Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Design of a power-saving strategy for a Collaborative Wireless Sensor Network of Multi-core Embedded Systems

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electronics, Major in Embedded Systems

Randy Steven Céspedes Deliyore

Cartago, Setiembre 20, 2022

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.

Randy Steven Céspedes Deliyore

Cartago, Setiembre 20, 2022

Céd: 3-0463-0326

This work is licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Maestría Académica en Eletrónica
Trabajo Final de Graduación
Acta de Aprobación de Tesis

Defensa de Trabajo Final de Graduación
Requisito para obtar por el título de Máster en Ingeniería Electrónica
Grado Académico de Magister Scientiae

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado **Design of a power-saving strategy for a Collaborative Wireless Sensor Network of Multi-core Embedded Systems**, realizado por **Randy Steven Céspedes Deliyore** Carné: **201054417**, y hace constar que cumple con las normas establecidas por la Unidad Interna de Posgrados de la Escuela de Ingeniería Eletrónica del Instituto Técnologico de Costa Rica.

Miembros del Tribunal Evaluador

CESAR ORLANDO GARITA RODRIGUEZ (FIRMA)
Firmado digitalmente por CESAR ORLANDO GARITA RODRIGUEZ (FIRMA)
Fecha: 2022.09.21 14:34:17 -06'00'

Dr. César Garita Rodríguez
Profesor Lector

SERGIO ALBERTO ARRIOLA VALVERDE (FIRMA)
Firmado digitalmente por SERGIO ALBERTO ARRIOLA VALVERDE (FIRMA)
Fecha: 2022.09.21 21:38:53 -06'00'

M. Sc. Sergio Arriola Valverde
Profesor Lector

GIANNINA ORTIZ QUESADA (FIRMA)
Firmado digitalmente por GIANNINA ORTIZ QUESADA (FIRMA)
Fecha: 2022.09.21 16:37:56 -06'00'

M.Sc. Giannina Ortíz Quesada
Evaluadora Externa

ANIBAL IGNACIO RUIZ BARQUERO (FIRMA)
Firmado digitalmente por ANIBAL IGNACIO RUIZ BARQUERO (FIRMA)
Fecha: 2022.09.21 13:49:21 -06'00'

M. Sc. Aníbal Ruiz Barquero
Profesor Asesor

Cartago, Setiembre 20, 2022

Resumen

Structural Health Monitoring (SHM) es un campo de ingeniería que enfrenta el hecho de que los sensores tienden a ser difíciles de acceder e instalar, lo que hace que los nodos de bajo consumo de energía en una red de sensores inalámbricos ayuden a facilitar su implementación, instalación y mantenimiento. Este trabajo presenta una propuesta de diseño para una estrategia de ahorro de energía para una red colaborativa utilizada en SHM para aumentar la duración de la batería de los nodos y permitir futuros despliegues con energía solar. Esta estrategia incluye; recomendaciones de perfilado de código, mediciones de carga de procesador y de voltaje-corriente para un enfoque de optimización de nivel de instrucción, implementación de manejo dinámico de potencia (DPM por sus siglás en inglés Dynamic Power Management), escalamiento dinámico de tensión y frecuencia (DFVS por sus siglás en inglés Dynamic Voltage and Frequency Scaling), mejoras en el protocolo de comunicación de la red colaborativa y evaluación tecnológica de redes de gran area de baja potencia (LPWAN por sus siglás en inglés Low Power Wide Area Network) en forma de comunicaciones LoRa (del inglés Long Range o largo alcance en español). Esta estrategia aumenta la autonomía de la batería de los nodos de la red hasta un 574,74 %, considerando tres horas y 31 minutos como línea base, alcanzando hasta veintitrés horas y 45 minutos; además, puede reducir la utilización de energía media en un 26% en los dispositivos maestros que para este trabajo se denominan dispositivos primarios y en un 28,58% para los dispositivos esclavos, tratados como secundarios. También redujo la cantidad de datos a transmitir hasta un 80% de 11,72 KB a 2,34 KB por hora por nodo. Además, se presentan algunos resultados significativos para las mejoras en utilización de la CPU y la reducción del consumo de energía, también se identifican algunas áreas de interés para futuras mejoras.

Palabras Clave: collaborative, DPM, DVFS, SHM, embedded, IoT, low-power, WSN

Abstract

Structural Health Monitoring (SHM) is an engineering field that faces the fact that sensors tend to be arduous to access and install, making nodes of low power consumption in a wireless sensor network aids in facilitating its deployability, installability, and maintainability. This work presents a design proposal for a power-saving strategy for a collaborative network used in SHM to increase the nodes' battery life and allow future solar power deployments. This strategy includes code profiling recommendations, CPU Load measurements and voltage-current measurements for an instruction-level optimization approach, implementation of Dynamic Power Management (DPM), Dynamic Voltage and Frequency Scaling (DFVS), collaborative network communication protocol improvements, and evaluation of Low-Power Wide Area Network (LPWAN) technology in the form of LoRa (Long Range) communications. This strategy increases the battery autonomy of the network nodes up to 574.74 %, considering three hours and 31 minutes as a baseline, achieving up to twenty-three hours and 45 minutes; moreover, it can reduce the mean power utilization by 26% in masters devices which for this work are called primary devices and 28.58% for slaves devices, treated as the secondary ones. It also reduced the amount of the data to be transmitted to up 80% from 11.72 KB to 2.34 KB per hour per node. Furthermore, some significant results for CPU utilization and power consumption reduction are presented, as well as identifying some areas of interest for future improvements.

Keywords: collaborative, DPM, DVFS, SHM, embedded, IoT, low-power, WSN

a mi hermosa familia

Agradecimientos

Quiero tomarme la libertad de dar el agradecimiento a varias personas que demostraron su apoyo incondicional a lo largo de esta gran etapa. Sin duda mi fe, fue uno de los pilares de mi motivación y sirvió de guía en los momentos difíciles a lo largo de este extenso trabajo que duro casi dos años. Quiero agradecer al MSc. Aníbal Ruíz Barquero, quien fue mi asesor de tesis, sin duda en muchas ocasiones actuó más allá de sus responsabilidades por su interés geniudo en que este proyecto fuera un éxito. Agradezco el tiempo que se tomó de sus tardes, noches, fines de semana y vacaciones para ayudarme a buscar alternativas a problemas que surgieron durante el desarrollo de la investigación y sus muy valiosos consejos. Quiero agradecer también mis dos lectores el Dr. César Garita Rodríguez y el MSc. Sergio Arriola Valverde por su apoyo durante la investigación y su muy valiosa guía para hacer este trabajo relevante y transferible en el futuro. A la MSc. Giannina Ortiz Quesada y al CIVCO por ser un apoyo incondicional a lo largo del desarrollo de la investigación y por gran ayuda económica para la compra de hardware y de horas de trabajo para asistentes. Quiero agradecer también al Ing. José Daniel Montoya y al Ing. Victor Fuentes quienes colaboraron enormemente durante el desarrollo de la investigación. Al MSc. Anibal Coto y al Dr. Johan Carvajal como coordinadores del programa de Maestría en Electrónica, quienes siempre estuvieron anuentes a colaborar cuando fuera necesario. Al señor Ronald Valverde por asegurarse de que el servidor de eBridge estuviera funcional.

A mi futura esposa Cynthia Isabel Taylor Herrera, quien me acompaño desde el inicio de la loca idea de querer estudiar una maestría, hasta el periodo de la culminación de esta. Sin duda sin su apoyo emocional, sus excelentes conocimientos de LaTEX y sus inagotables ganas de animarme en los momentos más difíciles no hubiera logrado completar este tan largo proyecto.Me gusta pensar que este es un logro de ambos. A mi madre Dully Deliyore, quien me escuchó en los momentos difíciles y con sus palabras de aliento me dio fuerzas, a mi padre Guillermo Céspedes, y a mi padrastro Luis Duran, quienes siempre me tuvieron en sus oraciones y me dieron porras para que siguiera adelante. A mi suegra María del Carmen Herrera que siempre se preocupo porque estuviera en excelentes condiciones para poder trabajar y se aseguro de que no tuviera que gastar tiempo preparando comida. A mi tio Jorge Deliyore por siempre apoyarme en el estudio y motivarme a querer progresar, desde que me ayudo a poder formar parte del Colegio Científico. A mi hermano Jaime Palermo por siempre darme apoyo y escucharme. A mi madrina Marita Garita y a mi tío Harry Deliyore, por sus palabras de aliento y oraciones.

Quiero agradecer a National Instruments por darme apoyo económico, tiempo e inclusive el préstamo de equipo para poder realizar una gran cantidad de experimentos y tener mi propio setup en el laboratorio de la empresa. A mis managers durante el periodo de estudio el señor Coamín Cruz, la señora Erin Bray y el señor Christian Loew quienes fueron un increíble apoyo para hacer que esto fuera posible. A Michael, Alberto y Marcela de seguridad que me ayudaron a reiniciar equipo múltiples veces inclusive en la madrugada para que pudiera continuar con los experimentos.

A mis queridos amigos Hector, Ana, Hugo, Rafa, Daniela y Nit por darme apoyo durante este tan largo proceso y motivarme a seguir. A Hugo por ayudarme a traducir cosas del alemán. Al Dr. Pablo Alvarado por tomarse el tiempo de trabajar en proveer un formato de tesis en LaTEX y mis compañeros Andrés Gómez, Andrés González y Andrés Vargas por su colaboración durante los cursos y apoyo a resolver problemas con el formato de la documentación.

Randy Steven Céspedes Deliyore

Cartago, Setiembre 20, 2022

Contents

List of Figures **iv**

List of Tables **ix**

Nomenclature **xiii**

1 Introduction **1**

1.1 Objectives and Document Structure . . . 3

2 Review of Related Literature **5**

2.1 Power Consumption in Embedded Systems . . . 5

2.1.1 Power Consumption Model . . . 6

2.2 Power Measurement in Embedded Systems . . . 11

2.3 Power Saving Strategies in Embedded Systems . . . 12

2.3.1 Dynamic Voltage and Frequency Scaling (DVFS) . . . 14

2.4 PET-aware Scheduling Algorithms/Strategies . . . 17

2.4.1 Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems . . . 17

2.4.2 Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination . . . 18

2.4.3 Energy-Aware Frame-Based Scheduling (EAFBFS) . . . 18

2.4.4 Improved least loss energy density algorithm (ILLED) . . . 19

2.4.5 Heterogeneous energy-aware real-time scheduler (HEARS) . . . 20

2.4.6 Dynamic energy-saving scheduling algorithm for a sporadic task in real-time embedded systems (DESSAST-RTES) . . . 20

2.5 Comparison of PET-ware Scheduling Algorithms . . . 21

2.6 Other Power-Saving Strategies . . . 22

2.6.1 Operating System Power Saving Strategies . . . 22

2.6.2 Software Level Optimizations . . . 22

3 Methodology **23**

3.1 Hardware and Software Update . . . 23

3.2 RPI3 B+ vs RPI4 B Comparison . . . 25

3.3 Hardware setup for experiments . . . 26

3.4 Code Profiling Strategy . . . 27

3.4.1 Reviewing Existing Profilers . . . 29
3.4.2 Creating a Custom Profiler . . . 30
3.5 Power Measurement of eBridge System . . . 32
3.5.1 Batteries used for testing . . . 34

4 Analysis of the eBridge Network **36**
4.1 eBridge 2019 Node Analysis . . . 36
4.1.1 eBridge 2019 code Line Profiling Results . . . 37
4.1.2 eBridge 2019 Master/Primary CPU, Events and Power Profiling . . 38
4.1.3 eBridge 2019 Slave/Secondary CPU,Events and Power Profiling . . 46
4.2 eBridge 2019 Battery Tests . . . 56
4.2.1 Battery test for Master/Primary . . . 56
4.2.2 Battery test for Slave/Secondary . . . 57
4.3 eBridge 2019 Task Interaction Analysis . . . 58
4.4 eBridge 2021 HW Configuration Standby Power Consumption . . . 61
4.5 Summary of Areas of Improvement for the EBridge 2019 Network . . . 62

5 Design **64**
5.1 Software Level Optimizations . . . 64
5.1.1 SetDateTime Refactoring . . . 64
5.1.2 Cellular Network Interface Handling . . . 65
5.1.3 Receive Thread Optimizations . . . 65
5.1.4 Device Class Optimizations . . . 66
5.2 eBridge Protocol Upgrades . . . 67
5.2.1 Device Configuration . . . 67
5.2.2 Slave/Secondary Node Message Handling . . . 68
5.2.3 Alarm Handling . . . 68
5.2.4 Consolidating Reports . . . 69
5.3 eBridge Hardware optimizations for power saving . . . 70
5.3.1 Use of IoT LoRa Network . . . 71
5.3.2 Dynamic Power Management and DVFS Proposal . . . 74

6 Validation **78**
6.1 eBridge 2021 Node Code Analysis . . . 78
6.1.1 eBridge 2021 Line Profiling . . . 78
6.1.2 eBridge 2021 Master/Primary CPU and Events Profiling . . . 79
6.1.3 eBridge 2021 Slave/Secondary CPU and Events Profiling . . . 85
6.2 eBridge 2021 Battery Tests . . . 90
6.2.1 Battery tests for Master/Primary eBridge 2021 code . . . 90
6.2.2 Battery tests for Slave/Secondary eBridge 2021 code . . . 95
6.3 Dynamic and Voltage and Frequency Scaling . . . 95
6.3.1 Effects of Dynamic Frequency Scaling . . . 97
6.3.2 CPU Voltage Changes . . . 97
6.4 Dynamic Power Management . . . 100

6.5 eBridge 2022 Node with LoRa Analysis . . . 101
6.5.1 eBridge 2022 with LoRa Line Profiling . . . 102
6.5.2 eBridge 2022 with LoRa Master/Primary CPU and Events Profiling 103
6.5.3 eBridge 2022 with LoRa Slave/Secondary CPU and Events Profiling 106
6.6 eBridge 2022 with LoRa Battery Tests . . . 109
6.6.1 Battery test for Master/Primary . . . 109
6.6.2 Battery test for Slave/Secondary . . . 110

7 Conclusions 111
7.1 Conclusions . . . 111
7.1.1 eBridge 2019 network analysis Conclusions . . . 111
7.1.2 Design Conclusions . . . 112
7.1.3 eBridge 2021 network analysis Conclusions . . . 112
7.1.4 DPM and DVFS on the eBridge Network Conclusions . . . 112
7.1.5 LoRA eBridge 2022 with LoRa Conclusions . . . 113
7.2 Future Work . . . 113

Bibliography 114

List of Figures

1.1 eBridge Project Timeline. Adapted from [3]. 2
1.2 eBridge Collaborative Network concept. Adapted from [4]. 3
1.3 eBridge Sensor Node at 2019 [10], [12]. 3
1.4 eBridge Network General Architecture. Adapted from [12]. 4

2.1 Dividing a processor in Functional Units[25]. 9
2.2 Dividing a processor into Functional Units[25]. 9
2.3 Power vs execution time of a task τ_i [25]. 12
2.4 Power vs execution time for multiple consecutive executions of task τ_i [25]. 12
2.5 Dynamic Voltage Scaling (DVS). 14
2.6 Dynamic Frequency Scaling (DFS). (a) Task executing with higher CPU Frequency. (b) Task executing with Lower CPU Frequency. 15
2.7 Idle times I_i of multiple tasks (τ_1,τ_2,τ_3). Adapted from [27]. 18
2.8 Software and Hardware Layer in Power Consumption. Adapted from [40]. . 22

3.1 eBridge Sensor 2019 vs 2021 Nodes. Adapted from [10], [12]. 24
3.2 WaveShare SIM7600G-H 4G Hat [42]. 25
3.3 Cortex A72 (ARM v8) Processor SoC Diagram [43]. 26
3.4 BCM2711 SoC Diagram [44]. 27
3.5 BCM2837B0 SoC Diagram [45]. 28
3.6 BCM2711 SoC Diagram [45]. 28
3.7 Current Hardware Setup eBridge. 29
3.8 Task execution time vs CPU Load profiling strategy. 30
3.9 Event Logger Concept 31
3.10 CPU Load Plot Concept. 31
3.11 Py-Spy example [51]. 32
3.12 Code Optimization Strategy. 32
3.13 Power Measurement Hardware Setup for eBridge Sensor Node. 33
3.14 PXIe-4309 Flexible Resolution PXIe Analog Input Module [55]. 33
3.15 ICR18650 4400mAh 3.7V Battery[56] 34
3.16 MI Power Bank and Wall Charger CBQ01ZM - Capacity 5200 mAh [57] . . 35
3.17 MI Power Bank 3 Ultra Compact - Capacity 10 000 mAh [58] 35

4.1 CPU Load (%) for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 4.1 . . . 38
4.2 Tasks Gantt plot eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.1 . . . 40
4.3 Tasks Gantt Zoomed plot eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+. Details of the test in table 4.1 . . . 40
4.4 Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.1 . . . 41
4.5 Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.1 . . . 42
4.6 CPU Load Box (%) plot for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 4.1 . . . 43
4.7 CPU Load % for eBridge 2019 code running as master/primary node running in eBridge 2021 HW configuration in RPI4 B with f = 1.5 GHz. Details of the test in table 4.2 . . . 43
4.8 Gantt Plot for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2 . . . 45
4.9 Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2 . . . 46
4.10 Tasks Gantt Plot vs Power (W) of eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz - I. Details of the test in table 4.1 . . . 47
4.11 Tasks Gantt Plot vs Power (W) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz - II. Details of the test in table 4.1 . . . 48
4.12 Tasks Gantt Plot vs Power (W) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2 . . . 49
4.13 CPU Load (%) eBridge 2019 code running as slave/secondary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.7 . . . 51
4.14 Tasks Gantt Plot eBridge 2019 code running as slave/secondary node running in eBridge 2021 HW in RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.7 . . . 51

4.15 Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as slave/secondary node running in eBridge 2021 HW in RPI3 B+ with f = 1.5 GHz. Details of the test in table 4.7 . . . 52
4.16 CPU Load for eBridge 2019 code running as a slave/secondary node running in eBridge 2021 HW in RPI4 B with f = 1.5 GHz. Details of the test in table 4.8 . . . 52
4.17 Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as a slave/secondary node running in eBridge 2021 HW in RPI4 B with f = 1.5 GHz. Details of the test in table 4.8 . . . 54
4.18 Tasks Gantt Plot vs Power (W) eBridge 2019 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.8 . . . 56
4.19 Tasks $\Gamma = \{\tau_0, \tau_2, \tau_3, \tau_4\}$ $C_{i,\tau_i}^{\vec{f}}$, and $E_{i,\tau_i}^{\vec{f}}$ Interaction Qualitative Diagram for a master/primary Node. . . . 58
4.20 Zoomed Gantt Plot for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2 . . . 59
4.21 Zoomed Tasks $\Gamma = \{\tau_0, \tau_3, \tau_4\}$ $C_{i,\tau_i}^{\vec{f}}$, and $E_{i,\tau_i}^{\vec{f}}$ Interaction Qualitative Diagram for a master/primary Node with Highlighted Power . . . 60
4.22 Zoomed Tasks $\Gamma = \{\tau_1, \tau_3, \tau_4\}$ $C_{i,\tau_i}^{\vec{f}}$, and $E_{i,\tau_i}^{\vec{f}}$ Interaction Qualitative Diagram for a slave/secondary Node . . . 60
4.23 Standby Power [W] vs Time (s) for the eBridge 2021 HW configuration of RPI3 B+ and RPI3 B. . . . 61

5.1 eBridge Embedded Software Now UML Classes Hierarchy . . . 67
5.2 eBridge Node Configuration Policy. . . . 68
5.3 eBridge Messages from Slave/Secondary to Master Failure. . . . 69
5.4 eBridge Alarm Handle Procedure Optimization. . . . 70
5.5 eBridge Consolidating Reports Procedure. . . . 71
5.6 eBridge Report Management. . . . 72
5.7 Bridge Structures in Costa Rica by State - Survey 2014-2018 [3]. . . . 73
5.8 LTE Coverage in Costa Rica [66]. . . . 74
5.9 Wireless access geographic coverage [67]. RFID (Radio Frequency Identification), NFC (Near Field Communication). . . . 75
5.10 Adafruit RFM95W LoRa Radio Transceiver [70]. . . . 76
5.11 eBridge 2022 Hardware Configurations version A and B. . . . 76
5.12 eBridge 2022 Class UML Diagram. . . . 77

6.1 CPU Load (%) for eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 6.1. . . . 80

6.2 CPU Load Box (%) plot for eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 6.1 . . . 80
6.3 Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.1 . . . 81
6.4 Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz, focus on Get Median τ_4. Details of the test in table 6.1 . . . 82
6.5 Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 6.1 . . . 82
6.6 CPU Load (%) for eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details in table 6.2 . . . 84
6.7 Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.2 85
6.8 Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 6.2 . . . 86
6.9 CPU Load (%) for eBridge 2021 code running as slave/secondary a sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 6.7 . . . 89
6.10 Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.7 . . . 89
6.11 Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 6.7 . . . 90
6.12 CPU Load (%) for eBridge 2021 code running as slave/secondary a sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details in table 6.8 . . . 92
6.13 Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.8 . . . 92
6.14 Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 6.8 . . . 93
6.15 Tasks Gantt Plot vs CPU Load (%) vs CPU Freq (MHz) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with DVFS, focused on Configuration τ_5,and τ_6. . . . 96

6.16 Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with DVFS, focused on Configuration τ_5,and τ_6 98
6.17 Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI3 B with DVFS, focused on XBee Send τ_1. . . . 99
6.18 Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2022 code running as a master/primary sensor node running in eBridge 2022 HW with RPI3 B+ with DVFS and DPM. 104
6.19 Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2022 code running as a slave/secondary sensor node running in eBridge 2022 HW with RPI4B with DVFS and DPM. 107

List of Tables

2.1 Algorithmic and Architectural Software Parameters [25] 8
2.2 Comparison Variables for PET-ware Scheduling Algorithms 21
2.3 Comparison of PET-ware Scheduling Algorithms 21

3.1 Previous vs Current eBridge Sensor Node Comparison 24
3.2 Frequency Bands Supported by SIM7600G-H 25
3.3 Profiler Comparison 30
3.4 Summary of Batteries used for testing 35

4.1 Measurement Summary eBridge code 2019 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI3 B+ 39
4.2 Measurement Summary eBridge code 2019 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI4 B 44
4.3 Comparison of $\vec{C_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{C_i^f}$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2019 in eBridge 2021 HW configuration as master/primary. Test details in tables 4.1 and 4.2 45
4.4 Comparison of Code CPU Load (%) statistical information of the eBridge 2019 code running on an RPI3 B+ with f = 1.4 GHz vs an RPI4 B as Master/Primary with f = 1.5 GHz. Test details in tables 4.1 and 4.2 46
4.5 Comparison of $\vec{E_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{E_i^f}$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2019 code as master/primary in eBridge 2021 HW configuration. Test details in tables 4.1 and 4.2. 47
4.6 Power Comparison (W) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW in both RPI3 B+ and RPI4 B. Test details in tables 4.1 and 4.2 48
4.7 Measurement Summary eBridge 2019 code running as Slave/Secondary Sensor Node in eBridge 2021 hardware configuration on RPI4 B+. 50
4.8 Measurement Summary eBridge 2019 code running as Slave/Secondary Sensor Node in eBridge 2021 hardware configuration on RPI4 B. 53
4.9 Comparison of Code CPU Load (%) statistical information of the eBridge 2019 code running as a slave/secondary node running in eBridge 2021 HW in RPI3 B+ and RPI4 B 53

4.10 Comparison of $\vec{C_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{C_i^f}$ with f=1.5 GHz for RPI4 B vectors statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2019 in eBridge 2021 HW configuration as slave/secondary. Test details in tables 4.7 and 4.8 54
4.11 Power Comparison (W) eBridge 2019 Code RPI3 B+ vs RPI4 B running as Slave/Secondary in eBridge 2021 HW. Test details in tables 4.7 and 4.8 55
4.12 Comparison of $\vec{E_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{E_i^f}$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2019 code as slave/secondary in eBridge 2021 HW configuration. Test details in tables 4.7 and 4.8 55
4.13 Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2019 code as master/primary in eBridge 2021 HW configuration. 57
4.14 Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2019 code as slave/secondary in eBridge 2021 HW configuration. 63
4.15 eBridge 2021 HW Configuration in RPI3 B+ and RPI4 B Power Consumption in Standby 63
4.16 Power Comparison (W) eBridge 2021 HW Configuration on code RPI3 B+ vs RPI4 B. Test details in table 4.15. 63

6.1 Measurement Summary eBridge code 2021 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI3 B+ 79
6.2 Measurement Summary eBridge code 2021 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI4 B 83
6.3 Comparison of Code CPU Load (%) statistical information of the eBridge 2021 code running as a master/primary node running in eBridge 2021 HW in RPI3 B+ and RPI4 B 84
6.4 Power Comparison (W) eBridge 2021 Optimized code running as master/primary sensor node running in eBridge 2021 HW in both RPI3 B+ and RPI4 B. Test details in tables 6.1 and 6.2 85
6.5 Comparison of $\vec{C_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{C_i^f}$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2021 Optimized code in eBridge 2021 HW configuration as master/primary. Test details in tables 6.1 and 6.2 86
6.6 Comparison of $\vec{E_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{E_i^f}$ with f=1.5 GHz for RPI4 B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2021 code as slave/secondary in eBridge 2021 HW configuration. Test details in tables 6.1 and 6.2 87
6.7 Measurement Summary eBridge code 2021 running as Slave/Secondary Sensor node in eBridge 2021 hardware configuration on RPI3 B+ 88
6.8 Measurement Summary eBridge code 2021 running as Slave/Secondary Sensor node in eBridge 2021 hardware configuration on RPI4 B 91

6.9 Comparison of Code CPU Load (%) statistical information of the eBridge 2021 code running as a slave/secondary node running in eBridge 2021 HW in RPI3 B+ and RPI4 B 91
6.10 Power Comparison (W) eBridge 2021 Optimized code running as slave/secondary sensor node running in eBridge 2021 HW in both RPI3 B+ and RPI4 B. Test details in tables 6.7 and 6.8 93
6.11 Comparison of $\vec{C_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{C_i^f}$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2021 Optimized code in eBridge 2021 HW configuration as slave/secondary. Test details in tables 6.1 and 6.2 94
6.12 Comparison of $\vec{E_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{E_i^f}$ with f=1.5 GHz for RPI4 B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2021 code as slave/secondary in eBridge 2021 HW configuration. Test details in tables 6.7 and 6.8 94
6.13 Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2021 code as master/primary in eBridge 2021 HW configuration. 95
6.14 Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2021 code as slave/secondary in eBridge 2021 HW configuration. 95
6.15 Mean Power [W] of RPI3 B+ and RPI4 B nodes after applying DFS settings compared to standby power 100
6.16 Mean Power [W] of RPI3 B+ and RPI4 B nodes after applying DVS undervoltage settings compared to standby power 100
6.17 Mean Power [W] of RPI3 B+ and RPI4 B nodes after applying DPM settings compared to standby power 102
6.18 Measurement Summary eBridge code 2022 running as Master/Primary Sensor node in eBridge 2022 hardware configuration A on RPI3 B+ 103
6.19 $\vec{E_i^f}$ with DVFS on RPI3 B+ vectors statistical information of $\Gamma = \{\tau_8, \tau_7, \tau_4, \tau_3, \tau_2, \tau_{10}\}$ of eBridge 2022 in eBridge 2022 HW configuration as master/primary and mean power. Test details in table 6.18. 105
6.20 $\vec{C_i^f}$ with DVFS on RPI3 B+ vectors statistical information of $\Gamma = \{\tau_8, \tau_7, \tau_4, \tau_3, \tau_2, \tau_{10}\}$ of eBridge 2022 in eBridge 2022 HW configuration as master/primary. Test details in table 6.18. 105
6.21 Measurement Summary eBridge code 2022 running as Master/Primary Sensor node in eBridge 2022 hardware configuration A on RPI3 B+ 106
6.22 $\vec{E_i^f}$ with DVFS on RPI4 B vectors statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_9\}$ of eBridge 2022 in eBridge 2022 HW configuration as slave/secondary and mean power. Test details in table 6.21. 108
6.23 $\vec{E_i^f}$ with DVFS and DPM in PI3 B+vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2021 Optimized code in eBridge 2021 HW configuration as slave/secondary. Test details in tables 6.1 and 6.2 108
6.24 Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2022 code as master/primary in eBridge 2022 HW configuration. 109

6.25 Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2022 code as slave/secondary in eBridge 2022 HW configuration. 110

Nomenclature

Abbreviations

ACPI	Advance Configuration and Power Management Interface
ALU	Arithmetic Logic Unit
ARM	Asynchrounous RISC Machine
CCD	Current Density Difference
CIVCO	in Spanish acronym Centro de Investigación en Vivienda y Construcción or Housing and Construction Investigation Center in English) of
CMOS	Complementary Metal Oxide Semiconductor
CONAVI	in Spanish Consejo National de Vialidad or National Road Council in English
CPU	Central Processing Unit
DCR	Dynamic Cache Reconfiguration
DDR	Double Data Rate Synchronous
DED	Dynamic Energy Dissipation
DESSASTRTES	Dynamic energy-saving scheduling algorithm for a sporadic task in real-time embedded systems
DFS	Dynamic Frequency Scaling
DM	Data Placement in Memory
DPM	Dynamic Power Management
DVFS	Dynamic Voltage and Frequency Scaling
EAFBFS	Energy-Aware Frame-Based Scheduling
EDF	Earliest Deadline First
EMPIOT	nergy Measurement Platform Internet of Things
FLPA	Functional Level Power Analysis
GPRS	General Packet Radio Service
HEARS	Heterogeneous energy-aware real-time scheduler
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ILLED	Least Loss Energy Density
IoT	Internet of Things
LoRa	Long Range
LTE	Long Term Evolution
MILP	Mixed-Interger linear programming
MM	Memory Mode

MOPT	in Spanish acronym Ministerio de Obras Públicas y Transportes or Transportation and Public Works Ministry in English
OS	Operating System
PET	Power-Energy-Temperature
PM	Power Management
PXIe	PCI eXtensions for Instrumentation
QMI	Qualcomm MSM interface
RPI	Raspberry Pi Embedded Board
RTOS	Real-Time Operating System
SED	Static Energy Dissipation
SHM	Structural Health Monitoring
SoC	System on Chip
TTS	Threshold Timeout/Sleep Algorithm
WiFi	Wireless Fidelity
WSN	Wireless Sensors Network

Chapter 1

Introduction

A good quality road infrastructure is crucial for boosting economic growth and poverty reduction in developing countries like Costa Rica, and bridges are one of the most essential components of it. Studies have proven a direct link between improving access to facilities, goods, job opportunities, health services, schools, and essential social services in increasing a country's human capital [1], which translates into poverty alleviation and a general improvement in the population quality of life.

Costa Rica has a national problem regarding the maintenance and renovation of road infrastructure, and bridges have been particularly affected by this. The country did not detail all its bridges; due to that, the CONAVI (by its acronym in Spanish *Consejo National de Vialidad* or National Road Council in English) worked with Costa Rica TEC from the period 2014-2018 to create a nationwide survey of bridges [2].

The survey mentioned above covered a total of 1 670 bridges out of which 37% were in a *deficient* conditions, 60% regular, and just 3% in optimal. A vital aspect of the study is that the survey did not cover rural areas and that out of the 47 905 km of roads nationwide, only 18% correspond to urban regions [3].

Structural Health Monitoring (SHM) is an emergent field of civil engineering that has the potential to perform a continuous evaluation of the security and structural integrity of a variety of infrastructures. The main objective of SHM is to know the structure's condition that empowers different teams to perform condition-based monitoring to help prevent catastrophic failures and prolong its life [4].

An effort from a multidisciplinary team of engineers from different schools of the Costa Rica TEC led by CIVCO (by its Spanish acronym *Centro de Investigación en Vivienda y Construcción* or Housing and Construction Investigation Center in English) of the school of Construction Engineering of Costa Rica TEC started the *eBridge* project in 2011 as part of the eScience investigation program. The group's primary focus is to provide innovative and low-cost SHM applications in Costa Rica. Some of the governmental groups that can benefit from this work are the National Emergency Commission (CNE by its Spanish acronym *Comisión Nacional de Emergencias*), the Costa Rican central government, CONAVI, and the Transportation and Public Works Ministry (MOPT by its Spanish Acronym *Ministerio de Obras Públicas y Transportes*) [5]. There are several rural communities in Costa Rica where the only road connection is a bridge, and having this structure damaged could significantly affect the quality of life of its population [4].

The project is divided into four stages named eBridge 1.0 (2011), 2.0 (2013), 3.0 (2016) and extension (2019). These and its effort are summarized in figure 1.1. Several multiple publications have been done since the investigation group started 2011 [2], [5]–[13]. In 2015, the group started putting together a proposal for a prototype [7], which was matured over the years on publications like [11] in which the variable of interest was structural vibration, and [10] in which the level of water below the bridge was the main point of interest. The process culminated with the development of a collaborative *Wireless Sensor Network (WSN)* in [12].

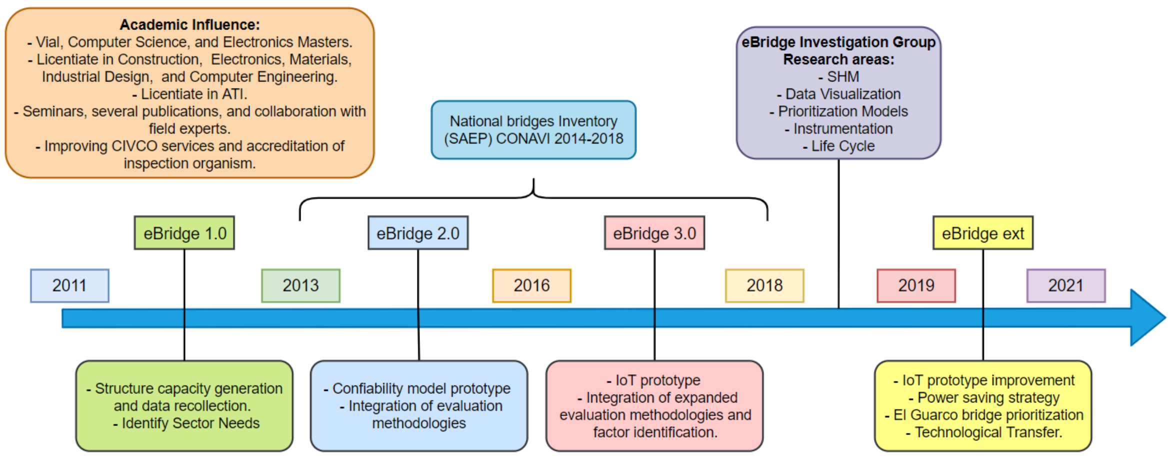

Academic Influence:
- Vial, Computer Science, and Electronics Masters.
- Licentiate in Construction, Electronics, Materials, Industrial Design, and Computer Engineering.
- Licentiate in ATI.
- Seminars, several publications, and collaboration with field experts.
- Improving CIVCO services and accreditation of inspection organism.
National bridges Inventory (SAEP) CONAVI 2014-2018
eBridge Investigation Group Research areas:
- SHM
- Data Visualization
- Prioritization Models
- Instrumentation
- Life Cycle
eBridge 1.0
eBridge 2.0
eBridge 3.0
eBridge ext
2011
2013
2016
2018
2019
2021
- Structure capacity generation and data recollection.
- Identify Sector Needs
- Confiability model prototype
- Integration of evaluation methodologies
- IoT prototype
- Integration of expanded evaluation methodologies and factor identification.
- IoT prototype improvement
- Power saving strategy
- El Guarco bridge prioritization
- Technological Transfer.

Figure 1.1: eBridge Project Timeline. Adapted from [3].

A collaborative network of sensors consists of a network in which the different sensors collaborate by sharing data and making joined decisions, in this case with the focus of helping prevent natural disasters by measuring the water level below the bridge. An example of the concept eBridge collaborative network can be seen in figure 1.2. By having multiple sensors in a single bridge, it is possible to ensure that other sensors' measurements are valid, and it is possible to increase the confidence behind water level reports. Also, it is possible to communicate multiple bridge networks across rivers, so if a disaster is detected on one bridge, the data can be transferred to other networks downriver. Aside from water level, the eBridge network is prepared to handle different measurements like structural vibration and deformation, temperature, etc. [12].

The eBridge network requires two types of wireless communication, one that allows sensors within the same network to talk and another that allows the network to communicate with the server. By the work done in [10], it was decided that the nodes would communicate with each other using the *XBee S2C*, device which uses the DigiMesh proprietary 2.4 GHz XBee 802.15.4 communication protocol, which can be used up to a distance of 1 200

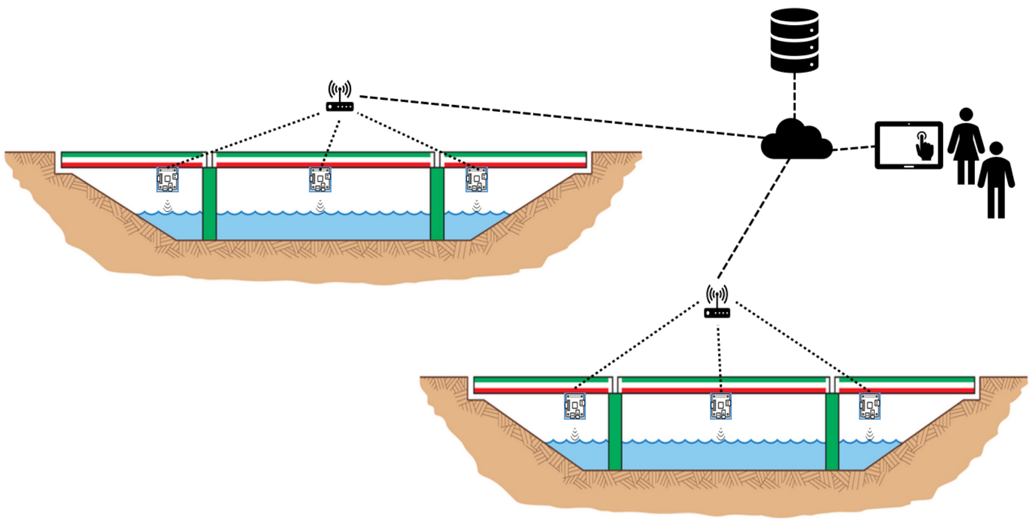


Figure 1.2: eBridge Collaborative Network concept. Adapted from [4].

m outdoors [14]. The cellular network uses General Packet Radio Service (GPRS) for device-server communication. The overall sensor node architecture can be seen in figure 1.3, and the general eBridge Network architecture in figure 1.4.

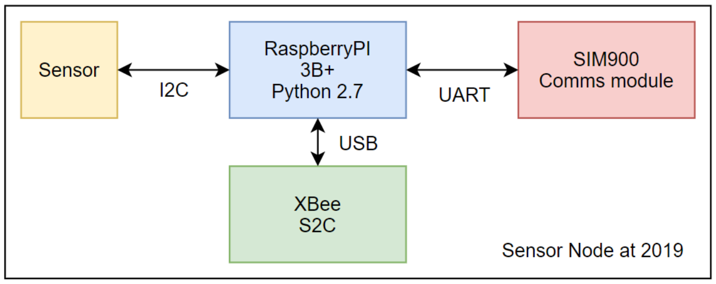


Figure 1.3: eBridge Sensor Node at 2019 [10], [12].

As mentioned in [12], the eBridge network, by the end of 2019, could only have 1 hour and 45 minutes of power autonomy when working with a battery of 4 400 mAh. Due to this is not practically possible to deploy the system on a large scale without human interaction. Due to this, the goal of this work would focus on creating a power-saving strategy for collaborative wireless sensor networks of multi-core embedded systems that translate into a higher power autonomy for the eBridge network.

1.1 Objectives and Document Structure

The project's general objective is to increase the energetic autonomy of the eBridge network from around two hours with a 4 400 mAh battery to twelve hours using a 5 200 mAh battery to expand its field deployment feasibility. The project's first stage was to

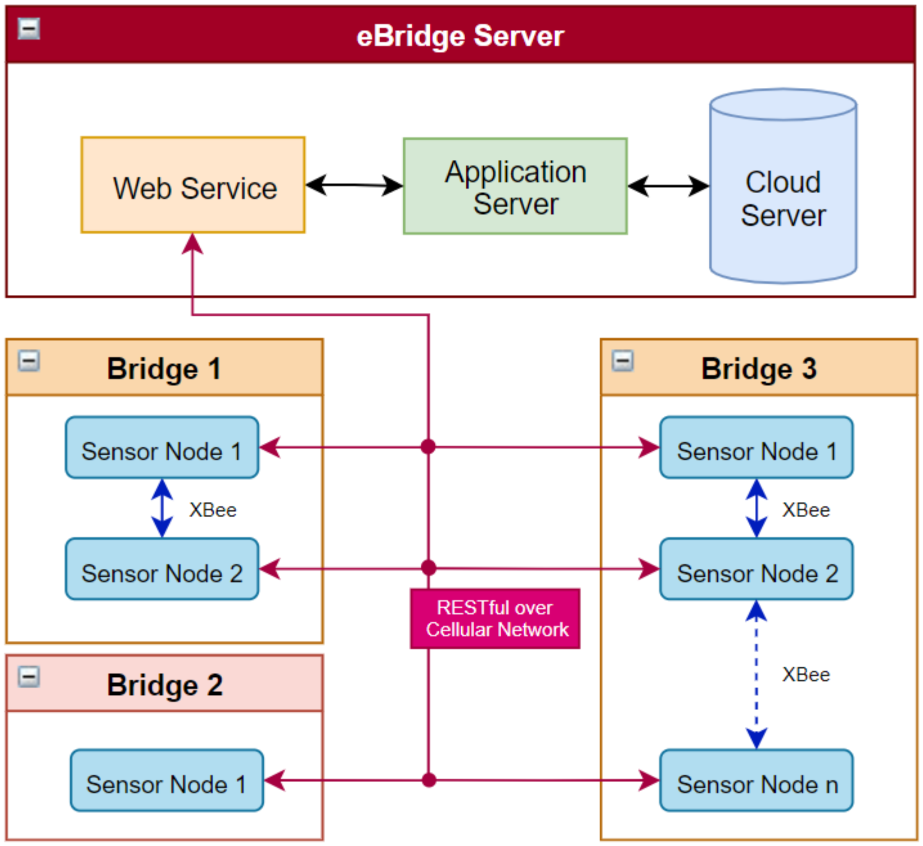


Figure 1.4: eBridge Network General Architecture. Adapted from [12].

make improvements over the existing network to improve existing hardware and increase the eBridge application performance. Once the application performance was improved up to its practical level, four specific objectives were the focus of this work.

First, accurately measure the power consumption of the eBridge network by using a task-level abstraction model on 5 200 mAh and 10 000 mAh batteries. Secondly, create a task execution model of the eBridge network for both primary and secondary systems with Raspberry 4B and Raspberry 3B + systems. Third, Measure task-level CPU load for primary and secondary systems with 5 200 mAh and 10 000 mAh batteries. Fourth, measure task-level current peak demand using a High-Resolution Voltage Acquisition board for both primary and secondary devices using 5 200 mAh and 10 000 mAh batteries. Finally, based on the results from the analysis points above, a strategy involving Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS).

This document is divided in the following way, the next chapter 2 includes the theoretical fundamentals of the development of this thesis work; in chapter 3, there is a detailed explanation of the proposed methodology to evaluate the eBridge network, in chapter 4 the results and analysis of the eBridge 2019 network, in 5 is possible to find the proposed plan for the eBridge 2021 and 2022 network, in 6 is the validation of the proposed design, and finally, in chapter 7 the conclusions of this thesis work are presented.

Chapter 2

Review of Related Literature

This chapter covers the study of related literature around the area of power consumption in embedded systems in section 2.1, how to perform power measurements in 2.2, and the different state-of-the-art power-saving strategies for embedded systems in 2.3, state of the art power-energy-temperature aware scheduling algorithms and strategies in 2.4, and finally other power-saving strategies in 2.6.

2.1 Power Consumption in Embedded Systems

With the increased use of multi-core battery-based embedded systems in IoT applications, there has been a need to ensure that these systems have enough power to complete their respective tasks. Estimating the energy consumption on an embedded system is not a trivial matter and is commonly an essential aspect of reducing it. There are three levels of abstraction from which power consumption can be analyzed. The first of these levels is the *Circuit Level*: The total energy or power consumption of a digital Complementary Metal Oxide Semiconductor (CMOS) circuit consists of two components, namely, static energy dissipation (SED) and dynamic energy dissipation (DED). SED occurs when the circuit is not operative, meaning that the CMOS is not switching, and there is a steady input that could be either high or low. A few years ago, this was considered insignificant, but as transistors become smaller and faster, it starts having more relevance [15].

The second abstraction level is the *Functional-Level*. This approach estimates energy consumption by analyzing the energy consumption of functional units on the processor [15]. A functional unit is a part of a processor that performs data processing operations or calculations, such as arithmetic logic units, multipliers, or those in charge of accessing memory or registers. An example is a unit in charge of accessing the cache memory, which is the processor's fastest and smallest on-chip memory. The cache is regarded as one of the greatest energy consumers on a processor. It is estimated that 43% of total energy consumption is used within these operations [15]. The parameters for the energy model of the functional levels are the access rate of the on-chip memory, the clock frequency, and the degree of parallelism. In [16], the energy consumption model of a multi-core embedded system with a hierarchy of shared memories is explored, being the L1 memory independent of each processor and the L2 shared between the cores.

A processor has different operating modes or states, in which some of its functional units can be on or off. These could be classified as **sleep mode**, **idle mode**, and **active mode**. Sleep mode refers to the state in which the system intentionally turns off a processor to save energy. Idle mode is when a processor is not actively working and thus is in a steady state, causing SED. Finally, active mode is when the processor executes a task. Keeping the processor idle for a long time could significantly impact a system's power budget. However, it is also described how turning off the processor by sending it to sleep mode is not always a good idea due to the amount of energy needed to start the system again [17].

The third level of abstraction is the: *Instruction-Level*. In the work done by [15], it is concluded that the number of instructions is proportional to the energy consumption in the processors. There is a direct relationship between the cycle counts and the energy consumption in processors, these counts are very hard to read due to pipeline stalls, which can be caused by faulty branch prediction, cache miss-hit, and pipeline hazards will lead to spending more clock cycles to complete a task in a processor.

Multiple authors have worked on creating task scheduling algorithms based on two approaches: Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM). DVFS is described as slowing down the central processing unit (CPU) frequency to reduce the operating voltage and, consequently, the power consumption. DPM refers to how processors can be put in sleep mode to save power when the load is low [18]. As is seen in the work done by [16], [18]–[21], a combination of both approaches are commonly used.

The main focus of this work is to provide a review of existing Power-Energy-Temperature (PET) task scheduling algorithms, as described in the review done by [22], using a combination of both DVFS and DPM on multi-core embedded systems for reducing energy consumption.

2.1.1 Power Consumption Model

As mentioned above, estimating the power consumption of an embedded system can be approached from multiple levels of abstraction. In this section, each one of the abstraction models will be expanded. It is essential to consider that it is possible to find studies performed on homogeneous multi-core processors, in which all the cores are of the same type, and heterogeneous multi-core processors, in which they are not.

Power vs Energy

This paper's key factor is understanding the difference between power and energy. Power is the rate at which the energy, measured in joules (J), is used (time (s)). This is described in equation 2.1.

$$P = \frac{E}{t_i} \tag{2.1}$$

From equation 2.1, equation 2.2 can be deducted.

$$E = P\dot{t}_i \tag{2.2}$$

Circuit Model

Multiple authors like [15], [17], [18], [23], [24] refer to the circuit level model as the combination of both the SED and DED models. This combination results in equation 2.3.

$$E_{total} = SED + DED \tag{2.3}$$

Being SED Static Energy Dissipation and DED Dynamic Energy Dissipation. The SED can be expressed in terms of the voltage supply of the processor V_{dd}, and the total leakage current $I_{leakage}$ [15], as is shown in equation 2.4. In the work done by [23], it was concluded that the SED represents around 5% of the total DED.

$$SED_{total} = I_{leakage} \cdot V_{dd} \tag{2.4}$$

The DED portion depends on two factors, first the $DED_{switching}$, which is the dissipated energy by the switching of the CMOS circuit, and secondly $DED_{short-circuit}$. This is described in equation 2.5.

$$DED_{total} = DED_{switching} + DED_{short-circuit} \tag{2.5}$$

The $DED_{short-circuit}$ depends of the short-circuit current, the clock frequency of the processor$f$, and the voltage supply of the processor V_{dd} (equation 2.6). On the other hand the $DED_{switching}$, depends of f, the effective switching capacitance C_{ef}, and V_{dd} (equation 2.7).

$$DED_{short-circuit} = I_{short-circuit} \cdot f \cdot V_{dd} \tag{2.6}$$

$$DED_{switching} = C_{ef} \cdot f \cdot {V_{dd}}^2 \tag{2.7}$$

By combining equations 2.4, 2.5, 2.6, and 2.7, with an activity factor k, which has a value of 0 if the processor is not active or a value of 1 otherwise, it is possible to come with the power model shown in equation 2.8.

$$P_{total} = k \cdot [I_{short-circuit} \cdot f \cdot V_{dd} + C_{ef} \cdot f \cdot {V_{dd}}^2] + I_{leakage} \cdot V_{dd} \tag{2.8}$$

Functional Level

The functional level of abstraction described by [15] is a model used since 2000 for authors like [25]. This model divides the processor into *functional units*; an example of this can be seen in figure 2.1. Each one of these functional units could be either a cluster of arithmetic logic units (ALUs), multipliers, or register banks that could be activated when certain operations occur. Laurent [25] refers to this model as Functional Level Power Analysis (FLPA), which depends on two types of parameters. First, the algorithmic parameter values depend on the executed algorithm itself, and the second ones are called architectural parameters, which depend on the processor configuration settled by its designer. Some of these parameters can be found in table 2.1.

