UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTUDIO DE CONSUMO ENERGÉTICO PARA ALGORITMOS DE RECOLECCIÓN DE DATOS EN REDES DE SENSORES INALÁMBRICOS BAJO EL PROTOCOLO LORAWAN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

SEBASTIÁN ALBERTO CIFUENTES CARRASCO

PROFESOR GUÍA:
JAVIER BUSTOS JIMÉNEZ
PROFESORA CO-GUÍA:
SANDRA CÉSPEDES UMAÑA

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
ANDRÉS MUÑOZ ORDENES

Este trabajo ha sido parcialmente financiado por el Proyecto FONDEF ID19I10363

SANTIAGO DE CHILE
2021

Resumen

El Internet de las Cosas, comúnmente conocido como IoT por sus siglas en inglés, es un paradigma tecnológico que se ha posicionado fuertemente en la última década. Una descripción simple del concepto es la de una red de objetos con sensores que está conectada a Internet.

El presente trabajo de memoria se desarrolla en el contexto IoT y se enmarca en el proyecto FONDEF titulado *Sistema abierto experto para apoyar la gestión de recursos hídricos mediante monitoreo de bajo costo en tiempo real de aguas superficiales y subterráneas*, el cual tiene por objetivo aportar al cuidado y estudio de los recursos hídricos en Chile. Para esto, se propone la implementación de un sistema de monitoreo remoto de masas de agua dulce, con el fin de facilitar su estudio y cuidado.

El monitoreo de este sistema es efectuado por dispositivos electrónicos llamados nodos sensores, los cuales están compuestos por un microcontrolador, sensores de distintos tipos, una fuente de poder y un dispositivo de transmisión inalámbrica. Sus principales tareas consisten en recolectar datos del ambiente que los rodea y transmitirlos. Además, estos dispositivos están pensados para ser desplegados en ambientes de difícil acceso, por lo que deben contar con autonomía energética, generalmente basada en una batería, y con capacidad de transmisión de datos a larga distancia, siendo esta última tarea energéticamente costosa.

Dado lo anterior, el consumo energético de los nodos sensores se vuelve un factor crítico, considerando el interés que se tiene en maximizar su tiempo de monitoreo autónomo. Es por esto, que en este trabajo de memoria se plantean algoritmos para la reducción del consumo energético en un nodo sensor. Estos algoritmos se enfocan en el proceso de recolección y envío de datos, haciendo uso de codificaciones para reducir su tamaño y así gastar menos tiempo transmitiendo, lo que conlleva a una reducción de su consumo de energía.

Para probar estos algoritmos, se desarrolla a nivel de hardware y software un nodo sensor funcional de tres iteraciones. Luego, se realizan pruebas de consumo energético para comparar a los algoritmos, resultando ser todos equivalentes entre sí energéticamente.

Por esto, se comparan los algoritmos a nivel computacional, resultando todos ser de orden lineal. Finalmente, se realiza una comparación de mejor y peor caso, resultando la codificación Diferencial como el mejor algoritmo.

Agradecimientos

Agradezco a mi madre, mi padre y mi hermano, por ser pilares fundamentales para mí a lo largo de mi carrera y de toda mi vida.

Además, le doy las gracias a mis amigos y amigas, con quienes tuve la suerte de compartir, aprender y crecer a través del tiempo. En especial, agradecer a los *frens*, quienes a pesar de la distancia física hicieron del encierro un lugar más agradable a través de sus conversaciones.

Agradezco especialmente a Valeria, por su apoyo constante a lo largo de este trabajo, agradeciendo especialmente sus palabras de aliento en los momentos más difíciles en los que me encontré.

También quiero agradecer a ese Sebastián del pasado, que decidió estudiar en el DCC y, a pesar de los imprevistos encontrados en el camino, buscó la forma de seguir adelante siempre con una sonrisa.

Finalmente, la profesora Sandra Céspedes agradece el apoyo del Proyecto ANID -Basal FB0008 en la realización de esta memoria.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Introducción

El Internet de las Cosas, comúnmente conocido como IoT por sus siglas en inglés, es un paradigma tecnológico que se ha posicionado fuertemente en la última década. Si bien no existe una definición formal del término, una descripción simple del Internet de las Cosas es la de una red de objetos con sensores que está conectada a Internet. Dentro de sus aplicaciones existentes, destaca el monitoreo de distintas variables ambientales relevantes para el quehacer humano, tales como el nivel de contaminación de las ciudades, los niveles de humedad presentes en plantaciones agrícolas, la temperatura de los árboles de un bosque nativo para alertar de incendios forestales, entre otras.

El presente trabajo de memoria se enmarca en el proyecto FONDEF 19I10363, titulado *istema abierto experto para apoyar la gestión de recursos hídricos mediante monitoreo de bajo costo en tiempo real de aguas superficiales y subterráneas*. El objetivo de este proyecto es aportar al cuidado y estudio de los recursos hídricos a nivel nacional. Para esto, se propone la implementación de un sistema de monitoreo remoto de masas de agua, con el fin de facilitar su estudio y permitir la detección temprana de eventos anómalos. Así , por ejemplo, un cambio brusco en los niveles de pH de un lago siendo monitoreado podría dar aviso de una posible contaminación en el lugar, permitiendo una pronta acción para reducir los daños asociados.

Este monitoreo se realiza a través del despliegue de una red de sensores inalámbricos en un área de interés, la cual se encarga de transmitir los datos a una fuente que los almacena. Luego, los datos son accedidos por un sistema que los analiza y posee la capacidad de levantar alertas a partir de anomalías encontradas en éstos.

La red de sensores inalámbricos está formada por nodos sensores, los cuales son piezas de hardware compuestas fundamentalmente por un microcontrolador, uno o más sensores, un componente de transmisión inalámbrica y una fuente de poder. Las dos tareas fundamentales de cada nodo sensor son la lectura de datos desde sus sensores y la transmisión de éstos, siendo esta última una tarea energéticamente costosa.

Además, un factor a considerar de estas redes es que su despliegue está pensado a realizarse en lugares alejados o de difícil acceso, por lo que cada nodo sensor de la red debe contar con autonomía energética, basada en el uso de baterías. Esta situación vuelve un factor crítico el consumo energético de éstos dispositivos.

Por otra parte, dada la posición espacial en que pueden desplegarse estos dispositivos, se hace necesario que éstos sean capaces de transmitir información a larga distancia, considerando un orden de kilómetros para esta transmisión. En este tipo de escenarios, se privilegia

el uso de redes de tipo LPWAN -acrónimo para *Low Power Wide Area Network*. Estas redes están diseñadas para permitir comunicaciones de largo alcance, haciendo uso de un bajo consumo energético para la transferencia de datos. Las redes de sensores contempladas en el proyecto utilizan el protocolo LoRaWAN [3], un protocolo LPWAN que destaca por sus transmisiones de largo alcance con bajo consumo energético.

El presente trabajo de memoria aborda el problema de reducir el consumo energético de los nodos sensores, desde un enfoque algorítmico, enfocándose particularmente en el proceso de recolección y transmisión de datos. Este problema se vuelve relevante dentro del contexto actual, ya que el paradigma del Internet de las Cosas se ha masificado y la recolección de datos a través de sensores inteligentes está tomando cada vez más terreno en nuestra vida diaria. Un ejemplo de esto es la existencia de proyecciones que estiman una cantidad de dispositivos interconectados del orden de los 10 billones para el año 2023 [4]. En la misma línea, considerando la cantidad de sensores que existen y los consumos energéticos asociados a éstos, es necesario promover la reducción del consumo de energía en IoT y generar conciencia de los avances tanto técnicos [5] como tecnológicos [6] que faciliten un uso más sustentable de éstas tecnologías.

Objetivos

El objetivo principal del trabajo de esta memoria es proponer e implementar algoritmos para la reducción del consumo energético en la fase de recolección y transmisión de datos de un prototipo experimental de nodo sensor para aguas subterráneas y superficiales.

Además, para cumplir con el objetivo principal antes mostrado, se plantean los siguientes objetivos específicos:

1. Desarrollar un nodo sensor funcional, tanto a nivel de hardware como de software, sobre el cual se implementarán los algoritmos planteadas. El enfoque del desarrollo debe considerar el buen uso de los recursos computacionales, con enfoque en la eficiencia energética del sistema.
2. Proponer estrategias de reducción de consumo energético que se adapten al contexto del problema abordado, a partir de la revisión de técnicas de recolección y envío de datos para redes de sensores inalámbricos presentes en la literatura.
3. Analizar y explicar las estrategias propuestas, desde un punto de vista de la lógica de funcionamiento, así como de su naturaleza algorítmica.
4. Validar el funcionamiento de las estrategias de reducción de energía planteadas mediante la medición de la energía consumida por un nodo sensor.

Capítulo 1

Problema

1.1. Contexto

El presente trabajo de memoria se realiza en el contexto del proyecto FONDEF *Sistema abierto experto para apoyar la gestión de recursos hídricos mediante monitoreo de bajo costo en tiempo real de aguas superficiales y subterráneas*. Este proyecto tiene por objetivo aportar al cuidado y estudio de los recursos hídricos presentes a nivel nacional. Para lograrlo, propone el desarrollo e implementación de un sistema computacional capaz de recolectar información de interés desde fuentes de agua dulce y transmitirla hacia un servidor central encargado de su almacenamiento y posterior procesamiento.

La implementación de la solución antes descrita se logra mediante el despliegue de uno o más dispositivos electrónicos en un área de interés, como por ejemplo un lago, los cuales deben ser capaces de medir parámetros fisicoquímicos presentes en el agua y transmitirlos hacia un servidor que los almacena. Luego de esto, los datos son accedidos por un sistema computacional experto, que realiza un análisis sobre éstos y puede ejecutar distintas acciones dependiendo de la naturaleza de los datos analizados. El proceso antes descrito puede dividirse principalmente en tres etapas:

1. Recolección y transmisión de datos
2. Filtrado y almacenamiento de datos
3. Procesamiento, análisis e interpretación de los datos

Este trabajo de memoria se enmarca en la fase de recolección y transmisión de datos.

1.2. Planteamiento

El problema a abordar en esta memoria es el planteamiento y estudio de estrategias para la reducción del consumo energético en la fase de recolección y transmisión de datos. Estas estrategias están pensadas para ser implementadas sobre los dispositivos electrónicos encargados de la medición y transmisión de los parámetros fisicoquímicos de interés de las fuentes de agua dulce a estudiar. Estos dispositivos se denominan **nodos sensores**.

Al momento de comenzar el presente trabajo, no existían nodos sensores funcionales sobre los cuales implementar y probar estrategias de reducción de consumo energético. Por lo tanto, el primer problema abordado fue el desarrollo e implementación de un prototipo funcional de un nodo sensor. Este sistema, el cual combina tanto el hardware a usarse como el software que lo administra, tiene por finalidad recopilar y transmitir información de la calidad de agua de un entorno dado, por lo que debe cumplir con una serie de requisitos. Estos requisitos son:

1. El sistema debe ser capaz de leer sensores de distinto tipo, ya sean analógicos o digitales.
2. El sistema debe poder diferenciar e identificar los distintos sensores con que trabaje.
3. El sistema debe poseer un mecanismo de almacenamiento no-volátil para guardar la información de los sensores que va siendo recopilada.
4. El sistema debe ser capaz de mantener una fecha y hora precisas con el fin de contextualizar las lecturas de sensores realizadas.
5. El sistema debe ser capaz de recopilar información de manera automática, en una frecuencia regular de tiempo, previamente establecida.
6. El sistema debe ser capaz de transmitir la información recopilada de manera inalámbrica.
7. El sistema debe ser capaz de transmitir la información de manera automática, en una frecuencia regular de tiempo, previamente establecida. Esta frecuencia debe ser mayor o igual que la frecuencia establecida para la recopilación automática de datos.

Junto con los requisitos antes mencionados, el sistema debe cumplir con una serie de restricciones que se listan a continuación:

1. El sistema debe funcionar en un entorno restringido, esto es, con restricciones tanto de memoria como de almacenamiento.
2. El sistema debe considerar un funcionamiento en el tiempo dado por una fuente de almacenamiento externa limitada, como por ejemplo, una batería.
3. El sistema debe considerar restricciones naturales a los protocolos de comunicación que se utilicen para la transmisión de los datos generados.
4. El método de transmisión inalámbrica debe realizarse a través del protocolo de red LoRaWAN.

Con lo anterior, el problema abordado en esta memoria se divide en tres partes. En primer lugar se presenta la necesidad de desarrollar e implementar el sistema de recolección y transmisión de datos. Posterior a esto, se plantean estrategias para el ahorro energético en el funcionamiento del nodo sensor. Por último, con el objetivo de observar el comportamiento energético del sistema y validar las estrategias planteadas, se genera un sistema de medición de corriente eléctrica, con el objetivo de conocer el consumo que el sistema desarrollado presenta.

Dentro de la solución propuesta en este trabajo de memoria puede verse la influencia de los distintos cursos de formación tomados a lo largo de la carrera. Así, por ejemplo, al tratar el problema con circuitos electrónicos y elementos de hardware, destacan los conocimientos adquiridos en el curso de Arquitectura de Computadores, facilitando la comprensión de la lógica detrás de éstos componentes y sus interacciones entre sí. Del mismo modo, para el

planteamiento de los requisitos y restricciones de la solución, se utilizaron las técnicas y métodos aprendidos en los cursos de Ingeniería de Software.

Por otro lado, al involucrar el problema abordado la necesidad de transmisión de datos desde un punto del espacio a otro, se hicieron necesarios los conocimientos adquiridos en el curso de Redes, facilitando así la comprensión y visualización de los procesos involucrados desde la toma de datos hasta su llegada a destino, haciéndose patente, además, la esencialidad del uso de buenos protocolos de comunicación. Finalmente, al contextualizarse el trabajo en un entorno restringido, tanto a nivel de recursos como de energía, los conocimientos adquiridos en cursos como Algoritmos y Estructuras de Datos, y Diseño y Análisis de Algoritmos se vuelven fundamentales para el desarrollo de una solución de software eficiente y de calidad.

Capítulo 2

Marco teórico

En este capítulo se definen y explican conceptos que son necesarios para la comprensión de la solución implementada. Además, se describe el estudio bibliográfico que se llevó a cabo previo a la realización del trabajo descrito en el presente documento.

2.1. Conceptos técnicos

2.1.1. LPWAN

Nombre proveniente de las siglas en inglés para Low Power Wide Area Network. Una LPWAN es un tipo de red inalámbrica de gran cobertura, diseñada para permitir comunicaciones de largo alcance [7]. Estas comunicaciones se caracterizan por presentar una baja tasa de transferencia de datos, así como por consumir una cantidad reducida de energía, por lo que este tipo de redes son ideales para contextos de desarrollo que involucran dispositivos restringidos.

2.1.2. LoRa

LoRa, del inglés *Low Range*, es una capa física y una técnica de modulación inalámbrica usada para crear enlaces de comunicación de largo alcance [1].

La principal ventaja que presenta está en la capacidad de largo alcance de la tecnología, siendo capaz un único *gateway* de cubrir ciudades enteras o cientos de kilómetros cuadrados. Este rango dependerá del ambiente o de las obstrucciones dadas para una determinada localidad.

2.1.3. LoRaWAN

LoRaWAN es un protocolo de red de área amplia y baja potencia, diseñado para conectar de manera inalámbrica objetos que funcionan con baterías a Internet, a través de redes regionales, nacionales o globales. Además, ofrece requisitos clave para el Internet de las Co-

sas, tales como comunicación bidireccional, seguridad de extremo a extremo y servicios de movilidad y localización [8].

LoRaWAN define el protocolo de comunicación y la arquitectura del sistema de red para una LPWAN, mientras que la capa física LoRa habilita el enlace de comunicación de largo alcance [1].

La arquitectura de red definida por LoRaWAN se basa en el objetivo de preservar la vida útil de las baterías asociadas a los nodos terminales de la red. Para esto, organiza a los nodos terminales en una topología de estrella, la cual se forma alrededor de *gateways* LoRa. Estos, a su vez se organizan de igual forma alrededor de un servidor de red alojado en la nube, el cual está encargado de administrar la red, gestionar los paquetes que entran, realizar chequeos de seguridad, entre otras cosas. La Figura 2.1 ilustra la arquitectura de red antes mencionada.

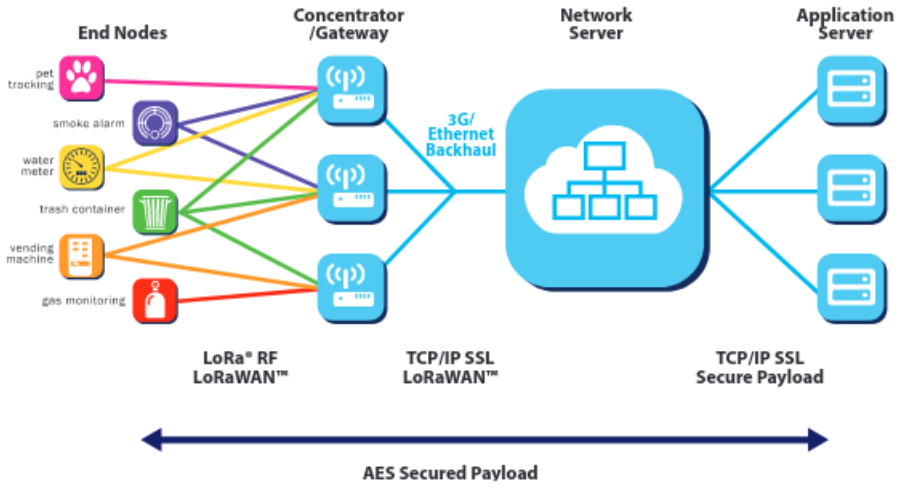

Figura 2.1: Arquitectura de red LoRaWAN. Figura extraída de [1]

Un aspecto importante a destacar de ésta arquitectura de red, es que los nodos terminales no se asocian a *gateways* específicos. En su lugar, un paquete transmitido por un nodo terminal puede enviarse a cualquier *gateway*, el cual se encarga de hacer llegar el paquete hasta el servidor de red, el que finalmente gestiona el destino de llegada del paquete.

