simulationTime:       Simulation Duration [300]
--coolDown:             Additional time to empty buffers [5]
--payloadSize:          Payload size of a UDP packet [1024]
--verbose:              Enable simulation logs [false]
--tracing:              Enable PCAP generation [false]
```

The configuration options for the general arguments are:

```
--PrintGlobals:              Print the list of globals.
--PrintGroups:               Print the list of groups.
--PrintGroup=[group]:        Print all TypeIds of group.
```

```
--PrintTypeIds:              Print all TypeIds.
--PrintAttributes=[typeid]:  Print all attributes of typeid.
--PrintHelp:                 Print this help message.
```

There is also the possibility of not configuring any option, in this case script adopts the default values, which are presented in square brackets:

```
Simulation Duration [300]
Additional time to empty buffers [5]
Payload size of a UDP packet [1024]
Enable simulation logs [false]
Enable PCAP generation [false]
```

A.3.2 Configuring the Simulation Script

To configure the simulation script options, the path must be informed to Waf together with the configuration option preceded by two dashes and the value in quotation marks, as follows:

```
./waf --run "path --option=value"
```

Let's assume a hypothetical situation in which we want to change the duration of the simulation from its default value of 300 seconds to 100 seconds. In this case the command to be entered in the terminal is:

```
./waf --run "scratch/singleLink --simulationTime=100"
```

The result of changing this parameter is displayed on the screen, as follows:

```
./waf --run "scratch/singleLink --simulationTime=100"
Waf: Entering directory `/home/user/repos/SWIPT3/SWIPT/build'
[ 846/1750] Compiling scratch/singleLink.cc
[1739/1750] Linking build/scratch/singleLink
Waf: Leaving directory `/home/user/repos/SWIPT3/SWIPT/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (6.642s)

Std: IEEE 802.11ah; simulationTime: 100s; coolDown: 5s;
Packet Size: 1024Bytes; verbose: 0; tracing: 0
```

Note that the `simulationTime` parameter now has a value equal to `100s`.

Note: It should be noted at this point that the value of 100s assigned to the variable `simulationTime: 100s` refers to the simulation interval for a combination of a single value of θ_s with a single value away, according to the test scenario defined in the dissertation.

The other parameters can be changed following the same pattern. Furthermore, several parameters can be configured at the same time, as shown in the following example:

```
./waf --run "scratch/singleLink --simulationTime=100 --coolDown=10
--payloadSize=512"

Waf: Entering directory `/home/user/repos/SWIPT3/SWIPT/build'
Waf: Leaving directory `/home/user/repos/SWIPT3/SWIPT/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.039s)

Std: IEEE 802.11ah; simulationTime: 100s; coolDown: 10s;
Packet Size: 512Bytes; verbose: 0; tracing: 0
```

Note that in the result above it was possible to change more than one parameter at the same time.

However, it is possible that the user does not wish to make configuration changes to the default parameters, in which case the command below must be entered in the terminal:

```
./waf --run scratch/singleLink
```

Notice the change in the command syntax. In this case, there are no quotes in the parameter that defines the script path.

A.3.3 Misconfiguration of Parameters

If the parameter value does not correspond to the expected data structure type, NS-3 displays a `Invalid argument value: parameter=value` message, as follows:

```
./waf --run "scratch/singleLink --simulationTime=qwert --coolDown=10
--payloadSize=512"
Waf: Entering directory `/home/user/repos/SWIPT3/SWIPT/build'
Waf: Leaving directory `/home/user/repos/SWIPT3/SWIPT/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (1.057s)

Invalid argument value: simulationTime=qwert

Command ['/home/SWIPT/build/scratch/singleLink',
 '--simulationTime=qwert', '--coolDown=10', '--payloadSize=512']
 exited with code 1
```

Furthermore, the parameter values must be in accordance with the test scenario defined in the article.

A.3.4 Running the Simulation

To run the simulation, choose the values of the desired configuration parameters, as presented in the previous subsections, or if you wish to run the simulation using the default values, enter the respective command in the terminal as presented previously.

The following excerpt presents information about executing script using default values.

However, it was necessary to suppress part of the information so that it could be presented in this document. Thus, as the simulation interval defined by the `simulationTime` variable is exhausted for each combination of θ_s and distance, a table line is inserted on the screen.

Furthermore, the table columns present the following parameters, from the first to the last column, respectively: distance in meters; transfer rate in kbps; flow rate in kbps; number of UDP packets transmitted, received and lost; number of frames transmitted, received and lost; θ_s; power collected per received frame; energy collected per received frame; signal power received by the device in dBm; Noise in dBm.; SNR in dB and frame chunk success rate (CSR: `Chunk Success Rate`).

Dist	Data	Tput	UDP			Frames		
(m)	Rate	(kbps)	Tx	Rx	Lost	Tx	Rx	Lost
1.50	150	0.00	5494	0	5494	2979	0	2979
1.50	150	139.86	5494	5122	372	1895	1891	4
1.50	150	140.30	5494	5138	356	1885	1877	8
1.50	150	140.22	5494	5135	359	1891	1888	3
1.50	150	140.30	5494	5138	356	1887	1879	8
1.50	150	140.30	5494	5138	356	1887	1879	8
1.50	150	139.84	5494	5121	373	1899	1895	4
1.50	150	139.86	5494	5122	372	1900	1896	4
1.50	150	140.22	5494	5135	359	1895	1892	3
1.50	150	140.30	5494	5138	356	1888	1880	8
1.50	150	140.08	5494	5130	364	1885	1879	6
1.50	150	140.22	5494	5135	359	1895	1892	3
1.50	150	140.22	5494	5135	359	1890	1887	3
1.50	150	140.08	5494	5130	364	1884	1878	6
1.50	150	139.86	5494	5122	372	1897	1893	4
1.50	150	139.92	5494	5124	370	1904	1900	4
1.50	150	140.22	5494	5135	359	1891	1888	3
1.50	150	139.92	5494	5124	370	1903	1899	4
1.50	150	140.25	5494	5136	358	1889	1886	3
1.50	150	140.08	5494	5130	364	1880	1874	6

A.3.5 Simulation Results

The results of the simulations can be found in files in .csv format in the folder located in the following subdirectory, in relation to the installation root directory, `</SWIPT/ResultsingleLink/`

subdir/> which contains the following files:

```
ls -la subdir
total 27452
drwxrwxr-x 2 user user    4096 Mar 19 06:47 .
drwxrwxr-x 3 user user    4096 Mar 19 06:47 ..
-rw-rw-r-- 1 user user 5096243 Mar 20 10:10 battery_remaining_energy_0.csv
-rw-rw-r-- 1 user user 3663122 Mar 20 10:10 energy_consumption_0.csv
-rw-rw-r-- 1 user user 1315154 Mar 20 10:10 harvested_power_0.csv
-rw-rw-r-- 1 user user 5723824 Mar 20 10:10 MonitorSniffRx_0.csv
-rw-rw-r-- 1 user user  942304 Mar 20 10:10 MonitorSniffRx_1.csv
-rw-rw-r-- 1 user user  815172 Mar 20 10:10 MonitorSniffTx_0.csv
-rw-rw-r-- 1 user user 5719121 Mar 20 10:10 MonitorSniffTx_1.csv
-rw-rw-r-- 1 user user    1020 Mar 20 10:10 OnMacPacketDropped_1.csv
-rw-rw-r-- 1 user user  342206 Mar 20 10:10 OnPhyRxDrop_0.csv
-rw-rw-r-- 1 user user 2999059 Mar 20 10:10 state_.csv
-rw-rw-r-- 1 user user    4998 Mar 20 10:10 Summary.csv
-rw-rw-r-- 1 user user 1445676 Mar 20 10:10 total_energy_harvested_0.csv
```

The name of the files above refers to the variable of interest observed during the simulation. Files that end with the number 0 before the file extension refer to the event observed by the node, while those that end in 1 refer to the events observed by the AP. There are exceptions for the `Summary.csv` file, as it presents network-wide information, and for the `state_.csv` file that refers to node states. The names of the files are shown below:

```
battery_remaining_energy_0.csv - Node battery energy
energy_consumption_0.csv - Energy consumed by the node
harvested_power_0.csv - Power collected by the node
MonitorSniffRx_0.csv - RX frames per node
MonitorSniffRx_1.csv - When RX by AP
MonitorSniffTx_0.csv - TX frames per node
MonitorSniffTx_1.csv - TX Frames by AP
OnMacPacketDropped_1.csv - Packets dropped on the MAC of the AP
OnPhyRxDrop_0.csv - Frames dropped in the PHY
state_.csv - Node PHY states.
Summary.csv - Summary of all information on the screen
total_energy_harvested_0.csv - Total energy collected
```

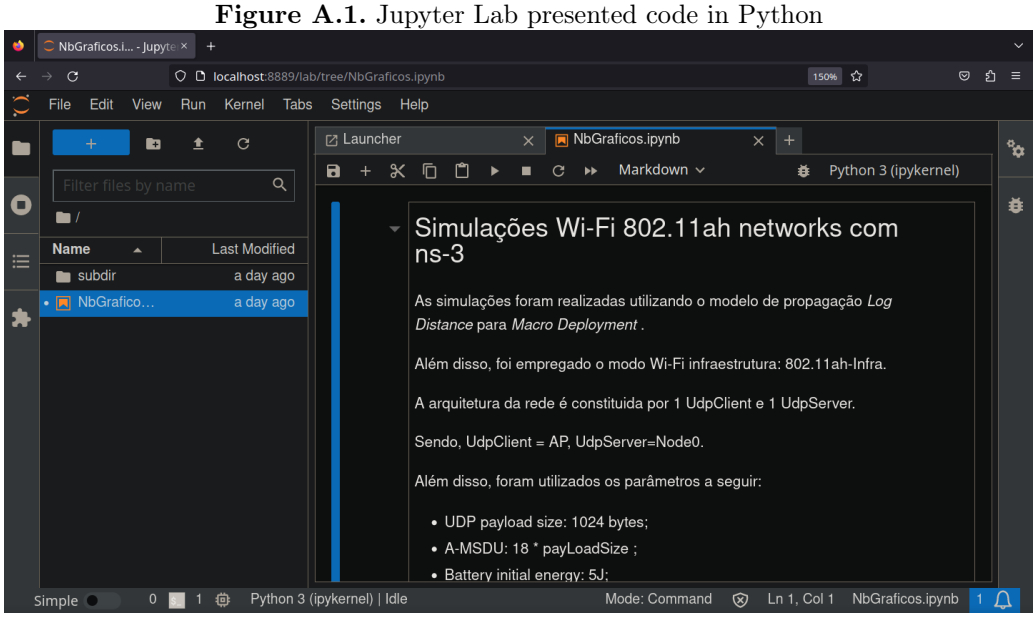
A.3.6 Graphical Results

The file `</SWIPT/ResultsingleLink/NbGraficos.ipynb>` can be used to visualize the results graphically. Therefore, at the end of the simulation, when new results are no longer displayed on the terminal screen and `prompt` is available, enter the following commands:

```
jupyter lab
```

After this, a page will open in your default browser, with the same content as shown below.

Figure A.1. Jupyter Lab presented code in Python



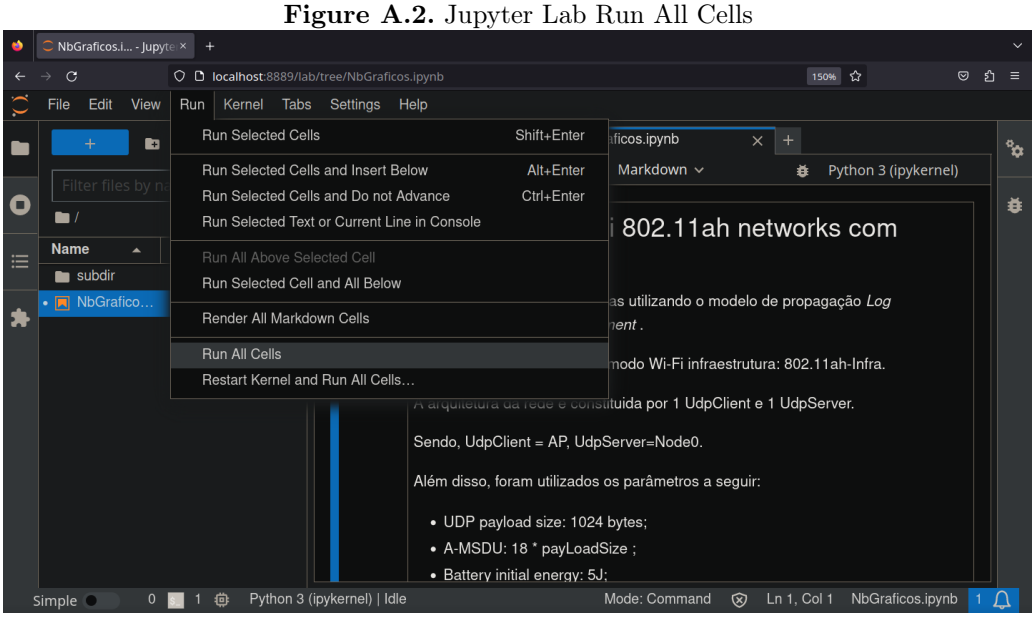
Source: Own authorship

To run the Python 3.8 code in this Notebook, perform the following operation: In the `Run` option, click on `Run All Cell`, as shown below:

After completing the execution of the code, the respective graph will be displayed at the end of each cell, as shown in the following figure:

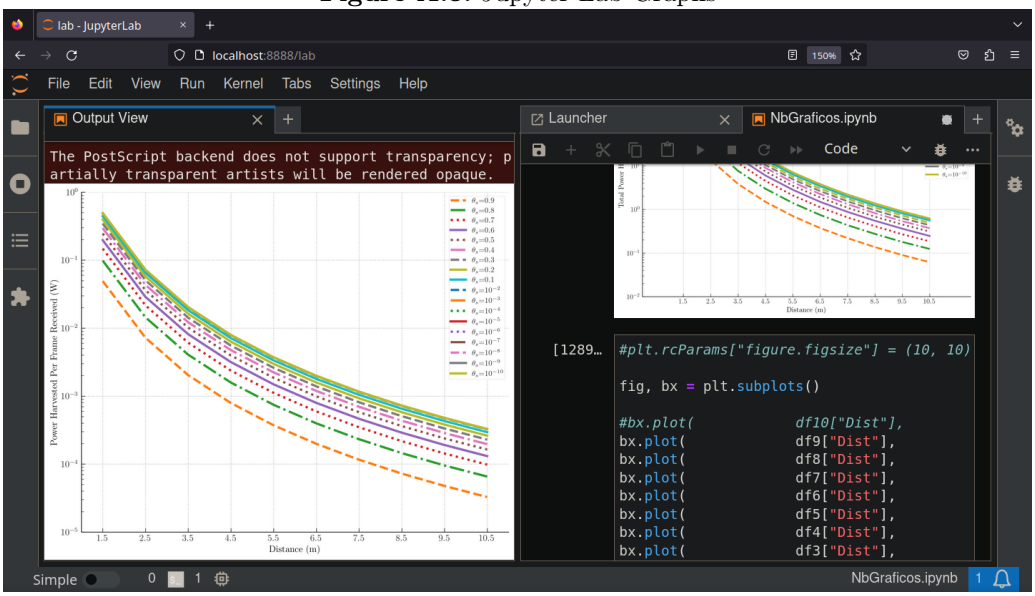
H

Figure A.2. Jupyter Lab Run All Cells



Source: Own authorship

Figure A.3. Jupyter Lab Graphs



Source: Own authorship

To generate the visualization above, right-click on the graph and select the `Create New View For Output` option. After that, drag the new window to the top left of the browser.

APPENDIX B

SINGLE LINK SCRIPT

```
/* -*-  Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2009 The Boeing Company
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/energy-module.h"
#include "ns3/udp-client-server-helper.h"
#include "ns3/udp-echo-helper.h"
#include "ns3/udp-echo-server.h"
#include "ns3/udp-echo-client.h"
#include "ns3/udp-server.h"
#include "ns3/udp-client.h"
#include "ns3/packet.h"
#include "ns3/on-off-helper.h"
#include "ns3/packet-sink.h"
#include "ns3/packet-sink-helper.h"
#include "ns3/command-line.h"
#include "ns3/rps.h"
```

```
39 #include "ns3/log.h"
40 #include "ns3/node.h"
41 #include "ns3/simulator.h"
42 #include "ns3/double.h"
43 #include "ns3/config.h"
44 #include "ns3/string.h"
45 #include "ns3/extension-headers.h"
46 #include "ns3/stats-module.h"
47 #include "ns3/application.h"
48 #include "ns3/flow-monitor.h"
49 #include "ns3/flow-monitor-helper.h"
50 #include "ns3/ipv4-flow-classifier.h"
51 #include "ns3/wifi-tx-vector.h"
52 #include "ns3/wifi-mode.h"
53 #include "ns3/mac-low.h"
54 #include "ns3/dca-txop.h"
55 #include "ns3/edca-txop-n.h"
56 #include "ns3/propagation-loss-model.h"
57
58 #include "ns3/core-module.h"
59 #include "ns3/network-module.h"
60 #include "ns3/applications-module.h"
61 #include "ns3/wifi-module.h"
62 #include "ns3/mobility-module.h"
63 #include "ns3/ipv4-global-routing-helper.h"
64 #include "ns3/internet-module.h"
65 #include "ns3/extension-headers.h"
66
67 #include <iostream>
68 #include <fstream>
69 #include <vector>
70 #include <string>
71 #include <iomanip>
72 #include <utility>
73 #include <stdio.h>
74 #include <stdlib.h>
75 #include <ctime>
76 #include <sys/stat.h>
77 #include <map>
78
79 //#include "wifi-example-apps.h"
80
81 using namespace std;
82 using namespace ns3;
83
84 NS_LOG_COMPONENT_DEFINE("WifiSimpleInterference");
85
86
```

```
int NRawSta;
uint32_t Nsta;
uint32_t onPhyTxDrop[3];
uint32_t onPhyRxDrop[3];
uint32_t onMacDrop[3];
uint32_t onMacDrop2[3];
uint32_t onCollision[3];
uint32_t onPktsRx[3];
uint32_t onPktsTx[3];
uint32_t onUdpRx;
uint32_t onUdpEchoRx;
uint32_t onUdpTx;
uint32_t txXRawBound[3];
uint32_t nTxRawSlot[3];
uint32_t NumberOfAPScheduledPacketForNodeInNextSlot;
uint32_t NumberOfAPSentPacketForNodeImmediately;
uint32_t onQueueDrop[3];
uint32_t phyTxEnd[3];
uint32_t phyRxEnd[3];

uint32_t harvestedPowerCounter;
uint32_t harvestedEnergyCounter;

double g_harvestedEnergyAvg;

double harvestedEnergy[3];
double g_harvestedPowerAvg;
double g_signalDbmAvg[3];
double g_noiseDbmAvg[3];
double consump;

double csr = 0.0;
double ber = 0.0;
double APTotalTimeRemainingWhenSendingPacketInSameSlot;
double dist= 0.0;
double psFactor = 0.0;
double m_rxSignalDbm[3];

RPSVector rps;
uint32_t nRps;         // Ordinal number of current RPS element; RPS Index
uint64_t totalRawSlots = 0;   // Total number of RAW slots in all RAW groups in all RPS
↪  elements
std::string RawConfigString; //
↪  RPS=2;{RAW=2;[0,1,1,204,2,0,1,16][0,1,1,412,1,0,17,32]}{RAW=1;[0,1,1,180,3,0,33,35]}
pageSlice pageS;
TIM tim;

uint16_t currentRps = 0;
```

```
133 uint16_t currentRawGroup = 0;
134 uint16_t currentRawSlot = 0;
135
136 uint32_t pagePeriod = 1; //  Number of Beacon Intervals between DTIM beacons that carry
    ↪  Page Slice element for the associated page
137 uint8_t pageIndex = 0;
138 uint32_t pageSliceLength = 1; //  Number of blocks in each TIM for the associated page
    ↪  except for the last TIM (1-31) (value 0 is reserved);
139 uint32_t pageSliceCount = 0;  //  Number of TIMs in a single page period (1-31)
140 uint8_t blockOffset = 0;    //  The 1st page slice starts with the block with blockOffset
    ↪  number
141 uint8_t timOffset = 0;      //  Offset in number of Beacon Intervals from the DTIM that
    ↪  carries the first page slice of the page
142
143 int SlotFormat = 0;     // 0;
144 int NRawSlotCount = 0; // 162;
145 uint32_t NRawSlotNum = 0;
146 uint32_t NGroup = 0;
147 uint32_t BeaconInterval = 1024000;
148 uint32_t txrate = 0;
149 uint32_t payloadSize = 1024;
150
151 string RAWConfigFile = "./OptimalRawGroup/RawConfig-test14.txt";
152
153 Time startTxUdp;
154
155 std::string dropReason[] =
156   {
157     "Unknown",
158     "PhyInSleepMode",
159     "PhyNotEnoughSignalPower",
160     "PhyUnsupportedMode",
161     "PhyPreampleHeaderReceptionFailed",
162     "PhyRxDuringChannelSwitching",
163     "PhyAlreadyReceiving",
164     "PhyAlreadyTransmitting",
165     "PhyPlcpReceptionFailed",
166     "MacNotForAP",
167     "MacAPToAPFrame",
168     "MacQueueDelayExceeded",
169     "MacQueueSizeExceeded",
170     "TCPTxBufferExceeded",
171     "PhyRxNotEnoughSNIR",
172     "DeviceIsOff"};
173
174 class assoc_record
175 {
176 public:
```

```
177   assoc_record();
178   bool GetAssoc();
179   void SetAssoc(std::string context, Mac48Address address);
180   void UnsetAssoc(std::string context, Mac48Address address);
181   void setstaid(uint16_t id);
182
183 private:
184   bool assoc;
185   uint16_t staid;
186 };
187
188 assoc_record::assoc_record()
189 {
190   assoc = false;
191   staid = 65535;
192 }
193
194 void assoc_record::setstaid(uint16_t id)
195 {
196   staid = id;
197 }
198
199 void assoc_record::SetAssoc(std::string context, Mac48Address address)
200 {
201   assoc = true;
202 }
203
204 void assoc_record::UnsetAssoc(std::string context, Mac48Address address)
205 {
206   assoc = false;
207 }
208
209 bool assoc_record::GetAssoc()
210 {
211   return assoc;
212 }
213
214 typedef std::vector<assoc_record *> assoc_recordVector;
215 assoc_recordVector assoc_vector;
216
217 uint32_t GetAssocNum(assoc_recordVector assoc_vector)
218 {
219
220   uint32_t AssocNum = 0;
221   for (assoc_recordVector::const_iterator index = assoc_vector.begin();
222      index != assoc_vector.end(); index++)
223   {
224     if ((*index)->GetAssoc())
```

```
225     {
226       AssocNum++;
227     }
228   }
229   return AssocNum;
230 }
231 
232 void OnUdpPacketSent(Ptr<const Packet> packet)
233 {
234   onUdpTx++;
235 }
236 
237 void udpPacketReceivedAtServer(Ptr<const Packet> packet, Address from)
238 { // works
239 
240   onUdpRx++;
241 
242 }
243 
244 void OnUdpEchoPacketReceived(Ptr<const Packet> packet, Address from)
245 { // works
246 
247   onUdpEchoRx++;
248 
249 }
250 
251 
252 Ptr <NistErrorRateModel> nist = CreateObject<NistErrorRateModel> ();
253 Ptr <YansErrorRateModel> yans = CreateObject<YansErrorRateModel> ();
254 
255 template <int node>
256 void MonitorRx(Ptr<const Packet> packet, uint16_t channelFreqMhz,
257          uint16_t channelNumber, uint32_t rate, bool isShortPreamble,
258          WifiTxVector txvector, double signalDbm, double noiseDbm)
259 {
260   onPktsRx[node]++;
261   g_signalDbmAvg[node] += ((signalDbm - g_signalDbmAvg[node]) / onPktsRx[node]);
262   g_noiseDbmAvg[node] += ((noiseDbm - g_noiseDbmAvg[node]) / onPktsRx[node]);
263   m_rxSignalDbm[node] = signalDbm;
264 
265   if (node == 0)
266   {
267     //ps = nist->GetChunkSuccessRate (WifiMode (txvector.GetMode()), std::pow (10.0,
      ↪  (signalDbm - noiseDbm)/10.0), (packet->GetSize () * 8));
268     double signalW = std::pow (10.0, (signalDbm / 10.0));
269     double noiseW = std::pow (10.0, (noiseDbm / 10.0));
270     double snr = signalW * 1.0 / noiseW;
271     if (txvector.GetMode ().GetConstellationSize () != 256)
```

```
272       {
273         csr = nist->GetChunkSuccessRate (txvector.GetMode(), snr, (8 * packet->GetSize ()));
274       }
275       else
276       {
277         csr = yans->GetChunkSuccessRate (txvector.GetMode(), snr, (8 * packet->GetSize ()));
278       }
279 
280       if (csr < 1.0)
281       {
282         cout << "CSR < 1.0, " << txvector.GetMode() << ", " << snr << ", " << 8 *
          ↪  packet->GetSize () << ", " << csr << endl;
283       }
284     }
285 
286     std::stringstream ss;
287     ss << "./ResultColetorSwipt/MonitorSniffRx_" << node << ".csv";
288     // ss << "./ResultColetorSwipt/MonitorSniffRx.csv";
289     static std::fstream f(ss.str().c_str(), std::ios::out);
290     // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " << i
      ↪  << ", " << sources.Get (i)->GetRemainingEnergy() << std::endl;
291     // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() <<
      ↪  std::endl;
292     f << Simulator::Now().GetSeconds() << ", "
293       << dist<< ", "
294       << psFactor << ", "
295       << packet << ", "
296       << channelFreqMhz << ", "
297       << channelNumber << ", "
298       << rate << ", "
299       << isShortPreamble << ", "
300       << txvector << ", "
301       << signalDbm << ", "
302       << noiseDbm << ", " ;
303     if (node == 0)
304     {
305       f << csr;
306     }
307     f << std::endl;
308 
309 }
310 
311 template <int node>
312 void PhyTxEnd (Ptr<const Packet> packet)
313 {
314   phyTxEnd[node]++;
315 }
316 
```

```
317 template <int node>
318 void PhyRxEnd (Ptr<const Packet> packet)
319 {
320
321   phyRxEnd[node]++;
322 }
323
324 template <int node>
325 void MonitorTx (Ptr<const Packet> packet, uint16_t channelFreqMhz,
326         uint16_t channelNumber, uint32_t rate, bool isShortPreamble,
327         WifiTxVector txvector)
328 {
329
330   std::stringstream ss;
331   ss << "./ResultColetorSwipt/MonitorSniffTx_" << node << ".csv";
332   // ss << "./ResultColetorSwipt/MonitorSniffTx.csv";
333   static std::fstream f(ss.str().c_str(), std::ios::out);
334   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " << i
      ↪  <<  ", " << sources.Get (i)->GetRemainingEnergy() << std::endl;
335   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", " << packet
      ↪  << ", "
336     << channelFreqMhz << ", "
337     << channelNumber << ", "
338     << rate << ", "
339     << isShortPreamble << ", "
340     << txvector << std::endl;
341   // cout << txvector << std::endl;
342   onPktsTx[node]++;
343 }
344
345 // OnPhyTxDrop(Ptr<const Packet> packet)
346 template <int node>
347 void OnPhyTxDrop(std::string context, Ptr<const Packet> packet, DropReason reason)
348 {
349   onPhyTxDrop[node]++;
350   std::stringstream ss;
351   // ss << "./ResultColetorSwipt/OnPhyTxDrop_" << node << ".csv";
352   ss << "./ResultColetorSwipt/OnPhyTxDrop_" << node << ".csv";
353   static std::fstream f(ss.str().c_str(), std::ios::out);
354   // f << Simulator::Now().GetSeconds() << ", " << packet << ", " << reason << std::endl;
355   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", " << packet
      ↪  << ", " << dropReason[reason] << std::endl;
356 }
357
358 // OnPhyRxDrop(Ptr<const Packet> packet)
359 template <int node>
360 void OnPhyRxDrop(std::string context, Ptr<const Packet> packet, DropReason reason)
361 {
```

```
362   onPhyRxDrop[node]++;
363   std::stringstream ss;
364   ss << "./ResultColetorSwipt/OnPhyRxDrop_" << node << ".csv";
365   static std::fstream f(ss.str().c_str(), std::ios::out);
366   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", " << packet
      ↪  << ", " << dropReason[reason] << std::endl;
367   // f << Simulator::Now().GetSeconds() << ", " << packet << std::endl;
368 }
369
370 template <int node>
371 void OnMacPacketDropped(std::string context, Mac48Address address)
372 {
373   onMacDrop[node]++;
374   std::stringstream ss;
375   ss << "./ResultColetorSwipt/OnMacPacketDropped_" << node << ".csv";
376   static std::fstream f(ss.str().c_str(), std::ios::out);
377   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " << i
      ↪  <<  ", " << sources.Get (i)->GetRemainingEnergy() << std::endl;
378   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() <<
      ↪  std::endl;
379   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  <<
      ↪  address << std::endl;
380 }
381
382 template <int node>
383 void OnMacPacketDropped2(std::string context, Ptr<const Packet> packet, DropReason reason)
384 {
385   onMacDrop2[node]++;
386   std::stringstream ss;
387   ss << "./ResultColetorSwipt/OnMacPacketDropped2_" << node << ".csv";
388   static std::fstream f(ss.str().c_str(), std::ios::out);
389   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " << i
      ↪  <<  ", " << sources.Get (i)->GetRemainingEnergy() << std::endl;
390   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() <<
      ↪  std::endl;
391   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  << packet
      ↪  << ", " << dropReason[reason] << std::endl;
392 }
393
394 template <int node>
395 void OnCollision(std::string context, uint32_t nrOfBackoffSlots)
396 {
397   onCollision[node]++;
398   std::stringstream ss;
399   ss << "./ResultColetorSwipt/OnCollision_" << node << ".csv";
400   static std::fstream f(ss.str().c_str(), std::ios::out);
401   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " << i
      ↪  <<  ", " << sources.Get (i)->GetRemainingEnergy() << std::endl;
```

```
402   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() <<
      ↪  std::endl;
403   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  <<
      ↪  nrOfBackoffSlots << std::endl;
404   // cout << "# of colisions Node: " << node << " " << onCollision[node] << std::endl;
405 }
406
407 static inline std::string PrintReceivedPacket(Ptr<Socket> socket)
408 {
409   Address addr;
410
411   std::ostringstream oss;
412
413   while (socket->Recv())
414   {
415     socket->GetSockName(addr);
416     InetSocketAddress iaddr = InetSocketAddress::ConvertFrom(addr);
417     onUdpRx ++;
418
419     //oss << "Received one packet!  Socket: " << iaddr.GetIpv4() << " port: " <<
      ↪  iaddr.GetPort();
420   }
421
422   return oss.str();
423 }
424
425 static void ReceivePacket(Ptr<Socket> socket)
426 {
427   PrintReceivedPacket(socket);
428   //NS_LOG_UNCOND(PrintReceivedPacket(socket));
429 }
430
431 static void GenerateTraffic(Ptr<Socket> socket, uint32_t pktSize,
432               uint32_t pktCount, Time pktInterval)
433 {
434   if (pktCount > 0)
435   {
436     onUdpTx ++;
437     socket->Send(Create<Packet>(pktSize));
438     Simulator::Schedule(pktInterval, &GenerateTraffic,
439               socket, pktSize, pktCount - 1, pktInterval);
440   }
441   else
442   {
443     socket->Close();
444   }
445 }
446
```

```
447
448 void PhyStateTraced(std::string context, Time start, Time duration, enum WifiPhy::State
    ↪  state)
449 {
450   std::stringstream ss;
451   // ss << "./ResultColetorSwipt/interference6/subdir/state_" << node << "_" <<
    ↪  config.simulationTime << "_" << config.psFactor << ".csv";
452   ss << "./ResultColetorSwipt/state_" << ".csv";
453   static std::fstream f(ss.str().c_str(), std::ios::out);
454   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " <<
    ↪  state << ", " << duration.GetSeconds() << ", " << node << std::endl;
455   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  << state
    ↪  << ", " << duration.GetSeconds() << ", " << std::endl;
456   //f << start.GetSeconds () << ", " << dist<< ", " <<  psFactor << ", "  << state << ", "
    ↪  << duration.GetSeconds() << ", " << node << std::endl;
457
458   // std::cout << std::fixed << std::setprecision(9) << Simulator::Now ().GetSeconds () <<
    ↪  "s, Node:"<< node << ", state=" << state << ", duration=" << duration << ", context:"
    ↪  << context << std::endl;
459 }
460
461 void RemainingEnergy(double oldValue, double newValue)
462 {
463   std::stringstream ss;
464   // ss << "./ResultColetorSwipt/interference6/subdir/remaining_energy_"<< node << "_" <<
    ↪  config.simulationTime << "_" << config.psFactor << ".csv";
465   ss << "./ResultColetorSwipt/battery_remaining_energy_0.csv";
466   static std::fstream f(ss.str().c_str(), std::ios::out);
467   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " <<
    ↪  oldValue - newValue << ", " << newValue << ", " << node << std::endl;
468   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  <<
    ↪  oldValue - newValue << ", " << newValue << ", " << std::endl;
469   // std::cout << Simulator::Now().GetSeconds () << "s, battery energy update=" << oldValue
    ↪  - newValue << "J, remaining energy=" << newValue << "J, node=" << node << std::endl;
470 }
471
472 /// Trace function for total energy consumption at node.
473 void TotalEnergy(double oldValue, double newValue)
474 {
475   consump = std::abs(newValue);
476   std::stringstream ss;
477   // ss << "./ResultColetorSwipt/interference6/subdir/energy_consumption_" <<  node << "_"
    ↪  << config.simulationTime << "_" << config.psFactor << ".csv";
478   ss << "./ResultColetorSwipt/energy_consumption_0.csv";
479   static std::fstream f(ss.str().c_str(), std::ios::out);
480   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " <<
    ↪  newValue - oldValue << ", " << newValue << ", " << node << std::endl;
481   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  <<
    ↪  newValue - oldValue << ", " << newValue << ", " << std::endl;
```

```
482   // std::cout << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() <<
      ↪  "s, energy consumed by WifiRadio=" << newValue << "J" << std::endl;
483 }
484
485 /// Trace function for the power harvested by the energy harvester.
486
487 void HarvestedPower(double oldValue, double newValue)
488 {
489   std::stringstream ss;
490   // ss << "./ResultColetorSwipt/interference6/subdir/harvested_power_" << node << "_" <<
      ↪  config.simulationTime << "_" << config.psFactor << ".csv";
491   ss << "./ResultColetorSwipt/harvested_power_0.csv";
492   static std::fstream f(ss.str().c_str(), std::ios::out);
493   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  <<
      ↪  newValue - oldValue << ", " << newValue << std::endl;
494   // std::cout << std::fixed << std::setprecision(9) << Simulator::Now().GetNanoSeconds()
      ↪  << "ns, harvested power=" << harvestedPower << "W" << std::endl;
495   // std::cout << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() <<
      ↪  "s" << ", Function Call harvested power= "<< newValue << "W" << std::endl;
496   harvestedPowerCounter++;
497   g_harvestedPowerAvg = std::abs(newValue);
498 }
499
500 /// Trace function for the total energy harvested by the node.
501 void TotalEnergyHarvested(double oldValue, double newValue)
502 {
503   std::stringstream ss;
504   // ss << "./ResultColetorSwipt/interference6/subdir/total_energy_harvested_" << node <<
      ↪  "_" << config.simulationTime << "_" << config.psFactor << ".csv";
505   ss << "./ResultColetorSwipt/total_energy_harvested_0.csv";
506   static std::fstream f(ss.str().c_str(), std::ios::out);
507   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  <<
      ↪  newValue - oldValue << ", " << newValue << ", " << std::endl;
508   // harvestedEnergy[node] = newValue-oldValue;
509   //std::cout << Simulator::Now().GetSeconds() << "s, total energy harvested=" << newValue
      ↪  << "J, nodeId=" << node << std::endl;
510
511   harvestedEnergyCounter++;
512   //g_harvestedEnergyAvg[node] = std::abs(newValue);
513   g_harvestedEnergyAvg = newValue;
514 }
515
516 void PopulateArpCache()
517 {
518   Ptr<ArpCache> arp = CreateObject<ArpCache>();
519   arp->SetAliveTimeout(Seconds(3600 * 24 * 365));
520   for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
521   {
```

```
522     Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol>();
523     NS_ASSERT(ip != 0);
524     ObjectVectorValue interfaces;
525     ip->GetAttribute("InterfaceList", interfaces);
526     for (ObjectVectorValue::Iterator j = interfaces.Begin(); j != interfaces.End(); j++)
527     {
528       Ptr<Ipv4Interface> ipIface =
529         (j->second)->GetObject<Ipv4Interface>();
530       NS_ASSERT(ipIface != 0);
531       Ptr<NetDevice> device = ipIface->GetDevice();
532       NS_ASSERT(device != 0);
533       Mac48Address addr = Mac48Address::ConvertFrom(device->GetAddress());
534       for (uint32_t k = 0; k < ipIface->GetNAddresses(); k++)
535       {
536         Ipv4Address ipAddr = ipIface->GetAddress(k).GetLocal();
537         if (ipAddr == Ipv4Address::GetLoopback())
538           continue;
539         ArpCache::Entry *entry = arp->Add(ipAddr);
540         entry->MarkWaitReply(0);
541         entry->MarkAlive(addr);
542         // std::cout << "Arp Cache: Adding the pair (" << addr << ","
543         //       << ipAddr << ")" << std::endl;
544       }
545     }
546   }
547   for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
548   {
549     Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol>();
550     NS_ASSERT(ip != 0);
551     ObjectVectorValue interfaces;
552     ip->GetAttribute("InterfaceList", interfaces);
553     for (ObjectVectorValue::Iterator j = interfaces.Begin();
554        j != interfaces.End(); j++)
555     {
556       Ptr<Ipv4Interface> ipIface =
557         (j->second)->GetObject<Ipv4Interface>();
558       ipIface->SetAttribute("ArpCache", PointerValue(arp));
559     }
560   }
561 }
562
563 uint16_t ngroup;
564 uint16_t nslot;
565 RPSVector configureRAW(RPSVector rpslist, string RAWConfigFile)
566 {
567   uint16_t NRPS = 0;
568   uint16_t NRAWPERBEACON = 0;
569   uint16_t Value = 0;
```

```
570   uint32_t page = 0;
571   uint32_t aid_start = 0;
572   uint32_t aid_end = 0;
573   uint32_t rawinfo = 0;
574
575   ifstream myfile(RAWConfigFile);
576   // 1. get info from config file
577
578   // 2. define RPS
579   if (myfile.is_open())
580   {
581     myfile >> NRPS;
582     NRPS = 1;
583     int totalNumSta = 0;
584     for (uint16_t kk = 0; kk < NRPS; kk++) // number of beacons covering all raw groups
585     {
586       RPS *m_rps = new RPS;
587       myfile >> NRAWPERBEACON;
588       ngroup = NRAWPERBEACON;
589       for (uint16_t i = 0; i < NRAWPERBEACON; i++) // raw groups in one beacon
590       {
591         // RPS *m_rps = new RPS;
592         RPS::RawAssignment *m_raw = new RPS::RawAssignment;
593
594         myfile >> Value;
595         m_raw->SetRawControl(Value); // support paged STA or not
596         myfile >> Value;
597         m_raw->SetSlotCrossBoundary(Value);
598         myfile >> Value;
599         m_raw->SetSlotFormat(Value);
600         myfile >> Value;
601         m_raw->SetSlotDurationCount(Value);
602         myfile >> Value;
603         nslot = Value;
604         m_raw->SetSlotNum(Value);
605         myfile >> page;
606         myfile >> aid_start;
607         myfile >> aid_end;
608         rawinfo = (aid_end << 13) | (aid_start << 2) | page;
609         m_raw->SetRawGroup(rawinfo);
610         totalNumSta = aid_end - aid_start + 1;
611         m_rps->SetRawAssignment(*m_raw);
612         delete m_raw;
613       }
614       rpslist.rpsset.push_back(m_rps);
615       // config.nRawGroupsPerRpsList.push_back(NRAWPERBEACON);
616     }
617     myfile.close();
```

```
618       NRawSta = totalNumSta;
619       // rpslist.rpsset[rpslist.rpsset.size() - 1]->GetRawAssigmentObj(
620       //    NRAWPERBEACON - 1).GetRawGroupAIDEnd();
621     }
622     else
623       cout << "Unable to open RAW configuration file \n";
624
625     return rpslist;
626   }
627
628   void configurePageSlice(void)
629   {
630     pageS.SetPageindex(pageIndex);
631     pageS.SetPagePeriod(pagePeriod);    // 2 TIM groups between DTIMs
632     pageS.SetPageSliceLen(pageSliceLength); // each TIM group has 1 block (2 blocks in 2 TIM
      ↪  groups)
633     pageS.SetPageSliceCount(pageSliceCount);
634     pageS.SetBlockOffset(blockOffset);
635     pageS.SetTIMOffset(timOffset);
636     // std::cout << "pageIndex=" << (int)pageIndex << ", pagePeriod=" << (int)pagePeriod <<
      ↪  ", pageSliceLength=" << (int)pageSliceLength << ", pageSliceCount=" <<
      ↪  (int)pageSliceCount << ", blockOffset=" << (int)blockOffset << ", timOffset=" <<
      ↪  (int)timOffset << std::endl;
637     //   page 0
638     //   8 TIM(page slice) for one page
639     //   4 block (each page)
640     //   8 page slice
641     //   both offset are 0
642   }
643
644   void configureTIM(void)
645   {
646     tim.SetPageIndex(pageIndex);
647     if (pageSliceCount)
648       tim.SetDTIMPeriod(pageSliceCount); // not necessarily the same
649     else
650       tim.SetDTIMPeriod(1);
651
652     // std::cout << "DTIM period=" << (int)pagePeriod << std::endl;
653   }
654
655   bool check(uint16_t aid, uint32_t index)
656   {
657     uint8_t block = (aid >> 6) & 0x001f;
658     NS_ASSERT(pageS.GetPageSliceLen() > 0);
659     // uint8_t toTim = (block - pageS.GetBlockOffset()) % pageS.GetPageSliceLen();
660     if (index == pageS.GetPageSliceCount() - 1 && pageS.GetPageSliceCount() != 0)
661     {
```

```
662       // the last page slice has 32 - the rest blocks
663       return (block <= 31) && (block >= index * pageS.GetPageSliceLen());
664     }
665     else if (pageS.GetPageSliceCount() == 0)
666     {
667       return true;
668     }
669
670     return (block >= index * pageS.GetPageSliceLen()) && (block < (index + 1) *
        ↪  pageS.GetPageSliceLen());
671   }
672
673   void checkRawAndTimConfiguration(void)
674   {
675     // std::cout << "Checking RAW and TIM configuration..." << std::endl;
676     bool configIsCorrect = true;
677     NS_ASSERT(rps.rpsset.size());
678     // Number of page slices in a single page has to equal number of different RPS elements
        ↪  because
679     // If #PS > #RPS, the same RPS will be used in more than 1 PS and that is wrong because
680     // each PS can accommodate different AIDs (same RPS means same stations in RAWs)
681     if (pageSliceCount)
682     {
683       NS_ASSERT(pagePeriod == rps.rpsset.size());
684     }
685     for (uint32_t j = 0; j < rps.rpsset.size(); j++)
686     {
687       uint32_t totalRawTime = 0;
688       for (uint32_t i = 0; i < rps.rpsset[j]->GetNumberOfRawGroups(); i++)
689       {
690         totalRawTime += (120 * rps.rpsset[j]->GetRawAssigmentObj(i).GetSlotDurationCount() +
          ↪  500) * rps.rpsset[j]->GetRawAssigmentObj(i).GetSlotNum();
691         auto aidStart = rps.rpsset[j]->GetRawAssigmentObj(i).GetRawGroupAIDStart();
692         auto aidEnd = rps.rpsset[j]->GetRawAssigmentObj(i).GetRawGroupAIDEnd();
693         configIsCorrect = check(aidStart, j) && check(aidEnd, j);
694         // AIDs in each RPS must comply with TIM in the following way:
695         // TIM0: 1-63; TIM1: 64-127; TIM2: 128-191; ...; TIM32: 1983-2047
696         // If RPS that belongs to TIM0 includes other AIDs (other than range [1-63])
          ↪  configuration is incorrect
697         NS_ASSERT(configIsCorrect);
698       }
699       NS_ASSERT(totalRawTime <= BeaconInterval);
700     }
701   }
702
703   // assumes each TIM has its own beacon - doesn't need to be the case as there has to be
      ↪  only PageSliceCount beacons between DTIMs
704   /*
```

```
705 void onChannelTransmission(Ptr<NetDevice> senderDevice, Ptr<Packet> packet)
706 {
707   int rpsIndex = currentRps - 1;
708   int rawGroup = currentRawGroup - 1;
709   int slotIndex = currentRawSlot - 1;
710   // cout << rpsIndex << "    " << rawGroup << "    " << slotIndex << "    " << endl;
711
712   uint64_t iSlot = slotIndex;
713   if (rpsIndex > 0)
714     for (int r = rpsIndex - 1; r >= 0; r--)
715       for (int g = 0; g < rps.rpsset[r]->GetNumberOfRawGroups(); g++)
716         iSlot += rps.rpsset[r]->GetRawAssigmentObj(g).GetSlotNum();
717
718   if (rawGroup > 0)
719     for (int i = rawGroup - 1; i >= 0; i--)
720       iSlot += rps.rpsset[rpsIndex]->GetRawAssigmentObj(i).GetSlotNum();
721
722   if (rpsIndex >= 0 && rawGroup >= 0 && slotIndex >= 0)
723   {
724     if (senderDevice->GetAddress() == apDevice.Get(0)->GetAddress())
725     {
726       // from AP
727       transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval[iSlot] +=
      ↪  packet->GetSerializedSize();
728     }
729     else
730     {
731       // from STA
732       transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval[iSlot] +=
      ↪  packet->GetSerializedSize();
733     }
734   }
735   std::cout << "------------- packetSerializedSize = " << packet->GetSerializedSize() <<
   ↪  std::endl;
736   std::cout << "------------- txAP[" << iSlot <<"] = " <<
   ↪  transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval[iSlot] << std::endl;
737   std::cout << "------------- txSTA[" << iSlot <<"] = " <<
   ↪  transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval[iSlot] << std::endl;
738 }
739 */
740
741 void RpsIndexTrace(uint16_t oldValue, uint16_t newValue)
742 {
743   currentRps = newValue;
744   //cout << "RPS: " << newValue << " at " << Simulator::Now().GetMicroSeconds() << endl;
745 }
746
747 void RawGroupTrace(uint8_t oldValue, uint8_t newValue)
```

```
748 {
749   currentRawGroup = newValue;
750   // cout << "  group " << std::to_string(newValue) << " at " <<
      ↪  Simulator::Now().GetMicroSeconds() << endl;
751 }
752
753 void RawSlotTrace(uint8_t oldValue, uint8_t newValue)
754 {
755   currentRawSlot = newValue;
756   // cout << "    slot " << std::to_string(newValue) << " at " <<
      ↪  Simulator::Now().GetMicroSeconds() << endl;
757 }
758
759 // void
760 // TxCallback (Ptr<CounterCalculator<uint32_t> > datac, std::string path, Ptr<const Packet>
    ↪  packet)
761 //{
762 //  NS_LOG_INFO ("Sent frame counted in " << datac->GetKey ());
763 //  datac->Update ();
764 //  // end TxCallback
765 // }
766
767 template <int node>
768 void OnTransmissionWillCrossRAWBoundary(std::string context, Time txDuration, Time
    ↪  remainingTimeInRawSlot)
769 {
770   txXRawBound[node]++;
771 }
772
773 template <int node>
774 void TxRawSlot(std::string context, uint16_t oldValue, uint16_t newValue)
775 {
776   nTxRawSlot[node] = newValue;
777   //cout << "Node " << node << " transmitted " << nTxRawSlot[node] << "pkts during Raw
      ↪  Slot" << endl;
778 }
779
780 // template <int node>
781 void OnAPPacketToTransmitReceived(string context, Ptr<const Packet> packet,
782                   Mac48Address to, bool isScheduled, bool isDuringSlotOfSTA,
783                   Time timeLeftInSlot)
784 {
785
786   if (isScheduled)
787   {
788     NumberOfAPScheduledPacketForNodeInNextSlot++;
789     std::cout << "AP: # Pkts for node next RawSlot: "
790           << NumberOfAPScheduledPacketForNodeInNextSlot << std::endl;
```