Table 2.1: Algorithmic and Architectural Software Parameters [25]

Software Parameters	
Algorithmic	
Name	Description
α	Parallelism rate
β	Processing unit rate
γ	Cache miss rate
τ	External memory access rate
ϵ	DMA access rate
Architectural	
Name	Description
W	Data width transferred by the DMA
F	Clock frequency in MHz
MM	Memory mode
DM	Data placement in memory
PM	Power management (units in sleep mode)

Any time a job is performed by the processor, the use of this model help to understand which of these *functional units* activate, as shown in figure 2.2. Knowing which of these *functional units* are active can be used to estimate power consumption.

As described by both authors [15], [25], the main drawback of this method is that it is not possible to make power consumption estimations of processors whose architecture is unknown. Also, creating the power model itself can take considerable effort, and if the architecture of the processor or the processor itself is changed in a project, this process would have to be repeated.

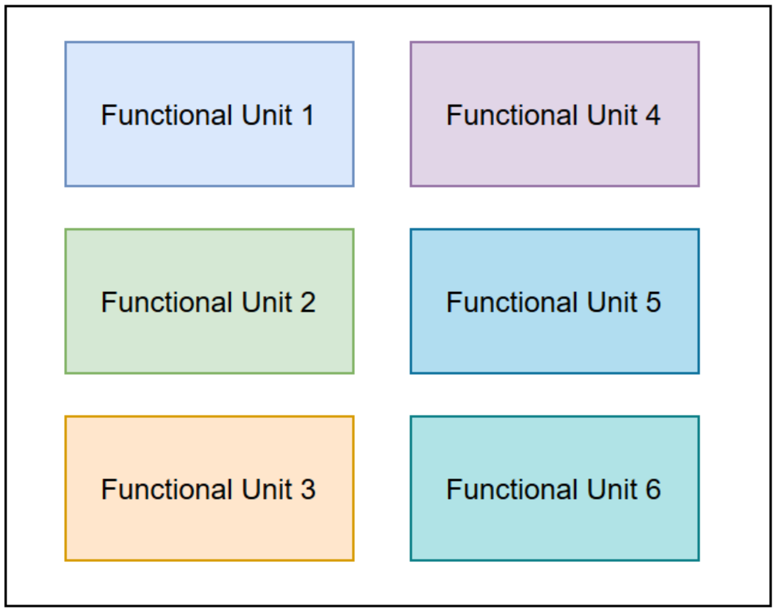


Figure 2.1: Dividing a processor in Functional Units[25].

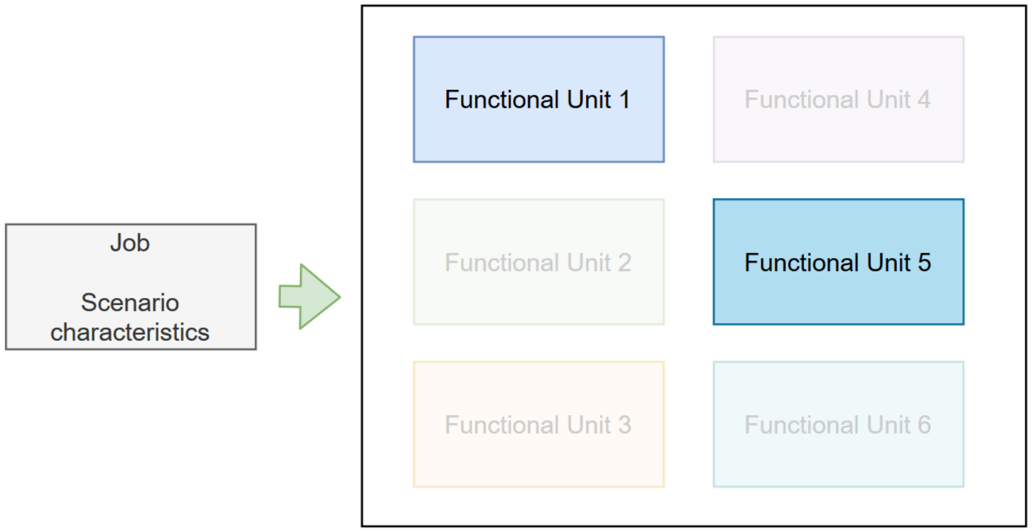


Figure 2.2: Dividing a processor into Functional Units[25].

The investigation done by [26] about a custom SED model was later expanded by [27]. This model uses processor-specific characteristics like the ones described by both [15], [25]. In the work done in this paper, this model will be explored. Both Laurent and Enging identified 15 different variables which can be used to estimate the power consumption for the 70 nm technology they used during their investigation. Also, they explored the energy dissipation caused by accessing cache memory Ec_{cache}, that can be divided into dynamic Ec^{dyn}_{cache}, and static Ec^{sta}_{cache} power consumption. This approach is described by equation 2.9.

$$Ec_{cache} = Ec^{dyn}_{cache} + Ec^{sta}_{cache} \tag{2.9}$$

The dynamic energy dissipation by accessing cache memory depends on three variables [26]: (1) the number of cache access operations N_{ca}, (2) the number of cache misses N_{cm}, and (3) the number of clock cycles involved in a given operation. The authors [26], [27], used a static energy dissipation value for each one of the cache memory success access operations. This energy dissipation is denoted by E_{access}, and the energy dissipated by every memory miss operation is represented by E_{miss}. This is summarized in equation 2.10.

$$Ec^{sta}_{cache} = N_{ca}\dot{E}_{access} + N_{cm}\dot{E}_{miss} \tag{2.10}$$

Instruction Level

The instruction level model works under the assumption that a real-time application has a set of independent frame-based tasks represented as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, where Γ represents a real-time application, and τ_i represents a task within that application [28]. There are three potential approaches for estimating the power consumption of a given task. First, a linear estimation like the one used by [28] assumes that each task can be defined by a set of homogeneous clock cycles; these cycles are represented by c_i. The linear estimation method also considers the frequency f_i at which the task is executed. The relationship between c_i and f_i is expressed in equation 2.11.

$$e_{time} = \frac{c_i}{f_i} \tag{2.11}$$

Secondly, the model proposed by [29] takes into consideration each task's worst execution time WC_i at a given frequency f_i to calculate that task DED, as is shown in equation 2.12.

$$DED_{task}(\tau_i) = C_{ef} \cdot f_i \cdot {V_{dd}}^2 \cdot WC_i \tag{2.12}$$

The third approach can be found in the work done by [24]; this model considers heterogeneous multi-core processors. In this model each task τ_i can be characterized by four different parameters $\left\langle D_i, T_i, \vec{C_i}, \vec{E_i} \right\rangle$, where D_i is the relative deadline of the task, T_i is the minimum time between to consecutive executions of τ_i, $\vec{C_i^m} = \left(\vec{C_i^1}, \vec{C_i^2}, \ldots, \vec{C_i^M}\right)$ is the vector of execution profiles of τ_i on a M set of different cores. Finally, $\vec{E_i^m} = \left(\vec{E_i^1}, \vec{E_i^2}, \ldots, \vec{E_i^M}\right)$ is the vector of energy profiles associated to each core in $\vec{C_i}$. The advantage of this model is that it can be combined with the model used by [29], which takes into consideration the worst execution to time WC_i at a frequency f_i for each one of the cores. It is possible to build a vector $\vec{C_i^m}$ with all the WC_i if task τ_i, resulting in $\vec{C_{i,j}^m}$ (with $j = 1{,}2, \ldots, WC_i$). Similarly, it is possible to define a vector with the energy dissipation $\vec{E_{i,j}^m}$ (with $j = 1{,}2, \ldots, WC_i$), related to each element in $\vec{C_{i,j}^m}$.

2.2 Power Measurement in Embedded Systems

In this section, several power measurement strategies for Embedded Systems will be covered. Recent works like [30] have focused on exploring the complexity and breadth of power measurement techniques for IoT applications based on embedded systems. One of the common trends in this area is creating a customize technique based on the type of embedded system and IoT application. As mentioned in section 1, the eBridge network uses the Raspberry PI 3B+ platform as its controller.

Based on the results of the IoT developer survey from the Eclipse Foundation [31], 43 % of IoT applications use Linux as their OS, 35 % use FreeRTOS, and 31 % use Windows. Also, 51 % of systems use *HTTP/HTTPS* as their main communication protocol. However, only 37 % of devices use cellular ((LTE, 4G, 5G, etc.) as their main source of connectivity. This means that the eBridge network follows common development trends for IoT devices.

The power measurement system for IoT applications tends to have very demanding requirements like high acquisition rates of up to 40 kHz, high dynamic range to be able to take measurements as low as 1 μA o up to several hundred mA, high resolution of up to 16 bits or better, multiple measurement channels, various trigger modes, local data storage, and most importantly a low-cost [23]. Most embedded systems lack a native solution to measure their power consumption, and not all systems have a software-based approach to model it accurately. Also, IoT networks tend to escalate to thousands of devices, so solutions like low-cost watt-meter tend to be impractical due to the significant cost of deploying it to the entire fleet [32].

A challenge in accurately measuring power for this type of application is that a whole IoT typically consists of the main controller and several peripheral devices. Most works focus on accurate measurements of the controller itself like [32], which focuses on a preliminary study of the impact of software on Raspberry Pi devices by creating a very accurate power consumption software estimation model with errors as low as 1.25%. State-of-the-art power measurement strategies like Energy Measurement Platform Internet of Things (EMPIOT) [30], and more recently [33]. However, these approaches require custom circuitry.

2.3 Power Saving Strategies in Embedded Systems

It is possible to come up with a graphical representation of the power consumption of a given task τ_i like the one shown in figure 2.3 for a homogeneous multi-core architecture. Also in figure 2.4 is possible to observe the execution time for consecutive executions of a task. It is important to consider that a task's execution time can change.

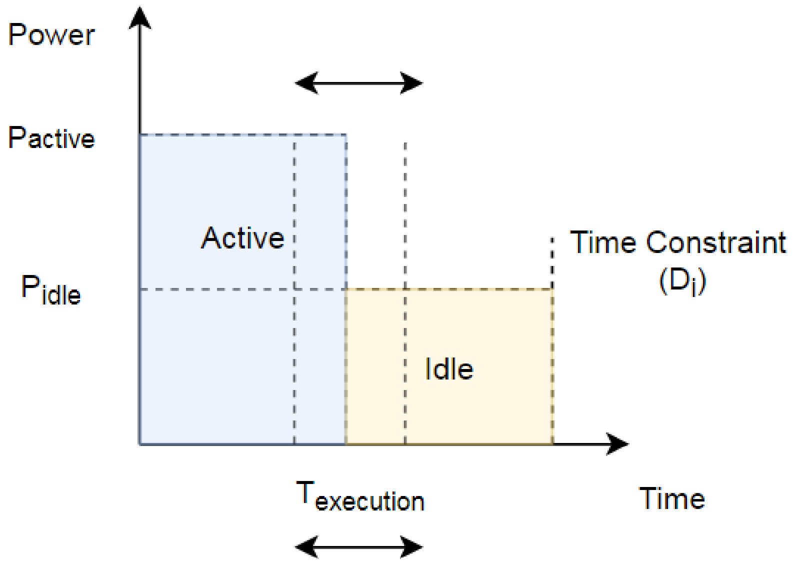


Figure 2.3: Power vs execution time of a task τ_i [25].

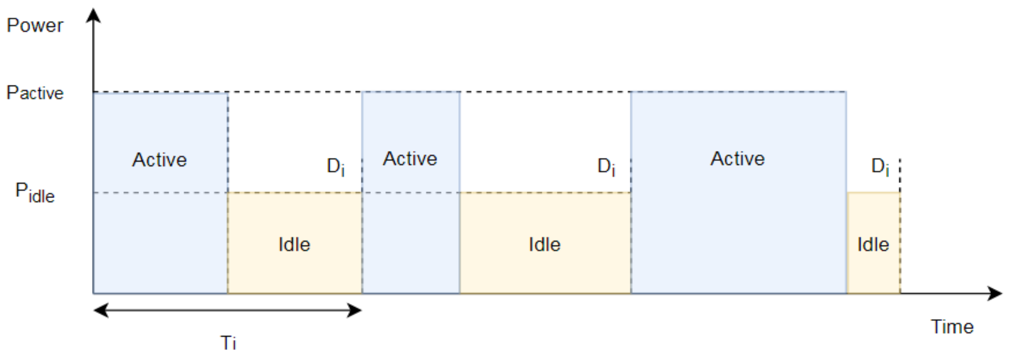


Figure 2.4: Power vs execution time for multiple consecutive executions of task τ_i [25].

Dynamic Power Management (DPM)

As described in [18], it is possible for the Operating System (OS) to turn off some of the functional units of a processor when the load is low; having multiple sleep modes in a processor can be beneficial. A user can control these characteristics by using an Advance Configuration and Power Management Interface (ACPI). In the work done by [17], [34], it is described how there is a considerable amount of energy dissipation associated with waking up a processor from a sleep state; this energy dissipation can be referred to as E_0. In the investigation done by [17], [34], it is proposed that it is worth sending a processor to sleep only if the energy dissipation of keeping the processor in idle mode is higher than the energy dissipation caused by changing its operating state from the sleep mode to active mode. This is generally referred to as the *break-even time* in equation 2.13.

$$t_0 = \frac{E_0}{P_{idle}} \tag{2.13}$$

Where P_{idle} is the power consumption of a processor in idle mode.

In the work done by [34], it was determined that embedded systems have the characteristic of dissipating different amounts of energy when they *wake up* from different low-power dissipation states. The power consumed by the field-deployed embedded systems, which is hard to get a continuous energy supply, was evaluated. An algorithm focused on searching for the correct sleep policy for a processor based on its function was developed to overcome this challenge. This algorithm will be explored in more detail in section 2.4.

DPM can be classified into three subcategories [35]: (1) Threshold Timeout/Sleep Algorithm (TTS), (2) Predictive DPM, and (3) Stochastic Method.

Threshold Timeout/Sleep Algorithm (TTS)

This DPM strategy was investigated by the work done by [34]. The main goal is to find the ideal wait time that the system should wait before sending the processor to sleep mode. This threshold time needs to be calculated considering that the system needs to use less energy to sleep and then return to an active state than staying in idle mode until the next task arrives.

Predictive DPM

The main goal of the predictive DPM strategy is to estimate the combined execution time of a series of tasks $(\tau_1, \tau_1, \ldots, \tau_n)$, assuming that the execution time of each task could be random. Then based on this calculation, the system needs to calculate the optimal next processor idle time T_{idle}.

The main challenge of this method is that it requires a lot of historical data from the possible execution times for each one of the tasks [35]; also, it requires the system to run calculations on the fly to determine the right moment to send the processor into sleep mode causing the processor's load to increase.

Stochastic Method

The defined stochastic methodology as an analysis procedure that depends on the *Markov Decision Process* is defined in [35]. This method is not used often due to the complexity of the analysis needed for its implementation.

2.3.1 Dynamic Voltage and Frequency Scaling (DVFS)

The Dynamic Voltage and Frequency Scaling (DVFS) is explored by multiple authors [16], [18]–[22], [24], [27], [36]–[38]. This technique is used to control an embedded system process performance, temperature, and energy consumption. Most mobile devices are equipped with processors that include firmware that allows the use of DVFS. This technique allows us to slow down a central processing unit's frequency and operating voltage (CPU). This results in a power consumption reduction[18]. As it is mentioned by both [16], [38], DVFS was introduced as a flexible computing technology that allows power saving on systems that are operating in energy-limited environments and with non-static workloads.

Systems with non-static workloads have the possibility of executing sporadic tasks, which, as indicated by their name, can execute at any time. Predicting these kinds of processing workloads is difficult. In figure 2.5, it is possible to observe how the power changes depending on different types of tasks; the red line represents the amount of power supplied to the processor. The process of changing a processor supply voltage is known as Dynamic Voltage Scaling (DVS).

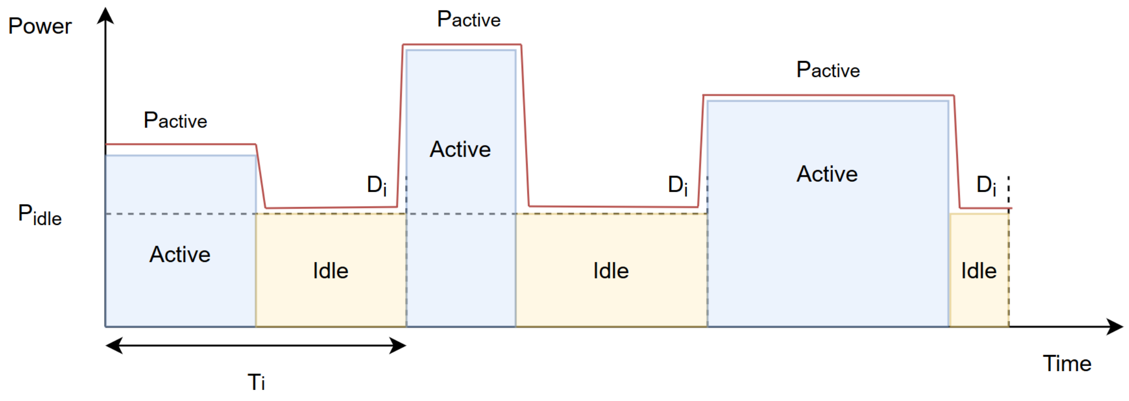


Figure 2.5: Dynamic Voltage Scaling (DVS).

Equation 2.7 provides a mathematical representation of a processor's total dynamic energy dissipation, which depends on the frequency. Since it is possible to use Dynamic Frequency Scaling (DFS) to reduce energy dissipation for a task by changing the frequency at which is executed. This is shown in figure 2.6 (a) and (b). F_{active} represents the frequency at which a task τ_i is executed, D_i its time constrain, $T_{execution}$ its execution time, and T_{missed} the time by which the D_i was missed.

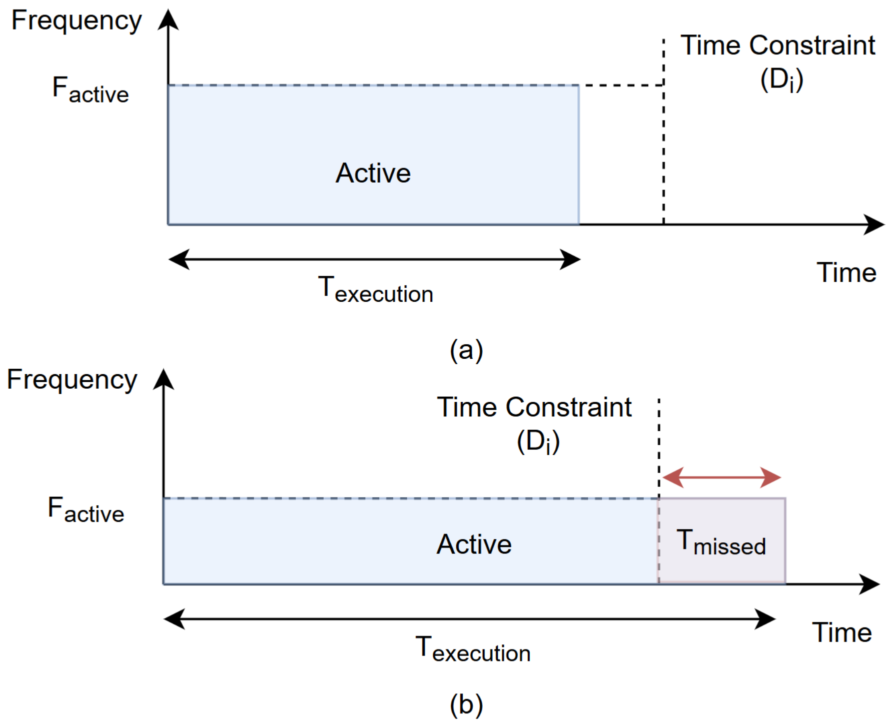


Figure 2.6: Dynamic Frequency Scaling (DFS). (a) Task executing with higher CPU Frequency. (b) Task executing with Lower CPU Frequency.

DVFS is the combination of applying the DVS and the DFS techniques in a processor. There are two main types of DVFS based on the work done by [18]: online and offline. The offline technique requires specific knowledge of the application and its tasks; the abstraction level commonly used to estimate power consumption is the *instruction level model* described previously. The first step for implementing the DFVS offline technique is to find what is known as the *best execution frequency* for each task, which is selected based on two main criteria: the first is that the selected frequency allows the task completion within its time constraints, and secondly that there is a considerable power consumption reduction.

It is essential to analyze the trade-off of lowering the frequency of a processor since this commonly increases the execution time of each task providing a chance of missing its deadline; on the other hand, it is possible to execute a job faster and use DPM to send the CPU to sleep mode. The DVFS online approach can be classified into two sub-types: reactive and proactive. In the proactive methodology, the Operating System (OS) tries to predict a processor's workload, and then DVFS is applied based on that prediction. On the contrary, in the reactive methodology, the workload is measured, and then the DVFS policy that reduces power consumption to the minimum is applied. The reactive approach has proven more efficient; however, defining the right policy requires a proper study of each system's workload types and operating modes. Somehow some other authors have used machine learning, specifically the Reinforced Learning (RL) technique, to overcome this challenge [18].

The use of DVFS does not always lead to power saving [22]. However, in some cases, when it is possible to define the *right policy* to apply DVFS, some authors have been able to reduce energy consumption by over 40%. The work done by [16] talked about how it is more complex to apply DVFS techniques in multi-core systems; one of the reasons is that each core can handle different workloads. Since the *right DVFS policy* depends on the workload, the user commonly needs to define one for each core. A major challenge in defining these policies for multi-core systems is that there could be scenarios where a task can move between cores. This adds another level of complexity because the task could be moving between what is known as *frequency domains*, whose execution time could change. We would explore this more in section 2.4

Also, it is possible to classify DVFS policies into two more subcategories [27], [39]: (1) intratask and (2) intertask.

Intratask DVFS

Intratask DVFS technique is defined as the methodology in which the processor's frequencies can be changed while executing a task. This approach allows the system to change its processor frequency during different stages of the execution of task τ_i [27]. If a section of code heavily utilizes memory, the processor may stall while waiting for the memory requests to complete. The processor frequency can be lower in this bottleneck situation, reducing the DED [39].

This methodology adds a new dimension of the complexity to the DFVS; the main focus of the user should be to answer these two questions: (1) when should the frequency be adjusted?, and (2) what is the *right* frequency value? The first question is the trickiest one to respond to and commonly is necessary to find at which *program phase* is best to make the frequency adjustment. A *program phase* is defined as a window of execution time within a program in which the program's characteristics are homogeneous [39]. It was concluded that during a specific *program phase*, there is only one optimal clock frequency. In a Real-Time program, there could be a large number of these phases, and one of the significant challenges of implementing the intratask DVFS technique is that each one of these phases has to be experimentally defined.

Intratask DVFS

This methodology allows us to change the processor frequency at the *intertask boundaries*. Meaning that each task τ_i is taken as a singular execution time window.

2.4 PET-aware Scheduling Algorithms/Strategies

In this section, several Power-Energy-Temperature aware scheduling algorithms will be described in more detail to be later compared in section 2.5.

2.4.1 Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems

The DVS and Dynamic Cache Reconfiguration (DCR) techniques are combined in a system aware of its leakage power (SED). DCR is a technique that allows the configuration of the cache memory parameters at run-time so that it is possible to save energy from cache memory accessing operations [26]. As described above, these operations can represent up to 43 % of the total DED of an embedded system [16].

This strategy is focused on three main actions: (1) the estimation of the processor's critical speed, (2) the real-time scaling and reconfiguration selection, and (3) what is known as *task procrastination.* Each one of these will be addressed in the subsections below.

Estimation of Critical Speed

A *critical speed* is the point where the speed of a processor can no longer be slowed down. Otherwise, the use of DVS would not result in a reduction in energy consumption. Also, it was found that the use of the DCR technique has a significant impact on finding the critical speed of a system and consequently reducing that system's energy consumption [26].

Real-Time Scaling and Reconfiguration Selection

A *configuration point* (v_j,c_k) is defined as the combination of the use of the DVS (v_j), and DCR (c_k) techniques. Also, it is indicated that is possible to build a *profile table*, which contains all the possible configuration points (v_j,c_k) for a processor. It is important to take into consideration that all the (v_j) points that are below the *critical speed* value are eliminated. Also, any configuration point which results in a smaller energy consumption reduction than other points is not included in the analysis [26].

Task procrastination

A modified version of the *Earliest Deadline First* (EDF) scheduling algorithm focuses on reducing a processor's busy/idle cycles as much as possible. As was described above, there is an overhead in waking up a processor from a sleep state; thus, having more significant idle times T_{idle} for a processor is usually desired [26].

2.4.2 Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination

The main goal of this technique is to model *idle time intervals* between multiple processors in a multi-core architecture. The purpose of this technique is to optimize the combined use of the DVFS and the DPM techniques simultaneously. The authors proposed a *mixed-integer linear programming* (MILP) approach for integrating both methods. This technique's main contribution to state-of-the-art was the definition of a way to model processors' idle intervals t_{idle}, which was not possible before this investigation. This innovation allows to integrate of the *idle times* within the scheduling logic of a processor and helped them to achieve a 9.1 % energy-saving in comparison to applying the DVFS, and the DPM techniques separately [27].

One of the main challenges faced during the definition of this methodology was to be able to determine the idle intervals t_{idle} of a given task. This idle time is defined by the closest task to the one under analysis. The example in figure 2.7, the idle interval I_1 of a task τ_1 is determined by its closets task, which in this case is τ_3. I_2 is determined by τ_3, and I_3 and I_4 are determined by the end of the Hy-period, which is the name the author gave to the execution window under study [27].

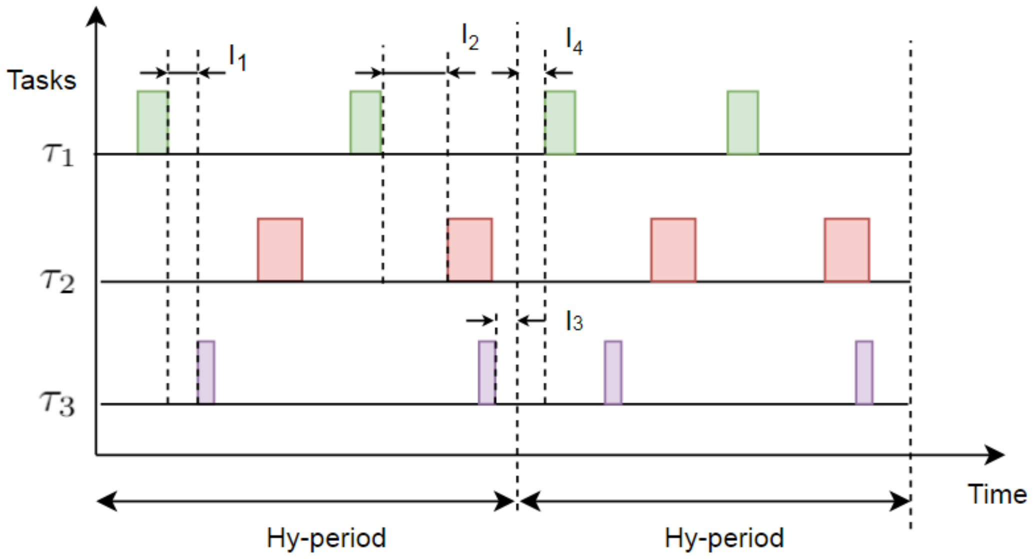


Figure 2.7: Idle times I_i of multiple tasks (τ_1,τ_2,τ_3). Adapted from [27].

2.4.3 Energy-Aware Frame-Based Scheduling (EAFBFS)

A homogeneous multi-core scheduling strategy for hard real-time systems called *Energy-Aware Frame-Based Scheduling* (EAFBFS) was proposed in [20]. This technique incorporates DVFS within the processor scheduling logic, intending to make the scheduling logic *energy-aware*. The authors used a semi-partitioning strategy with a two-level hierarchical scheduling scheme. The extreme level of this scheme divides time into frames or execution windows; the boundaries of these windows are defined by the arrival and departure of tasks. Based on the results found by the authors, this methodology helped restrict task migrations to other cores and what is known as *preemption overheads*. These *preemption overheads* are caused when a given task τ_1 is executing, and it is stopped by another task τ_2, so this one can execute. On the other hand, the *inner level* of the scheduling scheme is in charge of supervising tasks executed within the time frames described above.

The EAFBFS framework tries to allocate processor resources to a set of tasks so that independent cores can be operated at their specific *critical speeds* while the system ensures the fairness in the execution rates of all tasks consistently above a defined tolerance. The frequency allocation and mapping module or FAM is in charge of executing each frame's end. The FAM is part of the outer level scheduling scheme, while on the inner level, there is an ERFair (Early Release Fair) scheduler for each independent core. It is essential to mention that the ERFair schedule is considered the first known optimal real-time scheduler [20].

2.4.4 Improved least loss energy density algorithm (ILLED)

In energy-saving results, the ILLED algorithm outperforms other algorithms, like the MaxMin (MM) or the Least Loss Energy Density (LLED) algorithms. The ILLED task-to-core scheduling allocation algorithm was defined for heterogeneous multi-core platforms. It is composed of two phases: (1) the scheduler allocates the tasks to the different cores, each one of these tasks has an ideal execution frequency known by the scheduler, and it is possible to group tasks with similar execution frequencies. (2) The second scheduling phase refines the allocation performed by the first phase to achieve *better* sleep states that reduce DED. This could be achieved either by temporally increasing the frequency of a core to accumulate extra idle time or by collating tasks so that better sleep times could be achieved [24].

This methodology aims to identify the behavior of each task running at different frequencies in different cores; this information is later used to improve task allocation and scheduling. Doing this process helps to find what is known as a *favorite core*, which is the core at which a task can be assigned that will result in minimizing energy consumption, and that could help group tasks execute at similar frequencies. This can help to reduce the unnecessary overhead caused by switching core frequencies. The *energy density* of each task is computed on different cores, then the *density difference* of each task in their preferred core is computed. Finally, these *density differences* are stored in what is known as a *current density difference set* or CCD. The CCD is used at task allocation time, and it helps to define where to assign each task, trying to keep the lowest *density difference* possible, which results in the maximum energy-saving scenario.

In summary, this algorithm takes into consideration three variables: (1) independent core frequency set-points, (2) task energy consumption, and (3) special core sleep state that result in reducing DED.

2.4.5 Heterogeneous energy-aware real-time scheduler (HEARS)

HEARS's heuristic strategy addresses the lack of an energy-aware scheduling algorithm that allows task migrations between the cores of a heterogeneous multi-core system. The goal behind migrating these tasks is to lower resource utilization by distributing the load and consequently reducing the frequency at which a processor needs to operate to comply with its tasks' deadlines [21]. This scheme is different than the one presented in ILLED [24], in which a task cannot migrate between cores. While the HEARS algorithm is performing these task-to-core allocations, not only the energy demand is considered, but also the current operating frequencies of each one of the cores are taken into consideration. This helps avoid unnecessarily changing frequencies and prevents affecting the system's overall energy consumption.

The HEARS algorithm proposes a two-level hierarchy scheduling scheme. On the first level, the algorithm applies what is known as *deadline partitioning*, which is used to compute an ideal execution frame. This execution frame is the same concept described above, the same that was presented by [24] in his work. The second level of the scheduling scheme is applied within each execution frame, and tasks are scheduled to available cores, such that each task receives its appropriate execution share and the operating frequencies of the cores are scaled appropriately [21]. Something important is a task cannot be fully allocated to a single core, meaning that the concept of *favorite core* does not exist.

2.4.6 Dynamic energy-saving scheduling algorithm for a sporadic task in real-time embedded systems (DESSAST-RTES)

An efficient DVS algorithm that largely depends on acquiring more slack time from both high and low-priority tasks was proposed in [17]. Then the algorithm takes the benefit of the slack time to scale the processor's supply voltage down to reduce both DED and SED. This scheme considers the difference between actual execution time and WCET, so using a DFS methodology at the intertask level guarantees that all tasks meet their respective deadlines. As was described above, the concept of *critical speed*, which is used to reduce the SED when the processor is idle for a long time, was introduced by [17].

DESSAST-RTES assumes that there are two main queues for tasks in an embedded real-time system: (1) the wait-queue W_q, which contains tasks that have been already completed, and (2) the Ready-queue R_q, which contains the ones ready to execute. To reclaim as much slack time as possible, it is assumed that all tasks are initialized on W_q, then if a task is activated is moved to R_q. In case the running task is preempted by a higher-priority task is sent back to R_q.

It is possible to gain slack time from three sources: (1) when higher-priority tasks are completed earlier than their respective WCET, this remaining time can be used by other tasks τ_x, (2) when τ_x is activated, and it has remaining time, and (3) when running lower-priority tasks instances are preempted by higher-priority ones, τ_x can use the slack time from the lower priority task τ_L.

2.5 Comparison of PET-ware Scheduling Algorithms

As seen in section 2.4, the PET-ware scheduling algorithms have approached the energy-saving paradigm from multiple angles, thus making them difficult to compare. The variables on table 2.2 are define to help this process. Even though multiple authors mention their respective energy-saving achievements, it is hard to compare them since all have used different algorithms to benchmark their respective solutions. Recently, a model for benchmarking these algorithms on general-purpose multi-core processors is commonly different from those used in embedded systems, as defined by [22].

Table 2.2: Comparison Variables for PET-ware Scheduling Algorithms

PET-aware scheduling algorithms comparing variables

Variable name	Abbreviation
Improved Memory Access Techniques	IMAT
Dynamic Voltage and Frequency Scaling	DVFS
DVFS (Online or Offline)	OO
Type of DVFS (Intertask or Intratask)	DVFS - Type
Dynamic Power Management	DPM
Type of DPM (TTS, Predictive, Stochastic)	DPM -Type
Core Migration	CM
Core Architecture (Homogeneous or Heterogeneous)	CA
Favorite Core Scheme	FCS

The algorithm comparison is made in table 2.3.

Algorithm	IMAT	DVFS	OO	DVFS - Type	DPM	DPM - Type	CM	CA	FCS
[26]	Yes	Yes	Offline	Intertask	Yes	TTS	N.M	N.M	No
[27]	No	Yes	Online	Intertask	Yes	Predictive	N.M	N.M	No
[20]	No	Yes	Online	Intertask	No	Predictive	Restricted	Homogenous	No
[24]	No	Yes	Online	Intertask	Yes	TTS	Not allowed	Heterogenous	X
[21]	No	Yes	Online	Intertask	No	N.A	Allowed	Heterogenous	Not Allowed
[17]	No	Yes	Online	Intratask	Yes	Predictive	N.M	N.M	No

Table 2.3: Comparison of PET-ware Scheduling Algorithms

As can be seen, not many authors have worked with Intratask DVFS schemes in combination with a predictive DPM strategy.

2.6 Other Power-Saving Strategies

2.6.1 Operating System Power Saving Strategies

There are two types of OS-level optimizations, which are application-specific and general optimizations. The first is a group meant to modify the OS based on an application resource utilization, which translates into power consumption. The second group refers to OS-level modifications; however, it commonly requires a deep knowledge of the OS boot-up process, which is not common among developers [30]. These types of optimizations are hard to keep up to date due to constant OS and application updates, due to this work would focus on Software Level Optimizations, which are covered in section 2.6.2.

2.6.2 Software Level Optimizations

Many studies have shown that different instruction sets, source code structures, algorithms, and software architecture directly impact energy consumption due to their direct impact on the hardware. All the previous aspects could represent up to 80% of the power consumption [40]. This complies which what is described in section 2.1.1 and can be seen in figure 2.8. For more details on the hardware layer, refer to section 2.1.1.

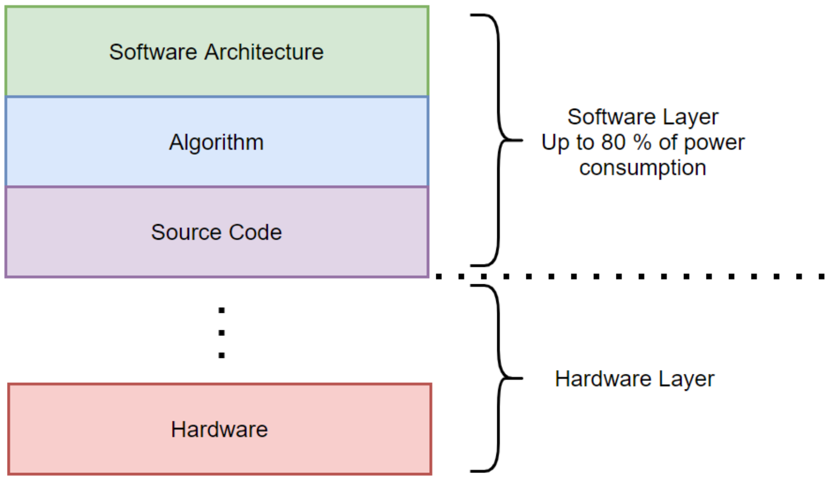


Figure 2.8: Software and Hardware Layer in Power Consumption. Adapted from [40].

This optimization requires deep knowledge of the application and good resource profiling techniques. To provide meaningful power-saving software optimizations, measuring the application's CPU, memory, storage, and network utilization is essential. The methodology for evaluating the eBridge network is described in chapter 3. This work proposes a multi-embedded system instruction-level CPU load and power analysis that considers the multiple nodes part of a WSN for SHM, which requires the creation of a custom software profiler.

Chapter 3

Methodology

This chapter describes the proposed solution to create the power-saving strategy for the eBridge collaborative network, all software/hardware modifications, experiments and results, and the overall power-saving strategy. This section is organized in the following way: in section 3.1 the process of updating the hardware and software of the eBridge network in area 3.3 there is a description of the hardware setup used for the analysis of the eBridge 2019, in section 3.4 there is a description of the code profiling strategy and the tools used and created for this purpose. And finally, in area 3.5, there is a description of the procedure and tools used to perform an instruction-level power analysis of the network.

3.1 Hardware and Software Update

As it can be seen in figure 1.3 in chapter 1 the eBridge 2019 sensor node was designed using a Raspberry Pi (RPI) 3B+ and Python2.7. The Raspberry3 B+ system would be in production up to January of 2026 [41]. By the time this thesis work was started in 2020, the newest Raspberry PI system on the market was model 4B, so it was critical to evaluate the performance of the eBridge sensor node in this model. Aside from the RPI model upgrade, other aspects of the sensor node were updated, like the cellular communication board; the main goal was to allow the system to connect to the 3G and 4G cellular networks. The previously used SIM900-based communication module was replaced by one using a SIM7600G-H due to its connectivity to all LTE Cat-4 networks globally. Also, Costa Rica no longer has a GPRS network. A summary of these can be seen in table 3.1 and in figure 3.1.

The WaveShare SIM7600G-H 4G Hat was selected as the new communication board for the eBridge 2021 sensor node [42]. The module can be seen in figure 3.2. Some of the features of why the board was selected are:

- Allow the use of *Qualcomm MSM Interface* (QMI) library, which allows a higher

Table 3.1: Previous vs Current eBridge Sensor Node Comparison

Component	**Previous Model**	**Actual Model**
Node Name	2019 hardware configuration	2021 hardware configuration
Controller	RaspberryPI 3 B+ CPU: 1.4 GHz QuadCore Cortex A53 (ARMV7) RAM: 1 GB LPDDR2	RaspberryPI 4 B CPU: 1.5 GHz QuadCore Cortex A72 (ARM v8) RAM: 4GB LPDDR4
Cellular Communications Card	SIM900 Network: GSM ~2G	SIM7600G-H Network: 3G and (LTE) 4G
Sensor	Ultrasonic Sensor ->Range up to 10m SRF08 ->I2C Consumption: ~15 mA Acquiring ~3mA standby	Ultrasonic Sensor ->Range up to 4 m SRF05 ->Digital I/O Consumption: ~15 mA Acquiring ~2mA standby
XBee	XBee-Pro S2C DigiMesh	XBee-Pro S2C DigiMesh

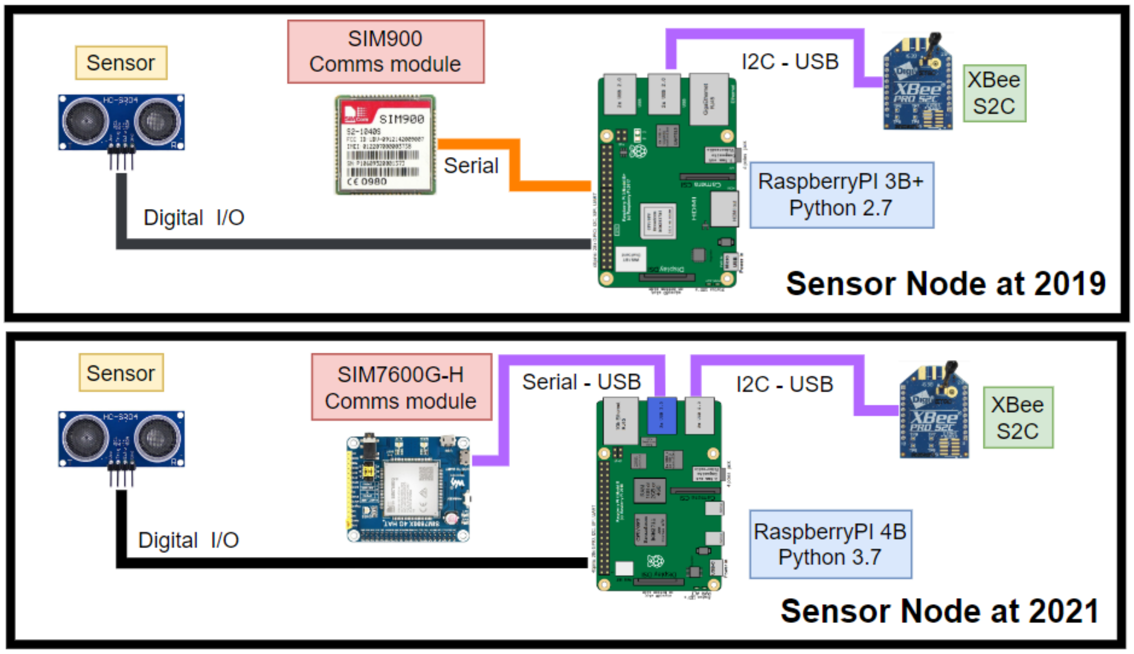


Figure 3.1: eBridge Sensor 2019 vs 2021 Nodes. Adapted from [10], [12].

level of abstraction compared to the AT commands, which is the common API for cellular communication boards.

- Provides an easy method to check the signal strength of the cellular connection.
- Board has a low-power mode that would allow turning off cellular communication to save power.
- Capabilities of Interfacing with cellular networks in table 3.2.

Using the WaveShare SIM7600G-H 4G Hat, the eBridge Network can be deployed globally whenever there is 2G, 3G, and 4G coverage. Also, the board includes the possibility of using GPS to locate the nodes; this could be used to create maps of eBridge nodes' locations and alarm locations.

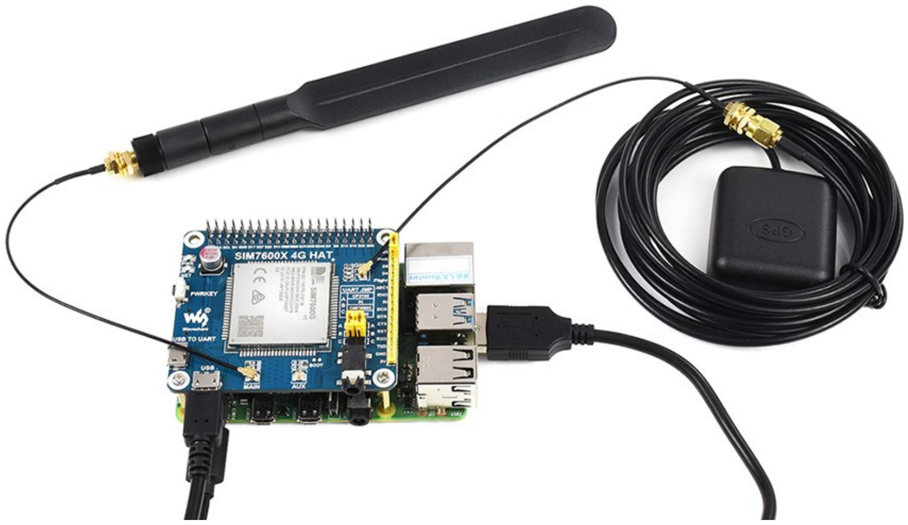


Figure 3.2: WaveShare SIM7600G-H 4G Hat [42].

Table 3.2: Frequency Bands Supported by SIM7600G-H

Frequency Bands supported by SIM7600G-H	
LTE Cat-4	**LTE-FDD**: B1/B2/B3/B4/B5/B7/B8/B12/B13/B18/B19/ B20/B25/B26/B28/B66 **LTE-TDD**: B34/B38/B39/B40/B41
3G	**UMTS/HSDPA/HSPA+**: B1/B2/B4/B5/B6/B8/B19
2G	**GSM/GPRS/EDGE**: 850/900/1800/1900 MHz

3.2 RPI3 B+ vs RPI4 B Comparison

In this section, there will be a comparison of the SoC BCM2711 used in RPI4 B and the Broadcom BCM2837B0 SoC in RPI3 B+. The BCM2711 SoC architecture is a considerable improvement from previous models as it includes a faster GPU that allows the use of up to 4K video, a direct PCIe connection with the USB 2 and 3 ports; a native Ethernet controller, and the capability of using up to 8 GB of RAM. It also includes an independent L1 cache and a shared L2 cache. Details of these can be seen in figure 3.3. When further looking at how the A72 is integrated into the BCM2711, it is possible to get to a diagram like the one in figure 3.4.

In the work done by [45], there is a comparison of the BCM2711 and BCM2837B0 SoC that can be seen in figures 3.5 and 3.6. The communication bandwidth of the SoC is increased by a factor of three compared to previous SoC versions. Also, the change in the BCM2711 allows a considerable improvement in the access to memory by changing from a maximum of 1 GB DDR2 memory in the BCM2837B0 to up to 8GB DDR4. All of these factors improve the performance as seen in [46]; however, they also increase the power consumption.

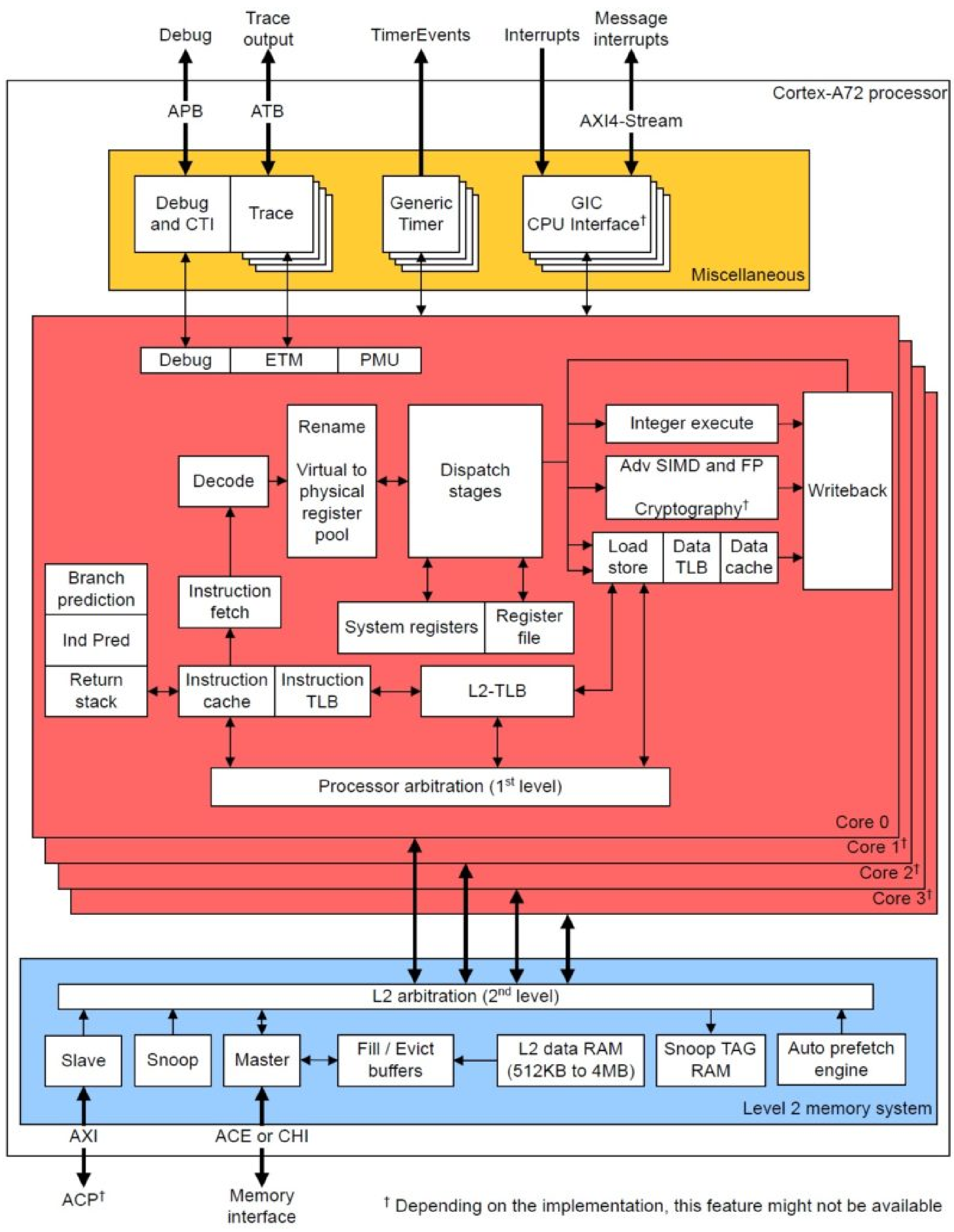


Figure 3.3: Cortex A72 (ARM v8) Processor SoC Diagram [43].