2.1.4. Comunicación secuencial

La comunicación secuencial o comunicación serie es el proceso de envío de datos de un bit a la vez, de manera sucesiva a través de un bus de datos.

2.1.5. SPI

Proveniente de las siglas en inglés para Serial Peripheral Interface, SPI es un protocolo de comunicación secuencial síncrono, usado para la transmisión de datos en distancias cortas. Se caracteriza por funcionar bajo una arquitectura de maestro-esclavo y necesitar cuatro líneas de señal para realizar la comunicación. Así, su interfaz de comunicación se compone de:

1. Una señal de reloj llamada CLK, enviada desde el maestro a todos los esclavos. De esta manera, todas las señales dentro de la comunicación son síncronas a ésta señal de reloj.

2. Una señal de selección de esclavos llamada SS, la cual se utiliza por el maestro de la comunicación para elegir con qué esclavo comunicarse.

3. Una línea de datos para enviar la información desde el maestro hacia los esclavos llamada MOSI, por su nombre en inglés *Master Out Slave In*.

4. Una línea de datos para recibir la información desde los esclavos hacia el maestro llamada MISO, por su nombre en inglés *Master In Slave Out*.

Los módulos de comunicación utilizados para hablar con tarjetas SD utilizan este protocolo para comunicarse con microcontroladores.

2.1.6. I2C

Su nombre proviene del inglés *Inter-Integrated Circuit*. I2C es un protocolo de comunicación secuencial síncrono diseñado para permitir la comunicación entre múltiples dispositivos. Al igual que SPI, está pensado para la transmisión de datos en distancias cortas.

Se caracteriza por funcionar bajo una arquitectura maestro-esclavo y necesitar dos líneas de señal para intercambiar información, las cuales se etiquetan por SCL y SDA. La línea SCL consiste en una señal de reloj, mientras que SDA es la línea por la cual se transportan los datos.

2.1.7. UART

Compuesto por las siglas de su nombre en inglés *Universal Asynchronous Receiver/Transmitter*, UART no es un protocolo de comunicación como los anteriormente descritos, sino una pieza de hardware dedicada a transmitir y recibir datos de manera secuencial y asíncrona. Está pensada para transmitir información exclusivamente entre dos dispositivos, a diferencia de los protocolos anteriores.

Para esto, la comunicación debe realizarse mediante el uso de dos UART, las cuales se comunican a través de dos líneas de señal etiquetadas por TX y RX. Además, al ser un método de comunicación asíncrono, se hace necesario definir una velocidad de transmisión entre los dos UART involucrados en la comunicación, permitiendo así un correcto envío y recepción de los datos a transmitir.

La forma de transmitir datos de la UART es a través de la generación de paquetes de bits, con los cuales se delimita la información que está siendo enviada. Una ventaja de este componente es que ofrece un algoritmo sencillo de verificación de errores en la transmisión, llamado **bit de paridad**, el cual señala si una cadena de bits contiene un número par o impar de unos.

2.2. Estudio bibliográfico

En la literatura, existe una gran variedad de algoritmos y técnicas para ahorro energético en la recolección y transmisión de datos en redes de sensores inalámbricos, con sus respectivos análisis de consumo de energía y calidad de transmisión de datos. La técnica más utilizada se denomina *Compressive Sensing*, técnica de procesamiento de señales de la cual existe una gran cantidad de variaciones [9] y que consiste en la reconstrucción completa de una señal a partir de un muestreo pequeño de ésta.

Dentro de la misma línea, existen trabajos en que se proponen nuevos algoritmos en el contexto de LoRaWAN, adjuntando también los respectivos análisis de energía y calidad de transmisión. Un ejemplo de esto es el estudio [10] en que se propone una versión mejorada de un algoritmo de *Compressive Sensing* para redes de sensores inalámbricos sobre LoRaWAN. Además, en el trabajo se comparan distintos algoritmos de *Compressive Sensing* con el algoritmo propuesto. Para ahondar en esta publicación se estudió una publicación del año 2008 [11] donde se explican los fundamentos de la técnica de *Compressed Sensing*. Enfocada desde el área de análisis de señales, en la publicación se explican las bases teóricas del funcionamiento de la técnica, las cuales se sustentan en la compresión de una señal a partir de un submuestreo de ésta y su posterior reconstrucción a partir de la resolución de un problema de optimización. Cabe destacar que la señal a comprimir debe cumplir con ciertos requisitos previos, al igual que el proceso por el cual se submuestrea. Con esto, en [10] se propone un algoritmo para redes con arquitectura de estrella, donde los nodos hablan directamente con los dispositivos de entrada o *gateways* LoRaWAN, mientras que se optimiza la reconstrucción de las señales de interés dentro de la red a partir de la mejora de un método previamente propuesto en la literatura. Así, el algoritmo se divide en el envío de los datos desde los nodos sensores, los cuales están sujetos a programas de "sueño" para ahorro de batería, y la reconstrucción de los datos recibidos por parte de la red.

Por otra parte, se analizaron publicaciones en que se detallan técnicas de compresión de datos no aplicadas a LoRaWAN, sino generalizadas a redes de sensores inalámbricos. El primer caso de estudio [12] corresponde a una propuesta de ahorro energético a nivel de red combinando compresión de datos a través de *Compressive Sensing* y una técnica de ruteo oportunista, para luego generar una reconstrucción de datos en el punto de llegada. Si bien la publicación reporta un prolongamiento de la vida de la red por un factor de entre 1.5 y 2, la implementación está pensada para una arquitectura de red de multisaltos o *multihop*, donde la compresión se va haciendo a medida que se avanza a través de la red hacia el nodo sumidero, por lo que escapa de la implementación presupuestada en el contexto del proyecto. Un caso similar ocurre en [13] y [14], donde los algoritmos también dependen de una arquitectura de red de multisaltos.

Para enfocarse en un contexto más cercano a la realidad del proyecto, se estudiaron algoritmos de compresión de datos que no requirieran comunicación entre nodos sensores, encontrándose en [15] un algoritmo orientado a comprimir las mediciones realizadas en ambientes donde los cambios de las variables medidas fueran continuos y lentos, usándose como ejemplo la medición de temperatura, concluyéndose que el algoritmo propuesto era útil en el contexto de redes de sensores inalámbricos. Otro caso estudiado de compresión se da en [16], en que se proponen algoritmos para redes de sensores inalámbricos con un radio de

compresión alcanzable de aproximadamente el 70 %.

Agregado a lo anterior, se estudiaron dos casos de codificación de datos en redes de sensores inalámbricos. En [17] se plantea una compresión de datos mediante codificación predictiva, obteniéndose una reducción de la energía consumida por los nodos y por tanto aumentado el tiempo de vida de la red. Por su parte en [18] se plantea un algoritmo de codificación basado en las diferencias encontradas entre distintos muestreos generados por la red. Si bien, ambas publicaciones consideran agrupación de mediciones desde distintos nodos sensores, es la lógica detrás de su método de compresión la que puede resultar útil para la compresión de múltiples muestras dentro de un único nodo sensor.

Finalmente, se encontraron estudios relacionados a la reducción de consumo energético utilizando métodos que no involucraban modificación de los datos transmitidos, como por ejemplo, modificar la tasa de envío de la red para optimizar el consumo de energía [19], o al estudio del rendimiento de la LoRaWAN a partir de distintas configuraciones de los parámetros de transmisión [20].

Con todo lo anterior, el trabajo de optimización a realizar se debe enfocar en la codificación de los datos a transmitir, modificando la cantidad de datos a enviar para reducir el consumo energético asociado al envío de éstos. Además, las restricciones técnicas de los nodos sensores del proyecto generan un caso de uso particular, con restricciones específicas de memoria y procesamiento, por lo que existe la posibilidad de que ciertos algoritmos presentes en la literatura no sean implementables en la práctica.

Capítulo 3

Solución

3.1. Arquitectura lógica del sistema

Para explicar la solución implementada se realiza, en primer lugar, una abstracción de las tareas que el sistema debe realizar, de manera tal que puedan definirse módulos de trabajo y relaciones entre éstos que faciliten la concepción de la solución y su posterior desarrollo.

Con esto en mente, y considerando el contexto descrito en el anterior capítulo, se plantea una arquitectura lógica del sistema, en la cual se identifican cuatro tareas principales que éste debe realizar. Estas tareas son:

1. **Recolección**: el sistema debe recolectar datos desde una fuente de interés a través de hardware especializado para esto. Además, debe poseer el software necesario para que esta recolección se haga de manera automática y arbitraria.
2. **Transformación**: el sistema debe transformar los datos con que trabaje, utilizando estructuras de datos convenientes según las distintas necesidades que se requieran.
3. **Almacenamiento**: el sistema debe almacenar datos tanto de manera volátil, como no volátil. Esto implica la capacidad del sistema de leer y escribir datos a voluntad, accediendo a éstos cuando sea necesario.
4. **Transmisión**: el sistema debe transmitir los datos que posea hacia dispositivos externos al mismo.

Estas tareas se relacionan tanto entre ellas como con el entorno que rodea al sistema, lo cual se muestra a continuación en la Figura 3.1, que establece el comportamiento lógico que debe presentar el sistema. El rectángulo principal que encasilla las tareas representa al sistema, mientras que las flechas determinan una relación de comunicación de datos entre los distintos objetos del diagrama.

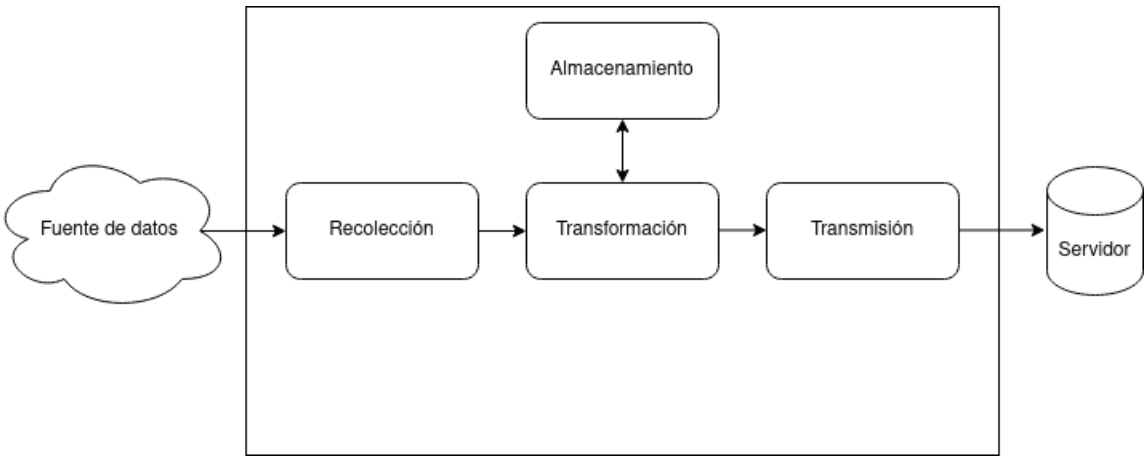

Figura 3.1: Diagrama lógico de la solución.

3.2. Arquitectura física del sistema

3.2.1. Componentes electrónicos utilizados

Luego de explicitar las tareas que el sistema debe desarrollar y sus respectivas interacciones, es necesario establecer el hardware sobre el cual se trabajará para realizar estas tareas. Esto es un factor relevante para el proceso de diseño de la solución, pues los distintos componentes que se utilicen determinarán las restricciones a seguir en la implementación de la solución. Con esto presente, a continuación se describen los componentes electrónicos principales de la solución implementada:

- **Arduino Nano**: placa de desarrollo basada en el microcontrolador ATmega328, un microchip con 32 KB de memoria flash, 2 KB de memoria SRAM y 1 KB de EEPROM. Cuenta con un total de 30 pines de comunicación, 22 de entrada/salida digital y 8 de entrada analógicos, dentro de los cuales existen interfaces de comunicación I2C, SPI y UART. Su voltaje de funcionamiento es de 5V.

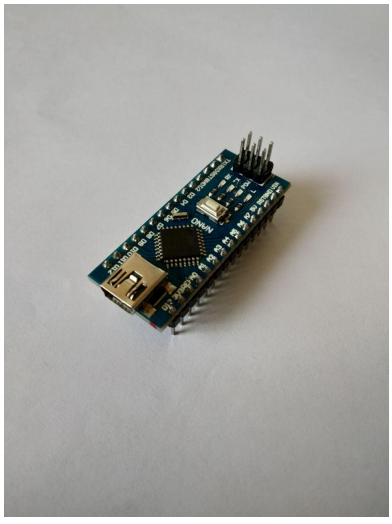

Figura 3.2: Arduino Nano utilizado en el sistema.

- **LoPy4**: placa orientada al desarrollo de aplicaciones de red y de Internet de las Cosas. Cuenta con un microcontrolador ESP32, con una memoria flash de 8 MB y una RAM de 4 MB. A nivel de comunicación inalámbrica, presenta cuatro tecnologías de integradas, siendo estas WiFi, Bluetooth, SigFox y LoRa. Por otra parte, cuenta con 28 pines, de los cuales 25 son de uso general, entre los cuales destacan aquellos dedicados a interfaces de comunicación I2C, UART y SPI. Su voltaje de operación es de 3.3V.

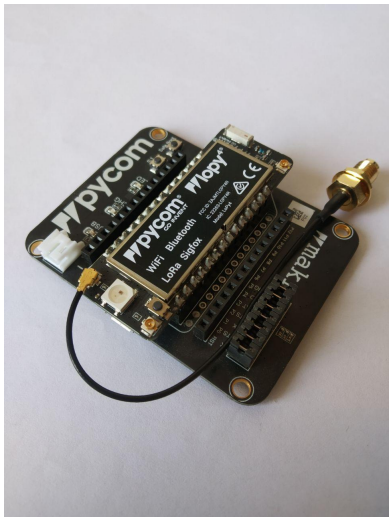

Figura 3.3: LoPy4 utilizada en el sistema.

- **Módulo lector de tarjeta microSD**: dispositivo electrónico utilizado como interfaz de lectura de tarjetas MicroSD. Cuenta con una interfaz de comunicación SPI, mediante la cual puede recibir comandos desde un microcontrolador para gestionar la tarjeta de memoria. Su voltaje de operación es de 3.3V, pero cuenta con un regulador de voltaje que permite una alimentación desde los 3.3V hasta 5.5V

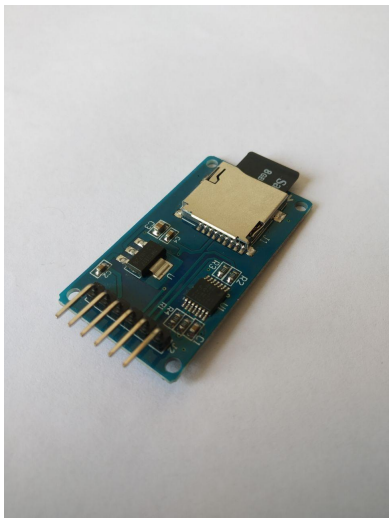

Figura 3.4: Módulo lector de tarjeta microSD utilizado en el sistema.

- **RTC DS3231**: sus siglas provienen del inglés *Real Time Clock*. Es un reloj de tiempo real de alta precisión con una interfaz de comunicación I2C. El reloj almacena informa-

ción de segundos, minutos, horas, días, meses y años. Su voltaje de funcionamiento es de 5V, requiriendo además una pila de litio -o pila de reloj-, la cual le permite mantener su información interna en caso de que la fuente de alimentación principal se apague.

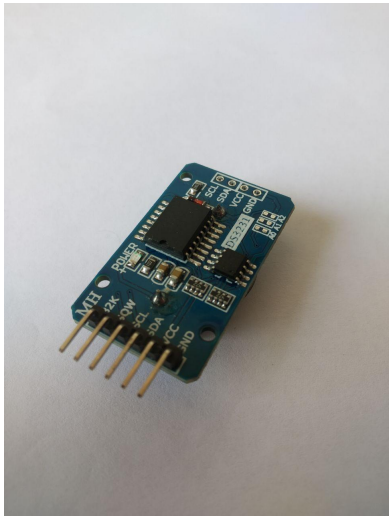

Figura 3.5: Reloj en tiempo real utilizado en el sistema.

- **Sensor de temperatura DS18B20**: termómetro digital resistente al agua de la compañía Maxim Integrated[1], capaz de medir temperaturas entre -10°C y 85°C. Posee un error de medición de ± 0.5°C y se comunica mediante una interfaz 1-Wire. Su voltaje de operación está en el rango de 3.3V a 5.0V.

Figura 3.6: Sensor de temperatura DS18B20 utilizado en el sistema.