```
791   }
792   else
793   {
794     NumberOfAPSentPacketForNodeImmediately++;
795     APTotalTimeRemainingWhenSendingPacketInSameSlot += timeLeftInSlot.GetSeconds();
796 
797     std::cout << "AP: # Pkts for node right now: "
798           << NumberOfAPSentPacketForNodeImmediately
799           << ", time left RawSlot: "
800           << NumberOfAPSentPacketForNodeImmediately << std::endl;
801   }
802 }
803 
804 template <int node>
805 void OnQueueDrop(Ptr<const Packet> packet, DropReason reason)
806 {
807   std::stringstream ss;
808   ss << "./ResultColetorSwipt/OnQueuePacketDropped_" << node << ".csv";
809   static std::fstream f(ss.str().c_str(), std::ios::out);
810   f << Simulator::Now().GetSeconds() << ", " << dist<< ", " <<  psFactor << ", "  << packet
      ↪  << ", " << dropReason[reason] << std::endl;
811 
812   onQueueDrop[node]++;
813 }
814 
815 typedef std::vector<Ptr<WifiMacQueue>> QueueVector;
816 QueueVector m_queue;
817 QueueVector m_queue2;
818 QueueVector BEqueue;
819 
820 void updateNodesQueueLength(NetDeviceContainer devices, NetDeviceContainer apDevice)
821 {
822   Ptr<NetDevice> netDe = devices.Get (0);
823   Ptr<WifiNetDevice> wifiDe = netDe->GetObject<WifiNetDevice> ();
824 
825   PointerValue ptr1;
826   wifiDe->GetMac ()->GetAttribute("BE_EdcaTxopN", ptr1);
827   Ptr<EdcaTxopN> BE_edca = ptr1.Get<EdcaTxopN> ();
828   PointerValue ptr2;
829   BE_edca->GetAttribute ("Queue", ptr2);
830   Ptr<WifiMacQueue> BE_edca_queue = ptr2.Get<WifiMacQueue> ();
831   m_queue.push_back (BE_edca_queue);
832 
833   Ptr<NetDevice> apNetDe;
834   Ptr<WifiNetDevice> wifiApDe;
835   apNetDe = apDevice.Get(0);
836   wifiApDe = apNetDe->GetObject<WifiNetDevice> ();
837   PointerValue apPtr1;
```

```
838   wifiApDe->GetMac ()->GetAttribute("BE_EdcaTxopN", apPtr1);
839   Ptr<EdcaTxopN> apBE_edca = apPtr1.Get<EdcaTxopN> ();
840   PointerValue apPtr2;
841   apBE_edca->GetAttribute("Queue", apPtr2);
842   Ptr<WifiMacQueue> apBE_edca_queue = apPtr2.Get<WifiMacQueue> ();
843   m_queue2.push_back (apBE_edca_queue);
844 
845   Simulator::Schedule(MicroSeconds (500000), &updateNodesQueueLength, devices, apDevice);
846 }
847 
848 double
849 DbmToW (double dbm)
850 {
851   double mw = std::pow (10.0,dbm/10.0);
852   return mw/1000.0;
853 }
854 
855 double
856 WToDbm (double w)
857 {
858   //double dbm = std::log10 (w * 1000.0) * 10.0;
859   double dbm = std::log10 (w) * 10.0;
860   return dbm;
861 }
862 
863 int main(int argc, char *argv[])
864 {
865   std::string trafficType = "udp";
866 
867   double simulationTime = 300.0;
868   double coolDown = 5.0;
869 
870   WifiPhyStandard standard = WIFI_PHY_STANDARD_80211ah;
871   uint32_t PpacketSize = 1000; // utilizar 1000 bytes
872   uint32_t headerLength = 66;
873   const uint32_t tcpPacketSize = 1448;
874 
875   double ccaMode1Threshold = -113.0;
876 
877   double startdist= 1.5; //  alterar para10m
878   double stopdist= 10.5; //  alterar para10m
879 
880   double delta = 3.2;       // microseconds
881   uint32_t IpacketSize = 1000; // bytes
882 
883   bool verbose = false;
884 
885   if (trafficType == "udp")
```

```
886   {
887     payloadSize = 1024; // 1000 bytes IPv4
888
889   }
890   else if (trafficType == "udpsocket")
891   {
892     payloadSize = 1000; // 1000 bytes IPv4
893   }
894   else if (trafficType == "udpecho")
895   {
896     payloadSize = 1024; // 1000 bytes IPv4
897     //payloadSize = 972; // 1000 bytes IPv4
898   }
899   else if (trafficType == "tcp")
900   {
901     payloadSize = 1448; // 1500 bytes IPv6
902     // payloadSize = 1500; // 1500 bytes IPv6
903     Config::SetDefault("ns3::TcpSocket::SegmentSize", UintegerValue(payloadSize));
904   }
905
906
907   std::cout << "Std: IEEE 802.11ah"
908         << "; TrafficType: " << trafficType
909         //<< "; EnergyDetecThreshold: " << energyDetectionThreshold
910         << "; simulationTime: " << simulationTime
911         << "s; AP txDbm: " << 30.0
912         << "dBm; Packet Size: " << payloadSize
913         << "Bytes\n"
914         << std::endl;
915
916   std::cout << "Dist"
917         << std::setw(7) << "Data"
918         << std::setw(8) << "Tput"
919         << std::setw(16) << "UDP"
920         << std::setw(26) << "Frames"
921         << std::setw(15) << "PS"
922         << std::setw(15) << "PowH"
923         << std::setw(9) << "EneH"
924         << std::setw(11) << "RSSI"
925         << std::setw(9) << "Noise"
926         << std::setw(8) << "SNR"
927         << std::setw(8) << "CSR"
928         << std::endl;
929
930   std::cout << "(m)"
931         << std::setw(8) << "Rate"
932         << std::setw(9) << "(kbps)"
933         << std::setw(7) << "Tx"
```

```
934         << std::setw(8) << "Rx"
935         << std::setw(8) << "Lost"
936         << std::setw(8) << "Tx"
937         << std::setw(8) << "Rx"
938         << std::setw(8) << "Lost"
939         << std::setw(11) << "Factor"
940         << std::setw(13) << "(W)"
941         << std::setw(9) << "(J)"
942         << std::setw(12) << "(dBm)"
943         << std::setw(8) << "(dBm)"
944         << std::setw(9) << "(dB)"
945         << std::endl;
946
947   std::vector <std::string> modes;
948
949   modes.push_back ("OfdmRate150KbpsBW1MHz");
950   modes.push_back ("OfdmRate300KbpsBW1MHz");
951     modes.push_back ("OfdmRate600KbpsBW1MHz");
952     modes.push_back ("OfdmRate900KbpsBW1MHz");
953     modes.push_back ("OfdmRate1_2MbpsBW1MHz");
954     modes.push_back ("OfdmRate1_8MbpsBW1MHz");
955     modes.push_back ("OfdmRate2_4MbpsBW1MHz");
956     modes.push_back ("OfdmRate2_7MbpsBW1MHz");
957     modes.push_back ("OfdmRate3MbpsBW1MHz");
958     modes.push_back ("OfdmRate3_6MbpsBW1MHz");
959     modes.push_back ("OfdmRate4MbpsBW1MHz");
960   modes.push_back ("OfdmRate650KbpsBW2MHz");
961     modes.push_back ("OfdmRate1_3MbpsBW2MHz");
962     modes.push_back ("OfdmRate1_95MbpsBW2MHz");
963     modes.push_back ("OfdmRate2_6MbpsBW2MHz");
964     modes.push_back ("OfdmRate3_9MbpsBW2MHz");
965     modes.push_back ("OfdmRate5_2MbpsBW2MHz");
966     modes.push_back ("OfdmRate5_85MbpsBW2MHz");
967     modes.push_back ("OfdmRate6_5MbpsBW2MHz");
968     modes.push_back ("OfdmRate7_8MbpsBW2MHz");
969
970   for (uint32_t index = 0; index < 1; index = index + 1)
971   {
972     for (dist = startdist; dist <= stopdist; dist += 1.0)
973     {
974       int ps = 0;
975       for (psFactor = 0.0; psFactor < 1.0;)
976       {
977                 uint32_t BeaconInterval = 102400;
978         double txPower = 0.0;
979         double apTxPower = 30.0;
980         //double apTxGain = 3.0;
981         double apTxGain = 6.0;
```

```
982         //double apTxGain = 0.0;
983         //double rxAntennaGain = 12.0;
984         //double rxAntennaGain = 12.0;
985         double rxAntennaGain = 6.0;
986 
987         double batteryLowThreshold = 0.10;
988         double batteryHighThreshold = 0.40;
989         double sourceInitialEnergyJ = 5.0;
990 
991         pagePeriod = 1;    //  Number of Beacon Intervals between DTIM beacons that carry
            ↪  Page Slice element for the associated page
992         pageIndex = 0;
993         pageSliceLength = 1; //  Number of blocks in each TIM for the associated page
            ↪  except for the last TIM (1-31) (value 0 is reserved);
994         pageSliceCount = 0;  //  Number of TIMs in a single page period (1-31)
995         blockOffset = 0;    //  The 1st page slice starts with the block with blockOffset
            ↪  number
996         timOffset = 0;      //  Offset in number of Beacon Intervals from the DTIM that
            ↪  carries the first page slice of the page
997 
998         // RPSVector rps;
999 
1000        rps = configureRAW(rps, RAWConfigFile);
1001 
1002        Nsta = NRawSta;
1003 
1004        configurePageSlice();
1005        configureTIM();
1006        checkRawAndTimConfiguration();
1007 
1008        bool s1g1MfieldEnabled = false;
1009                bool tracing = false;
1010 
1011        uint32_t numPackets = 4294967295u;
1012        uint32_t seed = 1;
1013 
1014                CommandLine cmd;
1015              cmd.AddValue("simulationTime", "Duração da Simulação", simulationTime);
1016              cmd.AddValue("coolDown", "Tempo adicional para p/ esvaziamento dos buffers",
                  ↪  coolDown);
1017              cmd.AddValue("payloadSize", "Tamanho da carga útil de um pacote UDP",
                  ↪  payloadSize);
1018              cmd.AddValue("verbose", "Habilitar logs da simulação", verbose);
1019              cmd.AddValue("tracing", "Habilitar geração de PCAP", tracing);
1020              cmd.Parse(argc, argv);
1021 
1022 
1023        totalRawSlots = 0;
```

```
1024        uint32_t totalRawGroups(0);
1025        for (int i = 0; i < rps.rpsset.size(); i++)
1026        {
1027          int nRaw = rps.rpsset[i]->GetNumberOfRawGroups();
1028          totalRawGroups += nRaw;
1029          // cout << "Total raw groups after rps " << i << " is " << totalRawGroups <<
              ↪  endl;
1030          for (int j = 0; j < nRaw; j++)
1031          {
1032            totalRawSlots += rps.rpsset[i]->GetRawAssigmentObj(j).GetSlotNum();
1033            // cout << "Total slots after group " << j << " is " << totalRawSlots << endl;
1034          }
1035        }
1036
1037        onUdpRx = 0;
1038        onUdpTx = 0;
1039        onUdpEchoRx = 0;
1040        NumberOfAPScheduledPacketForNodeInNextSlot = 0;
1041        NumberOfAPSentPacketForNodeImmediately = 0;
1042        APTotalTimeRemainingWhenSendingPacketInSameSlot = 0.0;
1043
1044        txrate = 0;
1045        csr = 0.0;
1046        ber = 0.0;
1047        consump = 0.0;
1048        harvestedEnergyCounter = 0;
1049        g_harvestedEnergyAvg = 0.0;
1050        g_harvestedPowerAvg = 0.0;
1051        harvestedPowerCounter = 0;
1052
1053        for (uint32_t j = 0; j <= 2 ; j++)
1054        {
1055          onPhyTxDrop[j] = 0;
1056          onPhyRxDrop[j] = 0;
1057          onMacDrop[j] = 0;
1058          onMacDrop2[j] = 0;
1059          onCollision[j] = 0;
1060          onPktsRx[j] = 0;
1061          onPktsTx[j] = 0;
1062          phyTxEnd[j] = 0;
1063          phyRxEnd[j] = 0;
1064          harvestedEnergy[j] = 0.0;
1065          g_signalDbmAvg[j] = 0.0;
1066          g_noiseDbmAvg[j] = 0.0;
1067          txXRawBound[j] = 0.0;
1068          nTxRawSlot[j] = 0.0;
1069          g_noiseDbmAvg[j] = 0.0;
1070          onQueueDrop[j] = 0.0;
```

```
1071        }
1072
1073        RngSeedManager::SetSeed(seed);
1074
1075        StringValue DataRate;
1076        uint32_t datarate = 0;
1077        uint32_t channelWidth = 0;
1078        if (index == 0)
1079        {
1080          DataRate = StringValue("OfdmRate150KbpsBW1MHz");
1081          datarate = 150000;
1082          channelWidth = 1;
1083        }
1084        else if (index == 1)
1085        {
1086          DataRate = StringValue("OfdmRate300KbpsBW1MHz");
1087          datarate = 300000;
1088          channelWidth = 1;
1089        }
1090        else if (index == 2)
1091        {
1092          DataRate = StringValue("OfdmRate600KbpsBW1MHz");
1093          datarate = 600000;
1094          channelWidth = 1;
1095        }
1096        else if (index == 3)
1097        {
1098          DataRate = StringValue("OfdmRate900KbpsBW1MHz");
1099          datarate = 900000;
1100          channelWidth = 1;
1101        }
1102        else if (index == 4)
1103        {
1104          DataRate = StringValue("OfdmRate1_2MbpsBW1MHz");
1105          datarate = 1200000;
1106          channelWidth = 1;
1107        }
1108        else if (index == 5)
1109        {
1110          DataRate = StringValue("OfdmRate1_8MbpsBW1MHz");
1111          datarate = 1800000;
1112          channelWidth = 1;
1113        }
1114        else if (index == 6)
1115        {
1116          DataRate = StringValue("OfdmRate2_4MbpsBW1MHz");
1117          datarate = 2400000;
1118          channelWidth = 1;
```

```
1119          }
1120          else if (index == 7)
1121          {
1122            DataRate = StringValue("OfdmRate2_7MbpsBW1MHz");
1123            datarate = 2700000;
1124            channelWidth = 1;
1125          }
1126          else if (index == 8)
1127          {
1128            DataRate = StringValue("OfdmRate3MbpsBW1MHz");
1129            datarate = 3000000;
1130            channelWidth = 1;
1131          }
1132          else if (index == 9)
1133          {
1134            DataRate = StringValue("OfdmRate3_6MbpsBW1MHz");
1135            datarate = 3600000;
1136            channelWidth = 1;
1137          }
1138          else if (index == 10)
1139          {
1140            DataRate = StringValue("OfdmRate4MbpsBW1MHz");
1141            datarate = 4000000;
1142            channelWidth = 1;
1143          }
1144          else if (index == 11)
1145          {
1146            DataRate = StringValue("OfdmRate650KbpsBW2MHz");
1147            datarate = 650000;
1148            channelWidth = 2;
1149          }
1150          else if (index == 12)
1151          {
1152            DataRate = StringValue("OfdmRate1_3MbpsBW2MHz");
1153            datarate = 1300000;
1154            channelWidth = 2;
1155          }
1156          else if (index == 13)
1157          {
1158            DataRate = StringValue("OfdmRate1_95MbpsBW2MHz");
1159            datarate = 1950000;
1160            channelWidth = 2;
1161          }
1162          else if (index == 14)
1163          {
1164            DataRate = StringValue("OfdmRate2_6MbpsBW2MHz");
1165            datarate = 2600000;
1166            channelWidth = 2;
```

```
1167        }
1168        else if (index == 15)
1169        {
1170          DataRate = StringValue("OfdmRate3_9MbpsBW2MHz");
1171          datarate = 3900000;
1172          channelWidth = 2;
1173        }
1174        else if (index == 16)
1175        {
1176          DataRate = StringValue("OfdmRate5_2MbpsBW2MHz");
1177          datarate = 5200000;
1178          channelWidth = 2;
1179        }
1180        else if (index == 17)
1181        {
1182          DataRate = StringValue("OfdmRate5_85MbpsBW2MHz");
1183          datarate = 5850000;
1184          channelWidth = 2;
1185        }
1186        else if (index == 18)
1187        {
1188          DataRate = StringValue("OfdmRate6_5MbpsBW2MHz");
1189          datarate = 6500000;
1190          channelWidth = 2;
1191        }
1192        else if (index == 19)
1193        {
1194          DataRate = StringValue("OfdmRate7_8MbpsBW2MHz");
1195          datarate = 7800000;
1196          channelWidth = 2;
1197        }
1198
1199        if (channelWidth == 1)
1200        {
1201          s1g1MfieldEnabled = true;
1202        }
1203        else
1204        {
1205          s1g1MfieldEnabled = false;
1206        }
1207
1208        double interPktInterval = payloadSize * 8.0 / datarate;
1209
1210        NodeContainer c;
1211        NodeContainer wifiApNode;
1212
1213
1214        c.Create(Nsta);
```

```
1215        wifiApNode.Create(1);
1216
1217        YansWifiChannelHelper wifiChannel;
1218        wifiChannel.AddPropagationLoss("ns3::LogDistancePropagationLossModel", // L (d) = 8
            ↪  + 37.6 log(d) + 21 log(f/900MHz)
1219                       "Exponent", DoubleValue(3.76),       // L = L_0 + 10*n*log(d/d_0)
                           ↪  + 21 log(900MHz /900MHz)
1220                       "ReferenceLoss", DoubleValue(8.0),     // path loss at reference
                           ↪  dist(dB)
1221                       "ReferenceDistance", DoubleValue(1.0)); //  reference dist(m)
1222        //wifiChannel.AddPropagationLoss("ns3::FriisPropagationLossModel");
1223        wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");
1224        Ptr<YansWifiChannel> channel = wifiChannel.Create();
1225
1226
1227
1228              if (verbose)
1229              {
1230                //LogComponentEnable ("UdpServer", LOG_LEVEL_ALL );
1231                //LogComponentEnable ("UdpClient", LOG_LEVEL_ALL );
1232                //LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_ALL );
1233                //LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_ALL );
1234
1235                //LogComponentEnable ("EdcaTxopN", LOG_LEVEL_ALL );
1236                //LogComponentEnable ("ApWifiMac", LOG_LEVEL_ALL );
1237                //LogComponentEnable ("StaWifiMac", LOG_LEVEL_ALL );
1238                //LogComponentEnable ("WifiNetDevice", LOG_LEVEL_ALL );
1239                //LogComponentEnable ("S1gRawCtr", LOG_LEVEL_ALL );
1240                //LogComponentEnable ("pageSlice", LOG_LEVEL_ALL );
1241                //LogComponentEnable ("RPS", LOG_LEVEL_ALL );
1242                //LogComponentEnable ("TIM", LOG_LEVEL_ALL );
1243                //LogComponentEnable ("PropagationLossModel", LOG_LEVEL_ALL );
1244                //LogComponentEnable ("LiIonEnergySource", LOG_LEVEL_ALL );
1245                //LogComponentEnable ("WifiPhyStateHelper", LOG_LEVEL_ALL );
1246
1247                //LogComponentEnable ("EnergySource", LOG_LEVEL_ALL );
1248                // LogComponentEnable ("BasicEnergySource", LOG_LEVEL_ALL );
1249                // LogComponentEnable ("DeviceEnergyModel", LOG_LEVEL_ALL );
1250                // LogComponentEnable ("WifiRadioEnergyModel", LOG_LEVEL_ALL );
1251                LogComponentEnable ("EnergyHarvester", LOG_LEVEL_ALL );
1252                LogComponentEnable ("SwiptHarvester", LOG_LEVEL_ALL );
1253                // LogComponentEnable ("YansWifiPhy", LOG_LEVEL_ALL );
1254                // LogComponentEnable ("RPS", LOG_LEVEL_ALL );
1255                // LogComponentEnable ("YansWifiChannel", LOG_LEVEL_ALL );
1256                // LogComponentEnable ("WifiPhyStateHelper", LOG_LEVEL_ALL );
1257                // wifi.EnableLogComponents ();  // Turn on all Wifi logging
1258              }
1259
```

```
1260        WifiHelper wifi;
1261        // The below set of helpers will help us to put together the wifi NICs we want
1262
1263
1264        wifi.SetStandard(WIFI_PHY_STANDARD_80211ah);
1265
1266        wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
1267               "DataMode", DataRate,
1268               "ControlMode", DataRate);
1269
1270        YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default();
1271
1272        // Receiver Node PHY Configuration
1273        // set it to zero; otherwise, gain will be added
1274        wifiPhy.SetChannel(channel);
1275        wifiPhy.Set("ShortGuardEnabled", BooleanValue(false));
1276        wifiPhy.Set("STBCEnabled", BooleanValue(false));
1277        wifiPhy.Set("GreenfieldEnabled", BooleanValue(false));
1278        wifiPhy.Set("S1gShortfieldEnabled", BooleanValue(false));
1279        wifiPhy.Set("S1gLongfieldEnabled", BooleanValue(false));
1280        wifiPhy.Set("S1g1MfieldEnabled", BooleanValue(s1g1MfieldEnabled));
1281        wifiPhy.Set("LdpcEnabled", BooleanValue(false)); // Vrf a utilizacao
1282
1283        wifiPhy.Set("CcaMode1Threshold", DoubleValue(ccaMode1Threshold));
1284        //wifiPhy.Set("RxGain", DoubleValue(0.0));
1285        wifiPhy.Set("RxGain", DoubleValue(rxAntennaGain));
1286        wifiPhy.Set("TxGain", DoubleValue(0.0));
1287        wifiPhy.Set("ChannelWidth", UintegerValue(channelWidth));
1288        wifiPhy.Set("TxPowerLevels", UintegerValue(1));
1289        wifiPhy.Set("TxPowerStart", DoubleValue(txPower));
1290        wifiPhy.Set("TxPowerEnd", DoubleValue(txPower)); // Calc. Link Budget. pot no
        ↪  interference?
1291        wifiPhy.Set("RxNoiseFigure", DoubleValue(6.8));
1292
1293        //RF Receiver Sensitivity:
        ↪  https://www.asiarf.com/shop/halow-lora-iot/wi-fi-halow-sub-ghz-wireless-module-morse-mic
1294        if (datarate < 3250001) // -105 dBm sensitivity (1% PER) @ 3250 kbps BPSK
1295        {
1296          wifiPhy.Set("EnergyDetectionThreshold", DoubleValue(-104.0));
1297        }
1298        else // -100 dBm sensitivity (0.1% BER) @ 9750 kbps QPSK
1299        {
1300          wifiPhy.Set("EnergyDetectionThreshold", DoubleValue(-100.0));
1301        }
1302
1303        // NistErrorRateModel does not work for MCS2_8
1304        if (datarate != 7800000)
1305        {
```

```
1306            wifiPhy.SetErrorRateModel("ns3::NistErrorRateModel");
1307          }
1308          else
1309          {
1310            wifiPhy.SetErrorRateModel("ns3::YansErrorRateModel");
1311          }
1312
1313          // ns-3 supports RadioTap and Prism tracing extensions for 802.11b
1314          wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);
1315          //wifiPhy.SetPcapDataLinkType(YansWifiPhyHelper::DLT_IEEE802_11);
1316
1317          // Add a non-QoS upper mac, and disable rate control
1318          S1gWifiMacHelper wifiMac = S1gWifiMacHelper::Default();
1319          //NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
1320
1321          // Set it to StaWifiMac mode
1322          Ssid ssid = Ssid("ns380211ah");
1323          wifiMac.SetType("ns3::StaWifiMac",
1324                  "Ssid", SsidValue(ssid),
1325                  "ActiveProbing", BooleanValue(false),
1326                  "MaxMissedBeacons", UintegerValue (4294967295u));
1327
1328          NetDeviceContainer devices = wifi.Install(wifiPhy, wifiMac, c);
1329
1330                  Mac48Address macAddressNode = Mac48Address ("00:00:00:00:00:01");
1331                  Mac48Address macAddressAp = Mac48Address ("00:00:00:00:00:02");
1332
1333          c.Get(0)->GetDevice (0)->SetAddress(macAddressNode);
1334
1335
1336          wifiMac.SetType("ns3::ApWifiMac",
1337                  "Ssid", SsidValue(ssid),
1338                  "BeaconInterval", TimeValue(MicroSeconds(BeaconInterval)),
1339                  "NRawStations", UintegerValue(NRawSta),
1340                  "RPSsetup", RPSVectorValue(rps),
1341                  "PageSliceSet", pageSliceValue(pageS),
1342                  "TIMSet", TIMValue(tim));
1343
1344
1345          wifiPhy.Set("TxGain", DoubleValue(apTxGain));
1346          wifiPhy.Set("RxGain", DoubleValue(0.0));
1347          wifiPhy.Set("TxPowerLevels", UintegerValue(1));
1348          wifiPhy.Set("TxPowerEnd", DoubleValue(apTxPower));
1349          wifiPhy.Set("TxPowerStart", DoubleValue(apTxPower));
1350          wifiPhy.Set("RxNoiseFigure", DoubleValue(6.8));
1351          wifiPhy.Set("EnergyDetectionThreshold", DoubleValue(-110.0));
1352          wifiPhy.Set("CcaMode1Threshold", DoubleValue(-113.0));
1353
```

```
1354        wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
1355                        "DataMode", DataRate,
1356                        "ControlMode", DataRate);
1357
1358        NetDeviceContainer apDevice;
1359        apDevice = wifi.Install(wifiPhy, wifiMac, wifiApNode);
1360
1361        wifiApNode.Get(0)->GetDevice (0)->SetAddress (macAddressAp);
1362
1363
1364        std::ostringstream oss;
1365        oss <<
            ↪  "/NodeList/*/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::ApWifiMac/"
1366        Config::ConnectWithoutContext(oss.str() + "RpsIndex",
            ↪  MakeCallback(&RpsIndexTrace));
1367        Config::ConnectWithoutContext(oss.str() + "RawGroup",
            ↪  MakeCallback(&RawGroupTrace));
1368        Config::ConnectWithoutContext(oss.str() + "RawSlot", MakeCallback(&RawSlotTrace));
1369
1370        // disable fragmentation for frames below 22000 bytes
1371        Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold",
1372                    StringValue("999999"));
1373        // turn off RTS/CTS for frames below 22000 bytes
1374        Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",
1375                    StringValue("999999"));
1376        // Fix non-unicast data rate to be the same as that of unicast
1377        Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode", DataRate);
1378
1379        // mobility.
1380        MobilityHelper mobility;
1381        Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>();
1382
1383        positionAlloc->Add(Vector(0.0, 0.0, 0.0));
1384        positionAlloc->Add(Vector(dist, 0.0, 0.0));
1385        mobility.SetPositionAllocator(positionAlloc);
1386
1387        mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
1388
1389        mobility.Install(wifiApNode);
1390        mobility.Install(c);
1391
1392        for (uint16_t kk = 0; kk < Nsta; kk++)
1393        {
1394          std::ostringstream STA;
1395          STA << kk;
1396          std::string strSTA = STA.str();
1397
1398          assoc_record *m_assocrecord = new assoc_record;
```

```
1399          m_assocrecord->setstaid(kk);
1400          Config::Connect(
1401            "/NodeList/" + strSTA +
              ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/Assoc",
1402            MakeCallback(&assoc_record::SetAssoc, m_assocrecord));
1403          Config::Connect(
1404            "/NodeList/" + strSTA +
              ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/DeAssoc
1405            MakeCallback(&assoc_record::UnsetAssoc, m_assocrecord));
1406          assoc_vector.push_back(m_assocrecord);
1407          delete m_assocrecord;
1408        }
1409
1410        InternetStackHelper internet;
1411        internet.Install(c);
1412        internet.Install(wifiApNode);
1413
1414        Ipv4AddressHelper ipv4;
1415        NS_LOG_INFO("Assign IP Addresses.");
1416        ipv4.SetBase("10.1.1.0", "255.255.255.0");
1417        Ipv4InterfaceContainer staNodeInterface = ipv4.Assign(devices);
1418        Ipv4InterfaceContainer apNodeInterface = ipv4.Assign(apDevice);
1419
1420        Ipv4GlobalRoutingHelper::PopulateRoutingTables();
1421        //std::cout << "Populating ARP cache..." << std::endl;
1422        PopulateArpCache();
1423
1424        /** Energy Model **/
1425        /*************************************************************************/
1426        ///* energy source */
1427        LiIonEnergySourceHelper liIonEnergySourceHelper;
1428
1429        liIonEnergySourceHelper.Set("LiIonEnergySourceInitialEnergyJ",
        ↪  DoubleValue(sourceInitialEnergyJ));
1430        liIonEnergySourceHelper.Set("LiIonEnergyLowBatteryThreshold",
        ↪  DoubleValue(batteryLowThreshold));
1431        liIonEnergySourceHelper.Set("LiIonEnergyHighBatteryThreshold",
        ↪  DoubleValue(batteryHighThreshold));
1432        liIonEnergySourceHelper.Set("InitialCellVoltage", DoubleValue(1.05));
1433        liIonEnergySourceHelper.Set("NominalCellVoltage", DoubleValue(1.06));
1434        liIonEnergySourceHelper.Set("ExpCellVoltage", DoubleValue(3.06));
1435        liIonEnergySourceHelper.Set("RatedCapacity", DoubleValue(2.45));
1436        liIonEnergySourceHelper.Set("NomCapacity", DoubleValue(1.1));
1437        liIonEnergySourceHelper.Set("ExpCapacity", DoubleValue(1.2));
1438        liIonEnergySourceHelper.Set("InternalResistance", DoubleValue(0.083));
1439        liIonEnergySourceHelper.Set("TypCurrent", DoubleValue(2.33));
1440        liIonEnergySourceHelper.Set("ThresholdVoltage", DoubleValue(0.3));
1441        liIonEnergySourceHelper.Set("PeriodicEnergyUpdateInterval",
        ↪  TimeValue(Seconds(simulationTime + coolDown + 2.0)));
```

```
1442          // install source
1443          EnergySourceContainer sources = liIonEnergySourceHelper.Install(c);
1444
1445          //Power consumption:
              ↪  https://www.asiarf.com/shop/halow-lora-iot/wi-fi-halow-sub-ghz-wireless-module-morse-mic
1446          /* device energy model */
1447          WifiRadioEnergyModelHelper radioEnergyHelper;
1448          // configure radio energy model
1449          if ((txPower == 0.0)&&(datarate < 3900000))
1450          {
1451            radioEnergyHelper.Set("TxCurrentA", DoubleValue(8.5e-3));
1452          }
1453          else
1454          {
1455            radioEnergyHelper.Set("TxCurrentA", DoubleValue(61.82e-3));
1456          }
1457
1458          if (datarate < 3900000)
1459          {
1460            //radioEnergyHelper.Set("RxCurrentA", DoubleValue(15.0e-3));
1461            radioEnergyHelper.Set("RxCurrentA", DoubleValue(6.0e-3));
1462          }
1463          else
1464          {
1465            radioEnergyHelper.Set("RxCurrentA", DoubleValue(27.0e-3));
1466          }
1467          radioEnergyHelper.Set("SleepCurrentA", DoubleValue(1.2e-6));
1468          radioEnergyHelper.Set("IdleCurrentA", DoubleValue(6.0e-3));
1469          radioEnergyHelper.Set("CcaBusyCurrentA", DoubleValue(6.0e-3));
1470          radioEnergyHelper.Set("SwitchingCurrentA", DoubleValue(8.5e-3));
1471
1472          // install device model
1473          DeviceEnergyModelContainer deviceModels = radioEnergyHelper.Install(devices,
              ↪  sources);
1474
1475          /* energy harvester */
1476          SwiptHarvesterHelper swiptHelper;
1477          // configure energy harvester
1478          swiptHelper.Set("AntennaNoise", DoubleValue(-111.0));
1479          swiptHelper.Set("PowerSplitFactor", DoubleValue(psFactor));
1480          swiptHelper.Set("SwiptEfficiency", DoubleValue(0.9));
1481          swiptHelper.Set("DCConversionEfficiency", DoubleValue(0.95));
1482          // install harvester on all energy sources
1483          EnergyHarvesterContainer harvesters = swiptHelper.Install(sources);
1484
1485          for (uint32_t i = 0; i < Nsta; i++)
1486          {
1487            Ptr<WifiPhy> phyp = c.Get(i)->GetDevice(0)->GetObject<WifiNetDevice>()->GetPhy();
```

```
1488          Ptr<SwiptHarvester> swiptharvester =
              ↪  harvesters.Get(i)->GetObject<SwiptHarvester>();
1489
1490          swiptharvester->SetUpSwiptPhyListener(phyp);
1491          // std::cout << "\n\n=======\n phypp:" << phyp << std::endl;
1492
1493          Ptr<YansWifiPhy> yansPhy =
              ↪  c.Get(i)->GetDevice(0)->GetObject<WifiNetDevice>()->GetPhy()->GetObject<YansWifiPhy>()
1494
1495          Ptr<YansWifiChannel> m_channel =
              ↪  c.Get(i)->GetDevice(0)->GetObject<WifiNetDevice>()->GetPhy()->GetChannel()->GetObject<
1496          m_channel->AddSwiptPointer(swiptharvester);
1497          // std::cout << "\n\n=======\n wifiChannelPtr:" << m_channel << "\t" <<
              ↪  swiptharvester << std::endl;
1498        }
1499
1500        Config::Set(
1501            "/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_EdcaTxopN/Queu
1502            UintegerValue( 60 / interPktInterval ));
1503            //UintegerValue(10));
1504
1505        Config::Set(
1506            "/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_EdcaTxopN/Queu
1507            //TimeValue(Seconds(600)));
1508            TimeValue(NanoSeconds(600000000000)));
1509
1510
1511        Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/Txop/MinC
              ↪  UintegerValue(15));
1512        Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/Txop/MaxC
              ↪  UintegerValue(1023));
1513        Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/Txop/Aifs
              ↪  UintegerValue(4));
1514
1515        Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/Rifs", TimeValue
              ↪  (MicroSeconds (2)));
1516
1517
1518        Config::Connect("/NodeList/0/DeviceList/*/Phy/$ns3::YansWifiPhy/State/State",
              ↪  MakeCallback(&PhyStateTraced));
1519
1520        updateNodesQueueLength(devices, apDevice);
1521
1522        sources.Get(0)->TraceConnectWithoutContext("RemainingEnergy",
              ↪  MakeCallback(&RemainingEnergy));
1523
1524        deviceModels.Get(0)->TraceConnectWithoutContext("TotalEnergyConsumption",
              ↪  MakeCallback(&TotalEnergy));
```

```
1525
1526        harvesters.Get(0)->TraceConnectWithoutContext("HarvestedPower",
            ↪  MakeCallback(&HarvestedPower));
1527
1528        harvesters.Get(0)->TraceConnectWithoutContext("TotalEnergyHarvested",
            ↪  MakeCallback(&TotalEnergyHarvested));
1529
1530        Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorSniff
            ↪  MakeCallback(&MonitorRx<0>));
1531        Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorSniff
            ↪  MakeCallback(&MonitorRx<1>));
1532        Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorSniff
            ↪  MakeCallback(&MonitorTx<0>));
1533        Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorSniff
            ↪  MakeCallback(&MonitorTx<1>));
1534
1535        Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
            ↪  MakeCallback(&PhyRxEnd<0>));
1536        Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
            ↪  MakeCallback(&PhyRxEnd<1>));
1537        Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
            ↪  MakeCallback(&PhyTxEnd<0>));
1538        Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
            ↪  MakeCallback(&PhyTxEnd<1>));
1539
1540        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxDropWithReason",
            ↪  MakeCallback(&OnPhyTxDrop<0>));
1541        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxDropWithReason",
            ↪  MakeCallback(&OnPhyTxDrop<1>));
1542
1543        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxDropWithReason",
            ↪  MakeCallback(&OnPhyRxDrop<0>));
1544        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxDropWithReason",
            ↪  MakeCallback(&OnPhyRxDrop<1>));
1545
1546        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/RemoteStationManager/MacTxData
            ↪  MakeCallback(&OnMacPacketDropped<0>));
1547        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/RemoteStationManager/MacTxData
            ↪  MakeCallback(&OnMacPacketDropped<1>));
1548
1549        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪  MakeCallback(&OnMacPacketDropped2<0>));
1550        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪  MakeCallback(&OnMacPacketDropped2<1>));
1551
1552        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪  MakeCallback(&OnCollision<0>));
1553        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪  MakeCallback(&OnCollision<1>));
```

```
1554
1555        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪ MakeCallback(&OnTransmissionWillCrossRAWBoundary<0>));
1556        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪ MakeCallback(&OnTransmissionWillCrossRAWBoundary<1>));
1557
1558        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪ MakeCallback(&TxRawSlot<0>)); // not implem
1559        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪ MakeCallback(&TxRawSlot<1>)); // not implem
1560        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3:
            ↪ MakeCallback(&TxRawSlot<1>));  // not implem
1561
1562        Config::Connect("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::ApWifiMac/PacketToTr
            ↪ MakeCallback(&OnAPPacketToTransmitReceived));
1563        Config::Connect("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::ApWifiMac/PacketToTr
            ↪ MakeCallback(&OnAPPacketToTransmitReceived));
1564
1565        Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::Regula
            ↪ MakeCallback(&OnQueueDrop<0>));
1566        Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::Regula
            ↪ MakeCallback(&OnQueueDrop<1>));
1567
1568        uint16_t port = 9;
1569        ApplicationContainer serverApp;
1570        ApplicationContainer clientApp;
1571
1572        if (trafficType == "udp")
1573        {
1574            UdpServerHelper server(port);
1575            //server.SetAttribute ("SetPacketWindowSize",UintegerValue (128));
1576            serverApp = server.Install(c.Get(0));
1577            serverApp.Start(Seconds(0.0));
1578            //serverApp.Stop(Seconds(simulationTime + 2.0));
1579            serverApp.Stop (Seconds (simulationTime + coolDown));
1580            serverApp.Get(0)->TraceConnectWithoutContext("Rx",
1581                      MakeCallback(&udpPacketReceivedAtServer));
1582
1583            UdpClientHelper client(staNodeInterface.GetAddress(0), port);
1584            client.SetAttribute("MaxPackets", UintegerValue(4294967295u));
1585            // client.SetAttribute ("Interval", TimeValue (Time ("0.000995328")));
                ↪ //packets/s
1586            //client.SetAttribute("Interval",
                ↪ TimeValue(Time(std::to_string(interPktInterval)))); // packets/s
1587            client.SetAttribute("Interval", TimeValue(Seconds(interPktInterval))); //
                ↪ packets/s
1588            client.SetAttribute("PacketSize", UintegerValue(payloadSize));
1589
```

```
1590            clientApp = client.Install(wifiApNode.Get(0));
1591            clientApp.Start(Seconds(1.0));
1592            clientApp.Stop(Seconds(simulationTime + 1.0));
1593            clientApp.Get(0)->TraceConnectWithoutContext("Tx",
                ↪  MakeCallback(&OnUdpPacketSent));
1594
1595          }
1596          else if (trafficType == "udpsocket")
1597          {
1598            TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
1599            Ptr<Socket> recvSink = Socket::CreateSocket (c.Get (0), tid);
1600            InetSocketAddress local = InetSocketAddress (Ipv4Address ("10.1.1.1"), 80);
1601            recvSink->Bind (local);
1602            recvSink->SetRecvCallback (MakeCallback (&ReceivePacket));
1603
1604            Ptr<Socket> source = Socket::CreateSocket (wifiApNode.Get (0), tid);
1605            InetSocketAddress remote = InetSocketAddress (Ipv4Address ("10.1.1.1"), 80);
1606            //source->SetAllowBroadcast (true);
1607            source->Connect (remote);
1608            Simulator::ScheduleWithContext (source->GetNode ()->GetId (),
1609                                    Seconds (1.0), &GenerateTraffic,
1610                                      source, payloadSize , numPackets,
                                        ↪  Seconds(interPktInterval));
1611          }
1612          else if (trafficType == "udpecho")
1613          {
1614            UdpEchoServerHelper myServer(port);
1615            serverApp = myServer.Install(c.Get(0));
1616            serverApp.Get(0)->TraceConnectWithoutContext("Rx",
                ↪  MakeCallback(&udpPacketReceivedAtServer));
1617            serverApp.Start(Seconds(0));
1618
1619            UdpEchoClientHelper clientHelper(staNodeInterface.GetAddress(0), 9); // address
                ↪  of remote node
1620            clientHelper.SetAttribute("MaxPackets", UintegerValue(4294967295u));
1621            // clientHelper.SetAttribute("Interval",
                ↪  TimeValue(MilliSeconds(interPktInterval)));
1622            clientHelper.SetAttribute("Interval",
                ↪  TimeValue(Time(std::to_string(interPktInterval)))); // packets/s
1623
1624            // clientHelper.SetAttribute("IntervalDeviation",
                ↪  TimeValue(MilliSeconds(config.trafficIntervalDeviation)));
1625            clientHelper.SetAttribute("PacketSize", UintegerValue(payloadSize));
1626
1627            clientApp = clientHelper.Install(wifiApNode.Get(0));
1628            clientApp.Get(0)->TraceConnectWithoutContext("Tx",
                ↪  MakeCallback(&OnUdpPacketSent));
1629            clientApp.Get(0)->TraceConnectWithoutContext("Rx",
                ↪  MakeCallback(&OnUdpEchoPacketReceived));
```

```
1630          clientApp.Start(Seconds(1.0));
1631          // clientApp.Stop(Seconds(simulationTime + 1));
1632        }
1633        else if (trafficType == "tcp")
1634        {
1635          uint16_t port = 50000;
1636          Address localAddress(InetSocketAddress(Ipv4Address::GetAny(), port));
1637          PacketSinkHelper packetSinkHelper("ns3::TcpSocketFactory", localAddress);
1638          serverApp = packetSinkHelper.Install(c.Get(0));
1639          serverApp.Start(Seconds(0.0));
1640          // serverApp.Stop (Seconds (simulationTime + coolDown));
1641
1642          OnOffHelper onoff("ns3::TcpSocketFactory", Ipv4Address::GetAny());
1643          onoff.SetAttribute("OnTime",
              ↪  StringValue("ns3::ConstantRandomVariable[Constant=1]"));
1644          onoff.SetAttribute("OffTime",
              ↪  StringValue("ns3::ConstantRandomVariable[Constant=0]"));
1645          onoff.SetAttribute("PacketSize", UintegerValue(payloadSize));
1646          onoff.SetAttribute("DataRate", DataRateValue(std::to_string(datarate))); //
              ↪  bit/s
1647          // onoff.SetAttribute ("DataRate", DataRateValue (1000000000)); //bit/s
1648          AddressValue remoteAddress(InetSocketAddress(staNodeInterface.GetAddress(0),
              ↪  port));
1649          onoff.SetAttribute("Remote", remoteAddress);
1650
1651          clientApp = onoff.Install(wifiApNode.Get(0));
1652          clientApp.Start(Seconds(1.0));
1653          // clientApp.Stop (Seconds (simulationTime + 1.0));
1654        }
1655
1656
1657              if (tracing)
1658              {
1659            wifiPhy.EnablePcap("./ResultColetorSwipt/wifi-80211ah-infra_" +
              ↪  std::to_string(int(dist)) + "_" + std::to_string(int(psFactor*10)),
              ↪  devices.Get(0), true); // Output what we are doing
1660            wifiPhy.EnablePcap("./ResultColetorSwipt/wifi-80211ah-infra_" +
              ↪  std::to_string(int(dist)) + "_" + std::to_string(int(psFactor*10)),
              ↪  apDevice, true); // Output what we are doing
1661              }
1662
1663        Simulator::Stop(Seconds(simulationTime + coolDown));
1664        Simulator::Run();
1665
1666        double txOffered = 0.0;
1667        double thruPut = 0.0;
1668        double thruPut2 = 0.0;
1669
```

```
1670        uint64_t totalPacketsThrough = 0;
1671        uint64_t totalPacketsEchoed = 0;
1672        uint64_t packtLost = 0;
1673        uint64_t packtRcv = 0;
1674        double throughput = 0.0;
1675        if (trafficType == "udp")
1676        {
1677          totalPacketsThrough = DynamicCast<UdpServer>(serverApp.Get(0))->GetReceived();
1678          throughput = totalPacketsThrough * payloadSize * 8.0 / (simulationTime * 1.0); //
              ↪ bps
1679        }
1680        else if (trafficType == "udpsocket")
1681        {
1682          totalPacketsThrough = onUdpRx;
1683          throughput = totalPacketsThrough * payloadSize * 8.0 / (simulationTime * 1.0); //
              ↪ bps
1684        }
1685        else if (trafficType == "udpecho")
1686        {
1687          totalPacketsThrough = onUdpRx;
1688          totalPacketsEchoed = onUdpEchoRx;
1689          packtLost = onUdpTx - onUdpRx;
1690          throughput = ((totalPacketsThrough + totalPacketsEchoed) * payloadSize * 8.0) /
              ↪ (simulationTime * 1.0); // bps
1691        }
1692        else if (trafficType == "tcp")
1693        {
1694          uint64_t totalBytesRx = DynamicCast<PacketSink>(serverApp.Get(0))->GetTotalRx();
1695          totalPacketsThrough = totalBytesRx / tcpPacketSize;
1696          // throughput = totalBytesRx * 8.0 / ((ApStopTime - AppStartTime) * 1.0);
              ↪ //Mbit/s
1697          throughput = totalBytesRx * 8.0 / ((simulationTime)*1.0); // Mbit/s
1698        }
1699
1700        std::cout << std::setprecision(2) << std::fixed
1701              << std::setw(2) << dist
1702             << std::setw(6) << std::setprecision(0) <<  datarate/1000
1703             // << std::setw(6) << std::setprecision(0) <<  txOffered
1704              << std::setprecision(2) << std::fixed
1705             // << std::setw(10) << thruPut
1706              << std::setw(10) << throughput / 1000
1707              << std::setw(8) << onUdpTx
1708              << std::setw(8) << totalPacketsThrough
1709              << std::setw(8) << onUdpTx - totalPacketsThrough
1710              // << std::setw(8) << packtLost
1711              << std::setw(8) << onPktsTx[1]
1712              << std::setw(8) << onPktsRx[0]
1713              << std::setw(8) << onPktsTx[1] - onPktsRx[0]
```

```
                        << std::setprecision(10) << std::fixed
              << std::setw(15) << psFactor;

        std::cout << std::setprecision(2) << std::scientific
              << std::setw(10) << g_harvestedPowerAvg / harvestedPowerCounter
              << std::setw(10) << g_harvestedEnergyAvg / harvestedEnergyCounter;

        std::cout << std::setprecision(2) << std::fixed
              << std::setw(9) << g_signalDbmAvg[0]
              << std::setw(9) << g_noiseDbmAvg[0]
              << std::setw(9) << (g_signalDbmAvg[0] - g_noiseDbmAvg[0])
              << std::setw(7) << csr
              << std::endl;


        std::stringstream ss;
        ss << "./ResultColetorSwipt/Summary.csv";
        static std::fstream f(ss.str().c_str(), std::ios::out);
        f << dist<< ", "
          << datarate/1000 << ", "
          << throughput/1000 << ", "
          << onUdpTx << ", "
          << totalPacketsThrough << ", "
          << onUdpTx - totalPacketsThrough << ", "
          << onPktsTx[1] << ", "
          << onPktsRx[0] << ", "
          << onPktsTx[1] - onPktsRx[0] << ", "
          << std::setprecision(10) << std::fixed
          << psFactor << ", "
                << std::setprecision(10) << std::scientific
          << g_harvestedPowerAvg / harvestedPowerCounter << ", "
          << harvestedPowerCounter << ", "
          << g_harvestedPowerAvg << ", "
          << g_harvestedEnergyAvg / harvestedEnergyCounter << ", "
          << harvestedEnergyCounter << ", "
          << g_harvestedEnergyAvg << ", "
          << consump << ", "
          << std::fixed << std::setprecision(2)
          << g_signalDbmAvg[0] << ", "
          << g_noiseDbmAvg[0] << ", "
          << (g_signalDbmAvg[0] - g_noiseDbmAvg[0]) << ", "
          << std::fixed << std::setprecision(6)
          << (apTxPower - m_rxSignalDbm[0] + std::log10(psFactor)*10.0 + rxAntennaGain) <<
          ↪  ", "
          << std::fixed << std::setprecision(2)
          << csr  << ", "
                << std::setprecision(10) << std::scientific
                << ((std::pow (10.0, g_signalDbmAvg[0]/10.0)*1.0e-3)/psFactor)
```