3.3 Hardware setup for experiments

The hardware setup in figure 3.7 was installed to run the test for the purpose of this work. Three bridges were configured; bridge number one has two sensor nodes with cellular and XBee communication, and bridges two and three only have one sensor node each, with cellular communication. Out of the four devices, there are two RPI model 3B+ and two

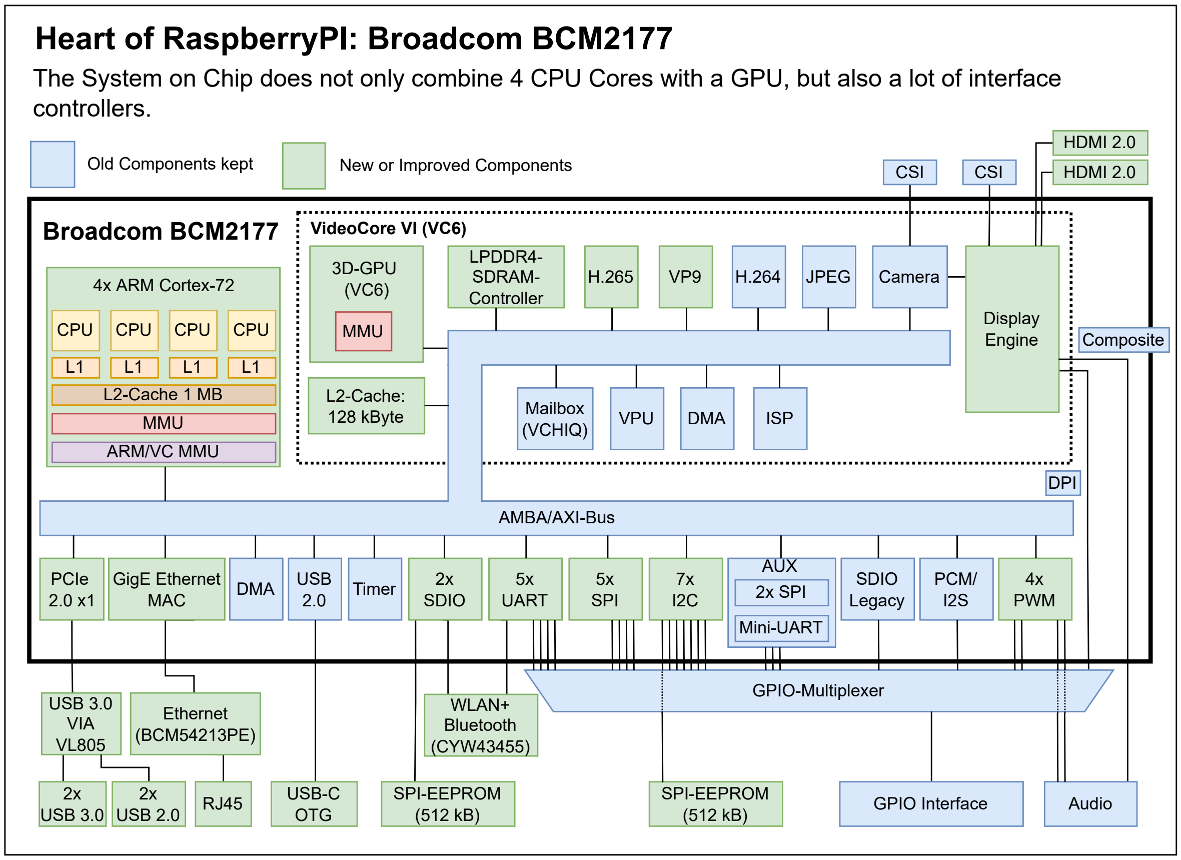


Figure 3.4: BCM2711 SoC Diagram [44].

model 4B. The goal of having the two old RPI is to measure the effectiveness of the power-saving strategy on the same plant used by [4]. Also, since the RPI3 B+ is going EOL (End of Life) in January of 2023 based on [47], it is critical to evaluate the power performance of the eBridge network in an RPI4 B system.

The **2021 hardware configuration** described in table 3.1 is done both in RPI3 B+ and RPI4 B systems. The goal is to compare is power performance of both units with the SIM7600G-H node and will be named **eBridge 2021 HW**, where HW stands for **hardware**.

3.4 Code Profiling Strategy

This section will cover the code profiling strategy for the eBridge code. As mentioned in section 2.6.2, measuring CPU, memory, disk, and the device network utilization, the initial strategy would be to focus on the CPU. A good understanding of CPU utilization is a crucial component of DPM and DVFS techniques.

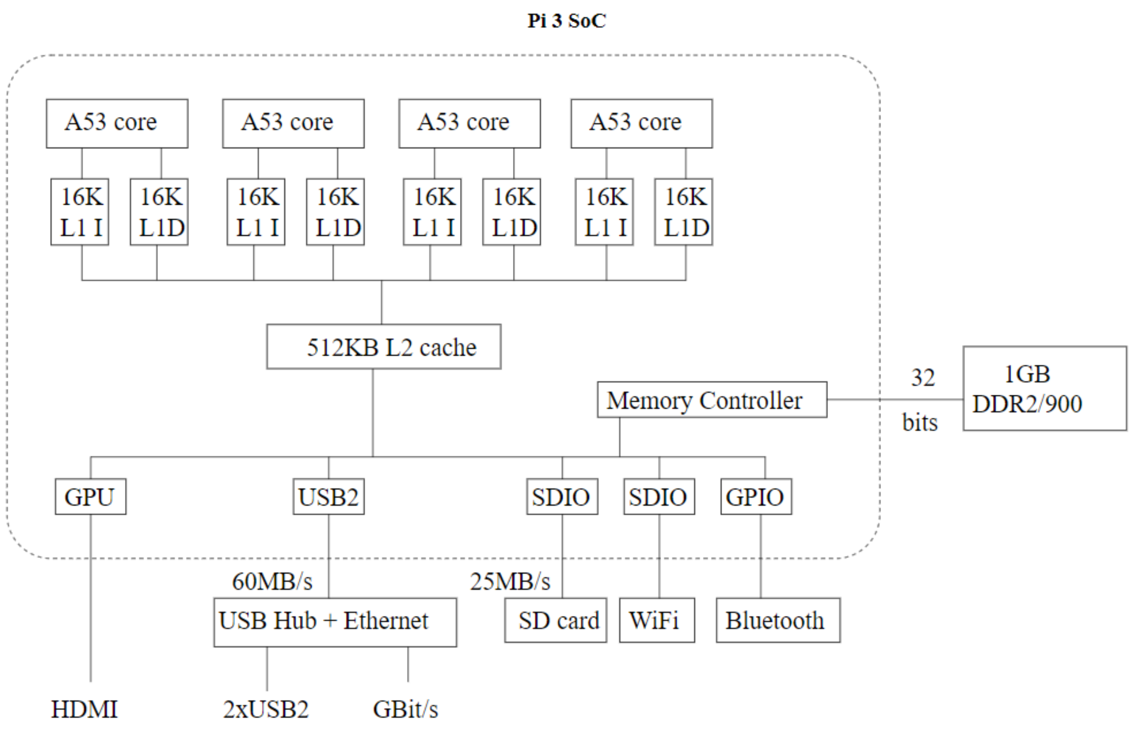


Figure 3.5: BCM2837B0 SoC Diagram [45].

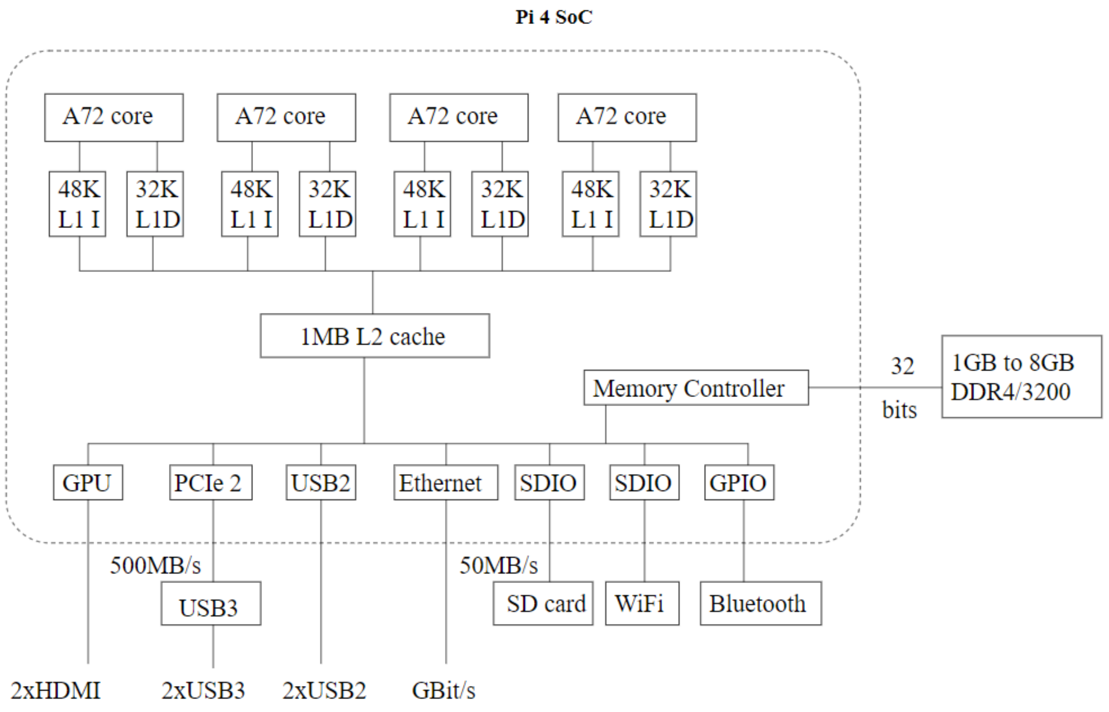


Figure 3.6: BCM2711 SoC Diagram [45].

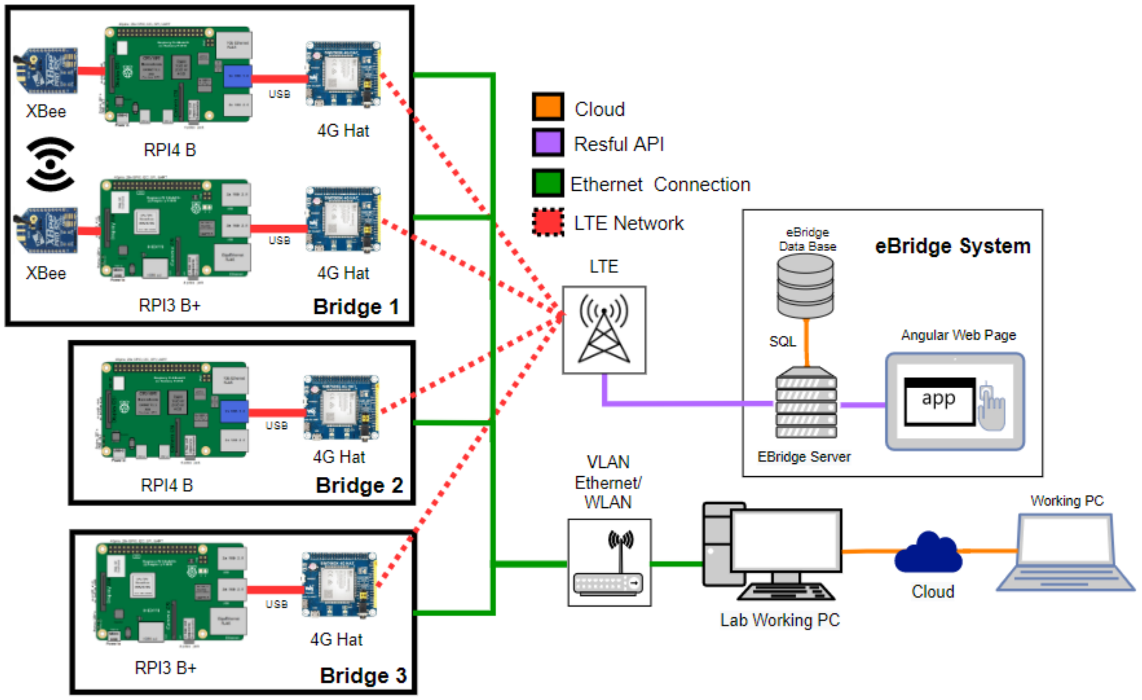


Figure 3.7: Current Hardware Setup eBridge.

3.4.1 Reviewing Existing Profilers

As described in figure 3.1, the 2021 model of the eBridge code was updated from Python 2.7 to Python 3.7, the newest version natively compatible with Raspbian Lite. The first step of the process was to investigate code profiling tools available to Linux distributions. Some of the desired features for a profiler were:

- Ability to profile at least CPU and other features like memory, disk, network, and peripheral usage over a long period of time.
- Logging capabilities, it is desired that the system can log the data in easily consumed file formats like CSV (comma separated value) or tab-delimited.
- Low CPU utilization. It is desired that the profiler has a low resource utilization.
- Low code execution time impact. It is desired that the profiler does not slow the code more than 10-15%.
- Support for the A53 and A72 ARMV7 CPUs.
- Line profiling is a desired feature since it would allow having a better idea of the impact of individual functions.

Table 3.3 summarizes the reviewed profilers. Based on these results, a custom profiler based on the combination of several profiling tools is created. Existing profilers do not include the possibility of getting instruction level measurements for multiple variables of interest like CPU Load, Memory, Disk, etc. The new profiler is capable of this. The profiler also includes several tools that allow its users to plot and analyze the captured data in an easy and graphical way. These tools can be reused by other teams performing a similar analysis on other systems.

Table 3.3: Profiler Comparison

Profiler	Logging	File Format	Impact	ARMV7	Profiling Feature								
					CPU	Memory	Disk	Network	Peripherals	Voltage	Frequency	Idle	Line
bcmstat [48]	No		Undetermined	X	X	X	X	X	X	X	X	X	
psutil [49]	Yes	Yes	Low	X	X	X	X	X	X	X	X	X	
top [50]	No		Low	X	X	X						X	
Py-Spy [51]	Yes	Yes	Low	X									X
Scalene [52]	Yes	No	Low	X	X	X	X						X

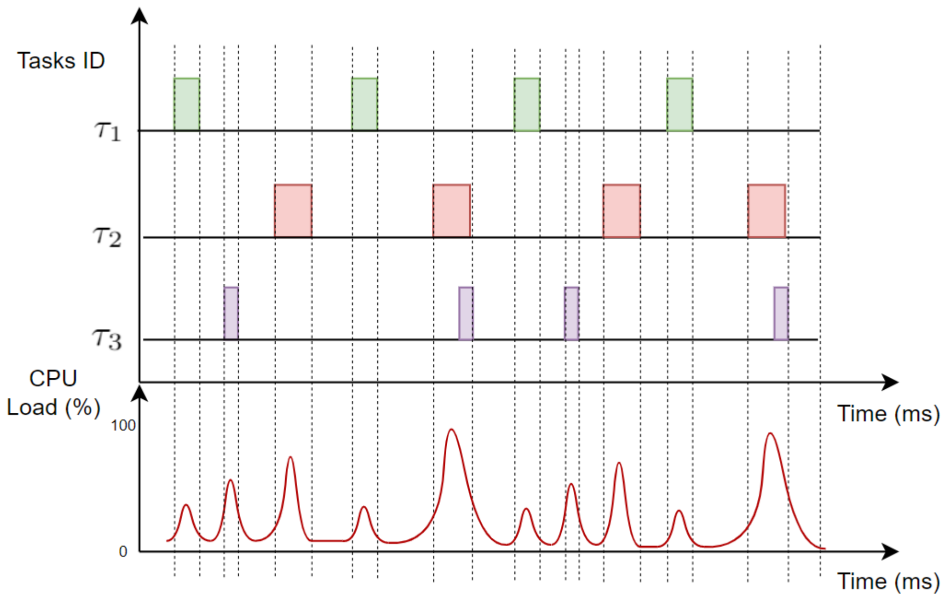


Figure 3.8: Task execution time vs CPU Load profiling strategy.

3.4.2 Creating a Custom Profiler

This section describes creating a custom profiler based on several existing tools. As it was mentioned in section 3.4.1, the selected tools were **py-spy** and **psutil**. Using these libraries, the Python *time* [53] library would be used to track the start and stop time of the eBridge code main functions to create a Gantt diagram. A conceptual representation of this can be seen in figure 3.9, and it was named the *Event Logger*.

After creating the Gantt diagram of the eBridge code main functions, the goal is to map this information against CPU load information. A concept of this can be seen in figure 3.8. This information would be critical to creating a task execution model of the code. Another desired feature is to map the load of the four cores on the A53 and A71 ARMV7 to create a plot like the one in figure 3.10.

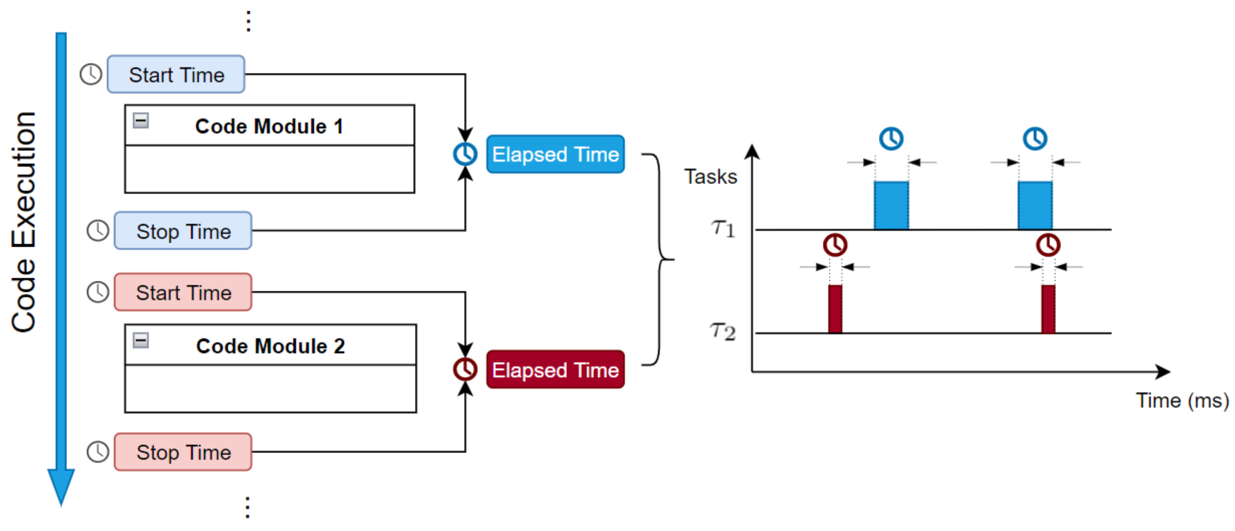


Figure 3.9: Event Logger Concept

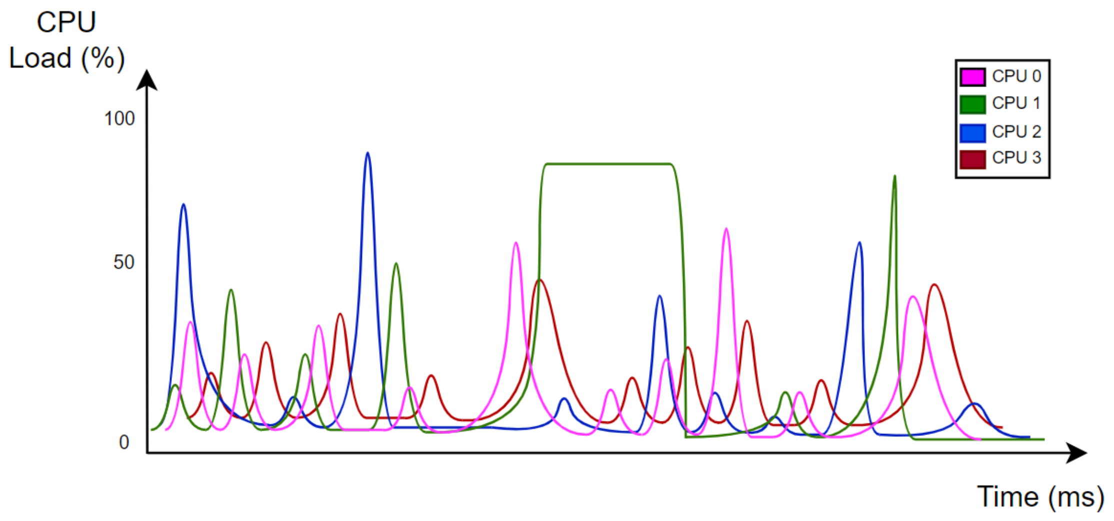
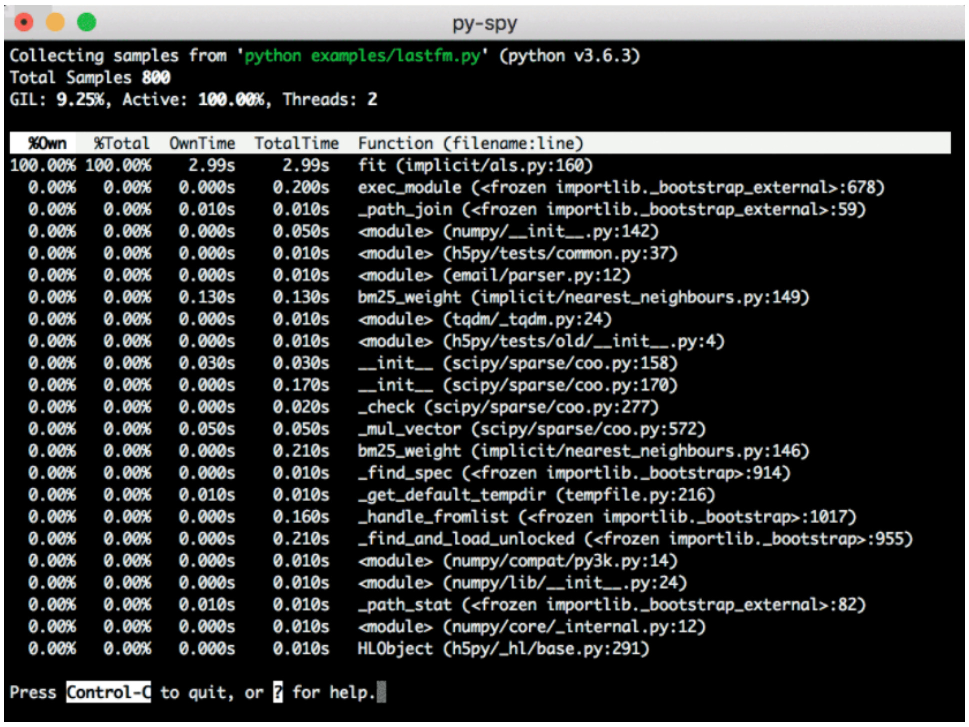


Figure 3.10: CPU Load Plot Concept.

Also, in parallel, the goal would be to use **py-spy** to identify what functions the code spends more time on. An example of the py-spy output can be seen in figure 3.11. This information would focus on the code refactoring efforts, following the process in figure 3.12. Since the eBridge application does not have Real-Time requirements, the same approach as the one used by [24] is used, where the relative deadline D_i is equal to T_i, which is the minimum time between executions of a task τ_i.

The code execution information that is returned by the **Event Logger** is used to build the vector $\vec{C_i^m} = \left(\vec{C_i^1}, \vec{C_i^2}, \ldots, \vec{C_i^M}\right)$ of execution profiles described in section 2.1.1. However, since the eBridge code is Python-based and thus depends on the Python GIL (Global Interpreter Lock) [54] is not possible to get the details of tasks execution for specific cores. Due to this, the execution profiles for a given task (τ_i) will be represented as



```
Collecting samples from 'python examples/lastfm.py' (python v3.6.3)
Total Samples 800
GIL: 9.25%, Active: 100.00%, Threads: 2

  %Own   %Total  OwnTime  TotalTime  Function (filename:line)
100.00% 100.00%   2.99s     2.99s   fit (implicit/als.py:160)
  0.00%   0.00%   0.000s    0.200s   exec_module (<frozen importlib._bootstrap_external>:678)
  0.00%   0.00%   0.010s    0.010s   _path_join (<frozen importlib._bootstrap_external>:59)
  0.00%   0.00%   0.000s    0.050s   <module> (numpy/__init__.py:142)
  0.00%   0.00%   0.000s    0.010s   <module> (h5py/tests/common.py:37)
  0.00%   0.00%   0.000s    0.010s   <module> (email/parser.py:12)
  0.00%   0.00%   0.130s    0.130s   bm25_weight (implicit/nearest_neighbours.py:149)
  0.00%   0.00%   0.000s    0.010s   <module> (tqdm/_tqdm.py:24)
  0.00%   0.00%   0.000s    0.010s   <module> (h5py/tests/old/__init__.py:4)
  0.00%   0.00%   0.030s    0.030s   __init__ (scipy/sparse/coo.py:158)
  0.00%   0.00%   0.000s    0.170s   __init__ (scipy/sparse/coo.py:170)
  0.00%   0.00%   0.000s    0.020s   _check (scipy/sparse/coo.py:277)
  0.00%   0.00%   0.050s    0.050s   _mul_vector (scipy/sparse/coo.py:572)
  0.00%   0.00%   0.000s    0.210s   bm25_weight (implicit/nearest_neighbours.py:146)
  0.00%   0.00%   0.000s    0.010s   _find_spec (<frozen importlib._bootstrap>:914)
  0.00%   0.00%   0.010s    0.010s   _get_default_tempdir (tempfile.py:216)
  0.00%   0.00%   0.000s    0.160s   _handle_fromlist (<frozen importlib._bootstrap>:1017)
  0.00%   0.00%   0.000s    0.210s   _find_and_load_unlocked (<frozen importlib._bootstrap>:955)
  0.00%   0.00%   0.000s    0.010s   <module> (numpy/compat/py3k.py:14)
  0.00%   0.00%   0.000s    0.010s   <module> (numpy/lib/__init__.py:24)
  0.00%   0.00%   0.010s    0.010s   _path_stat (<frozen importlib._bootstrap_external>:82)
  0.00%   0.00%   0.000s    0.010s   <module> (numpy/core/_internal.py:12)
  0.00%   0.00%   0.000s    0.010s   HLObject (h5py/_hl/base.py:291)

Press Control-C to quit, or ? for help.
```

Figure 3.11: Py-Spy example [51].

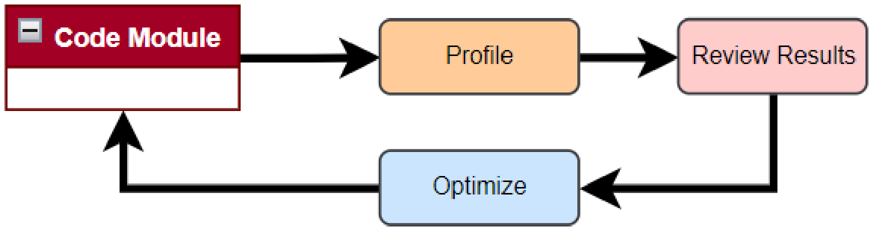


Figure 3.12: Code Optimization Strategy.

$\vec{C_i^f} = \left(C_i^1, C_i^2, \ldots, C_i^F\right)$. Also, each $C_i^F = (C_1, C_2, \ldots, C_n)$ vector, which is the group of execution times at a frequency F.

3.5 Power Measurement of eBridge System

To measure the power of the eBridge sensor node, it was decided to use the configuration in figure 3.13, using a shunt resistor and High-Resolution Voltage Acquisition board with the features described in section 2.2. The selected board was the **PXIe-4309** from National Instruments [55], which can be seen in figure 3.14. The shunt resistor shown in figure 3.13 is a 75 $mV/100A$ or 0.75 $m\Omega$. Some of the characteristics of the PXIe board are shown below:

- Anti-Alias filter.

- 18-bit 28-bit Resolution in voltage ranges ±0.1 V, ±1.0 V, ±10.0 V, and ±15.0 V.
- Up to 2 M S/s sample rate.
- Resolution of 9.6 μV_{pk-pk} for ±10.0 V range.

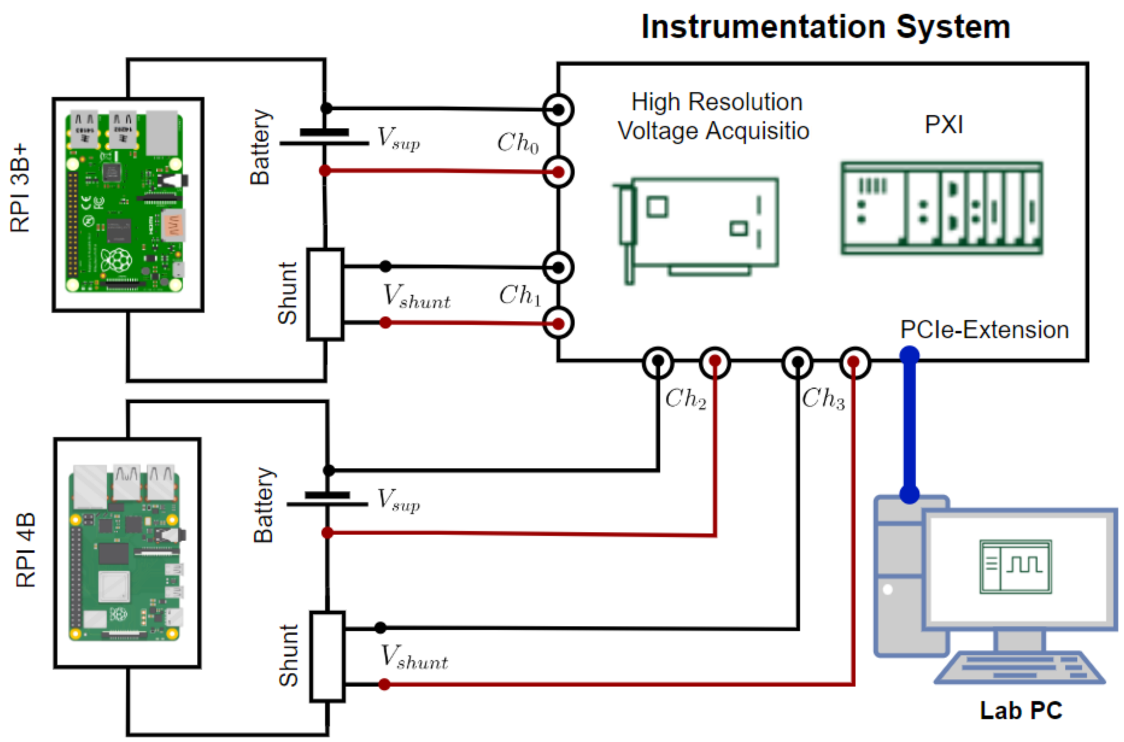


Figure 3.13: Power Measurement Hardware Setup for eBridge Sensor Node.

Figure 3.14: PXIe-4309 Flexible Resolution PXIe Analog Input Module [55].

The four voltage values captured by the PXIe-4309 board Ch_0, Ch_1, Ch_2, and Ch_3 channels, which are the V_{sup} and V_{shunt} voltages for both the RPI 3B+ and the RPI 4B respectively. The V_{shunt} value is used with the 0.75 $m\Omega$ resistance with Ohm law to calculate the current using the equation $I_{shunt} = V_{shunt}/R_{shunt}$. These values are then used to compute the node power using equation $P_{node} = V_{sup} \cdot I_{shunt}$.

Following the same logic in section 3.4.2, the vector of energy consumption for a given task (τ_i) will be represented as $\overrightarrow{E_i^f} = \left(E_i^1, E_i^2, \ldots, E_i^F\right)$, with f as the frequency configuration for the RPI cores. Each $E_i^F = (E_1, E_2, \ldots, E_m)$, which is the group of power consumption of a task (τ_i) at a frequency F. The nomenclature use in this document to represent the $\overrightarrow{C_i^F}$, $\overrightarrow{E_i^f}$ vectors of different tasks τ_i is going to be $\overrightarrow{C_{i,\tau_i}^f}$ for the execution profiles, and $\overrightarrow{E_{i,\tau_i}^f}$ for energy consumption.

In the next section 3.5.1 there will be a description of the Lithium-ion batteries used to test the eBridge network.

3.5.1 Batteries used for testing

This section includes a description of the batteries used for testing. At the end of the work done by [4], the eBridge 2019 node was tested using a Lithium-ion battery model ICR18650 4400mAh 3.7V [56] in figure 3.15. When testing this battery with the 2021 eBridge HW configuration using both RPI3 B+ and RPI4, it was possible to observe how the battery could not provide both powers when initiating an LTE connection, which caused the unit to reboot itself.

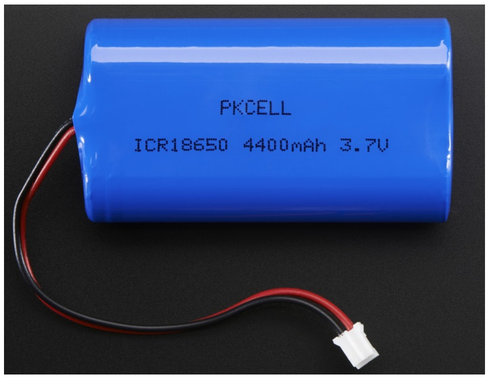


Figure 3.15: ICR18650 4400mAh 3.7V Battery[56]

When investigating the local market for batteries with similar charges, it was possible to find the MI Power Bank and Wall Charger CBQ01ZM with a capacity of 5200 mAh [57] in figure 3.16, which has the benefit that it can be easily recharged with external hardware by just connecting it to the 120 VAC power outlet. Also, it was decided to use the MI Power Bank 3 Ultra Compact 10 000 mAh battery to test the system with a larger battery capacity.

In table 3.4, there is a summary of the battery systems used in this work. There is a total of four 10 000 mAh and four 5 200 mAh four batteries; for simplicity, it will be named 10 000 mAh - [ID], where ID could have values from one to four. For example, 10 000 mAh - 3. The same nomenclature will be used for the 5 200 mAh batteries, for example, 5 200

Figure 3.16: MI Power Bank and Wall Charger CBQ01ZM - Capacity 5200 mAh [57]

Figure 3.17: MI Power Bank 3 Ultra Compact - Capacity 10 000 mAh [58]

mAh - 4.

Table 3.4: Summary of Batteries used for testing

Battery	Capacity (mAh)	Technology	Output	Quantity
Mi Power Bank and Wall Charger	5200 (18 Wh)	Lithium-Ion	5 V / 2.4 A	4
Mi Power Bank 3 Ultra Compact	10000 (37 Wh)	Lithium-Ion	5V / 2.4 A	4

Chapter 4

Analysis of the eBridge Network

The first part of this chapter includes the results of applying the methodology described in chapter 3 over the **eBridge 2019** network. Section 4.1 describes the line profiling, code profiling for master/primary and slave/secondary nodes in subsections 4.1.2 and 4.1.3 respectively. Section 4.2, includes the battery tests results for master/primary nodes in subsection 4.2.1, and the ones for slave/secondary nodes in subsection 4.2.2. There is a task interaction analysis for both types of nodes in section 4.3, and finally, in section 4.4, there is a detailed description of the Standby power consumption of the eBridge 2021 HW configuration on both RPI3 B+ and RPI4 B nodes.

As was described in the chapter 1, the eBridge network has multiple communications requirements for both inter-node and node-to-server communications. Additionally, the network can easily interface with different sensors and other peripheral devices through SPI, I2C, UART, USB, etc., while maintaining low power consumption. These characteristics are compatible with SoC-based embedded devices like the RPI3 B+ and RPI4 B systems. Also, these units have a Debian-based Linux environment compatible with multiple software tools and programming languages.

4.1 eBridge 2019 Node Analysis

The Initial results of the eBridge 2019 code are covered in this section; the line profiling results are in subsection 4.1.1, and the detailed results of the CPU and Events profiling to both master/primary and slave/secondary nodes in subsections 4.1.2, and 4.1.3. This analysis is performed using the hardware setup described in figure 3.7 and section 3.3. An assessment of the CPU profiling results using **psutil** with a constant power supply for both master/primary and slave/secondary nodes are covered in this section. The results for the line profiling can be observed in section 4.1.1, the results of the the profiling for a Master/Primary 2019 eBridge node can be seen in the section 4.1.2, and the ones for a Slave/Secondary node in section in 4.1.3. For the purpose of this chapter, the original eBridge code done by [4] is referred to as **eBride 2019 code**, and the new implementation is referred to as **eBridge 2021 code**. In the next section 4.1.2 the line profiling results for the **eBridge 2019 code** running on both RPI3 B+ and RPI4 B nodes is detailed.

4.1.1 eBridge 2019 code Line Profiling Results

The line profiling of the **eBridge 2019 code** done by the end of [4] was performed in RPI3B+ and RPI4 B nodes with and without XBee communication for over 200 hours, using the line profiling tool called **py-spy**. Some of the most extended tests took up to 38 hours. The results of the line profiling help identify the following functions as a focal point. These functions of interest would be named as τ_i following the instruction level approach described in section 2.1.1. These functions are described below:

- **XBee Received - ID 0** (τ_0): Used by a master/primary node to receive measurement reports from all of the slave/secondary nodes connected to it.
- **XBee Send - ID 1** (τ_1): Send all of the measurement reports stored in the memory of a slave/secondary node to a master/primary node.
- **Upload to Server - ID 2** (τ_2): This function allows the main system to upload the measurement reports generated by the master/primary node itself and all of the slave/secondary nodes connected to it, to the server.
- **Generate Report - ID 3** (τ_3): Generates a report out of the measurements performed by the node. This function uses the results from the **Get Median - ID 4** (τ_4) function, and it performs an evaluation of the alert level and the collaborative network's overall alarm level. These reports are stored in the node's internal memory until the node is ready to upload the data to the server in the case of a master/primary node or send over XBee in the case of a slave/secondary node.
- **Get Median - ID 4** (τ_4): This operation englobes the distance measurements and gets the average value of a series of measurements. For example, in the case of water level, at least 10 measurements are taken and then average each time this function is called. This value is used by the **Generate Report - ID 3** (τ_3) to generate reports.

Also, several threads were identified as focus points:

- **SetDateTime**: This thread is in charge of keeping the time of the node; this time is used by the functions above to decide when to perform actions. This thread is highly CPU intensive.
- **Recibir (Receive in English)**: This thread is in charge of receiving XBee messages, and it is only launched on master/primary nodes.

- **Device.Run**: This is the main thread for the **eBridge 2019 code**, which is part of the **Device** class defined in [4]. This thread controls the data acquisition, report generation, and data uploading to the server through the LTE network. Also, is in charge of communicating with the receive thread to process incoming reports from slave/secondary nodes, process alarms, and upload data to the server.

The analysis and optimizations focus on the functions above using the implement **Event Logger** described in figure 3.9. With these results the list of tasks $\Gamma = \{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$ is defined and would be used through the course of the document

4.1.2 eBridge 2019 Master/Primary CPU, Events and Power Profiling

In this section, you can find the results of the code profiling for primary node algorithms running on RPI 3B+ and RPI4. The details of the tests executed on the eBridge 2019 network running as master/primary can be seen in tables 4.1 and 4.2.

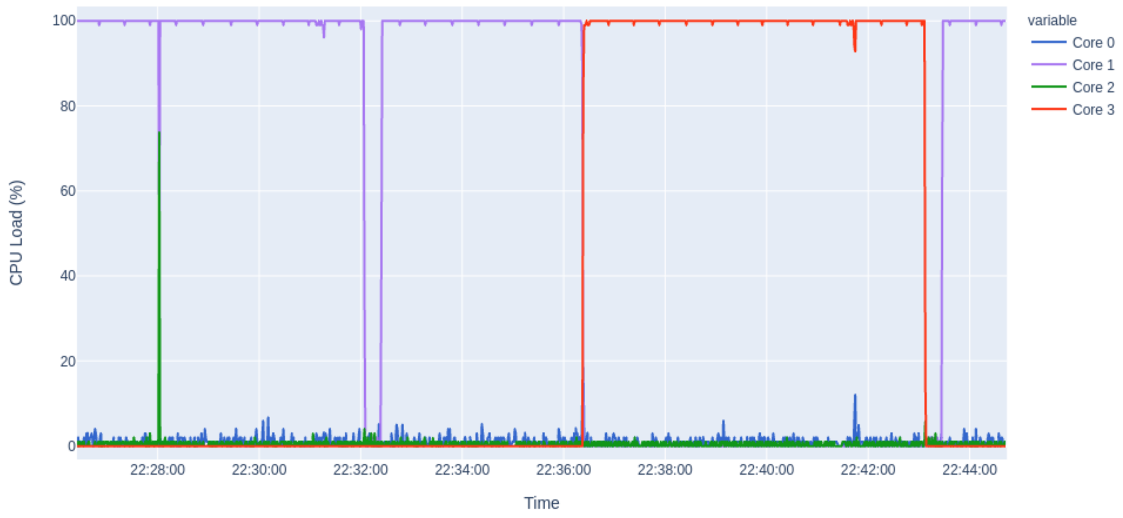


Figure 4.1: CPU Load (%) for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 4.1

As shown in figure 4.1, at least one of the CPUs of the system is at 100% utilization. When doing a more in-depth analysis of the statistical data for the CPUs behavior, it is possible to see the results in table 4.4. The **Core_1** has a mean CPU load of 66.34 % with a standard deviation(σ) of 47.08 % while the other cores had a mean value in less than 20 % for an RPI3 B+ node. To better understand CPU Load behavior the box plot in figure 4.6. This behavior is expected for a multi-threaded code like the **eBridge 2019 code** due to the Python GIL [54].

Table 4.1: Measurement Summary eBridge code 2019 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI3 B+

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI3 B+
Controller	Raspberry PI 3 B+
Device ID	RPI3-2
Code Running	eBridge 2019 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Master/Primary
Total Test Duration	1 hour 12 minutes and 6 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples by CPU Core	4296 samples
CPU Frequency applied to all cores(f)	1.4 GHz
Power Measurement Summary	
Power Source	CanaKit RPI3 B+ 5V 2.5A [59]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	8 652 000

When analyzing the distribution of the tasks over time, it is possible to obverse how several tasks overlap their execution, and in some cases, some of these tasks have longer execution times, as can be seen in figure 4.2, particularly it can be seen how both the **Get Median ID: 4** (τ_4) and **XBee Received ID:0** (τ_0) functions were taking an abnormal time to execute. When further analyzing these functions and checking the CPU load during this event, it was possible to get the behavior seen in figure 4.4. It is possible to see how at the time of these **stalling** event the CPU Load lowers down to 0%, in figures 4.3, and 4.5 there are zoomed views of this behavior.

This **stalling behavior** was caused due to shared resources among the measurement thread of which the **Get Median ID: 4** (τ_4) function is part, and the receiving data thread **XBee Received ID:0** (τ_0) is part of. The specific shared resource is a Python queue object named as **requestQueue** that was used to store measurements in the Primary/Master system from its own and Secondary/Slave systems sensors. In the way the eBridge network was implemented at the end of the work done by [4], it was possible

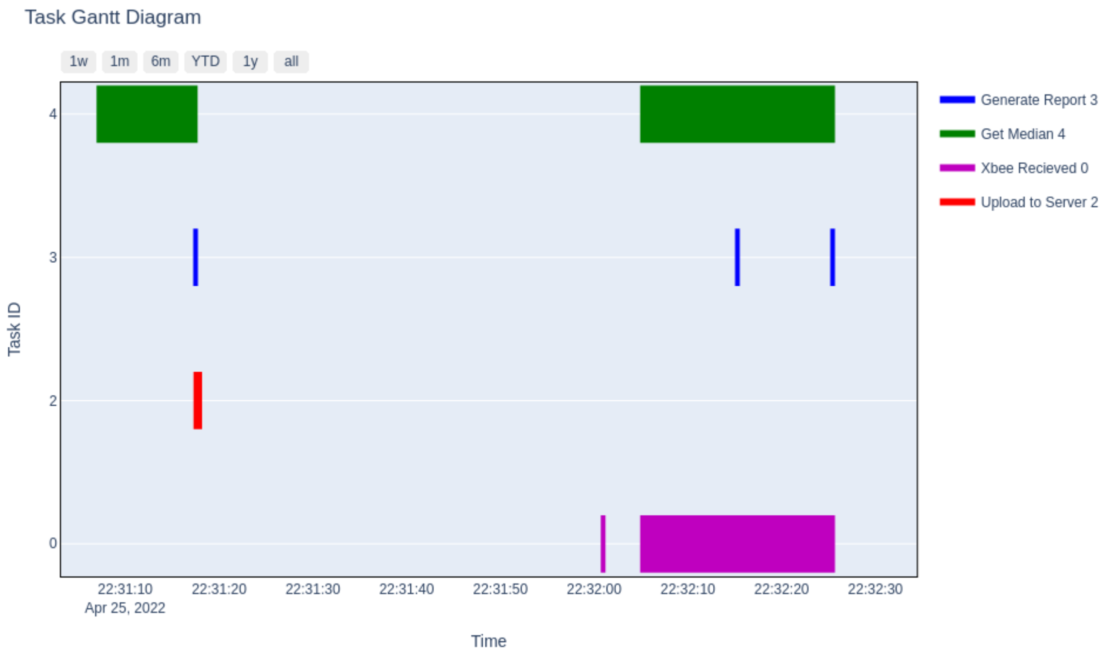


Figure 4.2: Tasks Gantt plot eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.1

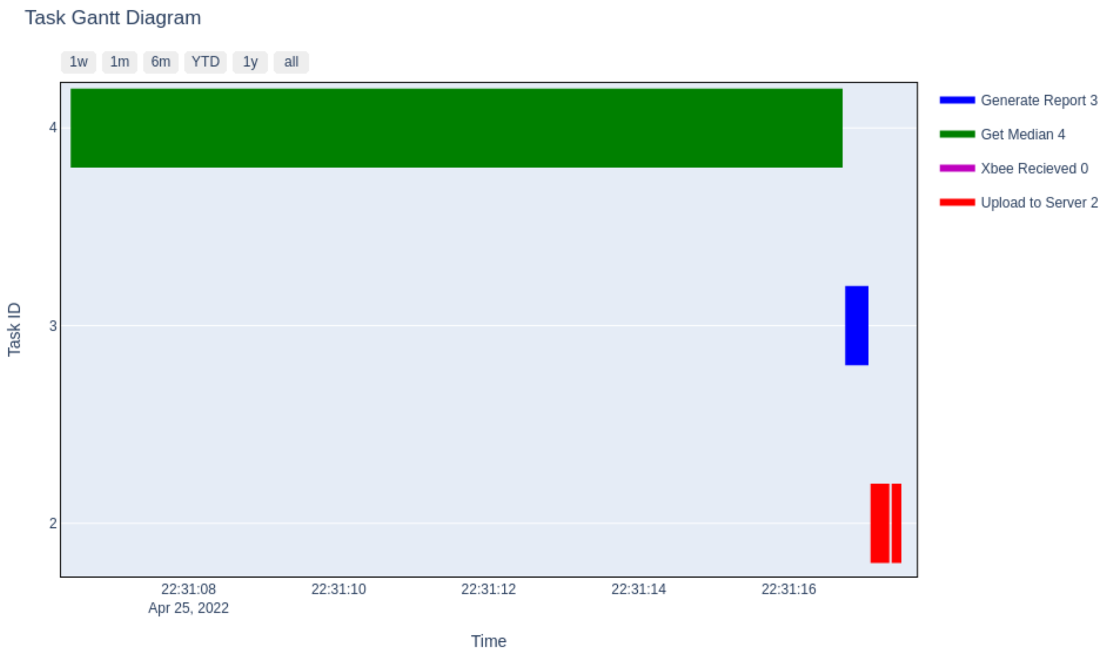


Figure 4.3: Tasks Gantt Zoomed plot eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+. Details of the test in table 4.1

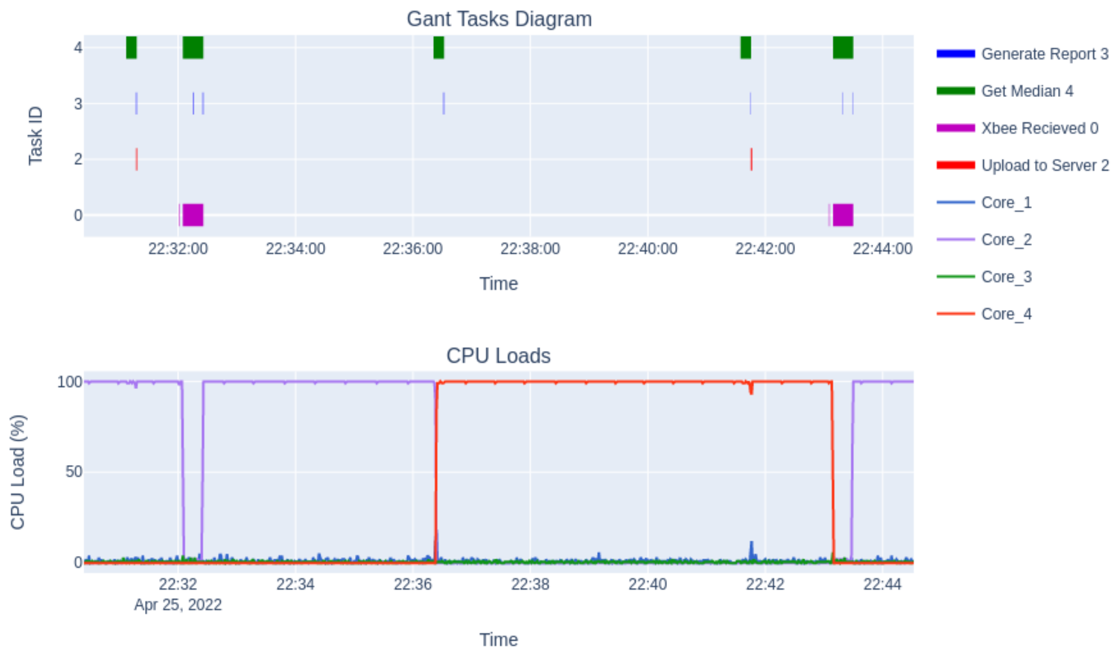


Figure 4.4: Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.1

to configure the **measurement** and the **sending message** frequencies. For both the master/primary slave/secondary nodes, the measurement frequency indicates how often a serious of measurements is performed on a specific node.