- **Conversor de nivel**: dispositivo electrónico que permite la comunicación entre sistemas que trabajan a distintos niveles de voltaje. Se divide en dos partes, la del lado izquierdo se compone de las entradas destinadas para el sistema con voltaje más alto, como por ejemplo 5V, y la del lado derecho está destinada al sistema con voltaje más bajo.

[1]https://www.maximintegrated.com/

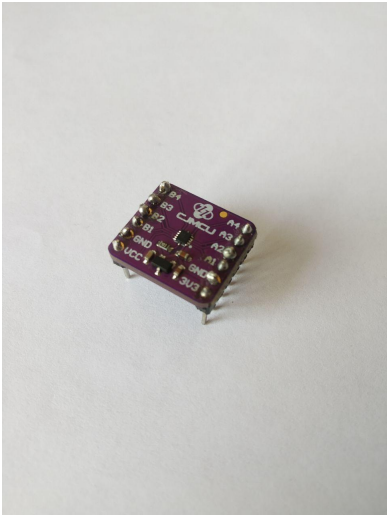

Figura 3.7: Conversor de nivel utilizado en el sistema.

3.2.2. Definición de responsabilidades

Con la existencia de dos microcontroladores entre los componentes, y considerando las capacidades que cada uno posee, se definen dos secciones o bloques de hardware a implementar. Esta definición se hace con el fin de repartir la carga existente en el proceso de recolección de datos, generando así dos sistemas independientes que pueden desarrollar tareas de manera paralela. Estos bloques se describen como sigue:

- **Bloque Sensor**: Bloque de hardware, controlado por el Arduino Nano, encargado de la recolección, transformación, almacenamiento y transmisión de datos hacia el Bloque Transmisor. Carece de capacidad de transmisión de datos de manera inalámbrica.
- **Bloque Transmisor**: Bloque de hardware, controlado por la LoPy4, encargado de la recepción de datos desde el Bloque Sensor y su posterior transmisión mediante radiofrecuencia a un dispositivo de salida, el cual se encarga de la llegada de datos hacia el servidor de destino. Su funcionamiento se asemeja a la de un servidor con un único cliente.

3.2.3. Diagrama de circuitos

Ya con los componentes electrónicos del sistema establecidos y la definición de los bloques de hardware con sus respectivas responsabilidades, se diseña el circuito a utilizar en la implementación de la solución. La Figura 3.8 muestra la disposición de los distintos componentes del sistema, junto con sus respectivas conexiones entre sí. Cabe señalar que el circuito mostrado se presenta sobre una placa de pruebas o *protoboard*, debido a que el sistema implementado corresponde al prototipo de una solución de hardware a ser desarrollada en etapas futuras del proyecto.

En la figura se pueden observar los dos bloques de hardware definidos anteriormente. El Bloque Sensor está compuesto por un Arduino Nano, un módulo RTC, un módulo de lectura de microSD y un sensor de temperatura DS18B20, mientras que el Bloque Transmisor se

compone por una LoPy4 y una antena, necesaria para la transmisión de datos mediante radiofrecuencia.

La comunicación entre ambos bloques se realiza mediante la conexión de los dos microcontroladores del sistema, la cual debe necesariamente realizarse a través de un conversor de nivel, presente en el costado izquierdo de la figura, debido a la diferencia en los voltajes de operación de los microcontroladores. Esta comunicación es de tipo secuencial y se establece a través de las interfaces UART presentes en cada microcontrolador.

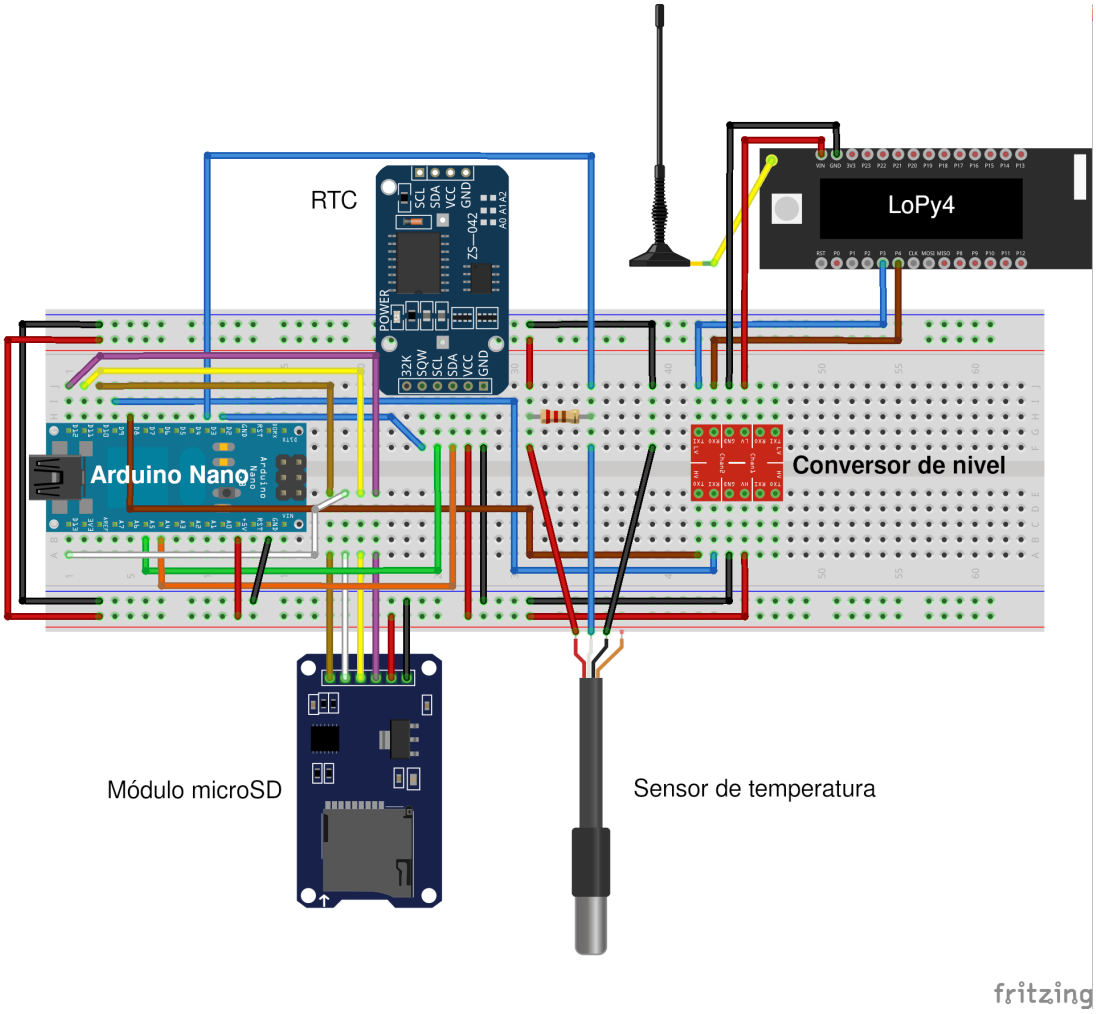

Figura 3.8: Circuito implementado en la solución

3.2.4. Diagrama de red del sistema

Teniendo ya establecidos los bloques de hardware y sus respectivas funcionalidades, se establece la arquitectura de red en que participará el sistema.

Como se dijo anteriormente, la comunicación entre Bloque Sensor y Bloque Transmisor se realiza a través de comunicación secuencial, conectada de manera alámbrica a través de las interfaces UART de cada bloque. Por otra parte, el Bloque Transmisor realiza el envío

de los datos hacia un dispositivo de salida o *Gateway* de manera inalámbrica, utilizando comunicación LoRa. Finalmente, los datos son transmitidos por el dispositivo de salida a través de comunicación Ethernet hacia la aplicación de llegada, almacenándose los datos en una base de datos dedicada.

La Figura 3.9 ilustra la arquitectura de red del sistema, junto con el flujo que debe seguir un paquete de datos transmitido desde el Bloque Sensor y sus respectivos modos de transmisión a lo largo de la red.

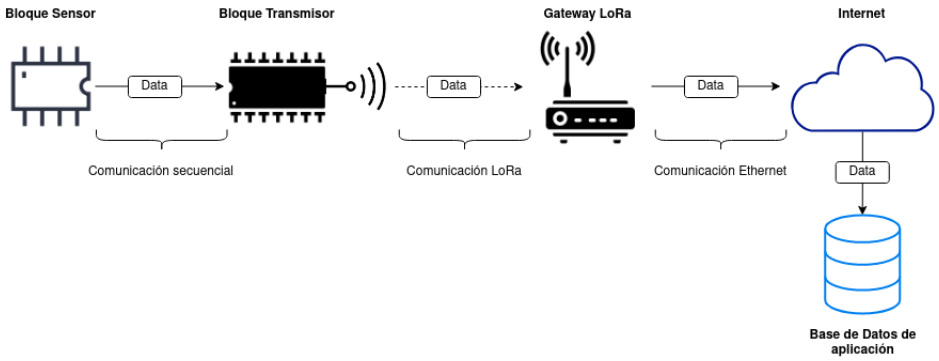

Figura 3.9: Diagrama de red del sistema.

3.3. Desarrollo del software del sistema

Con el hardware del sistema ya definido, procede el desarrollo del software que funcionará sobre éste. La implementación en cuestión debe considerar la gestión y el trabajo con los distintos dispositivos electrónicos antes descritos, así como la realización de las tareas planteadas a nivel de la lógica del sistema.

La solución de software se desarrolla con el objetivo de ser implementada sobre cada uno de los microcontroladores presentes en el sistema, Arduino Nano y LoPy4, los cuales se programan en los lenguajes C++ y Python, respectivamente. La metodología de programación usada para este desarrollo es de tipo incremental, considerando principalmente tres iteraciones de código funcional. Además, se utiliza Git como sistema de versionamiento a lo largo del proceso de desarrollo, con el objetivo de contar con un respaldo en caso de generarse errores a la hora de hacer cambios en el sistema.

3.3.1. Primera iteración

3.3.1.1. Objetivos

Con la primera iteración del sistema se busca el desarrollo de una solución de software que sea capaz de ejecutar las tareas de recolección, transformación y almacenamiento de datos. En esta iteración no se considera la tarea de transmisión de datos, por lo que se trabaja exclusivamente sobre el Bloque Sensor del sistema. Junto con esto, tampoco se considera la implementación de la comunicación esperada entre el anterior bloque y el Bloque Transmisor.

Por otra parte, la primera iteración también tiene por objetivo adentrarse en la programación de un sistema específicamente formulado para ejecutarse en un microcontrolador Arduino, familiarizándose con las distintas capacidades que el entorno de desarrollo ofrece, así como con las librerías existentes para facilitar el trabajo realizado.

3.3.1.2. Funcionalidades implementadas

En esta iteración se implementan las siguientes funcionalidades, a nivel de Bloque Sensor:

- Se implementa la lectura del sensor de temperatura DS18B20.
- Se implementa la comunicación entre el microcontrolador Arduino y el módulo RTC.
- Se implementa la comunicación entre el microcontrolador Arduino y el módulo lector microSD.
- Se implementa la escritura de datos en la tarjeta microSD, mediante la generación de un archivo CSV.
- Se implementa una rutina en que el Arduino Nano entra en Modo Dormido, un estado de bajo consumo energético en que el microcontrolador detiene la ejecución de su rutina, manteniendo energizada la memoria RAM, no existiendo pérdida de la información temporal almacenada en el sistema.
- Se implementa una rutina de alarma para la recolección automática de datos, en la cual el módulo RTC genera una interrupción a través de un pin digital del microcontrolador, sacándolo del Modo Dormido y devolviéndolo a la ejecución de su rutina.

3.3.1.3. Detalles de la implementación

Comunicación entre microcontrolador y componentes

Para establecer la comunicación entre el Arduino y los distintos componentes electrónicos del sistema se utilizan librerías dedicadas, disponibles en la página web oficial de Arduino[2]. En primer lugar, la comunicación con el sensor DS18B20 se realiza mediante el uso de la librería DallasTemperature[3], dedicada a la gestión de un grupo de sensores de temperatura de la compañía Maxim Integrated, entre los que se encuentra el utilizado en el sistema. Esta librería ofrece un manejo simplificado del sensor de temperatura, mediante el uso de una clase de C++, la cual ofrece métodos para la identificación del sensor conectado al circuito y su posterior consulta, obteniendo así los datos de temperatura que el sensor registre.

En segundo lugar, la comunicación con el módulo RTC se realiza utilizando la librería RTClib[4], desarrollada y gestionada por la compañía de electrónica Adafruit[5]. Esta librería realiza gestión del módulo RTC a través de una clase de C++, de igual forma que la librería anterior. Con esta clase es posible ajustar los datos temporales del RTC, configurar alarmas, consultar la hora del reloj, entre otras funcionalidades.

[2]https://www.arduino.cc/reference/en/libraries/
[3]https://www.arduino.cc/reference/en/libraries/dallastemperature/
[4]https://www.arduino.cc/reference/en/libraries/rtclib/
[5]https://www.adafruit.com/

Finalmente, la comunicación con el módulo lector microSD se realiza utilizando la librería SD[6], librería oficial de Arduino que facilita el manejo de memorias SD. La librería ofrece dos clases de C++, una orientada a la gestión de la memoria SD y otra orientada a la gestión de archivos dentro de ésta.

Rutina de interrupciones y toma de datos

Llamaremos rutina de interrupción al proceso que involucra las transiciones del Arduino Nano desde un estado activo de ejecución de su programa al Modo Dormido y su posterior retorno a un estado activo. Esta última transición, debido a una restricción del microcontrolador, puede realizarse únicamente mediante una interrupción externa, generada por una señal recibida a través de alguno de los pines de entrada presentes en el dispositivo, siendo necesaria la existencia de un componente externo que envíe esta señal.

Convenientemente, el módulo RTC utilizado en la solución cuenta con un sistema de alarmas configurables integrado, permitiendo enviar una señal por uno de sus puertos de salida en una fecha y hora determinada. Esta funcionalidad es la que se utiliza en la rutina de interrupción para devolver al microcontrolador a su estado de ejecución normal.

Si a la rutina de interrupción anterior se le agrega un valor numérico que represente la frecuencia con que el módulo RTC debe enviar la señal para despertar al Arduino, se tiene una rutina de ejecución con una frecuencia dada. Ahora, si la tarea ejecutada por el microcontrolador al despertar es la de recolectar los datos desde el sensor de temperatura, se genera una rutina de recolección de datos con una frecuencia determinada. Esta rutina fue la implementada para la toma de datos del sistema, y se describe como sigue:

1. Antes de iniciar el sistema, se define una frecuencia de recolección de datos F, que representa un intervalo de tiempo en segundos.
2. Al ejecutarse el sistema, el Arduino Nano solicita la hora y fecha actual al módulo RTC.
3. Con la hora y fecha consultada, se configura una alarma en el módulo RTC a ser enviada F segundos después de la fecha y hora medida.
4. El Arduino Nano pasa de su estado activo a Modo Dormido.
5. Llegada la fecha y hora configuradas por el Arduino en el segundo paso, el módulo RTC envía una señal al microcontrolador, devolviéndolo a su rutina de ejecución normal.
6. El Arduino Nano retoma su ejecución normal, consultando la fecha y hora al módulo RTC y la temperatura al sensor DS18B20, almacenando estas medidas para su posterior uso.
7. Utilizando la fecha y hora consultada en el paso anterior, el Arduino Nano vuelve al paso 3 de esta rutina.

El algoritmo antes descrito se ilustra en la Figura 3.10.

[6]https://www.arduino.cc/en/Reference/SD

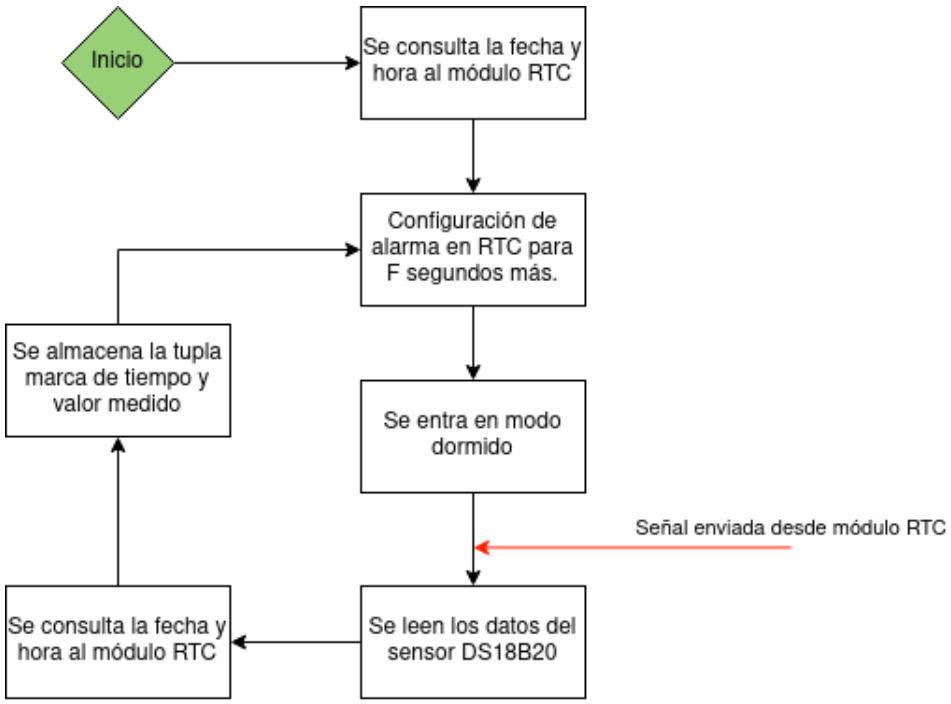

Figura 3.10: Rutina de interrupción y toma de datos implementada en la primera iteración.