```
1761          << std::endl;
1762
1763        for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); i++)
1764        {
1765
1766          Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol>();
1767          ip->Cleanup();
1768          // ip->Dispose();
1769          Ptr<WifiMac> mac = (*i)->GetObject<WifiMac>();
1770          mac->Cleanup();
1771          // mac->Dispose();
1772          Ptr<WifiNetDevice> net = (*i)->GetObject<WifiNetDevice>();
1773          net->Cleanup();
1774          // net->Dispose();
1775          Ptr<Node> node = (*i)->GetObject<Node>();
1776          node->Cleanup();
1777          // node->Dispose();
1778          Ptr<YansWifiPhy> yansPhy = (*i)->GetObject<YansWifiPhy>();
1779          yansPhy->Cleanup();
1780          // yansPhy->Dispose();
1781          Ptr<SwiptHarvester> sw = (*i)->GetObject<SwiptHarvester> ();
1782          sw->Cleanup ();
1783
1784          //Ptr<EdcaTxopN> edcaT = (*i)->GetObject<EdcaTxopN> ();
1785          //edcaT->Cleanup ();
1786
1787          Ptr<NetDevice> netDe = (*i)->GetObject<NetDevice> ();
1788          netDe->Cleanup ();
1789
1790          Ptr<YansWifiChannel> ch = (*i)->GetObject<YansWifiChannel> ();
1791          ch->Cleanup ();
1792        }
1793
1794        assoc_vector.erase (assoc_vector.begin(), assoc_vector.end());
1795        rps.rpsset.erase (rps.rpsset.begin(), rps.rpsset.end());
1796        m_queue.erase (m_queue.begin(), m_queue.end());
1797        m_queue2.erase (m_queue2.begin(), m_queue2.end());
1798        BEqueue.erase (BEqueue.begin(), BEqueue.end());
1799        serverApp.Get (0)->Cleanup ();
1800        clientApp.Get (0)->Cleanup ();
1801        c.Get (0)->Cleanup ();
1802        wifiApNode.Get (0)->Cleanup ();
1803        devices.Get (0)->Cleanup ();
1804        apDevice.Get (0)->Cleanup ();
1805
1806        //rps.setlen(0);
1807        //GetAssocNum()
1808        //GetAssocNum(assoc_vector);
```

```
1809
1810        g_harvestedPowerAvg = 0.0;
1811        g_harvestedEnergyAvg = 0.0;
1812
1813        Simulator::Destroy();
1814
1815                if (psFactor == 0.0)
1816                {
1817                    psFactor = 0.0000000001;
1818                }
1819                else if (psFactor > 0.0 && psFactor < 0.1)
1820                {
1821                    psFactor = psFactor * 10.0;
1822                }
1823                else
1824                {
1825                    psFactor += 0.1;
1826                }
1827      }
1828    }
1829  }
1830  return 0;
1831 }
```

APPENDIX C

NETWORK SCENARIO

```
/* -*-  Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2009 MIRKO BANCHI
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

 */

#include "s1g-test-tim-raw.h"
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/energy-module.h"
#include "ns3/udp-client-server-helper.h"
#include "ns3/udp-echo-helper.h"
#include "ns3/udp-echo-server.h"
#include "ns3/udp-echo-client.h"
#include "ns3/udp-server.h"
#include "ns3/udp-client.h"
#include "ns3/packet.h"
#include "ns3/on-off-helper.h"
#include "ns3/packet-sink.h"
#include "ns3/packet-sink-helper.h"
#include "ns3/command-line.h"
```

```
39 #include "ns3/rps.h"
40 #include "ns3/log.h"
41 #include "ns3/node.h"
42 #include "ns3/simulator.h"
43 #include "ns3/double.h"
44 #include "ns3/config.h"
45 #include "ns3/string.h"
46 #include "ns3/extension-headers.h"
47 #include "ns3/stats-module.h"
48 #include "ns3/application.h"
49 #include "ns3/flow-monitor.h"
50 #include "ns3/flow-monitor-helper.h"
51 #include "ns3/ipv4-flow-classifier.h"
52 #include "ns3/wifi-tx-vector.h"
53 #include "ns3/wifi-mode.h"
54 #include "ns3/mac-low.h"
55 #include "ns3/dca-txop.h"
56 #include "ns3/edca-txop-n.h"
57 #include "ns3/propagation-loss-model.h"
58 #include "ns3/buildings-propagation-loss-model.h"
59
60
61 #include "ns3/core-module.h"
62 #include "ns3/network-module.h"
63 #include "ns3/applications-module.h"
64 #include "ns3/wifi-module.h"
65 #include "ns3/mobility-module.h"
66 #include "ns3/ipv4-global-routing-helper.h"
67 #include "ns3/internet-module.h"
68 #include "ns3/extension-headers.h"
69
70 #include "ns3/ipv4-address.h"
71 #include "ns3/inet-socket-address.h"
72 #include "ns3/core-module.h"
73 #include "ns3/radiotap-header.h"
74 #include "ns3/flow-monitor-module.h"
75
76
77 #include <regex>
78 #include <iostream>
79 #include <fstream>
80 #include <vector>
81 #include <string>
82 #include <iomanip>
83 #include <utility>
84 #include <stdio.h>
85 #include <stdlib.h>
86 #include <ctime>
```

```
87  #include <sys/stat.h>
88  #include <map>
89
90  NS_LOG_COMPONENT_DEFINE("s1g-wifi-network-tim-raw");
91
92  uint32_t AssocNum = 0;
93  int64_t AssocTime = 0;
94  uint32_t StaNum = 0;
95  NetDeviceContainer staDeviceCont;
96  const int MaxSta = 8000;
97
98  uint32_t onFramesTx[31];
99  uint32_t onFramesRxAtNode[31];
100 uint32_t onFramesRxAtApFromNode[31];
101 uint32_t onFramesRxMonitor[31];
102 double g_signalDbmAvg[31];
103 double g_noiseDbmAvg[31];
104 double m_rxSignalDbm[31];
105 uint32_t NumberOfRetriesRx[31];
106 uint32_t NumberOfPowerBeaconsSent;
107
108
109 //Ptr<EdcaTxopN> m_dcf;
110
111 Configuration config;
112 Statistics stats;
113 SimulationEventManager eventManager;
114
115 Ptr<Socket> powerBeacon;
116 Ptr<Socket> recvSink;
117 TypeId tidPowerBeacon = TypeId::LookupByName("ns3::UdpSocketFactory");
118
119
120 TypeId tidInterference = TypeId::LookupByName("ns3::UdpSocketFactory");
121
122 std::string dropReason[] =
123   {
124     "Unknown",
125     "PhyInSleepMode",
126     "PhyNotEnoughSignalPower",
127     "PhyUnsupportedMode",
128     "PhyPreampleHeaderReceptionFailed",
129     "PhyRxDuringChannelSwitching",
130     "PhyAlreadyReceiving",
131     "PhyAlreadyTransmitting",
132     "PhyPlcpReceptionFailed",
133     "MacNotForAP",
134     "MacAPToAPFrame",
```

```
135     "MacQueueDelayExceeded",
136     "MacQueueSizeExceeded",
137     "TCPTxBufferExceeded",
138     "PhyRxNotEnoughSNIR",
139     "DeviceIsOff"};
140
141 Mac48Address macAdd[] =  {
142     "00:00:00:00:00:01", // 0
143     "00:00:00:00:00:02", // 1
144     "00:00:00:00:00:03", //2
145     "00:00:00:00:00:04", //3
146     "00:00:00:00:00:05", //4
147     "00:00:00:00:00:06", //5
148     "00:00:00:00:00:07", //6
149     "00:00:00:00:00:08", //7
150     "00:00:00:00:00:09", //8
151     "00:00:00:00:00:0a", //9
152     "00:00:00:00:00:0b", //10
153     "00:00:00:00:00:0c", //11
154     "00:00:00:00:00:0d", //12
155     "00:00:00:00:00:0e", //13
156     "00:00:00:00:00:0f", //14
157     "00:00:00:00:00:10", //15
158     "00:00:00:00:00:11", //16
159     "00:00:00:00:00:12", //17
160     "00:00:00:00:00:13", //18
161     "00:00:00:00:00:14", //19
162     "00:00:00:00:00:15", //20
163     "00:00:00:00:00:16", //21
164     "00:00:00:00:00:17", //22
165     "00:00:00:00:00:18", //23
166     "00:00:00:00:00:19", //24
167     "00:00:00:00:00:1a", //25
168     "00:00:00:00:00:1b", //26
169     "00:00:00:00:00:1c", //27
170     "00:00:00:00:00:1d", //28
171     "00:00:00:00:00:1e", //29
172     "00:00:00:00:00:1f", //30
173     "00:00:00:00:00:20",
174     "00:00:00:00:00:21",
175     "00:00:00:00:00:22",
176     "00:00:00:00:00:23",
177     "00:00:00:00:00:24",
178     "00:00:00:00:00:25",
179     "00:00:00:00:00:26",
180     "00:00:00:00:00:27",
181     "00:00:00:00:00:28",
182     "00:00:00:00:00:29",
```

```
    "00:00:00:00:00:2a",
    "00:00:00:00:00:2b",
    "00:00:00:00:00:2c",
    "00:00:00:00:00:2d",
    "00:00:00:00:00:2e",
    "00:00:00:00:00:2f",
    "00:00:00:00:00:30",
    "00:00:00:00:00:31",
    "00:00:00:00:00:32",
    "00:00:00:00:00:33",
    "00:00:00:00:00:34",
    "00:00:00:00:00:35",
    "00:00:00:00:00:36",
    "00:00:00:00:00:37",
    "00:00:00:00:00:38",
    "00:00:00:00:00:39",
    "00:00:00:00:00:3a",
    "00:00:00:00:00:3b",
    "00:00:00:00:00:3c",
    "00:00:00:00:00:3d",
    "00:00:00:00:00:3e",
    "00:00:00:00:00:3f",
    "00:00:00:00:00:40"};

class assoc_record
{
public:
  assoc_record();
  bool GetAssoc();
  void SetAssoc(std::string context, Mac48Address address);
  void UnsetAssoc(std::string context, Mac48Address address);
  void setstaid(uint16_t id);

private:
  bool assoc;
  uint16_t staid;
};

assoc_record::assoc_record()
{
  assoc = false;
  staid = 65535;
}

void assoc_record::setstaid(uint16_t id)
{
  staid = id;
}
```

```
231
232 void assoc_record::SetAssoc(std::string context, Mac48Address address)
233 {
234   assoc = true;
235 }
236
237 void assoc_record::UnsetAssoc(std::string context, Mac48Address address)
238 {
239   assoc = false;
240 }
241
242 bool assoc_record::GetAssoc()
243 {
244   return assoc;
245 }
246
247 typedef std::vector<assoc_record *> assoc_recordVector;
248 assoc_recordVector assoc_vector;
249
250 uint32_t GetAssocNum()
251 {
252   AssocNum = 0;
253   for (assoc_recordVector::const_iterator index = assoc_vector.begin();
254      index != assoc_vector.end(); index++)
255   {
256     if ((*index)->GetAssoc())
257     {
258       AssocNum++;
259     }
260   }
261   return AssocNum;
262 }
263
264 void PopulateArpCache()
265 {
266   Ptr<ArpCache> arp = CreateObject<ArpCache>();
267   arp->SetAliveTimeout(Seconds(3600 * 24 * 365));
268   for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
269   {
270     Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol>();
271     NS_ASSERT(ip != 0);
272     ObjectVectorValue interfaces;
273     ip->GetAttribute("InterfaceList", interfaces);
274     for (ObjectVectorValue::Iterator j = interfaces.Begin();
275        j != interfaces.End(); j++)
276     {
277       Ptr<Ipv4Interface> ipIface =
278         (j->second)->GetObject<Ipv4Interface>();
```

```
279       NS_ASSERT(ipIface != 0);
280       Ptr<NetDevice> device = ipIface->GetDevice();
281       NS_ASSERT(device != 0);
282       Mac48Address addr = Mac48Address::ConvertFrom(device->GetAddress());
283       for (uint32_t k = 0; k < ipIface->GetNAddresses(); k++)
284       {
285         Ipv4Address ipAddr = ipIface->GetAddress(k).GetLocal();
286         if (ipAddr == Ipv4Address::GetLoopback())
287           continue;
288         ArpCache::Entry *entry = arp->Add(ipAddr);
289         entry->MarkWaitReply(0);
290         entry->MarkAlive(addr);
291         std::cout << "Arp Cache: Adding the pair (" << addr << ","
292               << ipAddr << ")" << std::endl;
293       }
294     }
295   }
296   for (NodeList::Iterator i = NodeList::Begin(); i != NodeList::End(); ++i)
297   {
298     Ptr<Ipv4L3Protocol> ip = (*i)->GetObject<Ipv4L3Protocol>();
299     NS_ASSERT(ip != 0);
300     ObjectVectorValue interfaces;
301     ip->GetAttribute("InterfaceList", interfaces);
302     for (ObjectVectorValue::Iterator j = interfaces.Begin();
303        j != interfaces.End(); j++)
304     {
305       Ptr<Ipv4Interface> ipIface =
306         (j->second)->GetObject<Ipv4Interface>();
307       ipIface->SetAttribute("ArpCache", PointerValue(arp));
308     }
309   }
310 }
311
312 uint16_t ngroup;
313 uint16_t nslot;
314
315 RPSVector configureRAW(RPSVector rpslist, string RAWConfigFile)
316 {
317   uint16_t NRPS = 0;
318   uint16_t NRAWPERBEACON = 0;
319   uint16_t Value = 0;
320   uint32_t page = 0;
321   uint32_t aid_start = 0;
322   uint32_t aid_end = 0;
323   uint32_t rawinfo = 0;
324
325   ifstream myfile(RAWConfigFile);
326   //1. get info from config file
```

```
327
328   //2. define RPS
329   if (myfile.is_open()) {
330     myfile >> NRPS;
331     int totalNumSta = 0;
332     for (int kk = 0; kk < NRPS; kk++) // number of beacons covering all raw groups
333     {
334       RPS *m_rps = new RPS;
335       myfile >> NRAWPERBEACON;
336       ngroup = NRAWPERBEACON;
337       for (int i = 0; i < NRAWPERBEACON; i++) // raw groups in one beacon
338       {
339         //RPS *m_rps = new RPS;
340         RPS::RawAssignment *m_raw = new RPS::RawAssignment;
341
342         myfile >> Value;
343         m_raw->SetRawControl(Value);  //support paged STA or not
344         myfile >> Value;
345         m_raw->SetSlotCrossBoundary(Value);
346         myfile >> Value;
347         m_raw->SetSlotFormat(Value);
348         myfile >> Value;
349         m_raw->SetSlotDurationCount(Value);
350         myfile >> Value;
351         nslot = Value;
352         m_raw->SetSlotNum(Value);
353         myfile >> page;
354         myfile >> aid_start;
355         myfile >> aid_end;
356         rawinfo = (aid_end << 13) | (aid_start << 2) | page;
357         m_raw->SetRawGroup(rawinfo);
358         totalNumSta += aid_end - aid_start + 1;
359         m_rps->SetRawAssignment(*m_raw);
360         delete m_raw;
361       }
362       rpslist.rpsset.push_back(m_rps);
363       //config.nRawGroupsPerRpsList.push_back(NRAWPERBEACON);
364     }
365     myfile.close();
366     config.NRawSta = totalNumSta;
367     //rpslist.rpsset[rpslist.rpsset.size() - 1]->GetRawAssigmentObj(
368     //        NRAWPERBEACON - 1).GetRawGroupAIDEnd();
369   } else
370     cout << "Unable to open RAW configuration file \n";
371
372   return rpslist;
373 }
374 /*
```

```
375 pageslice element and TIM(DTIM) together accomplish page slicing.
376
377 Prior knowledge:
378 802.11ah support up to 8192 stations, they are constructed into: page, block,
379  subblock, sta.
380 there are 13 bit represent the AID of stations.
381  AID[11-12] represent page.
382  AID[6-10] represent block.
383  AID[3-5] represent subblock.
384  AID[0-2] represent sta.
385
386 A TIM(DTIM) element only support one page
387 A Page slice element only support one page
388
389  Concept of page slicing:
390  Between two DTIM beacon, there are many TIM beacons, only allow a TIM beacon include some
      ↪  blocks of one page is called page slice. One TIM beacon is called a page slice.
391  Page slcie element specify number of page slice between two DTIM, number of blocks in
      ↪  each
392  page slice.
393  Page slice element only appears together with DTIM.
394
395  Details:
396  Page slice element also indicates AP has buffered data for which block, if a station is in
      ↪  that block, the station should first sleep, then wake up at coresponding page
      ↪  slice(TIM beacon) which includes that block.
397
398  When station wake up at that block, it check whether AP has data for itself. If has, keep
      ↪  awake to receive packets and go to sleep in the next beacon.
399  */
400
401 void configurePageSlice(void)
402 {
403   config.pageS.SetPageindex(config.pageIndex);
404   config.pageS.SetPagePeriod(config.pagePeriod);      // 2 TIM groups between DTIMs
405   config.pageS.SetPageSliceLen(config.pageSliceLength); // each TIM group has 1 block (2
      ↪  blocks in 2 TIM groups)
406   config.pageS.SetPageSliceCount(config.pageSliceCount);
407   config.pageS.SetBlockOffset(config.blockOffset);
408   config.pageS.SetTIMOffset(config.timOffset);
409   std::cout << "pageIndex=" << (int)config.pageIndex << ", pagePeriod=" <<
      ↪  (int)config.pagePeriod << ", pageSliceLength=" << (int)config.pageSliceLength << ",
      ↪  pageSliceCount=" << (int)config.pageSliceCount << ", blockOffset=" <<
      ↪  (int)config.blockOffset << ", timOffset=" << (int)config.timOffset << std::endl;
410   //  page 0
411   //  8 TIM(page slice) for one page
412   //  4 block (each page)
413   //  8 page slice
```

```
414   //  both offset are 0
415 }
416
417 void configureTIM(void)
418 {
419   config.tim.SetPageIndex(config.pageIndex);
420   if (config.pageSliceCount)
421     config.tim.SetDTIMPeriod(config.pageSliceCount); // not necessarily the same
422   else
423     config.tim.SetDTIMPeriod(1);
424
425   std::cout << "DTIM period=" << (int)config.pagePeriod << std::endl;
426 }
427
428 void checkRawAndTimConfiguration(void)
429 {
430   std::cout << "Checking RAW and TIM configuration..." << std::endl;
431   bool configIsCorrect = true;
432   NS_ASSERT(config.rps.rpsset.size());
433   // Number of page slices in a single page has to equal number of different RPS elements
      ↪  because
434   // If #PS > #RPS, the same RPS will be used in more than 1 PS and that is wrong because
435   // each PS can accommodate different AIDs (same RPS means same stations in RAWs)
436   if (config.pageSliceCount)
437   {
438   //  NS_ASSERT (config.pagePeriod == config.rps.rpsset.size());
439   }
440   for (int j = 0; j < config.rps.rpsset.size(); j++)
441   {
442     uint32_t totalRawTime = 0;
443     for (int i = 0; i < config.rps.rpsset[j]->GetNumberOfRawGroups(); i++)
444     {
445       totalRawTime += (120 *
          ↪  config.rps.rpsset[j]->GetRawAssigmentObj(i).GetSlotDurationCount() + 500) *
          ↪  config.rps.rpsset[j]->GetRawAssigmentObj(i).GetSlotNum();
446       auto aidStart = config.rps.rpsset[j]->GetRawAssigmentObj(i).GetRawGroupAIDStart();
447       auto aidEnd = config.rps.rpsset[j]->GetRawAssigmentObj(i).GetRawGroupAIDEnd();
448       configIsCorrect = check(aidStart, j) && check(aidEnd, j);
449       // AIDs in each RPS must comply with TIM in the following way:
450       // TIM0: 1-63; TIM1: 64-127; TIM2: 128-191; ...; TIM32: 1983-2047
451       // If RPS that belongs to TIM0 includes other AIDs (other than range [1-63])
          ↪  configuration is incorrect
452       NS_ASSERT(configIsCorrect);
453     }
454     NS_ASSERT(totalRawTime <= config.BeaconInterval);
455   }
456 }
457 // assumes each TIM has its own beacon - doesn't need to be the case as there has to be
    ↪  only PageSliceCount beacons between DTIMs
```

```
458 bool check(uint16_t aid, uint32_t index)
459 {
460   uint8_t block = (aid >> 6) & 0x001f;
461   NS_ASSERT(config.pageS.GetPageSliceLen() > 0);
462   //uint8_t toTim = (block - config.pageS.GetBlockOffset()) %
      ↪  config.pageS.GetPageSliceLen();
463   if (index == config.pageS.GetPageSliceCount() - 1 && config.pageS.GetPageSliceCount() !=
      ↪  0)
464   {
465     // the last page slice has 32 - the rest blocks
466     return (block <= 31) && (block >= index * config.pageS.GetPageSliceLen());
467   }
468   else if (config.pageS.GetPageSliceCount() == 0)
469     return true;
470
471   return (block >= index * config.pageS.GetPageSliceLen()) && (block < (index + 1) *
      ↪  config.pageS.GetPageSliceLen());
472 }
473
474 void sendStatistics(bool schedule)
475 {
476   eventManager.onUpdateStatistics(stats);
477   eventManager.onUpdateSlotStatistics(
478     transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval,
479     transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval);
480   // reset
481   std::fill(transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval.begin(),
482         transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval.end(), 0);
483   std::fill(transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval.begin(),
484         transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval.end(), 0);
485
486   if (schedule)
487     Simulator::Schedule(Seconds(config.visualizerSamplingInterval), &sendStatistics, true);
488 }
489
490 void onSTADeassociated(int i)
491 {
492   eventManager.onNodeDeassociated(*nodes[i]);
493 }
494
495 void updateNodesQueueLength()
496 {
497   for (uint32_t i = 0; i < config.Nsta; i++)
498   {
499     nodes[i]->UpdateQueueLength();
500     stats.get(i).EDCAQueueLength = nodes[i]->queueLength;
501   }
502   Simulator::Schedule(Seconds(0.5), &updateNodesQueueLength);
```

```
503 }
504
505 void onSTAAssociated(int i)
506 {
507   cout << "Node " << std::to_string(i) << " is associated and has aid "
508      << nodes[i]->aId << endl;
509
510   for (int k = 0; k < config.rps.rpsset.size(); k++)
511   {
512     for (int j = 0; j < config.rps.rpsset[k]->GetNumberOfRawGroups(); j++)
513     {
514       if (config.rps.rpsset[k]->GetRawAssigmentObj(j).GetRawGroupAIDStart() <= i + 1 && i +
        ↪  1 <= config.rps.rpsset[k]->GetRawAssigmentObj(j).GetRawGroupAIDEnd())
515       {
516         nodes[i]->rpsIndex = k + 1;
517         nodes[i]->rawGroupNumber = j + 1;
518         nodes[i]->rawSlotIndex =
519           nodes[i]->aId % config.rps.rpsset[k]->GetRawAssigmentObj(j).GetSlotNum() + 1;
520         cout << "Node " << i << " with AID " << (int)nodes[i]->aId << " belongs to " <<
          ↪  (int)nodes[i]->rawSlotIndex << " slot of RAW group "
521          << (int)nodes[i]->rawGroupNumber << " within the " << (int)nodes[i]->rpsIndex << "
           ↪  RPS." << endl;
522
523       }
524     }
525   }
526
527   eventManager.onNodeAssociated(*nodes[i]);
528
529   // RPS, Raw group and RAW slot assignment
530
531   if (GetAssocNum() == config.Nsta)
532   {
533     cout << "All " << AssocNum << " stations associated at " <<
      ↪  Simulator::Now().GetMicroSeconds() << "us, configuring clients & server" << endl;
534
535     // association complete, start sending packets
536     stats.TimeWhenEverySTAIsAssociated = Simulator::Now();
537
538     if (config.trafficType == "udp")
539     {
540       std::cout << "UDP" << std::endl;
541       configureUDPServer();
542       configureUDPClients();
543       //configurePowerBeacon();
544     }
545     else if (config.trafficType == "udpecho")
546     {
```

```
547       configureUDPEchoServer();
548       configureUDPEchoClients();
549     }
550     else if (config.trafficType == "tcpecho")
551     {
552       configureTCPEchoServer();
553       configureTCPEchoClients();
554     }
555     else if (config.trafficType == "tcppingpong")
556     {
557       configureTCPPingPongServer();
558       configureTCPPingPongClients();
559     }
560     else if (config.trafficType == "tcpipcamera")
561     {
562       configureTCPIPCameraServer();
563       configureTCPIPCameraClients();
564     }
565     else if (config.trafficType == "tcpfirmware")
566     {
567       configureTCPFirmwareServer();
568       configureTCPFirmwareClients();
569     }
570     else if (config.trafficType == "tcpsensor")
571     {
572       configureTCPSensorServer();
573       configureTCPSensorClients();
574     }
575     updateNodesQueueLength();
576   }
577 }
578
579
580
581 void RawGroupTrace(uint8_t oldValue, uint8_t newValue)
582 {
583   currentRawGroup = newValue;
584   cout << "  group " << std::to_string(newValue) << " at " <<
     ↪  Simulator::Now().GetMicroSeconds() << endl;
585   if (config.filesOutput)
586   {
587     //std::stringstream ss;
588     //ss << "/media//RawGroupTrace.csv";
589     //static std::fstream f(ss.str().c_str(), std::ios::out);
590     //f << std::fixed << Simulator::Now().GetSeconds() << ", " << currentRawGroup <<
     ↪  std::endl;
591   }
592 }
```

```
593
594 void RawSlotTrace(uint8_t oldValue, uint8_t newValue)
595 {
596   currentRawSlot = newValue;
597   //cout << "    slot " << std::to_string(newValue) << " at " <<
    ↪  Simulator::Now().GetMicroSeconds() << endl;
598   //if (config.filesOutput)
599   //{
600   //  std::stringstream ss;
601   //  ss << "/media//RawSlotTrace.csv";
602   //  static std::fstream f(ss.str().c_str(), std::ios::out);
603   //  f << std::fixed << Simulator::Now().GetSeconds() << ", " << currentRawSlot <<
    ↪  std::endl;
604   //}
605 }
606
607 void PacketToApFromNode(Ptr<const Packet>, Mac48Address from)
608 {
609   uint8_t mac[6];
610     from.CopyTo (mac);
611     uint8_t aid_l = mac[5];
612     uint8_t aid_h = mac[4] & 0x1f;
613     uint16_t aid = (aid_h << 8) | (aid_l << 0);
614
615   //std::cout << Simulator::Now().GetMicroSeconds() << "us, AP RX PKT from: " << aid <<
    ↪  std::endl;
616     //NS_LOG_UNCOND ("Disassociation request from aid " << aid);
617
618   config.NumberOfPacketsToApFromNode[aid - 1]++;
619 }
620
621 void configureNodes(NodeContainer &wifiStaNode, NetDeviceContainer &staDevice)
622 {
623   cout << "Configuring STA Node trace sources..." << endl;
624
625   for (uint32_t i = 0; i < config.Nsta; i++)
626   {
627
628     cout << "Hooking up trace sources for STA " << i << endl;
629
630     NodeEntry *n = new NodeEntry(i, &stats, wifiStaNode.Get(i),
631                   staDevice.Get(i));
632
633     n->SetAssociatedCallback([=]
634                 { onSTAAssociated(i); });
635     n->SetDeassociatedCallback([=]
636                   { onSTADeassociated(i); });
637
```

```
638     nodes.push_back(n);
639     // hook up Associated and Deassociated events
640     Config::Connect(
641       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/Assoc",
642       MakeCallback(&NodeEntry::SetAssociation, n));
643     Config::Connect(
644       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/DeAssoc",
645       MakeCallback(&NodeEntry::UnsetAssociation, n));
646     Config::Connect(
647       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/NrOfTransmiss
648       MakeCallback(
649         &NodeEntry::OnNrOfTransmissionsDuringRAWSlotChanged,
650         n)); // not implem
651
652     Config::Connect("/NodeList/" + std::to_string(i) +
    ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/S1gBeaconMissed
    ↪  MakeCallback(&NodeEntry::OnS1gBeaconMissed, n));
653
654     Config::Connect(
655       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/PacketDropped
656       MakeCallback(&NodeEntry::OnMacPacketDropped, n));
657     Config::Connect(
658       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/PacketDropped
659       MakeCallback(&OnMacPacketDropped2));
660     Config::Connect(
661       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/Collision",
662       MakeCallback(&NodeEntry::OnCollision, n));
663     Config::Connect(
664       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/TransmissionW
665       MakeCallback(&NodeEntry::OnTransmissionWillCrossRAWBoundary,
666              n)); //?
667
668     // hook up TX
669     Config::Connect(
670       "/NodeList/" + std::to_string(i) +
        ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxBegin",
671       MakeCallback(&NodeEntry::OnPhyTxBegin, n));
672     Config::Connect(
673       "/NodeList/" + std::to_string(i) + "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
674       MakeCallback(&NodeEntry::OnPhyTxEnd, n));
675     Config::Connect(
```

```
676       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxDropWithReason",
677       MakeCallback(&NodeEntry::OnPhyTxDrop, n)); //?
678     Config::Connect(
679       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxDropWithReason",
680       MakeCallback(&OnPhyTxDrop2)); //?
681
682     // hook up RX
683     Config::Connect(
684       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxBegin",
685       MakeCallback(&NodeEntry::OnPhyRxBegin, n));
686     Config::Connect(
687       "/NodeList/" + std::to_string(i) + "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
688       MakeCallback(&NodeEntry::OnPhyRxEnd, n));
689     Config::Connect(
690       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxDropWithReason",
691       MakeCallback(&NodeEntry::OnPhyRxDrop,n));
692     Config::Connect(
693       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxDropWithReason",
694       MakeCallback(&OnPhyRxDrop2));
695
696     // hook up MAC traces
697     Config::Connect(
698       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/RemoteStationManager/MacTxRtsFailed",
699       MakeCallback(&NodeEntry::OnMacTxRtsFailed, n)); //?
700     Config::Connect(
701       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/RemoteStationManager/MacTxDataFailed",
702       MakeCallback(&NodeEntry::OnMacTxDataFailed, n));
703     Config::Connect(
704       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/RemoteStationManager/MacTxFinalRtsFailed",
705       MakeCallback(&NodeEntry::OnMacTxFinalRtsFailed, n)); //?
706     Config::Connect(
707       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/RemoteStationManager/MacTxFinalDataFailed",
708       MakeCallback(&NodeEntry::OnMacTxFinalDataFailed, n)); //?
709
710     // hook up PHY State change
711     Config::Connect(
712       "/NodeList/" + std::to_string(i) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/State/State",
713       MakeCallback(&NodeEntry::OnPhyStateChange, n));
```

```
714   }
715 }
716
717 int getBandwidth(string dataMode)
718 {
719   if (dataMode == "MCS1_0" || dataMode == "MCS1_1" || dataMode == "MCS1_2" || dataMode ==
      ↪  "MCS1_3" || dataMode == "MCS1_4" || dataMode == "MCS1_5" || dataMode == "MCS1_6" ||
      ↪  dataMode == "MCS1_7" || dataMode == "MCS1_8" || dataMode == "MCS1_9" || dataMode ==
      ↪  "MCS1_10")
720     return 1;
721
722   else if (dataMode == "MCS2_0" || dataMode == "MCS2_1" || dataMode == "MCS2_2" || dataMode
      ↪  == "MCS2_3" || dataMode == "MCS2_4" || dataMode == "MCS2_5" || dataMode == "MCS2_6"
      ↪  || dataMode == "MCS2_7" || dataMode == "MCS2_8")
723     return 2;
724
725   return 0;
726 }
727
728 string getWifiMode(string dataMode)
729 {
730   if (dataMode == "MCS1_0")
731     return "OfdmRate300KbpsBW1MHz";
732   else if (dataMode == "MCS1_1")
733     return "OfdmRate600KbpsBW1MHz";
734   else if (dataMode == "MCS1_2")
735     return "OfdmRate900KbpsBW1MHz";
736   else if (dataMode == "MCS1_3")
737     return "OfdmRate1_2MbpsBW1MHz";
738   else if (dataMode == "MCS1_4")
739     return "OfdmRate1_8MbpsBW1MHz";
740   else if (dataMode == "MCS1_5")
741     return "OfdmRate2_4MbpsBW1MHz";
742   else if (dataMode == "MCS1_6")
743     return "OfdmRate2_7MbpsBW1MHz";
744   else if (dataMode == "MCS1_7")
745     return "OfdmRate3MbpsBW1MHz";
746   else if (dataMode == "MCS1_8")
747     return "OfdmRate3_6MbpsBW1MHz";
748   else if (dataMode == "MCS1_9")
749     return "OfdmRate4MbpsBW1MHz";
750   else if (dataMode == "MCS1_10")
751     return "OfdmRate150KbpsBW1MHz";
752
753   else if (dataMode == "MCS2_0")
754     return "OfdmRate650KbpsBW2MHz";
755   else if (dataMode == "MCS2_1")
756     return "OfdmRate1_3MbpsBW2MHz";
```

```
757   else if (dataMode == "MCS2_2")
758     return "OfdmRate1_95MbpsBW2MHz";
759   else if (dataMode == "MCS2_3")
760     return "OfdmRate2_6MbpsBW2MHz";
761   else if (dataMode == "MCS2_4")
762     return "OfdmRate3_9MbpsBW2MHz";
763   else if (dataMode == "MCS2_5")
764     return "OfdmRate5_2MbpsBW2MHz";
765   else if (dataMode == "MCS2_6")
766     return "OfdmRate5_85MbpsBW2MHz";
767   else if (dataMode == "MCS2_7")
768     return "OfdmRate6_5MbpsBW2MHz";
769   else if (dataMode == "MCS2_8")
770     return "OfdmRate7_8MbpsBW2MHz";
771   return "";
772 }
773
774 void OnAPPhyRxDrop(std::string context, Ptr<const Packet> packet,
775             DropReason reason)
776 {
777   // THIS REQUIRES PACKET METADATA ENABLE!
778   auto pCopy = packet->Copy();
779   auto it = pCopy->BeginItem();
780   while (it.HasNext())
781   {
782
783     auto item = it.Next();
784     Callback<ObjectBase *> constructor = item.tid.GetConstructor();
785
786     ObjectBase *instance = constructor();
787     Chunk *chunk = dynamic_cast<Chunk *>(instance);
788     chunk->Deserialize(item.current);
789
790     if (dynamic_cast<WifiMacHeader *>(chunk))
791     {
792       WifiMacHeader *hdr = (WifiMacHeader *)chunk;
793
794
795       int staId = -1;
796       if (!config.useV6)
797       {
798         for (uint32_t i = 0; i < staNodeInterface.GetN(); i++)
799         {
800           if (wifiStaNode.Get(i)->GetDevice(0)->GetAddress() == hdr->GetAddr2())
801           {
802             staId = i;
803             break;
804           }
```

```
805         }
806       }
807       else
808       {
809         for (uint32_t i = 0; i < staNodeInterface6.GetN(); i++)
810         {
811           if (wifiStaNode.Get(i)->GetDevice(0)->GetAddress() == hdr->GetAddr2())
812           {
813             staId = i;
814             break;
815           }
816         }
817       }
818       if (staId != -1)
819       {
820         stats.get(staId).NumberOfDropsByReasonAtAP[reason]++;
821       }
822       delete chunk;
823       break;
824     }
825     else
826       delete chunk;
827   }
828 }
829
830 void OnAPPacketToTransmitReceived(string context, Ptr<const Packet> packet,
831                   Mac48Address to, bool isScheduled, bool isDuringSlotOfSTA,
832                   Time timeLeftInSlot)
833 {
834   int staId = -1;
835   if (!config.useV6)
836   {
837     for (uint32_t i = 0; i < staNodeInterface.GetN(); i++)
838     {
839       if (wifiStaNode.Get(i)->GetDevice(0)->GetAddress() == to)
840       {
841         staId = i;
842         break;
843       }
844     }
845   }
846   else
847   {
848     for (uint32_t i = 0; i < staNodeInterface6.GetN(); i++)
849     {
850       if (wifiStaNode.Get(i)->GetDevice(0)->GetAddress() == to)
851       {
852         staId = i;
```

```
853         break;
854       }
855     }
856   }
857   if (staId != -1)
858   {
859     if (isScheduled)
860       stats.get(staId).NumberOfAPScheduledPacketForNodeInNextSlot++;
861     else
862     {
863       stats.get(staId).NumberOfAPSentPacketForNodeImmediately++;
864       stats.get(staId).APTotalTimeRemainingWhenSendingPacketInSameSlot +=
865         timeLeftInSlot;
866     }
867   }
868 }
869
870 void onChannelTransmission(Ptr<NetDevice> senderDevice, Ptr<Packet> packet)
871 {
872   int rpsIndex = currentRps - 1;
873   int rawGroup = currentRawGroup - 1;
874   int slotIndex = currentRawSlot - 1;
875   //cout << rpsIndex << "    " << rawGroup << "    " << slotIndex << "    " << endl;
876
877   uint64_t iSlot = slotIndex;
878   if (rpsIndex > 0)
879     for (int r = rpsIndex - 1; r >= 0; r--)
880       for (int g = 0; g < config.rps.rpsset[r]->GetNumberOfRawGroups(); g++)
881         iSlot += config.rps.rpsset[r]->GetRawAssigmentObj(g).GetSlotNum();
882
883   if (rawGroup > 0)
884     for (int i = rawGroup - 1; i >= 0; i--)
885       iSlot += config.rps.rpsset[rpsIndex]->GetRawAssigmentObj(i).GetSlotNum();
886
887   if (rpsIndex >= 0 && rawGroup >= 0 && slotIndex >= 0)
888   {
889     if (senderDevice->GetAddress() == apDevice.Get(0)->GetAddress())
890     {
891       // from AP
892       transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval[iSlot] +=
          ↪ packet->GetSerializedSize();
893     }
894     else
895     {
896       // from STA
897       transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval[iSlot] +=
          ↪ packet->GetSerializedSize();
898     }
```

```
899   }
900   std::cout << "------------ packetSerializedSize = " << packet->GetSerializedSize() <<
      ↪  std::endl;
901   std::cout << "------------ txAP[" << iSlot <<"] = " <<
      ↪  transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval[iSlot] << std::endl;
902   std::cout << "------------ txSTA[" << iSlot <<"] = " <<
      ↪  transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval[iSlot] << std::endl;
903 }
904
905 int getSTAIdFromAddress(Ipv4Address from)
906 {
907   int staId = -1;
908   for (int i = 0; i < staNodeInterface.GetN(); i++)
909   {
910     if (staNodeInterface.GetAddress(i) == from)
911     {
912       staId = i;
913       break;
914     }
915   }
916   return staId;
917 }
918
919 void udpPacketReceivedAtServer(Ptr<const Packet> packet, Address from)
920 { // works
921   // cout << "++++++++++++udpPacketReceivedAtServer" << endl;
922   int staId = getSTAIdFromAddress(
923     InetSocketAddress::ConvertFrom(from).GetIpv4());
924   if (staId != -1)
925     nodes[staId]->OnUdpPacketReceivedAtAP(packet);
926   else
927     cout << "*** Node could not be determined from received packet at AP "
928        << endl;
929 }
930
931 void tcpPacketReceivedAtServer(Ptr<const Packet> packet, Address from)
932 {
933   int staId = getSTAIdFromAddress(
934     InetSocketAddress::ConvertFrom(from).GetIpv4());
935   if (staId != -1)
936     nodes[staId]->OnTcpPacketReceivedAtAP(packet);
937   else
938     cout << "*** Node could not be determined from received packet at AP "
939        << endl;
940 }
941
942 void tcpRetransmissionAtServer(Address to)
943 {
```

```
944   int staId = getSTAIdFromAddress(Ipv4Address::ConvertFrom(to));
945   if (staId != -1)
946     nodes[staId]->OnTcpRetransmissionAtAP();
947   else
948     cout << "*** Node could not be determined from received packet at AP "
949        << endl;
950 }
951
952 void tcpPacketDroppedAtServer(Address to, Ptr<Packet> packet,
953                 DropReason reason)
954 {
955   int staId = getSTAIdFromAddress(Ipv4Address::ConvertFrom(to));
956   if (staId != -1)
957   {
958     stats.get(staId).NumberOfDropsByReasonAtAP[reason]++;
959   }
960 }
961
962 void tcpStateChangeAtServer(TcpSocket::TcpStates_t oldState,
963               TcpSocket::TcpStates_t newState, Address to)
964 {
965
966   int staId = getSTAIdFromAddress(
967     InetSocketAddress::ConvertFrom(to).GetIpv4());
968   if (staId != -1)
969     nodes[staId]->OnTcpStateChangedAtAP(oldState, newState);
970   else
971     cout << "*** Node could not be determined from received packet at AP "
972        << endl;
973
974   // cout << Simulator::Now().GetMicroSeconds() << " ********** TCP SERVER SOCKET STATE
   ↪  CHANGED FROM " << oldState << " TO " << newState << endl;
975 }
976
977 void tcpIPCameraDataReceivedAtServer(Address from, uint16_t nrOfBytes)
978 {
979   int staId = getSTAIdFromAddress(
980     InetSocketAddress::ConvertFrom(from).GetIpv4());
981   if (staId != -1)
982     nodes[staId]->OnTcpIPCameraDataReceivedAtAP(nrOfBytes);
983   else
984     cout << "*** Node could not be determined from received packet at AP "
985        << endl;
986 }
987
988 void GenerateTraffic(Ptr<Socket> socket, uint32_t pktSize, uint32_t pktCount, Time
   ↪  pktInterval)
989 {
```

```
990   if (pktCount > 0)
991   {
992     //stats.get(3).NumberOfSentPackets++;
993     socket->Send(Create<Packet>(pktSize));
994     // cout << "\n\nGenerateTraffic: " << socket << ", "
995     //                 << pktSize << ", "
996     //                 << pktCount << ", "
997     //                   << Simulator::Now().GetNanoSeconds() << "ns, "
998     //                 << pktInterval << endl;
999
1000    Simulator::Schedule(pktInterval, &GenerateTraffic,
1001              socket, pktSize, pktCount - 1, pktInterval);
1002  }
1003  else
1004  {
1005    socket->Close();
1006  }
1007 }
1008
1009 void configureUDPServer()
1010 {
1011   UdpServerHelper myServer(9);
1012   serverApp = myServer.Install(wifiApNode);
1013   serverApp.Get(0)->TraceConnectWithoutContext("Rx",
1014                         MakeCallback(&udpPacketReceivedAtServer));
1015   serverApp.Start(Seconds(0));
1016
1017 }
1018
1019
1020
1021
1022 void configureUDPEchoServer()
1023 {
1024   UdpEchoServerHelper myServer(9);
1025   serverApp = myServer.Install(wifiApNode);
1026   serverApp.Get(0)->TraceConnectWithoutContext("Rx",
1027                         MakeCallback(&udpPacketReceivedAtServer));
1028   serverApp.Start(Seconds(0));
1029 }
1030
1031 void configureTCPEchoServer()
1032 {
1033   TcpEchoServerHelper myServer(80);
1034   serverApp = myServer.Install(wifiApNode);
1035   wireTCPServer(serverApp);
1036   serverApp.Start(Seconds(0));
1037 }
```

```
1038
1039 void configureTCPPingPongServer()
1040 {
1041   // TCP ping pong is a test for the new base tcp-client and tcp-server applications
1042   ObjectFactory factory;
1043   factory.SetTypeId(TCPPingPongServer::GetTypeId());
1044   factory.Set("Port", UintegerValue(81));
1045
1046   Ptr<Application> tcpServer = factory.Create<TCPPingPongServer>();
1047   wifiApNode.Get(0)->AddApplication(tcpServer);
1048
1049   auto serverApp = ApplicationContainer(tcpServer);
1050   wireTCPServer(serverApp);
1051   serverApp.Start(Seconds(0));
1052 }
1053
1054 void configureTCPPingPongClients()
1055 {
1056
1057   ObjectFactory factory;
1058   factory.SetTypeId(TCPPingPongClient::GetTypeId());
1059   factory.Set("Interval", TimeValue(MilliSeconds(config.trafficInterval)));
1060   factory.Set("PacketSize", UintegerValue(config.payloadSize));
1061
1062   factory.Set("RemoteAddress",
1063         Ipv4AddressValue(apNodeInterface.GetAddress(0)));
1064   factory.Set("RemotePort", UintegerValue(81));
1065
1066   Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable>();
1067
1068   for (uint16_t i = 0; i < config.Nsta; i++)
1069   {
1070
1071     Ptr<Application> tcpClient = factory.Create<TCPPingPongClient>();
1072     wifiStaNode.Get(i)->AddApplication(tcpClient);
1073     auto clientApp = ApplicationContainer(tcpClient);
1074     wireTCPClient(clientApp, i);
1075
1076     double random = m_rv->GetValue(0, config.trafficInterval);
1077     clientApp.Start(MilliSeconds(0 + random));
1078     // clientApp.Stop(Seconds(simulationTime + 1));
1079   }
1080 }
1081
1082 void configureTCPIPCameraServer()
1083 {
1084   ObjectFactory factory;
1085   factory.SetTypeId(TCPIPCameraServer::GetTypeId());
```

```
  factory.Set("Port", UintegerValue(82));

  Ptr<Application> tcpServer = factory.Create<TCPIPCameraServer>();
  wifiApNode.Get(0)->AddApplication(tcpServer);

  auto serverApp = ApplicationContainer(tcpServer);
  wireTCPServer(serverApp);
  serverApp.Start(Seconds(0));
  //  serverApp.Stop(Seconds(config.simulationTime));
}

void configureTCPIPCameraClients()
{

  ObjectFactory factory;
  factory.SetTypeId(TCPIPCameraClient::GetTypeId());
  factory.Set("MotionPercentage",
        DoubleValue(config.ipcameraMotionPercentage));
  factory.Set("MotionDuration",
        TimeValue(Seconds(config.ipcameraMotionDuration)));
  factory.Set("DataRate", UintegerValue(config.ipcameraDataRate));

  factory.Set("PacketSize", UintegerValue(config.payloadSize));

  factory.Set("RemoteAddress",
        Ipv4AddressValue(apNodeInterface.GetAddress(0)));
  factory.Set("RemotePort", UintegerValue(82));

  Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable>();

  for (uint16_t i = 0; i < config.Nsta; i++)
  {

    Ptr<Application> tcpClient = factory.Create<TCPIPCameraClient>();
    wifiStaNode.Get(i)->AddApplication(tcpClient);
    auto clientApp = ApplicationContainer(tcpClient);
    wireTCPClient(clientApp, i);

    clientApp.Start(MilliSeconds(0));
    // clientApp.Stop(Seconds(config.simulationTime));
  }
}

void configureTCPFirmwareServer()
{
  ObjectFactory factory;
  factory.SetTypeId(TCPFirmwareServer::GetTypeId());
  factory.Set("Port", UintegerValue(83));
```

```
1134
1135    factory.Set("FirmwareSize", UintegerValue(config.firmwareSize));
1136    factory.Set("BlockSize", UintegerValue(config.firmwareBlockSize));
1137    factory.Set("NewUpdateProbability",
1138          DoubleValue(config.firmwareNewUpdateProbability));
1139
1140    Ptr<Application> tcpServer = factory.Create<TCPFirmwareServer>();
1141    wifiApNode.Get(0)->AddApplication(tcpServer);
1142
1143    auto serverApp = ApplicationContainer(tcpServer);
1144    wireTCPServer(serverApp);
1145    serverApp.Start(Seconds(0));
1146    //  serverApp.Stop(Seconds(config.simulationTime));
1147  }
1148
1149  void configureTCPFirmwareClients()
1150  {
1151
1152    ObjectFactory factory;
1153    factory.SetTypeId(TCPFirmwareClient::GetTypeId());
1154    factory.Set("CorruptionProbability",
1155          DoubleValue(config.firmwareCorruptionProbability));
1156    factory.Set("VersionCheckInterval",
1157          TimeValue(MilliSeconds(config.firmwareVersionCheckInterval)));
1158    factory.Set("PacketSize", UintegerValue(config.payloadSize));
1159
1160    factory.Set("RemoteAddress",
1161          Ipv4AddressValue(apNodeInterface.GetAddress(0)));
1162    factory.Set("RemotePort", UintegerValue(83));
1163
1164    Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable>();
1165
1166    for (uint16_t i = 0; i < config.Nsta; i++)
1167    {
1168
1169      Ptr<Application> tcpClient = factory.Create<TCPFirmwareClient>();
1170      wifiStaNode.Get(i)->AddApplication(tcpClient);
1171      auto clientApp = ApplicationContainer(tcpClient);
1172      wireTCPClient(clientApp, i);
1173
1174      double random = m_rv->GetValue(0, config.trafficInterval);
1175      clientApp.Start(MilliSeconds(0 + random));
1176      clientApp.Stop(Seconds(config.simulationTime));
1177    }
1178  }
1179
1180  void configureTCPSensorServer()
1181  {
```