For example, for the test detailed in table 4.1, the measurement frequency is one minute, this means that the node performs a series of consecutive measurements, in this particular case 10, and it would get the average value of these measurements once every minute. This means the measurement value would be used to build a report that would be stored in the **requestQueue** object. In the case of the sending message frequency, the behavior changes depending on the type of node; for the master/primary node, this controls how often the node uploads the data to the server, while for a slave/secondary node, this controls how often this node sends its reports to the master/primary node.

One of the aspects of the eBridge network is that in a primary/master node, there is no awareness of when a slave/secondary node sends messages. For example, when the test described in table 4.1 was executed while the one in table 4.8 was also running on a node configured as a slave/secondary system as part of the same bridge and network. The relationship between both tests is discussed in section 4.1.3.

To evaluate the eBridge 2019 code done by [4] in a RPI4 B system, the test described in table 4.2 was performed. This test included the same type of configuration as the one in table 4.1, but the slave/secondary system had a different configuration, which can be

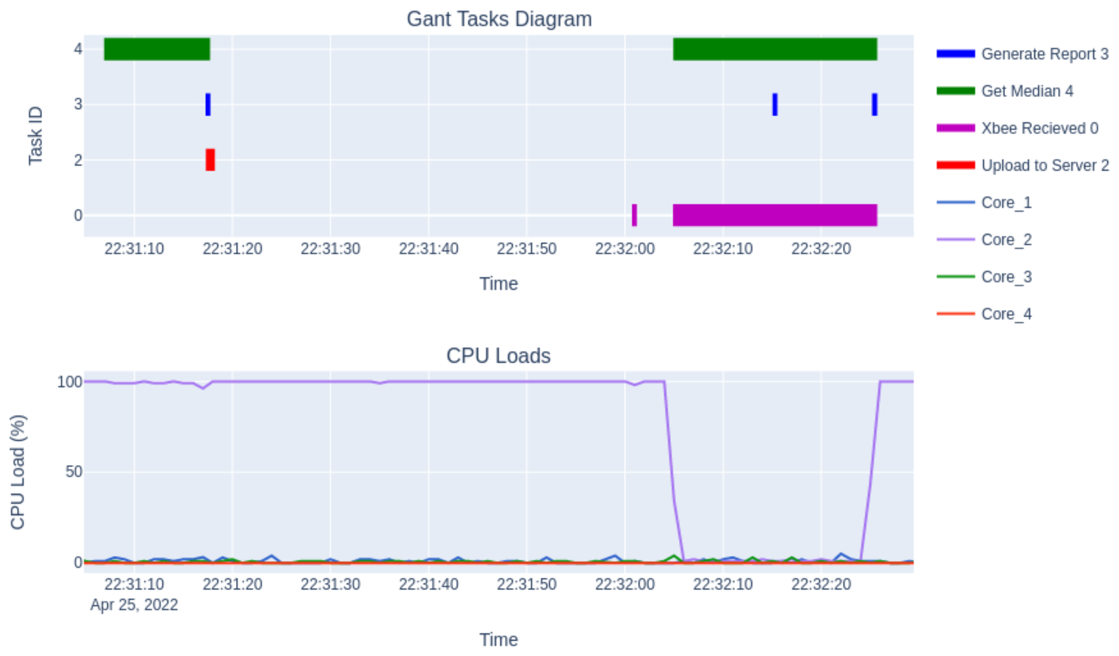


Figure 4.5: Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.1

seen in table 4.7. The CPU Load (%) plot is shown in figure 4.7.

As it could be observed in figure 4.8 the **stalling** issue between the **XBee Received ID:0** and the **Get Median ID:4** tasks is not present in the RPI4 B node, in contrast to the RPI3 B+ node. When further exploring the difference between the code execution between the two nodes, an analysis of the vector $\vec{C}_i^f$ is performed on both RPI3 B+ and RPI4 B nodes running as master/primary. These results can be observed on table 4.3, it can be seeing how there is a mean faster execution time of the $\vec{C}_i^f$ on the RPI4 B node for the **XBee Received ID:0** (τ_0) and the **Get Median ID:4** (τ_4) in comparison to the RPI3 B+.

In the specific case, the **XBee Received ID:0** has a mean execution time of 5.0689 s for an RPI3 B+ node, while for an RPI4 B system, the mean execution time is 0.1379 s, which is a 97.28 % smaller. This is because the A72 processor in the RPI4 B has higher processing power than the A53 in the RPI3 B+ as demonstrated in [61]. As described in section 3.2 the RPI4 B node, aside from higher processing power, also includes more communications controllers inside of the BCM2711 Broadcom SoC, in comparison to the RPI3 B+.

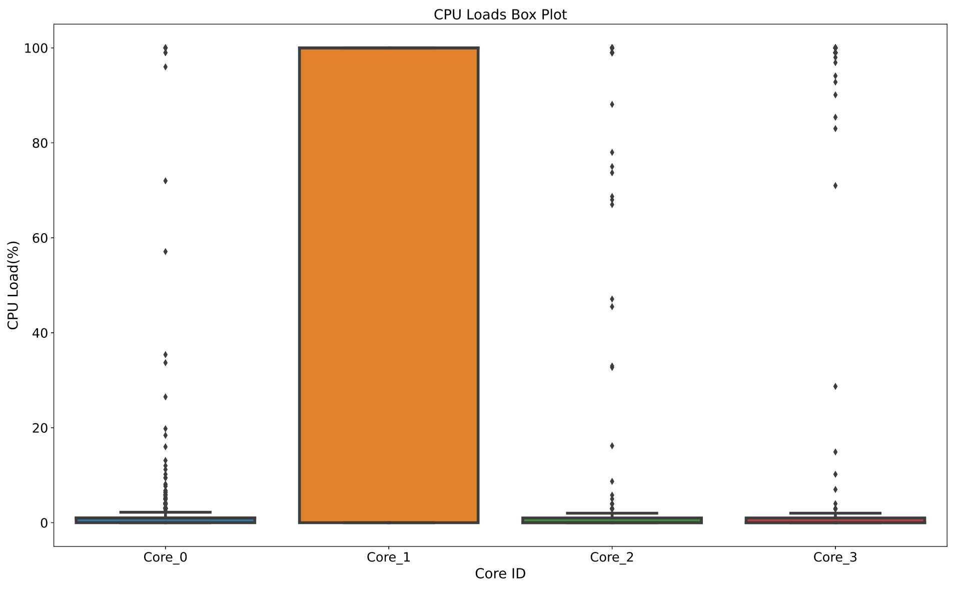


Figure 4.6: CPU Load Box (%) plot for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 4.1

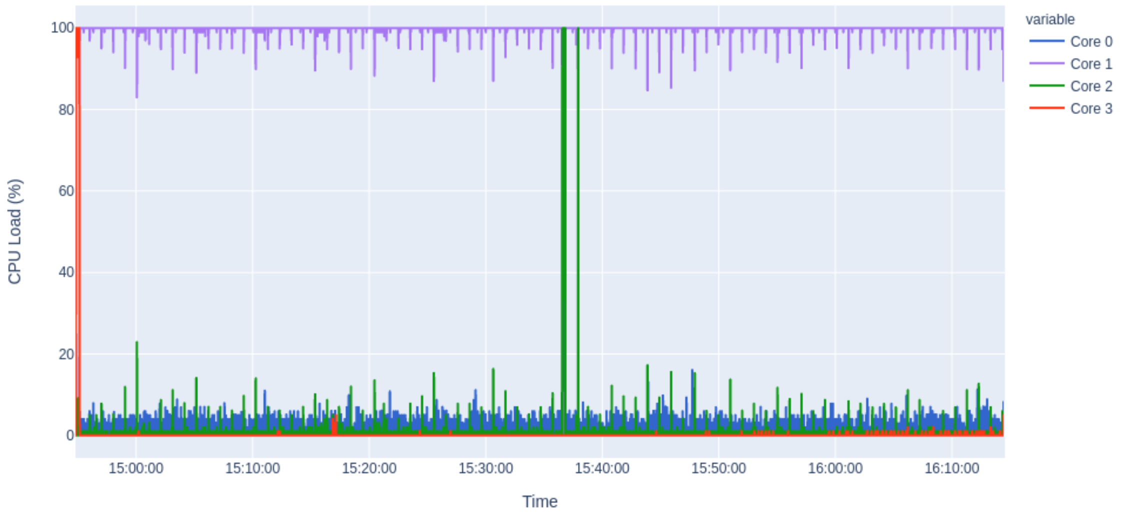


Figure 4.7: CPU Load % for eBridge 2019 code running as master/primary node running in eBridge 2021 HW configuration in RPI4 B with f = 1.5 GHz. Details of the test in table 4.2

In figure 4.9 it can be observed that even though there is overlapping of the **XBee Received ID:0** (τ_0) and the **Get Median ID:4** (τ_4) functions there is no change on the CPU behavior of the eBridge 2019 code running on a RPI4 B as a master/primary

Table 4.2: Measurement Summary eBridge code 2019 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI4 B

Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI4 B
Controller	Raspberry PI 4 B
Device ID	RPI4-2
Code Running	eBridge 2019 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Master/Primary
Total Test Duration	1 hour 19 minutes and 45 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples for Plot	4773 samples
CPU Frequency applied to all cores(f)	1.5 GHz
Power Measurement Summary	
Power Source	CanaKit RPI4 B 5V 3.5A [60]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	9 570 000

system. In the next section 4.1.3 an analysis of the eBridge 2019 code running on both RPI3 B+ and RPI4 systems working as slaves/secondary units is detailed. The results of these experiments allow an understanding of the difference between master/primary and slave/secondary nodes in CPU utilization, power consumption, and tasks' executions.

A review of the power information for RPI3 B+ and RPI4 B nodes running as master/primary. It is possible to obtain figures 4.10 and 4.11 for RPI3 B+; and figure 4.12 for RPI4. It can be observed that the **stalling** behavior in the RPI3 B+ node causes a power spike, while it is not present in the RPI4 B node. This is further addressed in section 4.3.

When doing a comparison of the $\vec{E}_i^f$ vector for an RPI3 B+ and RPI4 B in table 4.5, it is observable how the power can reach up to 4.4834 W on the RPI3 B+ during the stalling event for τ_4, while for RPI4 B there is a spike of 5.8934 W, which is 31.45 % higher. A similar pattern can be seen in table 4.6 where the average power of RPI4 B of 4.5788 W is 20.82 % higher than the RPI3B + average power of 3.7897 W.

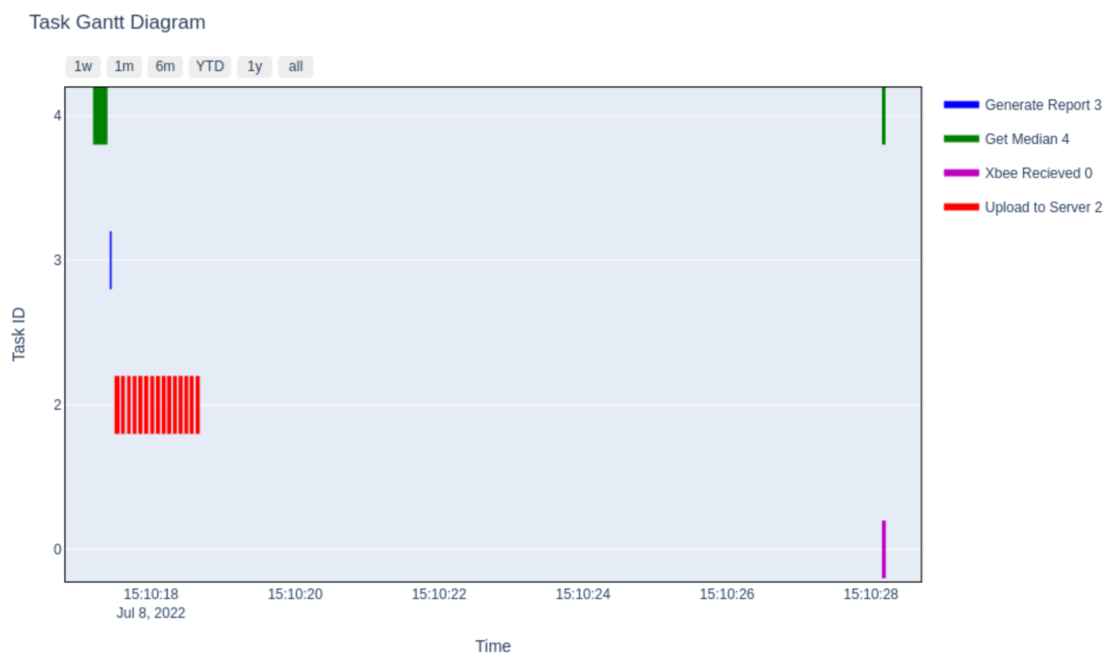


Figure 4.8: Gantt Plot for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2

Table 4.3: Comparison of $\vec{C}_i^f$ with f=1.4 GHz for RPI3 B+, and $\vec{C}_i^f$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2019 in eBridge 2021 HW configuration as master/primary. Test details in tables 4.1 and 4.2

Execution Time (s)				
Task ID	τ_4	τ_3	τ_2	τ_0
Running on RPI3 B+ node				
n	26	25	32	24
Mean	10.2114	0.0059	0.1454	5.0689
Std	0.0815	0.0056	0.0522	5.1774
Min	10.1304	0.0006	0.1079	0.0003
Max	10.2975	0.0161	0.2695	10.1485
Running on RPI4 B node				
n	109	108	135	60
Mean	0.1379	0.0132	0.0551	0.0275
Std	0.0579	0.0096	0.0049	0.0271
Min	0.0501	0.0005	0.0523	0.0003
Max	0.2167	0.0229	0.0726	0.0551

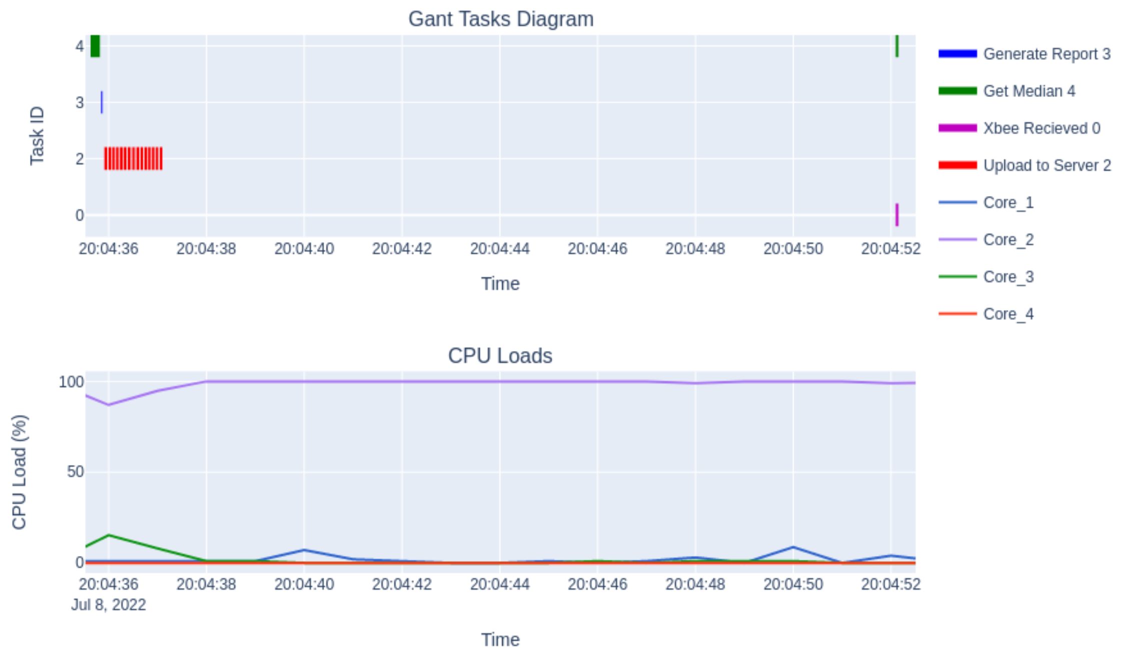


Figure 4.9: Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2

4.1.3 eBridge 2019 Slave/Secondary CPU,Events and Power Profiling

A similar study is done for a slave/secondary system with both an RPI3 B+ and an RPI4 B to evaluate the performance differences between running the eBridge 2019 code as a

Table 4.4: Comparison of Code CPU Load (%) statistical information of the eBridge 2019 code running on an RPI3 B+ with f = 1.4 GHz vs an RPI4 B as Master/Primary with f = 1.5 GHz. Test details in tables 4.1 and 4.2

CPU Load (%)				
Core	**1**	**2**	**3**	**4**
Code running on RPI3 B+				
Number of samples per core = 4296				
Mean	1.30	66.34	13.29	17.43
Std	5.59	47.08	33.48	37.78
Code running on RPI4 B				
Number of samples per core = 4773				
Mean	1.09	99.00	0.77	0.43
Std	2.75	9.08	5.77	6.30

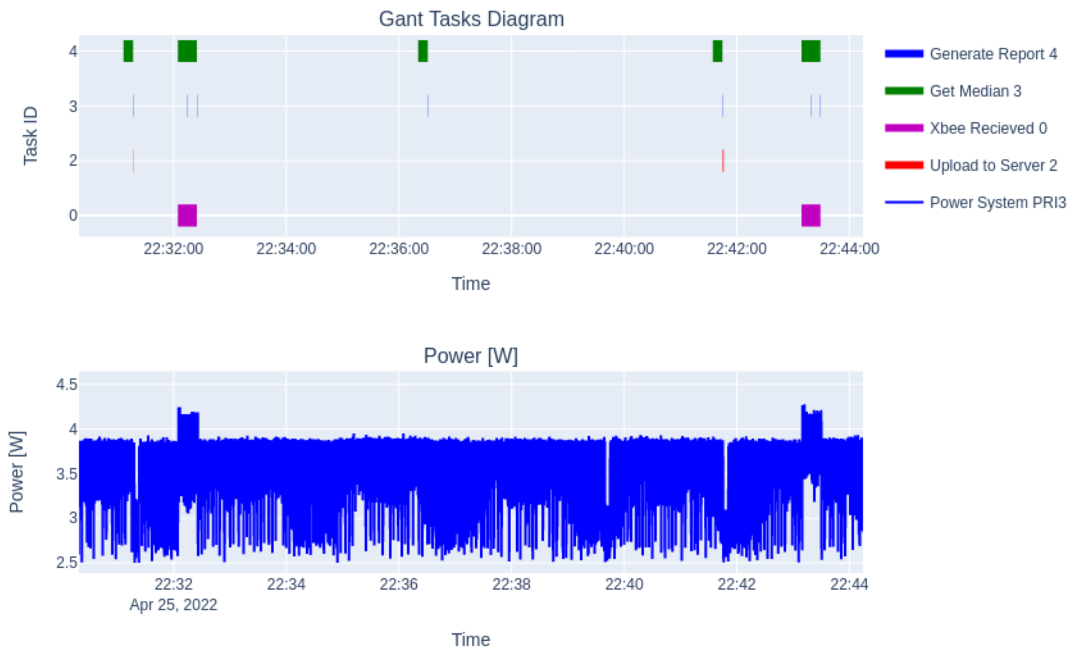


Figure 4.10: Tasks Gantt Plot vs Power (W) of eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz - I. Details of the test in table 4.1

Table 4.5: Comparison of $\vec{E}_i^f$ with f=1.4 GHz for RPI3 B+, and $\vec{E}_i^f$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2019 code as master/primary in eBridge 2021 HW configuration. Test details in tables 4.1 and 4.2.

	Power [W]			
Task ID	τ_4	τ_3	τ_2	τ_0
Code running on RPI3 B+				
n	3 049 705	1 823 178	1 042 711	1 250 461
Mean	3.7892	3.8020	3.8023	3.7431
Std	0.2688	0.2512	0.3820	0.25610
Min	2.503	2.541	2.510	2.521
Max	4.4834	4.5405	4.1800	4.5402
Code running on RPI4 B				
n	3 046 161	3 000 765	729 017	424 057
Mean	4.5737	4.5857	4.5863	4.5935
Std	0.0791	0.0619	0.0622	0.0565
Min	3.7586	4.0994	4.1133	4.1494
Max	5.8934	5.2218	5.1678	5.1830

Table 4.6: Power Comparison (W) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW in both RPI3 B+ and RPI4 B. Test details in tables 4.1 and 4.2

	Power [W]	
Node	**RPI3 B+**	**RPI4 B**
n	8 652 00	9 570 000
Mean	3.7897	4.5788
Std	0.2719	0.0698
Min	2.587	3.5161
Max	4.5405	5.8934

slave/secondary to build the vector $\vec{C_i^f}$. The details of these test can be seen in tables 4.7 and 4.8. These experiments had the same type of configuration as the experiments run in section 4.1.2.

The CPU Load (%) of a slave/secondary node running in an RPI3 B+ system can be seen in figure 4.13, while the one of a RPI4 B is shown in figure 4.16. In table 4.9 there is a comparison of the statistical data of the CPU Load (%) for both types of nodes. It is possible to see how, on an RPI3 B+ node, the load is distributed among Core_2 and

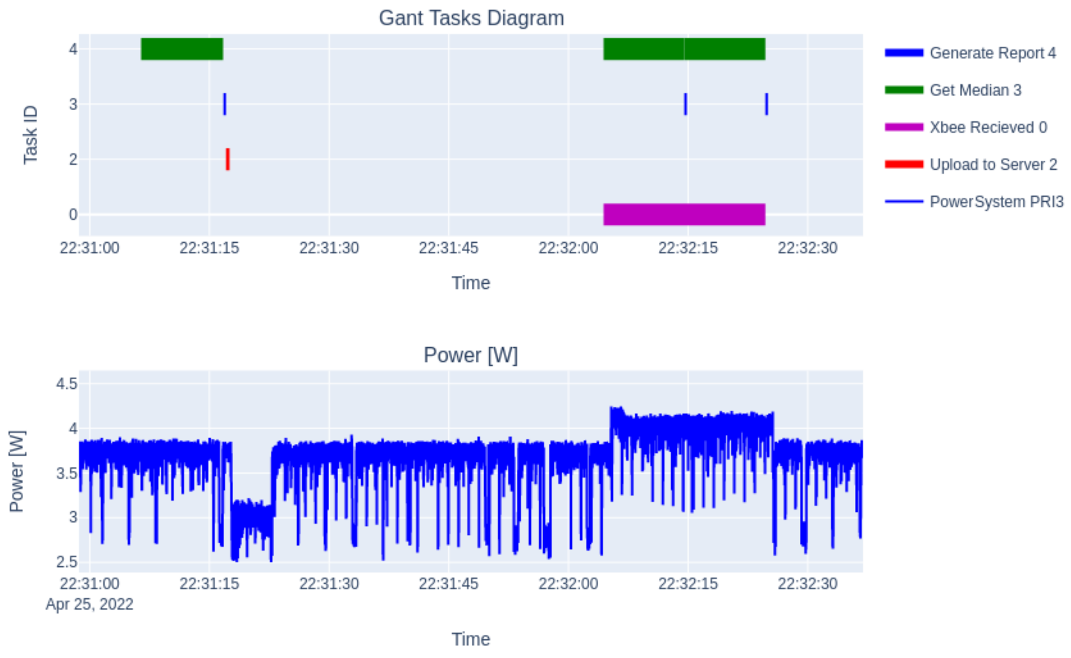


Figure 4.11: Tasks Gantt Plot vs Power (W) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz - II. Details of the test in table 4.1

Core_3; on the RPI4 B system, the mean load value of Core_2 is 99.50 %.

In figure 4.14 there is a zoomed view of the Gantt Plot of a the slave/secondary node running the eBridge 2019 code in the eBridge 2021 HW configuration in a RPI3 B+ unit, the test details can be seen in table 4.7. In this figure, it is possible to see the flow of execution of tasks $\Gamma = \{\tau_1, \tau_3, \tau_4\}$, in which the data is acquired, then the report is generated, and finally it is sent through the XBee communication board to the master/primary system. There are five instances of the **XBee Send ID:1** τ_1 task in the figure4.14, due to the unit configuration of doing a measurement per minute and then, after five minutes sending the data to the master/primary node as detailed in table 4.7.

Figure 4.15 shows how whenever the **Get Median ID:4** τ_4 task execution is followed by an execution of task **Generate Report ID:3** τ_3 or **XBee Send ID:1** τ_1, a part of the CPU load migrates briefly from one core to the other causing a reduction on one of the cores and an increase in another. In figure 4.13, it is possible to see how this behavior is repeated through the code execution and how it causes the full load in some cases to change from one core to another. The same behavior can be observed in the RPI4 B system in figure 4.16, where in an RPI3 B+ node, the CPU load is distributed in multiple cores, while in an RPI4 B unit, it falls in a single node.

In table 4.10 it is possible to see how the tasks **XBee Send ID: 1**(τ_1) and **Get Media ID: 4**(τ_4) have a higher execution time in a RPI3 B+ system than a RPI4 one. In the

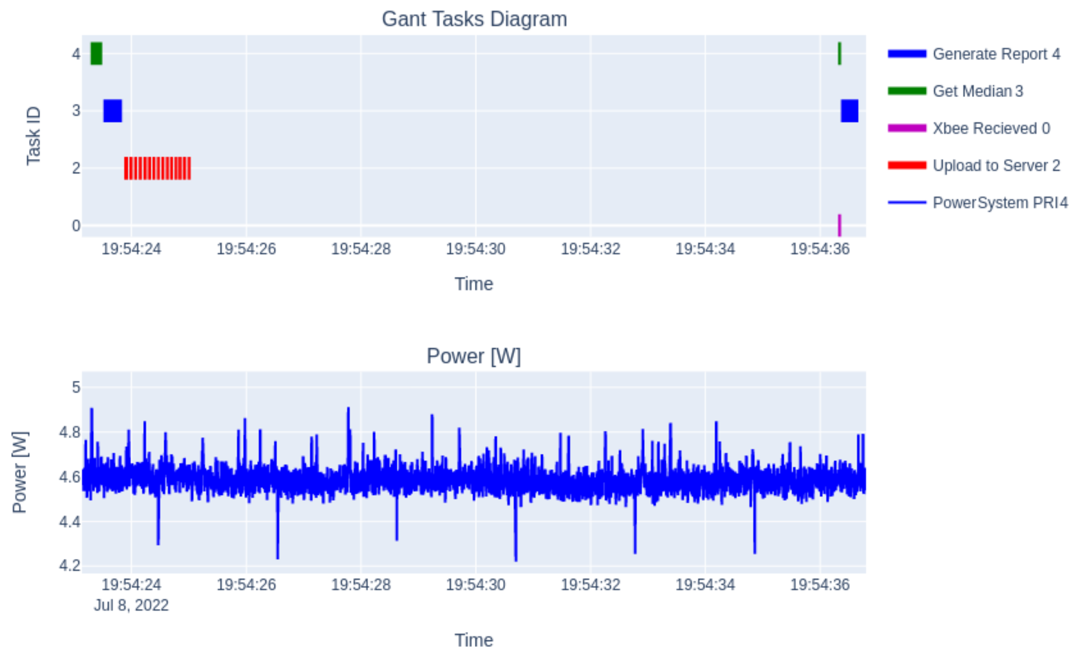


Figure 4.12: Tasks Gantt Plot vs Power (W) eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2

Table 4.7: Measurement Summary eBridge 2019 code running as Slave/Secondary Sensor Node in eBridge 2021 hardware configuration on RPI4 B+.

Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI3 B+
Controller	Raspberry PI3 B+
Device ID	RPI3-2
Code Running	eBridge 2019 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Slave/Secondary
Total Test Duration	1 hour 29 minutes and 24 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples for CPU	5364 samples
CPU Frequency applied to all cores(f)	1.4 GHz
Power Measurement Summary	
Power Source	CanaKit RPI3 B+ 5V 2.5A [59]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	10 728 000

case of (τ_1), the mean execution time on an RPI3 B+ is 0.6670 s compared to the one in the RPI4 B node of 0.2219 s, which is decreased by 66.73 %. For the case of τ_3 and τ_1 the execution times, there is no noticeable change. As it is mentioned 4.1.1, the **Get Median ID:4** τ_4 is the function in charge of taking 10 consecutive measurements and then performing an average calculation, which requires higher processing power in comparison to the other tasks of interested.

A similar trend can be seen in table 4.11 in which the mean power consumption of a slave/secondary node is 21.52% higher in an RPI4 B node with a value of 4.8200 W, while in an RPI3 B+ node is 3.9663 W. With a more in-depth analysis by obtaining the vector $\vec{E}_i^f$ with f = 1.5 GHz statistical information in table 4.12 it is possible to observe how the mean power for every of the tasks $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ is 21.55 %, 21.27 %, and 22.43 % higher respectively in a RPI4 B node than in a RPI3 B+.

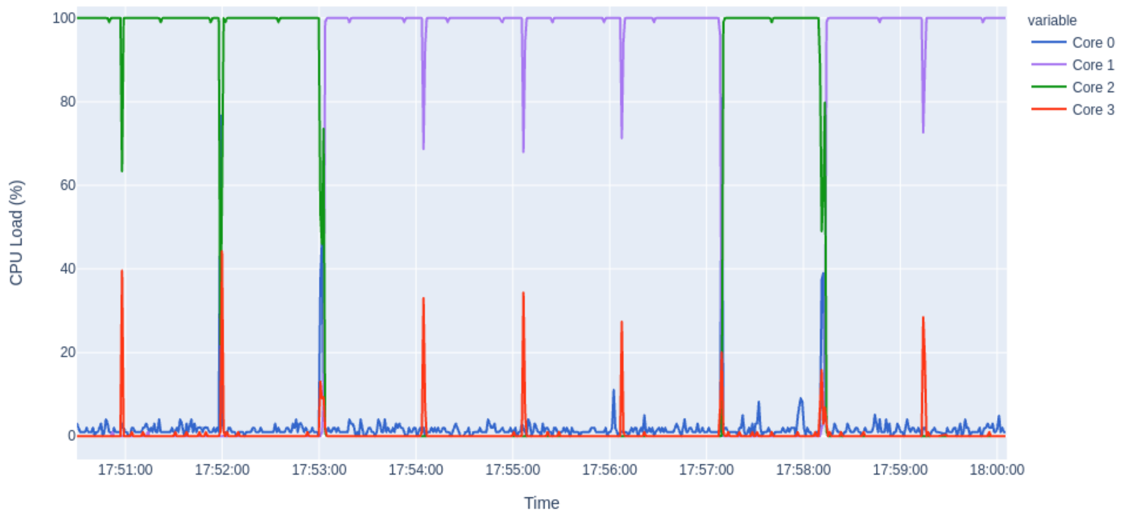


Figure 4.13: CPU Load (%) eBridge 2019 code running as slave/secondary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.7

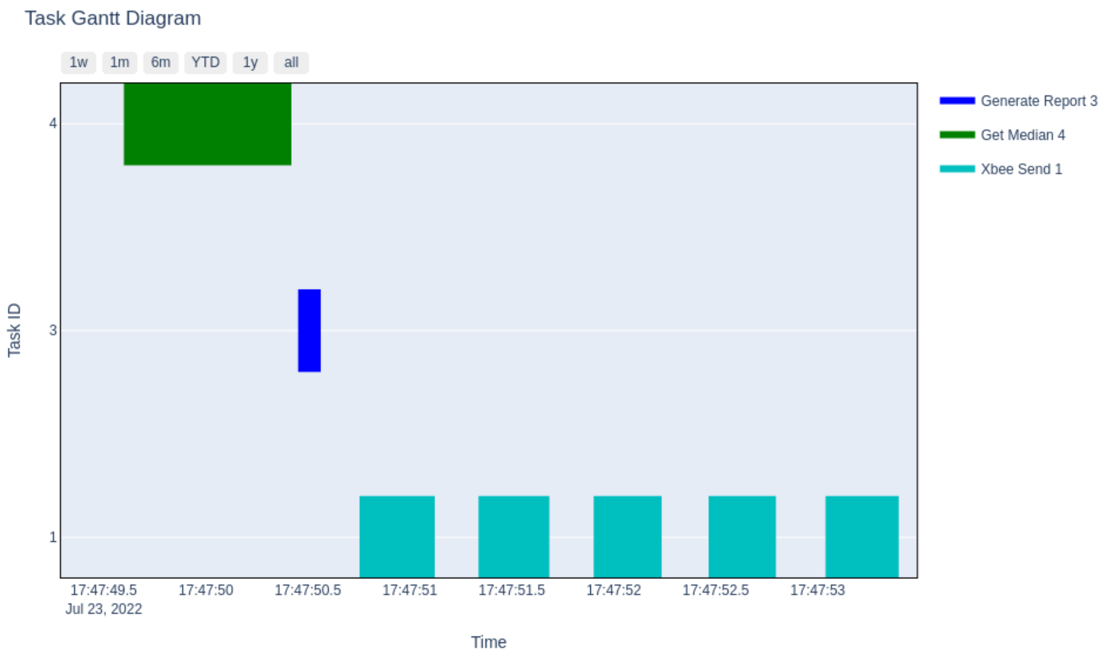


Figure 4.14: Tasks Gantt Plot eBridge 2019 code running as slave/secondary node running in eBridge 2021 HW in RPI3 B+ with f = 1.4 GHz. Details of the test in table 4.7

When comparing the data in tables 4.6 and 4.11, it can be observed how for the eBridge 2019 nodes the mean power consumption is 20.82 % higher for a master/primary node in a eBridge 2021 HW configuration in RPI4 B node versus a RPI3 B+, while for a slave/secondary unit is 21.52 % higher as well in RPI B versus RPI3 B+. This can be

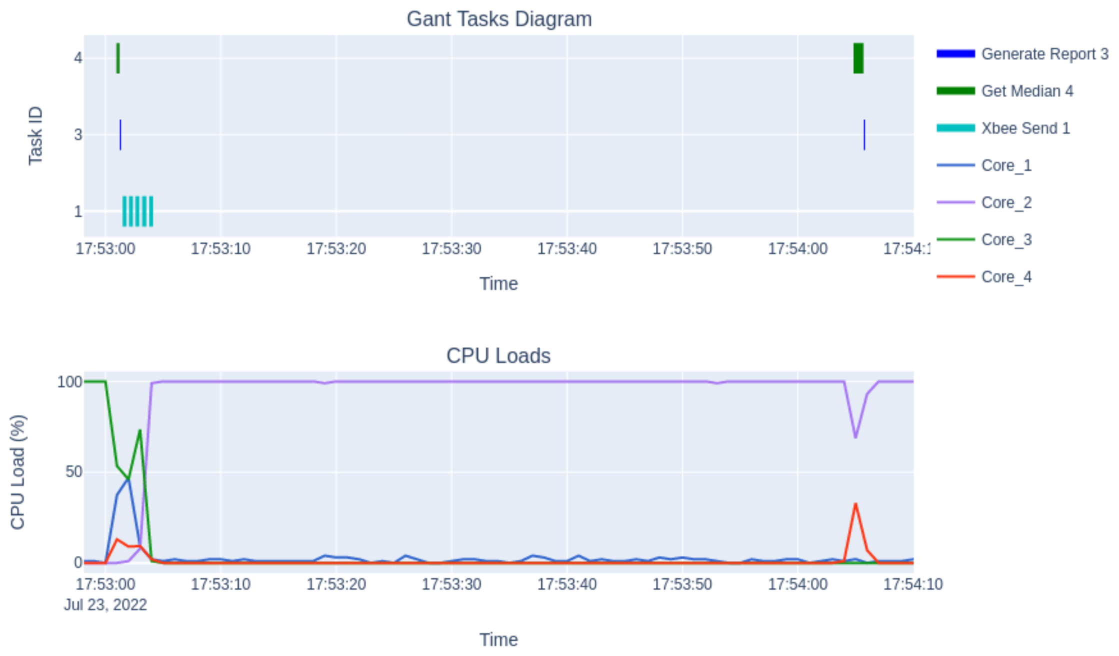


Figure 4.15: Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as slave/secondary node running in eBridge 2021 HW in RPI3 B+ with f = 1.5 GHz. Details of the test in table 4.7

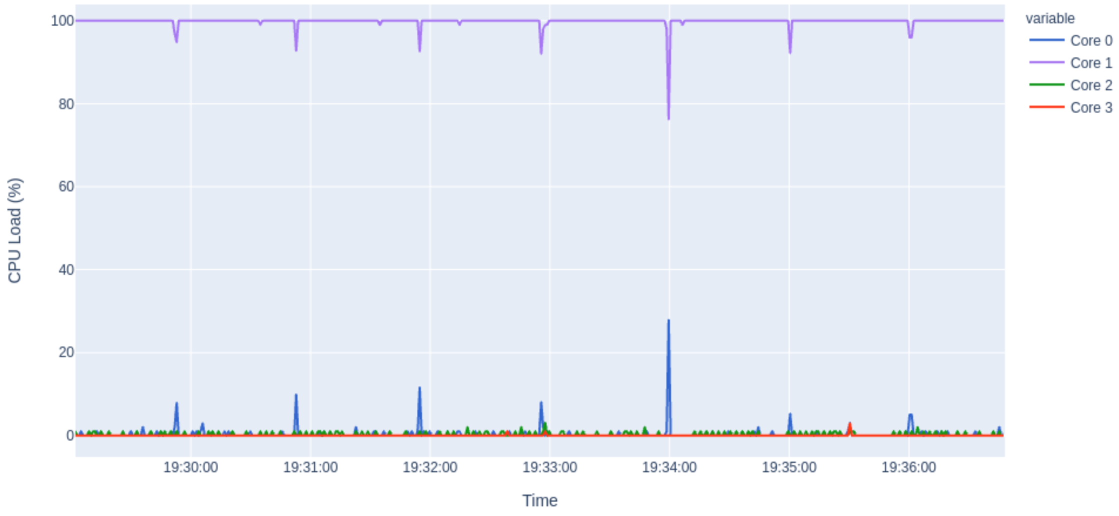


Figure 4.16: CPU Load for eBridge 2019 code running as a slave/secondary node running in eBridge 2021 HW in RPI4 B with f = 1.5 GHz. Details of the test in table 4.8

contrast with the results seeing in tables 4.5 and 4.12, specifically comparing the average power consumption for tasks **XBee Received ID: 0** (τ_0), a **XBee Send ID: 1** (τ_1)

Table 4.8: Measurement Summary eBridge 2019 code running as Slave/Secondary Sensor Node in eBridge 2021 hardware configuration on RPI4 B.

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI4 B
Controller	Raspberry PI 4 B
Device ID	RPI4-2
Code Running	eBridge 2019 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Slave/Secondary
Total Test Duration	1 hour 02 minutes and 03 seconds
System Configuration	Message Frequency: 10 min Measurement Frequency: 5 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples for Plot	3713 samples
CPU Frequency applied to all cores(f)	1.5 GHz
Power Measurement Summary	
Power Source	CanaKit RPI4 B 5V 3.5A [60]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	7 426 000

Table 4.9: Comparison of Code CPU Load (%) statistical information of the eBridge 2019 code running as a slave/secondary node running in eBridge 2021 HW in RPI3 B+ and RPI4 B

CPU Load (%)				
Core	**1**	**2**	**3**	**4**
Code running on RPI3 B+				
Number of samples per core = 5364				
Mean	5.23	67.41	19.78	8.92
Std	18.77	46.23	39.63	27.30
Code running on RPI4 B				
Number of samples per core = 3713				
Mean	0.55	99.50	0.28	0.01
Std	4.64	5.55	1.41	0.18

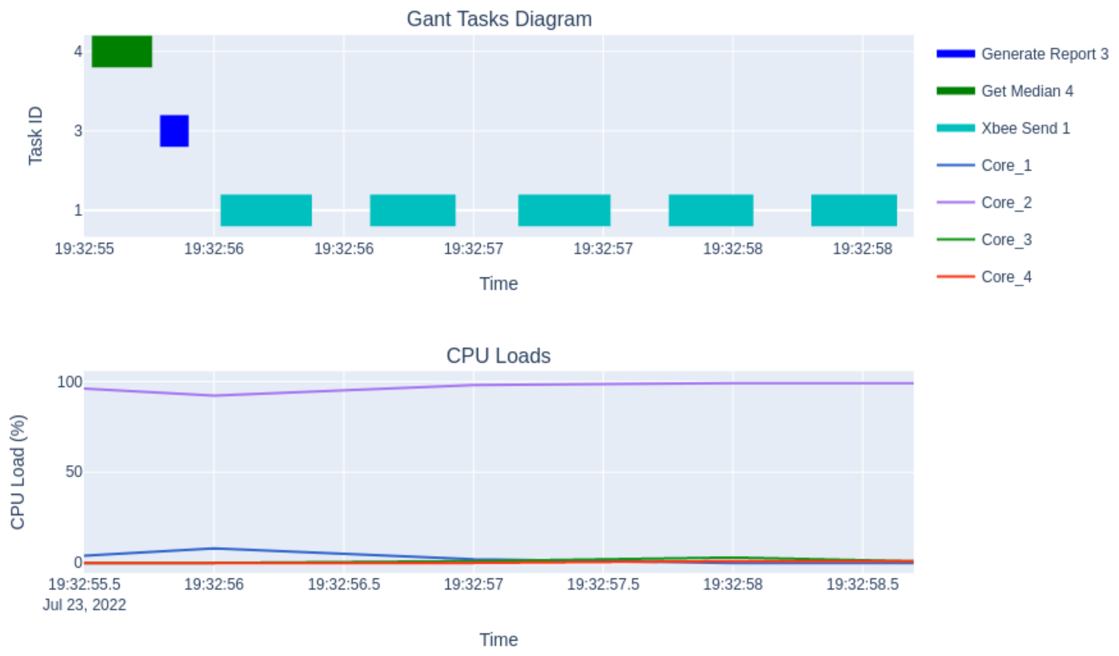


Figure 4.17: Tasks Gantt Plot vs CPU Load (%) eBridge 2019 code running as a slave/secondary node running in eBridge 2021 HW in RPI4 B with f = 1.5 GHz. Details of the test in table 4.8

Table 4.10: Comparison of $\vec{C_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{C_i^f}$ with f=1.5 GHz for RPI4 B vectors statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2019 in eBridge 2021 HW configuration as slave/secondary. Test details in tables 4.7 and 4.8

Execution Time (s)			
Task ID	τ_4	τ_3	τ_1
Running on RPI3 B+ node			
n	61	60	60
Mean	0.6670	0.0109	0.3479
Std	0.2735	0.0021	0.0191
Min	0.1914	0.0008	0.3086
Max	0.8560	0.1281	0.4082
Running on RPI4 B node			
n	88	87	85
Mean	0.2219	0.0103	0.3395
Std	0.0070	0.0007	0.0100
Min	0.2084	0.0053	0.3197
Max	0.2341	0.0105	0.3783

Table 4.11: Power Comparison (W) eBridge 2019 Code RPI3 B+ vs RPI4 B running as Slave/Secondary in eBridge 2021 HW. Test details in tables 4.7 and 4.8

Power [W]		
Node	**RPI3 B+**	**RPI4 B**
n	10 728 000	7 426 000
Mean	3.9663	4.8200
Std	0.1323	0.0999
Min	2.5023	3.7104
Max	4.6704	7.4968

Table 4.12: Comparison of $\vec{E_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{E_i^f}$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2019 code as slave/secondary in eBridge 2021 HW configuration. Test details in tables 4.7 and 4.8

Power [W]			
Task ID	τ_4	τ_3	τ_1
Code running on RPI3 B+			
n	3 991 629	3 361 668	860 160
Mean	3.9619	3.9739	3.9568
Std	0.1426	0.1279	0.1157
Min	2.5023	2.6945	2.6856
Max	4.3579	4.6907	4.2341
Code running on RPI4 B			
n	3 585 176	3 115 916	543 681
Mean	4.8156	4.8193	4.8442
Std	0.1114	0.0898	0.0744
Min	3.7104	4.0856	4.6535
Max	7.4968	7.2735	5.6917

in which in average for an RPI3 B+ node sending a message is 5.71 % higher than receiving a message, and for a RPI4 B node is 5.46 % higher. This complies with the XBee specifications [14], which states that the transmit current is 33 mA compared to the idle current of 28 mA.

The tables 4.6 and 4.11 make it possible to see that the highest power peak on both RPI3 B+ and RPI4 B nodes running as both master/primary in the eBridge 2019 HW configuration happens during the execution of the **Get Median ID:4** τ_4 task, with a value of 4.4834 W for RPI 3 B+, and 5.8934 W for RPI4 B. In the case of slave/secondary, the peak value for RPI3 B+ is 4.3579 W, and for an RPI4 B, the node is 7.4968 W, which corresponds with the data in table 4.11. In figure 4.18, it can be regarded that there are no noticeable power spikes during any of the execution of the tasks.

In the next section 4.2 a study of the battery performance of the eBridge 2019 code is shown.

4.2 eBridge 2019 Battery Tests

To evaluate the initial autonomy of the eBridge 2019 network, several battery tests are done in both RPI3 B+ and RPI4 B nodes with the 5 200 mAh and 10 000 mAh batteries mentioned in the section 3.5.1. These results can be seen in tables 4.13, and 4.14. Each configuration was tested twice using both types of batteries, 5 200 mAh, and 10 000 mAh.