Rutina de almacenamiento de datos

Como se describió anteriormente, un dato de interés recopilado por el sistema es la combinación de una medida realizada por el sensor de temperatura con las hora y fecha en que se efectuó la consulta de la medida. Estos datos pueden ser representados como una tupla constituida por un valor de punto flotante, que representa la medida del sensor, y un entero sin signo, que representa la marca de tiempo o *timestamp* en formato Unix en que se realizó la medida.

La rutina de almacenamiento de datos implementada consiste en la escritura de los datos, a medida que son recopilados por el sistema, en un archivo de texto con formato CSV. Esto se realiza con la intención de que los archivos sean directamente interpretables por un usuario que quiera tener acceso a los datos almacenados en la tarjeta de memoria.

Para realizar esto, se sigue la siguiente rutina:

1. Al encenderse el sistema, se genera un archivo dentro de la tarjeta microSD con nombre "data.csv". Al mismo tiempo, se genera un arreglo de valores de punto flotante (*float*) y un arreglo de enteros de 32 bits sin signo (*uint32_t*). Ambos arreglos tienen el mismo largo.
2. Cada vez que se realiza una medida del sensor de temperatura, se almacena el valor obtenido en el primer arreglo y su respectiva marca de tiempo en el segundo arreglo.
3. Cuando los arreglos se llenan, se itera sobre éstos, generando para cada par de valores una cadena de texto de la forma `"valor_sensor, marca_de_tiempo"`, la cual se escribe en una nueva línea dentro del archivo "data.csv".

4. Los arreglos se reinician y se vuelve al paso número 2 de la rutina.

3.3.1.4. Problemas identificados

Falta de escalabilidad para integración de nuevos sensores

Implementadas las funcionalidades antes descritas, se hace patente un problema de escalabilidad, tanto en el código desarrollado como en el formato en que se almacenan los datos del sistema.

En primer lugar, el formato en que se almacenan los datos no permite la adición de nuevos sensores, pues las tuplas almacenadas carecen de un identificador que permita diferenciar medidas generadas por distintos sensores. Sumado a lo anterior, tampoco se cuenta con una forma de identificación inequívoca para los distintos sensores que pueden ser usados en el sistema. En segundo lugar, la implementación a nivel de código carece de una lógica de escalabilidad para la adición de nuevas rutinas, encargadas del manejo de otros sensores. De esta manera, si se desea agregar un nuevo sensor al sistema, es necesario entrar a conocer el código en su completitud e identificar las zonas en las cuales se puede modificar el proceso de recolección de datos desde los sensores.

Método de almacenamiento ineficiente

Otro problema que se identifica en ésta iteración se presenta en la rutina de almacenamiento implementada. Al almacenarse los datos en un archivo de texto, se hace necesaria la conversión de éstos desde sus tipos originales, *float* y *uint32_t*, a formato *string*. Esta conversión, que debe realizarse cada vez que se agregaban datos al archivo, genera una carga de procesamiento al sistema que puede evitarse considerando otra estrategia de almacenamiento.

De la misma manera, y a pesar de que la funcionalidad no fue implementada en esta iteración, se identifica que el problema antes descrito también afectaría al proceso de lectura de los datos, debido a que en cada lectura tendría que hacerse un proceso de conversión de tipos a la inversa, haciéndose transformaciones desde texto hacia los tipos *float* y *uint32_t*, generando nuevamente una carga de procesamiento al sistema.

Rutina de interrupción insuficiente

El último problema identificado en esta iteración corresponde a la implementación realizada de la rutina de interrupción. Si bien el funcionamiento obtenido es el esperado para el proceso de recolección de datos, la rutina implementada no considera la existencia de una rutina de envío, que al igual que la recolección debe realizarse según una frecuencia determinada.

3.3.2. Segunda iteración

3.3.2.1. Objetivos

La segunda iteración del sistema busca implementar el proceso de transmisión de datos, tarea no implementada en la anterior iteración. Para esto, se integra al sistema el Bloque Transmisor. Además, a partir de los problemas identificados en la iteración anterior, se realizan modificaciones en la implementación de algunas funcionalidades del sistema, con el fin de volver más eficiente la rutina del microcontrolador Arduino.

3.3.2.2. Funcionalidades implementadas

En esta iteración se implementan las siguientes funcionalidades, a nivel de Bloque Sensor:

- Se abstrae el proceso de lectura de sensores, mediante el uso de *callbacks*, con el fin de mejorar la integración de nuevos sensores.
- Se modifica la rutina de almacenamiento de datos, con objetivo de volver más eficiente el proceso de escritura de datos. Los datos generados se almacenan en archivos binarios.
- Se modifica la rutina de interrupción para considerar tanto el proceso de recolección de datos como el envío de éstos.
- Se implementa la comunicación con el Bloque Transmisor, mediante comunicación secuencial.
- Se implementa la lectura de datos desde la tarjeta microSD.
- Se implementa el proceso de envío de datos hacia el Bloque Transmisor.

Por otra parte, a nivel de Bloque Transmisor se implementa lo siguiente:

- Se implementa una rutina de recepción de datos enviados desde el Bloque Sensor.
- Se implementa el envío de los datos recibidos desde el Bloque Sensor hacia un dispositivo de salida.

3.3.2.3. Detalles de la implementación

Abstracción en el proceso de recolección de datos

En la primera iteración se identifica un problema de escalabilidad en el sistema, asociado a la carencia de una estructura, a nivel de código, que permitiese la fácil adición de nuevos sensores al Bloque Sensor. Para solucionar este problema, se modifica la forma en que los datos son recolectados por el sistema, abandonando la estructura vigente orientada exclusivamente al sensor DS18B20.

La estrategia implementada puede entenderse como un **sistema de registro de sensores**, el cual consiste en la asociación de un sensor con su respectiva rutina de recolección de datos. Para lograr esto, se genera una identificación única para los distintos tipos de sensores que podrían asociarse al sistema. La Tabla 3.1 muestra el conjunto de sensores que podría ser

usado en el sistema, junto con la variable física que miden y su correspondiente identificador, expresado en notación binaria.

Sensor	**Variable a medir**	**ID**
DS18B20	Temperatura	0000
SHT20 I2C	Temperatura	0001
Gravity: Analog Water Pressure Sensor	Presión	0010
Pressure Sensor HK1100C	Presión	0011
Gravity: Analog pH Sensor/Meter Kit V2	PH	0100
Gravity: Analog Electrical Conductivity Meter	Electroconductividad	0101
Gravity: Analog TDS Sensor/Meter for Arduino	Electroconductividad	0110
Grove: Turbidity Sensor (Meter) for Arduino V1.0	Turbidez	0111

Tabla 3.1: Tabla de identificación de sensores a incorporar en el prototipo del nodo sensor.

Definidos los identificadores para los sensores, se procede a abstraer una rutina de recolección de datos a nivel de código. De manera esquemática, si se tiene un sensor S, se puede definir su rutina de recolección de datos, R_S, como el conjunto de operaciones necesarias para que el sensor entregue la medida física que está registrando en un momento dado. Esta rutina R_S puede entenderse como una función que no recibe ningún parámetro, pues la rutina conoce los pasos a seguir para obtener la medida del sensor S, y retorna un valor de punto flotante, que representa el valor físico medido por el sensor.

Con todo lo anterior, el sistema de registro de sensores funciona de la siguiente manera:

1. Antes de ejecutar la rutina, se debe declarar el número N de sensores que presentará el sistema.
2. Luego, deben registrarse los sensores a ser utilizados por el sistema a través de las ID que los identifiquen. Esta información se almacena en un arreglo de enteros sin signo de 8 bit, de tamaño N.
3. Después, para cada sensor S registrado anteriormente, deben definirse sus respectivas rutinas de recolección R_S. Cada rutina debe cumplir con ser una función que no reciba parámetros y retorne un valor de punto flotante. Estas funciones se almacenan en un arreglo de punteros a funciones, también de tamaño N.
4. Realizado lo anterior, al iniciarse el sistema se generarán en RAM los dos arreglos antes declarados.
5. Cuando el sistema tenga que recolectar los datos desde los sensores, iterará sobre los dos arreglos, generando tuplas de la forma `id_sensor, valor` para cada sensor registrado.

Rutina de almacenamiento y gestión de datos

La principal dificultad identificada en la primera iteración se generaba debido a que los datos estaban siendo almacenados en formato de texto, lo cual conllevaba la ejecución de operaciones de transformación de datos que entorpecían el proceso de escrituras y lecturas. Para solucionar esto, se decide almacenar los datos recopilados por el sistema en archivos

binarios, permitiendo así la ejecución de escrituras y lecturas sin la necesidad de realizar transformaciones de tipos.

Para lograr implementar este cambio, en primer lugar se hace necesaria la definición de la unidad mínima de información que se almacenaría en un archivo, equivalente a la tupla `"valor, marca_de_tiempo"` utilizada en la primera iteración. Así, se decide representar una lectura de un sensor como una tupla compuesta por tres elementos: el valor medido por el sensor, el tiempo en que esa medida fue tomada y el ID asociado al sensor.

Con esto, se define una estructura de datos afín a la representación establecida, la cual se muestra en la Figura 3.11. Esta estructura, a la que llamaremos *Sensor Reading* o simplemente lectura, se compone de un valor de punto flotante, un entero sin signo de 32 bits y un entero sin signo de 8 bits, permitiendo así almacenar la tupla antes definida.

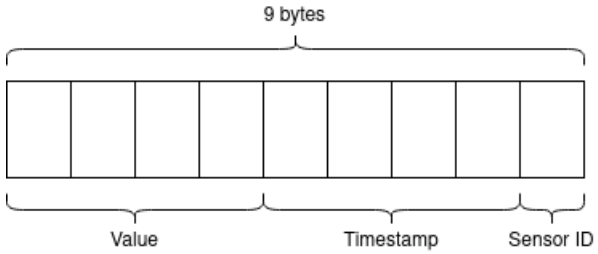

Figura 3.11: Estructura de datos *Sensor Reading*

A continuación, se analiza la forma de escribir y leer los datos en los archivos binarios a trabajar. La librería SD de Arduino, utilizada para la comunicación con el módulo microSD, utiliza un arreglo de 512 bytes para generar escrituras y lecturas de datos en la tarjeta. Esta implementación responde a un estándar de comunicación que utilizan las tarjetas SD, en que la transferencia de datos de escritura o lectura se realiza mediante bloques de 512 bytes.

Con lo anterior presente, se intenta hacer uso del arreglo utilizado por la librería SD, para modificarlo y usarlo a conveniencia del problema abordado, pero la implementación no permite una intervención directa en este arreglo. Por esta razón, y dado que no es posible definir un nuevo arreglo de 512 bytes para trabajar debido a las restricciones de memoria naturales al microcontrolador usado, se cambia la librería utilizada, pasándose a usar la librería SdFat[7]. Esta librería es la base sobre la cual se desarrolló la librería oficial de Arduino, y cuenta con funcionalidades que entregan más libertades a la hora de gestionar el almacenamiento en la tarjeta SD.

A partir de este cambio, se decide estructurar los archivos binarios como la unión de una cantidad definida de bloques de 512 bytes. A su vez, cada bloque se estructura para almacenar un conjunto de lecturas de sensores. Así, se define una estructura de datos de 512 bytes, a la que llamaremos *Data Block* o bloque de datos, consistente en un arreglo de lecturas de sensores. La Figura 3.12 muestra la estructura de cada bloque, la cual consiste en un contador, que indica la cantidad de lecturas presentes en el bloque, un arreglo de lecturas y un espacio de relleno, utilizado para alcanzar el tamaño de 512 bytes.

[7]https://www.arduino.cc/reference/en/libraries/sdfat/

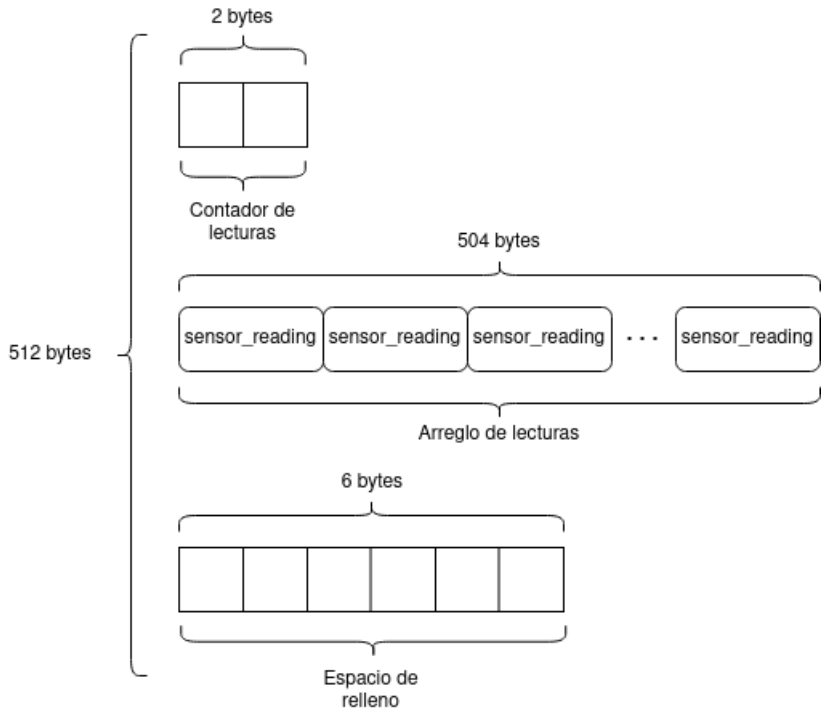

Figura 3.12: Estructura de datos *Data Block*

Ya con todo lo anterior definido, la rutina de almacenamiento se describe como sigue:

1. Antes de iniciarse el sistema, se configura un número B_{max} que representa el número de bloques máximo que puede contener un archivo dentro de la tarjeta microSD.
2. Al iniciarse el sistema, se genera un archivo de trabajo en la tarjeta microSD, llamado `"data###.bin"`.
3. Luego, se genera una estructura *Data block* en RAM, con un arreglo vacío de *Sensor Readings*.
4. A medida que se van generando lecturas de sensores, estas se almacenan en el *Data Block*, dentro del arreglo de *Sensor Readings*.
5. Cuando el arreglo está lleno, se escribe el *Data Block* en el archivo de trabajo y se reinicia el bloque.
6. Si luego de la escritura del bloque el archivo de trabajo llega al límite de bloques B_{max}, el archivo es renombrado con un nombre único y se cierra. Luego de esto, se abre un nuevo archivo de trabajo o `"data###.bin"`, y se vuelve al paso 4.

En el último paso de la rutina, se menciona el uso de un nombre único en el almacenamiento de un archivo binario. Este nombre único para cada archivo se obtiene generando nombres de manera secuencial, siguiendo el patrón `"dataXXX.bin"`, donde `XXX` es una secuencia de tres dígitos que representa un número de archivo. Este número de archivo inicia en 000, y al comenzar la ejecución del programa se actualiza, buscando dentro de la memoria microSD el archivo con el sufijo `XXX` más alto. De esta manera, el siguiente archivo a almacenar se guardará con el sufijo `XXX+1`, asegurando su unicidad dentro del sistema de archivos de la tarjeta de memoria.

Finalmente, en la Figura 3.13 se muestra un diagrama de flujo de la rutina completa de almacenamiento de datos.