```
1182   ObjectFactory factory;
1183   factory.SetTypeId(TCPSensorServer::GetTypeId());
1184   factory.Set("Port", UintegerValue(84));
1185
1186   Ptr<Application> tcpServer = factory.Create<TCPSensorServer>();
1187   wifiApNode.Get(0)->AddApplication(tcpServer);
1188
1189   auto serverApp = ApplicationContainer(tcpServer);
1190   wireTCPServer(serverApp);
1191   serverApp.Start(Seconds(0));
1192   //  serverApp.Stop(Seconds(config.simulationTime));
1193 }
1194
1195 void configureTCPSensorClients()
1196 {
1197
1198   ObjectFactory factory;
1199   factory.SetTypeId(TCPSensorClient::GetTypeId());
1200
1201   factory.Set("Interval", TimeValue(MilliSeconds(config.trafficInterval)));
1202   factory.Set("PacketSize", UintegerValue(config.payloadSize));
1203   factory.Set("MeasurementSize", UintegerValue(config.sensorMeasurementSize));
1204
1205   factory.Set("RemoteAddress",
1206         Ipv4AddressValue(apNodeInterface.GetAddress(0)));
1207   factory.Set("RemotePort", UintegerValue(84));
1208
1209   Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable>();
1210
1211   for (uint16_t i = 0; i < config.Nsta; i++)
1212   {
1213
1214     Ptr<Application> tcpClient = factory.Create<TCPSensorClient>();
1215     wifiStaNode.Get(i)->AddApplication(tcpClient);
1216     auto clientApp = ApplicationContainer(tcpClient);
1217     wireTCPClient(clientApp, i);
1218
1219     double random = m_rv->GetValue(0, config.trafficInterval);
1220     //clientApp.Start(MilliSeconds(0 + random));
1221     clientApp.Start(MilliSeconds(1));
1222     clientApp.Stop(Seconds(config.simulationTime));
1223   }
1224 }
1225
1226 void wireTCPServer(ApplicationContainer serverApp)
1227 {
1228   serverApp.Get(0)->TraceConnectWithoutContext("Rx",
1229                           MakeCallback(&tcpPacketReceivedAtServer));
```

```
  serverApp.Get(0)->TraceConnectWithoutContext("Retransmission",
                        MakeCallback(&tcpRetransmissionAtServer));
  serverApp.Get(0)->TraceConnectWithoutContext("PacketDropped",
                        MakeCallback(&tcpPacketDroppedAtServer));
  serverApp.Get(0)->TraceConnectWithoutContext("TCPStateChanged",
                        MakeCallback(&tcpStateChangeAtServer));

  if (config.trafficType == "tcpipcamera")
  {
    serverApp.Get(0)->TraceConnectWithoutContext("DataReceived",
                          MakeCallback(&tcpIPCameraDataReceivedAtServer));
  }
}

void wireTCPClient(ApplicationContainer clientApp, int i)
{

  clientApp.Get(0)->TraceConnectWithoutContext("Tx",
                        MakeCallback(&NodeEntry::OnTcpPacketSent, nodes[i]));
  clientApp.Get(0)->TraceConnectWithoutContext("Rx",
                        MakeCallback(&NodeEntry::OnTcpEchoPacketReceived, nodes[i]));

  clientApp.Get(0)->TraceConnectWithoutContext("CongestionWindow",
                        MakeCallback(&NodeEntry::OnTcpCongestionWindowChanged, nodes[i]));
  clientApp.Get(0)->TraceConnectWithoutContext("RTO",
                        MakeCallback(&NodeEntry::OnTcpRTOChanged, nodes[i]));
  clientApp.Get(0)->TraceConnectWithoutContext("RTT",
                        MakeCallback(&NodeEntry::OnTcpRTTChanged, nodes[i]));
  clientApp.Get(0)->TraceConnectWithoutContext("SlowStartThreshold",
                        MakeCallback(&NodeEntry::OnTcpSlowStartThresholdChanged,
                        ↪  nodes[i]));
  clientApp.Get(0)->TraceConnectWithoutContext("EstimatedBW",
                        MakeCallback(&NodeEntry::OnTcpEstimatedBWChanged, nodes[i]));

  clientApp.Get(0)->TraceConnectWithoutContext("TCPStateChanged",
                        MakeCallback(&NodeEntry::OnTcpStateChanged, nodes[i]));
  clientApp.Get(0)->TraceConnectWithoutContext("Retransmission",
                        MakeCallback(&NodeEntry::OnTcpRetransmission, nodes[i]));

  clientApp.Get(0)->TraceConnectWithoutContext("PacketDropped",
                        MakeCallback(&NodeEntry::OnTcpPacketDropped, nodes[i]));

  if (config.trafficType == "tcpfirmware")
  {
    clientApp.Get(0)->TraceConnectWithoutContext("FirmwareUpdated",
                          MakeCallback(&NodeEntry::OnTcpFirmwareUpdated, nodes[i]));
  }
  else if (config.trafficType == "tcpipcamera")
```

```
  {
    clientApp.Get(0)->TraceConnectWithoutContext("DataSent",
                            MakeCallback(&NodeEntry::OnTcpIPCameraDataSent, nodes[i]));
    clientApp.Get(0)->TraceConnectWithoutContext("StreamStateChanged",
                            MakeCallback(&NodeEntry::OnTcpIPCameraStreamStateChanged,
                                   nodes[i]));
  }
}

void configureTCPEchoClients()
{
  TcpEchoClientHelper clientHelper(apNodeInterface.GetAddress(0), 80); // address of remote
  ↪  node
  clientHelper.SetAttribute("MaxPackets", UintegerValue(4294967295u));
  clientHelper.SetAttribute("Interval",
                TimeValue(MilliSeconds(config.trafficInterval)));
  // clientHelper.SetAttribute("IntervalDeviation",
  ↪  TimeValue(MilliSeconds(config.trafficIntervalDeviation)));
  clientHelper.SetAttribute("PacketSize", UintegerValue(config.payloadSize));

  Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable>();

  for (uint16_t i = 0; i < config.Nsta; i++)
  {
    ApplicationContainer clientApp = clientHelper.Install(
      wifiStaNode.Get(i));
    wireTCPClient(clientApp, i);

    double random = m_rv->GetValue(0, config.trafficInterval);
    clientApp.Start(MilliSeconds(0 + random));
    // clientApp.Stop(Seconds(simulationTime + 1));
  }
}

void configureUDPClients()
{


  UdpClientHelper myClient(apNodeInterface.GetAddress(0), 9); // address of remote node
  myClient.SetAttribute("MaxPackets", config.maxNumberOfPackets);
  myClient.SetAttribute("PacketSize", UintegerValue(config.payloadSize));
  traffic_sta.clear();
  ifstream trafficfile(config.TrafficPath);
  if (trafficfile.is_open())
  {
    uint16_t sta_id;
    float sta_traffic;
    for (uint16_t kk = 0; kk < config.Nsta; kk++)
```

```
    {
      trafficfile >> sta_id;
      trafficfile >> sta_traffic;
      traffic_sta.insert(std::make_pair(sta_id, sta_traffic)); // insert data
      //cout << "sta_id = " << sta_id << " sta_traffic = " << sta_traffic << "\n";
    }
    trafficfile.close();
  }
  else
    cout << "Unable to open traffic file \n";


  for (int i=0; i< config.Nsta; i++)
  {

    myClient.SetAttribute("Interval", TimeValue(MicroSeconds(config.udpInterval))); // TODO
    ↪  add to nodeEntry and visualize
    Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable>();
    double randomStart = 0.0;
    randomStart = m_rv->GetValue(0.0, 1.0);
    ApplicationContainer clientApp = myClient.Install( wifiStaNode.Get(i));

    clientApp.Get(0)->TraceConnectWithoutContext("Tx",
    ↪  MakeCallback(&NodeEntry::OnUdpPacketSent, nodes[i]));
    clientApp.Start(Seconds(3.0 + randomStart));
  }
  AppStartTime = Simulator::Now().GetSeconds() + 1;
  // Simulator::Stop (Seconds (config.simulationTime+1));


}

void configureUDPEchoClients()
{
  UdpEchoClientHelper clientHelper(apNodeInterface.GetAddress(0), 9); // address of remote
  ↪  node
  clientHelper.SetAttribute("MaxPackets", UintegerValue(4294967295u));
  //clientHelper.SetAttribute("Interval", TimeValue(Seconds(5.0)));
  clientHelper.SetAttribute("Interval", TimeValue(MilliSeconds(config.trafficInterval)));

  // clientHelper.SetAttribute("IntervalDeviation",
  ↪  TimeValue(MilliSeconds(config.trafficIntervalDeviation)));
  clientHelper.SetAttribute("PacketSize", UintegerValue(config.payloadSize));

  Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable>();

  for (uint16_t i = 0; i < config.Nsta; i++)
  {
```

```
1367      ApplicationContainer clientApp = clientHelper.Install(
1368        wifiStaNode.Get(i));
1369      clientApp.Get(0)->TraceConnectWithoutContext("Tx",
1370                            MakeCallback(&NodeEntry::OnUdpPacketSent, nodes[i]));
1371      clientApp.Get(0)->TraceConnectWithoutContext("Rx",
1372                            MakeCallback(&NodeEntry::OnUdpEchoPacketReceived, nodes[i]));
1373
1374      double random = m_rv->GetValue(0, config.trafficInterval);
1375      clientApp.Start(Seconds(5 + random));
1376      // clientApp.Stop(Seconds(simulationTime + 1));
1377    }
1378  }
1379
1380  Time timeIdleArray[MaxSta];
1381  Time timeRxArray[MaxSta];
1382  Time timeTxArray[MaxSta];
1383  Time timeSleepArray[MaxSta];
1384  Time timeCollisionArray[MaxSta];
1385
1386  Time timeIdleNotAssociated[MaxSta];
1387  Time timeRxNotAssociated[MaxSta];
1388  Time timeTxNotAssociated[MaxSta];
1389  Time timeSleepNotAssociated[MaxSta];
1390  Time timeCollisionNotAssociated[MaxSta];
1391
1392  double dist[MaxSta];
1393
1394  // it prints the information regarding the state of the device
1395  void PhyStateTrace(std::string context, Time start, Time duration,
1396             enum WifiPhy::State state)
1397  {
1398
1399    /*Get the number of the node from the context*/
1400    /*context = "/NodeList/"+strSTA+"/DeviceList/'*'/Phy/$ns3::YansWifiPhy/State/State"*/
1401    unsigned first = context.find("t/");
1402    unsigned last = context.find("/D");
1403    string strNew = context.substr((first + 2), (last - first - 2));
1404
1405    int node = std::stoi(strNew);
1406
1407    if (nodes[node]->isAssociated)
1408    {
1409
1410      //std::stringstream ss;
1411      //ss << "/media/Ass.csv";
1412      //static std::fstream f(ss.str().c_str(), std::ios::out);
1413      //f << Simulator::Now().GetSeconds() << ", " << node << ", "  << state << ", " <<
      ↪  duration.GetSeconds() << std::endl;
```

```
1414
1415      switch (state)
1416      {
1417      case WifiPhy::State::SLEEP: // Sleep
1418        timeSleepArray[node] = timeSleepArray[node] + duration;
1419        // NS_LOG_UNCOND(to_string(node + 1) + ",SLEEP," + to_string(start.GetMicroSeconds())
          ↪  + " " + to_string(duration.GetMicroSeconds()));
1420        break;
1421      case WifiPhy::State::IDLE: // Idle
1422        timeIdleArray[node] = timeIdleArray[node] + duration;
1423        // NS_LOG_UNCOND(to_string(node + 1) + ",IDLE," + to_string(start.GetMicroSeconds())
          ↪  + " " + to_string(duration.GetMicroSeconds()));
1424        break;
1425      case WifiPhy::State::TX: // Tx
1426        timeTxArray[node] = timeTxArray[node] + duration;
1427        // NS_LOG_UNCOND (to_string(node+1) + ",TX," + to_string(start.GetMicroSeconds()) + "
          ↪  " + to_string(duration.GetMicroSeconds()));
1428        break;
1429      case WifiPhy::State::RX: // Rx
1430        timeRxArray[node] = timeRxArray[node] + duration;
1431        // NS_LOG_UNCOND (to_string(node+1) + ",RX," + to_string(start.GetMicroSeconds()) + "
          ↪  " + to_string(duration.GetMicroSeconds()));
1432        break;
1433      case WifiPhy::State::CCA_BUSY: // CCA_BUSY
1434        timeCollisionArray[node] = timeCollisionArray[node] + duration;
1435        // NS_LOG_UNCOND (to_string(node+1) + ",CCA_BUSY," +
          ↪  to_string(start.GetMicroSeconds()) + " " +
          ↪  to_string(duration.GetMicroSeconds()));
1436        break;
1437      }
1438    }
1439    else
1440    {
1441      //std::stringstream ss;
1442      //ss << "/media/NotAss.csv";
1443      //static std::fstream f(ss.str().c_str(), std::ios::out);
1444      //f << Simulator::Now().GetSeconds() << ", " << node << ", "  << state << ", " <<
       ↪  duration.GetSeconds() << std::endl;
1445
1446      switch (state)
1447      {
1448      case WifiPhy::State::SLEEP: // Sleep
1449        timeSleepNotAssociated[node] = timeSleepNotAssociated[node] + duration;
1450        // NS_LOG_UNCOND(to_string(node + 1) + ",SLEEP," + to_string(start.GetMicroSeconds())
          ↪  + " " + to_string(duration.GetMicroSeconds()));
1451        break;
1452      case WifiPhy::State::IDLE: // Idle
1453        timeIdleNotAssociated[node] = timeIdleNotAssociated[node] + duration;
```

```
1454        // NS_LOG_UNCOND(to_string(node + 1) + ",IDLE," + to_string(start.GetMicroSeconds())
            ↪  + " " + to_string(duration.GetMicroSeconds()));
1455        break;
1456      case WifiPhy::State::TX: // Tx
1457        timeTxNotAssociated[node] = timeTxNotAssociated[node] + duration;
1458        // NS_LOG_UNCOND (to_string(node+1) + ",TX," + to_string(start.GetMicroSeconds()) + "
            ↪  " + to_string(duration.GetMicroSeconds()));
1459        break;
1460      case WifiPhy::State::RX: // Rx
1461        timeRxNotAssociated[node] = timeRxNotAssociated[node] + duration;
1462        // NS_LOG_UNCOND (to_string(node+1) + ",RX," + to_string(start.GetMicroSeconds()) + "
            ↪  " + to_string(duration.GetMicroSeconds()));
1463        break;
1464      case WifiPhy::State::CCA_BUSY: // CCA_BUSY
1465        timeCollisionNotAssociated[node] = timeCollisionNotAssociated[node] + duration;
1466        // NS_LOG_UNCOND (to_string(node+1) + ",CCA_BUSY," +
            ↪  to_string(start.GetMicroSeconds()) + " " +
            ↪  to_string(duration.GetMicroSeconds()));
1467        break;
1468      }
1469    }
1470  }
1471
1472
1473
1474
1475  template <int node>
1476  void RemainingEnergy(double oldValue, double newValue)
1477  {
1478    if (config.filesOutput)
1479    {
1480      std::stringstream ss;
1481      ss << "/media//remaining_energy_" << node << ".csv";
1482      static std::fstream f(ss.str().c_str(), std::ios::out);
1483      f << Simulator::Now().GetSeconds() << ", " << node << ", " << config.psFactor << ", "
          ↪  << oldValue - newValue << ", " << newValue << std::endl;
1484    }
1485  }
1486
1487  /// Trace function for total energy consumption at node.
1488  template <int node>
1489  void TotalEnergy(double oldValue, double newValue)
1490  {
1491    double dista;
1492    if (node<10)
1493    {
1494      dista = 1.0;
1495    }
```

```
  else if ((node>=10) && (node<20))
  {
    distа = 1.75;
  }
  else if ((node>=20) && (node<30))
  {
    distа = 2.5;
  }


  config.consump[node] = std::abs(newValue);
  if (config.filesOutput)
  {
    std::stringstream ss;
    ss << "/media/energy_consumption_" << node << ".csv";
    static std::fstream f(ss.str().c_str(), std::ios::out);
    f << Simulator::Now().GetSeconds() << ", " << node << ", " << distа << ", " <<
    ↪  config.psFactor << ", " << newValue - oldValue << ", " << newValue << std::endl;
  }
}

/// Trace function for the power harvested by the energy harvester.
template <int node>
void HarvestedPower(double oldValue, double newValue)
{
  if (config.filesOutput)
  {
    std::stringstream ss;
    static std::fstream f(ss.str().c_str(), std::ios::out);
    f << Simulator::Now().GetSeconds() << ", " << node << ", " << config.psFactor << ", "
    ↪  << newValue - oldValue << ", " << newValue << std::endl;
  }
  config.harvestedPowerCounter[node]++;
  config.g_harvestedPowerAvg[node] = std::abs(newValue);
}

/// Trace function for the total energy harvested by the node.
template <int node>
void TotalEnergyHarvested(double oldValue, double newValue)
{
  if (config.filesOutput)
  {
    std::stringstream ss;
    ss << "/media/total_energy_harvested_" << node << ".csv";
    static std::fstream f(ss.str().c_str(), std::ios::out);
    f << Simulator::Now().GetSeconds() << ", " << node << ", " << config.psFactor << ", "
    ↪  << newValue - oldValue << ", " << newValue << std::endl;
}
```

```
1541
1542    config.harvestedEnergyCounter[node]++;
1543    config.g_harvestedEnergyAvg[node] = std::abs(newValue);
1544    //config.g_harvestedEnergyAvg[node] = newValue;
1545  }
1546
1547  double nodePosition(int i)
1548  {
1549    if ((i>=0) && (i<10))
1550    {
1551      return 1.0;
1552    }
1553    else if ((i>=10) && (i<20))
1554    {
1555      return 1.75;
1556    }
1557    else if ((i>=20) && (i<30))
1558    {
1559      return 2.5;
1560    }
1561
1562  }
1563
1564  void OnPhyTxDrop2(std::string context, Ptr<const Packet> packet, DropReason reason)
1565  {
1566       WifiMacHeader hdr;
1567      packet->PeekHeader(hdr);
1568
1569    Mac48Address source = hdr.GetAddr2();
1570      Mac48Address destination = hdr.GetAddr1();
1571
1572
1573  }
1574
1575
1576
1577
1578
1579  template <int node>
1580  void PhyStateTraced(std::string context, Time start, Time duration, enum WifiPhy::State
     ↪  state)
1581  {
1582
1583      std::stringstream ss;
1584    ss << "/media/_" << node << ".csv";
1585    static std::fstream f(ss.str().c_str(), std::ios::out);
1586    f << Simulator::Now().GetSeconds() << ", " << node << ", " <<  config.psFactor << ", "
     ↪  << state << ", " << duration.GetSeconds() <<  std::endl;
```

```
1587
1588
1589  }
1590
1591  template <int node>
1592  void RxEndOk (Ptr<const Packet> packet)
1593  {
1594    WifiMacHeader hdr;
1595      packet->PeekHeader(hdr);
1596
1597    Mac48Address source = hdr.GetAddr2();
1598      Mac48Address destination = hdr.GetAddr1();
1599
1600
1601    onFramesRxAtNode[node]++;
1602
1603    if (node==config.Nsta)
1604    {
1605      for (int i = 0; i < config.Nsta; i++)
1606      {
1607        //if (source == macAdd[i])
1608        if (source == Mac48Address(macAdd[i]))
1609        {
1610          onFramesRxAtApFromNode[i]++;
1611          //cout << "Source ADDR at AP: " << source << " , No of Frames " <<
          ↪  onFramesRxAtApFromNode[i] << endl;
1612          if (hdr.IsRetry())
1613          {
1614            NumberOfRetriesRx[i]++;
1615          }
1616        }
1617      }
1618    }
1619
1620  }
1621
1622  template <int node>
1623  void TxEndOk (Ptr<const Packet> packet)
1624  {
1625    onFramesTx[node]++;
1626  }
1627
1628  template <int node>
1629  void MonitorRx(Ptr<const Packet> packet, uint16_t channelFreqMhz,
1630           uint16_t channelNumber, uint32_t rate, bool isShortPreamble,
1631           WifiTxVector txvector, double signalDbm, double noiseDbm)
1632  {
1633      WifiMacHeader hdr;
```

```
1634      packet->PeekHeader(hdr);
1635
1636    Mac48Address source = hdr.GetAddr2();
1637      Mac48Address destination = hdr.GetAddr1();
1638
1639    std::stringstream ss;
1640    ss << "/media//MonitorRx_" << node <<".csv";
1641    static std::fstream f(ss.str().c_str(), std::ios::out);
1642    f << Simulator::Now().GetMicroSeconds() << ", "
1643      //<< dist << ", "
1644      << source << ", "
1645      << destination << ", "
1646      //<< node << ", "
1647        << config.psFactor << ", "
1648      << packet->GetSize () << ", "
1649        << channelFreqMhz << ", "
1650      << channelNumber << ", "
1651        << rate << ", "
1652        << isShortPreamble << ", "
1653        << txvector << ", "
1654        << signalDbm << ", "
1655        << noiseDbm
1656      << std::endl;
1657
1658
1659      if (destination==macAdd[node])
1660      {
1661          onFramesRxMonitor[node]++;
1662        g_signalDbmAvg[node] += ((signalDbm - g_signalDbmAvg[node]) /
          ↪  onFramesRxMonitor[node]);
1663        g_noiseDbmAvg[node] += ((noiseDbm - g_noiseDbmAvg[node]) / onFramesRxMonitor[node]);
1664        m_rxSignalDbm[node] = signalDbm;
1665
1666
1667      }
1668  }
1669
1670  template <int node>
1671  void MonitorTx (Ptr<const Packet> packet, uint16_t channelFreqMhz,
1672          uint16_t channelNumber, uint32_t rate, bool isShortPreamble,
1673          WifiTxVector txvector)
1674  {
1675    WifiMacHeader hdr;
1676      packet->PeekHeader(hdr);
1677
1678    Mac48Address source = hdr.GetAddr2();
1679      Mac48Address destination = hdr.GetAddr1();
1680
```

```
1681   std::stringstream ss;
1682   ss << "/media//MonitorSniffTx_" << node << ".csv";
1683   static std::fstream f(ss.str().c_str(), std::ios::out);
1684   // f << std::fixed << std::setprecision(9) << Simulator::Now().GetSeconds() << ", " << i
       ↪  << ", " << sources.Get (i)->GetRemainingEnergy() << std::endl;
1685   f << Simulator::Now().GetMicroSeconds() << ", "
1686     << source << ", "
1687       << destination << ", "
1688     << packet << ", "
1689     << channelFreqMhz << ", "
1690     << channelNumber << ", "
1691     << rate << ", "
1692     << isShortPreamble << ", "
1693     << txvector << std::endl;
1694   // cout << txvector << std::endl;
1695   //onFramesTx[node]++;
1696 }
1697
1698 int main(int argc, char *argv[])
1699 {
1700   RngSeedManager::SetSeed(config.seed);
1701   RngSeedManager::SetRun (config.seedRun);
1702
1703   bool OutputPosition = true;
1704   config = Configuration(argc, argv);
1705
1706   config.rps = configureRAW(config.rps, config.RAWConfigFile);
1707   config.Nsta = config.NRawSta;
1708
1709   configurePageSlice();
1710   configureTIM();
1711   checkRawAndTimConfiguration ();
1712
1713   config.NSSFile = config.trafficType + "_" + std::to_string(config.Nsta) + "sta_" +
       ↪  std::to_string(ngroup) + "Group_" + std::to_string(nslot) + "slots_" +
       ↪  std::to_string(config.payloadSize) + "payload_" + std::to_string(config.totaltraffic)
       ↪  + "Mbps_" + std::to_string(config.BeaconInterval) + "BI" + ".nss";
1714
1715   stats = Statistics(config.Nsta);
1716   eventManager = SimulationEventManager(config.visualizerIP,
1717                       config.visualizerPort, config.NSSFile);
1718
1719   for (int i=0; i<=config.Nsta; i++)
1720   {
1721     config.harvestedPowerCounter[i] = 0;
1722     config.harvestedEnergyCounter[i] = 0;
1723     config.g_harvestedPowerAvg[i] = 0.0;
1724     config.g_harvestedEnergyAvg[i] = 0.0;
```

```
1725      config.NumberOfPacketsToApFromNode[i] = 0;
1726      config.consump[i] = 0.0;
1727      onFramesTx[i] = 0;
1728      onFramesRxAtNode[i] = 0;
1729      onFramesRxAtApFromNode[i] = 0;
1730      onFramesRxMonitor[i] = 0;
1731      g_signalDbmAvg[i] = 0.0;
1732      g_noiseDbmAvg[i] = 0.0;
1733      m_rxSignalDbm[i] = 0.0;
1734      NumberOfRetriesRx[i] = 0;
1735    }
1736
1737
1738    uint32_t totalRawGroups(0);
1739    for (int i = 0; i < config.rps.rpsset.size(); i++)
1740    {
1741      uint8_t nRaw = config.rps.rpsset[i]->GetNumberOfRawGroups();
1742      totalRawGroups += nRaw;
1743      cout << "Total raw groups after rps " << i << " is " << totalRawGroups << endl;
1744      for (int j = 0; j < nRaw; j++)
1745      {
1746        config.totalRawSlots += config.rps.rpsset[i]->GetRawAssigmentObj(j).GetSlotNum();
1747        cout << "Total slots after group " << j << " is " << config.totalRawSlots << endl;
1748      }
1749    }
1750    transmissionsPerTIMGroupAndSlotFromAPSinceLastInterval = vector<long>(
1751      config.totalRawSlots, 0);
1752    transmissionsPerTIMGroupAndSlotFromSTASinceLastInterval = vector<long>(
1753      config.totalRawSlots, 0);
1754
1755    //RngSeedManager::SetSeed(config.seed);
1756    //RngSeedManager::SetRun (config.seedRun);
1757
1758    wifiStaNode.Create(config.Nsta);
1759    wifiApNode.Create(1);
1760
1761
1762
1763     YansWifiChannelHelper channelBuilder = YansWifiChannelHelper();
1764
1765     channelBuilder.AddPropagationLoss("ns3::LogDistancePropagationLossModel",
1766        "Exponent", DoubleValue(3.76),
1767        "ReferenceLoss", DoubleValue(23.3),
1768        "ReferenceDistance", DoubleValue(1.00));
1769
1770     channelBuilder.SetPropagationDelay(
1771        "ns3::ConstantSpeedPropagationDelayModel");
1772
```

```
1773
1774
1775    Ptr<YansWifiChannel> channel = channelBuilder.Create();
1776    channel->TraceConnectWithoutContext("Transmission",
1777                    MakeCallback(&onChannelTransmission)); // TODO
1778
1779    YansWifiPhyHelper phy = YansWifiPhyHelper::Default();
1780    phy.SetErrorRateModel("ns3::YansErrorRateModel");
1781    phy.SetChannel(channel);
1782    phy.Set("Frequency", UintegerValue (config.frequency));
1783    phy.Set("ShortGuardEnabled", BooleanValue(false));
1784    phy.Set("ChannelWidth", UintegerValue(getBandwidth(config.DataMode))); // changed
1785    phy.Set("EnergyDetectionThreshold", DoubleValue(-104.0));
1786    phy.Set("CcaMode1Threshold", DoubleValue(-107.0));
1787    phy.Set("TxGain", DoubleValue(6.0));
1788    phy.Set("RxGain", DoubleValue(6.0));
1789    phy.Set("TxPowerLevels", UintegerValue(1));
1790    phy.Set("TxPowerEnd", DoubleValue(0.0));
1791    phy.Set("TxPowerStart", DoubleValue(0.0));
1792    phy.Set("RxNoiseFigure", DoubleValue(6.8));
1793    phy.Set("LdpcEnabled", BooleanValue(true));
1794    phy.Set("S1g1MfieldEnabled", BooleanValue(config.S1g1MfieldEnabled));
1795
1796    WifiHelper wifi = WifiHelper::Default();
1797    wifi.SetStandard(WIFI_PHY_STANDARD_80211ah);
1798    S1gWifiMacHelper mac = S1gWifiMacHelper::Default();
1799
1800    Ssid ssid = Ssid("ns380211ah");
1801    StringValue DataRate;
1802    DataRate = StringValue(getWifiMode(config.DataMode)); // changed
1803
1804    Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold",
1805              StringValue("220000"));
1806    Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",
1807              StringValue("220000"));
1808    // Fix non-unicast data rate to be the same as that of unicast
1809    Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode", DataRate);
1810
1811    wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
1812                 "DataMode", DataRate,
1813                 "ControlMode", DataRate);
1814    //wifi.SetRemoteStationManager("ns3::ArfWifiManager");
1815
1816    mac.SetType("ns3::StaWifiMac",
1817          "Ssid", SsidValue(ssid),
1818          "ActiveProbing", BooleanValue(false));
1819
1820
```

```
1821  NetDeviceContainer staDevice = wifi.Install(phy, mac, wifiStaNode);
1822
1823  mac.SetType("ns3::ApWifiMac",
1824        "Ssid", SsidValue(ssid),
1825        "BeaconInterval", TimeValue(MicroSeconds(config.BeaconInterval)),
1826        "NRawStations", UintegerValue(config.NRawSta),
1827        "RPSsetup", RPSVectorValue(config.rps),
1828        "PageSliceSet", pageSliceValue(config.pageS),
1829        "TIMSet", TIMValue(config.tim));
1830
1831  phy.Set("TxGain", DoubleValue(13.0));
1832  phy.Set("RxGain", DoubleValue(13.0));
1833  phy.Set("TxPowerLevels", UintegerValue(1));
1834  phy.Set("TxPowerEnd", DoubleValue(30.0));
1835  phy.Set("TxPowerStart", DoubleValue(30.0));
1836  phy.Set("RxNoiseFigure", DoubleValue(6.8));
1837  phy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);
1838
1839
1840  apDevice = wifi.Install(phy, mac, wifiApNode);
1841
1842
1843  Config::Set(
1844    "/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_EdcaTxopN/Queue/MaxPac
1845
1846    UintegerValue(10));
1847  Config::Set(
1848    "/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/BE_EdcaTxopN/Queue/MaxDel
1849    TimeValue(NanoSeconds(6000000000000)));
1850
1851
1852  Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/Txop/MinCw",
      ↪  UintegerValue(15));
1853  Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/Txop/MaxCw",
      ↪  UintegerValue(1023));
1854  Config::Set("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/Txop/Aifsn",
      ↪  UintegerValue(3));
1855
1856
1857  std::ostringstream oss;
1858  oss << "/NodeList/" << wifiApNode.Get(0)->GetId()
1859    << "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::ApWifiMac/";
1860  Config::ConnectWithoutContext(oss.str() + "RpsIndex", MakeCallback(&RpsIndexTrace));
1861  Config::ConnectWithoutContext(oss.str() + "RawGroup", MakeCallback(&RawGroupTrace));
1862  Config::ConnectWithoutContext(oss.str() + "RawSlot", MakeCallback(&RawSlotTrace));
1863  Config::ConnectWithoutContext(oss.str() + "PacketToApFromNode",
      ↪  MakeCallback(&PacketToApFromNode));
1864
```

```
1865  // mobility.
1866  MobilityHelper mobility;
1867  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>();
1868
1869  positionAlloc->Add(Vector(200.0, 200.0, 0.0));
1870
1871  double angle = 2.0 * M_PI / 10.0; // angle between nodes
1872  // double angle = 2.0 * M_PI / 5;  // angle between nodes
1873  Vector center(200.0, 200.0, 0.0); // center of the circle
1874
1875  Vector position;
1876
1877  for (int i = 0; i < config.Nsta; ++i)
1878  {
1879    // double radius;
1880    double shift;
1881
1882    config.dist = nodePosition(i);
1883    if ((config.dist == 1.0))
1884    {
1885      shift = angle / 3;
1886    }
1887    else if (config.dist == 1.75)
1888    {
1889      //angle = 2.0 * M_PI * config.dist/ 10.0;
1890      shift = angle * 2 / 3;
1891    }
1892    else if (config.dist == 2.5)
1893    {
1894      //angle = 2.0 * M_PI * config.dist/ 10.0;
1895      shift = angle * 3 / 7;
1896    }
1897    else if (config.dist == 5.0)
1898    {
1899      //angle = 2.0 * M_PI * config.dist/ 10.0;
1900      shift = angle * 4 / 7;
1901    }
1902    else if (config.dist == 6.0)
1903    {
1904      angle = 2.0 * M_PI * config.dist/ 10.0;
1905      shift = angle * 5 / 7;
1906    }
1907    else if (config.dist == 7.0)
1908    {
1909      angle = 2.0 * M_PI * config.dist/ 14.0;
1910      shift = angle * 6 / 7;
1911    }
1912
```

```
1913
1914      double x = center.x + nodePosition(i) * cos(i * angle + shift);
1915      double y = center.y + nodePosition(i) * sin(i * angle + shift);
1916
1917
1918      double z = center.z;
1919      Vector position(x, y, z);
1920      positionAlloc->Add(position);
1921    }
1922
1923
1924    mobility.SetPositionAllocator(positionAlloc);
1925    mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
1926    mobility.Install(wifiApNode);
1927    mobility.Install(wifiStaNode);
1928
1929    /*
1930
1931     MobilityHelper mobilityApCamera;
1932     Ptr<ListPositionAllocator> positionAllocAp = CreateObject<ListPositionAllocator> ();
1933     positionAllocAp->Add (Vector (xpos, ypos, 0.0));
1934     mobilityApCamera.SetPositionAllocator (positionAllocAp);
1935     mobilityApCamera.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
1936     mobilityApCamera.Install (wifiApNode);
1937
1938     float deltaAngle = 2* M_PI / (config.tcpipcameraEnd - config.tcpipcameraStart +1);
1939     float angle = 0.0;
1940     double x = 0.0;
1941     double y = 0.0;
1942
1943     double Distance = 50.0;
1944
1945
1946     Ptr<UniformRandomVariable> m_rv = CreateObject<UniformRandomVariable> ();
1947
1948
1949     for (int i = config.tcpipcameraStart; i <= config.tcpipcameraEnd; i++)
1950     {
1951     x = cos(angle) * Distance + xpos;
1952     y = sin(angle) * Distance + ypos;
1953
1954     MobilityHelper mobilityCamera;
1955     Ptr<ListPositionAllocator> positionAllocSta = CreateObject<ListPositionAllocator> ();
1956     positionAllocSta->Add(Vector(x, y, 0.0));
1957     mobilityCamera.SetPositionAllocator(positionAllocSta);
1958     mobilityCamera.SetMobilityModel("ns3::ConstantPositionMobilityModel");
1959     mobilityCamera.Install(wifiStaNode.Get(i));
1960     angle += deltaAngle;
```

```
1961    }
1962
1963    */
1964
1965    /* Internet stack*/
1966    InternetStackHelper stack;
1967    stack.Install(wifiApNode);
1968    stack.Install(wifiStaNode);
1969
1970    Ipv4AddressHelper address;
1971
1972    address.SetBase("192.168.0.0", "255.255.0.0");
1973
1974    staNodeInterface = address.Assign(staDevice);
1975    apNodeInterface = address.Assign(apDevice);
1976
1977    // trace association
1978    std::cout << "Configuring trace sources..." << std::endl;
1979    for (uint16_t kk = 0; kk < config.Nsta; kk++)
1980    {
1981      std::ostringstream STA;
1982      STA << kk;
1983      std::string strSTA = STA.str();
1984
1985      assoc_record *m_assocrecord = new assoc_record;
1986      m_assocrecord->setstaid(kk);
1987      Config::Connect(
1988        "/NodeList/" + strSTA +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/Assoc",
1989        MakeCallback(&assoc_record::SetAssoc, m_assocrecord));
1990      Config::Connect(
1991        "/NodeList/" + strSTA +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::RegularWifiMac/$ns3::StaWifiMac/DeAssoc",
1992        MakeCallback(&assoc_record::UnsetAssoc, m_assocrecord));
1993      assoc_vector.push_back(m_assocrecord);
1994    }
1995
1996    std::cout << "Populating routing tables..." << std::endl;
1997    Ipv4GlobalRoutingHelper::PopulateRoutingTables();
1998    std::cout << "Populating ARP cache..." << std::endl;
1999    PopulateArpCache();
2000
2001    // configure tracing for associations & other metrics
2002    std::cout << "Configuring trace sinks for nodes..." << std::endl;
2003    configureNodes(wifiStaNode, staDevice);
2004
2005    Config::Connect(
2006      "/NodeList/" + std::to_string(config.Nsta) +
       ↪  "/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxDropWithReason",
```

```
2007      MakeCallback(&OnAPPhyRxDrop));
2008    Config::Connect(
2009      "/NodeList/" + std::to_string(config.Nsta) +
          ↪  "/DeviceList/0/$ns3::WifiNetDevice/Mac/$ns3::ApWifiMac/PacketToTransmitReceivedFromUpperLaye
2010      MakeCallback(&OnAPPacketToTransmitReceived));
2011
2012    Ptr<MobilityModel> mobility1 =
2013      wifiApNode.Get(0)->GetObject<MobilityModel>();
2014    Vector apposition = mobility1->GetPosition();
2015    if (OutputPosition)
2016    {
2017      uint32_t i = 0;
2018      while (i < config.Nsta)
2019      {
2020        Ptr<MobilityModel> mobility = wifiStaNode.Get(i)->GetObject<MobilityModel>();
2021        Vector position = mobility->GetPosition();
2022        nodes[i]->x = position.x;
2023        nodes[i]->y = position.y;
2024        std::cout << "Sta node#" << i << ", "
2025              << "position = " << position
2026              << std::endl;
2027        dist[i] = mobility->GetDistanceFrom(
2028          wifiApNode.Get(0)->GetObject<MobilityModel>());
2029        i++;
2030      }
2031      std::cout << "AP node, position = " << apposition << std::endl;
2032    }
2033
2034    for (int i=0; i < config.Nsta; ++i)
2035    {
2036      Ptr<MobilityModel> mobility = wifiStaNode.Get(i)->GetObject<MobilityModel>();
2037      Vector position = mobility->GetPosition();
2038      //double x = position.x;
2039      //double y = position.y;
2040      //std::cout << "Sta node#" << i << ", " << "position = " << position
2041      //    << std::endl;
2042
2043          if (config.filesOutput)
2044          {
2045        std::stringstream ss;
2046        ss << "/media//Mobility.csv";
2047        static std::fstream f(ss.str().c_str(), std::ios::out);
2048        f << i << ", " << nodes[i]->aId << ", " << position.x << ", " << position.y << ", "
          ↪  <<position.z << endl;
2049          }
2050    }
2051
2052    /*Print of the state of the stations*/
```

```
2053  for (uint32_t i = 0; i < config.Nsta; i++)
2054  {
2055    std::ostringstream STA;
2056    STA << i;
2057    std::string strSTA = STA.str();
2058
2059    Config::Connect(
2060      "/NodeList/" + strSTA + "/DeviceList/*/Phy/$ns3::YansWifiPhy/State/State",
2061      MakeCallback(&PhyStateTrace));
2062  }
2063
2064  eventManager.onStartHeader();
2065  eventManager.onStart(config);
2066  if (config.rps.rpsset.size() > 0)
2067  {
2068    for (int i = 0; i < config.rps.rpsset.size(); i++)
2069    {
2070      for (int j = 0;
2071         j < config.rps.rpsset[i]->GetNumberOfRawGroups(); j++)
2072         {
2073          eventManager.onRawConfig(i, j,
2074                     config.rps.rpsset[i]->GetRawAssigmentObj(j));
2075         }
2076    }
2077  }
2078  for (uint32_t i = 0; i < config.Nsta; i++)
2079    eventManager.onSTANodeCreated(*nodes[i]);
2080
2081  eventManager.onAPNodeCreated(apposition.x, apposition.y);
2082  eventManager.onStatisticsHeader();
2083
2084  sendStatistics(true);
2085
2086  /** Energy Model **/
2087  /*************************************************************************/
2088  ///* energy source */
2089  LiIonEnergySourceHelper liIonEnergySourceHelper;
2090
2091  liIonEnergySourceHelper.Set("LiIonEnergySourceInitialEnergyJ", DoubleValue(100));
2092  liIonEnergySourceHelper.Set("LiIonEnergyLowBatteryThreshold", DoubleValue(0.2));
2093  liIonEnergySourceHelper.Set("LiIonEnergyHighBatteryThreshold", DoubleValue(0.7));
2094  liIonEnergySourceHelper.Set("InitialCellVoltage", DoubleValue(1.00));
2095  liIonEnergySourceHelper.Set("NominalCellVoltage", DoubleValue(1.00));
2096  liIonEnergySourceHelper.Set("ExpCellVoltage", DoubleValue(1.01));
2097  liIonEnergySourceHelper.Set("RatedCapacity", DoubleValue(1.00));
2098  liIonEnergySourceHelper.Set("NomCapacity", DoubleValue(1.01));
2099  liIonEnergySourceHelper.Set("ExpCapacity", DoubleValue(1.02));
2100  liIonEnergySourceHelper.Set("InternalResistance", DoubleValue(0.083));
```

```
liIonEnergySourceHelper.Set("TypCurrent", DoubleValue(2.33));
liIonEnergySourceHelper.Set("ThresholdVoltage", DoubleValue(0.3));
liIonEnergySourceHelper.Set("PeriodicEnergyUpdateInterval",
  TimeValue(Seconds(config.simulationTime + config.CoolDownPeriod + 2.0)));
// install source

        cout << "\n\n\n\nRUN UP TILL HERE node: \n\n\n\n" <<endl;
EnergySourceContainer sources;
for (int i = 0; i < config.Nsta; i++)
{
  sources.Add(liIonEnergySourceHelper.Install(wifiStaNode.Get(i)));
}

// Power consumption:
  https://www.asiarf.com/shop/halow-lora-iot/wi-fi-halow-sub-ghz-wireless-module-morse-micro-mm6
/* device energy model */
WifiRadioEnergyModelHelper radioEnergyHelper;

// configure radio energy model
if (config.datarate < 3900000)
{
  radioEnergyHelper.Set("TxCurrentA", DoubleValue(8.5e-3));
  radioEnergyHelper.Set("RxCurrentA", DoubleValue(6.0e-3));
}
else
{
  radioEnergyHelper.Set("TxCurrentA", DoubleValue(61.82e-3));
  radioEnergyHelper.Set("RxCurrentA", DoubleValue(27.0e-3));
}

radioEnergyHelper.Set("SleepCurrentA", DoubleValue(1.2e-6));
radioEnergyHelper.Set("IdleCurrentA", DoubleValue(6.0e-3));
radioEnergyHelper.Set("CcaBusyCurrentA", DoubleValue(6.0e-3));
radioEnergyHelper.Set("SwitchingCurrentA", DoubleValue(8.5e-3));



// install device model
DeviceEnergyModelContainer deviceModels;

for (int i = 0; i < config.Nsta; i++)
{
  deviceModels.Add(radioEnergyHelper.Install(staDevice.Get(i), sources.Get(i)));
}

/* energy harvester */
SwiptHarvesterHelper swiptHelper;
// configure energy harvester
```

```
2147  swiptHelper.Set("AntennaNoise", DoubleValue(-111.0));
2148  swiptHelper.Set("PowerSplitFactor", DoubleValue (config.psFactor));
2149  swiptHelper.Set("SwiptEfficiency", DoubleValue(0.9));
2150  swiptHelper.Set("DCConversionEfficiency", DoubleValue(0.95));
2151  // install harvester on all energy sources
2152
2153  EnergyHarvesterContainer harvesters;
2154
2155  for (int i = 0; i < config.Nsta; i++)
2156  {
2157    harvesters.Add(swiptHelper.Install(sources.Get(i)));
2158  }
2159
2160  for (uint32_t i = 0; i < config.Nsta; i++)
2161  {
2162    Ptr<WifiPhy> phyp =
      ↪  wifiStaNode.Get(i)->GetDevice(0)->GetObject<WifiNetDevice>()->GetPhy();
2163    Ptr<SwiptHarvester> swiptharvester = harvesters.Get(i)->GetObject<SwiptHarvester>();
2164
2165    swiptharvester->SetUpSwiptPhyListener(phyp);
2166    // std::cout << "\n\n=======\n phypp:" << phyp << std::endl;
2167
2168    Ptr<YansWifiPhy> yansPhy =
      ↪  wifiStaNode.Get(i)->GetDevice(0)->GetObject<WifiNetDevice>()->GetPhy()->GetObject<YansWifiPh
2169
2170    Ptr<YansWifiChannel> m_channel =
      ↪  wifiStaNode.Get(i)->GetDevice(0)->GetObject<WifiNetDevice>()->GetPhy()->GetChannel()->GetObj
2171    m_channel->AddSwiptPointer(swiptharvester);
2172    // std::cout << "\n\n=======\n wifiChannelPtr:" << m_channel << "\t" << swiptharvester
      ↪  << std::endl;
2173  }
2174
2175    sources.Get(0)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<0>));
2176    sources.Get(1)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<1>));
2177    sources.Get(2)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<2>));
2178    sources.Get(3)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<3>));
2179    sources.Get(4)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<4>));
2180    sources.Get(5)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<5>));
2181    sources.Get(6)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<6>));
2182    sources.Get(7)->TraceConnectWithoutContext("RemainingEnergy",
      ↪  MakeCallback(&RemainingEnergy<7>));
```

```
2183      sources.Get(8)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<8>));
2184      sources.Get(9)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<9>));
2185      sources.Get(10)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<10>));
2186      sources.Get(11)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<11>));
2187      sources.Get(12)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<12>));
2188      sources.Get(13)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<13>));
2189      sources.Get(14)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<14>));
2190      sources.Get(15)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<15>));
2191      sources.Get(16)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<16>));
2192      sources.Get(17)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<17>));
2193      sources.Get(18)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<18>));
2194      sources.Get(19)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<19>));
2195      sources.Get(20)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<20>));
2196      sources.Get(21)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<21>));
2197      sources.Get(22)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<22>));
2198      sources.Get(23)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<23>));
2199      sources.Get(24)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<24>));
2200      sources.Get(25)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<25>));
2201      sources.Get(26)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<26>));
2202      sources.Get(27)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<27>));
2203      sources.Get(28)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<28>));
2204      sources.Get(29)->TraceConnectWithoutContext("RemainingEnergy",
          ↪  MakeCallback(&RemainingEnergy<29>));
2205
2206
2207      deviceModels.Get(0)->TraceConnectWithoutContext("TotalEnergyConsumption",
          ↪  MakeCallback(&TotalEnergy<0>));
```

```
    deviceModels.Get(1)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<1>));
    deviceModels.Get(2)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<2>));
    deviceModels.Get(3)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<3>));
    deviceModels.Get(4)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<4>));
    deviceModels.Get(5)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<5>));
    deviceModels.Get(6)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<6>));
    deviceModels.Get(7)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<7>));
    deviceModels.Get(8)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<8>));
    deviceModels.Get(9)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<9>));
    deviceModels.Get(10)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<10>));
    deviceModels.Get(11)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<11>));
    deviceModels.Get(12)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<12>));
    deviceModels.Get(13)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<13>));
    deviceModels.Get(14)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<14>));
    deviceModels.Get(15)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<15>));
    deviceModels.Get(16)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<16>));
    deviceModels.Get(17)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<17>));
    deviceModels.Get(18)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<18>));
    deviceModels.Get(19)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<19>));
    deviceModels.Get(20)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<20>));
    deviceModels.Get(21)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<21>));
    deviceModels.Get(22)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<22>));
    deviceModels.Get(23)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<23>));
    deviceModels.Get(24)->TraceConnectWithoutContext("TotalEnergyConsumption",
    ↪ MakeCallback(&TotalEnergy<24>));
```