4.2.1 Battery test for Master/Primary

For the case of master/primary nodes it can be seen how the total run time data corresponds with the overall power consumption data in table 4.6, mentioned in the previous section 4.1.2 is 20.82 % higher in a RPI4B vs a RPI3 B+ node. The total run times with the 5 200 mAh batteries for the RPI3B + nodes as master/primary are 2 hours, 15 minutes, 33 seconds, and 2 hours, 18 minutes, and 45 seconds. These values are 20.08 % and 20.7 % higher in comparison to the ones for the RPI4 B node. For the case of the

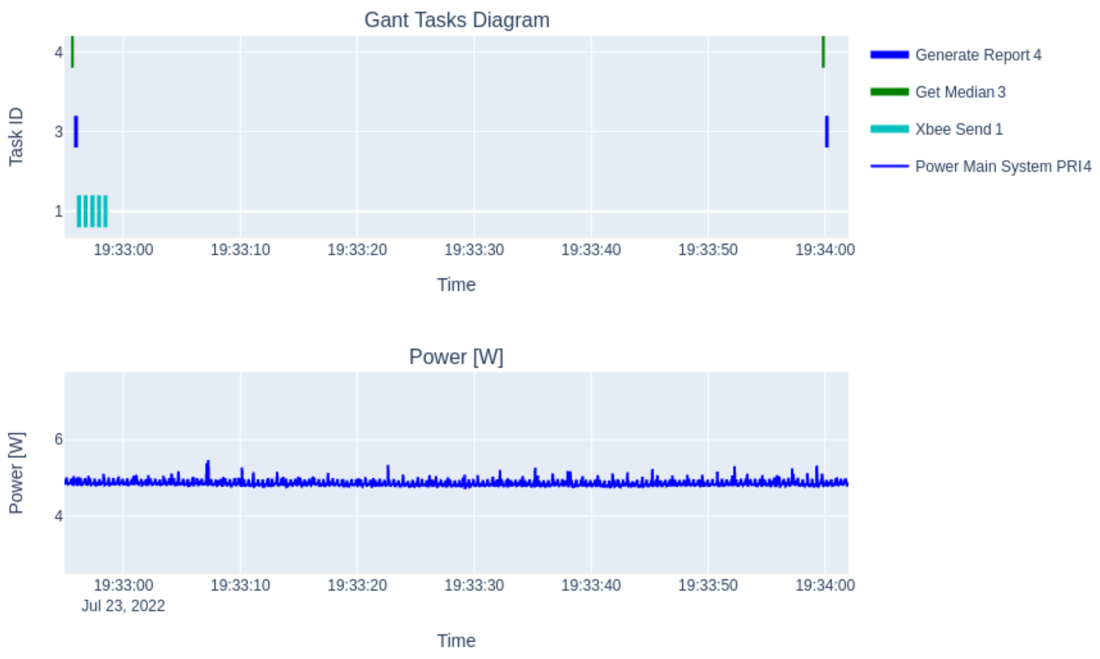


Figure 4.18: Tasks Gantt Plot vs Power (W) eBridge 2019 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.8

10 000 mAh, there is a similar behavior, in which the run times are 3 hours, 31 minutes, and 12 seconds; and 3 hours, 28 minutes, and 31 seconds; which are 21.08 %, and 22.19 % higher in comparison to the RPI4 B.

Table 4.13: Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2019 code as master/primary in eBridge 2021 HW configuration.

Type of Configuration:	Measurement Frequency = 1 min Message Frequency = 5 min	
Device ID	**Battery ID**	**Total Run Time [HH:MM:SS]**
RPI3-2	5 200 mAh - 1	02:15:33
RPI3-2	5 200 mAh -2	02:18:45
RPI3-2	10 000 mAh -1	03:31:12
RPI3-2	10 000 mAh -2	03:28:31
RPI4-2	5 200 mAh - 3	01:48:36
RPI4-2	5 200 mAh - 4	01:50:02
RPI4-2	10 000 mAh -3	03:21:28
RPI4-2	10 000 mAh -4	03:23:12

4.2.2 Battery test for Slave/Secondary

For the case of the slave/secondary node, the total power is 21.52 % higher on an RPI4 B node vs an RPI3 B+. The same as with the master/primary node, the time total run times with the 5200 mAh batteries are 17.75 % and 14.04 % higher for the RPI3 B+ respectively, compared to the run times for the RPI4 B node. These results are compatible with the battery tests performed by [4], which had a total run time of two hours for master/primary and slave/secondary nodes with the eBridge 2019 HW configuration with the ICR18650 4400mAh 3.7V battery for both master/primary and slave/secondary nodes. The next section 4.3 covers an analysis of the $\Gamma = \{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$ tasks interactions. A good understanding of the relationship between these tasks for both the master/primary and slave/secondary nodes is crucial to finding areas of improvement for resource utilization and power consumption. Also, it allows a better understanding of the interaction between nodes; for example, it helps understand slave/secondary nodes in master/primary ones.

4.3 eBridge 2019 Task Interaction Analysis

This sections covers an analysis of the $\Gamma = \{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$ tasks interactions. In figure 4.19 there is a qualitative representation of the interactions of the tasks for a master/primary node, which is the more complex one. As it is possible to see in the qualitative diagram, there is a possibility of having tasks executed simultaneously. For example, the stalling behavior that could be see in figures 4.4, 4.5, 4.10, and 4.11, it is caused due to the **RequestsQueue**, which as it is mention in section 4.1.1 it is use by several functions and threads, this queue is used by the **Device** main thread to store reports generated every time there is a measurement, it is also used by receive thread whenever the unit receives XBee messages. Even though it looks complex, the diagram in the figure 4.19 could happen in the master/primary node with only one slave/secondary unit connected to it.

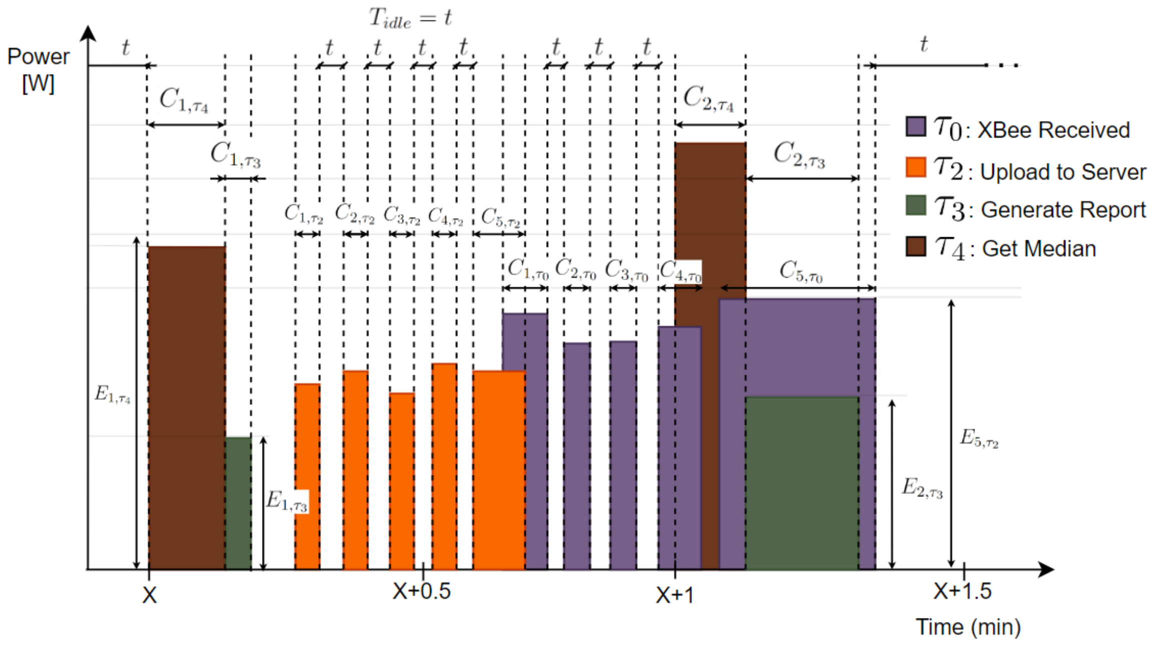


Figure 4.19: Tasks $\Gamma = \{\tau_0, \tau_2, \tau_3, \tau_4\}$ $C^{\vec{f}}_{i,\tau_i}$, and $E^{\vec{f}}_{i,\tau_i}$ Interaction Qualitative Diagram for a master/primary Node.

In figure 4.20 there is a zoomed view of the Gantt diagram in figure 4.8 where it can be observe how the master/primary node uploads 14 messages to the eBridge Server. This behavior was seen on the bridge where both the master/primary and slave/secondary nodes had a measurement frequency of one minute and a message frequency of five minutes. Depending on the number of nodes per bridge, this could significantly increase the resource utilization in a master/primary node, increasing both CPU and power consumption and thus reducing battery life. In a scenario of a bridge with one master/primary node and four slave/secondary nodes, a measurement frequency of one minute and a message frequency of one hour, there could be up to 300 messages per hour.

When building the $C^{\vec{f}}_{i,\tau_i}$ and $E^{\vec{f}}_{i,\tau_i}$ vectors for the tasks in figure 4.19 assuming a $F = 1.5$

GHz the scenario looks like this:

Tasks Execution Times:

- $C^F_{i,\tau 0} = (C_{1,\tau 0}, C_{2,\tau 0}, C_{3,\tau 0}, C_{4,\tau 0}, C_{5,\tau 0} \ldots, C_{n,\tau 0})$
- $C^F_{i,\tau 2} = (C_{1,\tau 2}, C_{2,\tau 2}, C_{3,\tau 2}, C_{4,\tau 2}, C_{5,\tau 2} \ldots, C_{n,\tau 2})$
- $C^F_{i,\tau 3} = (C_{1,\tau 3}, C_{2,\tau 3}, \ldots, C_{n,\tau 4})$
- $C^F_{i,\tau 4} = (C_{1,\tau 4}, C_{2,\tau 4}, \ldots, C_{n,\tau 4})$

Power Consumption:

- $E^F_{i,\tau 0} = (E_{1,\tau 0}, E_{2,\tau 0}, E_{3,\tau 0}, E_{4,\tau 0}, E_{5,\tau 0} \ldots, E_{n,\tau 0})$
- $E^F_{i,\tau 2} = (E_{1,\tau 2}, E_{2,\tau 2}, E_{3,\tau 2}, E_{4,\tau 2}, E_{5,\tau 2} \ldots, E_{n,\tau 2})$
- $E^F_{i,\tau 3} = (E_{1,\tau 3}, E_{2,\tau 3}, \ldots, C_{n,\tau 4})$
- $E^F_{i,\tau 4} = (E_{1,\tau 4}, E_{2,\tau 4}, \ldots, C_{n,\tau 4})$

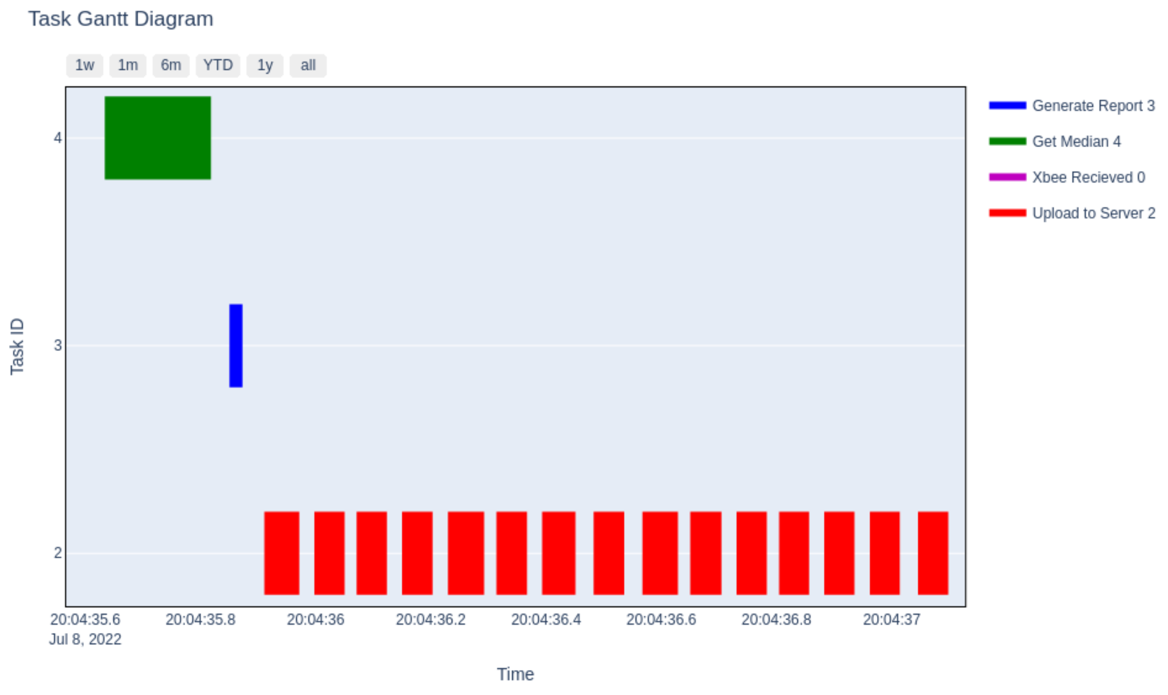


Figure 4.20: Zoomed Gantt Plot for eBridge 2019 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 4.2

One of the limitations of this approach is that in cases where the functions overlap, just like when the **stalling** behavior happens in figure 4.11. A qualitative representation of a power measurement of this type of interaction can be seen in figure 4.21, which is a zoomed area out of figure 4.19, close to the **x+1** minute. For a slave/secondary node, the scenario is simpler than the one from a master/primary node; in figure 4.22, it is possible to see how there is no task overlap.

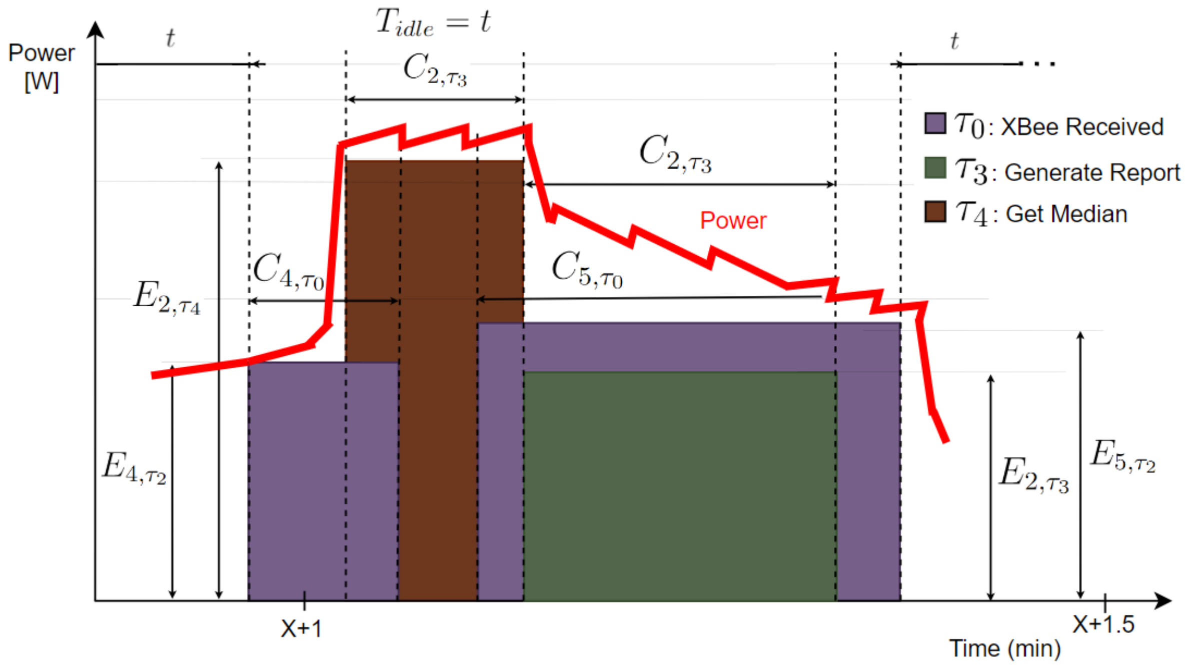


Figure 4.21: Zoomed Tasks $\Gamma = \{\tau_0, \tau_3, \tau_4\}$ $C^{\vec{f}}_{i,\tau_i}$, and $E^{\vec{f}}_{i,\tau_i}$ Interaction Qualitative Diagram for a master/primary Node with Highlighted Power

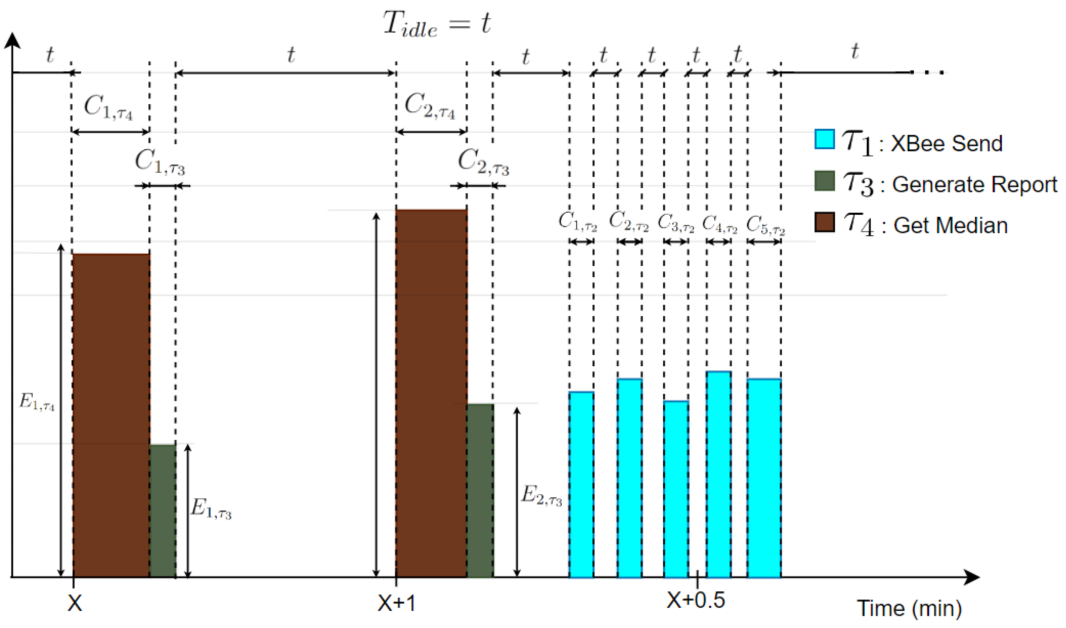


Figure 4.22: Zoomed Tasks $\Gamma = \{\tau_1, \tau_3, \tau_4\}$ $C^{\vec{f}}_{i,\tau_i}$, and $E^{\vec{f}}_{i,\tau_i}$ Interaction Qualitative Diagram for a slave/secondary Node

In the next section 4.4 there is a description of the eBridge 2021 HW configuration.

4.4 eBridge 2021 HW Configuration Standby Power Consumption

To have a baseline of the eBridge 2021 HW configuration's power consumption without executing code, the experiment described in table 4.15 was conducted. The results are plotted in figure 4.23, and the statistical summary of the test can be seen in table 4.16. When comparing this data with the one in table 4.6, it can be validated that the mean power consumption of the master/primary node in an RPI3 B+ node of 3.7897 W is 24.65 % higher than the standby power consumption of 3.0402 W. In comparison, for an RPI4 B node, the mean value of 4.5788 W is 49.53 % higher than the standby power of 3.0622 W.

It is possible to confirm that the slave/secondary nodes have similar behavior, in which for the RPI3 B+, the mean power consumption of 3.9663 W is 30.46 % higher than the standby consumption, and for the RPI4 B is 57.4 % higher as well with a value of 4.82 W. The standby power for both nodes is similar, with mean values around 3 W. In the next chapter 5, there is a description of the proposed optimizations for the eBridge network.

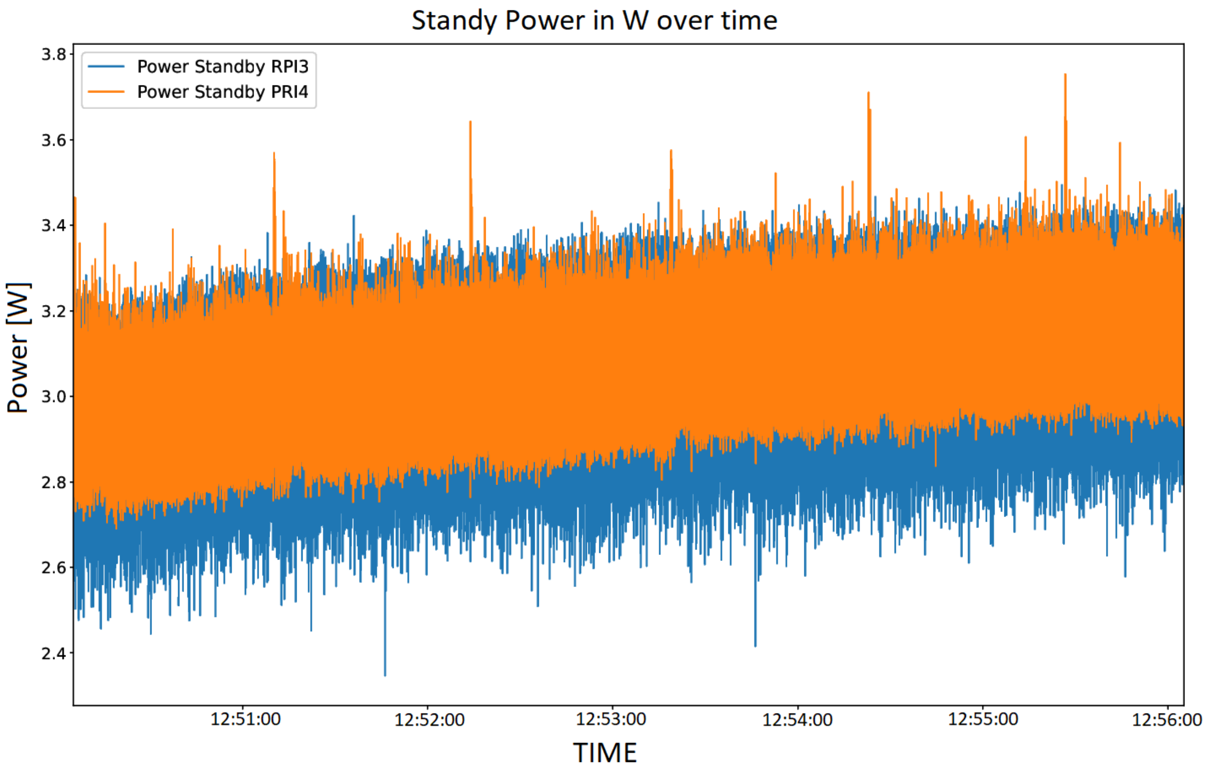


Figure 4.23: Standby Power [W] vs Time (s) for the eBridge 2021 HW configuration of RPI3 B+ and RPI3 B.

4.5 Summary of Areas of Improvement for the EBridge 2019 Network

This section includes a summary of the areas of improvement discovered for the eBridge 2019 Network. These areas fall under three categories:

1. **Software**: Some areas of improvement identified are around the system time tracking for the execution of critical tasks like taking measurements, sending data to master/primary devices, or uploading data to the server. Also, the eBridge 2019 code did not include any management of the cellular network interface because the communication with the APN network was only done once at the beginning of the code execution while having DPM techniques could improve power consumption. Finally, managing shared resources among the eBridge 2019 code is critical for preventing stalling behaviors, like the one caused by the **RequestQueue** FIFO.

2. **Protocol Upgrades**: Some areas of improvement for the eBridge network protocols are node configuration, message handling on slave/secondary nodes, alarm handling on both master/primary and slave/secondary nodes, and the overall number of messages being transmitted between nodes or upload to the server.

3. **Hardware Optimizations**: One of the hardware areas of improvement identified is the relatively high power consumption of the XBee communication board compared to other state-of-the-art communication technologies. Also, taking advantage of DPM and DVFS techniques could be an area of improvement.

In the next chapter 5, there is a description of the proposed optimizations for the eBridge network.

Table 4.14: Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2019 code as slave/secondary in eBridge 2021 HW configuration.

Type of Configuration:	Measurement Frequency = 1 min Message Frequency = 5 min	
Device ID	**Battery ID**	**Total Run Time [HH:MM:SS]**
RPI3-2	5 200 mAh - 1	01:54:33
RPI3-2	5 200 mAh -2	01:49:16
RPI3-2	10 000 mAh -3	03:09:35
RPI3-2	10 000 mAh -4	03:11:02
RPI4-2	5 200 mAh - 3	01:37:17
RPI4-2	5 200 mAh - 4	01:35:49
RPI4-2	10 000 mAh -1	02:54:26
RPI4-2	10 000 mAh -2	02:50:39

Table 4.15: eBridge 2021 HW Configuration in RPI3 B+ and RPI4 B Power Consumption in Standby

Hardware Configurations	
Hardware Configuration I	eBridge 2021 HW with RPI4 B
Hardware Configuration II	eBridge 2021 HW with RPI3 B+
Power Measurement Summary	
Power Source RPI4 B	CanaKit RPI4 B 5V 3.5A [60]
Power Source RPI3 B+	CanaKit RPI4 B+ 5V 2.5A [59]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	4
Total Test Duration	6 minutes
Number of Samples per Second (S/s)	20 000
Total number of samples per channel	7 200 000

Table 4.16: Power Comparison (W) eBridge 2021 HW Configuration on code RPI3 B+ vs RPI4 B. Test details in table 4.15.

Power [W]		
Node	**RPI3 B+**	**RPI4 B**
n	7 200 000	7 200 000
Mean	3.0402	3.0622
Std	0.0101	0.0092
Min	2.3472	2.9965
Max	3.4940	3.7532

Chapter 5

Design

This chapter covers the code optimizations over the 2019 version of the eBridge network to reduce CPU utilization and power consumption. As mentioned in section 2.6.2, software allocates up to 80 % of the power consumption of a system. In section 5.1 there is a detailed description of the software level optimizations done to the Python eBridge 2019 code, then there is a description of the communication protocol changes performed to the eBridge network to reduce power consumption in section 5.2, and finally, in 5.3 there would be details of hardware level optimizations proposed for the eBridge system.

5.1 Software Level Optimizations

This section explains the software changes implemented to improve the eBridge 2019 code, reduce CPU utilization, reduce power consumption, do shared resources arbitration, and improve code readability and maintainability.

5.1.1 SetDateTime Refactoring

As it was mentioned in section 4.1.1 there were three threads of interest in the code: **SetDateTime**, **Receive**, and **Device.Run**. As the first step, the code was refactored to remove the **SetDateTime** thread and change it for a function that allows the system to initialize the OS clock once the node has connected to the LTE network through the use of the Raspbian **timedatectl** utility to communicate with the Linux **Systemd** Service manager to update time and date settings. Since neither the RPI 3B+ nor the RPI4 B does not have an RTC (Real Time Clock), it is important to update system time periodically. It was not possible to find information on how much the RPI3 B+ or RPI4 B clock is expected to drift, and it is part of the future work recommendations for the eBridge Network.

5.1.2 Cellular Network Interface Handling

Implementation of a cellular interface management class named **Cell_Connection**. This class allows the eBridge 2021 code to instantiate an object to control the cellular interface; it includes the following capabilities:

- **Start a Cell Connection:** This method allows the eBridge 2021 node to use the **qmicli** [62] library that allows communication with devices that use the Qualcomm MSM Interface (QMI) protocol, which is used by the LTE cellular interface board from WaveShare. This function performs several operations:
  1. Starts the **wwan0** network interface on the RPI node using **raw_ip** protocol.
  2. Brings the Wave-Share LTE board to online mode. By default, this board is initialized as Low-Power when powered on.
  3. Connects to the APN network, in the case of the eBridge Network, it is connecting to the Kolbi3g network taking advantage of the *Mobile Devices Geolocalization cellular plans* [63].
  4. Establish a DHCP (Dynamic Host Configuration Protocol) client connection to the Kolbi3g APN through udhcpc, a small open source program recommended for embedded systems running embedded Linux [64].
- **Stop a Cell Connection:** Brings the cellular interface board to low-power mode and turns off the wwan0 Linux network interface. As described in section 5.3, there are several aspects to the power consumption of the cellular interface, which allow saving both the standby power consumption and the RPI OS-related consumption to keep the wwan0 interface alive. Bringing the wwan0 interface down also brings down the whole USB controller.
- **Cell to Low Power:** Allows the to put the cellular interface board into low power mode.
- **Get Cell Status:** Gets the cellular connection status of the Kolbi3g APN.
- **Get IPv4 IP address:** This method allows the system to identify its IPv4 addresses, and it lets the user know if there is a wwan0 network on.

5.1.3 Receive Thread Optimizations

The receive thread is the one that allows a master/primary node to receive messages from the slave/secondary node through the use of the XBee S2C communication board [14]. The thread in the eBridge 2019 code had a high polling rate of up to 0.5 seconds to check for newly received messages. This is unnecessary since the device can buffer incoming messages from several devices; the library already includes a software interruption that allows the communication node to store messages in memory while the program reads them. A class named **XBeeNode** is created to manage the XBee communication. This class includes the following capabilities:

- **Find XBee Device:** This method allows the eBridge node to automatically detect the XBee S2C communication board.
- **Get XBee MAC Address:** This method allows the eBridge node to get the XBee S2C communication board MAC address. This feature was not present in the eBridge 2019 network and caused major scalability issues since the user had to configure it from the eBridge server manually.
- **XBee Start:** This method allows the eBridge node to start the XBee S2C unit as either master/primary or secondary/slave. It allows the node to acquire an *alias* or *node id* that facilitates communication. This alias could be the same *eBridge Node ID*.
- **XBee Send:** Allows to send data to another node specified by the user. This method uses the **get network** method.
- **Get Network**: This method is used by the XBee Send method to find the alias of the receiver node. An error is returned to the user if the alias is not found.
- **XBee Receive:** Allows the node to receive data from other nodes.

5.1.4 Device Class Optimizations

As it is mentioned in section 4.1.1, the **Device.Run** class is the main part of the program of the eBridge node. As part of the optimizations, several parts of the code were refactored to contain both the **receive** and **run** threads to be part of the class. Additionally, each sensor on the system would be part of the class named **Sensor**, of which a class named **Ultrasonic Sensor** was generated. A diagram of the new software hierarchy can be seen in figure 5.1. Also, it can also be observed how the Cell Interface class is instantiated inside the Device class.

Properly managing shared resources is key to ensuring race conditions like the ones causing **stalling** behavior described in section 4.1.2. Even though queues are usually used to communicate asynchronous processes, they include locks to block completing threads temporarily [65]. To solve this issue it was decided to use the **put_nowait()** and **get_nowait()** methods to prevent the locking behavior. However, it is important to ensure the code includes error handling to handle *Timeout* exceptions and it is recommended to make use of the **queue.empty()** and **queue.qsize()** methods to monitor queue status.

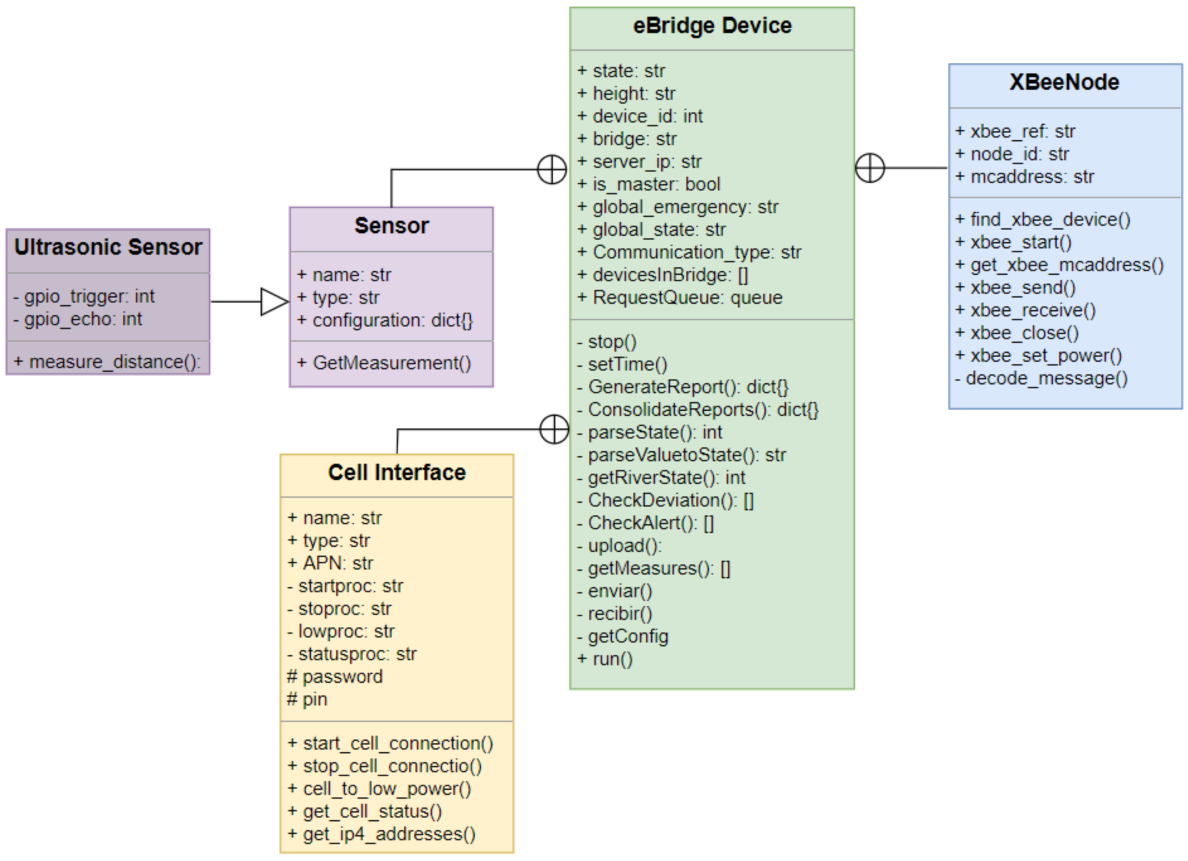


Figure 5.1: eBridge Embedded Software Now UML Classes Hierarchy

5.2 eBridge Protocol Upgrades

Based on the results in section 4.3, it was decided to perform modifications to the eBridge protocol proposed by the end of the work done by [4]. Four protocol changes are presented for the eBridge 2021 sensor node:

5.2.1 Device Configuration

Several changes have been made to the device configuration procedures of the eBridge 2019 code.

Nodes are configurable through the use of the .ini configuration file. This includes information about the node's desired ID, sensors to be configured in their respective I/O lines, server IP address, and service port for Restful communication. This allows the node to be quickly commissioned.

Depending on the node type of configuration, the unit could decide to turn on or off different hardware peripherals to save power. For example, in the case of a master/primary node, the cellular connection needs to be active to send emergency messages as soon as possible, so the cellular interface is set to low power. This reduces power consumption without renegotiating an IP address with the APN network. Cellular communication retrieves the device configuration for a slave/secondary node. In case the communication with the master/primary node is lost, both the cellular board and the OS wwan0 network interface are turned off to save power. This can be seen in figure 5.2.

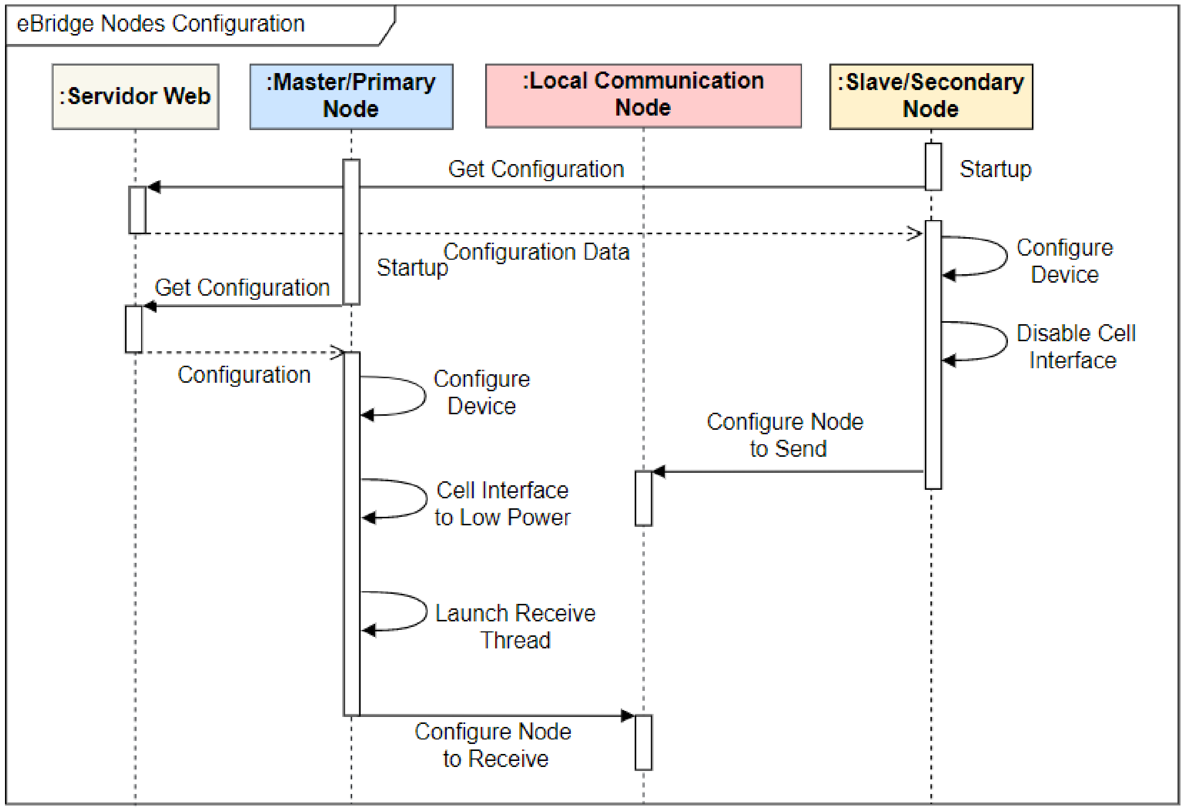


Figure 5.2: eBridge Node Configuration Policy.

5.2.2 Slave/Secondary Node Message Handling

As part of the improvement for the slave/secondary nodes, now the node is capable of detecting if there is a failure in the communication with the master/primary system; in case this happens, the node is capable of retrying up to ten times to send a message, and in case it still is unable to contact it the system would turn on its USB interface, bring up the *wwano* interface and bring up the cellular communication as an alternative method of ensuring a message reaches the server. Finally, when the message has been successfully received by the server, the cellular board and USB port are powered down again. This could be seen in figure 5.3.

5.2.3 Alarm Handling

Depending on the message frequency configuration on both primary and secondary nodes, sending an urgent message to the server could take a long time. For example, if a sec-

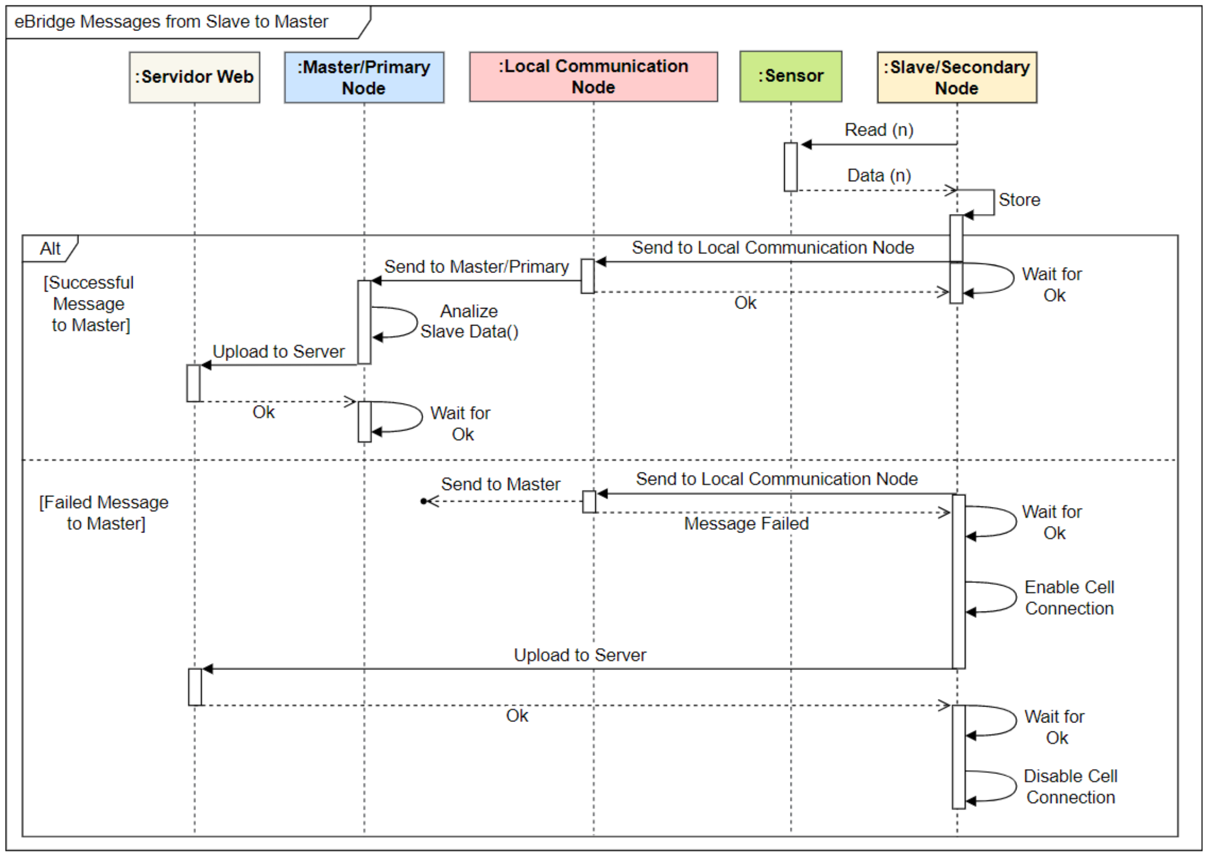


Figure 5.3: eBridge Messages from Slave/Secondary to Master Failure.

ondary node has a measurement frequency of one minute and a message frequency of 60 minutes, and a yellow level alarm is triggered at minute 37, the message would be sent up until the device reaches 60 minutes of message time. The policy was changed, and in cases, there is any type of alarm **green**, **yellow**, or **red**, the message would be sent as soon as possible from a slave/secondary node to the master/primary one. If there is no communication with the master/primary node, the procedure described in the bullet named **Slave/Secondary Node Message Handling** is used. The new paradigm can be seen in figure 5.4; if there is no alarm, the node stores all the messages in the **RequestQueue** to later be sent at the configured time.

5.2.4 Consolidating Reports

As mentioned in section 4.1.3, sending a message is 5.71 % higher than receiving a message for RPI3 B+, and for an RPI4 B node is 5.46 % higher. This is key to reducing the number of reports sent from slave/secondary nodes to master/primary ones and the reports sent to the eBridge Server. A new function named **Consolidate_Reports** is part of the eBridge Device class, as shown in figure 5.1, which is in charge of consolidating reports at the local node to send them either to the master/primary node or the server. Due to the new alarm handling policy described above, the reports in the **RequestQueue** have no alarm

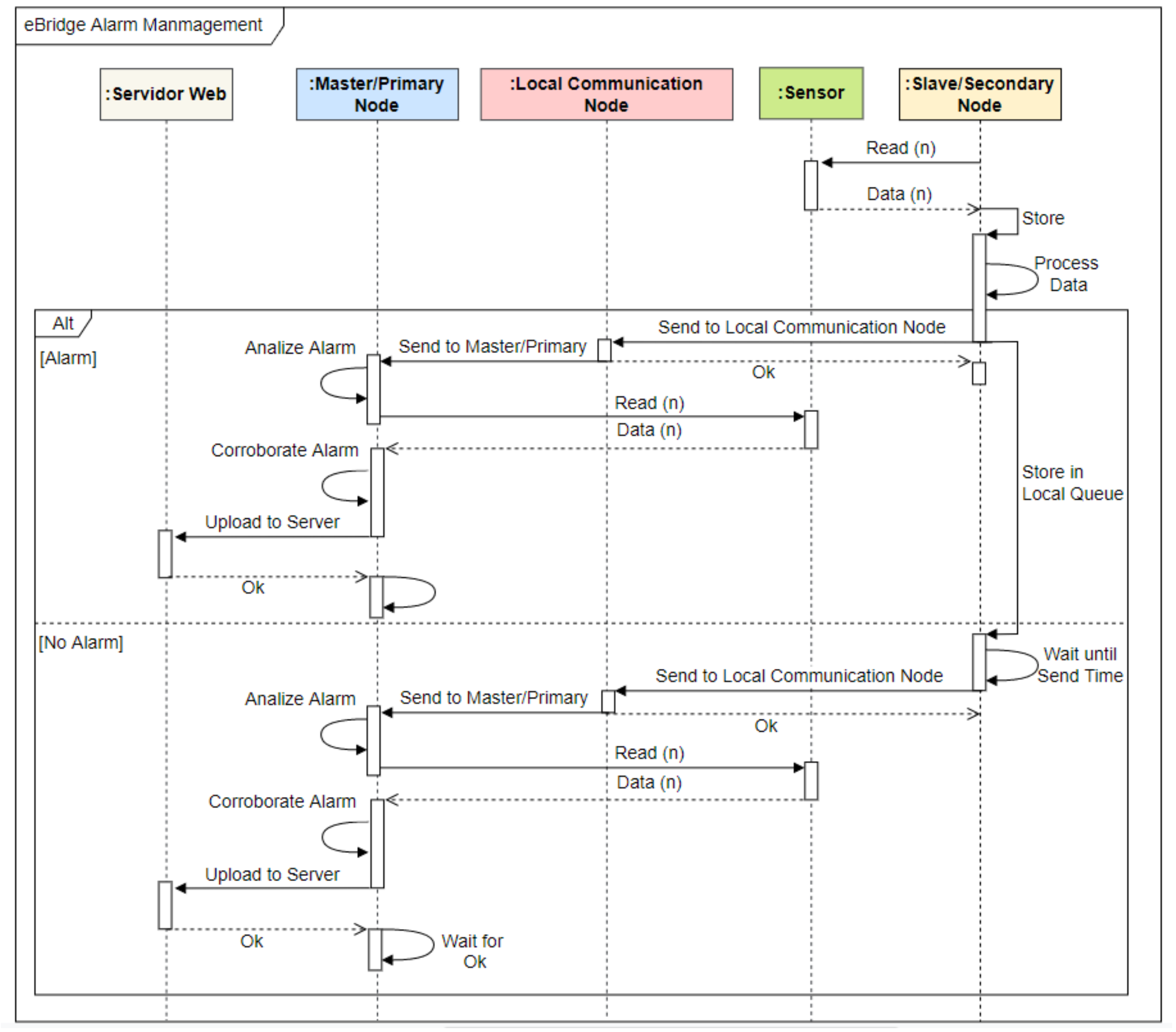


Figure 5.4: eBridge Alarm Handle Procedure Optimization.

level, and the process can be seen in figure 5.5. From the perspective of the secondary node, the procedure generates a single report with the average measurement value of all the sensor reads; when this procedure is executed in a master/primary node, the function consolidates reports by bridge and by the device. A summary of this can be seen in figure 5.6.

5.3 eBridge Hardware optimizations for power saving

In this section, there is a discussion of the hardware optimizations focused on the reduction of power consumption.

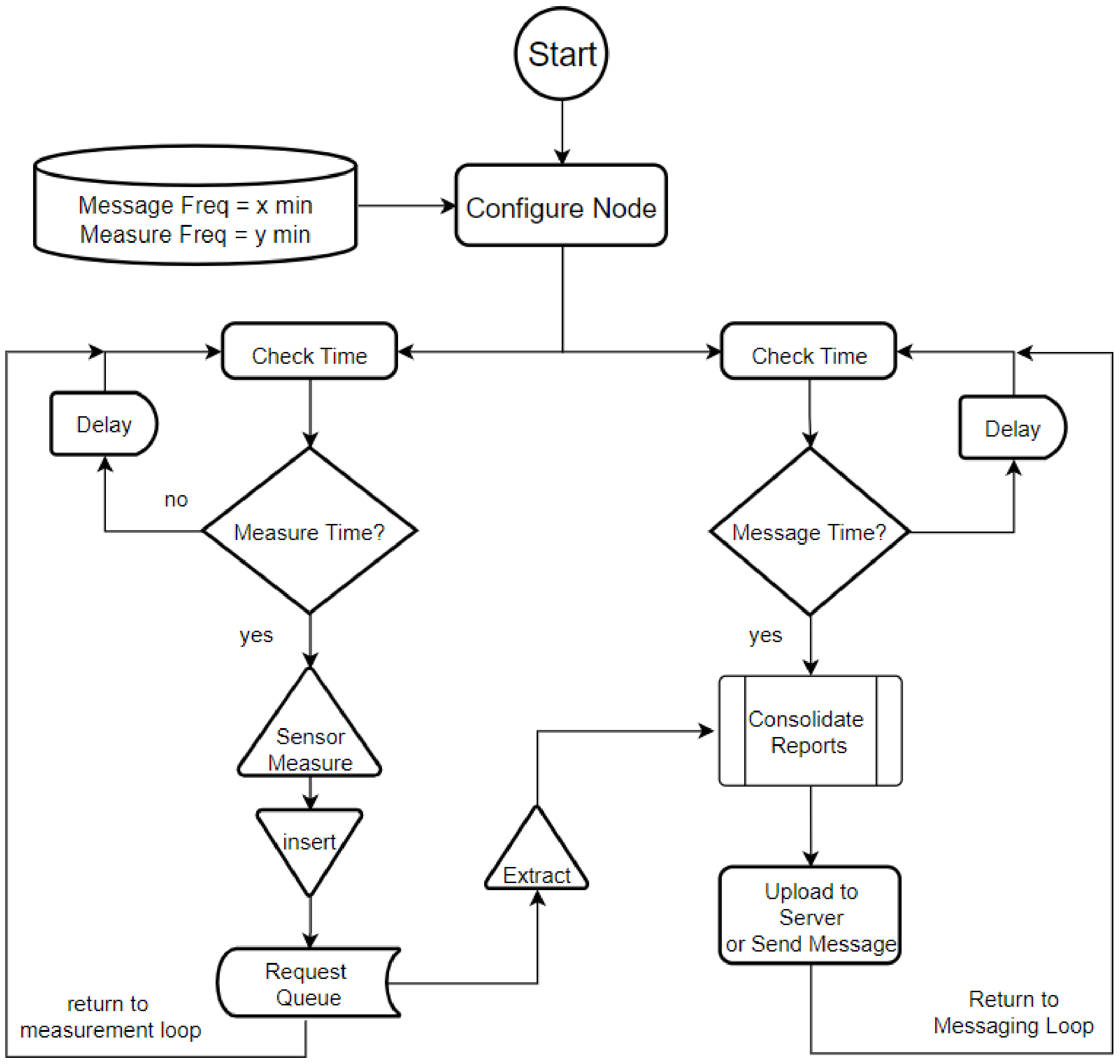


Figure 5.5: eBridge Consolidating Reports Procedure.