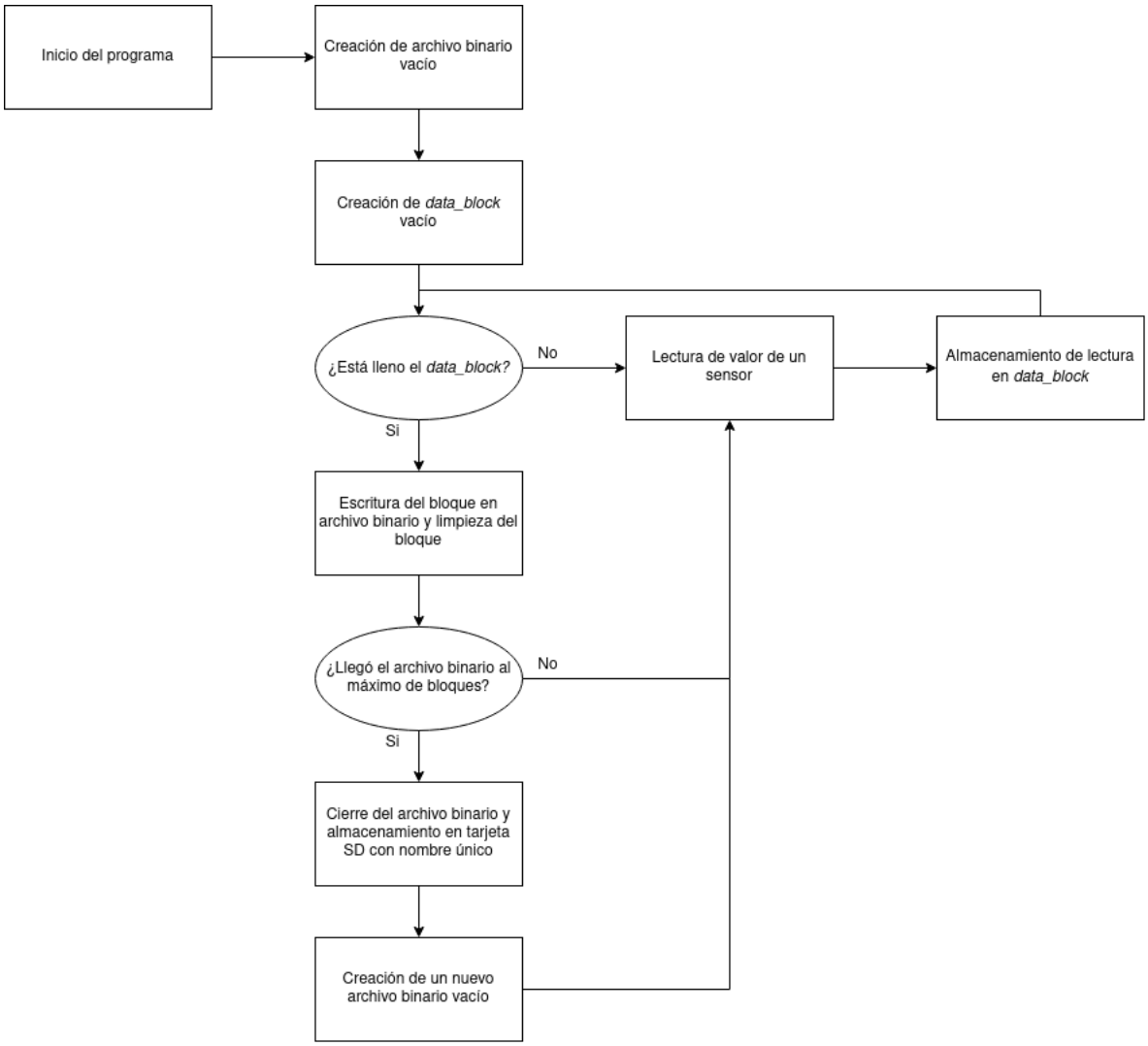

Figura 3.13: Diagrama de flujo del proceso de almacenamiento de datos

Rutina de interrupciones

En la primera iteración se implementa una rutina de interrupción, en la que el módulo RTC enviaba, cada cierto tiempo, una alarma al microcontrolador, provocando la recolección de datos desde el sensor de temperatura. A través de ésta rutina se cumple el requisito de recolección automática y frecuente de los datos por parte del sistema. El problema con esta implementación es que no considera el proceso de envío de datos hacia el Bloque Transmisor, lo cual también forma parte de los requisitos a cumplir. Para solucionar este problema, se considera la opción de incorporar una segunda alarma, generada también por el módulo RTC, la cual diera aviso del momento en que debía realizarse un envío de datos.

Las especificaciones técnicas del módulo RTC indican la existencia de dos alarmas pro-

gramables, una con una precisión de segundos, que se usa en la primera iteración, y otra con una precisión de minutos. La diferencia entre éstas dos alarmas se explica mejor con un ejemplo. La primera alarma puede ser configurada para enviarse a las 10:54:59, mientras que la segunda puede ser configurada para enviarse a las 10:54:00 o para las 10:55:00. Esta limitación descarta la idea de usar dos alarmas, pues restringiría la frecuencia de envío de datos a múltiplos de minutos.

En vista de lo anterior, el problema tuvo que solucionarse mediante la implementación de una rutina de interrupción con una única alarma, la cual fuese manejada por el microcontrolador para simular la existencia de dos alarmas, una para gatillar la recolección de datos y otra para gatillar el envío de datos.

Para explicar esta rutina, en primer lugar se denota por R al evento de recolección de datos y por E al evento de envío de datos. Además, se denota por T_R al tiempo, en segundos, que debe transcurrir entre eventos de recolección de datos. Así, por ejemplo, si $T_R = 5$, significa que cada 5 segundos deben recolectarse los datos del sensor de temperatura. De la misma manera, se denota por T_E al tiempo, en segundos, que debe transcurrir entre eventos de envío de datos. Nótese que necesariamente debe cumplirse $0 < T_R \leq T_E$, pues de otra forma se realizarían envíos vacíos, al no haberse ejecutado recolecciones previas.

Ahora bien, considerando el hecho de que para ejecutar el envío de datos debe necesariamente existir una o más ejecuciones previas de recolección de datos, podemos representar la cadena de eventos de recolección y envío de la forma:

$$\underbrace{R...R}_{n} E \underbrace{R...R}_{n} E... \tag{3.1}$$

con $n \geq 1$ la cantidad de veces que se recolectan datos previo a un envío.

Teniendo en cuenta que los eventos tanto de recolección como de envío deben ser gatillados por alarmas, se denota como $A_R(t)$ al evento de configuración de una alarma para ejecutar R dentro de un tiempo t. De la misma manera se denota como $A_E(t)$ a la configuración de la alarma para ejecutar E en un tiempo t. Con esto, se puede ver que la cadena de eventos $R...RER...RE$ sería equivalente a la cadena de alarmas:

$$\underbrace{A_R(T_R)...A_R(T_R)}_{n} A_E(T_E - nT_R) \underbrace{A_R((n+1)T_R - T_E)A_R(T_R)...A_R(T_R)}_{n} A_E(2T_E - 2nT_R) \tag{3.2}$$

De la expresión anterior podemos ver que la distancia entre cada evento de recolección R es de T_R segundos, así como la distancia entre cada evento de envío E es de T_E segundos, cumpliéndose así con las frecuencias definidas para ambos eventos. Cabe señalar que para el caso en que $T_E - nT_R = 0$ o $2T_E - 2nT_R = 0$, se entenderá que los eventos de recolección de datos y de envío de datos se ejecutarán secuencialmente.

El algoritmo implementado sigue la misma lógica anterior, generando alarmas para los eventos de recolección y envío a medida que avanza el tiempo. Para esto, el algoritmo utiliza dos medidas: el **tiempo transcurrido** entre dos eventos de envío y el **tiempo restante** para que el siguiente evento de envío ocurra. Este algoritmo funciona como sigue:

1. Antes de iniciarse el sistema se definen las variables T_R y T_E, cumpliéndose que $0 <$

$T_R \leq T_E$.

2. Al iniciarse el programa, se inicializa la variable $T_{trans} = 0$ que almacena el tiempo transcurrido desde el último envío.

3. El microcontrolador consulta la hora al RTC y configura la siguiente alarma con la siguiente subrutina:

   (a) Si $T_{trans} < T_E$, significa que aún no se realiza el siguiente envío. Por tanto, se calcula el tiempo restante para el siguiente evento de envío mediante $T_{rest} = T_E - T_{trans}$

      i. Si $T_{rest} > T_R$, se configura una alarma para recolectar datos dentro de T_R segundos. Se actualiza $T_{trans} += T_R$, y se pasa al Modo Dormido.

      ii. Si $T_{rest} < T_R$, se configura una alarma para enviar datos dentro de $T_R - T_{rest}$ segundos. Se actualiza $T_{trans} += T_R$ y se pasa al Modo Dormido.

      iii. Si $T_{rest} = T_R$, se configura una alarma para recolectar y luego enviar datos en T_{rest} segundos. Se actualiza $T_{trans} = 0$ y se pasa la Modo Dormido.

   (b) Si $T_{trans} > T_E$, significa que ya se realizó un envío. De esta manera, se calcula $T_{rest} = T_{trans} - T_E$. Con esto, se configura una alarma para recolectar datos dentro de T_{rest} segundos. Se actualiza $T_{trans} = T_{rest}$ y se pasa al Modo Dormido.

   (c) Si $T_{trans} = T_E$, significa que debe realizarse un evento de recolección y un evento de envío. Así, se configura una alarma para recolectar y luego enviar datos dentro de T_R segundos. Se actualiza $T_{trans} = 0$ y se pasa al Modo Dormido.

4. Al volver del Modo Dormido, el sistema vuelve al paso 3 de la rutina.

Para ilustrar mejor esta implementación, se adjunta la Figura 3.14, donde se esquematiza la anterior rutina.

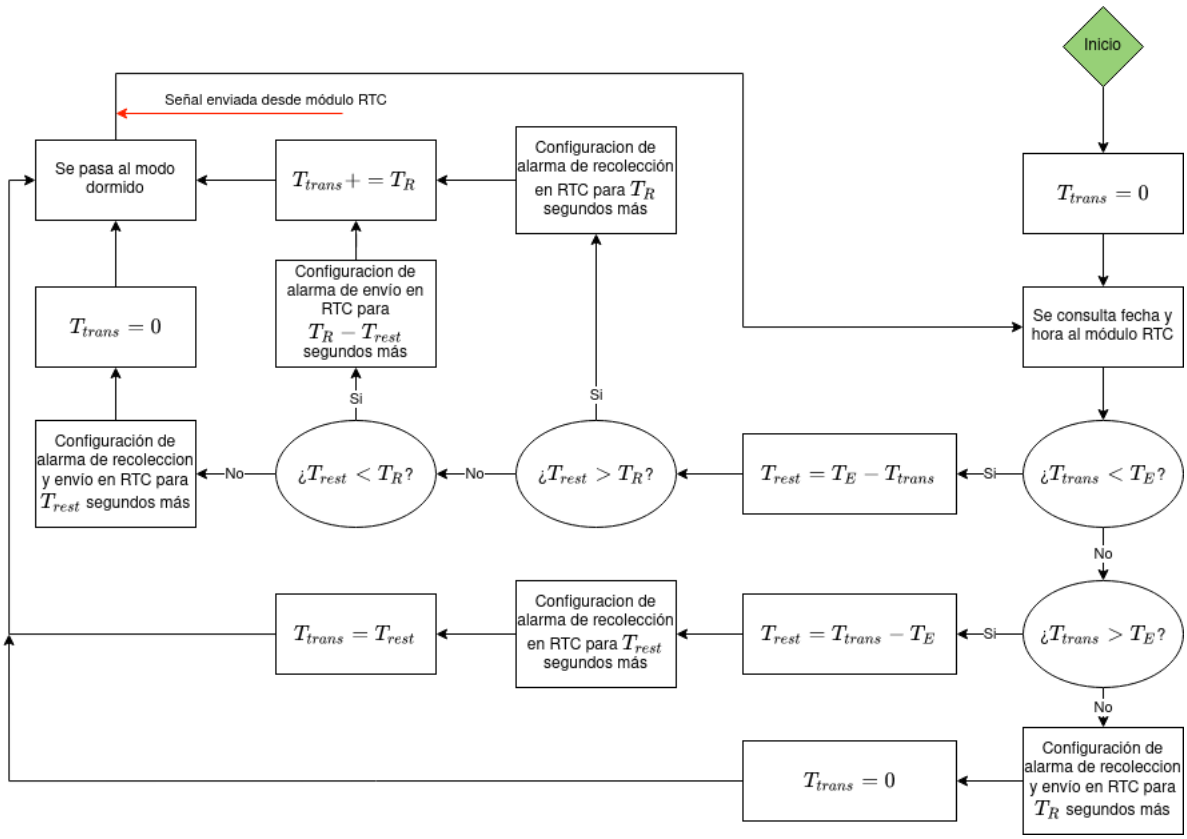

Figura 3.14: Rutina de interrupciones implementada en la segunda iteración.

Rutina de lectura y envío de datos hacia Bloque Transmisor

Teniendo implementadas la lógica de recolección y almacenamiento de datos, en conjunto con la rutina de alarmas para gatillar tanto la recolección como el envío de éstos, la siguiente funcionalidad a implementar es el envío de los datos recolectados por el sistema hacia el Bloque Transmisor inalámbrica.

Para explicar la rutina implementada, es necesario recordar las estructuras involucradas en el almacenamiento datos del sistema. En primer lugar, tenemos el **bloque de datos**, estructura presente en RAM que almacena las lecturas que se van realizando en el sensor. En segundo lugar tenemos un **archivo de trabajo**, archivo presente en la memoria microSD en el cual se escriben los bloques de datos que se van llenando a medida que se realizan lecturas de sensor. Por último, se tienen los **archivos presentes en memoria microSD**, los cuales son archivos almacenados previamente a lo largo de la ejecución del programa. La Figura 3.15 ilustra estos elementos, separando su localidad en el sistema y las estructuras que puedan tener asociadas.

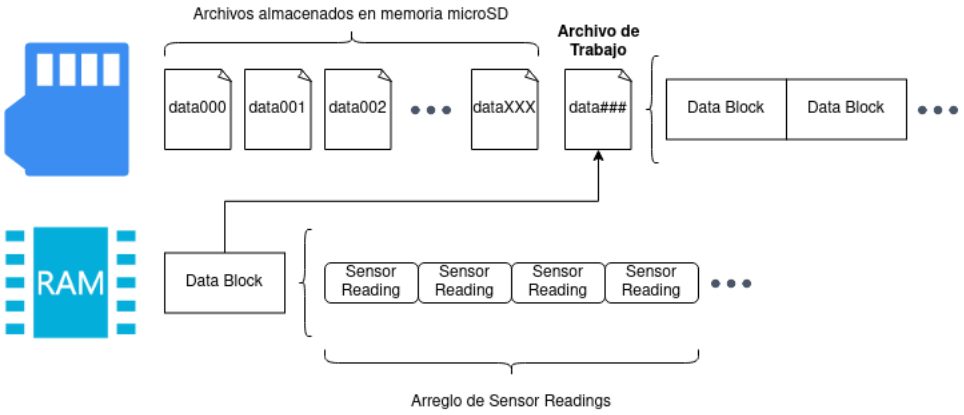

Figura 3.15: Elementos usados en el proceso de almacenamiento de datos.

Por otra parte, la rutina implementada se aprovecha de la naturaleza secuencial de los datos trabajados y de su forma de almacenamiento. En palabras simples, dado que los datos son recolectados con una frecuencia fija a lo largo del tiempo, estos pueden ordenarse de manera creciente con respecto al momento en que fueron recolectados. Por otra parte, los datos son almacenados en arreglos, los cuales luego se almacenan en archivos que se generan de manera creciente. Considerando esto, se puede asegurar que los archivos mantienen el orden de los datos. Así, en la Figura 3.15, los datos presentes en el archivo `data000` serán más antiguos que los datos presentes en el archivo `data001`.

La rutina de envío implementada sigue la lógica de etiquetar los datos según enviados y no enviados. Esta diferenciación se simplifica al ser los datos secuenciales, pues se puede generar un marcador sobre un único dato que establezca la línea divisoria entre los dos estados. De esta manera, si denotamos por D_{i} al dato generado en la i-ésima lectura del sensor de temperatura, podemos representar los M datos presentes en el sistema al momento de realizar un envío como una cadena de la forma

$$D_1 D_2 ... D_{M-1} D_M \tag{3.3}$$

Ahora bien, asumamos que los datos no enviados comienzan en el dato D_n con $1 < n < M$. Entonces, podemos dividir los datos presentes en el sistema de la forma

$$\underbrace{D_1 ... D_{n-1}}_{enviados} \underbrace{D_n ... D_M}_{no\ enviados} \tag{3.4}$$

Con lo anterior, el problema a solucionar se reduce al envío de los datos $D_n ... D_M$, los cuales pueden estar repartidos entre archivos ya almacenados, el archivo de trabajo y el bloque de datos presente en RAM.

Para solucionar esto, la rutina implementada utiliza tres variables llamadas `file_ptr`, `block_ptr` y `pos_ptr`, las cuales funcionan como punteros hacia el primer dato no enviado en el sistema. Para lograr esto, se sigue la siguiente lógica:

- La variable `pos_ptr` indica la posición dentro del bloque de datos presente en RAM del primer dato no enviado.

- Por otra parte, la variable `block_ptr` indica el número de bloque dentro del que se encuentra el primer dato no enviado.

- Por último, la variable `file_ptr` se utiliza para indicar el archivo en que se encuentra el primer dato no enviado.

La rutina de envío depende de estas tres variables para enviar los datos marcados como no enviados de manera ordenada. Estas variables se configuran a medida que los datos son recolectados y almacenados por el sistema y su configuración se realiza siguiendo la rutina ilustrada en la Figura 3.16.

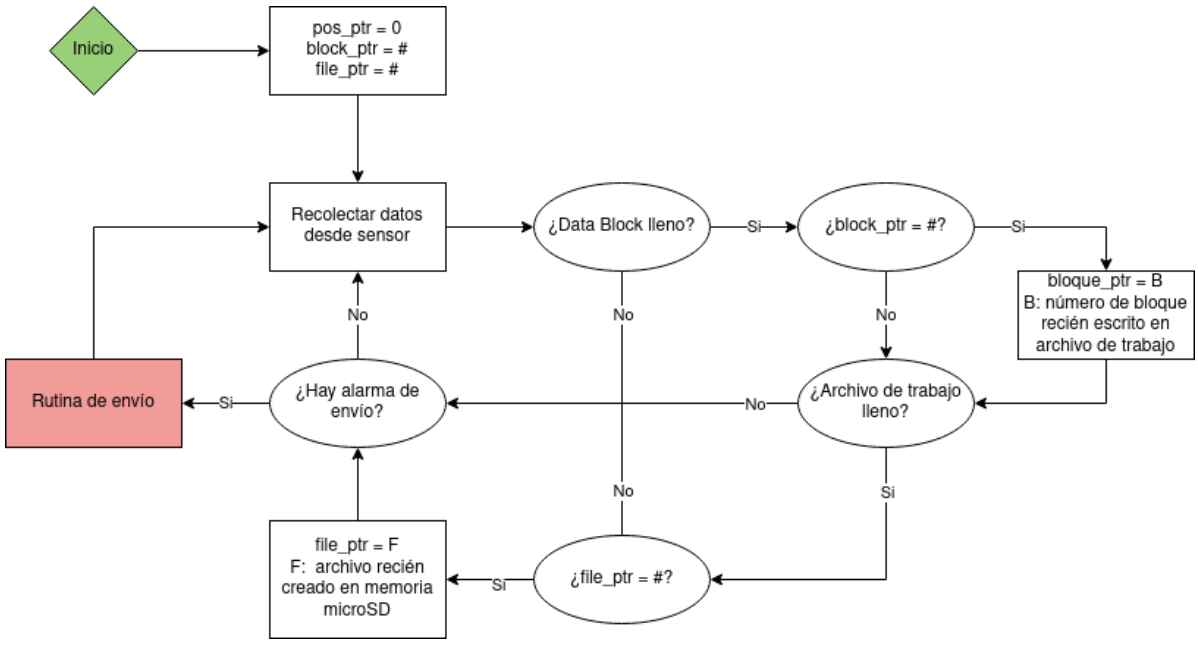

Figura 3.16: Rutina de asignación de las variables de envío en el proceso de recolección y almacenamiento de datos.