```
2232    deviceModels.Get(25)->TraceConnectWithoutContext("TotalEnergyConsumption",
        ↪  MakeCallback(&TotalEnergy<25>));
2233    deviceModels.Get(26)->TraceConnectWithoutContext("TotalEnergyConsumption",
        ↪  MakeCallback(&TotalEnergy<26>));
2234    deviceModels.Get(27)->TraceConnectWithoutContext("TotalEnergyConsumption",
        ↪  MakeCallback(&TotalEnergy<27>));
2235    deviceModels.Get(28)->TraceConnectWithoutContext("TotalEnergyConsumption",
        ↪  MakeCallback(&TotalEnergy<28>));
2236    deviceModels.Get(29)->TraceConnectWithoutContext("TotalEnergyConsumption",
        ↪  MakeCallback(&TotalEnergy<29>));
2237
2238    harvesters.Get(0)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<0>));
2239    harvesters.Get(1)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<1>));
2240    harvesters.Get(2)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<2>));
2241    harvesters.Get(3)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<3>));
2242    harvesters.Get(4)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<4>));
2243    harvesters.Get(5)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<5>));
2244    harvesters.Get(6)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<6>));
2245    harvesters.Get(7)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<7>));
2246    harvesters.Get(8)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<8>));
2247    harvesters.Get(9)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<9>));
2248    harvesters.Get(10)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<10>));
2249    harvesters.Get(11)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<11>));
2250    harvesters.Get(12)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<12>));
2251    harvesters.Get(13)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<13>));
2252    harvesters.Get(14)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<14>));
2253    harvesters.Get(15)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<15>));
2254    harvesters.Get(16)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<16>));
2255    harvesters.Get(17)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<17>));
2256    harvesters.Get(18)->TraceConnectWithoutContext("HarvestedPower",
        ↪  MakeCallback(&HarvestedPower<18>));
```

```
2257      harvesters.Get(19)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<19>));
2258      harvesters.Get(20)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<20>));
2259      harvesters.Get(21)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<21>));
2260      harvesters.Get(22)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<22>));
2261      harvesters.Get(23)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<23>));
2262      harvesters.Get(24)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<24>));
2263      harvesters.Get(25)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<25>));
2264      harvesters.Get(26)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<26>));
2265      harvesters.Get(27)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<27>));
2266      harvesters.Get(28)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<28>));
2267      harvesters.Get(29)->TraceConnectWithoutContext("HarvestedPower",
          ↪ MakeCallback(&HarvestedPower<29>));
2268
2269      harvesters.Get(0)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<0>));
2270      harvesters.Get(1)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<1>));
2271      harvesters.Get(2)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<2>));
2272      harvesters.Get(3)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<3>));
2273      harvesters.Get(4)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<4>));
2274      harvesters.Get(5)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<5>));
2275      harvesters.Get(6)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<6>));
2276      harvesters.Get(7)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<7>));
2277      harvesters.Get(8)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<8>));
2278      harvesters.Get(9)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<9>));
2279      harvesters.Get(10)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<10>));
2280      harvesters.Get(11)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<11>));
2281      harvesters.Get(12)->TraceConnectWithoutContext("TotalEnergyHarvested",
          ↪ MakeCallback(&TotalEnergyHarvested<12>));
```

```
2282    harvesters.Get(13)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<13>));
2283    harvesters.Get(14)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<14>));
2284    harvesters.Get(15)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<15>));
2285    harvesters.Get(16)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<16>));
2286    harvesters.Get(17)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<17>));
2287    harvesters.Get(18)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<18>));
2288    harvesters.Get(19)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<19>));
2289    harvesters.Get(20)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<20>));
2290    harvesters.Get(21)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<21>));
2291    harvesters.Get(22)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<22>));
2292    harvesters.Get(23)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<23>));
2293    harvesters.Get(24)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<24>));
2294    harvesters.Get(25)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<25>));
2295    harvesters.Get(26)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<26>));
2296    harvesters.Get(27)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<27>));
2297    harvesters.Get(28)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<28>));
2298    harvesters.Get(29)->TraceConnectWithoutContext("TotalEnergyHarvested",
        ↪  MakeCallback(&TotalEnergyHarvested<29>));
2299
2300
        ↪  Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd
        ↪  MakeCallback(&TxEndOk<0>));
2301    Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪  MakeCallback(&TxEndOk<1>));
2302    Config::ConnectWithoutContext("/NodeList/2/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪  MakeCallback(&TxEndOk<2>));
2303    Config::ConnectWithoutContext("/NodeList/3/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪  MakeCallback(&TxEndOk<3>));
2304    Config::ConnectWithoutContext("/NodeList/4/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪  MakeCallback(&TxEndOk<4>));
2305    Config::ConnectWithoutContext("/NodeList/5/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪  MakeCallback(&TxEndOk<5>));
```

```
2306    Config::ConnectWithoutContext("/NodeList/6/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<6>));
2307    Config::ConnectWithoutContext("/NodeList/7/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<7>));
2308    Config::ConnectWithoutContext("/NodeList/8/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<8>));
2309    Config::ConnectWithoutContext("/NodeList/9/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<9>));
2310    Config::ConnectWithoutContext("/NodeList/10/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<10>));
2311    Config::ConnectWithoutContext("/NodeList/11/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<11>));
2312    Config::ConnectWithoutContext("/NodeList/12/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<12>));
2313    Config::ConnectWithoutContext("/NodeList/13/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<13>));
2314    Config::ConnectWithoutContext("/NodeList/14/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<14>));
2315    Config::ConnectWithoutContext("/NodeList/15/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<15>));
2316
            ↪ Config::ConnectWithoutContext("/NodeList/16/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEn
            ↪ MakeCallback(&TxEndOk<16>));
2317    Config::ConnectWithoutContext("/NodeList/17/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<17>));
2318    Config::ConnectWithoutContext("/NodeList/18/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<18>));
2319    Config::ConnectWithoutContext("/NodeList/19/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<19>));
2320    Config::ConnectWithoutContext("/NodeList/20/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<20>));
2321    Config::ConnectWithoutContext("/NodeList/21/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<21>));
2322    Config::ConnectWithoutContext("/NodeList/22/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<22>));
2323    Config::ConnectWithoutContext("/NodeList/23/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<23>));
2324    Config::ConnectWithoutContext("/NodeList/24/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<24>));
2325    Config::ConnectWithoutContext("/NodeList/25/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<25>));
2326    Config::ConnectWithoutContext("/NodeList/26/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<26>));
2327    Config::ConnectWithoutContext("/NodeList/27/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<27>));
2328    Config::ConnectWithoutContext("/NodeList/28/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<28>));
2329    Config::ConnectWithoutContext("/NodeList/29/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<29>));
```

```
2330    //Config::ConnectWithoutContext("/NodeList/30/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyTxEnd",
        ↪ MakeCallback(&TxEndOk<30>));
2331
2332    Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<0>));
2333    Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<1>));
2334    Config::ConnectWithoutContext("/NodeList/2/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<2>));
2335    Config::ConnectWithoutContext("/NodeList/3/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<3>));
2336    Config::ConnectWithoutContext("/NodeList/4/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<4>));
2337    Config::ConnectWithoutContext("/NodeList/5/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<5>));
2338    Config::ConnectWithoutContext("/NodeList/6/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<6>));
2339    Config::ConnectWithoutContext("/NodeList/7/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<7>));
2340    Config::ConnectWithoutContext("/NodeList/8/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<8>));
2341    Config::ConnectWithoutContext("/NodeList/9/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<9>));
2342    Config::ConnectWithoutContext("/NodeList/10/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<10>));
2343    Config::ConnectWithoutContext("/NodeList/11/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<11>));
2344    Config::ConnectWithoutContext("/NodeList/12/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<12>));
2345    Config::ConnectWithoutContext("/NodeList/13/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<13>));
2346    Config::ConnectWithoutContext("/NodeList/14/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<14>));
2347    Config::ConnectWithoutContext("/NodeList/15/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<15>));
2348    Config::ConnectWithoutContext("/NodeList/16/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<16>));
2349    Config::ConnectWithoutContext("/NodeList/17/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<17>));
2350    Config::ConnectWithoutContext("/NodeList/18/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<18>));
2351    Config::ConnectWithoutContext("/NodeList/19/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<19>));
2352    Config::ConnectWithoutContext("/NodeList/20/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<20>));
2353    Config::ConnectWithoutContext("/NodeList/21/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<21>));
2354    Config::ConnectWithoutContext("/NodeList/22/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<22>));
```

```
2355    Config::ConnectWithoutContext("/NodeList/23/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<23>));
2356    Config::ConnectWithoutContext("/NodeList/24/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<24>));
2357    Config::ConnectWithoutContext("/NodeList/25/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<25>));
2358    Config::ConnectWithoutContext("/NodeList/26/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<26>));
2359    Config::ConnectWithoutContext("/NodeList/27/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<27>));
2360    Config::ConnectWithoutContext("/NodeList/28/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<28>));
2361    Config::ConnectWithoutContext("/NodeList/29/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<29>));
2362    Config::ConnectWithoutContext("/NodeList/30/DeviceList/0/$ns3::WifiNetDevice/Phy/PhyRxEnd",
        ↪ MakeCallback(&RxEndOk<30>));
2363
2364    Config::ConnectWithoutContext("/NodeList/0/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorSnifferRx
        ↪ MakeCallback(&MonitorRx<0>));
2365
        ↪ Config::ConnectWithoutContext("/NodeList/1/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<1>));
2366
        ↪ Config::ConnectWithoutContext("/NodeList/2/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<2>));
2367
        ↪ Config::ConnectWithoutContext("/NodeList/3/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<3>));
2368
        ↪ Config::ConnectWithoutContext("/NodeList/4/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<4>));
2369
        ↪ Config::ConnectWithoutContext("/NodeList/5/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<5>));
2370
        ↪ Config::ConnectWithoutContext("/NodeList/6/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<6>));
2371
        ↪ Config::ConnectWithoutContext("/NodeList/7/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<7>));
2372
        ↪ Config::ConnectWithoutContext("/NodeList/8/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<8>));
2373
        ↪ Config::ConnectWithoutContext("/NodeList/9/DeviceList/0/$ns3::WifiNetDevice/Phy/MonitorS
        ↪ MakeCallback(&MonitorRx<9>));
2374
        ↪ Config::ConnectWithoutContext("/NodeList/10/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<10>));
```

```
2375
        ↪ Config::ConnectWithoutContext("/NodeList/11/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<11>));
2376
        ↪ Config::ConnectWithoutContext("/NodeList/12/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<12>));
2377
        ↪ Config::ConnectWithoutContext("/NodeList/13/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<13>));
2378
        ↪ Config::ConnectWithoutContext("/NodeList/14/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<14>));
2379
        ↪ Config::ConnectWithoutContext("/NodeList/15/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<15>));
2380
        ↪ Config::ConnectWithoutContext("/NodeList/16/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<16>));
2381
        ↪ Config::ConnectWithoutContext("/NodeList/17/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<17>));
2382
        ↪ Config::ConnectWithoutContext("/NodeList/18/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<18>));
2383
        ↪ Config::ConnectWithoutContext("/NodeList/19/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<19>));
2384
        ↪ Config::ConnectWithoutContext("/NodeList/20/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<20>));
2385
        ↪ Config::ConnectWithoutContext("/NodeList/21/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<21>));
2386
        ↪ Config::ConnectWithoutContext("/NodeList/22/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<22>));
2387
        ↪ Config::ConnectWithoutContext("/NodeList/23/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<23>));
2388
        ↪ Config::ConnectWithoutContext("/NodeList/24/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<24>));
2389
        ↪ Config::ConnectWithoutContext("/NodeList/25/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<25>));
2390
        ↪ Config::ConnectWithoutContext("/NodeList/26/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<26>));
```

```
2391
        ↪ Config::ConnectWithoutContext("/NodeList/27/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<27>));
2392
        ↪ Config::ConnectWithoutContext("/NodeList/28/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<28>));
2393
        ↪ Config::ConnectWithoutContext("/NodeList/29/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<29>));
2394
        ↪ Config::ConnectWithoutContext("/NodeList/30/DeviceList/0/$ns3::WifiNetDevice/Phy/Monitor
        ↪ MakeCallback(&MonitorRx<30>));
2395
2396  Simulator::Stop(Seconds(config.simulationTime + config.CoolDownPeriod)); // allow up to a
      ↪ minute after the client & server apps are finished to process the queue
2397  Simulator::Run();
2398
2399  // Visualizer throughput
2400  int pay = 0, totalSuccessfulPackets = 0, totalSentPackets = 0, totalPacketsEchoed = 0;
2401  for (int i = 0; i < config.Nsta; i++)
2402  {
2403    totalSuccessfulPackets += stats.get(i).NumberOfSuccessfulPackets;
2404    totalSentPackets += stats.get(i).NumberOfSentPackets;
2405    totalPacketsEchoed += stats.get(i).NumberOfSuccessfulRoundtripPackets;
2406    pay += stats.get(i).TotalPacketPayloadSize;
2407    cout << i << " sent: " << stats.get(i).NumberOfSentPackets
2408       << " ; delivered: " << stats.get(i).NumberOfSuccessfulPackets
2409       << " ; echoed: " << stats.get(i).NumberOfSuccessfulRoundtripPackets
2410       << "; packetloss: "
2411       << stats.get(i).GetPacketLoss(config.trafficType) << endl;
2412  }
2413
2414  double throughput = 0;
2415  //uint32_t totalPacketsThrough = 0;
2416  if (config.trafficType == "udp")
2417  {
2418    throughput = totalSuccessfulPackets * config.payloadSize * 8./(config.simulationTime +
      ↪ config.CoolDownPeriod - 3);
2419    uint32_t totalPacketsThrough =
2420        DynamicCast<UdpServer>(serverApp.Get(0))->GetReceived();
2421    cout << "totalPacketsThrough " << totalPacketsThrough << " ++my "
2422       << totalSuccessfulPackets << endl;
2423    cout << "throughput " << throughput << " ++my "
2424       << pay * 8. / (config.simulationTime + config.CoolDownPeriod - 3) << endl;
2425    std::cout << "datarate"
2426          << "\t"
2427          << "throughput" << std::endl;
2428    std::cout << config.datarate << "bps\t\t" << throughput << "bps" << std::endl;
```

```
2429  }
2430  else if (config.trafficType == "udpecho")
2431  {
2432    double ulThroughput = 0, dlThroughput = 0;
2433    ulThroughput = totalSuccessfulPackets * config.payloadSize * 8 / (config.simulationTime
      ↪  * 1000000.0);
2434    dlThroughput = totalPacketsEchoed * config.payloadSize * 8 / (config.simulationTime *
      ↪  1000000.0);
2435    cout << "totalPacketsSent " << totalSentPackets << endl;
2436    cout << "totalPacketsDelivered " << totalSuccessfulPackets << endl;
2437    cout << "totalPacketsEchoed " << totalPacketsEchoed << endl;
2438    cout << "UL packets lost " << totalSentPackets - totalSuccessfulPackets << endl;
2439    cout << "DL packets lost " << totalSuccessfulPackets - totalPacketsEchoed << endl;
2440    cout << "Total packets lost " << totalSentPackets - totalPacketsEchoed << endl;
2441
2442    double throughput = (totalSuccessfulPackets + totalPacketsEchoed) * config.payloadSize
      ↪  * 8 / (config.simulationTime * 1000000.0);
2443    cout << "total throughput Kbit/s " << throughput * 1000 << endl;
2444
2445    std::cout << "datarate"
2446          << "\t"
2447          << "throughput" << std::endl;
2448    std::cout << config.datarate << "\t" << throughput * 1000 << " Kbit/s" << std::endl;
2449  }
2450  cout << "total packet loss % "
2451     << 100 - 100. * totalSuccessfulPackets / totalSentPackets << endl;
2452
2453
2454
2455    if (config.filesOutput)
2456  {
2457    for (int i = 0; i < config.Nsta; i++)
2458        {
2459        std::stringstream ss;
2460        ss << "/media/Summary.csv";
2461        static std::fstream f(ss.str().c_str(), std::ios::out);
2462        f << i << ", "
2463      << nodes[i]->aId << ", "
2464      << nodes[i]->isAssociated << ", "
2465      << (int)nodes[i]->rawGroupNumber << ", "
2466      << (int)nodes[i]->rawSlotIndex << ", "
2467      << nodes[i]->rpsIndex << ", "
2468        << std::fixed << std::setprecision(0)
2469            << config.datarate << ", "
2470
2471        << std::fixed << std::setprecision(2)
2472        << nodePosition (i) << ", "
2473
```

```
2474        //Throughput:
2475          << std::fixed << std::setprecision(2)
2476
2477        << throughput << ", "
2478          << (stats.get(i).NumberOfSuccessfulPackets +
              ↪  stats.get(i).NumberOfSuccessfulRoundtripPackets) * config.payloadSize * 8.0/
              ↪  ((config.simulationTime + config.CoolDownPeriod - 3.0) * 1.0) << ", "
              ↪  //throughput
2479
2480        //UDP:
2481          << std::fixed << std::setprecision(0)
2482          << stats.get(i).NumberOfSentPackets << ", "
2483          << stats.get(i).NumberOfSuccessfulPackets << ", "
2484          << stats.get(i).TotalPacketSentReceiveTime << ", "  //% Loss
2485          << stats.get(i).latency << ", "  //% Loss
2486          << stats.get(i).jitter << ", "  //% Loss
2487          << stats.get(i).jitterAcc << ", "  //% Loss
2488          << stats.get(i).getNumberOfDroppedPackets() << ", "  //% Loss
2489          << stats.get(i).GetPacketLoss(config.trafficType) << ", "  //% Loss
2490
2491        //Frames TX:
2492        //<< stats.get(i).NumberOfSuccessfullTx << ", "
2493        << onFramesTx[i]  << ", "
2494        //Frames RX @ AP:
2495        //<< config.NumberOfPacketsToApFromNode[i] << ", "
2496        << onFramesRxAtApFromNode[i] << ", "
2497        //Frames RX @ node:
2498        << onFramesRxAtNode[i] << ", "
2499        //Frame Loss %
2500          << 100 - 100. * onFramesRxAtApFromNode[i] / onFramesTx[i] << ", " //
2501        << stats.get(i).NumberOfTransmissionsDropped << ", "
2502        << stats.get(i).NumberOfReceivesDropped << ", "
2503        << stats.get(i).NumberOfReceiveDroppedByDestination << ", "
2504
2505        //psFactor
2506            << std::setprecision(10) << std::scientific
2507          << config.psFactor << ", "
2508
2509          << config.g_harvestedPowerAvg[i] / config.harvestedPowerCounter[i] << ", "
2510          << config.harvestedPowerCounter[i] << ", "
2511          << config.g_harvestedPowerAvg[i] << ", "
2512          << config.g_harvestedEnergyAvg[i] / config.harvestedEnergyCounter[i] << ", "
2513          << config.harvestedEnergyCounter[i] << ", "
2514          << config.g_harvestedEnergyAvg[i] << ", "
2515          << config.consump[i] << ", "
2516        << stats.get(i).NumberOfCollisions << ", "
2517        << stats.get(i).TotalNumberOfBackedOffSlots << ", "
2518        << stats.get(i).NumberOfMACTxMissedACK << ", "
```

```
2519        << stats.get(i).NumberOfMACTxMissedACKAndDroppedPacket << ", "
2520        << stats.get(i).NumberOfRetriesTx << ", "
2521        << stats.get(i).NumberOfRetriesDroppedTx << ", "
2522        << NumberOfRetriesRx[i] << ", "
2523        << stats.get(i).NumberOfRetriesDroppedRx << ", "
2524        << stats.get(i).NumberOfMACTxRTSFailed << ", "
2525        << stats.get(i).NumberOfTransmissionsCancelledDueToCrossingRAWBoundary << ", "
2526        << stats.get(i).NumberOfBeaconsMissed << ", "
2527
2528        << std::fixed << std::setprecision(2)
2529        << stats.TimeWhenEverySTAIsAssociated << ", "
2530
2531          << std::fixed << std::setprecision(2)
2532          << g_signalDbmAvg[i] << ", "
2533          << g_noiseDbmAvg[i] << ", "
2534          << (g_signalDbmAvg[i] - g_noiseDbmAvg[i])
2535              << std::endl;
2536          }
2537    }
2538
2539
2540
2541    Simulator::Destroy();
2542
2543    ofstream risultati;
2544    string addressresults = config.OutputPath + "moreinfo.txt";
2545    risultati.open(addressresults.c_str(), ios::out | ios::trunc);
2546
2547    risultati << "Sta
      ↪  node#,distance,timerx(notassociated),timeidle(notassociated),timetx(notassociated),timesleep(n
      ↪  << std::endl;
2548    //risultati << "Sta node#,distance,timerx,timeidle,timetx,timesleep,timecollision" <<
      ↪  std::endl;
2549    int i = 0;
2550    string spazio = ",";
2551
2552    while (i < config.Nsta)
2553    {
2554
2555      risultati << i << spazio << dist[i] << spazio << timeRxArray[i].GetSeconds() << ",(" <<
        ↪  timeRxNotAssociated[i].GetSeconds() << ")," << timeIdleArray[i].GetSeconds() <<
        ↪  ",(" << timeIdleNotAssociated[i].GetSeconds() << ")," <<
        ↪  timeTxArray[i].GetSeconds() << ",(" << timeTxNotAssociated[i].GetSeconds() << ")"
        ↪  << timeSleepArray[i].GetSeconds() << ",(" << timeSleepNotAssociated[i].GetSeconds()
        ↪  << ")," << timeCollisionArray[i].GetSeconds() << ",(" <<
        ↪  timeCollisionNotAssociated[i].GetSeconds() << ")" << std::endl;
2556
2557      cout << "================= Sleep time: " << stats.get(i).TotalSleepTime.GetSeconds()
        ↪  << endl;
```

```
2558      cout << "================== Tx time: " << stats.get(i).TotalTxTime.GetSeconds() <<
          ↪  endl;
2559      cout << "================== Rx time: " << stats.get(i).TotalRxTime.GetSeconds() <<
          ↪  endl;
2560      cout << "++++++++++++++++++IDLE time: " << stats.get(i).TotalIdleTime.GetSeconds() <<
          ↪  endl;
2561      //cout << "ooooooooooooooooooo TOTENERGY " << stats.get(i).TotalSleepTime.GetSeconds()
2562      //                    +  stats.get(i).TotalTxTime.GetSeconds()
2563      //                    +  stats.get(i).TotalRxTime.GetSeconds()
2564      //                    +  stats.get(i).TotalIdleTime.GetSeconds()
2565      //                     << " J" << endl;
2566      cout << "ooooooooooooooooooo TOTENERGY " << stats.get(i).GetTotalEnergyConsumption() <<
          ↪  " J" << endl;
2567      //cout << "Rx+Idle ENERGY " << stats.get(i).EnergyRxIdle << " mW" << endl;
2568      //cout << "Tx ENERGY " << stats.get(i).EnergyTx << " mW" << endl;
2569
2570      i++;
2571    }
2572
2573    risultati.close();
2574    return 0;
2575  }
```

APPENDIX D

YANSWIFICHANNEL

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2006,2007 INRIA
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Mathieu Lacage, <mathieu.lacage@sophia.inria.fr>
 */
#include "ns3/packet.h"
#include "ns3/simulator.h"
#include "ns3/mobility-model.h"
#include "ns3/net-device.h"
#include "ns3/node.h"
#include "ns3/log.h"
#include "ns3/pointer.h"
#include "ns3/object-factory.h"
#include "yans-wifi-channel.h"
#include "yans-wifi-phy.h"
#include "ns3/propagation-loss-model.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/swipt-harvester.h"


namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("YansWifiChannel");

```

```
39 NS_OBJECT_ENSURE_REGISTERED (YansWifiChannel);
40
41 TypeId
42 YansWifiChannel::GetTypeId (void)
43 {
44   static TypeId tid = TypeId ("ns3::YansWifiChannel")
45     .SetParent<WifiChannel> ()
46     .SetGroupName ("Wifi")
47     .AddConstructor<YansWifiChannel> ()
48     .AddAttribute ("PropagationLossModel", "A pointer to the propagation loss model
       ↪  attached to this channel.",
49                    PointerValue (),
50                    MakePointerAccessor (&YansWifiChannel::m_loss),
51                    MakePointerChecker<PropagationLossModel> ())
52     .AddAttribute ("PropagationDelayModel", "A pointer to the propagation delay model
       ↪  attached to this channel.",
53                    PointerValue (),
54                    MakePointerAccessor (&YansWifiChannel::m_delay),
55                    MakePointerChecker<PropagationDelayModel> ())
56   ;
57   return tid;
58 }
59
60 YansWifiChannel::YansWifiChannel ()
61 {
62 }
63 YansWifiChannel::~YansWifiChannel ()
64 {
65   NS_LOG_FUNCTION_NOARGS ();
66   m_phyList.clear ();
67   m_swiptList.clear ();
68 }
69
70 void
71 YansWifiChannel::SetPropagationLossModel (Ptr<PropagationLossModel> loss)
72 {
73   m_loss = loss;
74 }
75
76 void
77 YansWifiChannel::SetPropagationDelayModel (Ptr<PropagationDelayModel> delay)
78 {
79   m_delay = delay;
80 }
81
82 void
83 YansWifiChannel::Send (Ptr<YansWifiPhy> sender, Ptr<const Packet> packet, double
   ↪  txPowerDbm,
```

```
84                        WifiTxVector txVector, WifiPreamble preamble, uint8_t packetType,
                          ↪  Time duration) const
85  {
86    Ptr<MobilityModel> senderMobility = sender->GetMobility ()->GetObject<MobilityModel> ();
87    NS_ASSERT (senderMobility != 0);
88    uint32_t j = 0;
89    for (PhyList::const_iterator i = m_phyList.begin (); i != m_phyList.end (); i++, j++)
90      {
91        if (sender != (*i))
92          {
93            // For now don't account for inter channel interference
94            if ((*i)->GetChannelNumber () != sender->GetChannelNumber ())
95              {
96                continue;
97              }
98
99            Ptr<MobilityModel> receiverMobility = (*i)->GetMobility
              ↪  ()->GetObject<MobilityModel> ();
100           Time delay = m_delay->GetDelay (senderMobility, receiverMobility);
101           double rxPowerDbm = m_loss->CalcRxPower (txPowerDbm, senderMobility,
              ↪  receiverMobility);
102           double lossDbm = txPowerDbm - rxPowerDbm;
103
104           NS_LOG_DEBUG ("propagation: txPower=" << txPowerDbm
105                         << "dbm, rxPower=" << rxPowerDbm
106                         << "dbm, distance=" << senderMobility->GetDistanceFrom
                            ↪  (receiverMobility)
107                         << "m, delay=" << delay
108                         << "s, loss=" << lossDbm);
109
110
111           Ptr<Packet> copy = packet->Copy ();
112           Ptr<Object> dstNetDevice = m_phyList[j]->GetDevice ();
113           uint32_t dstNode;
114           if (dstNetDevice == 0)
115             {
116               dstNode = 0xffffffff;
117             }
118           else
119             {
120               dstNode = dstNetDevice->GetObject<NetDevice> ()->GetNode ()->GetId ();
121             }
122
123           double *atts = new double[3];
124           *atts = rxPowerDbm;
125           *(atts+1)= packetType;
126           *(atts+2)= duration.GetNanoSeconds();
127
```

```
128           Simulator::ScheduleWithContext (dstNode,
129                                           delay, &YansWifiChannel::Receive, this,
130                                           j, copy, atts, txVector, preamble);
131         }
132     }
133 }
134 // Added by JoseF in order to correlate the indexes of the SwiptHarvester and the
    ↪  respective YansWifiPhy
135 void
136 YansWifiChannel::AddSwiptPointer (Ptr<SwiptHarvester> swiptPointer)
137 {
138   m_swiptList.push_back (swiptPointer);
139 }
140
141 void
142 YansWifiChannel::Receive (uint32_t i, Ptr<Packet> packet, double *atts,
143                           WifiTxVector txVector, WifiPreamble preamble) const
144 {
145   Ptr<YansWifiPhy> wifiPhyPtr = m_phyList[i];
146   uint32_t idNode = wifiPhyPtr->GetDevice ()->GetObject<WifiNetDevice> ()->GetNode ()->
    ↪  GetId ();
147
148   double psFactor;
149   // In the configuration/simulation file the sta nodes must be created before the AP node
150   if (GetNDevices () - 1 != idNode)
151   {
152     m_swiptList[i]->PowerWPktReceived (packet, *atts, txVector, preamble,*(atts+1),
      ↪  NanoSeconds(*(atts+2)), wifiPhyPtr);
153     psFactor = m_swiptList[i]->GetPowerSplitFactor ();
154     m_phyList[i]->StartReceivePreambleAndHeader (packet, *atts, txVector,
      ↪  preamble,*(atts+1), NanoSeconds(*(atts+2)), psFactor, m_swiptList[i]);
155
156     NS_LOG_DEBUG (Simulator::Now().GetNanoSeconds () << "ns, YansWifiChannel N_Devices:" <<
      ↪  GetNDevices () - 1 << " , idNode:" << idNode);
157
158   }
159   else //This part applies only to the AP due to the impossibility of having SwiptHarvester
    ↪  installed
160   {
161     psFactor = 1.0;
162     Ptr<SwiptHarvester> sh = NULL;
163     m_phyList[i]->StartReceivePreambleAndHeader (packet, *atts, txVector,
      ↪  preamble,*(atts+1), NanoSeconds(*(atts+2)), psFactor, sh);
164   }
165   delete[] atts;
166 }
167
168 uint32_t
```

```
169 YansWifiChannel::GetNDevices (void) const
170 {
171   return m_phyList.size ();
172 }
173 Ptr<NetDevice>
174 YansWifiChannel::GetDevice (uint32_t i) const
175 {
176   return m_phyList[i]->GetDevice ()->GetObject<NetDevice> ();
177 }
178
179 void
180 YansWifiChannel::Add (Ptr<YansWifiPhy> phy)
181 {
182   m_phyList.push_back (phy);
183 }
184
185 int64_t
186 YansWifiChannel::AssignStreams (int64_t stream)
187 {
188   int64_t currentStream = stream;
189   currentStream += m_loss->AssignStreams (stream);
190   return (currentStream - stream);
191 }
192
193 } // namespace ns3
```

APPENDIX E

YANSWIFIPHY

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2005,2006 INRIA
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Authors: Mathieu Lacage <mathieu.lacage@sophia.inria.fr>
 *          Ghada Badawy <gbadawy@gmail.com>
 */

#include "yans-wifi-phy.h"
#include "yans-wifi-channel.h"
#include "wifi-mode.h"
#include "wifi-preamble.h"
#include "wifi-phy-state-helper.h"
#include "error-rate-model.h"
#include "ns3/simulator.h"
#include "ns3/packet.h"
#include "ns3/assert.h"
#include "ns3/log.h"
#include "ns3/double.h"
#include "ns3/uinteger.h"
#include "ns3/enum.h"
#include "ns3/pointer.h"
#include "ns3/net-device.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/boolean.h"
```

```
39 #include "ns3/node.h"
40 #include "ns3/node.h"
41 #include "ns3/swipt-harvester.h"
42 #include <cmath>
43 #include <iomanip>
44 #include <iostream>
45 #include <fstream>
46
47
48 namespace ns3 {
49
50 NS_LOG_COMPONENT_DEFINE ("YansWifiPhy");
51
52 NS_OBJECT_ENSURE_REGISTERED (YansWifiPhy);
53
54 TypeId
55 YansWifiPhy::GetTypeId (void)
56 {
57   static TypeId tid = TypeId ("ns3::YansWifiPhy")
58     .SetParent<WifiPhy> ()
59     .SetGroupName ("Wifi")
60     .AddConstructor<YansWifiPhy> ()
61     .AddAttribute ("EnergyDetectionThreshold",
62                    "The energy of a received signal should be higher than "
63                    "this threshold (dbm) to allow the PHY layer to detect the signal.",
64                    DoubleValue (-96.0),
65                    MakeDoubleAccessor (&YansWifiPhy::SetEdThreshold,
66                                        &YansWifiPhy::GetEdThreshold),
67                    MakeDoubleChecker<double> ())
68     .AddAttribute ("CcaMode1Threshold",
69                    "The energy of a received signal should be higher than "
70                    "this threshold (dbm) to allow the PHY layer to declare CCA BUSY
                       ↪  state.",
71                    DoubleValue (-99.0),
72                    MakeDoubleAccessor (&YansWifiPhy::SetCcaMode1Threshold,
73                                        &YansWifiPhy::GetCcaMode1Threshold),
74                    MakeDoubleChecker<double> ())
75     .AddAttribute ("TxGain",
76                    "Transmission gain (dB).",
77                    DoubleValue (1.0),
78                    MakeDoubleAccessor (&YansWifiPhy::SetTxGain,
79                                        &YansWifiPhy::GetTxGain),
80                    MakeDoubleChecker<double> ())
81     .AddAttribute ("RxGain",
82                    "Reception gain (dB).",
83                    DoubleValue (1.0),
84                    MakeDoubleAccessor (&YansWifiPhy::SetRxGain,
85                                        &YansWifiPhy::GetRxGain),
```

```
86                    MakeDoubleChecker<double> ())
87      .AddAttribute ("TxPowerLevels",
88                     "Number of transmission power levels available between "
89                     "TxPowerStart and TxPowerEnd included.",
90                     UintegerValue (1),
91                     MakeUintegerAccessor (&YansWifiPhy::m_nTxPower),
92                     MakeUintegerChecker<uint32_t> ())
93      .AddAttribute ("TxPowerEnd",
94                     "Maximum available transmission level (dbm).",
95                     DoubleValue (16.0206),
96                     MakeDoubleAccessor (&YansWifiPhy::SetTxPowerEnd,
97                                         &YansWifiPhy::GetTxPowerEnd),
98                     MakeDoubleChecker<double> ())
99      .AddAttribute ("TxPowerStart",
100                    "Minimum available transmission level (dbm).",
101                    DoubleValue (16.0206),
102                    MakeDoubleAccessor (&YansWifiPhy::SetTxPowerStart,
103                                        &YansWifiPhy::GetTxPowerStart),
104                    MakeDoubleChecker<double> ())
105     .AddAttribute ("RxNoiseFigure",
106                    "Loss (dB) in the Signal-to-Noise-Ratio due to non-idealities in the
                       ↪  receiver."
107                    " According to Wikipedia (http://en.wikipedia.org/wiki/Noise_figure),
                       ↪  this is "
108                    "\"the difference in decibels (dB) between"
109                    " the noise output of the actual receiver to the noise output of an "
110                    " ideal receiver with the same overall gain and bandwidth when the
                       ↪  receivers "
111                    " are connected to sources at the standard noise temperature T0 (usually
                       ↪  290 K)\".",
112                    DoubleValue (7),
113                    MakeDoubleAccessor (&YansWifiPhy::SetRxNoiseFigure,
114                                        &YansWifiPhy::GetRxNoiseFigure),
115                    MakeDoubleChecker<double> ())
116     .AddAttribute ("State",
117                    "The state of the PHY layer.",
118                    PointerValue (),
119                    MakePointerAccessor (&YansWifiPhy::m_state),
120                    MakePointerChecker<WifiPhyStateHelper> ())
121     .AddAttribute ("ChannelSwitchDelay",
122                    "Delay between two short frames transmitted on different frequencies.",
123                    TimeValue (MicroSeconds (250)),
124                    MakeTimeAccessor (&YansWifiPhy::m_channelSwitchDelay),
125                    MakeTimeChecker ())
126     .AddAttribute ("ChannelNumber",
127                    "Channel center frequency = Channel starting frequency + 5 MHz * nch.",
128                    UintegerValue (1),
129                    MakeUintegerAccessor (&YansWifiPhy::SetChannelNumber,
```

```
                                    &YansWifiPhy::GetChannelNumber),
                    MakeUintegerChecker<uint16_t> ())
    .AddAttribute ("Frequency",
                    "The operating frequency.",
                    UintegerValue (2407),
                    MakeUintegerAccessor (&YansWifiPhy::GetFrequency,
                                          &YansWifiPhy::SetFrequency),
                    MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("Transmitters",
                    "The number of transmitters.",
                    UintegerValue (1),
                    MakeUintegerAccessor (&YansWifiPhy::GetNumberOfTransmitAntennas,
                                          &YansWifiPhy::SetNumberOfTransmitAntennas),
                    MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("Receivers",
                    "The number of receivers.",
                    UintegerValue (1),
                    MakeUintegerAccessor (&YansWifiPhy::GetNumberOfReceiveAntennas,
                                          &YansWifiPhy::SetNumberOfReceiveAntennas),
                    MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("ShortGuardEnabled",
                    "Whether or not short guard interval is enabled.",
                    BooleanValue (false),
                    MakeBooleanAccessor (&YansWifiPhy::GetGuardInterval,
                                         &YansWifiPhy::SetGuardInterval),
                    MakeBooleanChecker ())
    .AddAttribute ("LdpcEnabled",
                    "Whether or not LDPC is enabled.",
                    BooleanValue (false),
                    MakeBooleanAccessor (&YansWifiPhy::GetLdpc,
                                         &YansWifiPhy::SetLdpc),
                    MakeBooleanChecker ())
    .AddAttribute ("STBCEnabled",
                    "Whether or not STBC is enabled.",
                    BooleanValue (false),
                    MakeBooleanAccessor (&YansWifiPhy::GetStbc,
                                         &YansWifiPhy::SetStbc),
                    MakeBooleanChecker ())
    .AddAttribute ("GreenfieldEnabled",
                    "Whether or not STBC is enabled.",
                    BooleanValue (false),
                    MakeBooleanAccessor (&YansWifiPhy::GetGreenfield,
                                         &YansWifiPhy::SetGreenfield),
                    MakeBooleanChecker ())
    .AddAttribute ("S1g1MfieldEnabled",
                    "Whether or not STBC is enabled.",
                    BooleanValue (false),
                    MakeBooleanAccessor (&YansWifiPhy::GetS1g1Mfield,
```

```
178                                     &YansWifiPhy::SetS1g1Mfield),
179                     MakeBooleanChecker ())
180     .AddAttribute ("S1gShortfieldEnabled",
181                     "Whether or not STBC is enabled.",
182                     BooleanValue (false),   // for test, temporarily
183                     MakeBooleanAccessor (&YansWifiPhy::GetS1gShortfield,
184                                     &YansWifiPhy::SetS1gShortfield),
185                     MakeBooleanChecker ())
186     .AddAttribute ("S1gLongfieldEnabled",
187                     "Whether or not STBC is enabled.",
188                     BooleanValue (false),
189                     MakeBooleanAccessor (&YansWifiPhy::GetS1gLongfield,
190                                     &YansWifiPhy::SetS1gLongfield),
191                     MakeBooleanChecker ())
192     .AddAttribute ("ChannelWidth", "Whether 1MHz, 2MHz, 4MHz, 8MHz, 16MHz, 20MHz or
        ↪  40MHz.",
193                     UintegerValue (1),
194                     MakeUintegerAccessor (&YansWifiPhy::GetChannelWidth,
195                                      &YansWifiPhy::SetChannelWidth),
196                     MakeUintegerChecker<uint32_t> ())
197   ;
198   return tid;
199 }
200
201 YansWifiPhy::YansWifiPhy ()
202   : m_initialized (false),
203     m_channelNumber (1),
204     m_endRxEvent (),
205     m_endPlcpRxEvent (),
206     m_channelStartingFrequency (0),
207     m_mpdusNum (0),
208     m_plcpSuccess (false)
209 {
210   NS_LOG_FUNCTION (this);
211   m_random = CreateObject<UniformRandomVariable> ();
212   m_state = CreateObject<WifiPhyStateHelper> ();
213 }
214
215 YansWifiPhy::~YansWifiPhy ()
216 {
217   NS_LOG_FUNCTION (this);
218 }
219
220 void
221 YansWifiPhy::DoDispose (void)
222 {
223   NS_LOG_FUNCTION (this);
224   m_channel = 0;
```

```
225   m_deviceRateSet.clear ();
226   m_deviceMcsSet.clear ();
227   m_device = 0;
228   m_mobility = 0;
229   m_state = 0;
230 }
231
232 void
233 YansWifiPhy::DoInitialize ()
234 {
235   NS_LOG_FUNCTION (this);
236   m_initialized = true;
237 }
238
239 void
240 YansWifiPhy::ConfigureStandard (enum WifiPhyStandard standard)
241 {
242   NS_LOG_FUNCTION (this << standard);
243   switch (standard)
244     {
245     case WIFI_PHY_STANDARD_80211a:
246       Configure80211a ();
247       break;
248     case WIFI_PHY_STANDARD_80211b:
249       Configure80211b ();
250       break;
251     case WIFI_PHY_STANDARD_80211g:
252       Configure80211g ();
253       break;
254     case WIFI_PHY_STANDARD_80211_10MHZ:
255       Configure80211_10Mhz ();
256       break;
257     case WIFI_PHY_STANDARD_80211_5MHZ:
258       Configure80211_5Mhz ();
259       break;
260     case WIFI_PHY_STANDARD_holland:
261       ConfigureHolland ();
262       break;
263     case WIFI_PHY_STANDARD_80211n_2_4GHZ:
264       m_channelStartingFrequency = 2407;
265       Configure80211n ();
266       break;
267     case WIFI_PHY_STANDARD_80211n_5GHZ:
268       m_channelStartingFrequency = 5e3;
269       Configure80211n ();
270       break;
271      case WIFI_PHY_STANDARD_80211ah:
272       Configure80211ah ();
```

```
273       break;
274     default:
275       NS_ASSERT (false);
276       break;
277     }
278 }
279 
280 void
281 YansWifiPhy::SetRxNoiseFigure (double noiseFigureDb)
282 {
283   NS_LOG_FUNCTION (this << noiseFigureDb);
284   m_interference.SetNoiseFigure (DbToRatio (noiseFigureDb));
285 }
286 
287 void
288 YansWifiPhy::SetTxPowerStart (double start)
289 {
290   NS_LOG_FUNCTION (this << start);
291   m_txPowerBaseDbm = start;
292 }
293 
294 void
295 YansWifiPhy::SetTxPowerEnd (double end)
296 {
297   NS_LOG_FUNCTION (this << end);
298   m_txPowerEndDbm = end;
299 }
300 
301 void
302 YansWifiPhy::SetNTxPower (uint32_t n)
303 {
304   NS_LOG_FUNCTION (this << n);
305   m_nTxPower = n;
306 }
307 
308 void
309 YansWifiPhy::SetTxGain (double gain)
310 {
311   NS_LOG_FUNCTION (this << gain);
312   m_txGainDb = gain;
313 }
314 
315 void
316 YansWifiPhy::SetRxGain (double gain)
317 {
318   NS_LOG_FUNCTION (this << gain);
319   m_rxGainDb = gain;
320 }
```

```
321
322 void
323 YansWifiPhy::SetEdThreshold (double threshold)
324 {
325   NS_LOG_FUNCTION (this << threshold);
326   m_edThresholdW = DbmToW (threshold);
327 }
328
329 void
330 YansWifiPhy::SetCcaMode1Threshold (double threshold)
331 {
332   NS_LOG_FUNCTION (this << threshold);
333   m_ccaMode1ThresholdW = DbmToW (threshold);
334 }
335
336 void
337 YansWifiPhy::SetErrorRateModel (Ptr<ErrorRateModel> rate)
338 {
339   m_interference.SetErrorRateModel (rate);
340 }
341
342 void
343 YansWifiPhy::SetDevice (Ptr<NetDevice> device)
344 {
345   m_device = device;
346 }
347
348 void
349 YansWifiPhy::SetMobility (Ptr<MobilityModel> mobility)
350 {
351   m_mobility = mobility;
352 }
353
354 double
355 YansWifiPhy::GetRxNoiseFigure (void) const
356 {
357   return RatioToDb (m_interference.GetNoiseFigure ());
358 }
359
360 double
361 YansWifiPhy::GetTxPowerStart (void) const
362 {
363   return m_txPowerBaseDbm;
364 }
365
366 double
367 YansWifiPhy::GetTxPowerEnd (void) const
368 {
```

```
369   return m_txPowerEndDbm;
370 }
371
372 double
373 YansWifiPhy::GetTxGain (void) const
374 {
375   return m_txGainDb;
376 }
377
378 double
379 YansWifiPhy::GetRxGain (void) const
380 {
381   return m_rxGainDb;
382 }
383
384 double
385 YansWifiPhy::GetEdThreshold (void) const
386 {
387   return WToDbm (m_edThresholdW);
388 }
389
390 double
391 YansWifiPhy::GetCcaMode1Threshold (void) const
392 {
393   return WToDbm (m_ccaMode1ThresholdW);
394 }
395
396 Ptr<ErrorRateModel>
397 YansWifiPhy::GetErrorRateModel (void) const
398 {
399   return m_interference.GetErrorRateModel ();
400 }
401
402 Ptr<NetDevice>
403 YansWifiPhy::GetDevice (void) const
404 {
405   return m_device;
406 }
407
408 Ptr<MobilityModel>
409 YansWifiPhy::GetMobility (void)
410 {
411   if (m_mobility != 0)
412     {
413       return m_mobility;
414     }
415   else
416     {
```

```
417       return m_device->GetNode ()->GetObject<MobilityModel> ();
418     }
419 }
420 
421 double
422 YansWifiPhy::CalculateSnr (WifiMode txMode, double ber) const
423 {
424   return m_interference.GetErrorRateModel ()->CalculateSnr (txMode, ber);
425 }
426 
427 Ptr<WifiChannel>
428 YansWifiPhy::GetChannel (void) const
429 {
430   return m_channel;
431 }
432 
433 void
434 YansWifiPhy::SetChannel (Ptr<YansWifiChannel> channel)
435 {
436   m_channel = channel;
437   m_channel->Add (this);
438 }
439 
440 void
441 YansWifiPhy::SetChannelNumber (uint16_t nch)
442 {
443   if (!m_initialized)
444     {
445       //this is not channel switch, this is initialization
446       NS_LOG_DEBUG ("start at channel " << nch);
447       m_channelNumber = nch;
448       return;
449     }
450 
451   NS_ASSERT (!IsStateSwitching ());
452   switch (m_state->GetState ())
453     {
454     case YansWifiPhy::RX:
455       NS_LOG_DEBUG ("drop packet because of channel switching while reception");
456       m_endPlcpRxEvent.Cancel ();
457       m_endRxEvent.Cancel ();
458       goto switchChannel;
459       break;
460     case YansWifiPhy::TX:
461       NS_LOG_DEBUG ("channel switching postponed until end of current transmission");
462       Simulator::Schedule (GetDelayUntilIdle (), &YansWifiPhy::SetChannelNumber, this,
      ↪  nch);
463       break;
```

```
464       case YansWifiPhy::CCA_BUSY:
465       case YansWifiPhy::IDLE:
466         goto switchChannel;
467         break;
468       case YansWifiPhy::SLEEP:
469         NS_LOG_DEBUG ("channel switching ignored in sleep mode");
470         break;
471       default:
472         NS_ASSERT (false);
473         break;
474       }
475
476     return;
477
478   switchChannel:
479
480     NS_LOG_DEBUG ("switching channel " << m_channelNumber << " -> " << nch);
481     m_state->SwitchToChannelSwitching (m_channelSwitchDelay);
482     m_interference.EraseEvents ();
483     /*
484      * Needed here to be able to correctly sensed the medium for the first
485      * time after the switching. The actual switching is not performed until
486      * after m_channelSwitchDelay. Packets received during the switching
487      * state are added to the event list and are employed later to figure
488      * out the state of the medium after the switching.
489      */
490     m_channelNumber = nch;
491   }
492
493   uint16_t
494   YansWifiPhy::GetChannelNumber (void) const
495   {
496     return m_channelNumber;
497   }
498
499   Time
500   YansWifiPhy::GetChannelSwitchDelay (void) const
501   {
502     return m_channelSwitchDelay;
503   }
504
505   double
506   YansWifiPhy::GetChannelFrequencyMhz () const
507   {
508     return m_channelStartingFrequency + 5 * GetChannelNumber ();
509   }
510
511   void
```