5.3.1 Use of IoT LoRa Network

When reviewing the power analysis and study done to the eBridge network in section 4.1, it was necessary to look for alternatives for two communication mechanisms used to communicate with other nodes and the server. One of the critical aspects of this type of study is a good understanding of where the bridges are located in the Costa Rica landscape. This can be seen in figure 5.7, where it is possible to see the results of the bridge survey in [3] when compared with the cellular coverage map of Costa Rica in figure 5.8 it can be observed that Cellular network provides coverage for the mast majority of regions within the country.

As mentioned in work done by [67] IoT devices like the eBridge node require energy-efficient communication mechanisms to deploy scalable networks. Technologies like 2G, 4G, and 4G are designed primarily for large coverage applications involving voice, video,

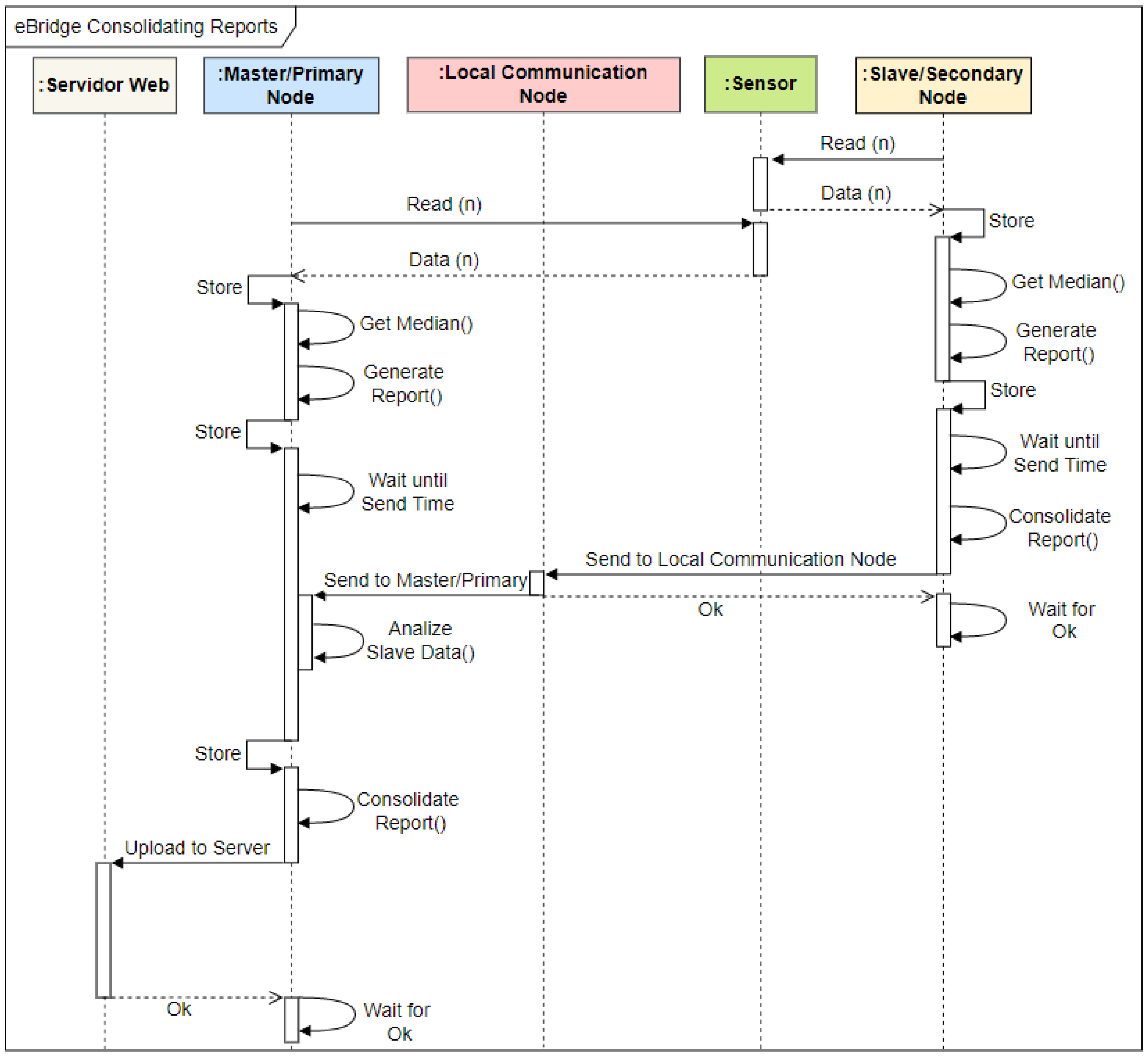


Figure 5.6: eBridge Report Management.

and high data transmissions. However, these usually do not meet the performance metrics for sensor applications, especially regarding power consumption. Technologies like LoRa® (short for Long Range) is Semtech's radio modulation technology of low-power, wide area networks [68]. LoRa® is part of the LPWAN (Low Power Wide Area Network), which proposes a suitable replacement for 2G, 3G, and 4G technologies. As shown in figure 5.9, LoRaWan (LoRa Wide Area Network) is implementing Semtech's radio modulation technology for Wide Area Networks.

Unfortunately, in Costa Rica, there is still not enough coverage of any of the Low Power WAN technologies described in figure 5.9. Companies like **The Things Network** have already installed 3 LoRaWAN gateways in San José [69], and it is possible that in the future, it could be possible to replace the WaveShare 4G HAT and the XBee communication board for a single LoRaWAN communication board. Changing the XBee S2C communication board by an Adafruit RFM95W LoRa Radio Transceiver [70] in figure 5.10 allows

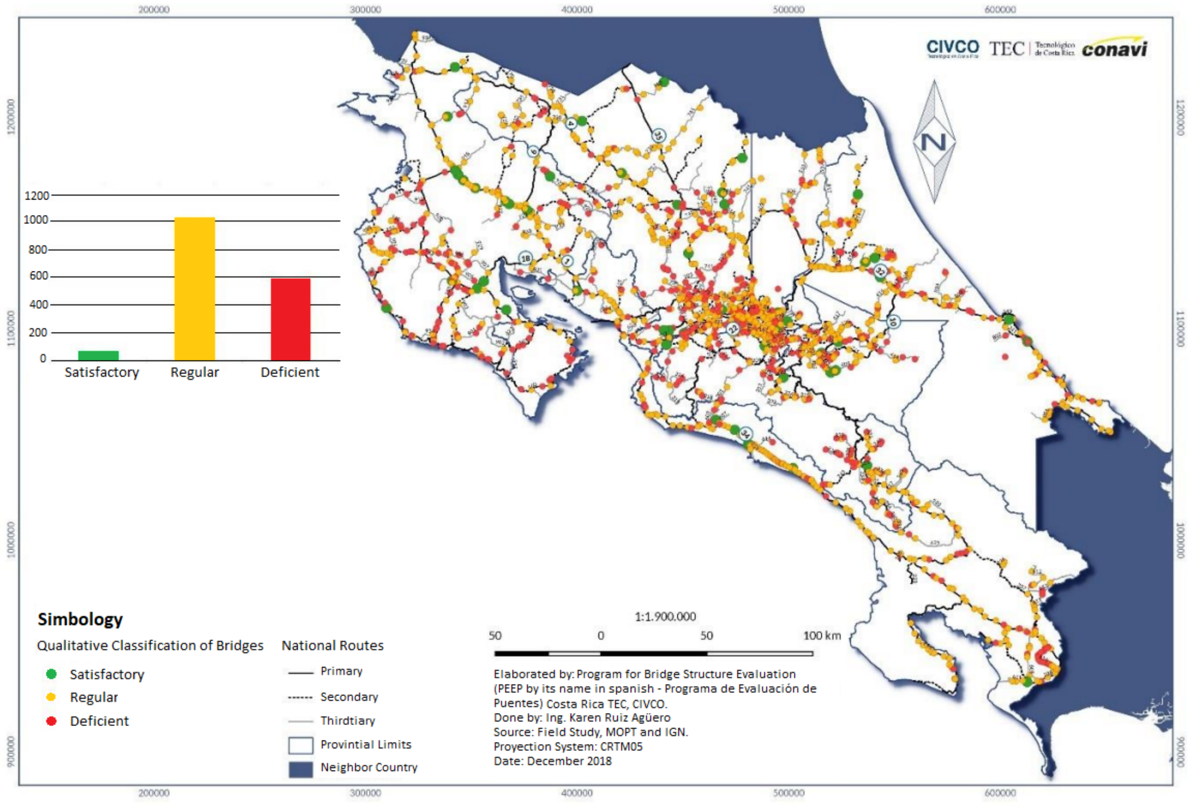

CIVCO TEC Tecnológico de Costa Rica conavi
N
1200
1000
800
600
400
200
0
Satisfactory
Regular
Deficient
Simbology
Qualitative Classification of Bridges
● Satisfactory
● Regular
● Deficient
National Routes
Primary
Secondary
Thirdtiary
Provintial Limits
Neighbor Country
1:1.900.000
50 0 50 100 km
Elaborated by: Program for Bridge Structure Evaluation (PEEP by its name in spanish - Programa de Evaluación de Puentes) Costa Rica TEC, CIVCO.
Done by: Ing. Karen Ruiz Agüero
Source: Field Study, MOPT and IGN.
Proyection System: CRTM05
Date: December 2018

Figure 5.7: Bridge Structures in Costa Rica by State - Survey 2014-2018 [3].

a considerable power-saving since it uses UART communication with the GPIO ports of the RPI system. It would enable DPM management by turning off the USB ports of the RPI; the energetic impact is explored in the chapter 6.

In figure 5.11 there is a diagram of the proposed eBridge node 2022, which is referred to as the **eBridge 2022 HW configuration**. As shown, there are two versions of the configuration, **version A** keeps a USB connection with the WaveShare 4G HAT and is meant to work on master/primary devices, while **version B** has a UART connection. Since the primary/primary device would require more bandwidth to communicate with the eBridge server than the slave device.

To control the RFM9X module to the eBridge node, a class named **RFM9X** is created, allowing sending and receiving messages. Additionally, since the module does not include a native way to verify if a message is received, functions to request and send an acknowledge message are implemented. To allow reconfiguration, the **eBridge Device** class can now be configured to use either XBee or LoRa communication with the use of the configuration file flag named **Communication Type**, which could take the value of *XBee* or *RFM9X*. The new class diagram can be seen in figure 5.12.

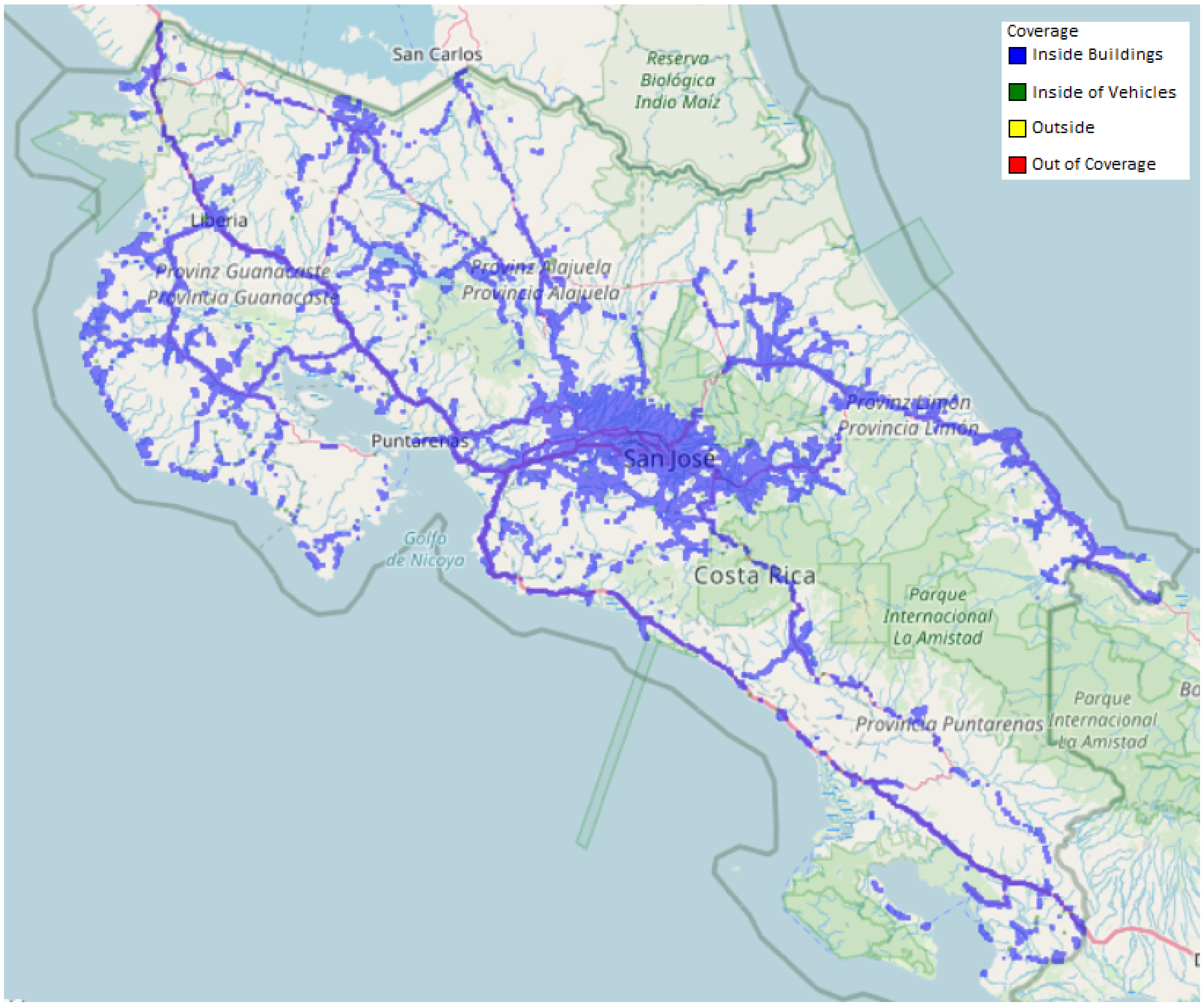


Figure 5.8: LTE Coverage in Costa Rica [66].

5.3.2 Dynamic Power Management and DVFS Proposal

This section describes the different hardware modules to apply the DPM approach to save power. The RPI hardware provides the flexibility of powering down several hardware modules. The goal is to measure the impact of completely powering down the ones that would not be used and dynamically powering on and off the ones that do not require continuous on. The modules to power down are:

- RPI WiFi interface.
- RPI Bluetooth interface.
- RPI Ethernet ports.
- RPI LEDS includes the two Ethernet interface LEDs, the trigger, and the power LED.
- RPI HDMI interface.

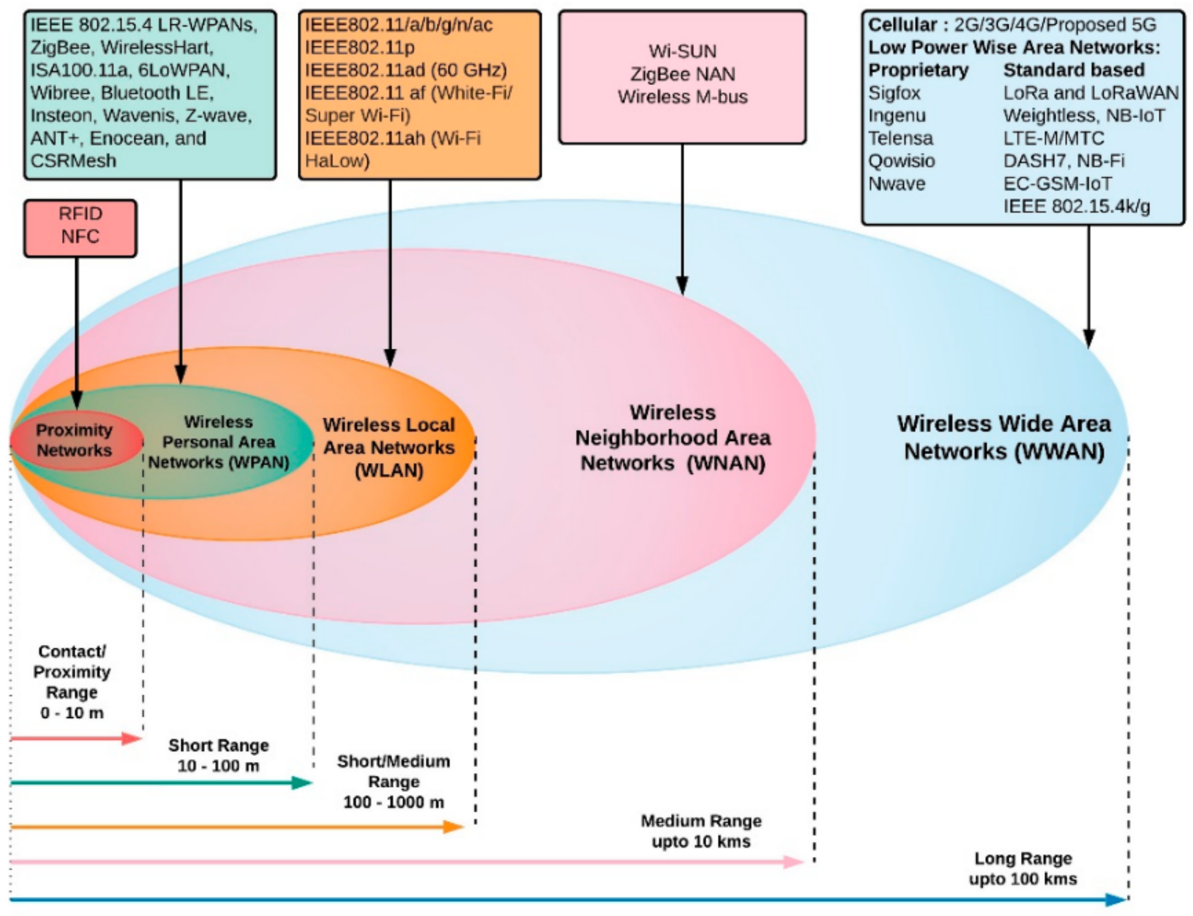

IEEE 802.15.4 LR-WPANs, ZigBee, WirelessHart, ISA100.11a, 6LoWPAN, Wibree, Bluetooth LE, Insteon, Wavenis, Z-wave, ANT+, Enocean, and CSRMesh
IEEE802.11/a/b/g/n/ac IEEE802.11p IEEE802.11ad (60 GHz) IEEE802.11 af (White-Fi/ Super Wi-Fi) IEEE802.11ah (Wi-Fi HaLow)
Wi-SUN ZigBee NAN Wireless M-bus
Cellular : 2G/3G/4G/Proposed 5G
Low Power Wise Area Networks:
Proprietary: Sigfox, Ingenu, Telensa, Qowisio, Nwave
Standard based: LoRa and LoRaWAN, Weightless, NB-IoT, LTE-M/MTC, DASH7, NB-Fi, EC-GSM-IoT, IEEE 802.15.4k/g
RFID NFC
Proximity Networks
Wireless Personal Area Networks (WPAN)
Wireless Local Area Networks (WLAN)
Wireless Neighborhood Area Networks (WNAN)
Wireless Wide Area Networks (WWAN)
Contact/ Proximity Range 0 - 10 m
Short Range 10 - 100 m
Short/Medium Range 100 - 1000 m
Medium Range upto 10 kms
Long Range upto 100 kms

Figure 5.9: Wireless access geographic coverage [67]. RFID (Radio Frequency Identification), NFC (Near Field Communication).

- RPI USB Interface for slave/secondary units. For the master, it would be dynamically managed as needed.

RPI3 B+ allows manual voltage and frequency scaling on its ARM53 V7 CPU, while the RPI4 B unit allows dynamic voltage and frequency scaling on its ARM72 V8 CPU. As part of this research, the goal is to test the impact of lowering the frequency and voltage on the total power consumption of the eBridge node. DVFS can be enabled in the RPI3 B+ and the RPI4 B nodes by using the **dvfs** flag on the **/boot/config.exe** file of the RaspberryPI OS [71]. In the next chapter 6 there are detailed results of the tests proposed in this chapter.

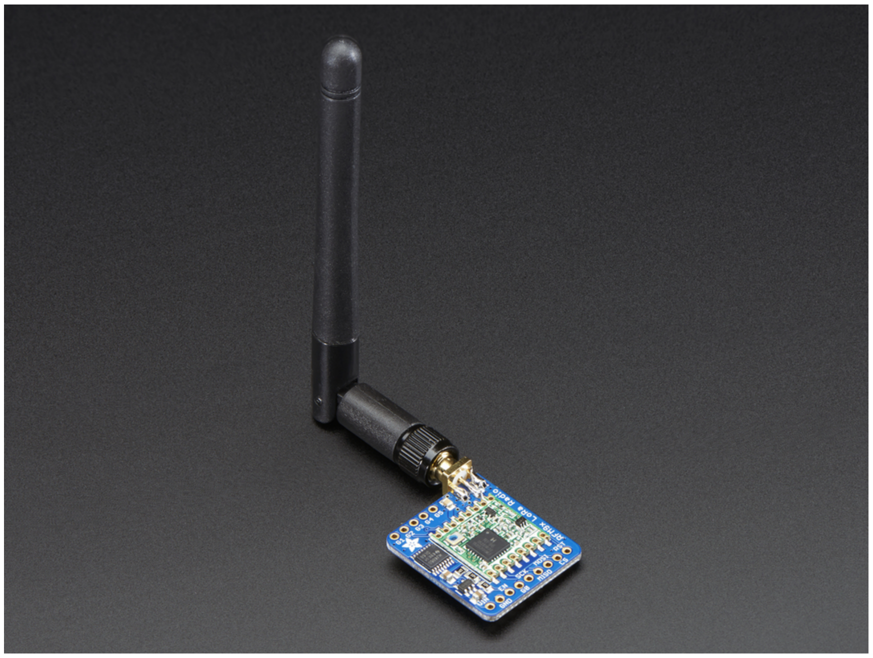


Figure 5.10: Adafruit RFM95W LoRa Radio Transceiver [70].

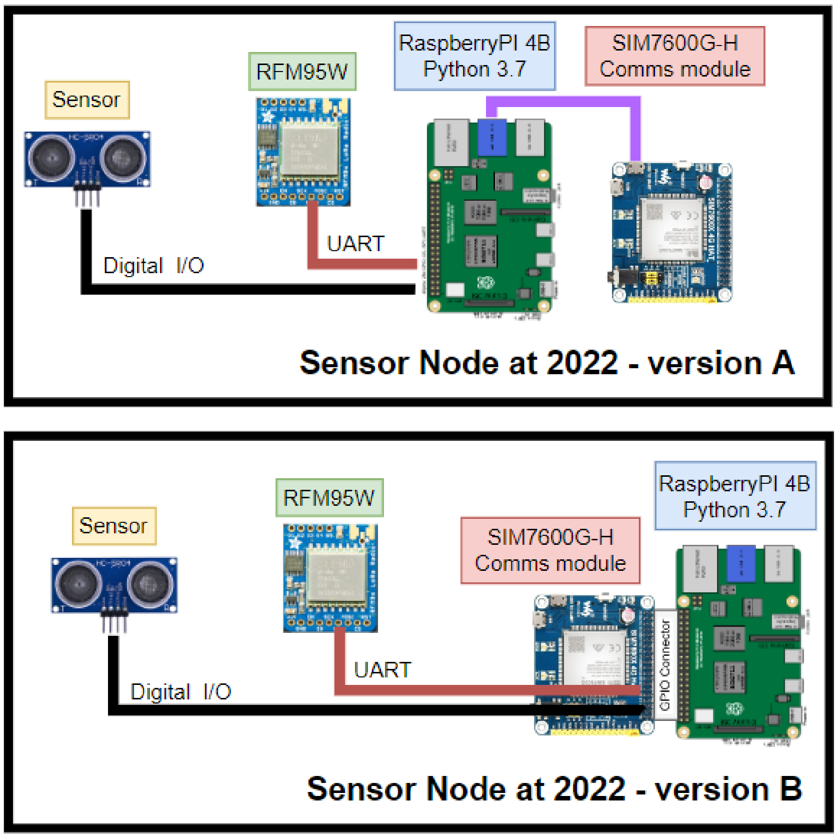


Figure 5.11: eBridge 2022 Hardware Configurations version A and B.

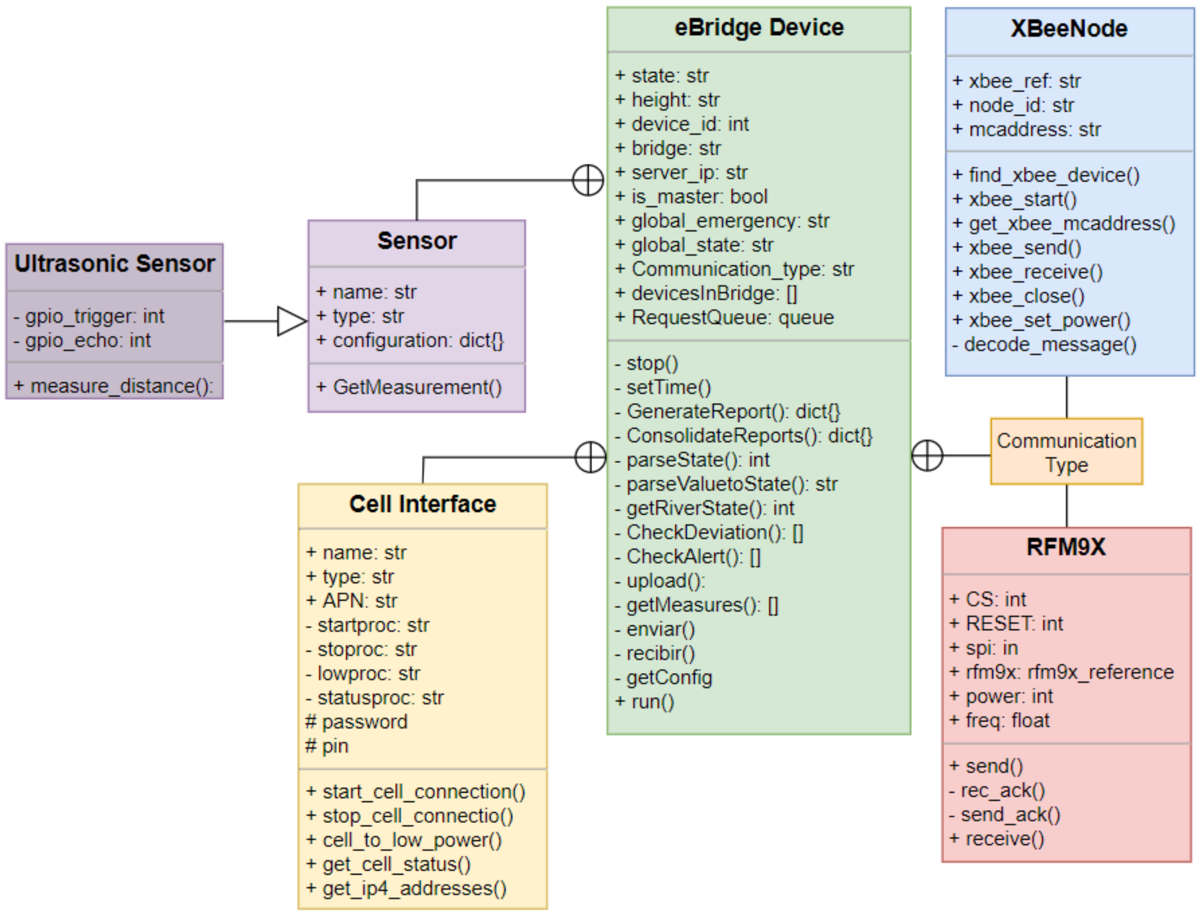


Figure 5.12: eBridge 2022 Class UML Diagram.

Chapter 6

Validation

In this chapter, there is a detailed description of the results of implementing the design recommendations explained in chapter 5. In section 6.1, there is a summary of the new tasks identified to perform line profiling for the 2021 and 2022 eBridge codes. The eBridge 2021 code is the code with XBee communication, and the eBridge 2022 is the one with the LoRa communication. The DVFS results can be found in section 6.3, and the results for the eBridge 2022 implementation with LoRa for both master/primary and slave/secondary are described in section 6.5.

6.1 eBridge 2021 Node Code Analysis

In section 6.1.2 it is possible to observe the profiling results for the eBridge 2021 Optimized code for master/primary nodes with XBee communication and in section 6.1.3 the ones for slave/secondary nodes. The results of the battery tests for the eBridge 2021 optimized code for with XBee can be observed in section 6.2, the ones for master/primary can be found in subsection 6.2.1, and for slave/secondary in 6.2.2.

6.1.1 eBridge 2021 Line Profiling

The same strategy followed in section 4.1.1 was applied to the optimized eBridge 2021 code described in the previous chapter 5. Some additional functions of interest were added:

- **Get Configuration - ID 5** (τ_5): Used by both types of nodes to retrieve a configuration from the eBridge Server.
- **Configure - ID 6** (τ_6): Allows a node to configure itself, initiate the **Sensor**, **Cell_Interface**, and **XBeeNode** or **RFM9X** classes.

- **Start Cell Connection - ID 7** (τ_7): As described in subsection 5.1.2 of chapter 5, it is used to start a cellular connection.
- **Stop Cell Connection - ID 8** (τ_8): Used to stop the cell connection.

6.1.2 eBridge 2021 Master/Primary CPU and Events Profiling

This section describes the tests performed on RPI3 B+ and RPI4 B nodes running the eBridge 2021 code, which is the optimized version of the eBridge 2019 code. In the table 6.1, there is a summary of the experiment ran on RPI 3B+. The results for CPU Load can be seen in figures 6.1, and 6.2 from which it can be concluded that there is a considerably lower CPU utilization compared to the original code, the highest CPU value is 26 %. The specific values are discussed further in this section.

Table 6.1: Measurement Summary eBridge code 2021 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI3 B+

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI3 B+
Controller	Raspberry PI 3 B+
Device ID	RPI3-2
Code Running	eBridge 2021 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Master/Primary
Total Test Duration	1 hour 7 minutes and 6 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples by CPU Core	4026 samples
CPU Frequency applied to all cores(f)	1.4 GHz
Power Measurement Summary	
Power Source	CanaKit RPI3 B+ 5V 2.5A [59]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	8 114 000

Two significant events in which the eBridge 2021 code increased CPU utilization were

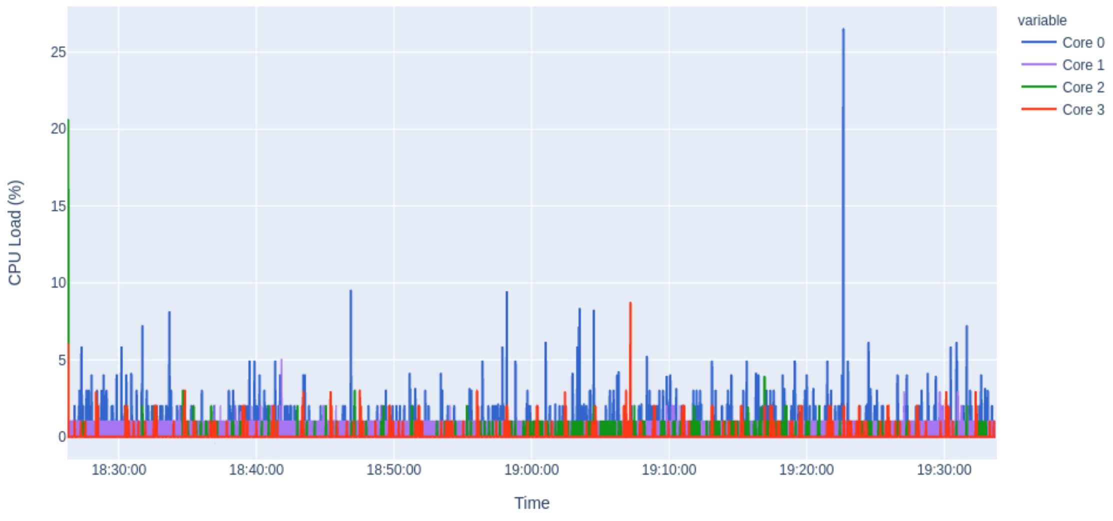


Figure 6.1: CPU Load (%) for eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 6.1.

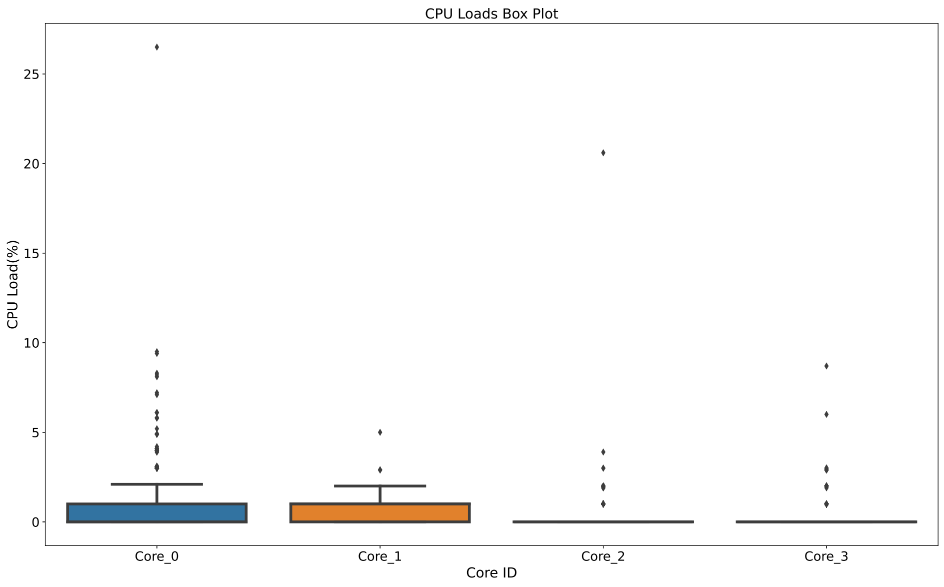


Figure 6.2: CPU Load Box (%) plot for eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 6.1

identified. The first is the configuration stage, which can be observed in figure 6.3, where the CPU utilization briefly goes over 20 %. The other event is the execution of the tasks **Get Median ID: 4**τ_4 and **Generate Report ID: 3**τ_3, this can be seen in figure 6.4, where the CPU reaches its maximum utilization of 26 %. The configuration stage ideally does not have to be executed regularly in normal deployment circumstances so it can be neglected. When reviewing the power utilization for the RPI3 B+ in figure 6.5, it is possible to see that there are no noticeable power spikes, and the power plot stays stable around a value between 2.8 W and 3.2 W. Also, there are no stalling behaviors during the execution of the **XBee Receive - ID:0** τ_0 task, so the changes described in subsection 5.1.4 to the **RequestQueue** object to prevent thread locking worked on RPI3 B+ nodes.

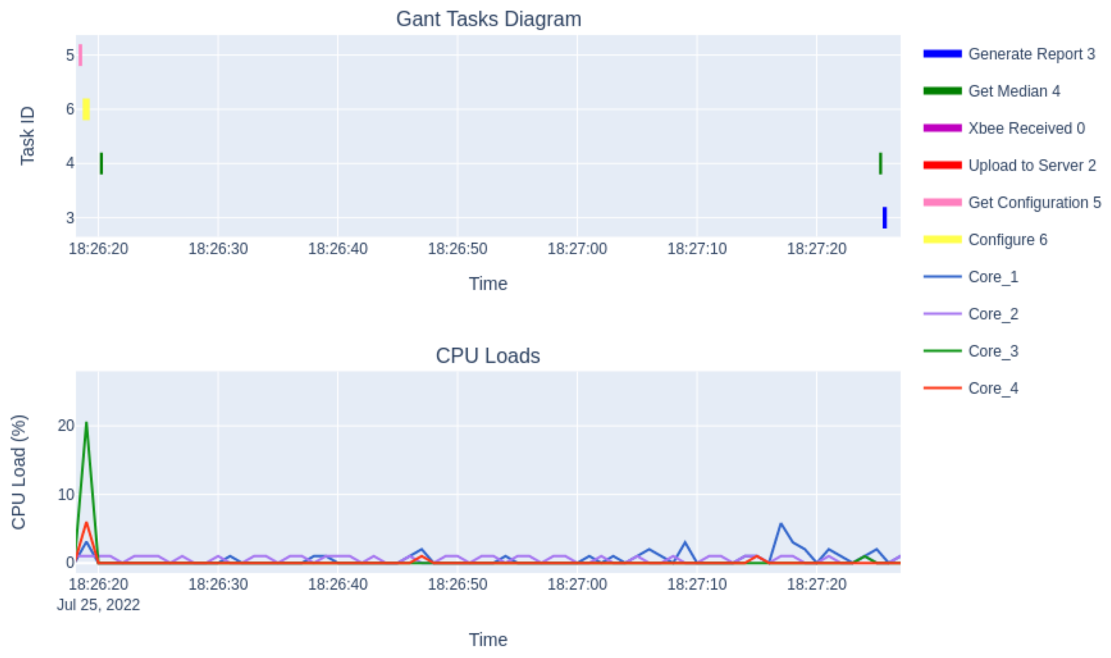


Figure 6.3: Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.1

The summary of the test performed on an RPI4 B node running the eBridge 2021 optimized code as master/primary can be observed in table 6.2. The CPU Load % results can be seen in figure 6.6, in which it is possible to observe that there is a 10 % power spike during the initialization of the node, that later when reviewing the Gantt Plot in figure 6.7, it is possible to see that it corresponds to the configuration stage where the **Get Configuration ID:5**τ_5, and **Configure ID:6**τ_6 tasks execute. However, it is possible to see how the CPU Load spike during the execution of the tasks **Get Median ID: 4**τ_4 and **Generate Report ID: 3**τ_3 is not present in the RPI4 B node.

The power plot for the PPI4 B node running as master/primary in figure 6.8 presents a similar behavior to the one in figure 6.5 for the RPI3 B+, where there are no noticeable power spikes during the execution of the code. When comparing the power data for both types of RPI devices running as master, it is possible to build table 6.4, in which it is possible to see that the mean power for RPI3 B+ of 3.0848 W that is 18.6 smaller than

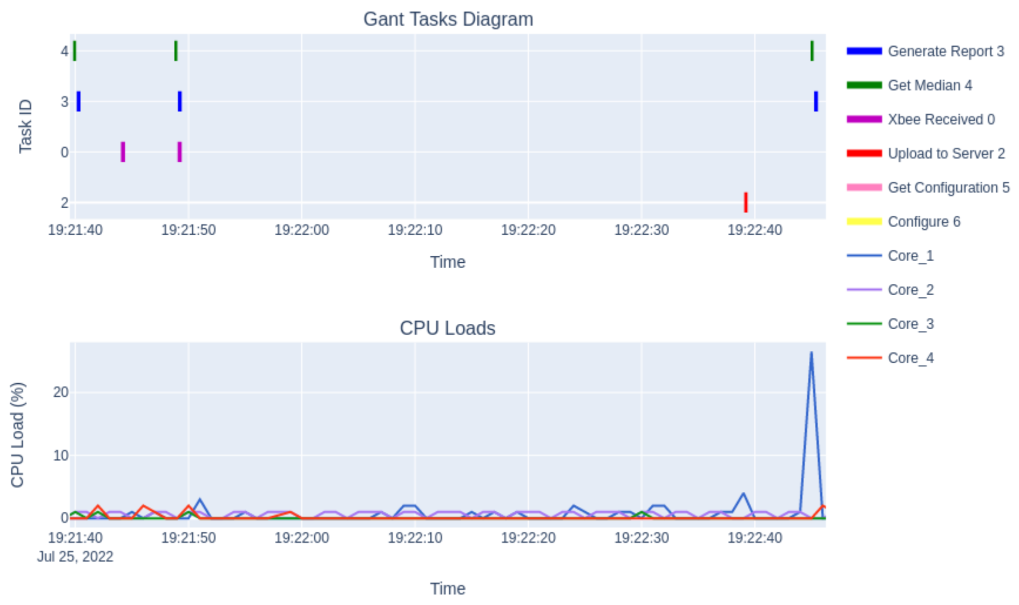


Figure 6.4: Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz, focus on Get Median τ_4. Details of the test in table 6.1

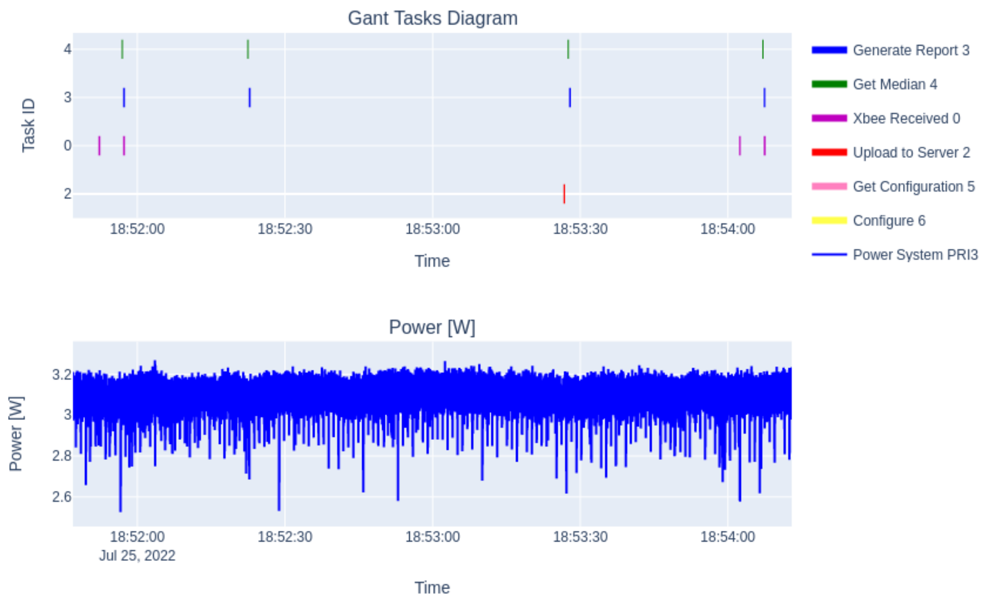


Figure 6.5: Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 6.1

Table 6.2: Measurement Summary eBridge code 2021 running as Master/Primary Sensor node in eBridge 2021 hardware configuration on RPI4 B

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI4 B
Controller	Raspberry PI 4 B
Device ID	RPI4-2
Code Running	eBridge 2021 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Master/Primary
Total Test Duration	1 hour 7 minutes and 19 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples by CPU Core	4039 samples
CPU Frequency applied to all cores(f)	1.4 GHz
Power Measurement Summary	
Power Source	CanaKit RPI4 B 5V 3.5A [60]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	8 078 000

the 3.9797 W mean value for the eBridge 2019 code. Also, this value is only 1.45 % of the standby mean power value for the PRI3 B+ of 3.0402 W described in section 4.4.

For the RPI4 B node, the mean power value of the optimized code is 3.3781 W, which is 26.22 % smaller than the 4.5788 W for the eBridge 2019 code, and just 10.3 % higher than the standby power. This reduction in power utilization is directly related to the decrease in CPU Load for both the RPI3 B+ and RPI4 B nodes that can be seen in table 6.3. The mean CPU load % for both types of nodes is less than one percent. The $\vec{C}_i^f$ statistical information for both the RPI3 B+ and the RPI4 B running as master/primary can be found in table 6.5, it can be noticed how the time scale had to be changed from seconds (s) to milliseconds (ms), compared to the data in section 4.1.2 for the 2019 eBridge code. It can be observed that the execution times are considerably lower with a lower better CPU utilization.

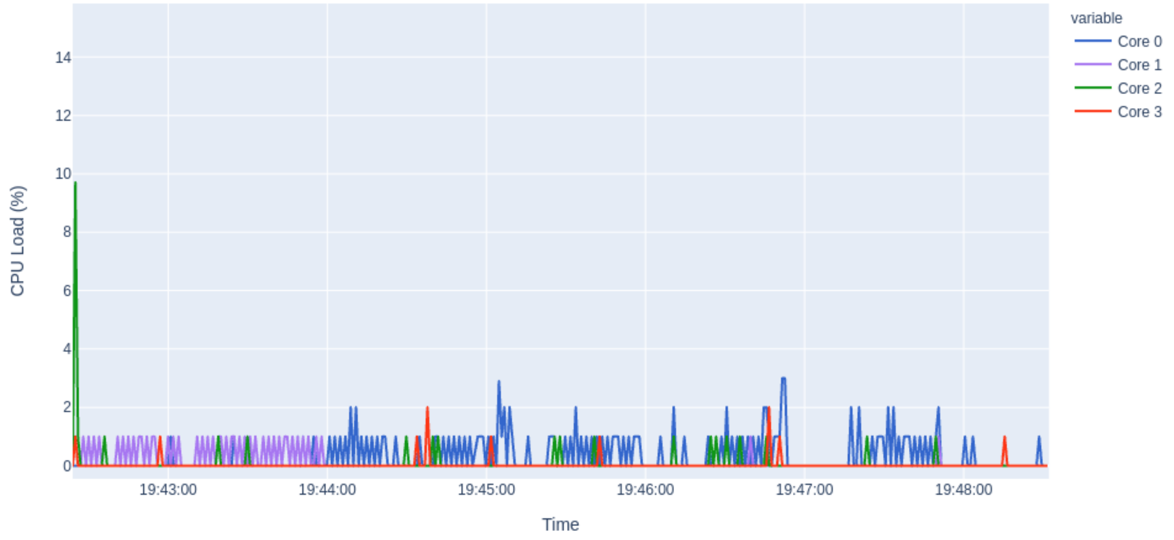


Figure 6.6: CPU Load (%) for eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details in table 6.2

Table 6.3: Comparison of Code CPU Load (%) statistical information of the eBridge 2021 code running as a master/primary node running in eBridge 2021 HW in RPI3 B+ and RPI4 B

CPU Load (%)				
Core	**1**	**2**	**3**	**4**
Code running on RPI3 B+				
Number of samples per core = 4026				
Mean	0.45	0.61	0.07	0.09
Std	0.97	0.51	0.43	1.62
Code running on RPI4 B				
Number of samples per core = 4039				
Mean	0.26	0.01	0.02	0.23
Std	0.6033	0.4249	0.23	0.34

The data of the $\vec{E}_i^f$ vector for both RPI3 B+ and RPI4 B nodes running as master/primary is summarized in table 6.6, where is possible to see that in average the most power expensive task is **XBee Received ID:0**τ_0 task, with an average value of 3.4452 W for RPI3 B+ and 3.7311 W for RPI4 B, which are 11.68 % and 10.45 % higher respectively compared to the mean power values for their respective type of node. A summary of the tests results of the eBridge 2021 code running as slave/secondary can be seen in the next section 6.1.3.