En la Figura 3.16 se puede ver una casilla roja etiquetada como `Rutina de envío`. Ésta, es la rutina que se ejecuta cuando se configura una alarma de envío de datos y la encargada del envío de todos los datos que estén marcados como no enviados por el sistema. La Figura 3.17 ilustra el algoritmo implementado para ejecutar esta tarea.

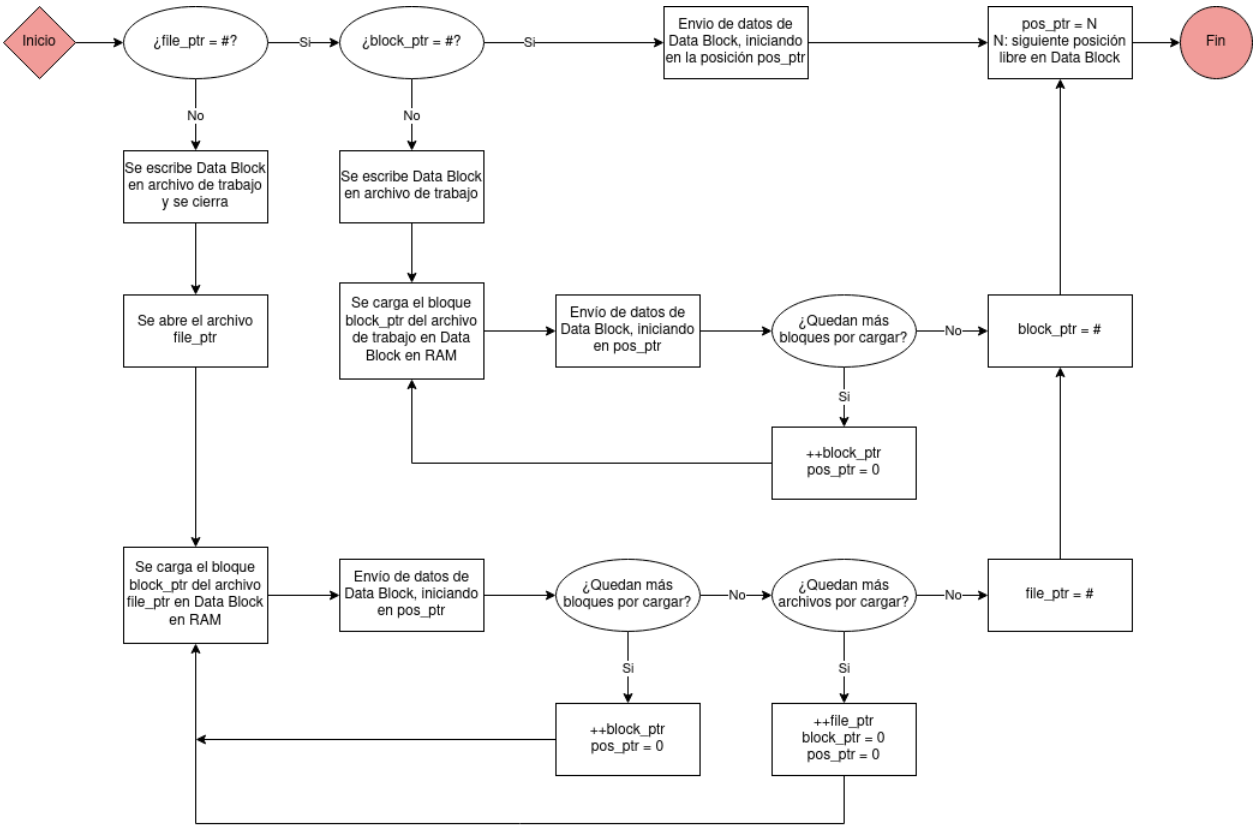

Figura 3.17: Rutina de lectura y envío de datos hacia el Bloque Transmisor.

De esta manera, combinando ambas rutinas mostradas anteriormente, se implementa la rutina de lectura y envío de datos hacia el Bloque Transmisor.

Rutina de recepción y envío de datos en Bloque Transmisor

Ya con el envío de datos desde el Bloque Sensor hacia el Bloque Transmisor implementado, se procede a implementar la rutina de envío de datos desde el Bloque Transmisor hacia un dispositivo de salida. Como se mencionó en la sección 3.2.4, la transmisión de datos desde el Bloque Transmisor se realiza de manera inalámbrica, usando comunicación LoRa, luego de recibir datos desde el Bloque Sensor. Para esto, se implementa la siguiente rutina:

1. Antes de iniciarse el sistema, se configura un tiempo de espera en segundos del Bloque Transmisor al que denotaremos T_S. Además, se configura el tamaño en bytes de los paquetes LoRaWAN que serán transmitidos hacia el dispositivo de salida, al que denotaremos por P_{LoRa}.
2. Al iniciarse el sistema, se configura un socket de comunicación LoRaWAN entre el Bloque Transmisor y el *Gateway* LoRa dentro de la red. Por otra parte, se configura la interfaz UART del microcontrolador LoPy4 para recibir la comunicación secuencial con el Bloque Sensor.
3. El Bloque Transmisor consulta si hay datos esperando a ser leídos en la compuerta

UART:

- En caso de haber datos, estos se leen y se dividen en trozos de 9 bytes, tamaño de una estructura *Sensor Reading*. Estos trozos se empaquetan en mensajes LoRaWAN, los cuales no deben superar los P_{LoRa} bytes de tamaño. Luego, los paquetes se transmiten a través del socket LoRaWAN hasta que no queden trozos por transmitir.
- En caso de no haber datos, el dispositivo se duerme durante T_S segundos.

4. Terminado lo anterior, se vuelve al paso 3 de la rutina.

Como puede verse a través de la rutina de envío implementada, el comportamiento del Bloque Transmisor se asemeja al comportamiento de un servidor, considerando que solo realiza acciones cuando recibe datos transmitidos desde el Bloque Sensor. Esta decisión de diseño se basa en el consumo energético asociado a la placa LoPy4, la cual presenta un mayor consumo que un Arduino Nano, debido tanto a su mayor capacidad de procesamiento como al lenguaje sobre el que trabaja.

3.3.2.4. Problemas identificados

Errores y corrupción de la tarjeta microSD

A lo largo de la implementación tanto de la rutina de almacenamiento como de la rutina de envío de datos, se presentan inconvenientes en algunos de los procesos de gestión de archivos dentro de la tarjeta microSD. En particular, el proceso de creación de archivos en algunas ocasiones corrompe la tarjeta microSD, dejando los archivos dentro de ésta inaccesibles o corruptos. La Figura 3.18 muestra los archivos presentes en una tarjeta corrompida, luego de ejecutarse la rutina de almacenamiento de datos.

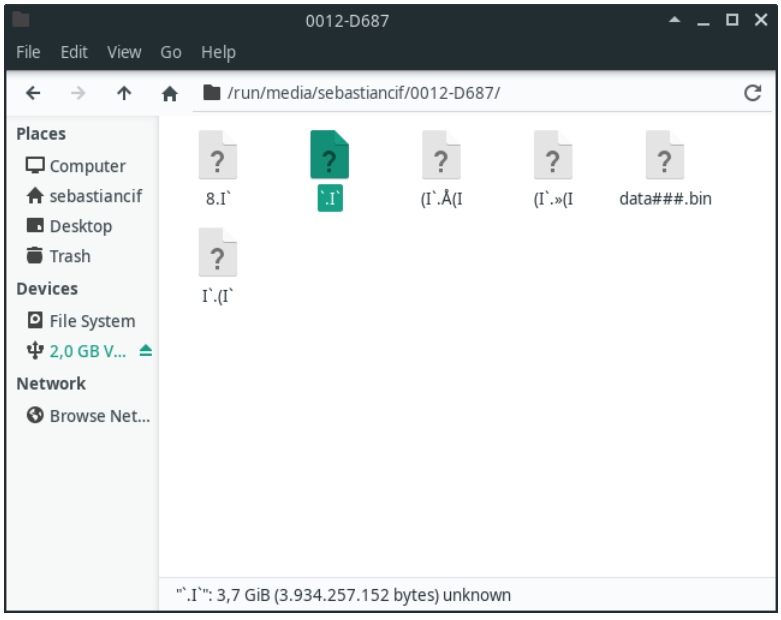

Figura 3.18: Archivos dentro de una tarjeta microSD corrompida.

En comparación, la Figura 3.19 muestra el sistema de archivos de una tarjeta microSD no corrupta, luego de ejecutarse la rutina de almacenamiento corregida. Nótese la existencia en ambos casos del archivo de trabajo `data###.bin` y la diferencia significativa entre el tamaño de un archivo correcto -1 KB- y el de un archivo corrupto -3.7 GB-.

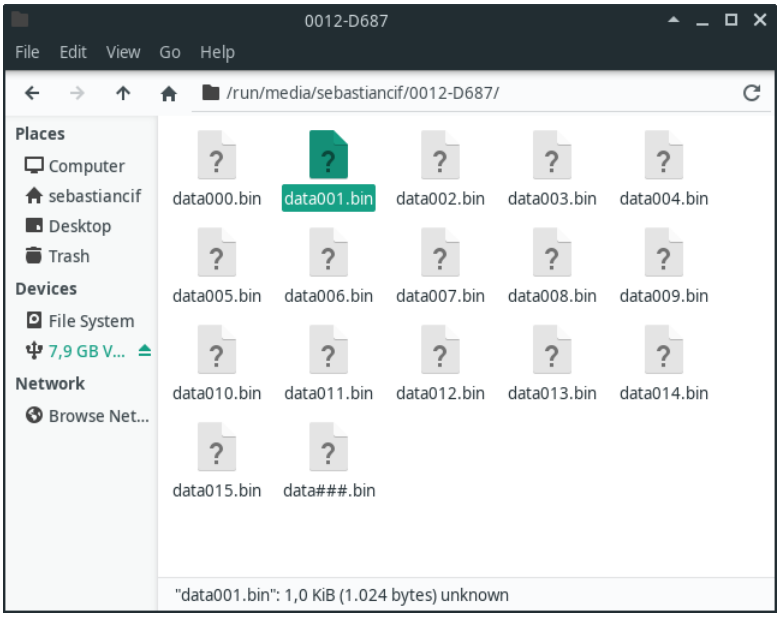

Figura 3.19: Archivos dentro de una tarjeta microSD sana.

Por otra parte, también se presentan errores en tiempo de ejecución a lo largo de las rutinas que gestionaban archivos. Así, por ejemplo, en ocasiones la tarjeta microSD no es reconocida por el sistema o el sistema es incapaz de escribir datos en la tarjeta. A lo largo de la implementación se buscan e identifican las razones por las que estos eventos ocurren.

Por una parte se identificaron **razones deterministas** que generaban errores y corrupción en la tarjeta microSD utilizada. Estas razones se explican debido a una mala implementación, a nivel de código, de una rutina de escritura, lectura o creación de archivos. Así, por ejemplo, la tarjeta microSD se corrompía si se intentaba abrir un nuevo archivo, sin cerrar correctamente uno abierto con anterioridad, o presentaba errores si se intentaba escribir un bloque de memoria más allá del límite establecido para un archivo. Estos errores se corregían a medida que se encontraban a lo largo del código.

Por otra parte, se identificaron **razones no deterministas** que generaban errores a la hora de interactuar con la tarjeta microSD, llegando también a corromperla. Se les llama no deterministas debido a que se presentaban de manera aleatoria a lo largo de una ejecución del sistema y no respondían a una lógica a nivel de ejecución del programa implementado. Así, por ejemplo, al ejecutar una versión funcional de la rutina de almacenamiento de manera repetida, se presentaban ocasiones en que la tarjeta microSD reportaba incapacidad para escribir o leer datos. En otras oportunidades, al ejecutar una versión funcional de la rutina de almacenamiento, la cual no corrompía la tarjeta microSD usada, ésta se corrompía de todas formas en alguna ejecución posterior.

Estos errores ralentizaron la implementación de la segunda iteración, pues, por una parte, la corrupción de la tarjeta microSD se considera como un error crítico en el sistema, por lo que la solución de este problema no podía pasarse por alto. Por otra parte, la aparición de estos errores no deterministas se atacó en una primera instancia desde una perspectiva de software, pues era el único elemento que mutaba a lo largo del trabajo realizado.

Finalmente, se determina que esta clase de errores respondían a cuestiones de hardware y no de software. En particular, estos errores se asociaron al uso de una tarjeta de prototipado o *protoboard*, las cuales pueden presentar fallas a la hora de usarse debido a una serie de factores, tales como suciedad en las conexiones, soldaduras sueltas, conexiones flojas, entre otras. Por ejemplo, se presentaron ocasiones en que el sistema no era capaz de detectar la tarjeta microSD al ejecutar su rutina, problema que se solucionaba moviendo los cables del sistema. De esta manera, al aparecer esta clase de errores, y teniendo descartados errores de software, se procedía a buscar su corrección mediante el uso de distintos pines dentro de la *protoboard* o cambiando los cables utilizados en el circuito hasta lograr el funcionamiento esperado del sistema.

Ausencia de detección de errores en la transmisión hacia el Bloque Transmisor

Otro problema identificado en esta iteración se relaciona con la corrupción de datos a la hora de transmitir. Al transmitir datos desde un dispositivo a otro, estos necesariamente deben viajar a través de un medio físico para llegar a su destino. Esta transmisión puede verse afectada por perturbaciones del medio físico en que viaja o mal funcionamiento de los componentes de hardware que participan en la transmisión, afectando la integridad de los datos que son transmitidos.

Los datos que viajan desde el Bloque Sensor hacia el Bloque Transmisor se transmiten mediante comunicación secuencial, utilizando las interfaces UART de los dos microcontroladores usados en el sistema. El problema de esta comunicación es que cuenta con un algoritmo de detección de errores con baja capacidad para identificar corrupciones en los datos enviados.

La interfaz UART ofrece un algoritmo de detección de errores llamado **bit de paridad**, el cual indica la paridad o imparidad de la cantidad de unos que se transmiten en una cadena de bits. De esta manera, si quien envía los datos indica que la cantidad de unos en el mensaje es par, y al llegar a destino la cantidad de unos del mensaje es impar, entonces el receptor del mensaje sabe que la data no llegó íntegra a destino. Si bien esta verificación permite identificar errores en la transmisión, su capacidad es limitada, pues es susceptible a errores de transmisión más complejos. Por ejemplo, si se transmite la cadena de bits '1010', su bit de paridad indicará una cantidad par de unos. Ahora, si la cadena se corrompe y se transforma en '1111', el receptor verificará que existe una cantidad de unos par en el mensaje, y no será capaz de detectar que la integridad del mensaje se vio comprometida en la transmisión.

Con lo anterior presente, se identifica la necesidad de implementar un algoritmo para verificar la integridad de los mensajes transmitidos desde el Bloque Sensor hacia el Bloque Transmisor.

Rutina de recepción y envío de datos ineficiente

El último problema identificado en esta iteración se encuentra en la rutina implementada en el Bloque Transmisor. Si bien esta rutina cumple con la tarea de recibir los datos desde el Bloque Sensor y transmitirlos de manera inalámbrica hacia el dispositivo de salida de la red, presenta limitaciones en su implementación que son mejorables.

El principal problema de la rutina implementada es la fuerte dependencia que tiene su ejecución con el tiempo de espera T_S que se define antes de iniciar el programa. Si este tiempo es muy alto, el Bloque Transmisor puede generar acumulaciones de datos en la interfaz de entrada de la comunicación, generando cuellos de botella o pérdida de los datos que no alcanzan a ser leídos. Por el contrario, si el tiempo de espera es bajo, el Bloque Transmisor consultará con alta frecuencia la recepción de datos, generando un gasto innecesario de energía para el sistema, volviéndose energéticamente ineficiente.

Por otra parte, la lectura de datos desde la interfaz UART y su envío inmediato también puede generar demoras a la hora de recibir los datos desde el Bloque Sensor, pues los tiempos de transmisión de los datos hacia el dispositivo de salida pueden ser variables, dependiendo de las condiciones de la red y la cantidad de datos que se necesite a transmitir.