```
512 YansWifiPhy::SetSleepMode (void)
513 {
514   NS_LOG_FUNCTION (this);
515   switch (m_state->GetState ())
516     {
517     case YansWifiPhy::TX:
518       NS_LOG_DEBUG ("setting sleep mode postponed until end of current transmission");
519       Simulator::Schedule (GetDelayUntilIdle (), &YansWifiPhy::SetSleepMode, this);
520       break;
521     case YansWifiPhy::RX:
522       NS_LOG_DEBUG ("setting sleep mode postponed until end of current reception");
523       Simulator::Schedule (GetDelayUntilIdle (), &YansWifiPhy::SetSleepMode, this);
524       break;
525     case YansWifiPhy::SWITCHING:
526       NS_LOG_DEBUG ("setting sleep mode postponed until end of channel switching");
527       Simulator::Schedule (GetDelayUntilIdle (), &YansWifiPhy::SetSleepMode, this);
528       break;
529     case YansWifiPhy::CCA_BUSY:
530     case YansWifiPhy::IDLE:
531       NS_LOG_DEBUG ("setting sleep mode");
532       m_state->SwitchToSleep ();
533       break;
534     case YansWifiPhy::SLEEP:
535       NS_LOG_DEBUG ("already in sleep mode");
536       break;
537     default:
538       NS_ASSERT (false);
539       break;
540     }
541 }
542
543
544 void
545 YansWifiPhy::ResumeFromSleep (void)
546 {
547   NS_LOG_FUNCTION (this);
548   switch (m_state->GetState ())
549     {
550     case YansWifiPhy::TX:
551     case YansWifiPhy::RX:
552     case YansWifiPhy::IDLE:
553     case YansWifiPhy::CCA_BUSY:
554     case YansWifiPhy::SWITCHING:
555     case YansWifiPhy::OFF:
556       {
557         NS_LOG_DEBUG ("not in sleep mode, there is nothing to resume");
558         break;
559       }
```

```
560       case YansWifiPhy::SLEEP:
561         {
562           NS_LOG_DEBUG ("resuming from sleep mode");
563           Time delayUntilCcaEnd = m_interference.GetEnergyDuration (m_ccaMode1ThresholdW);
564           m_state->SwitchFromSleep (delayUntilCcaEnd);
565           break;
566         }
567       default:
568         {
569           NS_ASSERT (false);
570           break;
571         }
572       }
573   }
574
575   void
576   YansWifiPhy::SetReceiveOkCallback (RxOkCallback callback)
577   {
578     m_state->SetReceiveOkCallback (callback);
579   }
580
581   void
582   YansWifiPhy::SetReceiveErrorCallback (RxErrorCallback callback)
583   {
584     m_state->SetReceiveErrorCallback (callback);
585   }
586
587   void
588   YansWifiPhy::StartReceivePreambleAndHeader (Ptr<Packet> packet,
589                                               double rxPowerDbm,
590                                               WifiTxVector txVector,
591                                               enum WifiPreamble preamble,
592                                               uint8_t packetType,
593                                               Time rxDuration,
594                                               double psFactor,
595                                               Ptr<SwiptHarvester> swiptH)
596   {
597     //This function should be later split to check separately wether plcp preamble and plcp
      ↪  header can be successfully received.
598     //Note: plcp preamble reception is not yet modeled.
599     //NS_LOG_UNCOND (packet << "\t" << rxPowerDbm << " dbm " << "\t" << txVector.GetMode ()
      ↪  << "\t" << packet->GetSize ()); //test
600     NS_LOG_FUNCTION (this << packet << rxPowerDbm << txVector.GetMode () << preamble <<
      ↪  (uint32_t)packetType);
601     AmpduTag ampduTag;
602     rxPowerDbm += m_rxGainDb;
603
604     // THIS PART WAS ADDED BY JOSEF IN ORDER TO IMPLEMENT THE SWIPT PSFACTOR         //
```

```
605   //===========================================================================//
606   m_swiptHarvester = swiptH; // To be passed to the InterferenceHelper class
607   m_psFactor = psFactor;
608   double rxPowerWat = DbmToW (rxPowerDbm);
609   double rxPowerW;
610   if (m_state->GetState ()== YansWifiPhy::OFF)
611   {
612     rxPowerW = rxPowerWat;
613   }
614   else
615   {
616     rxPowerW = rxPowerWat * m_psFactor;
617   }
618   //===========================================================================//
619
620   Time endRx = Simulator::Now () + rxDuration;
621   Time preambleAndHeaderDuration = CalculatePlcpPreambleAndHeaderDuration (txVector,
      ↪  preamble);
622
623
624
625     Ptr<InterferenceHelper::Event> event;
626     event = m_interference.Add (packet->GetSize (),
627                                 txVector,
628                                 preamble,
629                                 rxDuration,
630                                 rxPowerW); // THE PSFACTOR GETS PASSED TO THE
                                    ↪  INTERFERENCEHELPER WITH THE VALUE OF RXPOWERW
631
632   switch (m_state->GetState ())
633     {
634     case YansWifiPhy::OFF:
635       NS_LOG_DEBUG ("drop packet because of device is OFF");
636       NotifyRxDrop (packet, DeviceIsOff);
637       m_plcpSuccess = false;
638       break;
639     case YansWifiPhy::SWITCHING:
640       NS_LOG_DEBUG ("drop packet because of channel switching");
641       NotifyRxDrop (packet, PhyRxDuringChannelSwitching);
642       m_plcpSuccess = false;
643       /*
644        * Packets received on the upcoming channel are added to the event list
645        * during the switching state. This way the medium can be correctly sensed
646        * when the device listens to the channel for the first time after the
647        * switching e.g. after channel switching, the channel may be sensed as
648        * busy due to other devices' tramissions started before the end of
649        * the switching.
650        */
```

```
651       if (endRx > Simulator::Now () + m_state->GetDelayUntilIdle ())
652         {
653           //that packet will be noise _after_ the completion of the
654           //channel switching.
655           goto maybeCcaBusy;
656         }
657       break;
658     case YansWifiPhy::RX:
659       NS_LOG_DEBUG ("drop packet because already in Rx (power=" <<
660                     rxPowerW << "W)");
661       NotifyRxDrop (packet, PhyAlreadyReceiving);
662       if (endRx > Simulator::Now () + m_state->GetDelayUntilIdle ())
663         {
664           //that packet will be noise _after_ the reception of the
665           //currently-received packet.
666           goto maybeCcaBusy;
667         }
668       break;
669     case YansWifiPhy::TX:
670       NS_LOG_DEBUG ("drop packet because already in Tx (power=" <<
671                     rxPowerW << "W)");
672       NotifyRxDrop (packet, PhyAlreadyTransmitting);
673       if (endRx > Simulator::Now () + m_state->GetDelayUntilIdle ())
674         {
675           //that packet will be noise _after_ the transmission of the
676           //currently-transmitted packet.
677           goto maybeCcaBusy;
678         }
679       break;
680     case YansWifiPhy::CCA_BUSY:
681     case YansWifiPhy::IDLE:
682
683       //NS_LOG_UNCOND ("rxPowerW " << rxPowerW << "\t" << ",m_edThresholdW " <<
       ↪  m_edThresholdW << "," << packet);
684
685       if (rxPowerW > m_edThresholdW) //checked here, no need to check in the payload
       ↪  reception (current implementation assumes constant rx power over the packet
       ↪  duration)
686         {
687           if (preamble == WIFI_PREAMBLE_NONE && m_mpdusNum == 0)
688             {
689               NS_LOG_DEBUG ("drop packet because no preamble has been received");
690               NotifyRxDrop (packet, PhyPreampleHeaderReceptionFailed);
691               goto maybeCcaBusy;
692             }
693           else if (preamble == WIFI_PREAMBLE_NONE && m_plcpSuccess == false) //A-MPDU
           ↪  reception fails
694             {
```

```
695               NS_LOG_DEBUG ("Drop MPDU because no plcp has been received");
696               NotifyRxDrop (packet, PhyPreampleHeaderReceptionFailed);
697               goto maybeCcaBusy;
698             }
699           else if (preamble != WIFI_PREAMBLE_NONE && packet->PeekPacketTag (ampduTag) &&
          ↪  m_mpdusNum == 0)
700             {
701               //received the first MPDU in an MPDU
702               m_mpdusNum = ampduTag.GetNoOfMpdus () - 1;
703             }
704           else if (preamble == WIFI_PREAMBLE_NONE && packet->PeekPacketTag (ampduTag) &&
          ↪  m_mpdusNum > 0)
705             {
706               //received the other MPDUs that are part of the A-MPDU
707               if (ampduTag.GetNoOfMpdus () < m_mpdusNum)
708                 {
709                   NS_LOG_DEBUG ("Missing MPDU from the A-MPDU " << m_mpdusNum -
                    ↪  ampduTag.GetNoOfMpdus ());
710                   m_mpdusNum = ampduTag.GetNoOfMpdus ();
711                 }
712               else
713                 {
714                   m_mpdusNum--;
715                 }
716             }
717           else if (preamble != WIFI_PREAMBLE_NONE && m_mpdusNum > 0 )
718             {
719               NS_LOG_DEBUG ("Didn't receive the last MPDUs from an A-MPDU " << m_mpdusNum);
720               m_mpdusNum = 0;
721             }
722
723           NS_LOG_DEBUG ("sync to signal (power=" << rxPowerW << "W)");
724           //sync to signal
725           m_state->SwitchToRx (rxDuration);
726           NS_ASSERT (m_endPlcpRxEvent.IsExpired ());
727           NotifyRxBegin (packet);
728           m_interference.NotifyRxStart ();
729
730           if (preamble != WIFI_PREAMBLE_NONE)
731             {
732               NS_ASSERT (m_endPlcpRxEvent.IsExpired ());
733               m_endPlcpRxEvent = Simulator::Schedule (preambleAndHeaderDuration,
                ↪  &YansWifiPhy::StartReceivePacket, this,
734                                                       packet, txVector, preamble, event,
                                                        ↪  swiptH);
735             }
736
737           NS_ASSERT (m_endRxEvent.IsExpired ());
```

```
738           m_endRxEvent = Simulator::Schedule (rxDuration, &YansWifiPhy::EndReceive, this,
739                                               packet, preamble, packetType, event, swiptH);
740         }
741       else
742         {
743           NS_LOG_DEBUG ("drop packet because signal power too Small (" <<
744                         rxPowerW << "<" << m_edThresholdW << ")");
745           NotifyRxDrop (packet, PhyNotEnoughSignalPower);
746           m_plcpSuccess = false;
747           goto maybeCcaBusy;
748         }
749       break;
750     case YansWifiPhy::SLEEP:
751       NS_LOG_DEBUG ("Check if a packet will be dropped because in sleep mode");
752       NotifyRxDrop (packet, PhyInSleepMode);
753       m_plcpSuccess = false;
754       //}
755       break;
756     }
757
758   return;
759
760 maybeCcaBusy:
761   //We are here because we have received the first bit of a packet and we are
762   //not going to be able to synchronize on it
763   //In this model, CCA becomes busy when the aggregation of all signals as
764   //tracked by the InterferenceHelper class is higher than the CcaBusyThreshold
765
766   Time delayUntilCcaEnd = m_interference.GetEnergyDuration (m_ccaMode1ThresholdW);
767   if (!delayUntilCcaEnd.IsZero ())
768     {
769       m_state->SwitchMaybeToCcaBusy (delayUntilCcaEnd);
770     }
771 }
772
773 void
774 YansWifiPhy::StartReceivePacket (Ptr<Packet> packet,
775                                  WifiTxVector txVector,
776                                  enum WifiPreamble preamble,
777                                  //uint8_t packetType,
778                                  Ptr<InterferenceHelper::Event> event,
779                                  Ptr<SwiptHarvester> swiptHarvester)
780 {
781   //NS_LOG_FUNCTION (this << packet << txVector.GetMode () << preamble <<
   ↪  (uint32_t)packetType);
782   NS_LOG_FUNCTION (this << packet << txVector.GetMode () << preamble);
783
784
```

```
785   NS_ASSERT (IsStateRx ());
786   NS_ASSERT (m_endPlcpRxEvent.IsExpired ());
787   AmpduTag ampduTag;
788   WifiMode txMode = txVector.GetMode ();
789
790   struct InterferenceHelper::SnrPer snrPer;
791
792   Time preambleAndHeaderDuration = CalculatePlcpPreambleAndHeaderDuration (txVector,
      ↪  preamble);
793
794   snrPer = m_interference.CalculatePlcpHeaderSnrPer (event, preambleAndHeaderDuration,
      ↪  m_psFactor, swiptHarvester);
795
796   NS_LOG_DEBUG ("snr=" << snrPer.snr << ", per=" << snrPer.per);
797
798   if (m_random->GetValue () > snrPer.per)   //plcp reception succeeded
799     {
800       if (IsModeSupported (txMode) || IsMcsSupported (txMode))
801         {
802           NS_LOG_DEBUG ("receiving plcp payload");     //endReceive is already scheduled
803           m_plcpSuccess = true;
804         }
805       else     //mode is not allowed
806         {
807           NS_LOG_DEBUG ("drop packet because it was sent using an unsupported mode (" <<
              ↪  txMode << ")");
808           //NS_LOG_UNCOND ("drop packet because it was sent using an unsupported mode (" <<
              ↪  txMode << ")");
809           NotifyRxDrop (packet, PhyUnsupportedMode);
810           m_plcpSuccess = false;
811         }
812     }
813   else   //plcp reception failed
814     {
815       NS_LOG_DEBUG ("drop packet because plcp preamble/header reception failed");
816       NotifyRxDrop (packet, PhyPlcpReceptionFailed);
817       m_plcpSuccess = false;
818     }
819 }
820
821 void
822 YansWifiPhy::SendPacket (Ptr<const Packet> packet, WifiTxVector txVector, WifiPreamble
    ↪  preamble, uint8_t packetType)
823 {
824   NS_LOG_FUNCTION (this << packet << txVector.GetMode () << preamble <<
      ↪  (uint32_t)txVector.GetTxPowerLevel () << (uint32_t)packetType);
825   /* Transmission can happen if:
826    *  - we are syncing on a packet. It is the responsability of the
```

```
827     *    MAC layer to avoid doing this but the PHY does nothing to
828     *    prevent it.
829     *  - we are idle
830     */
831   NS_ASSERT (!m_state->IsStateTx () && !m_state->IsStateSwitching ());
832 
833   if (m_state->IsStateSleep ())
834     {
835         NS_LOG_DEBUG ("Dropping packet because in sleep mode");
836         NotifyTxDrop (packet, PhyInSleepMode);
837         return;
838     }
839 
840   Time txDuration = CalculateTxDuration (packet->GetSize (), txVector, preamble,
    ↪  GetFrequency (), packetType, 1);
841   if (m_state->IsStateRx ())
842     {
843       m_endPlcpRxEvent.Cancel ();
844       m_endRxEvent.Cancel ();
845       m_interference.NotifyRxEnd ();
846     }
847   NotifyTxBegin(packet, txDuration);
848   uint32_t dataRate500KbpsUnits;
849   if (txVector.GetMode ().GetModulationClass () == WIFI_MOD_CLASS_HT || txVector.GetMode
    ↪  ().GetModulationClass () == WIFI_MOD_CLASS_S1G)
850     {
851       dataRate500KbpsUnits = 128 + WifiModeToMcs (txVector.GetMode ());
852     }
853   else
854     {
855       dataRate500KbpsUnits = txVector.GetMode ().GetDataRate () * txVector.GetNss () /
        ↪  500000;
856     }
857   bool isShortPreamble = (WIFI_PREAMBLE_SHORT == preamble);
858   NotifyMonitorSniffTx (packet, (uint16_t)GetChannelFrequencyMhz (), GetChannelNumber (),
    ↪  dataRate500KbpsUnits, isShortPreamble, txVector);
859   m_state->SwitchToTx (txDuration, packet, GetPowerDbm (txVector.GetTxPowerLevel ()),
    ↪  txVector, preamble);
860   m_channel->Send (this, packet, GetPowerDbm (txVector.GetTxPowerLevel ()) + m_txGainDb,
    ↪  txVector, preamble, packetType, txDuration);
861 }
862 
863 void
864 OnTxEnd (YansWifiPhy* obj, Ptr<const Packet> packet)
865 {
866     obj->NotifyTxEnd (packet);
867 }
868 
```

```
869 void
870 YansWifiPhy::NotifyTxBegin (Ptr<const Packet> packet, Time duration)
871 {
872     WifiPhy::NotifyTxBegin(packet);
873 
874     //std::cout << this->m_device->GetAddress() << " " <<
    ↪  Simulator::Now().GetMicroSeconds() << " Scheduling end tx " <<
    ↪  time.GetMicroSeconds()  << std::endl;
875     Simulator::Schedule(duration, &OnTxEnd, this, packet);
876 }
877 
878 uint32_t
879 YansWifiPhy::GetNModes (void) const
880 {
881   return m_deviceRateSet.size ();
882 }
883 
884 WifiMode
885 YansWifiPhy::GetMode (uint32_t mode) const
886 {
887   return m_deviceRateSet[mode];
888 }
889 
890 bool
891 YansWifiPhy::IsModeSupported (WifiMode mode) const
892 {
893   for (uint32_t i = 0; i < GetNModes (); i++)
894     {
895       if (mode == GetMode (i))
896         {
897           return true;
898         }
899     }
900   return false;
901 }
902 bool
903 YansWifiPhy::IsMcsSupported (WifiMode mode)
904 {
905     //NS_LOG_UNCOND ("IsMcsSupported, " << mode << "\t" << GetNMcs ());
906   for (uint32_t i = 0; i < GetNMcs (); i++)
907     {
908       //NS_LOG_UNCOND ("IsMcsSupported-aa, " << mode << "\t" << McsToWifiMode (GetMcs
      ↪  (i)));
909       if (mode == McsToWifiMode (GetMcs (i)))
910         {
911           return true;
912         }
913     }
```

```
914       //NS_LOG_UNCOND ("IsMcsSupported not supported, " << mode);
915     return false;
916   }
917
918   uint32_t
919   YansWifiPhy::GetNTxPower (void) const
920   {
921     return m_nTxPower;
922   }
923
924   void
925   YansWifiPhy::Configure80211a (void)
926   {
927     NS_LOG_FUNCTION (this);
928     m_channelStartingFrequency = 5e3; //5.000 GHz
929
930     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6Mbps ());
931     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate9Mbps ());
932     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate12Mbps ());
933     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate18Mbps ());
934     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate24Mbps ());
935     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate36Mbps ());
936     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate48Mbps ());
937     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate54Mbps ());
938   }
939
940   void
941   YansWifiPhy::Configure80211b (void)
942   {
943     NS_LOG_FUNCTION (this);
944     m_channelStartingFrequency = 2407; //2.407 GHz
945
946     m_deviceRateSet.push_back (WifiPhy::GetDsssRate1Mbps ());
947     m_deviceRateSet.push_back (WifiPhy::GetDsssRate2Mbps ());
948     m_deviceRateSet.push_back (WifiPhy::GetDsssRate5_5Mbps ());
949     m_deviceRateSet.push_back (WifiPhy::GetDsssRate11Mbps ());
950   }
951
952   void
953   YansWifiPhy::Configure80211g (void)
954   {
955     NS_LOG_FUNCTION (this);
956     m_channelStartingFrequency = 2407; //2.407 GHz
957
958     m_deviceRateSet.push_back (WifiPhy::GetDsssRate1Mbps ());
959     m_deviceRateSet.push_back (WifiPhy::GetDsssRate2Mbps ());
960     m_deviceRateSet.push_back (WifiPhy::GetDsssRate5_5Mbps ());
961     m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate6Mbps ());
```

```
962   m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate9Mbps ());
963   m_deviceRateSet.push_back (WifiPhy::GetDsssRate11Mbps ());
964   m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate12Mbps ());
965   m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate18Mbps ());
966   m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate24Mbps ());
967   m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate36Mbps ());
968   m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate48Mbps ());
969   m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate54Mbps ());
970 }
971
972 void
973 YansWifiPhy::Configure80211_10Mhz (void)
974 {
975   NS_LOG_FUNCTION (this);
976   m_channelStartingFrequency = 5e3; //5.000 GHz, suppose 802.11a
977
978   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3MbpsBW10MHz ());
979   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate4_5MbpsBW10MHz ());
980   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6MbpsBW10MHz ());
981   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate9MbpsBW10MHz ());
982   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate12MbpsBW10MHz ());
983   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate18MbpsBW10MHz ());
984   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate24MbpsBW10MHz ());
985   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate27MbpsBW10MHz ());
986 }
987
988 void
989 YansWifiPhy::Configure80211_5Mhz (void)
990 {
991   NS_LOG_FUNCTION (this);
992   m_channelStartingFrequency = 5e3; //5.000 GHz, suppose 802.11a
993
994   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_5MbpsBW5MHz ());
995   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_25MbpsBW5MHz ());
996   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3MbpsBW5MHz ());
997   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate4_5MbpsBW5MHz ());
998   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6MbpsBW5MHz ());
999   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate9MbpsBW5MHz ());
1000  m_deviceRateSet.push_back (WifiPhy::GetOfdmRate12MbpsBW5MHz ());
1001  m_deviceRateSet.push_back (WifiPhy::GetOfdmRate13_5MbpsBW5MHz ());
1002 }
1003
1004 void
1005 YansWifiPhy::ConfigureHolland (void)
1006 {
1007   NS_LOG_FUNCTION (this);
1008   m_channelStartingFrequency = 5e3; //5.000 GHz
1009   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6Mbps ());
```

```
1010   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate12Mbps ());
1011   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate18Mbps ());
1012   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate36Mbps ());
1013   m_deviceRateSet.push_back (WifiPhy::GetOfdmRate54Mbps ());
1014 }
1015
1016 void
1017 YansWifiPhy::Configure80211n (void)
1018 {
1019   NS_LOG_FUNCTION (this);
1020   if (m_channelStartingFrequency >= 2400 && m_channelStartingFrequency <= 2500) //at 2.4
      ↪  GHz
1021     {
1022       m_deviceRateSet.push_back (WifiPhy::GetDsssRate1Mbps ());
1023       m_deviceRateSet.push_back (WifiPhy::GetDsssRate2Mbps ());
1024       m_deviceRateSet.push_back (WifiPhy::GetDsssRate5_5Mbps ());
1025       m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate6Mbps ());
1026       m_deviceRateSet.push_back (WifiPhy::GetDsssRate11Mbps ());
1027       m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate12Mbps ());
1028       m_deviceRateSet.push_back (WifiPhy::GetErpOfdmRate24Mbps ());
1029     }
1030   if (m_channelStartingFrequency >= 5000 && m_channelStartingFrequency <= 6000) //at 5 GHz
1031     {
1032       m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6Mbps ());
1033       m_deviceRateSet.push_back (WifiPhy::GetOfdmRate12Mbps ());
1034       m_deviceRateSet.push_back (WifiPhy::GetOfdmRate24Mbps ());
1035     }
1036   m_bssMembershipSelectorSet.push_back (HT_PHY);
1037   for (uint8_t i = 0; i < 8; i++)
1038     {
1039       m_deviceMcsSet.push_back (i);
1040     }
1041 }
1042
1043 void
1044 YansWifiPhy::Configure80211ah (void)
1045 {
1046     NS_LOG_FUNCTION (this);
1047     m_channelStartingFrequency = 9e2;
1048
1049     // need to check for 802.11ah
1050     //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6Mbps ());
1051     //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate300KbpsBW1MHz ());
1052     //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate650KbpsBW2MHz ());
1053
1054     //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate12Mbps ());
1055     //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate24Mbps ());
1056
```

```
1057     if (GetChannelWidth () == 2)
1058     {
1059     //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate150KbpsBW1MHz ());
1060     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate300KbpsBW1MHz ());
1061     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate600KbpsBW1MHz ());
1062     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate900KbpsBW1MHz ());
1063     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_2MbpsBW1MHz ());
1064     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_8MbpsBW1MHz ());
1065     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_4MbpsBW1MHz ());
1066     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_7MbpsBW1MHz ());
1067     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3MbpsBW1MHz ());
1068     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3_6MbpsBW1MHz ());
1069     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate4MbpsBW1MHz ());
1070
1071
1072     //supportedmodes.push_back (WifiPhy::GetOfdmRate150KbpsBW1MHz ());
1073     //mandatory MCS 0 to 7, 2Mhz
1074
1075     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate650KbpsBW2MHz ());
1076     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_3MbpsBW2MHz ());
1077     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_95MbpsBW2MHz ());
1078     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_6MbpsBW2MHz ());
1079     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3_9MbpsBW2MHz ());
1080     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate5_2MbpsBW2MHz ());
1081     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate5_85MbpsBW2MHz ());
1082     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6_5MbpsBW2MHz ());
1083     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate7_8MbpsBW2MHz ());
1084     }
1085   else if (GetChannelWidth () == 1)
1086   {
1087     //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate150KbpsBW1MHz ());
1088     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate300KbpsBW1MHz ());
1089     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate600KbpsBW1MHz ());
1090     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate900KbpsBW1MHz ());
1091     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_2MbpsBW1MHz ());
1092     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_8MbpsBW1MHz ());
1093     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_4MbpsBW1MHz ());
1094     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_7MbpsBW1MHz ());
1095     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3MbpsBW1MHz ());
1096     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3_6MbpsBW1MHz ());
1097     m_deviceRateSet.push_back (WifiPhy::GetOfdmRate4MbpsBW1MHz ());
1098   }
1099   else if (GetChannelWidth () == 4)
1100   {
1101       //m_deviceRateSet.push_back (WifiPhy::GetOfdmRate150KbpsBW1MHz ());
1102       m_deviceRateSet.push_back (WifiPhy::GetOfdmRate300KbpsBW1MHz ());
1103       m_deviceRateSet.push_back (WifiPhy::GetOfdmRate600KbpsBW1MHz ());
1104       m_deviceRateSet.push_back (WifiPhy::GetOfdmRate900KbpsBW1MHz ());
```

```
1105        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_2MbpsBW1MHz ());
1106        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_8MbpsBW1MHz ());
1107        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_4MbpsBW1MHz ());
1108        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_7MbpsBW1MHz ());
1109        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3MbpsBW1MHz ());
1110        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3_6MbpsBW1MHz ());
1111        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate4MbpsBW1MHz ());
1112
1113
1114        //supportedmodes.push_back (WifiPhy::GetOfdmRate150KbpsBW1MHz ());
1115        //mandatory MCS 0 to 7, 2Mhz
1116
1117        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate650KbpsBW2MHz ());
1118        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_3MbpsBW2MHz ());
1119        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_95MbpsBW2MHz ());
1120        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_6MbpsBW2MHz ());
1121        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate3_9MbpsBW2MHz ());
1122        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate5_2MbpsBW2MHz ());
1123        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate5_85MbpsBW2MHz ());
1124        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate6_5MbpsBW2MHz ());
1125        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate7_8MbpsBW2MHz ());
1126
1127        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate1_35MbpsBW4MHz ());
1128        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate2_7MbpsBW4MHz ());
1129        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate4_05MbpsBW4MHz ());
1130        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate5_4MbpsBW4MHz ());
1131        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate8_1MbpsBW4MHz ());
1132        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate10_8MbpsBW4MHz ());
1133        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate12_15MbpsBW4MHz ());
1134        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate13_5MbpsBW4MHz ());
1135        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate16_2MbpsBW4MHz ());
1136        m_deviceRateSet.push_back (WifiPhy::GetOfdmRate18MbpsBW4MHz ());
1137    }
1138
1139     m_bssMembershipSelectorSet.push_back(S1G_PHY);
1140     for (uint8_t i=0; i <11; i++)
1141     {
1142         //NS_LOG_UNCOND ("YansWifiPhy::Configure80211ah");
1143         m_deviceMcsSet.push_back(i);
1144     }
1145 }
1146
1147
1148 void
1149 YansWifiPhy::RegisterListener (WifiPhyListener *listener)
1150 {
1151   m_state->RegisterListener (listener);
1152 }
```

```
1153
1154 void
1155 YansWifiPhy::UnregisterListener (WifiPhyListener *listener)
1156 {
1157   m_state->UnregisterListener (listener);
1158 }
1159
1160 bool
1161 YansWifiPhy::IsStateCcaBusy (void)
1162 {
1163   return m_state->IsStateCcaBusy ();
1164 }
1165
1166 bool
1167 YansWifiPhy::IsStateIdle (void)
1168 {
1169   return m_state->IsStateIdle ();
1170 }
1171
1172 bool
1173 YansWifiPhy::IsStateBusy (void)
1174 {
1175   return m_state->IsStateBusy ();
1176 }
1177
1178 bool
1179 YansWifiPhy::IsStateRx (void)
1180 {
1181   return m_state->IsStateRx ();
1182 }
1183
1184 bool
1185 YansWifiPhy::IsStateTx (void)
1186 {
1187   return m_state->IsStateTx ();
1188 }
1189
1190 bool
1191 YansWifiPhy::IsStateSwitching (void)
1192 {
1193   return m_state->IsStateSwitching ();
1194 }
1195
1196 bool
1197 YansWifiPhy::IsStateSleep (void)
1198 {
1199   return m_state->IsStateSleep ();
1200 }
```

```
1201
1202 Time
1203 YansWifiPhy::GetStateDuration (void)
1204 {
1205   return m_state->GetStateDuration ();
1206 }
1207
1208 Time
1209 YansWifiPhy::GetDelayUntilIdle (void)
1210 {
1211   return m_state->GetDelayUntilIdle ();
1212 }
1213
1214 Time
1215 YansWifiPhy::GetLastRxStartTime (void) const
1216 {
1217   return m_state->GetLastRxStartTime ();
1218 }
1219
1220 double
1221 YansWifiPhy::DbToRatio (double dB) const
1222 {
1223   double ratio = std::pow (10.0, dB / 10.0);
1224   return ratio;
1225 }
1226
1227 double
1228 YansWifiPhy::DbmToW (double dBm) const
1229 {
1230   double mW = std::pow (10.0, dBm / 10.0);
1231   return mW / 1000.0;
1232 }
1233
1234 double
1235 YansWifiPhy::WToDbm (double w) const
1236 {
1237   return 10.0 * std::log10 (w * 1000.0);
1238 }
1239
1240 double
1241 YansWifiPhy::RatioToDb (double ratio) const
1242 {
1243   return 10.0 * std::log10 (ratio);
1244 }
1245
1246 double
1247 YansWifiPhy::GetEdThresholdW (void) const
1248 {
```

```
1249   return m_edThresholdW;
1250 }
1251
1252 double
1253 YansWifiPhy::GetPowerDbm (uint8_t power) const
1254 {
1255   NS_ASSERT (m_txPowerBaseDbm <= m_txPowerEndDbm);
1256   NS_ASSERT (m_nTxPower > 0);
1257   double dbm;
1258   if (m_nTxPower > 1)
1259     {
1260       dbm = m_txPowerBaseDbm + power * (m_txPowerEndDbm - m_txPowerBaseDbm) / (m_nTxPower -
       ↪  1);
1261     }
1262   else
1263     {
1264       NS_ASSERT_MSG (m_txPowerBaseDbm == m_txPowerEndDbm, "cannot have TxPowerEnd !=
       ↪  TxPowerStart with TxPowerLevels == 1");
1265       dbm = m_txPowerBaseDbm;
1266     }
1267   return dbm;
1268 }
1269
1270 void
1271 YansWifiPhy::EndReceive (Ptr<Packet> packet,
1272                          enum WifiPreamble preamble,
1273                          uint8_t packetType,
1274                          Ptr<InterferenceHelper::Event> event,
1275                          Ptr<SwiptHarvester> swiptHarvester)
1276 {
1277   NS_LOG_FUNCTION (this << packet << event);
1278   NS_ASSERT (IsStateRx ());
1279   NS_ASSERT (event->GetEndTime () == Simulator::Now ());
1280
1281   struct InterferenceHelper::SnrPer snrPer;
1282   snrPer = m_interference.CalculatePlcpPayloadSnrPer (event, m_psFactor, swiptHarvester);
1283   m_interference.NotifyRxEnd ();
1284
1285   WifiMacHeader hdr;
1286   packet->PeekHeader(hdr);
1287
1288         Mac48Address source = hdr.GetAddr2();
1289
1290   uint8_t mac[6];
1291   source.CopyTo (mac);
1292   uint8_t aid_l = mac[5];
1293   uint8_t aid_h = mac[4] & 0x1f;
1294   uint16_t aid = (aid_h << 8) | (aid_l << 0);
```

```
1295
1296    double random = m_random->GetValue ();
1297    bool result = (random > snrPer.per);
1298
1299
1300    //NS_LOG_UNCOND ("YansWifiPhy::EndReceive, mode=" << (event->GetPayloadMode
        ↪  ().GetDataRate ()) <<
1301                //   ", snr=" << snrPer.snr << ", per=" << snrPer.per << ", size=" <<
                    ↪  packet->GetSize ());
1302
1303    //std::stringstream ss;
1304          //ss << "/media/jose/ext32v2/test3/subdir/YansEndRcvPkt.csv";
1305          //static std::fstream f (ss.str().c_str(), std::ios::out);
1306    //uint32_t nodeID = GetDevice ()->GetNode ()->GetId ();
1307          ////f << std::fixed << std::setprecision(9) << Simulator::Now ().GetSeconds () <<
              ↪  ", " << nodeID << ", " << snrPer.snr << ", " << snrPer.per << ", " <<
              ↪  event->GetRxPowerW () << std::endl;
1308          //f << Simulator::Now ().GetSeconds () << ", " << m_psFactor << ", " << source <<
              ↪  ", " << aid - 1 << ", " << nodeID <<  ", " << snrPer.snr << ", " << random <<
              ↪  ", " << snrPer.per << ", " << result << ", " << event->GetRxPowerW () << ", "
              ↪  << m_plcpSuccess << std::endl;
1309
1310
1311    if (m_plcpSuccess == true)
1312      {
1313        NS_LOG_DEBUG ("mode=" << (event->GetPayloadMode ().GetDataRate ()) << ", snr=" <<
          ↪  snrPer.snr << ", per=" << snrPer.per << ", size=" << packet->GetSize ());
1314
1315          //double snrtest1 = CalculateSnr (WifiPhy::GetOfdmRate300KbpsBW1MHz (), 0.1);
              ↪  //test
1316          //double snrtest2 = CalculateSnr (WifiPhy::GetOfdmRate300KbpsBW1MHz (),
              ↪  5.07867e-11); //test
1317          //double snrtest3 = CalculateSnr (WifiPhy::GetOfdmRate300KbpsBW1MHz (), 0); //test
1318          //NS_LOG_UNCOND ("1153--YansWifiPhy::EndReceive, snrtest1=" << snrtest1 << ",
              ↪  snrtest2=" << snrtest2 << ", snrtest3=" << snrtest3);
1319
1320         // NS_LOG_UNCOND ("YansWifiPhy::EndReceive, mode=" << (event->GetPayloadMode
          ↪  ().GetDataRate ()) << ", snr=" << snrPer.snr << ", per=" << snrPer.per << ",
          ↪  size=" << packet->GetSize () << "," << packet);
1321
1322        if (random > snrPer.per)
1323          {
1324            NotifyRxEnd (packet);
1325            uint32_t dataRate500KbpsUnits;
1326            if ((event->GetPayloadMode ().GetModulationClass () == WIFI_MOD_CLASS_HT) ||
                ↪  (event->GetPayloadMode ().GetModulationClass () == WIFI_MOD_CLASS_S1G))
1327              {
1328                dataRate500KbpsUnits = 128 + WifiModeToMcs (event->GetPayloadMode ());
```

```
1329            }
1330          else
1331            {
1332              dataRate500KbpsUnits = event->GetPayloadMode ().GetDataRate () *
                  ↪  event->GetTxVector ().GetNss () / 500000;
1333            }
1334          bool isShortPreamble = (WIFI_PREAMBLE_SHORT == event->GetPreambleType ());
1335          double signalDbm = RatioToDb (event->GetRxPowerW ()) + 30;
1336          double noiseDbm = RatioToDb (event->GetRxPowerW () / snrPer.snr) -
              ↪  GetRxNoiseFigure () + 30;
1337          NotifyMonitorSniffRx (packet, (uint16_t)GetChannelFrequencyMhz (),
              ↪  GetChannelNumber (), dataRate500KbpsUnits, isShortPreamble,
              ↪  event->GetTxVector (), signalDbm, noiseDbm);
1338          m_state->SwitchFromRxEndOk (packet, snrPer.snr, event->GetTxVector (),
              ↪  event->GetPreambleType ());
1339
1340          //NS_LOG_UNCOND ("YansWifiPhy::EndReceive, SwitchFromRxEndOk, "  << packet);
1341        }
1342      else
1343        {
1344          /* failure. */
1345          //NS_LOG_UNCOND ("YansWifiPhy::EndReceive-reason1");
1346          NotifyRxDrop (packet, PhyRxNotEnoughSNIR);
1347          m_state->SwitchFromRxEndError (packet, snrPer.snr);
1348        }
1349    }
1350  else
1351    {
1352      //notify rx end
1353      //NS_LOG_UNCOND ("YansWifiPhy::EndReceive-reason2");
1354      m_state->SwitchFromRxEndError (packet, snrPer.snr);
1355    }
1356
1357  if (preamble == WIFI_PREAMBLE_NONE && packetType == 2)
1358    {
1359      m_plcpSuccess = false;
1360    }
1361}
1362
1363int64_t
1364YansWifiPhy::AssignStreams (int64_t stream)
1365{
1366  NS_LOG_FUNCTION (this << stream);
1367  m_random->SetStream (stream);
1368  return 1;
1369}
1370
1371void
```

```
1372 YansWifiPhy::SetFrequency (uint32_t freq)
1373 {
1374   m_channelStartingFrequency = freq;
1375 }
1376
1377 void
1378 YansWifiPhy::SetNumberOfTransmitAntennas (uint32_t tx)
1379 {
1380   m_numberOfTransmitters = tx;
1381 }
1382
1383 void
1384 YansWifiPhy::SetNumberOfReceiveAntennas (uint32_t rx)
1385 {
1386   m_numberOfReceivers = rx;
1387 }
1388
1389 void
1390 YansWifiPhy::SetLdpc (bool Ldpc)
1391 {
1392   m_ldpc = Ldpc;
1393 }
1394
1395 void
1396 YansWifiPhy::SetStbc (bool stbc)
1397 {
1398   m_stbc = stbc;
1399 }
1400
1401 void
1402 YansWifiPhy::SetGreenfield (bool greenfield)
1403 {
1404   m_greenfield = greenfield;
1405 }
1406
1407 void
1408 YansWifiPhy::SetS1g1Mfield (bool s1g1mfield)
1409 {
1410   m_s1g1mfield = s1g1mfield;
1411 }
1412
1413 void
1414 YansWifiPhy::SetS1gShortfield (bool s1gshortfield)
1415 {
1416   m_s1gshortfield = s1gshortfield;
1417 }
1418
1419 void
```

```
1420 YansWifiPhy::SetS1gLongfield (bool s1glongfield)
1421 {
1422   m_s1glongfield = s1glongfield;
1423 }
1424
1425 bool
1426 YansWifiPhy::GetGuardInterval (void) const
1427 {
1428   return m_guardInterval;
1429 }
1430
1431 void
1432 YansWifiPhy::SetGuardInterval (bool guardInterval)
1433 {
1434   m_guardInterval = guardInterval;
1435 }
1436
1437 uint32_t
1438 YansWifiPhy::GetFrequency (void) const
1439 {
1440   return m_channelStartingFrequency;
1441 }
1442
1443 uint32_t
1444 YansWifiPhy::GetNumberOfTransmitAntennas (void) const
1445 {
1446   return m_numberOfTransmitters;
1447 }
1448
1449 uint32_t
1450 YansWifiPhy::GetNumberOfReceiveAntennas (void) const
1451 {
1452   return m_numberOfReceivers;
1453 }
1454
1455 bool
1456 YansWifiPhy::GetLdpc (void) const
1457 {
1458   return m_ldpc;
1459 }
1460
1461 bool
1462 YansWifiPhy::GetStbc (void) const
1463 {
1464   return m_stbc;
1465 }
1466
1467 bool
```

```
1468 YansWifiPhy::GetGreenfield (void) const
1469 {
1470   return m_greenfield;
1471 }
1472
1473 bool
1474 YansWifiPhy::GetS1g1Mfield (void) const
1475 {
1476     return m_s1g1mfield;
1477 }
1478
1479 bool
1480 YansWifiPhy::GetS1gShortfield (void) const
1481 {
1482     return m_s1gshortfield;
1483 }
1484
1485 bool
1486 YansWifiPhy::GetS1gLongfield (void) const
1487 {
1488     return m_s1glongfield;
1489 }
1490
1491 void
1492 YansWifiPhy::SetChannelWidth(uint32_t channelwidth)
1493 {
1494     NS_ASSERT_MSG (channelwidth == 1 || channelwidth == 2 || channelwidth == 4|
      ↪  channelwidth == 8| channelwidth ==16|| channelwidth == 20 || channelwidth == 40 ||
      ↪  channelwidth == 80 || channelwidth == 160, "wrong channel width value");
1495     m_channelWidth = channelwidth;
1496 }
1497
1498 uint32_t
1499 YansWifiPhy::GetChannelWidth(void) const
1500 {
1501     return m_channelWidth;
1502 }
1503
1504 uint32_t
1505 YansWifiPhy::GetNBssMembershipSelectors (void) const
1506 {
1507   return m_bssMembershipSelectorSet.size ();
1508 }
1509
1510 uint32_t
1511 YansWifiPhy::GetBssMembershipSelector (uint32_t selector) const
1512 {
1513   return m_bssMembershipSelectorSet[selector];
```

```
1514 }
1515
1516 WifiModeList
1517 YansWifiPhy::GetMembershipSelectorModes (uint32_t selector)
1518 {
1519   uint32_t id = GetBssMembershipSelector (selector);
1520     //NS_LOG_UNCOND ("YansWifiPhy id " << id);
1521   WifiModeList supportedmodes;
1522   if (id == HT_PHY)
1523     {
1524       //mandatory MCS 0 to 7
1525       supportedmodes.push_back (WifiPhy::GetOfdmRate6_5MbpsBW20MHz ());
1526       supportedmodes.push_back (WifiPhy::GetOfdmRate13MbpsBW20MHz ());
1527       supportedmodes.push_back (WifiPhy::GetOfdmRate19_5MbpsBW20MHz ());
1528       supportedmodes.push_back (WifiPhy::GetOfdmRate26MbpsBW20MHz ());
1529       supportedmodes.push_back (WifiPhy::GetOfdmRate39MbpsBW20MHz ());
1530       supportedmodes.push_back (WifiPhy::GetOfdmRate52MbpsBW20MHz ());
1531       supportedmodes.push_back (WifiPhy::GetOfdmRate58_5MbpsBW20MHz ());
1532       supportedmodes.push_back (WifiPhy::GetOfdmRate65MbpsBW20MHz ());
1533     }
1534   if (id == S1G_PHY)
1535     {
1536       //mandatory MCS 0 to 7, 1Mhz
1537       supportedmodes.push_back (WifiPhy::GetOfdmRate300KbpsBW1MHz ());
1538       supportedmodes.push_back (WifiPhy::GetOfdmRate600KbpsBW1MHz ());
1539       supportedmodes.push_back (WifiPhy::GetOfdmRate900KbpsBW1MHz ());
1540       supportedmodes.push_back (WifiPhy::GetOfdmRate1_2MbpsBW1MHz ());
1541       supportedmodes.push_back (WifiPhy::GetOfdmRate1_8MbpsBW1MHz ());
1542       supportedmodes.push_back (WifiPhy::GetOfdmRate2_4MbpsBW1MHz ());
1543       supportedmodes.push_back (WifiPhy::GetOfdmRate2_7MbpsBW1MHz ());
1544       supportedmodes.push_back (WifiPhy::GetOfdmRate3MbpsBW1MHz ());
1545       supportedmodes.push_back (WifiPhy::GetOfdmRate150KbpsBW1MHz ());
1546     //mandatory MCS 0 to 7, 2Mhz
1547       supportedmodes.push_back (WifiPhy::GetOfdmRate650KbpsBW2MHz ());
1548       supportedmodes.push_back (WifiPhy::GetOfdmRate1_3MbpsBW2MHz ());
1549       supportedmodes.push_back (WifiPhy::GetOfdmRate1_95MbpsBW2MHz ());
1550       supportedmodes.push_back (WifiPhy::GetOfdmRate2_6MbpsBW2MHz ());
1551       supportedmodes.push_back (WifiPhy::GetOfdmRate3_9MbpsBW2MHz ());
1552       supportedmodes.push_back (WifiPhy::GetOfdmRate5_2MbpsBW2MHz ());
1553       supportedmodes.push_back (WifiPhy::GetOfdmRate5_85MbpsBW2MHz ());
1554       supportedmodes.push_back (WifiPhy::GetOfdmRate6_5MbpsBW2MHz ());
1555     }
1556
1557   return supportedmodes;
1558 }
1559
1560 uint8_t
1561 YansWifiPhy::GetNMcs (void) const
```

```
1562 {
1563   return m_deviceMcsSet.size ();
1564 }
1565
1566 uint8_t
1567 YansWifiPhy::GetMcs (uint8_t mcs) const
1568 {
1569   return m_deviceMcsSet[mcs];
1570 }
1571
1572 uint32_t
1573 YansWifiPhy::WifiModeToMcs (WifiMode mode)
1574 {
1575     uint32_t mcs = 0;
1576     if (mode.GetUniqueName() == "OfdmRate5_85MbpsBW16MHz" || mode.GetUniqueName() ==
     ↪  "OfdmRate6_5MbpsBW16MHz" )
1577     {
1578         mcs = 0;
1579     }
1580     else if (mode.GetUniqueName() == "OfdmRate3MbpsBW4MHz" || mode.GetUniqueName() ==
     ↪  "OfdmRate5_85MbpsBW8MHz" )
1581     {
1582         mcs = 1;
1583     }
1584     else if (mode.GetUniqueName() == "OfdmRate17_55MbpsBW16MHz" )
1585     {
1586         mcs =2;
1587     }
1588     else if (mode.GetUniqueName() == "OfdmRate11_7MbpsBW8MHz" || mode.GetUniqueName() ==
     ↪  "OfdmRate13MbpsBW8MHz"  || mode.GetUniqueName() == "OfdmRate23_4MbpsBW16MHz" ||
     ↪  mode.GetUniqueName() == "OfdmRate26MbpsBW16MHz")
1589     {
1590         mcs = 3;
1591     }
1592     else if (mode.GetUniqueName() == "OfdmRate19_5MbpsBW8MHz" || mode.GetUniqueName() ==
     ↪  "OfdmRate35_1MbpsBW16MHz"  || mode.GetUniqueName() == "OfdmRate39MbpsBW16MHz" )
1593     {
1594         mcs = 4;
1595     }
1596     else if (mode.GetUniqueName() == "OfdmRate2_7MbpsBW1MHz" || mode.GetUniqueName() ==
     ↪  "OfdmRate3MbpsBW1MHzShGi" || mode.GetUniqueName() == "OfdmRate6_5MbpsBW2MHzShGi" ||
     ↪  mode.GetUniqueName() == "OfdmRate13_5MbpsBW4MHzShGi" || mode.GetUniqueName() ==
     ↪  "OfdmRate29_25MbpsBW8MHzShGi" ||
1597              mode.GetUniqueName() == "OfdmRate58_5MbpsBW16MHzShGi" || mode.GetUniqueName()
                  ↪  == "OfdmRate135MbpsBW40MHzShGi" || mode.GetUniqueName() ==
                  ↪  "OfdmRate65MbpsBW20MHzShGi" )
1598     {
1599         mcs = 6;
```

```
1600        }
1601        else if (mode.GetUniqueName() == "OfdmRate6_5MbpsBW2MHz" )
1602        {
1603            mcs = 7;
1604        }
1605        else if (mode.GetUniqueName() == "OfdmRate4MbpsBW1MHzShGi" || mode.GetUniqueName() ==
            ↪ "OfdmRate18MbpsBW4MHzShGi" || mode.GetUniqueName() == "OfdmRate39MbpsBW8MHzShGi" ||
            ↪ mode.GetUniqueName() == "OfdmRate78MbpsBW16MHzShGi")
1606        {
1607            mcs = 8;
1608        }
1609        else if (mode.GetModulationClass() == WIFI_MOD_CLASS_S1G )
1610        {
1611            switch (mode.GetDataRate ())
1612            {
1613                case 300000:
1614                case 333300:
1615                case 650000:
1616                case 722200:
1617                case 1350000:
1618                case 1500000:
1619                case 2925000:
1620                case 3250000:
1621                    mcs = 0;
1622                    break;
1623                case 600000:
1624                case 666700:
1625                case 1300000:
1626                case 1444400:
1627                case 2700000:
1628                case 6500000:
1629                case 11700000:
1630                case 13000000:
1631                    mcs = 1;
1632                    break;
1633                case 900000:
1634                case 1000000:
1635                case 1950000:
1636                case 2166700:
1637                case 4050000:
1638                case 4500000:
1639                case 8775000:
1640                case 9750000:
1641                case 19500000:
1642                    mcs=2;
1643                    break;
1644                case 1200000:
1645                case 1333300:
```