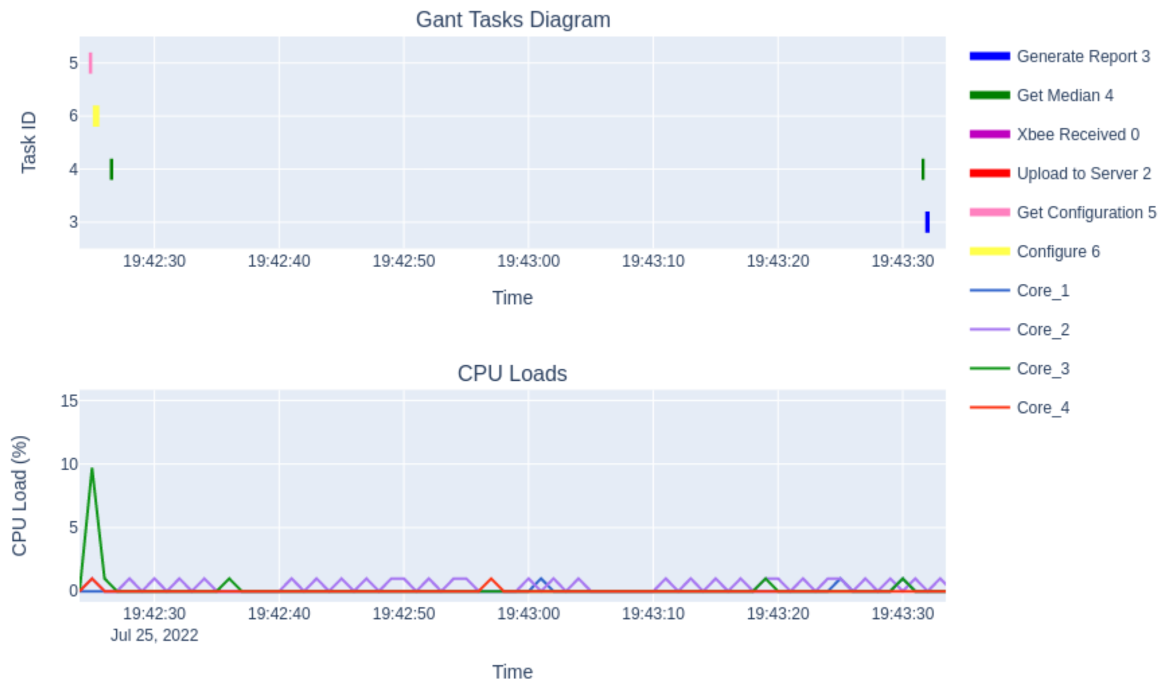


Figure 6.7: Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.2

Table 6.4: Power Comparison (W) eBridge 2021 Optimized code running as master/primary sensor node running in eBridge 2021 HW in both RPI3 B+ and RPI4 B. Test details in tables 6.1 and 6.2

	Power [W]	
Node	**RPI3 B+**	**RPI4 B**
n	8 114 000	8 078 000
Mean	3.0848	3.3781
Std	0.076	0.061
Min	2.1064	2.7873
Max	3.3155	4.1480

6.1.3 eBridge 2021 Slave/Secondary CPU and Events Profiling

A detailed description of the results for slave/secondary nodes running the eBridge 2021 optimized code with XBee communication can be found in this section. The details of the test performed in RPI3 B+ and RPI4 B nodes can be observed in tables 6.7, and 6.8 respectively. Both tests were run with the same type of configuration as the tests for master devices in section 6.1.2.

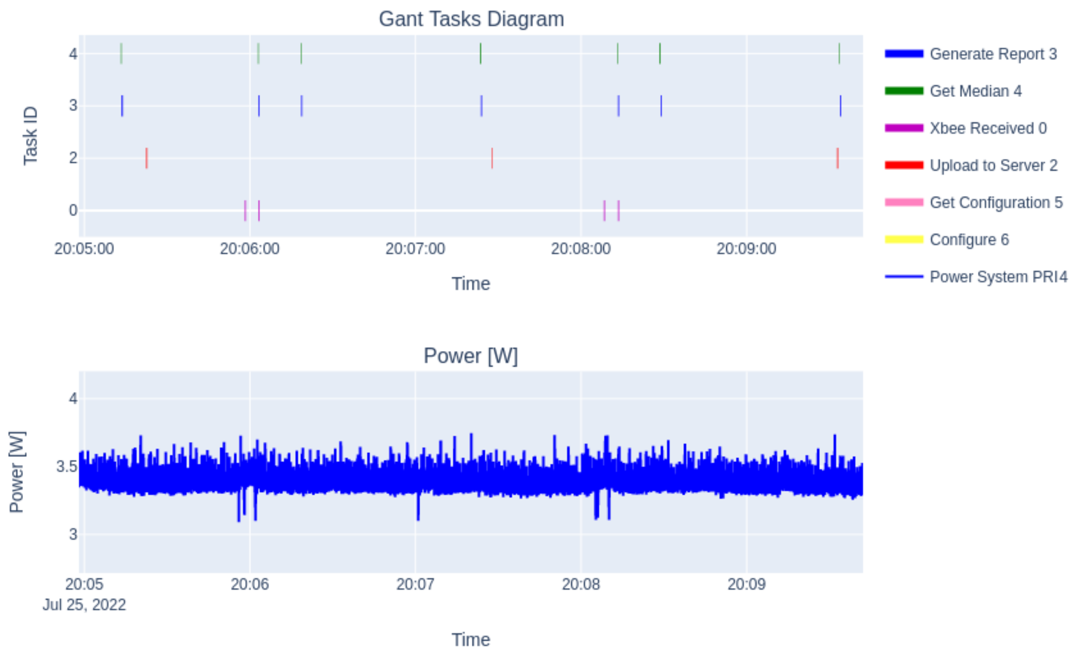


Figure 6.8: Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 6.2

Table 6.5: Comparison of $\vec{C}_i^f$ with f=1.4 GHz for RPI3 B+, and $\vec{C}_i^f$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2021 Optimized code in eBridge 2021 HW configuration as master/primary. Test details in tables 6.1 and 6.2

Execution Time (ms)				
Task ID	τ_4	τ_3	τ_2	τ_0
Running on RPI3 B+ node				
n	94	93	32	62
Mean	3.284	0.996	35.543	4.901
Std	0.390	0.126	3.625	6.965
Min	2.615	0.919	24.754	0.443
Max	5.395	1.598	40.087	50.195
Running on RPI4 B node				
n	107	107	37	72
Mean	2.054	0.658	20.82	26.59
Std	0.431	0.172	4.080	24.35
Min	1.564	0.582	17.26	0.265
Max	5.556	1.936	41.50	85.62

In figures 6.9, and 6.10, it is possible to notice a similar behavior as the one for RPI3 B+ master nodes running the eBridge 2021 code where there is a CPU spoke during the execution of the configuration stage where the **Get Configuration ID:5** τ_5, and **Configure ID:6** τ_6 tasks execute. The same happens when the tasks **Get Median ID: 4**τ_4 and **Generate Report ID: 3** τ_3 execute. It can be noted how the **XBee Send ID:1** τ_1 task does not impact the CPU utilization.

The power plot for the RPI3B + node running the eBridge 2021 optimized plot can be observed in figure 6.11, where it is possible to see that the only noticeable change in power is after the execution of the configuration stage, where the power stays around 2.6 W for a few seconds. This behavior would be further addressed in section 6.3.

The details of the test performed over the RPI4 B node with the eBridge 2021 optimized code are in table 6.8. The CPU Load results in figure 6.12, show how the CPU load does not go over 12 %. When further analyzing this against the task execution, it is possible to see the results in figure 6.13, where it can be observed that there is also a CPU spike during the configuration phase, the same as for the RPI3 B running as slave/secondary and for both nodes running as master/primary. The $\vec{C}_i^f$ statistical information for both the RPI3 B+ and the RPI4 B were very similar, and this does not provide additional or relevant information for discussion.

The power plot information of the eBridge 2021 code running as slave/secondary in RPI4 B is present in figure 6.14 has a similar trend as the other power plots for the optimized code like figures 6.11, 6.5, and 6.8, in which is not possible to see any relevant power spikes

Table 6.6: Comparison of $\vec{E}_i^f$ with f=1.4 GHz for RPI3 B+, and $\vec{E}_i^f$ with f=1.5 GHz for RPI4 B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_2, \tau_0\}$ of eBridge 2021 code as slave/secondary in eBridge 2021 HW configuration. Test details in tables 6.1 and 6.2

	Power [W]			
Task ID	τ_4	τ_3	τ_2	τ_0
	Code running on RPI3 B+			
n	2 159 792	2 479 784	1 702 104	1 755 856
Mean	3.0758	3.0862	3.0859	3.4452
Std	0.082	0.0778	0.0716	0.0824
Min	2.5005	2.5007	2.5059	2.5005
Max	3.2817	3.3155	3.2811	3.6456
	Code running on RPI4 B			
n	2 899 872	2 307 032	1 978 000	1 371 864
Mean	3.3721	3.3753	3.6571	3.7311
Std	0.0685	0.0561	0.0538	0.0465
Min	2.8154	2.9744	2.9485	3.0063
Max	3.8935	3.9734	3.923	4.0228

Table 6.7: Measurement Summary eBridge code 2021 running as Slave/Secondary Sensor node in eBridge 2021 hardware configuration on RPI3 B+

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI3 B+
Controller	Raspberry PI 3 B+
Device ID	RPI3-2
Code Running	eBridge 2021 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Slave/Secondary
Total Test Duration	1 hour 17 minutes and 6 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples by CPU Core	4626 samples
CPU Frequency applied to all cores(f)	1.4 GHz
Power Measurement Summary	
Power Source	CanaKit RPI3 B+ 5V 2.5A [59]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	9 252 000

during the execution of any of the tasks. However, when viewing the data in table 6.10, of slave/secondary nodes running the optimized code, it can be noted the mean power of 3.0768 W for an RPI3 B+ is 22.43 % smaller than the 3.09663 W of the eBridge 2019 code, for RPI4 B the 3.4224 W is 29 % smaller than the previous value of 4.82 W.

The statistical summary of the $\vec{E}_i^f$ vectors for both RPI3 B+ and RPI4 B nodes as slave/secondary with the eBridge 2021 optimized code is presented in table 6.12. The improvements on average power for the **Get Median ID:4** τ_4, **Generate Report ID:3** τ_3, and **XBee Send ID:1** τ_1 tasks, are of 22.32 %, 22.17 %, and 22.43 % respectively. When analyzing the vector of task execution's $\vec{C}_i^f$ for the slave/secondary eBridge 2021 optimize code in both RPI3 B+ and RPI4 B nodes, it is possible to come with the summary in table 6.11, where the data is very similar to the one for the master/primary node in table 6.5. Overall the execution time of the tasks **Get Median ID:4** τ_4, **Generate Report ID:3** τ_3 was reduced, while the one for **XBee Send ID:1** τ_1 was increased. This was done to help reduce the CPU utilization of the slave/secondary node. In the next

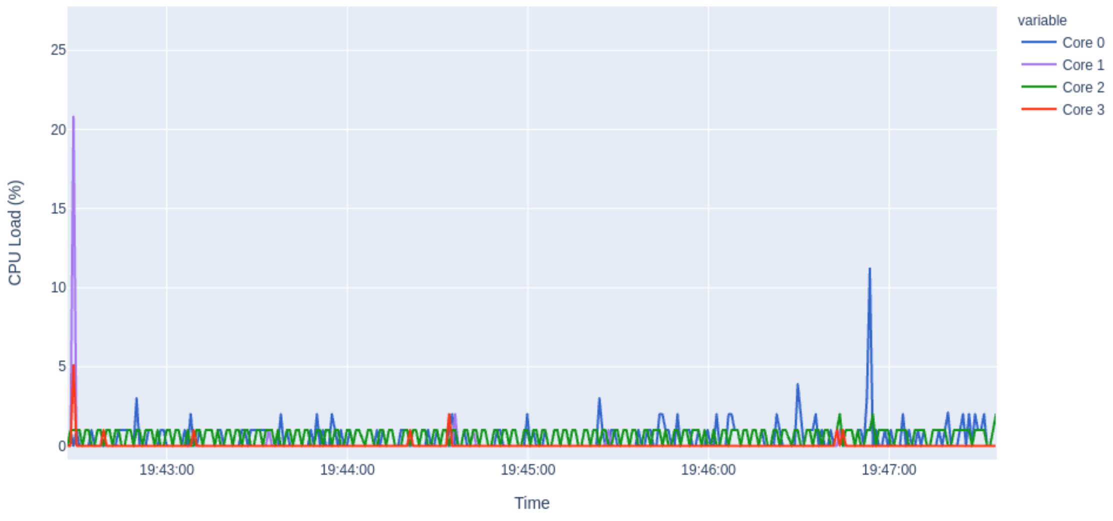


Figure 6.9: CPU Load (%) for eBridge 2021 code running as slave/secondary a sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details in table 6.7

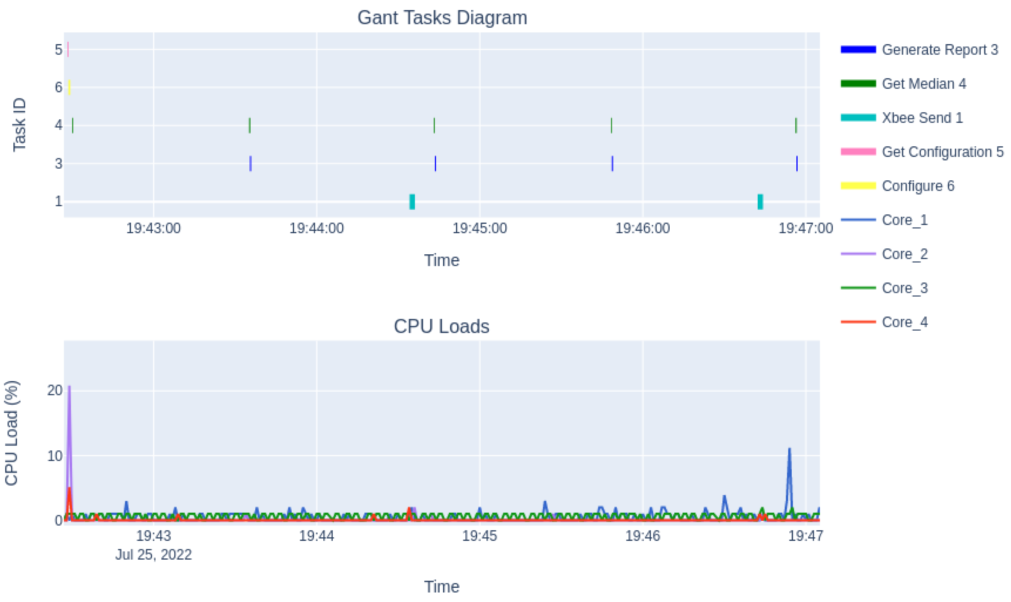


Figure 6.10: Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.7

section 6.2 there is a detailed description of the battery test results performed on the eBridge 2021 code.

6.2 eBridge 2021 Battery Tests

This section aims to measure the impact of the code improvements described in chapter 5 done on the eBridge 2019 code. The tests were done using the same 5200 mAh and 10 000 mAh batteries mentioned in section 3.5.1, which were used to test the eBridge 2019 code. The results for master/primary nodes can be found in subsection 6.2.1, and the ones for slave/secondary ones in subsection 6.2.2.

6.2.1 Battery tests for Master/Primary eBridge 2021 code

Initial battery test results in table 6.13 for master/primary nodes allow an improvement of 226.87 % with a 5 200 mAh from two hours, 15 minutes, and 33 seconds to seven hours, 23 minutes, and four seconds for an RPI3 B+ with the eBridge 2021 HW configuration, after using the eBridge 2021 optimized code. In the case of the 10 000 mAh battery, the improvement is 316.67 % from three hours, 31 minutes, and 12 seconds to 14 hours, 40

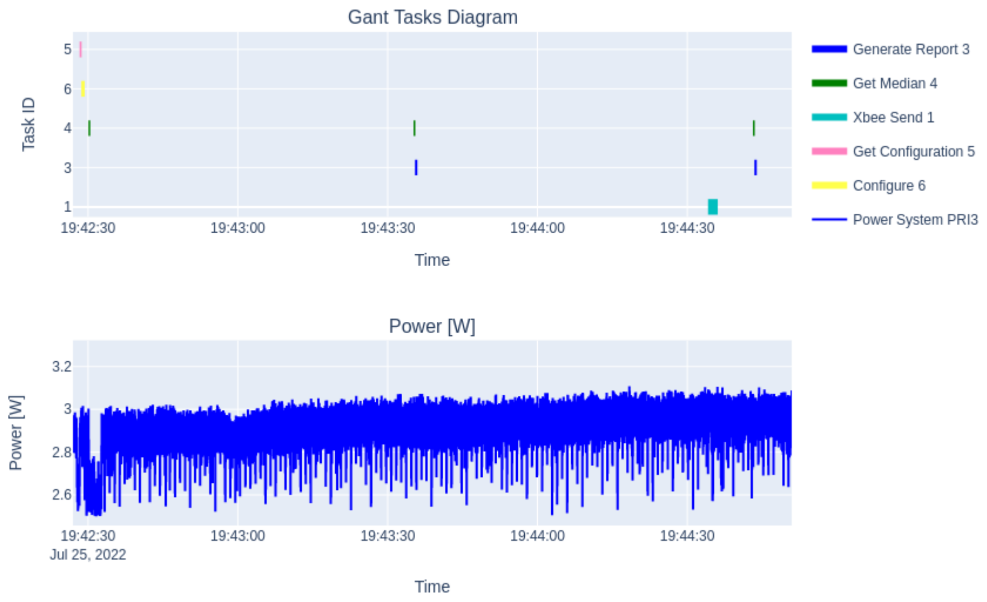


Figure 6.11: Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI3 B+ with f = 1.4 GHz. Details of the test in table 6.7

Table 6.8: Measurement Summary eBridge code 2021 running as Slave/Secondary Sensor node in eBridge 2021 hardware configuration on RPI4 B

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2021 HW with RPI4 B
Controller	Raspberry PI 4 B
Device ID	RPI4-2
Code Running	eBridge 2021 code
Cellular Communications Board	SIM7600G-H
XBee Board	S2C DigiMesh
Type of the Device	Slave/Secondary
Total Test Duration	1 hour 7 minutes and 37 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples by CPU Core	4057 samples
CPU Frequency applied to all cores(f)	1.4 GHz
Power Measurement Summary	
Power Source	CanaKit RPI4 B 5V 3.5A [60]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	8 114 000

Table 6.9: Comparison of Code CPU Load (%) statistical information of the eBridge 2021 code running as a slave/secondary node running in eBridge 2021 HW in RPI3 B+ and RPI4 B

CPU Load (%)				
Core	**1**	**2**	**3**	**4**
Code running on RPI3 B+				
Number of samples per core = 4626				
Mean	0.75	0.04	0.39	0.02
Std	1.10	0.37	0.54	0.15
Code running on RPI4 B				
Number of samples per core = 4057				
Mean	0.26	0.01	0.02	0.23
Std	0.59	0.19	0.14	0.44

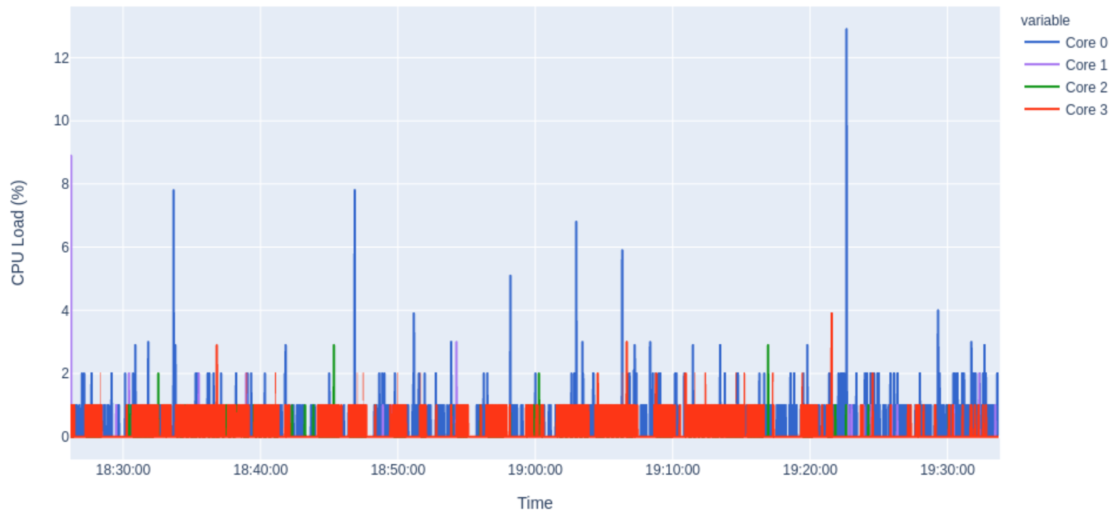


Figure 6.12: CPU Load (%) for eBridge 2021 code running as slave/secondary a sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details in table 6.8

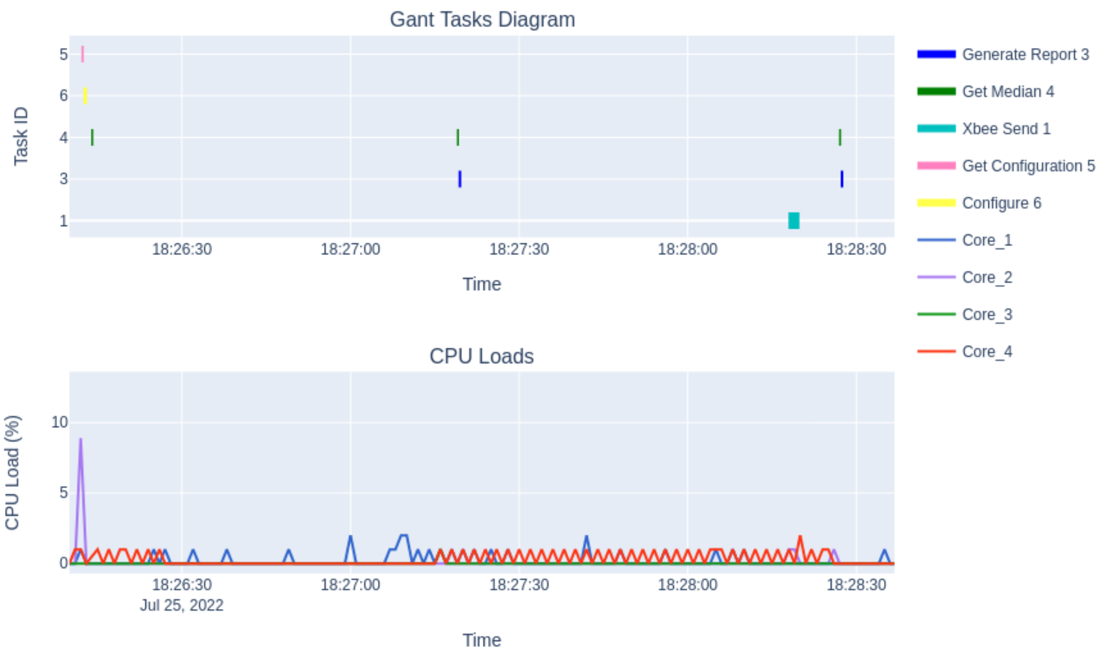


Figure 6.13: Tasks Gantt Plot vs CPU Load (%) eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz, focus on Configuration τ_5,and τ_6. Details of the test in table 6.8

minutes, and one second. For the case of the RPI4 B node, the improvement for a 5 200 mAh battery is 218.19 % from one hour, 48 minutes, and 36 seconds to five hours,

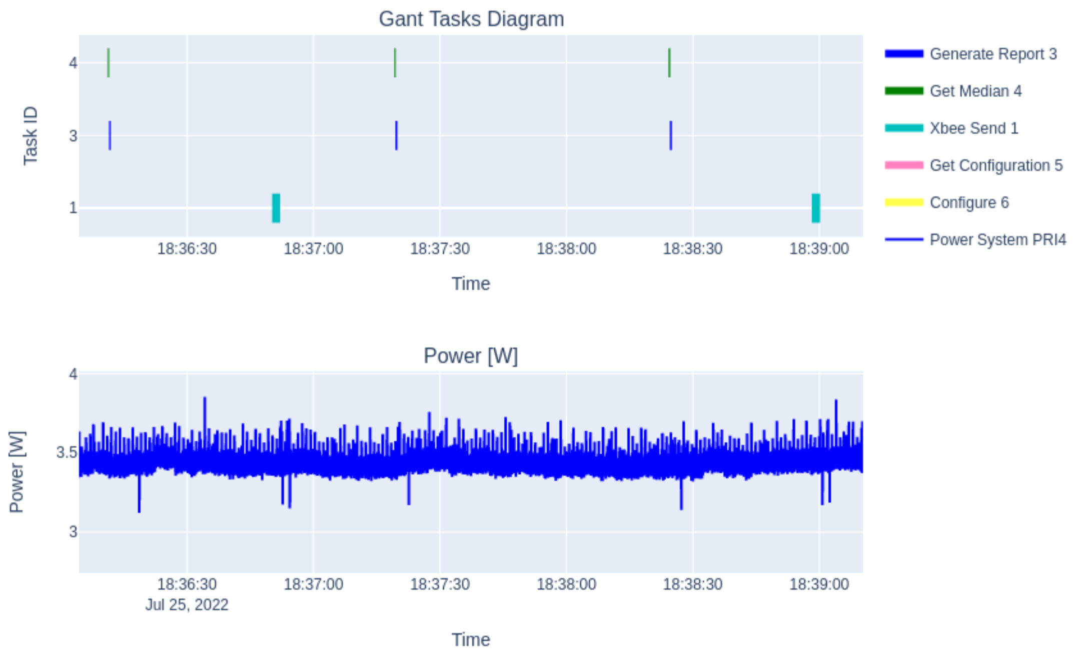


Figure 6.14: Tasks Gantt Plot vs Power (W) of eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI4 B with f = 1.5 GHz. Details of the test in table 6.8

Table 6.10: Power Comparison (W) eBridge 2021 Optimized code running as slave/secondary sensor node running in eBridge 2021 HW in both RPI3 B+ and RPI4 B. Test details in tables 6.7 and 6.8

	Power [W]	
Node	**RPI3 B+**	**RPI4 B**
n	9 252 000	8 114 000
Mean	3.0768	3.4224
Std	0.070	0.028
Min	2.1221	2.7939
Max	3.2998	4.1431

45 minutes, and 33 seconds. On the other hand, with a battery of 10 000 mAh the improvement is 228.13 % from three hours, 21 minutes, and 28 seconds to eleven hours, one minute, and four seconds.

Table 6.11: Comparison of $\vec{C_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{C_i^f}$ with f=1.5 GHz for RPI 4B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2021 Optimized code in eBridge 2021 HW configuration as slave/secondary. Test details in tables 6.1 and 6.2

Execution Time (ms)			
Task ID	τ_4	τ_3	τ_1
Running on RPI3 B+ node			
n	71	70	36
Mean	2.881	0.955	1791
Std	0.310	0.183	0.868
Min	2.567	0.805	1790.7
Max	3.672	1.855	1.7925
Running on RPI4 B node			
n	61	60	31
Mean	2.033	0.667	1791
Std	0.339	0.114	0.395
Min	1.118	0.575	1789
Max	3.243	0.938	1792

Table 6.12: Comparison of $\vec{E_i^f}$ with f=1.4 GHz for RPI3 B+, and $\vec{E_i^f}$ with f=1.5 GHz for RPI4 B vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2021 code as slave/secondary in eBridge 2021 HW configuration. Test details in tables 6.7 and 6.8

Power [W]			
Task ID	τ_4	τ_3	τ_1
Code running on RPI3 B+			
n	3 646 320	3 187 740	2 447 744
Mean	3.0775	3.0906	3.1811
Std	0.075	0.0600	0.0622
Min	2.500	2.5021	3.0874
Max	3.2831	3.2811	3.2757
Code running on RPI4 B			
n	2 830 992	3 173 784	2 106 872
Mean	3.4215	3.3516	3.6115
Std	0.073	0.0598	0.056
Min	2.8056	2.9874	3.0239
Max	3.8958	3.9515	3.9027

Table 6.13: Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2021 code as master/primary in eBridge 2021 HW configuration.

Type of Configuration:	Measurement Frequency = 1 min Message Frequency = 5 min	
Device ID	**Battery ID**	**Total Run Time [HH:MM:SS]**
RPI3-2	5 200 mAh - 1	07:23:04
RPI3-2	10 000 mAh -1	14:40:01
RPI4-2	5 200 mAh - 3	05:45:33
RPI4-2	10 000 mAh -3	11:01:04

6.2.2 Battery tests for Slave/Secondary eBridge 2021 code

The battery tests results for slave/secondary nodes are detailed in table 6.14. The improvement for RPI3 B+ is 225 % with a battery of 5 200 mAh, from one hour, 54 minutes, and 33 seconds to six hours, 12 minutes, and 17 seconds. With a battery of 10 000 mAh, the enhancement is 321.83 % from three hours, 31 minutes, and 12 seconds to 13 hours, 19 minutes, and 43 seconds. In the next section 6.3, there is a description of the DFVS technique applied to the eBridge Network.

Table 6.14: Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2021 code as slave/secondary in eBridge 2021 HW configuration.

Type of Configuration:	Measurement Frequency = 1 min Message Frequency = 5 min	
Device ID	**Battery ID**	**Total Run Time [HH:MM:SS]**
RPI3-2	5 200 mAh - 1	06:12:17
RPI3-2	10 000 mAh -1	13:19:43
RPI4-2	5 200 mAh - 3	05:22:29
RPI4-2	10 000 mAh -3	10:12:04

6.3 Dynamic and Voltage and Frequency Scaling

As described in [71], the RPI4 firmware includes the possibility of executing DVFS in the Broadcom BCM2711 SoC compared to the Broadcom BCM2837B0 SoC based on the documentation at [72]. However, the experiments performed on both RPI3 B+ and RPI4 B proved that both SoC could do DVFS. It was possible to observe in figure 6.15 how the Raspberry PI OS CPU governor can change the frequency of the minimum default setpoint to the maximum and return to the minimum in the second form of an RPI 3B+ running as master/primary during the configuration time of the node. This same behavior can be seen in all of the nodes during the configuration section and during the execution of other functions like **Get Median ID:4** τ_4, and **Generate Report ID:3** τ_3.

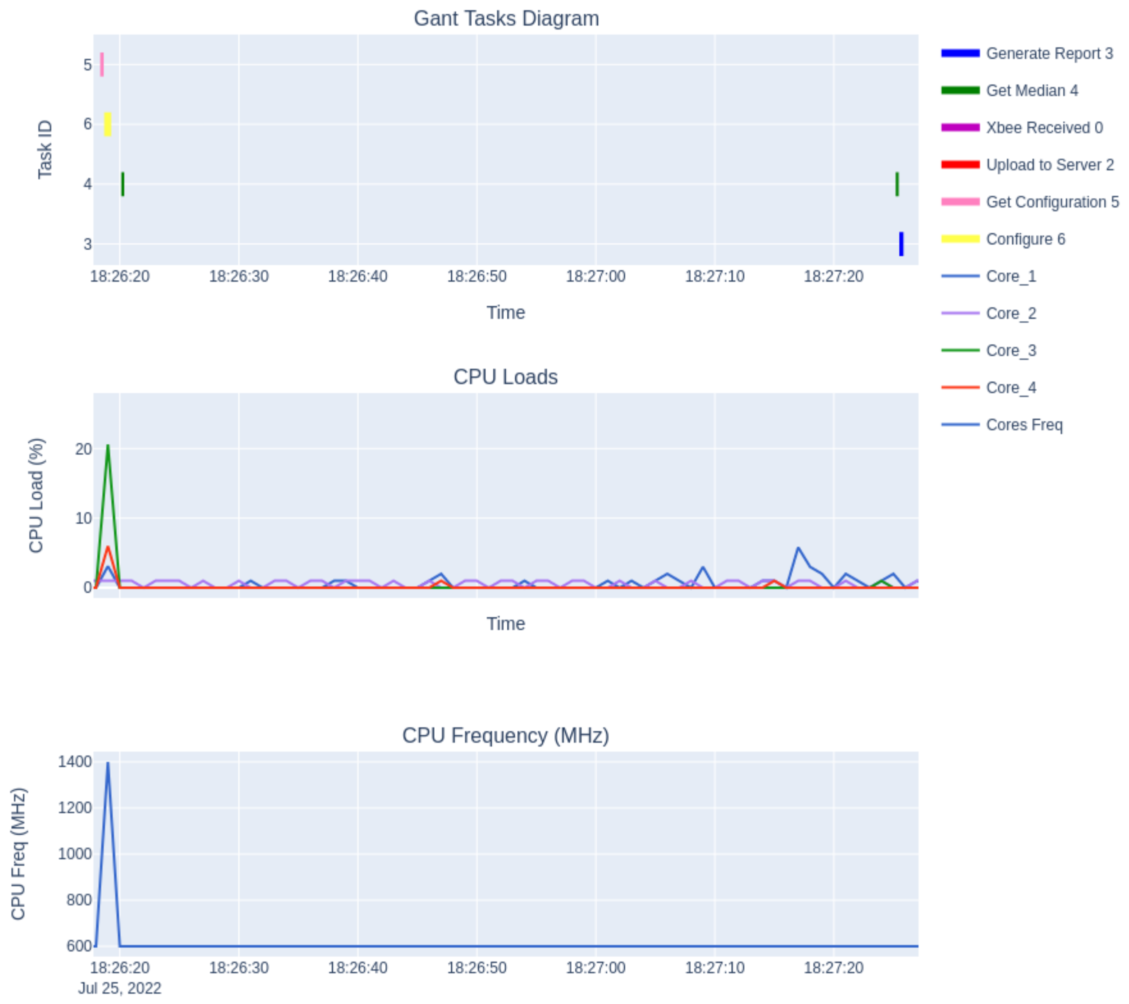


Figure 6.15: Tasks Gantt Plot vs CPU Load (%) vs CPU Freq (MHz) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI3 B+ with DVFS, focused on Configuration τ_5,and τ_6.

In figure 6.15 the CPU Load (%) does not go over 20 % for the configuration, and 12 % on the data acquiring and report generation events. However, the CPU governor stills bring the frequency to the top value of 1.4 GHz for an RPI3 B+ node. The same behavior can be observed in figure 6.16 for a master/primary code running the eBridge 2021 optimized code in an RPI4 B node; however, this time the frequency goes up to 1.5 GHz. In the following subsection 6.3.1 there is a detailed explanation of the effects of performing DFS on the eBridge Code.

6.3.1 Effects of Dynamic Frequency Scaling

This section covers the effects of DFS on the eBridge 2021 optimized code in both RPI3 B+ and RPI4 B nodes. In figure 6.16, it is possible to see the effect of the DFS in the power consumption during the configuration event. There are two power spikes in the plot, one of around 3.5 W that is aligned with the frequency shift from 600 MHz to 1.5 GHz, and another one of around 4 W that happens right after the first execution of the **Get Median ID:4**τ_4 function. This 4W spike is caused due to the launch of the **Device.Run()** and the **Receive** threads described in section 4.1.1. After this is possible to see how the power stabilizes around 3.3 W, which is the average power consumption of the master/primary node running the eBridge 2021 optimized code as shown in table 6.4. For the RPI3 B+, it is possible to set frequencies from 600 MHz to 1.4 GHz. It is also possible to reduce this value even further, but it can cause instability. The lowest it was possible to run the eBridge 2021 Optimized code was 400 MHz. In the case of the RPI4 B, typical frequency values go from 600 MHz to 1.5 GHz. In the case of RPI4 B, it was also possible to test the eBridge 2021 optimized with frequencies down to 400 MHz. As shown in table 6.15, the standby means power value is -5.61 % for an RPI3 B+ and -4 % for an RPI4 B compared to the default standby mean power. In the next section 6.3.2, the effects of doing CPU voltage changes can be seen.

A similar behavior can be seen in slave/secondary nodes when executing the **XBee Send ID:1** τ_1 function in a RPI3 B+. As it is shown in figure 6.17, where after having the frequency shift from 1.4 GHz to 600 MHz, there is a small power drop to around 2.9 W.as it would be further discussed in section 6.1, it is hard to notice the effect of DFS when having low CPU utilization. In the next section 6.3.2, the effects of changing the voltage of the Cortex-A53 and Cortex-A72 for the RPI3 B+ and RPI4 B are explored.

6.3.2 CPU Voltage Changes

As it was mentioned in section 5.3.2, the DVFS scaling of the RPI nodes can be enabled by using the **dvfs** flag on the **/boot/config.txt** file. When using a value of **dvfs=1**, it is possible to enable the undervoltage feature of the RPI's Broadcom SoCs. Changing the Cortex-A53 and Cortex-A72 core voltages is to use the **overvoltage**, which could take values from -16 to 8 [71]; these represent voltage values from 0.8 V to 1.4 V with increments of 0.025 V. Applying the undervoltage settings to RPI3 B+ and RPI4 B nodes compared to the node standby power mean power can be seen in table 6.16. It can be observed that the maximum mean power reduction can be obtained with the **-16** undervoltage setting for the RPI3 B+ and **-12** for the RPI4 B node. With these settings of **-16**, for the RPI3 B+ and **-12** for the RPI4 B, on the **overclock** variable it is possible obtain a -6.48 % reduction for RPI3 B+, and a -7.62 % for RPI4 B. In the next section 6.4 the details of applying the DPM technique on the eBridge are explored.

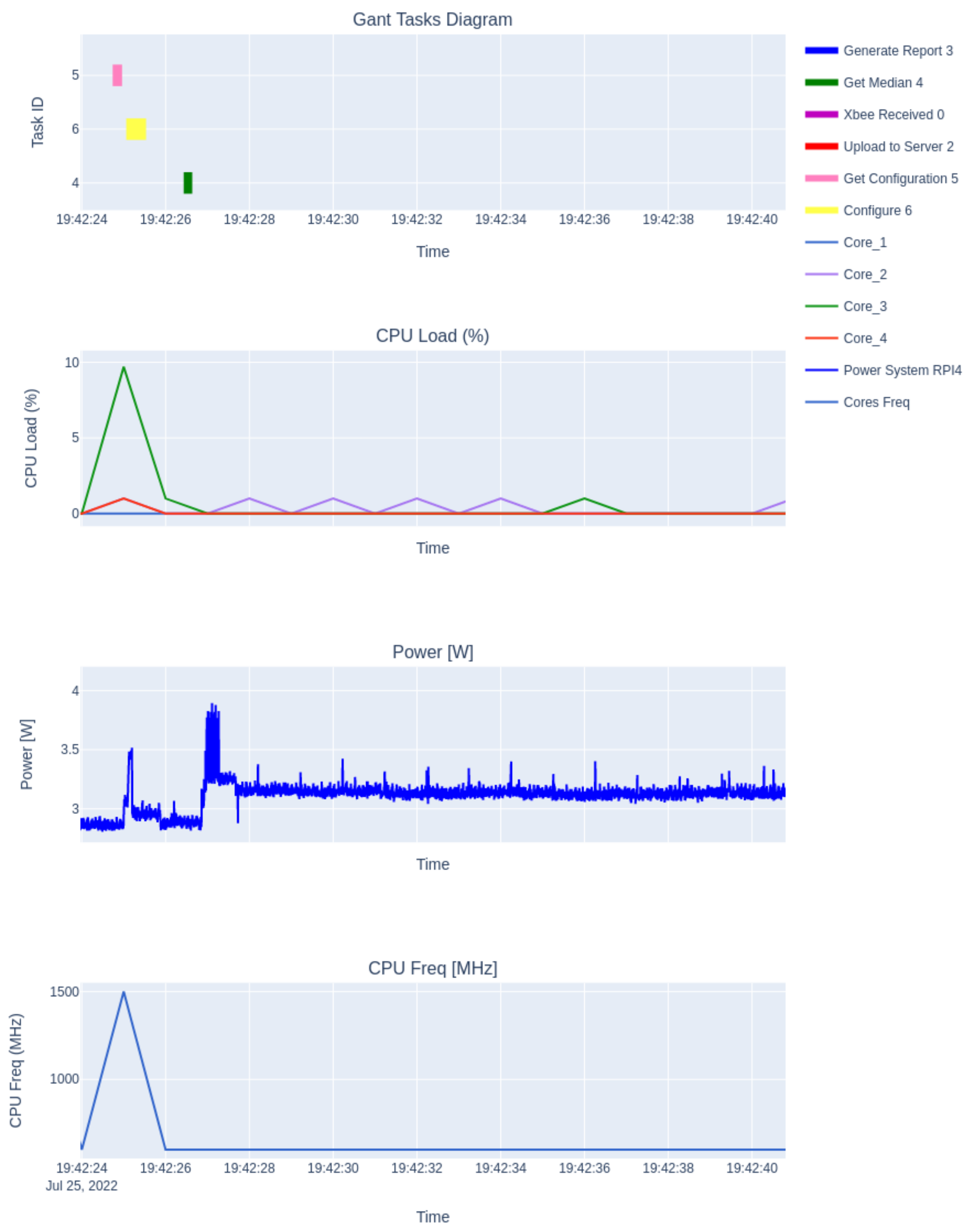


Figure 6.16: Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2021 code running as master/primary sensor node running in eBridge 2021 HW with RPI4 B with DVFS, focused on Configuration τ_5,and τ_6

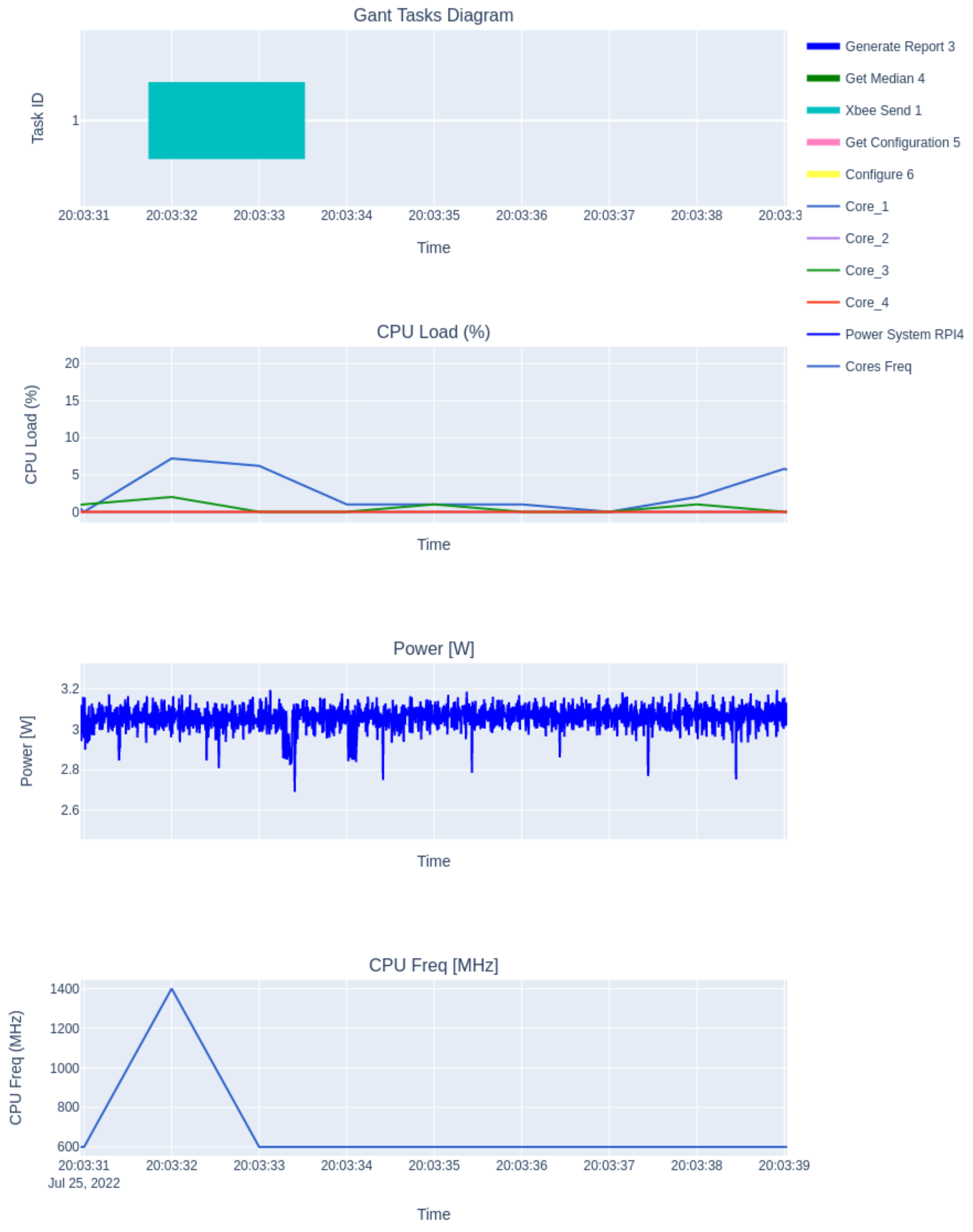


Figure 6.17: Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2021 code running as a slave/secondary sensor node running in eBridge 2021 HW with RPI3 B with DVFS, focused on XBee Send τ_1.

Table 6.15: Mean Power [W] of RPI3 B+ and RPI4 B nodes after applying DFS settings compared to standby power

Total Test Run Time (s)	60			
Device for Test	PXI-4309			
Sampling Rate (S/s)	20 000			
Frequency Settings	**Mean Power [W]**			
	RPI3 B+	**Change (%) vs Standby**	**RPI4 B**	**Change (%) vs Standby**
Default Settings n = 7 200 000 1.4 GHz for RPI3 B+ 1.5 GHz for RPI4 B	3.0402	-	3.0622	-
600 MHz n = 1 220 000	2.8765	-5.38	2.9487	-3.71
400 MHz n = 1 220 000	2.8695	-5.61	2.9397	-4

Table 6.16: Mean Power [W] of RPI3 B+ and RPI4 B nodes after applying DVS undervoltage settings compared to standby power

Total Test Run Time (s)	60			
Device for Test	PXI-4309			
Sampling Rate (S/s)	20 000			
CPU Voltage Setting [-16,8]	**Mean Power [W]**			
	RPI3 B+	**Change (%) vs Standby**	**RPI4 B**	**Change (%) vs Standby**
Default Settings n = 7 200 000	3.0402	-	3.0622	-
-2 n = 1 220 000	2.9423	- 3.22	2.9455	-3.81
-4 n = 1 220 000	2.9072	-4.37	2.8700	-6.28
-8 n = 1 520 000	2.881	-5.24	2.8852	-5.78
-12 n = 1 460 000	2.8432	- 6.48	2.8290	-7.62
-16 n = 1 240 000	2.8403	-6.58	2.8457	-7.19

6.4 Dynamic Power Management

As described in section 5.3.2, several hardware components of the RPI nodes were turned off to save power. This section describes of the effects these actions had on the system's standby power. These settings could be easily controllable from the eBridge code if necessary. However, in the current implementation, all of these hardware modules are powered down, except for the USB interface, which is handled by the **Cell_Interface** class.

- **RPI WiFi interface**: The RPI3 B+ and RPI4 B WiFi chips can be powered down using the **dtoverlay=pi3-disable-wifi** flag on the **/boot/config.txt** file.
- **RPI Bluetooth interface**: It can also be turned off for both nodes using the **dtoverlay=pi3-disable-bt** flag on the **/boot/config.txt** file as well.
- **RPI Ethernet ports**: The network interfaces of the RPI can be turned off with the command **sudo ifconfig [interface name] down**. This is used by the **Cell_Interface** class to turn on and off the wwan0 interface used by the LTE communication board.
- **RPI LEDS**: A total of six LEDs, including the two Ethernet interface LEDs, the trigger, and the power LED. These can be turned off from the **/boot/config.txt** file with the following flags: **eth_led0**, **eth_led1**, **act_led_trigger**, **act_led_activelow**, **pwr_led_trigger**, and **pwr_led_activelow**.
- **RPI HDMI interface**: Using the command **sudo tvservice –off**.
- **RPI USB Interface**: It can be turned down with command **echo [interface number] — sudo tee /sys/bus/usb/drivers/usb/unbind**.

The effects on the mean power value for both RPI3 B+ and RPI4 B nodes can be seen in table 6.17; with all the DPM settings applied, it was possible to obtain a reduction of 14.01 % mean standby power for the RPI3B+, and one of 14.95 for the RPI4B node. Out off these settings, the powering down of the USB interface was the one that caused the most significant reduction; for both nodes, it was over a 12 % reduction in mean power. In the next section 6.5, there is a detailed description of the eBridge 2022 implementation with the LoRa communication board, its impact on CPU Load, and power consumption for both master/primary and slave/secondary nodes. This data is also compared to the performance of the eBridge 2021 optimized code with the XBee communication board.

6.5 eBridge 2022 Node with LoRa Analysis

This section covers the experiments conducted on the eBridge 2022 version A and B HW configurations on both RPI3 B+ and RPI4 B nodes running as master/primary and slave/secondary. In the first subsection 6.5.1, there is a description of the additional functions to monitor on the eBridge 2022 code, in 6.5.2 is possible to find the results for a master/primary node, and finally in 6.5.3 the ones for slave/secondary.