3.3.3. Tercera iteración

3.3.3.1. Objetivos

La tercera iteración tiene por objetivo desarrollar mejoras sobre lo implementado en las iteraciones anteriores. Este enfoque se toma en vista de que las funcionalidades elementales del sistema estaban implementadas, sin embargo podían realizarse mejoras a ciertos procesos.

De esta manera, uno de los objetivos de esta iteración es la mejora de los procesos de transmisión de datos desde el Bloque Sensor hacia el Bloque Transmisor. Por otra parte, esta iteración considera también la implementación de mejoras a nivel de la eficiencia energética del sistema. Estos objetivos además se relacionan fuertemente con algunos de los problemas identificados en la iteración pasada, por lo que la resolución de éstos problemas también se ve implicada en esta iteración.

3.3.3.2. Funcionalidades implementadas

En esta iteración se implementan las siguientes modificaciones en el sistema:

- Se implementa una unidad reducida de comunicación, generada a nivel de Bloque Sensor.
- Se implementa un algoritmo de detección de errores para la transmisión entre Bloque Sensor y Bloque Transmisor.
- Se modifica la rutina de recepción y envío de datos en el Bloque Transmisor.
- Se define un protocolo de transmisión de datos entre Bloque Sensor y Bloque Transmi-

sor.

3.3.3.3. Detalles de la implementación

Unidad reducida de comunicación

En la segunda iteración se definió la estructura de datos *Sensor Reading*, ilustrada en la Fig 3.11. Esta estructura de 9 bytes se implementó para representar una lectura de un sensor, almacenando una tupla de la forma `[valor, marca_de_tiempo, ID]`, y se estableció como la unidad mínima de información que se almacenaría en el sistema.

Considerando lo anterior, se evalúa la posibilidad de definir una estructura de datos destinada a la transmisión, la cual fuese de menor tamaño que una *Sensor Reading*, para de esta manera reducir la cantidad de información que debe ser transmitida a la hora de realizarse un envío de datos.

Ahora, considerando que una *Sensor Reading* se estableció como la tupla mínima que define una lectura de sensor, es necesario analizar de qué manera se podía reducir el espacio utilizado para representar esta información. Por una parte, se ve que el campo `ID` toma como valor máximo el número 7, como puede verse en la Tabla 3.1, por lo que puede representarse en menos de 1 byte de información. Luego se ve que el campo `marca_de_tiempo` debe necesariamente almacenarse utilizando 4 bytes de información, pues este es el tamaño mínimo para representar tiempo con precisión de segundos utilizando el sistema Unix. Con lo anterior, se determina que la única manera de reducir el tamaño de la información transmitida era representar el campo `valor` en menos bytes.

El campo `valor` se representa en una *Sensor Reading* mediante un valor de punto flotante con signo, almacenado en 4 bytes de información, teniendo capacidad para representar valores en el rango de $\pm 1{,}18 * 10^{-38}$ a $\pm 3{,}4 * 10^{38}$, con una cantidad de aproximadamente 7 cifras significativas. Este rango de representación escapa, en primer lugar, al rango de valores posibles que podrían ser medidos por un sensor en el sistema, dadas las magnitudes de los parámetros fisicoquímicos trabajados. Por otra parte, a nivel de proyecto se estimó que una precisión de punto flotante de una cifra significativa era suficiente para representar los valores a medir desde las fuentes de agua en estudio.

Por estas razones, se decide representar el campo `valor` utilizando 2 bytes de información, reservando 1 bit para signo y 15 bits para representación numérica, estableciéndose el rango de valores a representar entre $[-32767, 32767]$. Ahora bien, esta representación por sí sola se hace insuficiente, pues no permite la representación de un valor real con una cifra significativa de precisión. Por esta razón, se establece que la codificación traería implícita la precisión de un decimal esperada, de manera tal que, por ejemplo, el número 32767 representaría el valor real 3276,7, reduciéndose así la magnitud del rango de valores representables al rango $[-3276{,}7, 3276{,}7]$. Este rango, si bien es menor al rango inicial, es suficiente para representar los valores esperados a medirse dentro del funcionamiento normal del Nodo Sensor, tanto a nivel de magnitud como de precisión.

Con lo anterior, se define una nueva estructura de datos llamada *Sensing Unit*, la cual

codifica la lectura de un sensor en 7 bytes, dividiéndose éstos en dos partes, una cabecera con información de la lectura y un *payload*, donde se almacena la información. La Figura 3.20 muestra esta estructura a nivel de bytes.

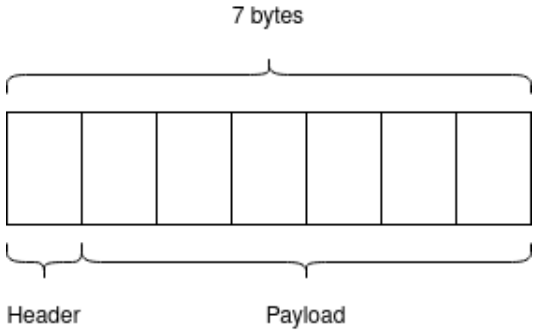

Figura 3.20: Estructura de datos *Sensing Unit*.

La cabecera de esta estructura se definió originalmente para almacenar solo la ID del sensor, pero como se mencionó previamente, la ID del sensor puede almacenarse en menos de 1 byte, por lo que se define un campo llamado *Coding Info*, el cual da información del tipo de codificación que acarrea el respectivo *Sensing Unit*. Esta decisión se toma considerando la posibilidad de implementar otros tipos de codificación que puedan aplicarse también sobre el sistema. Por otra parte, se reservan 2 bits dentro de la subestructura, dejando la posibilidad de agregar otra información que se quiera transmitir.La Figura 3.21 muestra la estructura a nivel de bits de esta subestructura.

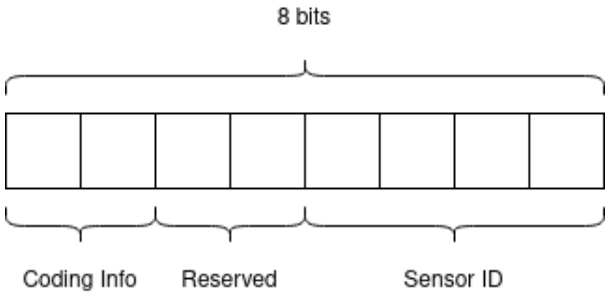

Figura 3.21: Subestructura *Header* de una *Sensing Unit*.

Por último, el *payload* de esta estructura se compone de una marca de tiempo, representada en 4 bytes, y el valor medido por un sensor, codificado en 2 bytes según las reglas planteadas previamente. La Figura 3.22 muestra el detalle en bytes de la subestructura.

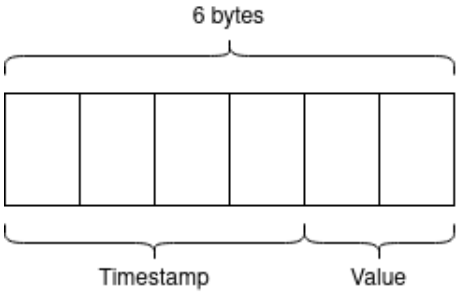

Figura 3.22: Subestructura *Payload* de una *Sensing Unit*.

De esta manera, se define una estructura de datos dedicada a la transmisión, la cual permite transportar la información de una lectura de sensor en 7 bytes, en lugar de los 9 bytes usados en la iteración anterior. Para lograr esta transformación, se agregó un paso extra previo al envío de datos hacia el Bloque Transmisor, el cual consiste en la transformación de un conjunto de *Sensor Readings* a un conjunto de misma cardinalidad de *Sensing Units*, reduciendo así el tamaño de la información transmitida.

Algoritmo de detección de errores

Uno de los problemas identificados en la iteración anterior fue la necesidad de implementar un algoritmo para la detección de errores que puedan ocurrir en la transmisión de datos desde el Bloque Sensor hacia el Bloque Transmisor. Para esto, se busca en la literatura información con respecto a los distintos algoritmos utilizados para la detección de errores en la transmisión de datos.

De esta búsqueda, se determina que el método más común de realizar detección de errores en la transmisión es a través del uso de *checksums*. Estos *checksums* pueden generarse de distintas maneras y ofrecer distintas capacidades de detección de errores, dependiendo del algoritmo generador que se utilice.

Para comparar las capacidades de detección de errores, una métrica utilizada es la **Distancia de Hamming** o HD por sus siglas en inglés. La HD de un algoritmo de *checksum* se define como el número más pequeño de errores de bit para el cual existe al menos un caso en que el *checksum* no es capaz de detectar la falla. Por ejemplo, el **bit de paridad** mencionado en la iteración anterior es un tipo de *checksum* con una HD = 2, pues si en la transmisión de un mensaje determinado se generaran 2 errores de bit dentro de éste, el bit de paridad seguirá estando correcto, fallando así la detección de la falla presente en el mensaje.

Con lo anterior, se evalúa el uso de dos algoritmos para la generación de *checksum*, los cuales ofrecen la mayor capacidad de detección de errores. Por una parte se considera utilizar un algoritmo CRC, de sus siglas en inglés *Cyclic redundancy check* y por otra el algoritmo de *checksum* de Fletcher.

El *algoritmo de CRC*, es realmente una familia de algoritmos, los cuales se definen por dos parámetros a elegir en su implementación. El primero es el largo en bits del *checksum* o

CRC a generar, el cual define también el nombre del algoritmo. De esta manera, si se desea generar un *CRC* de 8 bits de largo, el algoritmo se llama CRC-8. El segundo parámetro a elegir es un polinomio característico que se usará para la generación del *checksum*. Este polinomio no es más que una palabra binaria, la cual se interpreta como un polinomio de un grado determinado. La lógica detrás del algoritmo es la de tratar a los mensajes binarios como polinomios binarios y generar un *checksum* a partir del resto generado a partir de la división polinomial del mensaje por el polinomio característico del algoritmo.

A modo de ejemplo, supongamos que se tiene un mensaje binario M el cual será transmitido y un algoritmo CRC con un polinomio característico P. El algoritmo generará un *checksum* basado en el resto generado tras la división del polinomio equivalente $M(x)$ por el polinomio equivalente $P(x)$, es decir:

$$M(x) = Q(x)P(x) + R(x) \tag{3.5}$$

donde $Q(x)$ es el cociente de la división y $R(x)$ el resto. Calculado esto, se transmite MR, agregando el *checksum* calculado al final del mensaje.

Ahora, supongamos que el receptor recibe un mensaje $M'R$. Para verificar la integridad del mensaje, el receptor utiliza el mismo algoritmo de CRC con polinomio característico P calculando:

$$M'(x) = Q'(x)P(x) + R'(x) \tag{3.6}$$

De esta manera, si $R' = R$ significa que la integridad del mensaje no se vio comprometida. En caso contrario, el receptor puede determinar que el mensaje recibido no es el mismo que el mensaje enviado.

Por otra parte, el algoritmo de *checksum* de Fletcher [21] es un algoritmo de detección de errores que se plantea como una opción más eficiente computacionalmente a un algoritmo CRC, pero con menor capacidad de detección de errores. Este algoritmo se basa en el cálculo de un *checksum* aritmético, el cual utiliza aritmética modular para su generación. Para esto, el mensaje al cual se desea calcular el *checksum* se trabaja como un arreglo de enteros de un tamaño determinado.

Al igual que el algoritmo CRC, el algoritmo de Fletcher se define a partir de parámetros a elegir en su implementación. El primer parámetro es el tamaño K, en bits, de los enteros presentes en el mensaje a trabajar. De esta manera, el mensaje se trabaja como una secuencia de enteros de K-bits. El segundo parámetro es el valor del módulo M a utilizar en la aritmética modular. El tercer parámetro es la cantidad de enteros de K-bits que se utilizarán en el *checksum* a generar. El planteamiento original del algoritmo utiliza $K = 8$, calzando con el tamaño estándar de un byte, y $R = 2$, generando así un *checksum* de 16 bits de tamaño. Además, en [21] se estudia el uso de $M = 2^K$ y $M = 2^K - 1$, siendo este último valor el que mejor capacidad de detección de errores ofrece.

La generación del *checksum* para un mensaje $M = b_1 b_2 ... b_n$, con b_i un entero de K-bits, se realiza a través del cálculo de R sumas denotadas por $C(r)$ con $r = 0, ..., R-1$, las cuales se generan a medida que se lee el mensaje M. De esta manera, a la hora de leer el entero b_i,

las R sumas se actualizan de la forma

$$\begin{aligned} C(0) &= C(0) + b_{\mathrm{i}} \pmod{M} \\ C(1) &= C(1) + C(0) \pmod{M} \\ &\ldots \\ C(r) &= C(r) + C(r-1) \pmod{M} \\ &\ldots \\ C(R-1) &= C(R-1) + C(R-2) \pmod{M} \end{aligned} \tag{3.7}$$

Al terminar de leer el mensaje, los valores $C(r)$ calculados se utilizan para generar el *checksum*, el cual se transmite en conjunto con el mensaje original. Luego, y de igual manera que en el caso de CRC, el receptor computa el *checksum* con el mensaje recibido y verifica que los valores correspondan al *checksum* generado por el remitente.

Finalmente, en el planteamiento del algoritmo en [21] se plantean las capacidades de detección éste, concluyendo que el algoritmo alcanza HD= 3 para palabras de un tamaño no mayor a $K(2^K - 1)$ o palabras de un tamaño mayor, pero en que los errores de dos bits se presenten a una distancia menor que $K(2^K - 1)$.

Con lo anterior expuesto, se debe decidir el algoritmo a utilizar en el sistema. Para esto, se considera que los datos transmitidos desde el Bloque Sensor hacia el Bloque Transmisor viajarán en palabras de 49 bytes de largo, tamaño de la conjunción de 7 estructuras *Sensing Unit*. Esta decisión se toma debido, por una parte, a la implementación realizada para la codificación de las lecturas de sensor a *Sensor Reading* explicada anteriormente, y por otra parte considerando que el paquete LoRaWAN más pequeño que se puede transmitir tiene una capacidad de 51 bytes.

De esta manera, se evalúa el uso del algoritmo de Fletcher y un algoritmo CRC, considerando la necesidad de generar un *checksum* para una palabra de 392 bits de largo. En la literatura se encuentra un estudio [2] que realiza una selección de los mejores polinomios para algoritmos CRC. Esta selección se realiza considerando las capacidades de detección de error obtenidas, mediante el uso de HD como métrica, para tamaños de *CRC* dados y largos de palabra máximo en que se mantienen las capacidades de detección. La Figura 3.23 muestra la tabla resumen de selección de polinomios realizada en el estudio.

Max length at HD Polynomial	CRC Size (bits)													
	3	4	5	6	7	8	9	10	11	12	13	14	15	16
HD=2	2048+ 0x5	2048+ 0x9	2048+ 0x12	2048+ 0x21	2048+ 0x48	2048+ 0xA6	2048+ 0x167	2048+ 0x327	2048+ 0x64D	–	–	–	–	–
HD=3		11 0x9	26 0x12	57 0x21	120 0x48	247 0xA6	502 0x167	1013 0x327	2036 0x64D	2048 0xB75	–	–	–	–
HD=4			10 0x15	25 0x2C	56 0x5B	119 0x97	246 0x14B	501 0x319	1012 0x583	2035 0xC07	2048 0x102A	2048 0x21E8	2048 0x4976	2048 0xBAAD
HD=5						9 0x9C	13 0x185	21 0x2B9	26 0x5D7	53 0x8F8	none	113 0x212D	136 0x6A8D	241 0xAC9A
HD=6							8 0x13C	12 0x28E	22 0x532	27 0xB41	52 0x1909	57 0x372B	114 0x573A	135 0xC86C
HD=7									12 0x571	none	12 0x12A5	13 0x28A9	16 0x5BD5	19 0x968B
HD=8										11 0xA4F	11 0x10B7	11 0x2371	12 0x630B	15 0x8FDB

Figura 3.23: Tabla de mejores polinomios para tamaños de CRC dados y largos máximo de palabra, en bits. Tabla extraída de [2].

De la figura anterior, podemos ver que para alcanzar un HD= 3 para una palabra a codificar de 392 bits de tamaño, es necesario el uso de un CRC de 9 bits de largo.

Además, y nuevamente usando la literatura, se llega a [22], en que se comparan los algoritmos de *checksum* más comunes utilizados en el contexto de sistemas embebidos. En este estudio se determina que el costo computacional de un algoritmo CRC es aproximadamente el doble que el del algoritmo de Fletcher, mientras que la capacidad de detección de errores del primer algoritmo es órdenes de magnitud mayor que la del segundo. A su vez, sugiere la existencia de métodos de optimización de la computación del algoritmo CRC a través del uso de tablas de búsqueda, las cuales precomputan los valores a generar entre el polinomio característico del algoritmo y las palabras de bits esperadas como entrada. Así, por ejemplo, una tabla de búsqueda para un algoritmo de CRC-8 almacena 256 valores de 8 bits cada uno, los cuales representan todas las posibles salidas frente a palabras de 8 bits de entrada.

Con todo lo anterior, se toma la decisión de utilizar el algoritmo de Fletcher, considerando el uso de un *checksum* de 16 bits. Esta decisión se sustenta principalmente en el menor costo computacional que presenta el algoritmo de Fletcher, el cual supone un menor gasto energético para el sistema, al tener que realizar menos operaciones para la generación de *checksums*. Si bien esta diferencia de costo computacional puede ser resuelta mediante el uso de tablas de búsqueda para el algoritmo CRC, se descarta esta opción al evaluarse la limitación de recursos disponibles en el sistema.