```
1646            case 2600000:
1647            case 2888900:
1648            case 5400000:
1649            case 6000000:
1650                mcs=3;
1651                break;
1652            case 1800000:
1653            case 2000000:
1654            case 3900000:
1655            case 4333300:
1656            case 8100000:
1657            case 9000000:
1658            case 17550000:
1659                mcs = 4;
1660                break;
1661            case 2400000:
1662            case 2666700:
1663            case 5200000:
1664            case 5777800:
1665            case 10800000:
1666            case 12000000:
1667            case 23400000:
1668            case 26000000:
1669            case 46800000:
1670            case 52000000:
1671                mcs=5;
1672                break;
1673            case 5850000:
1674            case 12150000:
1675            case 26325000:
1676            case 52650000:
1677                mcs=6;
1678                break;
1679            case 3000000:
1680            case 3333300:
1681            case 7222200:
1682            case 13500000:
1683            case 15000000:
1684            case 29250000:
1685            case 32500000:
1686            case 58500000:
1687            case 65000000:
1688                mcs=7;
1689                break;
1690            case 3600000:
1691            case 7800000:
1692            case 8666700:
1693            case 16200000:
```

```
1694            case 35100000:
1695            case 70200000:
1696                mcs=8;
1697                break;
1698            case 4000000:
1699            case 4444400:
1700            case 18000000:
1701            case 20000000:
1702            case 39000000:
1703            case 43333300:
1704            case 78000000:
1705            case 86666700:
1706                mcs=9;
1707                break;
1708            case 150000:
1709            case 166700:
1710                mcs=10;
1711                break;
1712        }
1713    }
1714    else
1715    {
1716        switch (mode.GetDataRate())
1717        {
1718            case 6500000:
1719            case 7200000:
1720            case 13500000:
1721            case 15000000:
1722                mcs=0;
1723                break;
1724            case 13000000:
1725            case 14400000:
1726            case 27000000:
1727            case 30000000:
1728                mcs=1;
1729                break;
1730            case 19500000:
1731            case 21700000:
1732            case 40500000:
1733            case 45000000:
1734                mcs=2;
1735                break;
1736            case 26000000:
1737            case 28900000:
1738            case 54000000:
1739            case 60000000:
1740                mcs=3;
1741                break;
```

```
1742            case 39000000:
1743            case 43300000:
1744            case 81000000:
1745            case 90000000:
1746                mcs=4;
1747                break;
1748            case 52000000:
1749            case 57800000:
1750            case 108000000:
1751            case 120000000:
1752                mcs=5;
1753                break;
1754            case 58500000:
1755            case 121500000:
1756                mcs=6;
1757                break;
1758            case 65000000:
1759            case 72200000:
1760            case 135000000:
1761            case 150000000:
1762                mcs=7;
1763                break;
1764        }
1765    }
1766    return mcs;
1767 }
1768
1769
1770 WifiMode
1771 YansWifiPhy::McsToWifiMode (uint8_t mcs)
1772 {
1773    //NS_LOG_UNCOND ("YansWifiPhy::McsToWifiMode (uint8_t mcs) " << mcs);
1774    WifiMode mode;
1775    switch (mcs)
1776    {
1777        case 10:
1778            if (!GetGuardInterval() && GetChannelWidth() == 1)
1779            {
1780                mode = WifiPhy::GetOfdmRate150KbpsBW1MHz ();
1781            }
1782            else
1783            {
1784                mode = WifiPhy::GetOfdmRate166_7KbpsBW1MHz ();
1785            }
1786            break;
1787        case 9:
1788
1789            if (!GetGuardInterval() && GetChannelWidth() == 1)
```

```
1790            {
1791                mode = WifiPhy::GetOfdmRate4MbpsBW1MHz ();
1792            }
1793            else if (GetGuardInterval() && GetChannelWidth() == 1)
1794            {
1795                mode = WifiPhy::GetOfdmRate4_444_4MbpsBW1MHz ();
1796            }
1797            else if (!GetGuardInterval() && GetChannelWidth() == 4)
1798            {
1799                mode = WifiPhy::GetOfdmRate18MbpsBW4MHz ();
1800            }
1801            else if (GetGuardInterval() && GetChannelWidth() == 4)
1802            {
1803                mode = WifiPhy::GetOfdmRate20MbpsBW4MHz ();
1804            }
1805            else if (!GetGuardInterval() && GetChannelWidth() == 8)
1806            {
1807                mode = WifiPhy::GetOfdmRate39MbpsBW8MHz ();
1808            }
1809            else if (GetGuardInterval() && GetChannelWidth() == 8)
1810            {
1811                mode = WifiPhy::GetOfdmRate43_333_3MbpsBW8MHz ();
1812            }
1813            else if (!GetGuardInterval() && GetChannelWidth() == 16)
1814            {
1815                mode = WifiPhy::GetOfdmRate78MbpsBW16MHz ();
1816            }
1817            else
1818            {
1819                mode = WifiPhy::GetOfdmRate86_666_7MbpsBW16MHz ();
1820            }
1821            break;
1822        case 8:
1823            if (!GetGuardInterval() && GetChannelWidth() == 1)
1824            {
1825                mode = WifiPhy::GetOfdmRate3_6MbpsBW1MHz ();
1826            }
1827            else if (GetGuardInterval() && GetChannelWidth() == 1)
1828            {
1829                mode = WifiPhy::GetOfdmRate4MbpsBW1MHzShGi ();
1830            }
1831            else if (!GetGuardInterval() && GetChannelWidth() == 2)
1832            {
1833                mode = WifiPhy::GetOfdmRate7_8MbpsBW2MHz ();
1834            }
1835            else if (GetGuardInterval() && GetChannelWidth() == 2)
1836            {
1837                mode = WifiPhy::GetOfdmRate8_666_7MbpsBW2MHz ();
```

```
1838            }
1839            else if (!GetGuardInterval() && GetChannelWidth() == 4)
1840            {
1841                mode = WifiPhy::GetOfdmRate16_2MbpsBW4MHz ();
1842            }
1843            else if (GetGuardInterval() && GetChannelWidth() == 4)
1844            {
1845                mode = WifiPhy::GetOfdmRate18MbpsBW4MHzShGi ();
1846            }
1847            else if (!GetGuardInterval() && GetChannelWidth() == 8)
1848            {
1849                mode = WifiPhy::GetOfdmRate35_1MbpsBW8MHz ();
1850            }
1851            else if (GetGuardInterval() && GetChannelWidth() == 8)
1852            {
1853                mode = WifiPhy::GetOfdmRate39MbpsBW8MHzShGi ();
1854            }
1855            else if (!GetGuardInterval() && GetChannelWidth() == 16)
1856            {
1857                mode = WifiPhy::GetOfdmRate70_2MbpsBW16MHz ();
1858            }
1859            else if (GetGuardInterval() && GetChannelWidth() == 16)
1860            {
1861                mode = WifiPhy::GetOfdmRate78MbpsBW16MHzShGi ();
1862            }
1863            break;
1864        case 7:
1865            if (!GetGuardInterval() && GetChannelWidth() == 1)
1866            {
1867                mode = WifiPhy::GetOfdmRate3MbpsBW1MHz ();
1868            }
1869            else if (GetGuardInterval() && GetChannelWidth() == 1)
1870            {
1871                mode = WifiPhy::GetOfdmRate3_333_3MbpsBW1MHz ();
1872            }
1873            else if (!GetGuardInterval() && GetChannelWidth() == 2)
1874            {
1875                mode = WifiPhy::GetOfdmRate6_5MbpsBW2MHz ();
1876            }
1877            else if (GetGuardInterval() && GetChannelWidth() == 2)
1878            {
1879                mode = WifiPhy::GetOfdmRate7_222_2MbpsBW2MHz ();
1880            }
1881            else if (!GetGuardInterval() && GetChannelWidth() == 4)
1882            {
1883                mode = WifiPhy::GetOfdmRate13_5MbpsBW4MHz ();
1884            }
1885            else if (GetGuardInterval() && GetChannelWidth() == 4)
```

```
1886            {
1887                mode = WifiPhy::GetOfdmRate15MbpsBW4MHz ();
1888            }
1889            else if (!GetGuardInterval() && GetChannelWidth() == 8)
1890            {
1891                mode = WifiPhy::GetOfdmRate29_25MbpsBW8MHz ();
1892            }
1893            else if (GetGuardInterval() && GetChannelWidth() == 8)
1894            {
1895                mode = WifiPhy::GetOfdmRate32_5MbpsBW8MHz ();
1896            }
1897            else if (!GetGuardInterval() && GetChannelWidth() == 16)
1898            {
1899                mode = WifiPhy::GetOfdmRate58_5MbpsBW16MHz ();
1900            }
1901            else if (GetGuardInterval() && GetChannelWidth() == 16)
1902            {
1903                mode = WifiPhy::GetOfdmRate65MbpsBW16MHz ();
1904            }
1905            else if (!GetGuardInterval() && GetChannelWidth() == 20)
1906            {
1907                mode =  WifiPhy::GetOfdmRate65MbpsBW20MHz ();
1908            }
1909            else if(GetGuardInterval() && GetChannelWidth() == 20)
1910            {
1911                mode = WifiPhy::GetOfdmRate72_2MbpsBW20MHz ();
1912            }
1913            else if (!GetGuardInterval() && GetChannelWidth() == 40)
1914            {
1915                mode = WifiPhy::GetOfdmRate135MbpsBW40MHz ();
1916            }
1917            else
1918            {
1919                mode = WifiPhy::GetOfdmRate150MbpsBW40MHz ();
1920            }
1921            break;
1922        case 6:
1923            if (!GetGuardInterval() && GetChannelWidth() == 1)
1924            {
1925                mode = WifiPhy::GetOfdmRate2_7MbpsBW1MHz ();
1926            }
1927            else if (GetGuardInterval() && GetChannelWidth() == 1)
1928            {
1929                mode = WifiPhy::GetOfdmRate3MbpsBW1MHzShGi ();
1930            }
1931            else if (!GetGuardInterval() && GetChannelWidth() == 2)
1932            {
1933                mode = WifiPhy::GetOfdmRate5_85MbpsBW2MHz ();
```

```
1934            }
1935            else if (GetGuardInterval() && GetChannelWidth() == 2)
1936            {
1937                mode = WifiPhy::GetOfdmRate6_5MbpsBW2MHzShGi ();
1938            }
1939            else if (!GetGuardInterval() && GetChannelWidth() == 4)
1940            {
1941                mode = WifiPhy::GetOfdmRate12_15MbpsBW4MHz ();
1942            }
1943            else if (GetGuardInterval() && GetChannelWidth() == 4)
1944            {
1945                mode = WifiPhy::GetOfdmRate13_5MbpsBW4MHzShGi ();
1946            }
1947            else if (!GetGuardInterval() && GetChannelWidth() == 8)
1948            {
1949                mode = WifiPhy::GetOfdmRate26_325MbpsBW8MHz ();
1950            }
1951            else if (GetGuardInterval() && GetChannelWidth() == 8)
1952            {
1953                mode = WifiPhy::GetOfdmRate29_25MbpsBW8MHzShGi ();
1954            }
1955            else if (!GetGuardInterval() && GetChannelWidth() == 16)
1956            {
1957                mode = WifiPhy::GetOfdmRate52_65MbpsBW16MHz ();
1958            }
1959            else if (GetGuardInterval() && GetChannelWidth() == 16)
1960            {
1961                mode = WifiPhy::GetOfdmRate58_5MbpsBW16MHzShGi ();
1962            }
1963            else if (!GetGuardInterval() && GetChannelWidth() == 20)
1964            {
1965                mode = WifiPhy::GetOfdmRate58_5MbpsBW20MHz ();
1966
1967            }
1968            else if(GetGuardInterval() && GetChannelWidth() == 20)
1969            {
1970                mode =  WifiPhy::GetOfdmRate65MbpsBW20MHzShGi ();
1971
1972            }
1973            else if (!GetGuardInterval() && GetChannelWidth() == 40)
1974            {
1975                mode = WifiPhy::GetOfdmRate121_5MbpsBW40MHz ();
1976
1977            }
1978            else
1979            {
1980                mode= WifiPhy::GetOfdmRate135MbpsBW40MHzShGi ();
1981
```

```
1982            }
1983            break;
1984        case 5:
1985            if (!GetGuardInterval() && GetChannelWidth() == 1)
1986            {
1987                mode = WifiPhy::GetOfdmRate2_4MbpsBW1MHz ();
1988            }
1989            else if (GetGuardInterval() && GetChannelWidth() == 1)
1990            {
1991                mode = WifiPhy::GetOfdmRate2_666_7MbpsBW1MHz ();
1992            }
1993            else if (!GetGuardInterval() && GetChannelWidth() == 2)
1994            {
1995                mode = WifiPhy::GetOfdmRate5_2MbpsBW2MHz ();
1996            }
1997            else if (GetGuardInterval() && GetChannelWidth() == 2)
1998            {
1999                mode = WifiPhy::GetOfdmRate5_777_8MbpsBW2MHz ();
2000            }
2001            else if (!GetGuardInterval() && GetChannelWidth() == 4)
2002            {
2003                mode = WifiPhy::GetOfdmRate10_8MbpsBW4MHz ();
2004            }
2005            else if (GetGuardInterval() && GetChannelWidth() == 4)
2006            {
2007                mode = WifiPhy::GetOfdmRate12MbpsBW4MHz ();
2008            }
2009            else if (!GetGuardInterval() && GetChannelWidth() == 8)
2010            {
2011                mode = WifiPhy::GetOfdmRate23_4MbpsBW8MHz ();
2012            }
2013            else if (GetGuardInterval() && GetChannelWidth() == 8)
2014            {
2015                mode = WifiPhy::GetOfdmRate26MbpsBW8MHz ();
2016            }
2017            else if (!GetGuardInterval() && GetChannelWidth() == 16)
2018            {
2019                mode = WifiPhy::GetOfdmRate46_8MbpsBW16MHz ();
2020            }
2021            else if (GetGuardInterval() && GetChannelWidth() == 16)
2022            {
2023                mode = WifiPhy::GetOfdmRate52MbpsBW16MHz ();
2024            }
2025            else if (!GetGuardInterval() && GetChannelWidth() == 20)
2026            {
2027                mode = WifiPhy::GetOfdmRate52MbpsBW20MHz ();
2028
2029            }
```

```
2030            else if(GetGuardInterval() && GetChannelWidth() == 20)
2031            {
2032                mode = WifiPhy::GetOfdmRate57_8MbpsBW20MHz ();
2033            }
2034            else if (!GetGuardInterval() && GetChannelWidth() == 40)
2035            {
2036                mode = WifiPhy::GetOfdmRate108MbpsBW40MHz ();
2037
2038            }
2039            else
2040            {
2041                mode = WifiPhy::GetOfdmRate120MbpsBW40MHz ();
2042
2043            }
2044            break;
2045        case 4:
2046            if (!GetGuardInterval() && GetChannelWidth() == 1)
2047            {
2048                mode = WifiPhy::GetOfdmRate1_8MbpsBW1MHz ();
2049            }
2050            else if (GetGuardInterval() && GetChannelWidth() == 1)
2051            {
2052                mode = WifiPhy::GetOfdmRate2MbpsBW1MHz ();
2053            }
2054            else if (!GetGuardInterval() && GetChannelWidth() == 2)
2055            {
2056                mode = WifiPhy::GetOfdmRate3_9MbpsBW2MHz ();
2057            }
2058            else if (GetGuardInterval() && GetChannelWidth() == 2)
2059            {
2060                mode = WifiPhy::GetOfdmRate4_333_3MbpsBW2MHz ();
2061            }
2062            else if (!GetGuardInterval() && GetChannelWidth() == 4)
2063            {
2064                mode = WifiPhy::GetOfdmRate8_1MbpsBW4MHz ();
2065            }
2066            else if (GetGuardInterval() && GetChannelWidth() == 4)
2067            {
2068                mode = WifiPhy::GetOfdmRate9MbpsBW4MHz ();
2069            }
2070            else if (!GetGuardInterval() && GetChannelWidth() == 8)
2071            {
2072                mode = WifiPhy::GetOfdmRate17_55MbpsBW8MHz ();
2073            }
2074            else if (GetGuardInterval() && GetChannelWidth() == 8)
2075            {
2076                mode = WifiPhy::GetOfdmRate19_5MbpsBW8MHz ();
2077            }
```

```
2078            else if (!GetGuardInterval() && GetChannelWidth() == 16)
2079            {
2080                mode = WifiPhy::GetOfdmRate35_1MbpsBW16MHz ();
2081            }
2082            else if (GetGuardInterval() && GetChannelWidth() == 16)
2083            {
2084                mode = WifiPhy::GetOfdmRate39MbpsBW16MHz ();
2085            }
2086            else if (!GetGuardInterval() && GetChannelWidth() == 20)
2087            {
2088                mode = WifiPhy::GetOfdmRate39MbpsBW20MHz ();
2089            }
2090            else if(GetGuardInterval() && GetChannelWidth() == 20)
2091            {
2092                mode = WifiPhy::GetOfdmRate43_3MbpsBW20MHz ();
2093            }
2094            else if (!GetGuardInterval() && GetChannelWidth() == 40)
2095            {
2096                mode = WifiPhy::GetOfdmRate81MbpsBW40MHz ();
2097
2098            }
2099            else
2100            {
2101                mode = WifiPhy::GetOfdmRate90MbpsBW40MHz ();
2102
2103            }
2104            break;
2105        case 3:
2106            if (!GetGuardInterval() && GetChannelWidth() == 1)
2107            {
2108                mode = WifiPhy::GetOfdmRate1_2MbpsBW1MHz ();
2109            }
2110            else if (GetGuardInterval() && GetChannelWidth() == 1)
2111            {
2112                mode = WifiPhy::GetOfdmRate1_333_3MbpsBW1MHz ();
2113            }
2114            else if (!GetGuardInterval() && GetChannelWidth() == 2)
2115            {
2116                mode = WifiPhy::GetOfdmRate2_6MbpsBW2MHz ();
2117            }
2118            else if (GetGuardInterval() && GetChannelWidth() == 2)
2119            {
2120                mode = WifiPhy::GetOfdmRate2_8889MbpsBW2MHz ();
2121            }
2122            else if (!GetGuardInterval() && GetChannelWidth() == 4)
2123            {
2124                mode = WifiPhy::GetOfdmRate5_4MbpsBW4MHz ();
2125            }
```

```
2126            else if (GetGuardInterval() && GetChannelWidth() == 4)
2127            {
2128                mode = WifiPhy::GetOfdmRate6MbpsBW4MHz ();
2129            }
2130            else if (!GetGuardInterval() && GetChannelWidth() == 8)
2131            {
2132                mode = WifiPhy::GetOfdmRate11_7MbpsBW8MHz ();
2133            }
2134            else if (GetGuardInterval() && GetChannelWidth() == 8)
2135            {
2136                mode = WifiPhy::GetOfdmRate13MbpsBW8MHz ();
2137            }
2138            else if (!GetGuardInterval() && GetChannelWidth() == 16)
2139            {
2140                mode = WifiPhy::GetOfdmRate23_4MbpsBW16MHz ();
2141            }
2142            else if (GetGuardInterval() && GetChannelWidth() == 16)
2143            {
2144                mode = WifiPhy::GetOfdmRate26MbpsBW16MHz ();
2145            }
2146            else if (!GetGuardInterval() && GetChannelWidth() == 20)
2147            {
2148                mode =  WifiPhy::GetOfdmRate26MbpsBW20MHz ();
2149
2150            }
2151            else if(GetGuardInterval() && GetChannelWidth() == 20)
2152            {
2153                mode = WifiPhy::GetOfdmRate28_9MbpsBW20MHz ();
2154
2155            }
2156            else if (!GetGuardInterval() && GetChannelWidth() == 40)
2157            {
2158                mode = WifiPhy::GetOfdmRate54MbpsBW40MHz ();
2159
2160            }
2161            else
2162            {
2163                mode = WifiPhy::GetOfdmRate60MbpsBW40MHz ();
2164            }
2165            break;
2166        case 2:
2167            if (!GetGuardInterval() && GetChannelWidth() == 1)
2168            {
2169                mode = WifiPhy::GetOfdmRate900KbpsBW1MHz ();
2170            }
2171            else if (GetGuardInterval() && GetChannelWidth() == 1)
2172            {
2173                mode = WifiPhy::GetOfdmRate1MbpsBW1MHz ();
```

```
2174            }
2175            else if (!GetGuardInterval() && GetChannelWidth() == 2)
2176            {
2177                mode = WifiPhy::GetOfdmRate1_95MbpsBW2MHz ();
2178            }
2179            else if (GetGuardInterval() && GetChannelWidth() == 2)
2180            {
2181                mode = WifiPhy::GetOfdmRate2_166_7MbpsBW2MHz ();
2182            }
2183            else if (!GetGuardInterval() && GetChannelWidth() == 4)
2184            {
2185                mode = WifiPhy::GetOfdmRate4_05MbpsBW4MHz ();
2186            }
2187            else if (GetGuardInterval() && GetChannelWidth() == 4)
2188            {
2189                mode = WifiPhy::GetOfdmRate4_5MbpsBW4MHz ();
2190            }
2191            else if (!GetGuardInterval() && GetChannelWidth() == 8)
2192            {
2193                mode = WifiPhy::GetOfdmRate8_775MbpsBW8MHz ();
2194            }
2195            else if (GetGuardInterval() && GetChannelWidth() == 8)
2196            {
2197                mode = WifiPhy::GetOfdmRate9_75MbpsBW8MHz ();
2198            }
2199            else if (!GetGuardInterval() && GetChannelWidth() == 16)
2200            {
2201                mode = WifiPhy::GetOfdmRate17_55MbpsBW16MHz ();
2202            }
2203            else if (GetGuardInterval() && GetChannelWidth() == 16)
2204            {
2205                mode = WifiPhy::GetOfdmRate19_5MbpsBW16MHz ();
2206            }
2207            else if (!GetGuardInterval() && GetChannelWidth() == 20)
2208            {
2209                mode = WifiPhy::GetOfdmRate19_5MbpsBW20MHz ();
2210
2211            }
2212            else if(GetGuardInterval() && GetChannelWidth() == 20)
2213            {
2214                mode = WifiPhy::GetOfdmRate21_7MbpsBW20MHz ();
2215
2216            }
2217            else if (!GetGuardInterval() && GetChannelWidth() == 40)
2218            {
2219                mode =  WifiPhy::GetOfdmRate40_5MbpsBW40MHz ();
2220
2221            }
```

```
2222            else
2223            {
2224                mode = WifiPhy::GetOfdmRate45MbpsBW40MHz ();
2225
2226            }
2227            break;
2228        case 1:
2229            if (!GetGuardInterval() && GetChannelWidth() == 1)
2230            {
2231                mode = WifiPhy::GetOfdmRate600KbpsBW1MHz ();
2232            }
2233            else if (GetGuardInterval() && GetChannelWidth() == 1)
2234            {
2235                mode = WifiPhy::GetOfdmRate666_7KbpsBW1MHz ();
2236            }
2237            else if (!GetGuardInterval() && GetChannelWidth() == 2)
2238            {
2239                mode = WifiPhy::GetOfdmRate1_3MbpsBW2MHz ();
2240            }
2241            else if (GetGuardInterval() && GetChannelWidth() == 2)
2242            {
2243                mode = WifiPhy::GetOfdmRate1_444_4MbpsBW2MHz ();
2244            }
2245            else if (!GetGuardInterval() && GetChannelWidth() == 4)
2246            {
2247                mode = WifiPhy::GetOfdmRate2_7MbpsBW4MHz ();
2248            }
2249            else if (GetGuardInterval() && GetChannelWidth() == 4)
2250            {
2251                mode = WifiPhy::GetOfdmRate3MbpsBW4MHz ();
2252            }
2253            else if (!GetGuardInterval() && GetChannelWidth() == 8)
2254            {
2255                mode = WifiPhy::GetOfdmRate5_85MbpsBW8MHz ();
2256            }
2257            else if (GetGuardInterval() && GetChannelWidth() == 8)
2258            {
2259                mode = WifiPhy::GetOfdmRate6_5MbpsBW8MHz ();
2260            }
2261            else if (!GetGuardInterval() && GetChannelWidth() == 16)
2262            {
2263                mode = WifiPhy::GetOfdmRate11_7MbpsBW16MHz ();
2264            }
2265            else if (GetGuardInterval() && GetChannelWidth() == 16)
2266            {
2267                mode = WifiPhy::GetOfdmRate13MbpsBW16MHz ();
2268            }
2269            else if (!GetGuardInterval() && GetChannelWidth() == 20)
```

```
2270            {
2271                mode = WifiPhy::GetOfdmRate13MbpsBW20MHz ();
2272
2273            }
2274            else if(GetGuardInterval() && GetChannelWidth() == 20)
2275            {
2276                mode =  WifiPhy::GetOfdmRate14_4MbpsBW20MHz ();
2277            }
2278            else if (!GetGuardInterval() && GetChannelWidth() == 40)
2279            {
2280                mode = WifiPhy::GetOfdmRate27MbpsBW40MHz ();
2281
2282            }
2283            else
2284            {
2285                mode = WifiPhy::GetOfdmRate30MbpsBW40MHz ();
2286            }
2287            break;
2288        case 0:
2289        default:
2290            if (!GetGuardInterval() && GetChannelWidth() == 1)
2291            {
2292                mode = WifiPhy::GetOfdmRate300KbpsBW1MHz ();
2293            }
2294            else if (GetGuardInterval() && GetChannelWidth() == 1)
2295            {
2296                mode = WifiPhy::GetOfdmRate333_3KbpsBW1MHz ();
2297            }
2298            else if (!GetGuardInterval() && GetChannelWidth() == 2)
2299            {
2300                mode = WifiPhy::GetOfdmRate650KbpsBW2MHz ();
2301            }
2302            else if (GetGuardInterval() && GetChannelWidth() == 2)
2303            {
2304                mode = WifiPhy::GetOfdmRate722_2KbpsBW2MHz ();
2305            }
2306            else if (!GetGuardInterval() && GetChannelWidth() == 4)
2307            {
2308                mode = WifiPhy::GetOfdmRate1_35MbpsBW4MHz ();
2309            }
2310            else if (GetGuardInterval() && GetChannelWidth() == 4)
2311            {
2312                mode = WifiPhy::GetOfdmRate1_5MbpsBW4MHz ();
2313            }
2314            else if (!GetGuardInterval() && GetChannelWidth() == 8)
2315            {
2316                mode = WifiPhy::GetOfdmRate2_925MbpsBW8MHz ();
2317            }
```

```
2318            else if (GetGuardInterval() && GetChannelWidth() == 8)
2319            {
2320                mode = WifiPhy::GetOfdmRate3_25MbpsBW8MHz ();
2321            }
2322            else if (!GetGuardInterval() && GetChannelWidth() == 16)
2323            {
2324                mode = WifiPhy::GetOfdmRate5_85MbpsBW16MHz ();
2325            }
2326            else if (GetGuardInterval() && GetChannelWidth() == 16)
2327            {
2328                mode = WifiPhy::GetOfdmRate6_5MbpsBW16MHz ();
2329            }
2330            else if (!GetGuardInterval() && GetChannelWidth() == 20)
2331            {
2332                mode = WifiPhy::GetOfdmRate6_5MbpsBW20MHz ();
2333
2334            }
2335            else if(GetGuardInterval() && GetChannelWidth() == 20)
2336            {
2337                mode = WifiPhy::GetOfdmRate7_2MbpsBW20MHz ();
2338            }
2339            else if (!GetGuardInterval() && GetChannelWidth() == 40)
2340            {
2341                mode = WifiPhy::GetOfdmRate13_5MbpsBW40MHz ();
2342
2343            }
2344            else
2345            {
2346                mode = WifiPhy::GetOfdmRate15MbpsBW40MHz ();
2347            }
2348            break;
2349    }
2350    return mode;
2351 }
2352
2353
2354 } //namespace ns3
```

APPENDIX F

SWIPTHARVESTER

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2023 University of Brasilia, Brasília, DF, Brazil.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * adouble with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: José Antônio de França Junior <aspect_josef@hotmail.com>
 */

#include "swipt-harvester.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("SwiptHarvester");

NS_OBJECT_ENSURE_REGISTERED (SwiptHarvester);

TypeId
SwiptHarvester::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::SwiptHarvester")
  .SetParent<EnergyHarvester> ()
  .SetGroupName ("Energy")
  .AddConstructor<SwiptHarvester> ()
  .AddAttribute ("PowerSplitFactor",
                 "The ratio of rxPowerW shared between the receiver and the harvester",
                 DoubleValue (0.5),
                 MakeDoubleAccessor (&SwiptHarvester::SetPowerSplitFactor,
```

```
                                    &SwiptHarvester::GetPowerSplitFactor),
                    MakeDoubleChecker<double> ())
    .AddAttribute ("AntennaNoise",
                    "RX antenna noise in dBm",
                    DoubleValue (-111.0), //
                    MakeDoubleAccessor (&SwiptHarvester::SetAntennaNoise,
                                        &SwiptHarvester::GetAntennaNoise),
                    MakeDoubleChecker<double> ())
    .AddAttribute ("SwiptEfficiency",
                   "AC-DC conversion eficiency from the SwiptHarvester",
                   DoubleValue (0.9),
                   MakeDoubleAccessor (&SwiptHarvester::SetSwiptEfficiency,
                                       &SwiptHarvester::GetSwiptEfficiency),
                   MakeDoubleChecker<double> ())
    .AddAttribute ("DCConversionEfficiency",
                   "DC-DC energy conversion eficiency from the harvester to the battery",
                   DoubleValue (0.95),
                   MakeDoubleAccessor (&SwiptHarvester::SetDCConversionEfficiency,
                                       &SwiptHarvester::GetDCConversionEfficiency),
                   MakeDoubleChecker<double> ())

    .AddTraceSource ("HarvestedPower",
                     "Harvested power by the SwiptHarvester.",
                     MakeTraceSourceAccessor (&SwiptHarvester::m_totalHarvestedPower),
                     "ns3::TracedValue::DoubleCallback")
    .AddTraceSource ("TotalEnergyHarvested",
                     "Total energy harvested by the SwiptHarvester.",
                     MakeTraceSourceAccessor (&SwiptHarvester::m_totalEnergyHarvestedJ),
                     "ns3::TracedValue::DoubleCallback")
    ;
    return tid;
}

class SwiptPhyListener : public WifiPhyListener
{
public:
    /**
     * Create a SwiptPhyListener for a given Harvester.
     *
     */
      SwiptPhyListener (ns3::SwiptHarvester *swiptHarvester)
      : m_swiptHarvester (swiptHarvester)
      {
          NS_LOG_FUNCTION(this);
      }
      virtual ~SwiptPhyListener ()
      {
          NS_LOG_FUNCTION(this);
```

```
87      }
88
89      virtual void NotifyRxStart (Time duration)
90      {
91        NS_LOG_FUNCTION(this << duration);
92        m_swiptHarvester->NotifyRxStartNow (duration);
93      }
94
95      virtual void NotifyRxEndOk (void)
96      {
97        NS_LOG_FUNCTION(this);
98        m_swiptHarvester->NotifyRxEndOkNow();
99      }
100
101     virtual void NotifyRxEndError (void)
102     {
103       NS_LOG_FUNCTION(this);
104       m_swiptHarvester->NotifyRxEndErrorNow();
105     }
106
107     virtual void NotifyTxStart (Time duration, double txPowerDbm)
108     {
109       NS_LOG_FUNCTION(this << duration << txPowerDbm);
110       m_swiptHarvester->NotifyTxStartNow (duration, txPowerDbm);
111     }
112
113     virtual void NotifyMaybeCcaBusyStart (Time duration)
114     {
115       NS_LOG_FUNCTION(this << duration);
116       m_swiptHarvester->NotifyMaybeCcaBusyStartNow(duration);
117     }
118
119     virtual void NotifySwitchingStart (Time duration)
120     {
121       NS_LOG_FUNCTION(this << duration);
122       m_swiptHarvester->NotifySwitchingStartNow(duration);
123     }
124
125     virtual void NotifySleep (void)
126     {
127       NS_LOG_FUNCTION(this);
128       m_swiptHarvester->NotifySleepNow ();
129     }
130
131     virtual void NotifyWakeup (void)
132     {
133       NS_LOG_FUNCTION(this);
134       m_swiptHarvester->NotifyWakeupNow();
```

```
135     }
136
137     virtual void NotifyOff (void)
138     {
139       NS_LOG_FUNCTION(this);
140       m_swiptHarvester->NotifyOffNow ();
141     }
142
143     virtual void NotifyOn (void)
144     {
145       NS_LOG_FUNCTION(this);
146       m_swiptHarvester->NotifyOnNow ();
147     }
148
149     virtual void NotifyIdle (void)
150     {
151       NS_LOG_FUNCTION(this);
152       m_swiptHarvester->NotifyIdleNow ();
153     }
154
155 private:
156     ns3::SwiptHarvester *m_swiptHarvester;
157 };
158
159 SwiptHarvester::SwiptHarvester ()
160 {
161   NS_LOG_FUNCTION (this);
162   m_rxPowerW = 0.0;
163   m_energyHarvested = 0.0;
164   m_totalEnergyHarvestedJ = 0.0;
165   m_totalHarvestedPower = 0.0;
166   m_plcpHeaderPowerHarvested = 0.0;
167   m_plcpPayloadPowerHarvested = 0.0;
168   m_harvestedPower = 0.0;
169   m_energyHarvested = 0.0;
170 }
171
172 SwiptHarvester::~SwiptHarvester ()
173 {
174   NS_LOG_FUNCTION (this);
175   m_rxPowerW = 0.0;
176   m_energyHarvested = 0.0;
177   m_totalEnergyHarvestedJ = 0.0;
178   m_totalHarvestedPower = 0.0;
179   m_plcpHeaderPowerHarvested = 0.0;
180   m_plcpPayloadPowerHarvested = 0.0;
181   m_harvestedPower = 0.0;
182   m_energyHarvested = 0.0;
```

```
}

void
SwiptHarvester::PowerWPktReceived (Ptr<const Packet> packet,
                                   double rxPowerDbm,
                                   WifiTxVector txVector,
                                   enum WifiPreamble preamble,
                                   uint8_t packetType,
                                   Time rxPktDuration,
                                   Ptr<YansWifiPhy> phyObject)
{
  Time now = Simulator::Now ();


  double rxPowerW;
  phyLayer = phyObject;


  WifiMacHeader hdr;
  packet->PeekHeader(hdr);

  if (phyObject->IsStateIdle () || phyObject->IsStateCcaBusy ())
  {
      NS_LOG_FUNCTION (this << Simulator::Now ().GetNanoSeconds() << phyObject);
      uint32_t nodeId = phyObject->GetDevice ()->GetObject<WifiNetDevice>
      ()-> GetNode () -> GetId ();
      double rxAntennaGain = m_wifiPhy->GetObject<YansWifiPhy> ()->GetRxGain ();
      rxPowerW = DbmToW (rxPowerDbm + rxAntennaGain) + DbmToW (m_antennaNoise);
      m_duration = rxPktDuration;


      if (m_psFactor > 0.95)
      {
        m_rxPowerW = 0.0;
        rxPowerW = 0.0;
        m_plcpHeaderPowerHarvested = 0.0;
        m_plcpPayloadPowerHarvested = 0.0;
        m_harvestedPower = 0.0;
        m_energyHarvested = 0.0;
      }
      else if (m_psFactor <= 0.95)
      {
        //Time now = Simulator::Now ();
        double inv_psFactor =  1.00 - m_psFactor;
        m_rxPowerW = rxPowerW * inv_psFactor;
  }
  else
  {
```

```
231       m_rxPowerW = 0.0;
232       rxPowerW = 0.0;
233       m_plcpHeaderPowerHarvested = 0.0;
234       m_plcpPayloadPowerHarvested = 0.0;
235       m_harvestedPower = 0.0;
236       m_energyHarvested = 0.0;
237       //UpdateSwiptHarvester ();
238     }
239
240   }
241
242   void
243   SwiptHarvester::UpdateSwiptHarvester (void)
244   {
245     NS_LOG_FUNCTION (this << "UpdateSwiptHarvester");
246     m_energyHarvested = 0.0;
247     m_harvestedPower = 0.0;
248
249     // Do not update if simulation has stopped
250     if (Simulator::IsFinished ())
251     {
252       NS_LOG_DEBUG ("SwiptHarvester: Simulation Finished.");
253       return;
254     }
255
256     // Calculate the Power Harvested
257     m_harvestedPower = m_rxPowerW * m_eta;
258
259     // Calculate the Energy Harvested
260     m_energyHarvested = m_harvestedPower * m_duration.GetSeconds ();
261
262
263       // Update the total energy harvested
264       m_totalEnergyHarvestedJ += m_energyHarvested;
265
266       //Update the total energy harvested
267       m_totalHarvestedPower += m_harvestedPower;
268   }
269
270
271
272   void
273   SwiptHarvester::UpdateEnergySourceSleep (Ptr<const Packet> packet,
274                                            WifiTxVector txVector,
275                                            enum WifiPreamble preamble,
276                                            uint8_t packetType,
277                                            Time rxPktDuration,
278                                            Ptr<YansWifiPhy> phyObject,
```

```
                                        double rxPowerSleep,
                                        double energySleep)
{
  NS_LOG_FUNCTION (this << energySleep);

  uint32_t nodeId = phyObject->GetDevice ()->GetObject<WifiNetDevice>
  ()-> GetNode () -> GetId ();


  m_totalHarvestedPower += rxPowerSleep;
  m_totalEnergyHarvestedJ += energySleep;

  Ptr<LiIonEnergySource> bat =  GetEnergySource ()-> GetObject<LiIonEnergySource> ();
  bat->UpdateEnergySleep(energySleep);
}

double
SwiptHarvester::DbmToW (double dBm) const
{
  NS_LOG_FUNCTION(this << dBm);
  double mW = std::pow (10.0, dBm / 10.0);
  return mW / 1000.0;
}

double
SwiptHarvester::GetRxPowerW (void)
{
  NS_LOG_FUNCTION (this);
  return m_rxPowerW;
}

void
SwiptHarvester::SetAntennaNoise (double antennaNoise)
{
  NS_LOG_FUNCTION (this << antennaNoise);
  m_antennaNoise = antennaNoise;
}

double
SwiptHarvester::GetAntennaNoise (void) const
{
  NS_LOG_FUNCTION (this);
  return m_antennaNoise;
}

void
SwiptHarvester::SetUpSwiptPhyListener (Ptr<WifiPhy> phy)
{
```

```
327   NS_LOG_FUNCTION (this << phy);
328   m_rxing2 = false;
329   m_swiptPhyListener = new SwiptPhyListener (this);
330   m_wifiPhy = phy;
331   m_wifiPhy->RegisterListener (m_swiptPhyListener);
332 }
333
334 void
335 SwiptHarvester::UnregisterSwiptPhyListener (Ptr<WifiPhy> phy)
336 {
337   NS_LOG_FUNCTION (this << phy);
338
339   phy->UnregisterListener (m_swiptPhyListener);
340   m_registered = false;
341   m_unregistered = true;
342 }
343
344 void
345 SwiptHarvester::DoInitialize (void)
346 {
347   NS_LOG_FUNCTION (this);
348   m_rxPowerW = 0.0;
349   m_energyHarvested = 0.0;
350   m_totalEnergyHarvestedJ = 0.0;
351   m_totalHarvestedPower = 0.0;
352   m_plcpHeaderPowerHarvested = 0.0;
353   m_plcpPayloadPowerHarvested = 0.0;
354   m_harvestedPower = 0.0;
355   m_energyHarvested = 0.0;
356
357 }
358
359 void
360 SwiptHarvester::DoDispose (void)
361 {
362   NS_LOG_FUNCTION (this);
363   m_rxPowerW = 0.0;
364   m_energyHarvested = 0.0;
365   m_totalEnergyHarvestedJ = 0.0;
366   m_totalHarvestedPower = 0.0;
367   m_plcpHeaderPowerHarvested = 0.0;
368   m_plcpPayloadPowerHarvested = 0.0;
369   m_harvestedPower = 0.0;
370   m_energyHarvested = 0.0;
371 }
372
373 void
374 SwiptHarvester::SetPowerSplitFactor (double psFactor)
```

```
375 {
376   NS_LOG_FUNCTION (this << psFactor);
377   m_psFactor = psFactor;
378 }
379
380 double
381 SwiptHarvester::GetPowerSplitFactor (void) const
382 {
383   NS_LOG_FUNCTION (this);
384   return m_psFactor;
385 }
386
387 void
388 SwiptHarvester::SetSwiptEfficiency (double eta)
389 {
390   NS_LOG_FUNCTION (this << eta);
391   m_eta = eta;
392 }
393
394 double
395 SwiptHarvester::GetSwiptEfficiency (void) const
396 {
397   NS_LOG_FUNCTION (this);
398   return m_eta;
399 }
400
401 void
402 SwiptHarvester::SetDCConversionEfficiency (double beta)
403 {
404   NS_LOG_FUNCTION (this << beta);
405   m_beta = beta;
406 }
407
408 double
409 SwiptHarvester::GetDCConversionEfficiency (void) const
410 {
411   NS_LOG_FUNCTION (this);
412   return m_beta;
413 }
414
415 //SwiptHarvester::DoGetPower (void)
416 double
417 SwiptHarvester::DoGetPower (void) const
418 {
419   NS_LOG_FUNCTION (this);
420
421   if (m_rxing2)
422   {
```

```
423
424       double powerRecharge = (m_harvestedPower +
425       m_plcpHeaderPowerHarvested + m_plcpPayloadPowerHarvested)*m_beta;
426       Ptr<LiIonEnergySource> bat =  GetEnergySource ()-> GetObject<LiIonEnergySource> ();
427       bat->IncreaseRemainingEnergy (m_energyHarvested);
428
429       return powerRecharge;
430       //return 0.0;
431   }
432
433   else
434   {
435     return 0.0;
436   }
437 }
438
439 void
440 SwiptHarvester::NotifyTxStartNow (Time duration, double txPowerDbm)
441 {
442   NS_LOG_FUNCTION(this << duration << txPowerDbm);
443   m_txing = true;
444   m_rxing2 = false;
445   m_off = false;
446   m_sleep = false;
447   m_maybeCca = false;
448   m_switching = false;
449   m_idle = false;
450   Time m_dur = duration;
451
452 }
453
454 void
455 SwiptHarvester::NotifyRxStartNow (Time duration)
456 {
457   NS_LOG_FUNCTION (this << duration << "m_rxing2 = true");
458
459   m_rxing2 = true;
460   m_off = false;
461   m_txing = false;
462   m_maybeCca = false;
463   m_sleep = false;
464   m_switching = false;
465   m_idle = false;
466
467   Time dur = duration;
468   m_plcpHeaderPowerHarvested = 0;
469   m_plcpPayloadPowerHarvested = 0;
470   m_harvestedPower = 0.0;
```

```
471
472   UpdateSwiptHarvester ();
473 }
474
475 void
476 SwiptHarvester::NotifyRxEndOkNow (void)
477 {
478   NS_LOG_FUNCTION(this);
479   m_harvestedPower = 0.0;
480   m_plcpHeaderPowerHarvested = 0.0;
481   m_plcpPayloadPowerHarvested = 0.0;
482
483   m_maybeCca = false;
484   m_rxing2 = false;
485   m_off = false;
486   m_txing = false;
487   m_sleep = false;
488   m_switching = false;
489   m_idle = true;
490
491 }
492
493 void
494 SwiptHarvester::NotifyRxEndErrorNow (void)
495 {
496   NS_LOG_FUNCTION(this);
497   m_harvestedPower = 0.0;
498   m_plcpHeaderPowerHarvested = 0.0;
499   m_plcpPayloadPowerHarvested = 0.0;
500
501   m_sleep = false;
502   m_maybeCca = false;
503   m_rxing2 = false;
504   m_off = false;
505   m_txing = false;
506   m_switching = false;
507   m_idle = true;
508
509 }
510
511 void
512 SwiptHarvester::NotifyMaybeCcaBusyStartNow (Time duration)
513 {
514   NS_LOG_FUNCTION(this << duration);
515   m_maybeCca = true;
516   m_txing = false;
517   m_rxing2 = false;
518   m_off = false;
```

```
519   m_sleep = false;
520   m_switching = false;
521   m_idle = false;
522 
523   Time m_dur = duration;
524   m_energyHarvested = 0.0;
525 
526 }
527 
528 void
529 SwiptHarvester::NotifySwitchingStartNow (Time duration)
530 {
531   NS_LOG_FUNCTION(this << duration);
532   m_sleep = false;
533   m_txing = false;
534   m_maybeCca = false;
535   m_rxing2 = false;
536   m_off = false;
537   m_switching = true;
538   m_idle = false;
539 
540   Time m_dur = duration;
541 
542 }
543 
544 void
545 SwiptHarvester::NotifySleepNow (void)
546 {
547   NS_LOG_FUNCTION(this);
548   m_sleep = true;
549   m_txing = false;
550   m_maybeCca = false;
551   m_rxing2 = false;
552   m_off = false;
553   m_switching = false;
554   m_idle = false;
555 
556   UpdateSwiptHarvester ();
557 }
558 
559 void
560 SwiptHarvester::NotifyWakeupNow (void)
561 {
562   NS_LOG_FUNCTION(this);
563   m_sleep = false;
564   m_rxing2 = false;
565   m_off = false;
566   m_txing = false;
```

```
567   m_maybeCca = false;
568   m_switching = false;
569   m_idle = false;
570 }
571
572 void
573 SwiptHarvester::NotifyPlcpHeader (double noiseInterferenceHeaderW,
574                                   Time noiseInterferenceHeaderDuration,
575                                   Ptr<SwiptHarvester> swiptH)
576 {
577   NS_LOG_FUNCTION(this);
578
579   if (swiptH!=NULL)
580   {
581     //if (GetPowerSplitFactor () != 1.0)
582     if ((m_rxing2) && (m_psFactor < 0.95))
583     {
584
585       m_headerDuration = noiseInterferenceHeaderDuration;
586       m_interferenceWHeader = std::abs (noiseInterferenceHeaderW);
587       m_plcpHeaderEnergyHarvested = (m_interferenceWHeader *
          ↪  noiseInterferenceHeaderDuration.GetSeconds ()/GetPowerSplitFactor ()) * (1.0
          ↪  -GetPowerSplitFactor ())* m_eta;
588       totalInterferenceHeader += m_plcpHeaderEnergyHarvested ;
589
590       // Update the total energy harvested
591       m_totalEnergyHarvestedJ += m_plcpHeaderEnergyHarvested;
592
593       // Calculate the Power Harvested
594       m_plcpHeaderPowerHarvested = (m_plcpHeaderEnergyHarvested /
          ↪  noiseInterferenceHeaderDuration.GetSeconds ()) ;
595
596       m_totalHarvestedPower += m_plcpHeaderPowerHarvested;
597
598     }
599     else
600     {
601       m_headerDuration = Seconds (0.0);
602       m_interferenceWHeader = 0.0;
603       m_plcpHeaderEnergyHarvested = 0.0;
604       m_plcpHeaderPowerHarvested = 0.0;
605
606     }
607   }
608 }
609
610 void
611 SwiptHarvester::NotifyPlcpPayload (double noiseInterferencePayloadW,
```

```
612                                     Time noiseInterferenceWDuration,
613                                     Ptr<SwiptHarvester> swiptH)
614 {
615   NS_LOG_FUNCTION(this);
616
617   if (swiptH!=NULL)
618   {
619     double payloadDuration;
620
621     //if (GetPowerSplitFactor () != 1.0)
622     if ((m_rxing2) && (m_psFactor < 0.95))
623     {
624       payloadDuration = noiseInterferenceWDuration.GetSeconds () -
       ↪  m_headerDuration.GetSeconds ();
625       m_interferenceWPayload = std::abs (noiseInterferencePayloadW);
626
627       m_plcpPayloadEnergyHarvested =
       ↪  (m_interferenceWPayload*payloadDuration/GetPowerSplitFactor ()) * (1.0 -
       ↪  GetPowerSplitFactor ()) * m_eta ;
628
629       // Update the total energy harvested
630       m_totalEnergyHarvestedJ += m_plcpPayloadEnergyHarvested;
631
632       // Calculate the Power Harvested
633       m_plcpPayloadPowerHarvested = (m_plcpPayloadEnergyHarvested / payloadDuration);
634
635       // Update the total energy harvested
636       m_totalHarvestedPower += m_plcpPayloadPowerHarvested;
637
638       totalInterferencePayload += m_plcpPayloadEnergyHarvested;
639     }
640     else
641     {
642       payloadDuration = 0.0;
643       m_interferenceWPayload = 0.0;
644       m_plcpPayloadEnergyHarvested = 0.0;
645       m_plcpPayloadPowerHarvested = 0.0;
646     }
647
648   }
649 }
650
651 void
652 SwiptHarvester::NotifyOffNow (void)
653 {
654   NS_LOG_FUNCTION(this);
655   m_off = true;
656   m_rxing2 = false;
```

```
657   m_sleep = false;
658   m_txing = false;
659   m_switching = false;
660   m_maybeCca = false;
661   m_idle = false;
662 
663 }
664 
665 void
666 SwiptHarvester::NotifyOnNow (void)
667 {
668   NS_LOG_FUNCTION(this);
669   m_off = false;
670   m_rxing2 = false;
671   m_sleep = false;
672   m_txing = false;
673   m_switching = false;
674   m_maybeCca = false;
675 
676 }
677 
678 void
679 SwiptHarvester::NotifyIdleNow (void)
680 {
681   NS_LOG_FUNCTION(this);
682   m_off = false;
683   m_rxing2 = false;
684   m_sleep = false;
685   m_txing = false;
686   m_switching = false;
687   m_maybeCca = false;
688   m_idle = true;
689 }
690 
691 } // namespace ns3
```