6.5.1 eBridge 2022 with LoRa Line Profiling

Aside from the functions previously monitored for the eBridge 2019 and the 2021 Optimized code, it was also important to profile the functions below:

- **RFM9X Send ID:9** $\tau_1 0$: This function is responsible for sending the eBridge reports from slave/secondary nodes to the master/primary node using the RFM95W LoRa module.
- **RFM9X Received ID:10** τ_9: This function is in charge of receiving messages from a master/primary node over the LoRa RFM95W module in a slave/secondary node.

The CPU and Events profiling results for a master/primary node are discussed in the following subsection.

Table 6.17: Mean Power [W] of RPI3 B+ and RPI4 B nodes after applying DPM settings compared to standby power

Total Test Run Time (s)	60			
Device for Test	PXI-4309			
Sampling Rate (S/s)	20 000			
DPM Setting [-16,8]	**Mean Power [W]**			
	RPI3 B+	**Change (%) vs Standby**	**RPI4 B**	**Change (%) vs Standby**
Default Settings n = 7 200 000	3.0402	-	3.0622	-
WiFi and Bluetooth Interfaces n = 1 260 000	3.0362	-0.13	2.8585	-6.65
wwan0 n = 7 200 000	2.9902	-1.64	3.0477	-0.47
LEDs n = 1 280 000	2.8447	-6.43	2.8694	-6.3
HDMI n = 1 460 000	2.9747	-2.15	2.9598	-3.34
USB Interface n = 7 200 000	2.6632	-12.4	2.6959	-11.96
All Settings n = 1 320 000	2.6142	-14.01	2.6043	-14.95

6.5.2 eBridge 2022 with LoRa Master/Primary CPU and Events Profiling

The details of the tests performed for eBridge 2022 master/primary nodes on PRI3B+ can be found in the table 6.18. The 2022 HW configuration for the master/primary node with the RFM9X LoRa module is used for this test. The total test time was two hours, and it implemented DFVS with a maximum frequency of 1.4 GHz and a minimum of 600 MHz. Also, it included the DPM techniques shown in section 6.4.

Table 6.18: Measurement Summary eBridge code 2022 running as Master/Primary Sensor node in eBridge 2022 hardware configuration A on RPI3 B+

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2022 HW A with RPI3 B+
Controller	Raspberry PI 3 B+
Device ID	RPI3-2
Code Running	eBridge 2022 code
Cellular Communications Board	SIM7600G-H
LoRa Board	RFM95W
Type of the Device	Master/Primary
Total Test Duration	2 hours and 42 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples by CPU Core	7196 samples
CPU Frequency applied to all cores(f)	DVFS enabled
Power Measurement Summary	
Power Source	CanaKit RPI3 B+ 5V 2.5A [59]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	14 392 000

Figure 6.18 it is possible to see the relationship of the Task Gantt plot, the CPU Load (%), the Power, and the CPU Frequency in MHz for the RPI3 B+ running as master/primary with the 2022 eBridge code. It is possible to see how every time there is a Frequency change; it causes and power spike that could go up to 3.5 W.

When further reviewing the power values for the $\Gamma = \{\tau_8, \tau_7, \tau_4, \tau_3, \tau_2, \tau_{10}\}$ tasks it is possible to come up with the $\vec{E_i^f}$ vector's statistical information in table 6.19. From these

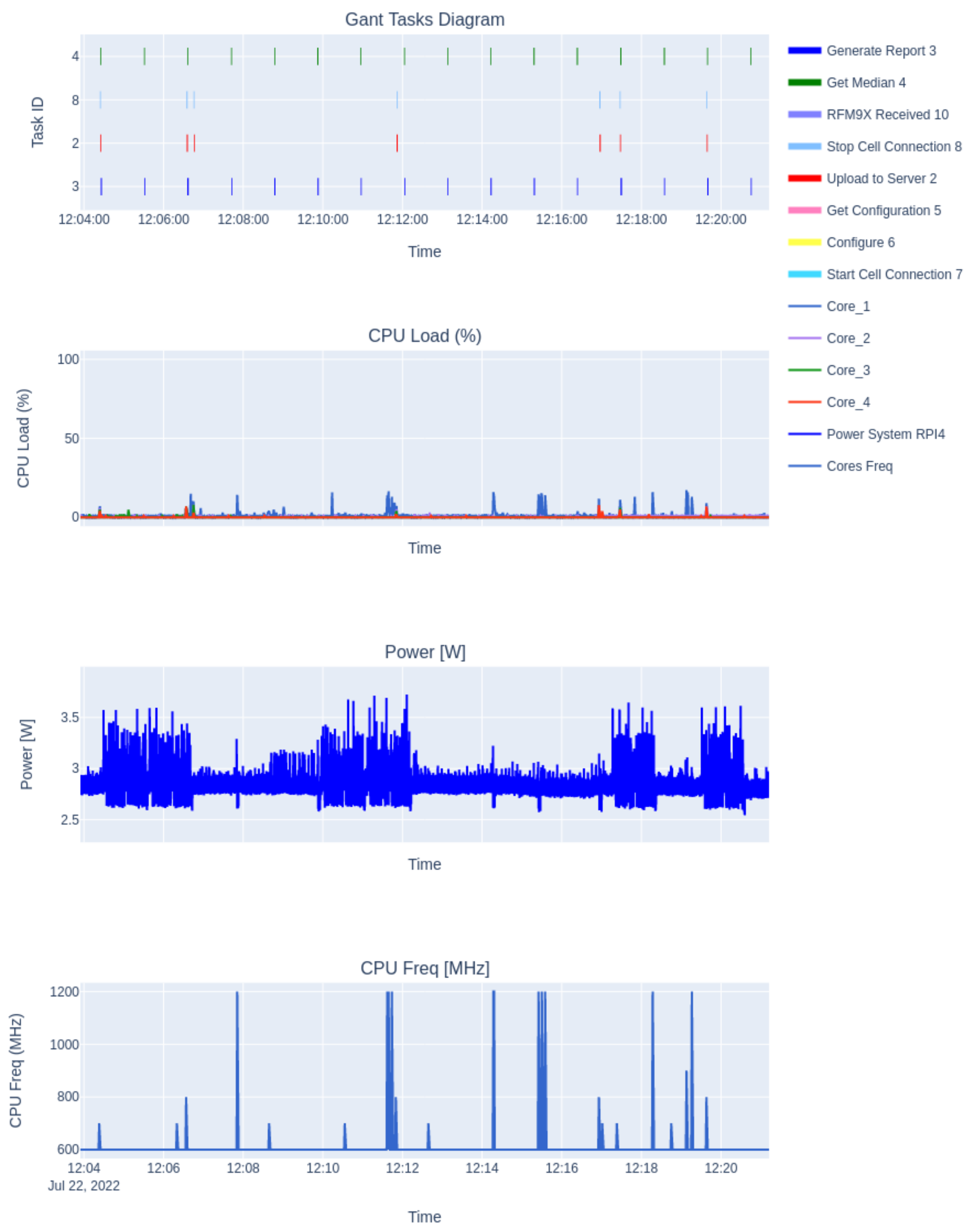


Figure 6.18: Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2022 code running as a master/primary sensor node running in eBridge 2022 HW with RPI3 B+ with DVFS and DPM.

data, the highest power spike is generated by the **Get Median ID:4** τ_4 task with a value of 3.9108 W. Also, it can be observed how the mean power of 2.8503 W of the task

RFM9X Receive ID:10 τ_{10} is 23.85 % smaller than the one for **XBee Received ID:0** τ_0 of 3.7431 W. The mean power of this configuration of 2.8297 W is 25.33 % smaller than the 3.7897 W of the eBridge 2019 code and 8.27 % smaller than the 3.0848 W of the 2021 Optimized code with XBee.

Table 6.19: $\vec{E_i^f}$ with DVFS on RPI3 B+ vectors statistical information of $\Gamma = \{\tau_8, \tau_7, \tau_4, \tau_3, \tau_2, \tau_{10}\}$ of eBridge 2022 in eBridge 2022 HW configuration as master/primary and mean power. Test details in table 6.18.

Power [W]							
Task ID	τ_8	τ_7	τ_4	τ_3	τ_2	τ_{10}	Power
n	249 496	181 576	1 498 160	1 533 760	93 000	65 504	14 392 000
Mean	2.8283	2.8537	2.8295	2.8271	2.8190	2.8503	2.8297
Std	0.1194	0.1324	0.1130	0.1148	0.1055	0.1303	0.1155
Min	2.5608	2.3649	2.5314	2.5420	2.5764	2.6532	2.3588
Max	3.7603	3.5845	3.9108	3.7498	3.6913	3.3988	3.9902

The $\vec{C_i^f}$ vector statistical information on table 6.20, shows the average execution time of the tasks of interest for the 2022 eBridge code with LoRa. From the perspective of the master/primary node, receiving LoRa messages takes less time than receiving from the XBee DigiMesh board. The average execution time of task **RFM9X Receive ID:10** τ_{10} is 4.1 ms, while for **XBee Receive ID:0** τ_0 the mean execution code for the optimized code is of 4.9 ms, however the maximum execution time for τ_{10} is 10.22 ms, while for τ_0 is 50.20 ms.

Table 6.20: $\vec{C_i^f}$ with DVFS on RPI3 B+ vectors statistical information of $\Gamma = \{\tau_8, \tau_7, \tau_4, \tau_3, \tau_2, \tau_{10}\}$ of eBridge 2022 in eBridge 2022 HW configuration as master/primary. Test details in table 6.18.

Execution Time (s)						
Task ID	τ_8	τ_7	τ_4	τ_3	τ_2	τ_{10}
n	49	49	117	116	48	48
Mean	0.2263	0.2229	0.0036	0.1089	0.0395	0.0041
Std	0.0051	0.0328	0.0011	0.2065	0.0053	0.0041
Min	0.2227	0.1212	0.0026	0.0008	0.0231	0.0004
Max	2.2482	0.2428	0.0095	0.5434	0.0499	0.01022

In the next subsection 6.5.3 the results of the eBridge 2022 code running in RPI4B as slave/secondary node is discussed.

6.5.3 eBridge 2022 with LoRa Slave/Secondary CPU and Events Profiling

This section includes the results of the test described in table 6.21, which runs the eBridge 2022 code with a LoRa communication node in an RPI4 B node with the 2022 HW configuration for slave/secondary systems. This test ran for 2 hours and 47 seconds with the DPM setting described in section 6.4. Also, the node had DVFS enabled with a range of frequencies from 600 MHz to 1.5 GHz.

Table 6.21: Measurement Summary eBridge code 2022 running as Master/Primary Sensor node in eBridge 2022 hardware configuration A on RPI3 B+

CPU and Events Measurement Summary	
Hardware Configuration	eBridge 2022 HW A with RPI4 B
Controller	Raspberry PI 4 B
Device ID	RPI4-2
Code Running	eBridge 2022 code
Cellular Communications Board	SIM7600G-H
LoRa Board	RFM95W
Type of the Device	Slave/Secondary
Total Test Duration	2 hours and 42 seconds
System Configuration	Message Frequency: 5 min Measurement Frequency: 1 min
Number of Cores	4
Number of Samples per second per CPU (S/s)	1
Number of Samples by CPU Core	7208 samples
CPU Frequency applied to all cores(f)	DVFS enabled
Power Measurement Summary	
Power Source	CanaKit RPI3 B+ 5V 3.5A [60]
Instrument	PXIe-4309
Shunt Resistor	$75mV/100A$ (0.075 $m\Omega$)
Number of Channels	2
Channel names and IDs	Power Supply Voltage: V_{sup} Shunt Resistor Voltage: V_{shunt}
Number of Samples per Second (S/s)	2000
Total number of samples per channel	14 416 000

Figure 6.19 shows the relationship between the Gantt Plot of tasks execution, the CPU Load (%), the power consumption, and the Core Frequency in MHz for the 2022 code running in RPI4 B as a slave/secondary. From this plot is possible to see how on every execution of the **RFM9X Send ID:9** τ_9, there is a transient increase of CPU load up to 100 % utilization, and as a consequence, a frequency change to 1.5 GHz. One of the challenges of using the RFM95W LoRa communications module could be retrying to send

the messages if it does not get confirmation from the receptor node, as seen in the last execution of the τ_9 task in the figure.

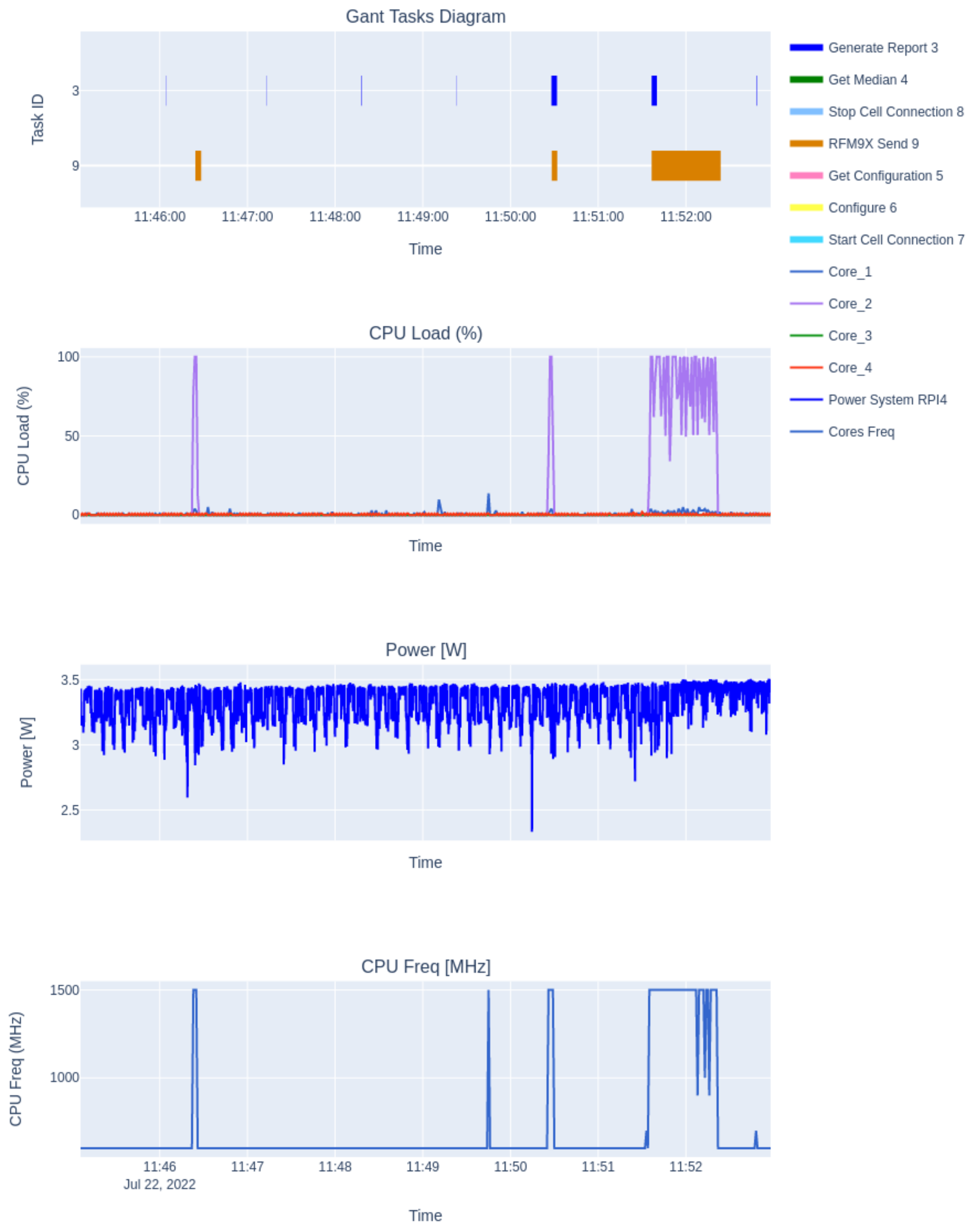


Figure 6.19: Tasks Gantt Plot vs CPU Load (%) vs Power [W] vs CPU Freq (MHz) eBridge 2022 code running as a slave/secondary sensor node running in eBridge 2022 HW with RPI4B with DVFS and DPM.

The data in table 6.22 there is a summary of $\vec{E_i^f}$ vector's statistical data for $\Gamma = \{\tau_4, \tau_3, \tau_9\}$ tasks in slave/secondary node running in a RPI4 B node. It is possible to observe how sending LoRa messages is 6.4 % energetically cheaper than sending XBee messages; the mean power value for **RFM9X Send ID:9** is 3.4270, while the one for **XBee Received ID:1** τ_1 is 3.6615 W. The Start Cell Connection - ID 7 and Stop Cell Connection - ID 8 are not analyzed since they are executed only once during the node initialization.

Table 6.22: $\vec{E_i^f}$ with DVFS on RPI4 B vectors statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_9\}$ of eBridge 2022 in eBridge 2022 HW configuration as slave/secondary and mean power. Test details in table 6.21.

Power [W]				
Task ID	τ_4	τ_3	τ_9	Power
n	1 433 496	1 455 744	716 192	14 416 000
Mean	3.6709	3.6728	3.6778	3.172
Std	0.0816	0.0786	0.0727	0.0075
Min	2.4999	2.6734	2.6828	2.2010
Max	3.8006	3.7900	3.7963	3.5577

The statistical data of the execution profiles for tasks $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ can be seen in table 6.23. It is possible to see how the slave/secondary node takes a long time of up to 64 s to send a LoRa message compared to the 0.3395 s execution time of the **XBee Receive ID:0** τ_0. Looking for a way to improve the LoRa communication could be useful in reducing the mean CPU Load of the node.

Table 6.23: $\vec{E_i^f}$ with DVFS and DPM in PI3 B+vectors' statistical information of $\Gamma = \{\tau_4, \tau_3, \tau_1\}$ of eBridge 2021 Optimized code in eBridge 2021 HW configuration as slave/secondary. Test details in tables 6.1 and 6.2

Execution Time (s)			
Task ID	τ_4	τ_3	τ_9
n	88	87	37
Mean	0.0020	10.510	37.00
Std	0.0003	23.583	53.76
Min	0.0016	0.0005	3.30
Max	0.0033	64.03	64.35

The mean power for the slave/secondary 2022 code with LoRa of 3.172 W is 7.89 % smaller than the 3.4224 W for the optimized code in an RPI4 B node. The following section summarizes the battery tests conducted on the eBridge 2022 code and 2022 HW configuration with the RFM9X LoRa communication board.

6.6 eBridge 2022 with LoRa Battery Tests

The eBridge 2022 network was tested using 5 200 mAh, and 10 000 mAh batteries on both RPI3 B+ and RPI4 B nodes; these batteries are described in subsection 3.5.1. These results can be seen in tables 6.24, and 6.25.

6.6.1 Battery test for Master/Primary

This section covers the battery tests performed on master/primary nodes using the RFM9X LoRa module in RPI3 B+ and PRI4 B nodes; the results are shown in table 6.24. It is possible to see a time improvement of 500 % using a 5 200 mAh battery from the 2019 eBridge code running in the 2021 HW configuration in an RPI3 B+; the time improved from 2 hours, 15 minutes, and 33 seconds to 13 hours, 33 minutes, and 30 seconds. For the battery of 10 000 mAh, the improvement was 574.74 % from three hours, 31 minutes, and 12 seconds to 23 hours, 45 minutes, and 3 seconds. For an RPI4 B with a battery of 5 200 mAh, the improvement was 281.69 % from one hour, 50 minutes and two seconds to six hours, 59 minutes, and 59 seconds. In the case of the 10 000 mAh, the enhancement was 347.94 % from three hours, 23 minutes, and 12 seconds to 15 hours, ten minutes, and 13 seconds.

Table 6.24: Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2022 code as master/primary in eBridge 2022 HW configuration.

Type of Configuration:	Measurement Frequency = 1 min Message Frequency = 5 min	
Device ID	**Battery ID**	**Total Run Time [HH:MM:SS]**
RPI3-2	5 200 mAh - 1	13:33:30
RPI3-2	10 000 mAh -1	23:45:03
RPI4-2	5 200 mAh - 3	06:59:59
RPI4-2	10 000 mAh -3	15:10:13

6.6.2 Battery test for Slave/Secondary

The battery tests results for the slave/secondary node using LoRa with the eBridge 2022 HW configuration in both RPI3 B+ and RPI4 can be seen in table 6.25. The battery life improvement for a slave in RPI3 B+ was 557.67 % with a battery of 5 200 mAh from one hour, 54 minutes, and 33 seconds to 12 hours, 33 minutes, and 22 seconds. For a battery of 10 0000 mAh, the enhancement was 586 % from three hours, eleven minutes, and two seconds to 21 hours, 50 minutes, and 34 seconds. For an RPI4 B node, it was 306.6 % from one hour, 37 minutes, 17 seconds to six hours, 35 minutes, and 33 seconds with a battery of 5 200 mAh, with a battery of 10 000 mAh, the total run time improved 414.86 % from two hours, 54 minutes, and 26 seconds to 14 hours, 58 minutes, and five seconds.

Table 6.25: Battery tests run time results on RPI3 B+ and RPI4 B of the eBridge 2022 code as slave/secondary in eBridge 2022 HW configuration.

Type of Configuration:	Measurement Frequency = 1 min Message Frequency = 5 min	
Device ID	**Battery ID**	**Total Run Time [HH:MM:SS]**
RPI3-2	5 200 mAh - 1	12:33:22
RPI3-2	10 000 mAh -1	21:50:34
RPI4-2	5 200 mAh - 3	06:35:33
RPI4-2	10 000 mAh -3	12:19:21
RPI4-2	10 000 mAh -2	14:58:05

In the next chapter 7 the conclusions of this thesis work are presented.

Chapter 7

Conclusions

This chapter is focused on presenting the main conclusions of the performed investigation, which was aimed at complying with the objectives described in chapter 1. This is divided into section 7.1 with the conclusions, and in section 7.2 future work is covered.

7.1 Conclusions

The battery life autonomy using a 5200 mAh improved from 2 hours, 15 minutes, and 33 seconds to 13 hours, 33 minutes, and 30 seconds using an RPI3 B+ node and the updated 2022 HW configuration in a master/primary device, and for a slave/secondary node the time improved from one hour, 49 minutes, and 16 seconds to 12 hours, 33 minutes and 22 seconds, thus complying with the general thesis objective.

7.1.1 eBridge 2019 network analysis Conclusions

The mean power of the eBridge 2019 code running as master/primary of 3.7897 W is 24.65 % higher than the standby power for an RPI3 B+ node of 3.0402 W. The one for an RPI4 B node of 4.5788 W is 49.53 % higher than the mean standby power of 3.0622 %. This means that for the same code and peripherals, the RPI4 B node consumes 20.82 % under the same stress conditions, while the standby power of RPI4 B is only 0.72 % while in standby. This is directly reflected in the battery life of both nodes in which the RPI3 B+ has 20.08 % and 20.07 % higher run time with a battery of 5 200 mAh, and a 21.08 % and 22.19 % with 10 000 mAh batteries than the RPI4B while running as master/primary. For a slave secondary node, the power consumption of RPI4 B nodes is 21.52 % higher.

7.1.2 Design Conclusions

Improving the eBridge **Alarm Handling** and **Messaging** protocols was key to reducing the power consumption of a slave/secondary system. The protocol changes reduced the messages per hour from 60 to 12 per node in case of no alarms for a measurement frequency of one minute and message frequency of five minutes. Based on [4], the size of each report is 200 bytes; this allowed a reduction in the data sent from a slave/secondary node to a master/primary of 80% from 11.72 KB to 2.34 KB per hour per node. Also, it reduced the number of letters to the eBridge server for a single configuration with one master/primary device and single slave/secondary device, and the same frequencies as before from 120 reports to 24. It was deducted that sending messages over XBee was 5.71 % more expensive energetically than receiving messages for an RPI3 B+ and 5.46 % for an RPI4 B node. Based on [73], the cost per KB is 1.95 CRC (Costa Rican Colones); this means a reduction of LTE cost from 45.7 CRC to 9.126 CRC per hour, which could represent up to 26 605.4 CRC per month.

7.1.3 eBridge 2021 network analysis Conclusions

Using an **Instruction-Level** power analysis allows for identifying focus areas in the software that enables the programmer to have a more significant impact on resource utilization; creating a task interaction model from this approach was vital in identifying areas of improvement. By performing software optimizations in the specified code modules, it was possible to improve mean power consumption by 18.6 % and 26 % for RPI3 B+ and RPI4, respectively, from values of 3.797 W to 3.0848 W and from 4.5788 W to 3.3781 W while running as master/primary. For a slave/secondary node, the improvement on RPI3 B+ was 23.01 % from 3.9663 W to 3.0768 W, and for RPI4 B was 28.58 % from 4.8200 W to 3.4224 W. Also, it was possible to reduce the maximum power spike on a slave/secondary system from a max of 7.5 W to 4.1431 W, which is a 41.16 % reduction.

7.1.4 DPM and DVFS on the eBridge Network Conclusions

The combination of DPM strategies on the RPI3 B+ and the RPI4 B node reduced the mean standby power of both nodes by 14.01 % and 14.95 %, respectively. The standby power for RPI3 B+ improved from 3.0402 W to 2.6142 with all the eBridge peripherals connected.

Applying Undervoltage settings to the Cortex-A53 on the RPI3 B+ and the Cortex-A72 on the RPI4 B with a value of -8 [1.0 V per Core] allows the standby power to be reduced by 5.24 % for RPI3 B+ and 5.78 & RPI4 B. This was the lowest setting in which it was possible to run the eBridge 2021 optimized code and the eBridge 2022 code with LoRa. This was combined with DVFS scaling allowing the CPU to lower its frequencies to 600 MHz while on standby.

7.1.5 LoRA eBridge 2022 with LoRa Conclusions

Implementing the LoRa RFM95W communication board on the eBrdige 2021 Optimized code allowed to reduce the mean power from Optimized code by 8.27 % for master/primary nodes from 3.0848 W to 2.8297 for RPI3 B+. This represents an improvement of 23.85 % compared to the eBridge 2019 code. For slave/secondary, the mean power was reduced by 7.89 % with the 2022 code with LoRa compared to the 2021 optimized code. This allowed an improvement in battery life by 500 % using a 5 200 mAh battery from the 2019 eBridge code running in the 2021 HW configuration in an RPI3 B+; the time improved from 2 hours, 15 minutes, and 33 seconds to 13 hours, 33 minutes, and 30 seconds. For the battery of 10 000 mAh, the improvement was 574.74 % from three hours, 31 minutes, and 12 seconds to 23 hours, 45 minutes, and 3 seconds. For an RPI4 B with a battery of 5 200 mAh, the improvement was 281.69 % from one hour, 50 minutes, and two seconds to six hours, 59 minutes, and 59 seconds. In the case of the 10 000 mAh, the enhancement was 347.94 % from three hours, 23 minutes, and 12 seconds to 15 hours, ten minutes, and 13 seconds.

7.2 Future Work

This section covers some suggestions for future work.

- Move the node to 10 000 mAh hour batteries; this would increase the battery life of the eBridge node up to almost 24 hours using RPI3 B+ nodes and leave room for expandability.
- Consider the possibility of moving away from using the LTE network for a LoRaWAN solution when more gateways are available.
- Work on looking for alternative LoRa communication boards that may require less CPU utilization.
- Implement a power solution with a solar panel to facilitate the nodes' deployability.
- Move to a more minor embedded system. A suitable replacement now that the eBridge for the RPI3 B+ and RPI4 B is the RaspberryPi Pico, which uses the RP2040 SoC [74] in combination with the SIM7080G Narrow Band IoT module from Waveshare [75], which works over the LTE network. This would have a total cost of under 50 USD and is compatible with MicroPython [76], a programming language highly compatible with Python3, making it easy to learn for new programmers.

Bibliography

[1] B. Seetanah, S. Ramessur, and S. Rojid, "Does Infrastructure Alleviates Poverty in Developing Countries," *International Journal of Applied Econometrics and Quantitative Studies*, vol. 9, no. 2, 2009, ISSN: 1988-0081 (cit. on p. 1).

[2] G. Ortiz, "eBridge: Predicción remota de fallas en puentes," *Investiga TEC*, pp. 10–11, 2012 (cit. on pp. 1, 2).

[3] Tecnológico De Costa Rica, "Inventario de puentes en rutas nacionales de Costa Rica 2014-201," Tech. Rep. 506, 2019. [Online]. Available: `https://www.tec.ac.cr/sites/default/files/media/doc/informe_final_inventario_y_evaluacion_puentes_-_2014-2018.pdf` (cit. on pp. 1, 2, 71, 73).

[4] A. Ruiz, "Diseño de redes de sensores colaborativas para monitorización de salud estructural de puentes," M.S. thesis, Tecnológico de Costa Rica (TEC), 2019. [Online]. Available: `https://repositoriotec.tec.ac.cr/handle/2238/10700` (cit. on pp. 1–3, 27, 34, 36–39, 41, 57, 67, 112).

[5] C. Garita and G. Ortiz, "Integrando Información Estregégica para Monitoreo de Puentes Nacionales.pdf," in *III Jornadas Costarricenses de Investigación en Computación e Informática JoCICI*, Cartago, 2019. DOI: `https://doi.org/10.18845/mct.v0i0.4529`. [Online]. Available: `https://revistas.tec.ac.cr/index.php/memorias/article/view/4529` (cit. on p. 2).

[6] C. Garita, "Vista de Enfoques de integración de información para sistemas de monitoreo de salud estructural de puentes.pdf," *Tecnología en Marcha*, pp. 96–107, 2015 (cit. on p. 2).

[7] A. Obando and C. Garita, "Diseño general de una red inalámbrica para monitoreo de salud de puentes," San Salvador, El Salvador, Sep. 2015. [Online]. Available: `https://www.researchgate.net/publication/282295016_Diseno_General_de_una_Red_Inalambrica_para_Monitoreo_de_Salud_de_Puentes` (cit. on p. 2).

[8] F. Picado-Alvarado and G. Ortiz-Quesada, "Desarrollo de un modelo de confiabilidad para el análisis del desempeño de puentes. Un caso de estudio en Costa Rica," *Revista Tecnología en Marcha*, vol. 30, no. 1, p. 79, 2017, ISSN: 0379-3982. DOI: `10.18845/tm.v30i1.3087` (cit. on p. 2).

[9] G. Ortiz and J. Mora, "Requirements Analysis for a National Bridge Monitoring System," *Revista Tecnología en Marcha*, vol. 31, no. 4, pp. 63–72, 2018 (cit. on p. 2).

[10] M. C. Abarca, A. R. Barquero, C. Garita, and G. Ortiz, "Preliminary design of a low-cost water level monitoring system for bridges," *Proceedings of the 2018 IEEE 38th Central America and Panama Convention, CONCAPAN 2018*, 2018. DOI: 10.1109/CONCAPAN.2018.8596666 (cit. on pp. 2, 3, 24).

[11] M. Gutiérrez and C. Garita, "Prototype development of a wireless embedded system for bridge monitoring," *2017 IEEE 37th Central America and Panama Convention, CONCAPAN 2017*, vol. 2018-January, pp. 1–6, 2018. DOI: 10.1109/CONCAPAN.2017.8278482 (cit. on p. 2).

[12] A. Ruiz-Barquero, C. Garita, and G. Ortiz, "Collaborative Sensors Networks for Structural Health Monitoring of Bridges in Costa Rica," in *XLVI Latin American Computing Conference (CLEI)*, 2021, pp. 417–426, ISBN: 9781665415606. DOI: 10.1109/clei52000.2020.00055 (cit. on pp. 2–4, 24).

[13] G. Ortiz-quesada and C. Garita-rodríguez, "Priorización de intervenciones en puentes utilizando indicadores Indicators for Bridge Actions Prioritization," vol. 34, pp. 134–142, 2021 (cit. on p. 2).

[14] Digi, *Digi xbee® 802.15.4*, Accessed on 2021-7-25, 2021. [Online]. Available: https://www.digi.com/products/embedded-systems/digi-xbee/rf-modules/2-4-ghz-rf-modules/xbee-802-15-4#specifications (cit. on pp. 3, 55, 65).

[15] M. Engin, "Energy Efficiency of Embedded Controllers," *2019 8th Mediterranean Conference on Embedded Computing, MECO 2019 - Proceedings*, no. June, pp. 1–4, 2019. DOI: 10.1109/MECO.2019.8760289 (cit. on pp. 5–10).

[16] J. Boudjadar, "An efficient energy-driven scheduling of DVFS-multicore systems with a hierarchy of shared memories," *Proceedings - 2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real-Time Applications, DS-RT 2017*, vol. 2017-Janua, pp. 1–8, 2017. DOI: 10.1109/DISTRA.2017.8167661 (cit. on pp. 6, 14, 16, 17).

[17] C. Deng, R. Guo, H. Wang, and A. Peng, "A dynamic power management algorithm for sporadic tasks in real-time embedded systems," *Proceedings - 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 10th IEEE International Conference on Big Data Science and Engineering and 14th IEEE International Symposium on Parallel and Distributed Proce*, pp. 2073–2078, 2016. DOI: 10.1109/TrustCom.2016.0318 (cit. on pp. 6, 7, 13, 20, 21).

[18] S. A. Carvalho, D. C. Cunha, and A. G. Silva-Filho, "Autonomous Power Management for Embedded Systems Using a Non-linear Power Predictor," *Proceedings - 20th Euromicro Conference on Digital System Design, DSD 2017*, pp. 22–29, 2017. DOI: 10.1109/DSD.2017.68 (cit. on pp. 6, 7, 12, 14, 15).

[19] K. M. Cho, C. H. Liang, J. Y. Huang, and C. S. Yang, "Design and implementation of a general purpose power-saving scheduling algorithm for embedded systems," *2011 IEEE International Conference on Signal Processing, Communications and Computing, ICSPCC 2011*, pp. 1–5, 2011. DOI: 10.1109/ICSPCC.2011.6061645 (cit. on pp. 6, 14).

[20] S. Moulik, A. Sarkar, and H. K. Kapoor, "Energy aware frame based fair scheduling," *Sustainable Computing: Informatics and Systems*, vol. 18, pp. 66–77, 2018, ISSN: 22105379. DOI: 10.1016/j.suscom.2018.03.003. [Online]. Available: https://doi.org/10.1016/j.suscom.2018.03.003 (cit. on pp. 6, 14, 18, 19, 21).

[21] S. Moulik, R. Chaudhary, and Z. Das, "HEARS: A heterogeneous energy-aware real-time scheduler," *Microprocessors and Microsystems*, vol. 72, p. 102 939, 2020. DOI: 10.1016/j.micpro.2019.102939. [Online]. Available: https://doi.org/10.1016/j.micpro.2019.102939 (cit. on pp. 6, 14, 20, 21).

[22] I. Ahmad, H. F. Sheikh, and A. Aved, "Benchmarking the task scheduling algorithms for performance, energy, and temperature optimization," *Sustainable Computing: Informatics and Systems*, vol. 25, p. 100 339, 2020, ISSN: 22105379. DOI: 10.1016/j.suscom.2019.07.002. [Online]. Available: https://doi.org/10.1016/j.suscom.2019.07.002 (cit. on pp. 6, 14, 16, 21).

[23] A. Potsch, A. Berger, and A. Springer, "Efficient analysis of power consumption behaviour of embedded wireless IoT systems," *I2MTC 2017 - 2017 IEEE International Instrumentation and Measurement Technology Conference, Proceedings*, pp. 1–5, 2017. DOI: 10.1109/I2MTC.2017.7969658 (cit. on pp. 7, 11).

[24] M. A. Awan, P. M. Yomsi, G. Nelissen, and S. M. Petters, "Energy-aware task mapping onto heterogeneous platforms using DVFS and sleep states," *Real-Time Systems*, vol. 52, no. 4, pp. 450–485, 2016, ISSN: 15731383. DOI: 10.1007/s11241-015-9236-x (cit. on pp. 7, 10, 14, 19–21, 31).

[25] J. Laurent, N. Julien, E. Senn, and E. Martin, "Functional level power analysis: An efficient approach for modeling the power consumption of complex processors," *Proceedings - Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 666–667, 2004. DOI: 10.1109/DATE.2004.1268921 (cit. on pp. 8–10, 12).

[26] W. Wang and P. Mishra, "Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems," *Proceedings of the IEEE International Conference on VLSI Design*, pp. 357–362, 2010, ISSN: 10639667. DOI: 10.1109/VLSI.Design.2010.22 (cit. on pp. 10, 17, 21).

[27] G. Chen, K. Huang, and A. Knoll, "Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 3s, 2014. DOI: 10.1145/2567935 (cit. on pp. 10, 14, 16, 18, 21).

[28] S. Z. Sheikh and M. A. Pasha, “An improved model for system-level energy minimization on real-time systems,” *Proceedings - IEEE Computer Society’s Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, vol. 2019-October, pp. 276–282, 2019, ISSN: 15267539. DOI: 10.1109/MASCOTS.2019.00039 (cit. on p. 10).

[29] X. Piao, H. Kim, Y. Cho, *et al.*, “Energy consumption optimization in real-time embedded systems,” *Proceedings - 2009 International Conference on Embedded Software and Systems, ICESS 2009*, pp. 281–287, 2009. DOI: 10.1109/ICESS.2009.43 (cit. on pp. 10, 11).

[30] I. Amirtharaj, “Energy Measurement and Profiling of Internet of Things Devices,” Computer Engineering Master’s Theses, Santa Clara University, 2018. [Online]. Available: https://scholarcommons.scu.edu/cseng_mstr/5 (cit. on pp. 11, 22).

[31] Eclipse Foundation, “2020 IoT Developer Survey Key Findings,” no. October, p. 31, 2020. [Online]. Available: https://f.hubspotusercontent10.net/hubfs/5413615/2020%20IoT%C2%A0Developer%20Survey%20Report.pdf (cit. on p. 11).

[32] K. Kesrouani, H. Kanso, and A. Noureddine, “A Preliminary Study of the Energy Impact of Software in Raspberry Pi devices,” *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE*, vol. 2020-September, pp. 231–234, 2020, ISSN: 15244547. DOI: 10.1109/WETICE49692.2020.00052 (cit. on p. 11).

[33] “Accurate Power Consumption Evaluation for Peripherals in Ultra Low-Power embedded systems,” *GIoTS 2020 - Global Internet of Things Summit, Proceedings*, pp. 3–8, 2020 (cit. on p. 11).

[34] C. Hou, Q. Zhao, and S. Member, “A New Optimal Algorithm for Energy Saving Embedded in Embedded System With Multiple Sleep Modes,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 2, pp. 706–719, 2016. DOI: 10.1109/TVLSI.2015.2414827 (cit. on p. 13).

[35] X. Zhang, L. Song, H. Sun, and G. Ye, “A user-configured low power optimizing mechanism on embedded system,” *Proceedings - 2013 International Conference on Computer Sciences and Applications, CSA 2013*, pp. 769–774, 2013. DOI: 10.1109/CSA.2013.184 (cit. on pp. 13, 14).

[36] M. Jafari-Nodoushan and A. Ejlali, “An optimal analytical solution for maximizing expected battery lifetime using the calculus of variations,” *Integration*, vol. 71, no. June 2019, pp. 86–94, 2020. DOI: 10.1016/j.vlsi.2019.11.002. [Online]. Available: https://doi.org/10.1016/j.vlsi.2019.11.002 (cit. on p. 14).

[37] U. U. Tariq, H. Ali, L. Liu, J. Panneerselvam, and X. Zhai, “Energy-efficient Static Task Scheduling on VFI-based NoC-HMPSoCs for Intelligent Edge Devices in Cyber-physical Systems,” *ACM Transactions on Intelligent Systems and Technology*, vol. 10, no. 6, pp. 1–23, 2019, ISSN: 21576912. DOI: 10.1145/3336121 (cit. on p. 14).

[38] A. Toma, S. Pagani, J. J. Chen, W. Karl, and J. Henkel, "An Energy-Efficient Middleware for Computation Offloading in Real-Time Embedded Systems," *Proceedings - 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2016*, pp. 228–237, 2016. DOI: 10.1109/RTCSA.2016.50 (cit. on p. 14).

[39] J. Peraza, A. Tiwari, M. Laurenzano, L. Carrington, and A. Snavely, *PMaC's green queue: a framework for selecting energy optimal DVFS configurations in large scale MPI applications*, 2016. DOI: 10.1002/cpe.3184 (cit. on p. 16).

[40] G. Luo, B. Guo, Y. Shen, H. Y. Liao, and L. Ren, "Analysis and optimization of embedded software energy consumption on the source code and algorithm level," *Proceedings of the 2009 4th International Conference on Embedded and Multimedia Computing, EM-Com 2009*, vol. 1, pp. 3–7, 2009. DOI: 10.1109/EM-COM.2009.5402965 (cit. on p. 22).

[41] R. P. Foundation, *Raspberry pi 4 model b specifications*, Accessed on 2021-8-1, 2021. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/ (cit. on p. 23).

[42] ——, *Sim7600g-h 4g hat for raspberry pi, lte cat-4 4g / 3g / 2g support, gnss positioning, global band*, Accessed on 2021-8-1, 2021. [Online]. Available: https://www.waveshare.com/sim7600g-h-4g-hat.htm (cit. on pp. 23, 25).

[43] P. J. Drongowski, *Raspberry pi 4 arm cortex-a72 processor*, Accessed on 2022-7-04, 2020. [Online]. Available: http://sandsoftwaresound.net/raspberry-pi-4-arm-cortex-a72-processor/ (cit. on p. 26).

[44] C. Windeck, *Raspberry pi 4 model b: Blockschaltbild des broadcom bcm2711*, Accessed on 2022-7-15, 2022. [Online]. Available: https://www.heise.de/ct/artikel/Raspberry-Pi-4-Model-B-Blockschaltbild-des-Broadcom-BCM2711-4514399.html?affiliateId=17957 (cit. on p. 27).

[45] Sci-Pi, *Pi 4 soc*, Accessed on 2022-7-15, 2022. [Online]. Available: https://www.sci-pi.org.uk/arch/soc.html (cit. on pp. 25, 28).

[46] L. Hattersley, *Raspberry pi 4 vs raspberry pi 3b+*, Accessed on 2021-10-22, 2019. [Online]. Available: https://magpi.raspberrypi.com/articles/raspberry-pi-4-vs-raspberry-pi-3b-plus (cit. on p. 25).

[47] N. Heath, *Raspberry pi 3 b+: Co-creator eben upton reveals all about the new board*, Accessed on 2022-4-05, 2018. [Online]. Available: https://www.techrepublic.com/article/raspberry-pi-3-b-co-creator-eben-upton-reveals-all-about-the-new-board (cit. on p. 27).

[48] MilhouseVH, *Bcmstat*, Accessed on 2021-2-6, 2020. [Online]. Available: https://github.com/MilhouseVH/bcmstat (cit. on p. 30).

[49] G. Rodola, *Cross-platform lib for process and system monitoring in python - psutil 5.8.0.* Accessed on 2021-2-7, 2020. [Online]. Available: https://pypi.org/project/psutil/ (cit. on p. 30).

[50] J. C. Warner, *Top*, Accessed on 2021-2-8. [Online]. Available: https://linux.die.net/man/1/top (cit. on p. 30).

[51] B. Frederickson, *Py-spy*, Accessed on 2021-2-7, 2021. [Online]. Available: https://github.com/benfred/py-spy (cit. on pp. 30, 32).

[52] E. Berger, *Scalene: A high-performance cpu and memory profiler for python*, Accessed on 2021-2-6, 2021. [Online]. Available: https://pypi.org/project/scalene/0.9.16/ (cit. on p. 30).

[53] P. S. Foundation, *Time access and conversions*, Accessed on 2021-3-5. [Online]. Available: https://docs.python.org/3/library/time.html (cit. on p. 30).

[54] A. Ajitsaria, *What is the python global interpreter lock (gil)?* Accessed on 2021-6-18, 2021. [Online]. Available: https://realpython.com/python-gil/ (cit. on pp. 31, 38).

[55] N. Instruments, *Pxie-4309 - specifications*, Accessed on 2022-7-5, 2022. [Online]. Available: https://www.ni.com/docs/en-US/bundle/pxie-4309-specs/resource/377030a.pdf (cit. on pp. 32, 33).

[56] M. Jungman, *Polymer li-ion battery technology specification - model icr18650 4400mah 3.7v*, Accessed on 2020-9-22, 2014. [Online]. Available: https://cdn-shop.adafruit.com/product-files/354/C449_-_ICR18650_4400mAh_3.7V_with_PCM_20140728_APPROVED_8.18.pdf (cit. on p. 34).

[57] M. S. CR, *Mi power bank and wall charger 5200 mah*, Accessed on 2022-6-30, 2022. [Online]. Available: https://mistorecr.com/producto/mi-power-bank-and-wall-charger-5200-mah/ (cit. on pp. 34, 35).

[58] ——, *Mi power bank 3 ultra compact 10000 mah*, Accessed on 2022-6-30, 2022. [Online]. Available: https://mistorecr.com/producto/mi-power-bank-3-ultra-compact-10000mah/ (cit. on p. 35).

[59] CanaKit, *Raspberry pi 3 model b+ starter kit*, Accessed on 2020-12-10, 2020. [Online]. Available: https://www.canakit.com/raspberry-pi-3-model-b-plus-starter-kit.html (cit. on pp. 39, 50, 63, 79, 88, 103).

[60] ——, *Raspberry pi 4 model b starter kit*, Accessed on 2020-12-10, 2020. [Online]. Available: https://www.canakit.com/raspberry-pi-4-starter-kit.html (cit. on pp. 44, 53, 63, 83, 91, 106).

[61] R. Zwetsloot, *Raspberry pi 4 specs and benchmarks*, Accessed on 2022-5-17, 2019. [Online]. Available: https://magpi.raspberrypi.com/articles/raspberry-pi-4-specs-benchmarks (cit. on p. 42).

[62] freedesktop.org, *What is libqmi?* Accessed on 2020-8-13, 2017. [Online]. Available: https://www.freedesktop.org/wiki/Software/libqmi/ (cit. on p. 65).

[63] Kolbi, *Planes de dispositivos*, Accessed on 2020-2-3, 2017. [Online]. Available: https://www.kolbi.cr/wps/portal/kolbi_dev/negocios/pymes/conectividad/planes-kolbi-dispositivos (cit. on p. 65).

[64] K. Yaghmour, J. Masters, and G. Ben, *Building Embedded Linux Systems, 2nd Edition*, Second. USA: O'Reilly, Associates, Inc., 2008, ISBN: 9780596529680 (cit. on p. 65).

[65] Python.org, *A synchronized queue class*, https://docs.python.org/3/library/queue.html, Accessed on 2021-2-23, 2021 (cit. on p. 66).

[66] S. de Telecomunicaciones SUTEL, *3g and 4g coverage costa rica by carrier*, Accessed on 2021-2-23, 2021. [Online]. Available: https://mapas.sutel.go.cr/ (cit. on p. 74).

[67] B. S. Chaudhari, M. Zennaro, and S. Borkar, "LPWAN technologies: Emerging application characteristics, requirements, and design considerations," *Future Internet*, vol. 12, no. 3, 2020, ISSN: 19995903. DOI: 10.3390/fi12030046 (cit. on pp. 71, 75).

[68] L. D. Portal, *Getting started with lora - what is lora?* Accessed on 2022-3-14, 2022. [Online]. Available: https://lora-developers.semtech.com/learn/get-started/what-is-lora (cit. on p. 72).

[69] M. Vargas, *The things network san josé, costa rica*, Accessed on 2022-5-27, 2022. [Online]. Available: https://www.thethingsnetwork.org/community/sanjose/ (cit. on p. 72).

[70] Adafruit, *Adafruit rfm95w lora radio transceiver breakout - 868 or 915 mhz - radiofruit*, Accessed on 2022-6-02, 2022. [Online]. Available: https://www.adafruit.com/product/3072 (cit. on pp. 72, 76).

[71] R. P. Ltd, *Raspberrypi documentation - frequency managment and thermal control*, Accessed on 2022-6-12, 2022. [Online]. Available: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#frequency-management-and-thermal-control (cit. on pp. 75, 95, 97).

[72] ——, *Raspberrypi documentation - processors*, Accessed on 2022-6-12, 2022. [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html (cit. on p. 95).

[73] Kolbi, *Precios de internet móvil como valor agregado*, Accessed on 2022-8-3, 2022. [Online]. Available: https://www.kolbi.cr/wps/portal/kolbidev/personas/postpago/informacion-postpago/tarifas-postpago (cit. on p. 112).

[74] J. Adams, *Raspberry pi rp2040: Our microcontroller for the masses*, Accessed on 2022-8-15, 2021. [Online]. Available: https://www.arm.com/blogs/blueprint/raspberry-pi-rp2040 (cit. on p. 113).

[75] Waveshare, *Sim7080g nb-iot / cat-m(emtc) / gnss module for raspberry pi pico, global band support*, Accessed on 2022-8-15, 2022. [Online]. Available: https://www.waveshare.com/Pico-SIM7080G-Cat-M-NB-IoT.htm (cit. on p. 113).

[76] MicroPython, *Micro python*, Accessed on 2022-8-15, 2022. [Online]. Available: https://micropython.org/ (cit. on p. 113).