APPENDIX G

LIIONENERGYSOURCE

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2010 Andrea Sacco
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: Andrea Sacco <andrea.sacco85@gmail.com>
 */

#include "li-ion-energy-source.h"
#include "ns3/log.h"
#include "ns3/assert.h"
#include "ns3/double.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/simulator.h"

#include <cmath>

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("LiIonEnergySource");

NS_OBJECT_ENSURE_REGISTERED (LiIonEnergySource);

TypeId
LiIonEnergySource::GetTypeId (void)
{
```

```
39  static TypeId tid = TypeId ("ns3::LiIonEnergySource")
40    .SetParent<EnergySource> ()
41    .SetGroupName ("Energy")
42    .AddConstructor<LiIonEnergySource> ()
43    .AddAttribute ("LiIonEnergySourceInitialEnergyJ",
44                   "Initial energy stored in basic energy source.",
45                   DoubleValue (31752.0),  // in Joules
46                   MakeDoubleAccessor (&LiIonEnergySource::SetInitialEnergy,
47                                       &LiIonEnergySource::GetInitialEnergy),
48                   MakeDoubleChecker<double> ())
49    .AddAttribute ("LiIonEnergyLowBatteryThreshold",
50                   "Low battery threshold for LiIon energy source.",
51                   DoubleValue (0.10), // as a fraction of the initial energy
52                   MakeDoubleAccessor (&LiIonEnergySource::m_lowBatteryTh),
53                   MakeDoubleChecker<double> ())
54    .AddAttribute ("LiIonEnergyHighBatteryThreshold",
55                   "High battery threshold for basic energy source.",
56                   DoubleValue (0.15), // as a fraction of the initial energy
57                   MakeDoubleAccessor (&LiIonEnergySource::m_highBatteryTh),
58                   MakeDoubleChecker<double> ())
59    .AddAttribute ("InitialCellVoltage",
60                   "Initial (maximum) voltage of the cell (fully charged).",
61                   DoubleValue (4.05), // in Volts
62                   MakeDoubleAccessor (&LiIonEnergySource::SetInitialSupplyVoltage,
63                                       &LiIonEnergySource::GetSupplyVoltage),
64                   MakeDoubleChecker<double> ())
65    .AddAttribute ("NominalCellVoltage",
66                   "Nominal voltage of the cell.",
67                   DoubleValue (3.6),  // in Volts
68                   MakeDoubleAccessor (&LiIonEnergySource::m_eNom),
69                   MakeDoubleChecker<double> ())
70    .AddAttribute ("ExpCellVoltage",
71                   "Cell voltage at the end of the exponential zone.",
72                   DoubleValue (3.6),  // in Volts
73                   MakeDoubleAccessor (&LiIonEnergySource::m_eExp),
74                   MakeDoubleChecker<double> ())
75    .AddAttribute ("RatedCapacity",
76                   "Rated capacity of the cell.",
77                   DoubleValue (2.45),   // in Ah
78                   MakeDoubleAccessor (&LiIonEnergySource::m_qRated),
79                   MakeDoubleChecker<double> ())
80    .AddAttribute ("NomCapacity",
81                   "Cell capacity at the end of the nominal zone.",
82                   DoubleValue (1.1),  // in Ah
83                   MakeDoubleAccessor (&LiIonEnergySource::m_qNom),
84                   MakeDoubleChecker<double> ())
85    .AddAttribute ("ExpCapacity",
86                   "Cell Capacity at the end of the exponential zone.",
```

```
87                    DoubleValue (1.2),  // in Ah
88                    MakeDoubleAccessor (&LiIonEnergySource::m_qExp),
89                    MakeDoubleChecker<double> ())
90      .AddAttribute ("InternalResistance",
91                    "Internal resistance of the cell",
92                    DoubleValue (0.083),  // in Ohms
93                    MakeDoubleAccessor (&LiIonEnergySource::m_internalResistance),
94                    MakeDoubleChecker<double> ())
95      .AddAttribute ("TypCurrent",
96                    "Typical discharge current used to fit the curves",
97                    DoubleValue (2.33), // in A
98                    MakeDoubleAccessor (&LiIonEnergySource::m_typCurrent),
99                    MakeDoubleChecker<double> ())
100     .AddAttribute ("ThresholdVoltage",
101                   "Minimum threshold voltage to consider the battery depleted.",
102                   DoubleValue (3.3), // in Volts
103                   MakeDoubleAccessor (&LiIonEnergySource::m_minVoltTh),
104                   MakeDoubleChecker<double> ())
105     .AddAttribute ("PeriodicEnergyUpdateInterval",
106                   "Time between two consecutive periodic energy updates.",
107                   TimeValue (Seconds (1.0)),
108                   MakeTimeAccessor (&LiIonEnergySource::SetEnergyUpdateInterval,
109                                     &LiIonEnergySource::GetEnergyUpdateInterval),
110                   MakeTimeChecker ())
111     .AddTraceSource ("RemainingEnergy",
112                      "Remaining energy at BasicEnergySource.",
113                      MakeTraceSourceAccessor (&LiIonEnergySource::m_remainingEnergyJ),
114                      "ns3::TracedValue::DoubleCallback")
115   ;
116   return tid;
117 }
118
119 LiIonEnergySource::LiIonEnergySource ()
120   : m_drainedCapacity (0.0),
121     m_lastUpdateTime (Seconds (0.0))
122 {
123   NS_LOG_FUNCTION (this);
124 }
125
126 LiIonEnergySource::~LiIonEnergySource ()
127 {
128   NS_LOG_FUNCTION (this);
129 }
130
131 void
132 LiIonEnergySource::SetInitialEnergy (double initialEnergyJ)
133 {
134   NS_LOG_FUNCTION (this << initialEnergyJ);
```

```
135   NS_ASSERT (initialEnergyJ >= 0);
136   m_initialEnergyJ = initialEnergyJ;
137   // set remaining energy to be initial energy
138   m_remainingEnergyJ = m_initialEnergyJ;
139 }
140
141 double
142 LiIonEnergySource::GetInitialEnergy (void) const
143 {
144   NS_LOG_FUNCTION (this);
145   return m_initialEnergyJ;
146 }
147
148 void
149 LiIonEnergySource::SetInitialSupplyVoltage (double supplyVoltageV)
150 {
151   NS_LOG_FUNCTION (this << supplyVoltageV);
152   m_eFull = supplyVoltageV;
153   m_supplyVoltageV = supplyVoltageV;
154 }
155
156 double
157 LiIonEnergySource::GetSupplyVoltage (void) const
158 {
159   NS_LOG_FUNCTION (this);
160   return m_supplyVoltageV;
161 }
162
163 void
164 LiIonEnergySource::SetEnergyUpdateInterval (Time interval)
165 {
166   NS_LOG_FUNCTION (this << interval);
167   m_energyUpdateInterval = interval;
168 }
169
170 Time
171 LiIonEnergySource::GetEnergyUpdateInterval (void) const
172 {
173   NS_LOG_FUNCTION (this);
174   return m_energyUpdateInterval;
175 }
176
177 double
178 LiIonEnergySource::GetRemainingEnergy (void)
179 {
180   NS_LOG_FUNCTION (this);
181   // update energy source to get the latest remaining energy.
182   UpdateEnergySource ();
```

```
183   return m_remainingEnergyJ;
184 }
185
186 double
187 LiIonEnergySource::GetEnergyFraction (void)
188 {
189   NS_LOG_FUNCTION (this);
190   // update energy source to get the latest remaining energy.
191   UpdateEnergySource ();
192   return m_remainingEnergyJ / m_initialEnergyJ;
193 }
194
195 void
196 LiIonEnergySource::DecreaseRemainingEnergy (double energyJ)
197 {
198   NS_LOG_FUNCTION (this << energyJ);
199   //NS_ASSERT (energyJ >= 0);
200   m_remainingEnergyJ -= energyJ;
201
202   // check if remaining energy is 0
203   if (m_supplyVoltageV <= m_minVoltTh)
204     {
205       HandleEnergyDrainedEvent ();
206     }
207 }
208
209 void
210 LiIonEnergySource::IncreaseRemainingEnergy (double energyJ)
211 {
212   NS_LOG_FUNCTION (this << energyJ);
213   m_remainingEnergyJ += energyJ;
214 }
215
216 void
217 LiIonEnergySource::UpdateEnergySource (void)
218 {
219   NS_LOG_FUNCTION (this);
220   NS_LOG_DEBUG ("LiIonEnergySource:Updating remaining energy at node #" <<
221                 GetNode ()->GetId ());
222
223   double remainingEnergy = m_remainingEnergyJ;
224
225   // do not update if simulation has finished
226   if (Simulator::IsFinished ())
227     {
228       return;
229     }
230
```

```
231   //m_energyUpdateEvent.Cancel ();
232
233   if (!m_depleted)
234   {
235     CalculateRemainingEnergy ();
236   }
237
238   m_lastUpdateTime = Simulator::Now ();
239
240   if (!m_depleted && m_remainingEnergyJ < m_lowBatteryTh * m_initialEnergyJ)
241     {
242       m_depleted = true;
243       HandleEnergyDrainedEvent ();
244     }
245   else if (m_depleted && m_remainingEnergyJ > m_highBatteryTh * m_initialEnergyJ)
246     {
247       m_depleted = false;
248       HandleEnergyRechargedEvent ();
249     }
250
251   m_energyUpdateEvent = Simulator::Schedule (m_energyUpdateInterval,
252                                              &LiIonEnergySource::UpdateEnergySource,
253                                              this);
254 }
255
256
257 /*
258  * Private functions start here.
259  */
260 void
261 LiIonEnergySource::DoInitialize (void)
262 {
263   NS_LOG_FUNCTION (this);
264   UpdateEnergySource ();  // start periodic update
265 }
266
267 void
268 LiIonEnergySource::DoDispose (void)
269 {
270   NS_LOG_FUNCTION (this);
271   CalculateRemainingEnergy ();
272   BreakDeviceEnergyModelRefCycle ();  // break reference cycle
273 }
274
275
276 void
277 LiIonEnergySource::HandleEnergyDrainedEvent (void)
278 {
```

```
279   NS_LOG_FUNCTION (this);
280   NS_LOG_DEBUG ("LiIonEnergySource:Energy depleted at node #" <<
281                 GetNode ()->GetId ());
282   NotifyEnergyDrained (); // notify DeviceEnergyModel objects
283   if (m_remainingEnergyJ <= 0)
284     {
285       m_remainingEnergyJ = 0; // energy never goes below 0
286     }
287 }
288
289 void
290 LiIonEnergySource::HandleEnergyRechargedEvent (void)
291 {
292   NS_LOG_FUNCTION (this);
293   NS_LOG_DEBUG ("BasicEnergySource:Energy recharged!");
294
295   NotifyEnergyRecharged (); // notify DeviceEnergyModel objects
296 }
297
298 void
299 LiIonEnergySource::CalculateRemainingEnergy (void)
300 {
301   NS_LOG_FUNCTION (this);
302   double totalCurrentA = CalculateTotalCurrent ();
303   Time duration = Simulator::Now () - m_lastUpdateTime;
304   NS_ASSERT (duration.GetSeconds () >= 0);
305   // energy = current * voltage * time
306   double energyToDecreaseJ = totalCurrentA * m_supplyVoltageV * duration.GetSeconds ();
307    ;
308   m_drainedCapacity += (totalCurrentA * duration.GetSeconds () / 3600);
309   // update the supply voltage
310   m_supplyVoltageV = GetVoltage (totalCurrentA);
311   NS_LOG_DEBUG ("LiIonEnergySource:Remaining energy = " << m_remainingEnergyJ);
312 }
313
314 double
315 LiIonEnergySource::GetVoltage (double i) const
316 {
317   NS_LOG_FUNCTION (this << i);
318
319   // integral of i in dt, drained capacity in Ah
320   double it = m_drainedCapacity;
321
322   // empirical factors
323   double A = m_eFull - m_eExp;
324   double B = 3 / m_qExp;
325
326   // slope of the polarization curve
```

```
327   double K = std::abs ( (m_eFull - m_eNom + A * (std::exp (-B * m_qNom) - 1)) * (m_qRated -
      ↪  m_qNom) / m_qNom);
328
329   // constant voltage
330   double E0 = m_eFull + K + m_internalResistance * m_typCurrent - A;
331
332   double E = E0 - K * m_qRated / (m_qRated - it) + A * std::exp (-B * it);
333
334   // cell voltage
335   double V = E - m_internalResistance * i;
336
337   NS_LOG_DEBUG ("Voltage: " << V << " with E: " << E);
338
339   return V;
340 }
341
342 } // namespace ns3
```

APPENDIX H

SWIPTHARVESTERHELPER

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2023 University of Brasilia, Brasília, DF, Brazil.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author: José Antônio de França Junior <aspect_josef@hotmail.com>
 */

#include "swipt-harvester-helper.h"
#include "ns3/energy-harvester.h"

namespace ns3 {

SwiptHarvesterHelper::SwiptHarvesterHelper ()
{
  m_swiptHarvester.SetTypeId ("ns3::SwiptHarvester");
}

SwiptHarvesterHelper::~SwiptHarvesterHelper ()
{
}

void
SwiptHarvesterHelper::Set (std::string name, const AttributeValue &v)
{
  m_swiptHarvester.Set (name, v);
```

```
}

Ptr<EnergyHarvester>
SwiptHarvesterHelper::DoInstall (Ptr<EnergySource> source) const
{
  NS_ASSERT (source != 0);
  Ptr<Node> node = source->GetNode ();

  // Create a new Swipt Harvester
  Ptr<EnergyHarvester> harvester = m_swiptHarvester.Create<EnergyHarvester> ();
  NS_ASSERT (harvester != 0);

  // Connect the Swipt Harvester to the Energy Source
  source->ConnectEnergyHarvester (harvester);
  harvester->SetNode (node);
  harvester->SetEnergySource (source);
  return harvester;
}

} // namespace ns3
```

APPENDIX I

INTERFERENCEHELPER

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2005,2006 INRIA
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Authors: Mathieu Lacage <mathieu.lacage@sophia.inria.fr>
 *          Sébastien Deronne <sebastien.deronne@gmail.com>
 */

// Recalculate energy level for the duration of the interfering frame
// Make sure the frame interference power gets multiplied by (1-psFactor)
// Implement the formulas for energy and power once again
// For the recharging routine, there needs to be defined another variable rather than
↪  m_rxing
// Check the whole fluxogram operation

#include "interference-helper.h"
#include "wifi-phy.h"
#include "error-rate-model.h"
#include "ns3/simulator.h"
#include "ns3/log.h"
#include "ns3/swipt-harvester.h"
#include <algorithm>

namespace ns3 {

```

```
38  NS_LOG_COMPONENT_DEFINE ("InterferenceHelper");
39
40  /****************************************************************
41   *       Phy event class
42   ****************************************************************/
43
44  InterferenceHelper::Event::Event (uint32_t size, WifiTxVector txVector,
45                                    enum WifiPreamble preamble,
46                                    Time duration, double rxPower)
47    : m_size (size),
48      m_txVector (txVector),
49      m_preamble (preamble),
50      m_startTime (Simulator::Now ()),
51      m_endTime (m_startTime + duration),
52      m_rxPowerW (rxPower)
53  {
54  }
55
56  InterferenceHelper::Event::~Event ()
57  {
58  }
59
60  Time
61  InterferenceHelper::Event::GetDuration (void) const
62  {
63    return m_endTime - m_startTime;
64  }
65
66  Time
67  InterferenceHelper::Event::GetStartTime (void) const
68  {
69    return m_startTime;
70  }
71
72  Time
73  InterferenceHelper::Event::GetEndTime (void) const
74  {
75    return m_endTime;
76  }
77
78  double
79  InterferenceHelper::Event::GetRxPowerW (void) const
80  {
81    return m_rxPowerW;
82  }
83
84  uint32_t
85  InterferenceHelper::Event::GetSize (void) const
```

```
86  {
87    return m_size;
88  }
89
90  WifiTxVector
91  InterferenceHelper::Event::GetTxVector (void) const
92  {
93    return m_txVector;
94  }
95
96  WifiMode
97  InterferenceHelper::Event::GetPayloadMode (void) const
98  {
99    return m_txVector.GetMode ();
100 }
101
102 enum WifiPreamble
103 InterferenceHelper::Event::GetPreambleType (void) const
104 {
105   return m_preamble;
106 }
107
108
109 /****************************************************************
110  *       Class which records SNIR change events for a
111  *       short period of time.
112  ****************************************************************/
113
114 InterferenceHelper::NiChange::NiChange (Time time, double delta)
115   : m_time (time),
116     m_delta (delta)
117 {
118 }
119
120 Time
121 InterferenceHelper::NiChange::GetTime (void) const
122 {
123   return m_time;
124 }
125
126 double
127 InterferenceHelper::NiChange::GetDelta (void) const
128 {
129   return m_delta;
130 }
131
132 bool
133 InterferenceHelper::NiChange::operator < (const InterferenceHelper::NiChange& o) const
```

```
{
  return (m_time < o.m_time);
}


/****************************************************************
 *       The actual InterferenceHelper
 ****************************************************************/

InterferenceHelper::InterferenceHelper ()
  : m_errorRateModel (0),
    m_firstPower (0.0),
    m_rxing (false)
{
}

InterferenceHelper::~InterferenceHelper ()
{
  EraseEvents ();
  m_errorRateModel = 0;
}

Ptr<InterferenceHelper::Event>
InterferenceHelper::Add (uint32_t size, WifiTxVector txVector,
                         enum WifiPreamble preamble,
                         Time duration, double rxPowerW)
{
  Ptr<InterferenceHelper::Event> event;

  event = Create<InterferenceHelper::Event> (size,
                                             txVector,
                                             preamble,
                                             duration,
                                             rxPowerW);
  AppendEvent (event);
  return event;
}


void
InterferenceHelper::SetNoiseFigure (double value)
{
  m_noiseFigure = value;
}

double
InterferenceHelper::GetNoiseFigure (void) const
{
```

```
182   return m_noiseFigure;
183 }
184
185 void
186 InterferenceHelper::SetErrorRateModel (Ptr<ErrorRateModel> rate)
187 {
188   m_errorRateModel = rate;
189 }
190
191 Ptr<ErrorRateModel>
192 InterferenceHelper::GetErrorRateModel (void) const
193 {
194   return m_errorRateModel;
195 }
196
197 Time
198 InterferenceHelper::GetEnergyDuration (double energyW)
199 {
200   Time now = Simulator::Now ();
201   double noiseInterferenceW = 0.0;
202   Time end = now;
203   noiseInterferenceW = m_firstPower;
204   for (NiChanges::const_iterator i = m_niChanges.begin (); i != m_niChanges.end (); i++)
205     {
206       noiseInterferenceW += i->GetDelta ();
207       end = i->GetTime ();
208       if (end < now)
209         {
210           continue;
211         }
212       if (noiseInterferenceW < energyW)
213         {
214           break;
215         }
216     }
217   return end > now ? end - now : MicroSeconds (0);
218 }
219
220 void
221 InterferenceHelper::AppendEvent (Ptr<InterferenceHelper::Event> event)
222 {
223   Time now = Simulator::Now ();
224   if (!m_rxing)
225     {
226       NiChanges::iterator nowIterator = GetPosition (now);
227       for (NiChanges::iterator i = m_niChanges.begin (); i != nowIterator; i++)
228         {
229           m_firstPower += i->GetDelta ();
```

```
230         }
231       m_niChanges.erase (m_niChanges.begin (), nowIterator);
232       m_niChanges.insert (m_niChanges.begin (), NiChange (event->GetStartTime (),
          ↪  event->GetRxPowerW ()));
233     }
234   else
235     {
236       AddNiChangeEvent (NiChange (event->GetStartTime (), event->GetRxPowerW ()));
237     }
238   AddNiChangeEvent (NiChange (event->GetEndTime (), -event->GetRxPowerW ()));
239
240 }
241
242 double
243 InterferenceHelper::CalculateSnr (double signal, double noiseInterference, WifiMode mode)
    ↪  const
244 {
245   //thermal noise at 290K in J/s = W
246   static const double BOLTZMANN = 1.3803e-23;
247   //Nt is the power of thermal noise in W
248   double Nt = BOLTZMANN * 290.0 * mode.GetBandwidth ();
249   //receiver noise Floor (W) which accounts for thermal noise and non-idealities of the
      ↪  receiver
250   double noiseFloor = m_noiseFigure * Nt;
251   double noise = noiseFloor + noiseInterference;
252   double snr = signal / noise;
253   NS_LOG_DEBUG ("signal= " << signal << ", noise=" << noiseFloor << ", interference=" <<
      ↪  noiseInterference << ", snr=" << snr);
254   //std::cout << Simulator::Now().GetSeconds() << " InterferenceHelper CalculateSnr,
      ↪  signal= " << signal << "W, noiseFloor=" << noiseFloor << "W, noiseInterference=" <<
      ↪  noiseInterference << "W, snr=" << snr << std::endl;
255   return snr;
256 }
257
258 double
259 InterferenceHelper::CalculateNoiseInterferenceW (Ptr<InterferenceHelper::Event> event,
    ↪  NiChanges *ni) const
260 {
261   double noiseInterference = m_firstPower;
262   NS_ASSERT (m_rxing);
263   for (NiChanges::const_iterator i = m_niChanges.begin () + 1; i != m_niChanges.end ();
      ↪  i++)
264     {
265       if ((event->GetEndTime () == i->GetTime ()) && event->GetRxPowerW () == -i->GetDelta
          ↪  ())
266         {
267           break;
268         }
```

```
269       ni->push_back (*i);
270     }
271   ni->insert (ni->begin (), NiChange (event->GetStartTime (), noiseInterference));
272   ni->push_back (NiChange (event->GetEndTime (), 0));
273   return noiseInterference;
274 }
275
276 double
277 InterferenceHelper::CalculateChunkSuccessRate (double snir, Time duration, WifiMode mode)
    ↪ const
278 {
279   if (duration == NanoSeconds (0))
280     {
281       return 1.0;
282     }
283   uint32_t rate = mode.GetPhyRate ();
284   uint64_t nbits = (uint64_t)(rate * duration.GetSeconds ());
285   double csr = m_errorRateModel->GetChunkSuccessRate (mode, snir, (uint32_t)nbits);
286   return csr;
287 }
288
289 double
290 InterferenceHelper::CalculatePlcpPayloadPer (Ptr<const InterferenceHelper::Event> event,
291                                              NiChanges *ni,
292                                              Time duration,
293                                              Ptr<SwiptHarvester> swiptH) const
294 {
295   NS_LOG_FUNCTION (this);
296   double psr = 1.0; /* Packet Success Rate */
297   NiChanges::iterator j = ni->begin ();
298   Time previous = (*j).GetTime ();
299   WifiMode payloadMode = event->GetPayloadMode ();
300   WifiPreamble preamble = event->GetPreambleType ();
301   Time plcpHeaderStart;
302   Time plcpHsigHeaderStart;
303   Time plcpHtTrainingSymbolsStart;
304   Time plcpPayloadStart;
305
306   Time plcpTrainingSymbolsStart;
307   Time plcpSigAStart;
308   Time plcpS1gTrainingSymbolsStart;
309   Time plcpSigBStart;
310  if (payloadMode.GetModulationClass () != WIFI_MOD_CLASS_S1G)
311   {
312     plcpHeaderStart = (*j).GetTime () + WifiPhy::GetPlcpPreambleDuration (payloadMode,
      ↪ preamble); //packet start time + preamble
313     plcpHsigHeaderStart = plcpHeaderStart + WifiPhy::GetPlcpHeaderDuration (payloadMode,
      ↪ preamble); //packet start time + preamble + L-SIG
```

```
314       plcpHtTrainingSymbolsStart = plcpHsigHeaderStart + WifiPhy::GetPlcpHtSigHeaderDuration
          ↪  (preamble); //packet start time + preamble + L-SIG + HT-SIG
315       plcpPayloadStart = plcpHtTrainingSymbolsStart +
          ↪  WifiPhy::GetPlcpHtTrainingSymbolDuration (preamble,event->GetTxVector ()); //packet
          ↪  start time + preamble + L-SIG + HT-SIG + HT Training
316     }
317   else
318     {
319       plcpHeaderStart = (*j).GetTime () + WifiPhy::GetPlcpPreambleDuration (payloadMode,
          ↪  preamble); //packet start time + preamble
320       plcpTrainingSymbolsStart = plcpHeaderStart + WifiPhy::GetPlcpHeaderDuration
          ↪  (payloadMode, preamble); //packet start time + preamble + L-SIG
321       plcpSigAStart = plcpTrainingSymbolsStart + WifiPhy::GetPlcpTrainingSymbolDuration
          ↪  (preamble,event->GetTxVector()); //packet start time + preamble + L-SIG + LTF
322       plcpS1gTrainingSymbolsStart = plcpSigAStart + WifiPhy::GetPlcpSigADuration (preamble);
          ↪  //packet start time + preamble + L-SIG + LTF + S1G-A
323       plcpSigBStart = plcpS1gTrainingSymbolsStart + WifiPhy::GetPlcpS1gTrainingSymbolDuration
          ↪  (preamble,event->GetTxVector()); //packet start time + preamble + L-SIG + LTF +
          ↪  S1G-A + S1G Training
324       plcpPayloadStart = plcpSigBStart + WifiPhy::GetPlcpSigBDuration (preamble); ////packet
          ↪  start time + preamble + L-SIG + LTF + S1G-A + S1G Training + S1G-B
325     }
326     double noiseInterferenceW = (*j).GetDelta ();
327     double noiseInterferenceWCalc = noiseInterferenceW;
328     double powerW = event->GetRxPowerW ();
329     j++;
330     while (ni->end () != j)
331       {
332         Time current = (*j).GetTime ();
333         NS_LOG_DEBUG ("previous= " << previous << ", current=" << current);
334         NS_ASSERT (current >= previous);
335         //Case 1: Both previous and current point to the payload
336         if (previous >= plcpPayloadStart)
337           {
338             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
339                                                             noiseInterferenceW,
340                                                             payloadMode),
341                                               current - previous,
342                                               payloadMode);
343
344             NS_LOG_DEBUG ("Both previous and current point to the payload: mode=" <<
              ↪  payloadMode << ", psr=" << psr);
345           }
346         //Case 2: previous is before payload and current is in the payload
347         else if (current >= plcpPayloadStart)
348           {
349             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
350                                                             noiseInterferenceW,
```

```
                                                    payloadMode),
                                          current - plcpPayloadStart,
                                          payloadMode);
          NS_LOG_DEBUG ("previous is before payload and current is in the payload: mode="
          ↪  << payloadMode << ", psr=" << psr);
        }
      if ((*j).GetDelta () > 0)
      {
        noiseInterferenceWCalc += (*j).GetDelta ();
      }
      noiseInterferenceW += (*j).GetDelta ();
      previous = (*j).GetTime ();
      j++;
    }

  double per = 1 - psr;

  if(swiptH != NULL)
  {
    swiptH->InterfPlcpPayload (noiseInterferenceW, duration, swiptH);

    int32_t nodeId = swiptH->GetNode ()->GetId ();
  }
  return per;
}

double
InterferenceHelper::CalculatePlcpHeaderPer (Ptr<const InterferenceHelper::Event> event,
                                          NiChanges *ni,
                                          Time PlcpAndHeaderDuration,
                                          Ptr<SwiptHarvester> swiptH) const
{
  NS_LOG_FUNCTION (this);
  double psr = 1.0; /* Packet Success Rate */
  NiChanges::iterator j = ni->begin ();
  Time previous = (*j).GetTime ();
  WifiMode payloadMode = event->GetPayloadMode ();
  WifiPreamble preamble = event->GetPreambleType ();
  WifiMode htHeaderMode;
  if (preamble == WIFI_PREAMBLE_HT_MF)
    {
      //mode for PLCP header fields sent with HT modulation
      htHeaderMode = WifiPhy::GetHTPlcpHeaderMode (payloadMode, preamble);
    }
  WifiMode headerMode = WifiPhy::GetPlcpHeaderMode (payloadMode, preamble);
  Time plcpHeaderStart;
  Time plcpHsigHeaderStart;
  Time plcpHtTrainingSymbolsStart;
```

```
398   Time plcpPayloadStart;
399
400   Time plcpTrainingSymbolsStart;
401   Time plcpSigAStart;
402   Time plcpS1gTrainingSymbolsStart;
403   Time plcpSigBStart;
404
405   if (payloadMode.GetModulationClass () != WIFI_MOD_CLASS_S1G)
406   {
407     Time plcpHeaderStart = (*j).GetTime () + WifiPhy::GetPlcpPreambleDuration (payloadMode,
      ↪  preamble); //packet start time + preamble
408     Time plcpHsigHeaderStart = plcpHeaderStart + WifiPhy::GetPlcpHeaderDuration
      ↪  (payloadMode, preamble); //packet start time + preamble + L-SIG
409     Time plcpHtTrainingSymbolsStart = plcpHsigHeaderStart +
      ↪  WifiPhy::GetPlcpHtSigHeaderDuration (preamble); //packet start time + preamble +
      ↪  L-SIG + HT-SIG
410     Time plcpPayloadStart = plcpHtTrainingSymbolsStart +
      ↪  WifiPhy::GetPlcpHtTrainingSymbolDuration (preamble, event->GetTxVector ());
      ↪  //packet start time + preamble + L-SIG + HT-SIG + HT Training
411   }
412   else
413   {
414     Time plcpHeaderStart = (*j).GetTime () + WifiPhy::GetPlcpPreambleDuration (payloadMode,
      ↪  preamble); //packet start time + preamble
415     Time plcpTrainingSymbolsStart = plcpHeaderStart + WifiPhy::GetPlcpHeaderDuration
      ↪  (payloadMode, preamble); //packet start time + preamble + L-SIG
416     Time plcpSigAStart = plcpTrainingSymbolsStart + WifiPhy::GetPlcpTrainingSymbolDuration
      ↪  (preamble,event->GetTxVector()); //packet start time + preamble + L-SIG + LTF
417     Time plcpS1gTrainingSymbolsStart = plcpSigAStart + WifiPhy::GetPlcpSigADuration
      ↪  (preamble); //packet start time + preamble + L-SIG + LTF + S1G-A
418     Time plcpSigBStart = plcpS1gTrainingSymbolsStart +
      ↪  WifiPhy::GetPlcpS1gTrainingSymbolDuration (preamble,event->GetTxVector()); //packet
      ↪  start time + preamble + L-SIG + LTF + S1G-A + S1G Training
419     Time plcpPayloadStart = plcpSigBStart + WifiPhy::GetPlcpSigBDuration (preamble);
      ↪  ////packet start time + preamble + L-SIG + LTF + S1G-A + S1G Training + S1G-B
420   }
421   double noiseInterferenceW = (*j).GetDelta ();
422   double powerW = event->GetRxPowerW ();
423   j++;
424   while (ni->end () != j)
425   {
426     Time current = (*j).GetTime ();
427     NS_LOG_DEBUG ("previous= " << previous << ", current=" << current);
428     NS_ASSERT (current >= previous);
429     if (payloadMode.GetModulationClass () != WIFI_MOD_CLASS_S1G)
430     {
431       //Case 1: previous and current after playload start: nothing to do
432       if (previous >= plcpPayloadStart)
```

```
433         {
434           psr *= 1;
435           NS_LOG_DEBUG ("Case 1 - previous and current after playload start: nothing to
              ↪ do");
436         }
437       //Case 2: previous is in HT-SIG or in HT training: Non HT will not enter here since
          ↪ it didn't enter in the last two and they are all the same for non HT
438       else if (previous >= plcpHsigHeaderStart)
439         {
440           NS_ASSERT ((preamble != WIFI_PREAMBLE_LONG) && (preamble !=
              ↪ WIFI_PREAMBLE_SHORT));
441           //Case 2a: current after payload start
442           if (current >= plcpPayloadStart)
443             {
444               psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
445                                                               noiseInterferenceW,
446                                                               htHeaderMode),
447                                                 plcpPayloadStart - previous,
448                                                 htHeaderMode);
449
450               NS_LOG_DEBUG ("Case 2a - previous is in HT-SIG or in HT training and current
                  ↪ after payload start: mode=" << htHeaderMode << ", psr=" << psr);
451             }
452           //Case 2b: current is in HT-SIG or in HT training
453           else
454             {
455               psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
456                                                               noiseInterferenceW,
457                                                               htHeaderMode),
458                                                 current - previous,
459                                                 htHeaderMode);
460
461               NS_LOG_DEBUG ("Case 2b - previous is in HT-SIG or in HT training and current
                  ↪ is in HT-SIG or in HT training: mode=" << htHeaderMode << ", psr=" <<
                  ↪ psr);
462             }
463         }
464       //Case 3: previous in L-SIG: GF will not reach here because it will execute the
          ↪ previous if and exit
465       else if (previous >= plcpHeaderStart)
466         {
467           NS_ASSERT (preamble != WIFI_PREAMBLE_HT_GF);
468           //Case 3a: current after payload start
469           if (current >= plcpPayloadStart)
470             {
471               //Case 3ai: Non HT format (No HT-SIG or Training Symbols)
472               if (preamble == WIFI_PREAMBLE_LONG || preamble == WIFI_PREAMBLE_SHORT)
473                 {
```

```
474                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
475                                                                 noiseInterferenceW,
476                                                                 headerMode),
477                                                   plcpPayloadStart - previous,
478                                                   headerMode);
479
480                 NS_LOG_DEBUG ("Case 3ai - previous in L-SIG and current after payload
                    ↪  start: mode=" << headerMode << ", psr=" << psr);
481               }
482             //Case 3aii: HT mixed format
483             else
484               {
485                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
486                                                                 noiseInterferenceW,
487                                                                 htHeaderMode),
488                                                   plcpPayloadStart - plcpHsigHeaderStart,
489                                                   htHeaderMode);
490
491                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
492                                                                 noiseInterferenceW,
493                                                                 headerMode),
494                                                   plcpHsigHeaderStart - previous,
495                                                   headerMode);
496
497                 NS_LOG_DEBUG ("Case 3aii - previous in L-SIG and current after payload
                    ↪  start: HT mode=" << htHeaderMode << ", non-HT mode=" << headerMode <<
                    ↪  ", psr=" << psr);
498               }
499           }
500         //Case 3b: current in HT-SIG or in HT training symbol. Non HT will not come here
            ↪  since it went in previous if or if the previous if is not true this will be
            ↪  not true
501         else if (current >= plcpHsigHeaderStart)
502           {
503             NS_ASSERT ((preamble != WIFI_PREAMBLE_LONG) && (preamble !=
                ↪  WIFI_PREAMBLE_SHORT));
504
505             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
506                                                             noiseInterferenceW,
507                                                             htHeaderMode),
508                                               current - plcpHsigHeaderStart,
509                                               htHeaderMode);
510             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
511                                                             noiseInterferenceW,
512                                                             headerMode),
513                                               plcpHsigHeaderStart - previous,
514                                               headerMode);
515
```

```
516             NS_LOG_DEBUG ("Case 3b - previous in L-SIG and current in HT-SIG or in HT
                ↪  training symbol: HT mode=" << htHeaderMode << ", non-HT mode=" <<
                ↪  headerMode << ", psr=" << psr);
517           }
518         //Case 3c: current with previous in L-SIG
519         else
520           {
521             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
522                                                             noiseInterferenceW,
523                                                             headerMode),
524                                               current - previous,
525                                               headerMode);
526
527             NS_LOG_DEBUG ("Case 3c - current with previous in L-SIG: mode=" << headerMode
                ↪  << ", psr=" << psr);
528           }
529       }
530     //Case 4: previous is in the preamble works for all cases
531     else
532       {
533         if (current >= plcpPayloadStart)
534           {
535             //Non HT format (No HT-SIG or Training Symbols)
536             if (preamble == WIFI_PREAMBLE_LONG || preamble == WIFI_PREAMBLE_SHORT)
537               {
538                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
539                                                                 noiseInterferenceW,
540                                                                 headerMode),
541                                                   plcpPayloadStart - plcpHeaderStart,
542                                                   headerMode);
543
544                 NS_LOG_DEBUG ("Case 4a - previous is in the preamble: mode=" <<
                    ↪  headerMode << ", psr=" << psr);
545               }
546             //HT format
547             else
548               {
549                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
550                                                                 noiseInterferenceW,
551                                                                 htHeaderMode),
552                                                   plcpPayloadStart - plcpHsigHeaderStart,
553                                                   htHeaderMode);
554
555                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
556                                                                 noiseInterferenceW,
557                                                                 headerMode),
558                                                   plcpHsigHeaderStart - plcpHeaderStart,
                                                    ↪  //HT GF: plcpHsigHeaderStart -
                                                    ↪  plcpHeaderStart = 0
```

```
559                                                 headerMode);
560
561                 NS_LOG_DEBUG ("Case 4a - previous is in the preamble: HT mode=" <<
                    ↪  htHeaderMode << ", non-HT mode=" << headerMode << ", psr=" << psr);
562               }
563           }
564         //non HT will not come here
565         else if (current >= plcpHsigHeaderStart)
566           {
567             NS_ASSERT ((preamble != WIFI_PREAMBLE_LONG) && (preamble !=
                  ↪  WIFI_PREAMBLE_SHORT));
568
569             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
570                                                             noiseInterferenceW,
571                                                             htHeaderMode),
572                                               current - plcpHsigHeaderStart,
573                                               htHeaderMode);
574
575             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
576                                                             noiseInterferenceW,
577                                                             headerMode),
578                                               plcpHsigHeaderStart - plcpHeaderStart, //HT
                                                  ↪  GF: plcpHsigHeaderStart -
                                                  ↪  plcpHeaderStart = 0
579                                               headerMode);
580
581             NS_LOG_DEBUG ("Case 4b - previous is in the preamble: HT mode=" <<
                  ↪  htHeaderMode << ", non-HT mode=" << headerMode << ", psr=" << psr);
582           }
583         //GF will not come here
584         else if (current >= plcpHeaderStart)
585           {
586             NS_ASSERT (preamble != WIFI_PREAMBLE_HT_GF);
587
588             psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
589                                                             noiseInterferenceW,
590                                                             headerMode),
591                                               current - plcpHeaderStart,
592                                               headerMode);
593
594             NS_LOG_DEBUG ("Case 4c - previous is in the preamble: mode=" << headerMode <<
                  ↪  ", psr=" << psr);
595           }
596       }
597   }
598   else
599   {
600       //Case 1: previous and current after playload start: nothing to do
```

```
601         if (previous >= plcpPayloadStart)
602         {
603             psr *= 1;
604             NS_LOG_DEBUG ("Case 1 - previous and current after playload start: nothing to
                ↪  do");
605         }
606         //Case 2: previous is in S1G-A or in S1G training or in S1G-B. only SIG_LONG
            ↪  enter
607         else if (previous >= plcpSigAStart)
608         {
609             NS_ASSERT ((preamble != WIFI_PREAMBLE_S1G_SHORT) && (preamble !=
                ↪  WIFI_PREAMBLE_S1G_1M));
610             //Case 2a: current after payload start
611             if (current >= plcpPayloadStart)
612             {
613                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
614                                                                 noiseInterferenceW,
615                                                                 headerMode),
616                                                   plcpPayloadStart - previous,
617                                                   headerMode);
618
619                 NS_LOG_DEBUG ("Case 2a - previous is in S1G-A or in S1G training or in
                    ↪  S1G-B and current after payload start: mode=" << htHeaderMode << ",
                    ↪  psr=" << psr);
620             }
621             //Case 2b: current is in S1G-A or in S1G training or in S1G-B
622             else
623             {
624                 psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
625                                                                 noiseInterferenceW,
626                                                                 headerMode),
627                                                   current - previous,
628                                                   headerMode);
629
630                 NS_LOG_DEBUG ("Case 2b - previous is in S1G-A or in S1G training or in
                    ↪  S1G-B and current is S1G-A or in S1G training or in S1G-B: mode=" <<
                    ↪  htHeaderMode << ", psr=" << psr);
631             }
632         }
633         //Case 3: previous in LTF or SIG: S1G_LONG will not reach here because it will
            ↪  execute the previous if and exit
634         else if (previous >= plcpHeaderStart)
635         {
636             NS_ASSERT (preamble != WIFI_PREAMBLE_S1G_LONG);
637             //Case 3a: current after payload start
638             if (current >= plcpPayloadStart)
639             {
640
```

```
641                     psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
642                                                                     noiseInterferenceW,
643                                                                     headerMode),
644                                                      plcpPayloadStart - previous,
645                                                      headerMode);
646
647                     NS_LOG_DEBUG ("Case 3aii - previous in LTF and current after payload
                        ↪  start: HT mode=" << htHeaderMode << ", non-HT mode=" << headerMode
                        ↪  << ", psr=" << psr);
648
649                 }
650                 //Case 3b: current with previous in LTF or SIG
651                 else
652                 {
653                     psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
654                                                                     noiseInterferenceW,
655                                                                     headerMode),
656                                                       current - previous,
657                                                       headerMode);
658
659                     NS_LOG_DEBUG ("Case 3c - current with previous in LTF or SIG: mode=" <<
                        ↪  headerMode << ", psr=" << psr);
660                 }
661             }
662             //Case 4: previous is in the preamble works for all cases
663             else
664             {
665                 if (current >= plcpPayloadStart)
666                 {
667                         psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
668                                                                         noiseInterferenceW,
669                                                                         headerMode),
670                                                           plcpPayloadStart - plcpHeaderStart,
                                                          ↪  //For S1G_LONG,  plcpHeaderStart
                                                          ↪  equals plcpSigAStart
671                                                           headerMode);
672
673                         NS_LOG_DEBUG ("Case 4a - previous is in the preamble: mode=" <<
                            ↪  headerMode << ", psr=" << psr);
674
675                 }
676                 //only S1G_LONG come here
677                 else if (current >= plcpSigAStart)
678                 {
679                     NS_ASSERT ((preamble != WIFI_PREAMBLE_S1G_SHORT) && (preamble !=
                        ↪  WIFI_PREAMBLE_S1G_1M));
680
681                     psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
```

```
                                                                noiseInterferenceW,
                                                                headerMode),
                                                    current - plcpSigAStart,
                                                    headerMode);

                    NS_LOG_DEBUG ("Case 4b - previous is in the preamble: mode=" << headerMode
                    ↪  << ", psr=" << psr);
                }
                //S1G_LONG will not come here
                else if (current >= plcpHeaderStart)
                {
                    NS_ASSERT (preamble != WIFI_PREAMBLE_S1G_LONG);

                    psr *= CalculateChunkSuccessRate (CalculateSnr (powerW,
                                                                noiseInterferenceW,
                                                                headerMode),
                                                    current - plcpHeaderStart,
                                                    headerMode);

                    NS_LOG_DEBUG ("Case 4c - previous is in the preamble: mode=" << headerMode
                    ↪  << ", psr=" << psr);
                }
            }
        }

        noiseInterferenceW += (*j).GetDelta ();
        previous = (*j).GetTime ();
        j++;
      }

  double per = 1 - psr;

  if(swiptH != NULL)
  {
    swiptH->InterfPlcpHeader (noiseInterferenceW, PlcpAndHeaderDuration, swiptH);

    int32_t nodeId = swiptH->GetNode ()->GetId ();

  }
  return per;
}

double
InterferenceHelper::DbmToW (double dBm)
{
  NS_LOG_FUNCTION(this << dBm);
  double mW = std::pow (10.0, dBm / 10.0);
  return mW / 1000.0;
```

```
728 }
729
730 struct InterferenceHelper::SnrPer
731 InterferenceHelper::CalculatePlcpPayloadSnrPer (Ptr<InterferenceHelper::Event> event,
732                                                 double psFactor,
733                                                 Ptr<SwiptHarvester> swiptH)
734 {
735
736   NiChanges ni;
737   double noiseInterferenceW = CalculateNoiseInterferenceW (event, &ni);
738   Time noiseInterferenceWDuration = event->GetDuration ();
739   Time now = Simulator::Now();
740
741   double snr = CalculateSnr (event->GetRxPowerW (),
742                              noiseInterferenceW,
743                              event->GetPayloadMode ());
744   /* calculate the SNIR at the start of the packet and accumulate
745    * all SNIR changes in the snir vector.
746    */
747
748   double per = CalculatePlcpPayloadPer (event, &ni, noiseInterferenceWDuration, swiptH);
749
750   struct SnrPer snrPer;
751   snrPer.snr = snr;
752   snrPer.per = per;
753
754   return snrPer;
755 }
756
757 struct InterferenceHelper::SnrPer
758 InterferenceHelper::CalculatePlcpHeaderSnrPer (Ptr<InterferenceHelper::Event> event,
759                                                Time PlcpAndHeaderDuration,
760                                                double psFactor,
761                                                Ptr<SwiptHarvester> swiptH)
762 {
763
764   NiChanges ni;
765   double noiseInterferenceHeaderW = CalculateNoiseInterferenceW (event, &ni);
766   Time now = Simulator::Now();
767
768   double snr = CalculateSnr (event->GetRxPowerW (),
769                              noiseInterferenceHeaderW,
770                              WifiPhy::GetPlcpHeaderMode (event->GetPayloadMode (),
                               ↪  event->GetPreambleType ()));
771
772
773   /* calculate the SNIR at the start of the plcp header and accumulate
774    * all SNIR changes in the snir vector.
```

```
775     */
776
777   double per = CalculatePlcpHeaderPer (event, &ni, PlcpAndHeaderDuration, swiptH);
778
779   //Time noiseInterferenceHeaderDuration = GetEnergyDuration (DbmToW (-110));
780   //Time noiseInterferenceHeaderDuration = event->GetDuration ();
781
782   struct SnrPer snrPer;
783   snrPer.snr = snr;
784   snrPer.per = per;
785
786   return snrPer;
787 }
788
789 void
790 InterferenceHelper::EraseEvents (void)
791 {
792   m_niChanges.clear ();
793   m_rxing = false;
794   m_firstPower = 0.0;
795 }
796
797 InterferenceHelper::NiChanges::iterator
798 InterferenceHelper::GetPosition (Time moment)
799 {
800   return std::upper_bound (m_niChanges.begin (), m_niChanges.end (), NiChange (moment, 0));
801 }
802
803 void
804 InterferenceHelper::AddNiChangeEvent (NiChange change)
805 {
806   m_niChanges.insert (GetPosition (change.GetTime ()), change);
807 }
808
809 void
810 InterferenceHelper::NotifyRxStart ()
811 {
812   NS_LOG_FUNCTION (this);
813   m_rxing = true;
814 }
815
816 void
817 InterferenceHelper::NotifyRxEnd ()
818 {
819   NS_LOG_FUNCTION (this);
820   m_rxing = false;
821 }
822
```

```
823 } //namespace ns3
```