Protocolo de transmisión de datos entre Bloque Sensor y Bloque Transmisor

Con las dos implementaciones anteriormente descritas, cambió el tamaño y la forma de los datos a enviar entre Bloque Sensor y Bloque Transmisor, por lo que fue necesario definir un protocolo de comunicación entre ambos Bloques.

Para esto, en primer lugar se establece que el Bloque Sensor transmitiría los datos en bloques de máximo siete *Sensing Unit*, con un tamaño máximo de 49 bytes por bloque. Esto

se define debido a la necesidad de realizar la transformación de *Sensor Reading* a *Sensing Unit*, lo cual requiere un espacio en RAM destinado a este proceso. De esta manera, se considera que 49 bytes era un costo que el Bloque Sensor podía asumir, además de ser el tamaño máximo de datos que pueden ser transmitidos en el paquete LoRaWAN más pequeño que puede usarse.

Agregado a lo anterior, se determina que cada bloque de datos debía transmitirse junto a su respectivo *checksum*, generado a partir del algoritmo de Fletcher implementado. Así, el tamaño máximo de una transmisión generada por el Bloque Sensor tendría un tamaño de 51 bytes, considerando los 2 bytes asociados al *checksum*.

De esta manera, se establece que para realizar una transmisión de M *Sensing Unit*, el Bloque Sensor debe asegurar el envío de la máxima cantidad de bloques de 51 bytes posible, teniendo el último bloque la libertad de tener un tamaño menor a 51 bytes. Esto último se tiene pues el valor M de datos a transmitir no necesariamente es un valor múltiplo de 7. Con esto, si consideramos $M = 7Q + R$ se tendrá una transmisión de Q bloques de 51 bytes y un último bloque de $7R + 2$ bytes, cuando el valor de R sea distinto de 0. La Figura 3.24 ilustra la estructura de transmisión antes descrita.

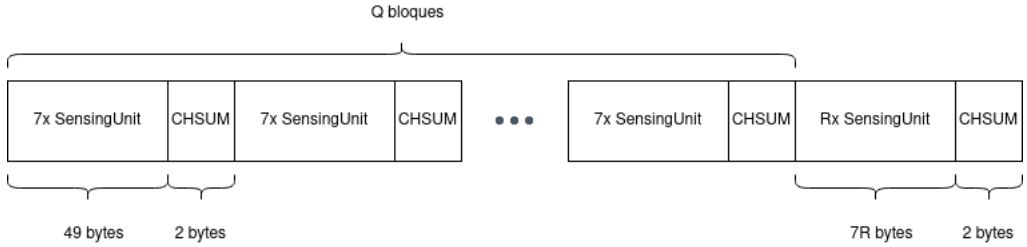

Figura 3.24: Estructura de una transmisión de datos desde el Bloque Sensor hacia el Bloque Transmisor.

Con lo anterior establecido, el Bloque Transmisor puede recibir los datos enviados por el Bloque Sensor, verificar la integridad de cada uno de los bloques y descartar aquellos datos que no serán transmitidos hacia el dispositivo de salida.

Rutina de recepción y envío de datos en Bloque Transmisor

En la iteración anterior se determinó la necesidad de modificar la rutina implementada en el Bloque Transmisor, debido a elementos de la implementación que la hacían ineficiente.

De esta manera, se modifica la rutina de recepción y envío de datos de este Bloque, considerándose también las implementaciones realizadas en esta iteración. Los cambios fundamentales realizados a la rutina son los siguientes:

- Se elimina el tiempo de espera T_S, el cual determinaba el tiempo que el Bloque Transmisor iba a dormir antes de consultar por la existencia de datos transmitidos desde el Bloque Sensor.

- Se agrega una rutina de interrupción, similar a la implementada en el Bloque Sensor. De esta manera, el Bloque Transmisor se mantiene en un estado de bajo consumo, o Modo Dormido, y se activa mediante una señal enviada por el Bloque Sensor cuando éste necesita comenzar una transmisión de datos.
- Se modifica la forma de recepción de los datos, cambiando la lógica de recepción y envío inmediato. Se implementa en su lugar una recepción continua de todos los datos que ingresan al Bloque Transmisor, leyendo el puerto de entrada de datos hasta que no haya más datos por recibir. Para asegurar esto, se implementa el uso de un tiempo de espera T_w por los datos recibidos. De esta manera, si pasados T_w segundos no existen nuevos datos por leer a través de la entrada de datos, se asume que todos los datos del Bloque Sensor ya fueron enviados.

Para mayor claridad, la rutina implementada se describe en la Figura 3.25.

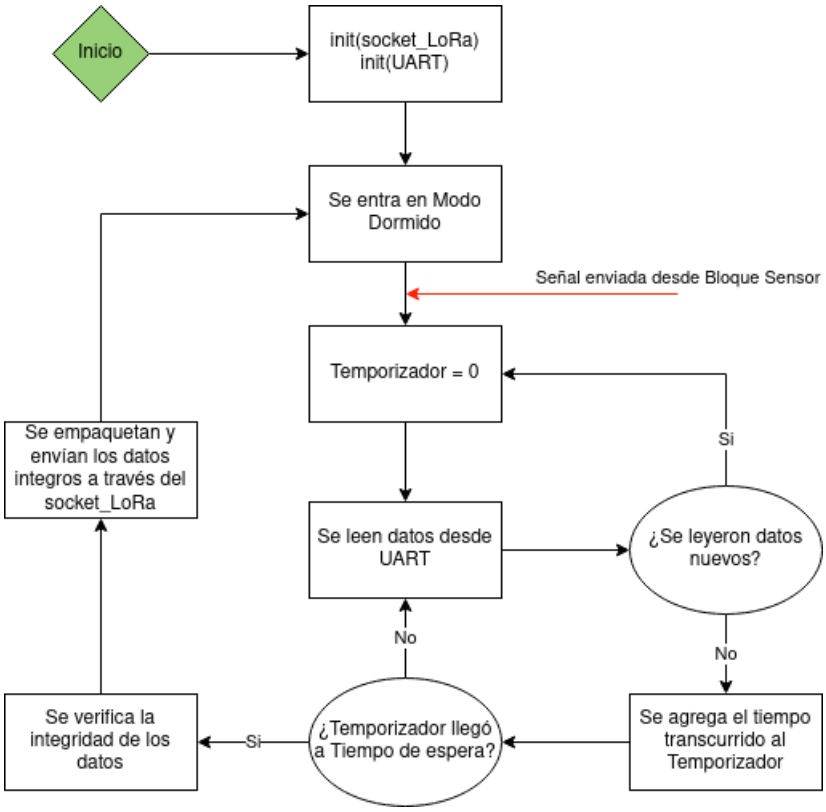

Figura 3.25: Rutina de recepción y envío de datos del Bloque Transmisor

3.4. Estrategias de reducción consumo

Ya con la implementación funcional de un Nodo Sensor, se procede a buscar estrategias para reducir el consumo de energía en el proceso de recolección y transmisión de datos. Para esto, se estudia el consumo energético esperado de los distintos procesos involucrados en la ejecución del sistema. Dentro de estos, el proceso más caro, en términos de energía, se encuentra a la hora de transmitir los datos de manera inalámbrica desde el Bloque Transmisor. Esto se debe principalmente a la necesidad de energizar la antena utilizada para realizar la

transmisión inalámbrica, y mantenerla encendida durante el tiempo que sea necesario para enviar todos los datos a transmitir.

De esta manera, se determina que una reducción en la cantidad de información a transmitir podría disminuir el tiempo de encendido de la antena del Bloque Transmisor, reduciendo con ello el consumo energético del sistema. Sin embargo, esta reducción debía considerar las distintas limitaciones existentes en el sistema, considerando también las restricciones presentes en el proceso de transmisión de datos desde el Bloque Sensor al Bloque Transmisor.

A partir de este enfoque, se diseñan e implementan tres técnicas de codificación de los datos a ser transmitidos, una de las cuales fue presentada en la sección 3.3.3, cuando se definió la unidad reducida de comunicación. A esta codificación la llamaremos Codificación Base, pues será el piso de comparación y la unidad de codificación por defecto a la hora de transmitir información.

3.4.1. Aspectos generales de las codificaciones

En una primera instancia, se considera la idea de implementar codificaciones de datos que generaran paquetes de información de tamaño variable, para de esta manera reducir al máximo posible la información a enviar. Esta idea se descarta debido a requerimientos del proyecto, en el cual se determina el uso de la estructura *Sensing Unit* como unidad mínima de transmisión.

Considerando esta restricción, llamaremos **codificación** al proceso de transformar un conjunto de *Sensor Readings* en un conjunto de *Sensing Units*, el cual puede tener una cardinalidad menor o igual a la cardinalidad del conjunto de lecturas.

Recordemos que una *Sensing Unit* es una estructura de datos de 7 bytes, los cuales se dividen en una cabecera de 1 byte, la cual almacena información de la estructura, y un *payload* de 6 bytes. Por lo tanto, las codificaciones implementadas deben restringirse a trabajar en el campo *payload* de la estructura, teniendo la libertad de gestionar los 6 bytes de cualquier manera. Por otra parte, la cabecera cuenta con un campo *Coding Info*, en el cual se puede almacenar información de la codificación usada.

Usando lo anterior, hablaremos de un *payload* X o *header* Y, cuando nos refiramos a estos campos codificados según el método X o Y, respectivamente. Por ejemplo, al referirnos a *payload* Base, nos referiremos a una subestructura *payload* que respeta la estructura de codificación Base. De la misma manera, al decir a *header* Diferencial, nos referiremos a un *header* cuya *Coding Info* corresponda a la codificación Diferencial.

3.4.2. Codificación Base

Llamaremos codificación Base a la codificación que transforma una estructura *Sensor Reading* en una estructura *Sensing Unit*, comprimiendo el valor de punto flotante de la primera a una versión de 2 bytes de tamaño. Para esto, el valor contenido en la *Sensor Reading* a codificar se representa usando 1 bit de signo y 15 bits para la representación del

valor, trabajando así en el rango de $[-32767, 32767]$. Además, el valor representado se debe interpretar como una representación en punto fijo en base 10 con un dígito para la parte fraccionaria. A modo de ejemplo, el valor codificado 127 debe interpretarse como el número 12,7. De esta manera, el rango de representación que posee la codificación Base para los valores reales codificados es $[-3276{,}7, 3276{,}7]$, con pasos de 0,1 entre cada valor.

El campo *payload* de esta estructura se compone de una marca de tiempo Unix almacenada en 4 bytes y una representación en punto fijo con un dígito para la parte fraccionaria almacenada en dos bytes. La Figura 3.26 muestra la estructura del *payload* Base.

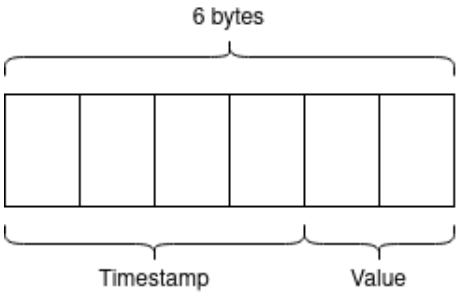

Figura 3.26: Estructura de un *Payload* Base

Algoritmo de codificación

El algoritmo de codificación Base es sencillo, al tratarse de una transformación 1 a 1 entre estructuras. La Figura 3.27 contiene un esquema para explicar éste algoritmo.

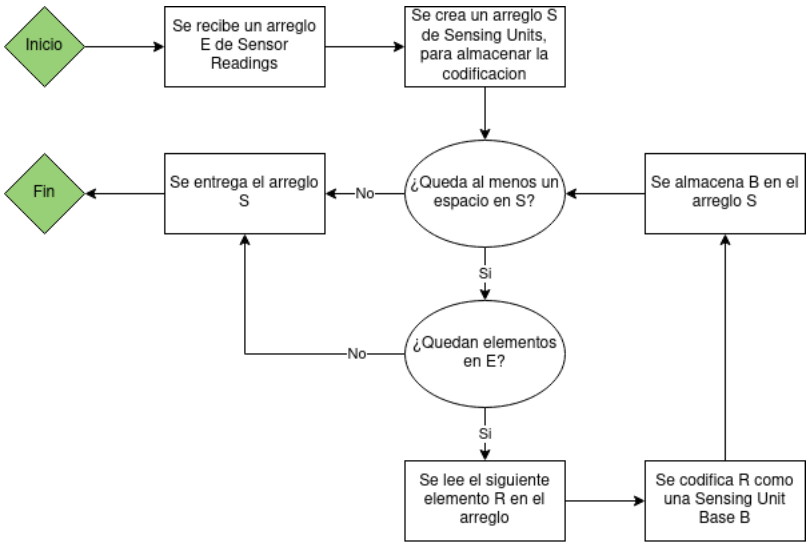

Figura 3.27: Rutina de codificación Base para un conjunto de *Sensor Readings* dado

De lo anterior, se puede ver que el algoritmo de codificación es de orden $\mathcal{O}(N)$ para un arreglo de N *Sensor Readings* como entrada.

Mejor y peor caso

Esta codificación se comporta de igual forma para cualquier entrada de datos, generando siempre, a partir de un conjunto de N *Sensor Readings*, un conjunto de N *Sensing Units*, reduciendo el tamaño total de las medidas de $9N$ bytes a $7N$ bytes. De esta manera, el mejor y peor caso son el mismo.

3.4.3. Codificación Repetición

Llamaremos codificación Repetición a una codificación formulada para trabajar sobre lecturas de sensor cuyos valores se repiten a lo largo del tiempo. La finalidad de esta codificación es ahorrar el envío de información redundante a través de la red.

Para lograr esto, la codificación aprovecha la naturaleza secuencial y frecuente de los datos generados por el sistema. En otras palabras, los datos del sistema se generan a través de lecturas de sensor frecuentes y equidistantes en el tiempo. De esta manera, se puede ahorrar parte de la información transmitida si en la codificación de ésta se aprovecha esta característica de los datos.

Con esto en mente, el *payload* de esta codificación, mostrado en la Figura 3.28, almacena dos valores, un intervalo de tiempo, que representa el intervalo de tiempo que separa a las medidas codificadas entre sí, y un valor de repeticiones, que indica la cantidad de veces que se repitió un determinado valor.

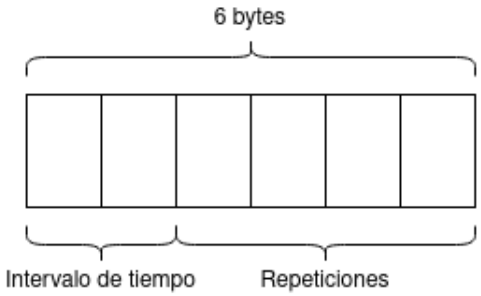

Figura 3.28: Estructura de un *Payload* Repetición

Como se puede notar, el *payload* de esta codificación no contiene la información suficiente como para representar una lectura de un sensor, por lo que se hace necesario el uso de más información para que la codificación sea de utilidad. Para esto, la codificación utiliza dos estructuras *Sensing Unit* adyacentes para representar la repetición de información. De esta manera, en primer lugar se utiliza una *Sensing Unit* con un *payload* Base y un *header* Repetición, la cual almacena el valor medido que debe ser repetido. A continuación, se agrega una *Sensing Unit* con codificación Repetición. Esto se ilustra en la Figura 3.29, se muestra una cadena de 14 bytes, representando dos *Sensing Unit*, donde la primera *Sensing Unit*, mostrada de color naranjo, acarrea la información de la lectura repetir, mientras que en color azul se muestra la *Sensing Unit* acarreando la información de repetición de la lectura.

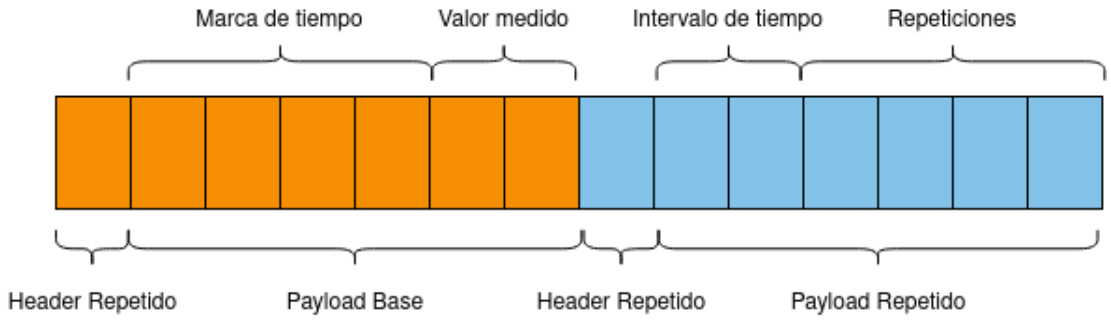

Figura 3.29: Uso de dos *Sensing Unit* para la codificación Repetición

De esta manera, a través de la conjunción de dos *Sensing Unit* se puede representar la aparición repetida de un mismo valor.

Notemos que, dado que se necesitan 14 bytes para representar la información, esta codificación se vuelve efectiva cuando un valor aparece repetido 3 o más veces, de otra manera la codificación necesita el mismo o más espacio que la codificación base para representar la información.

Algoritmo de codificación

El algoritmo de codificación Repetición implementado se basa en la búsqueda de 3 *Sensor Reading* consecutivas cuyos valores sean iguales. En caso de que esta condición no se cumpla, las lecturas se transforman mediante codificación Base. La Figura 3.30 muestra el algoritmo implementado para esta codificación.

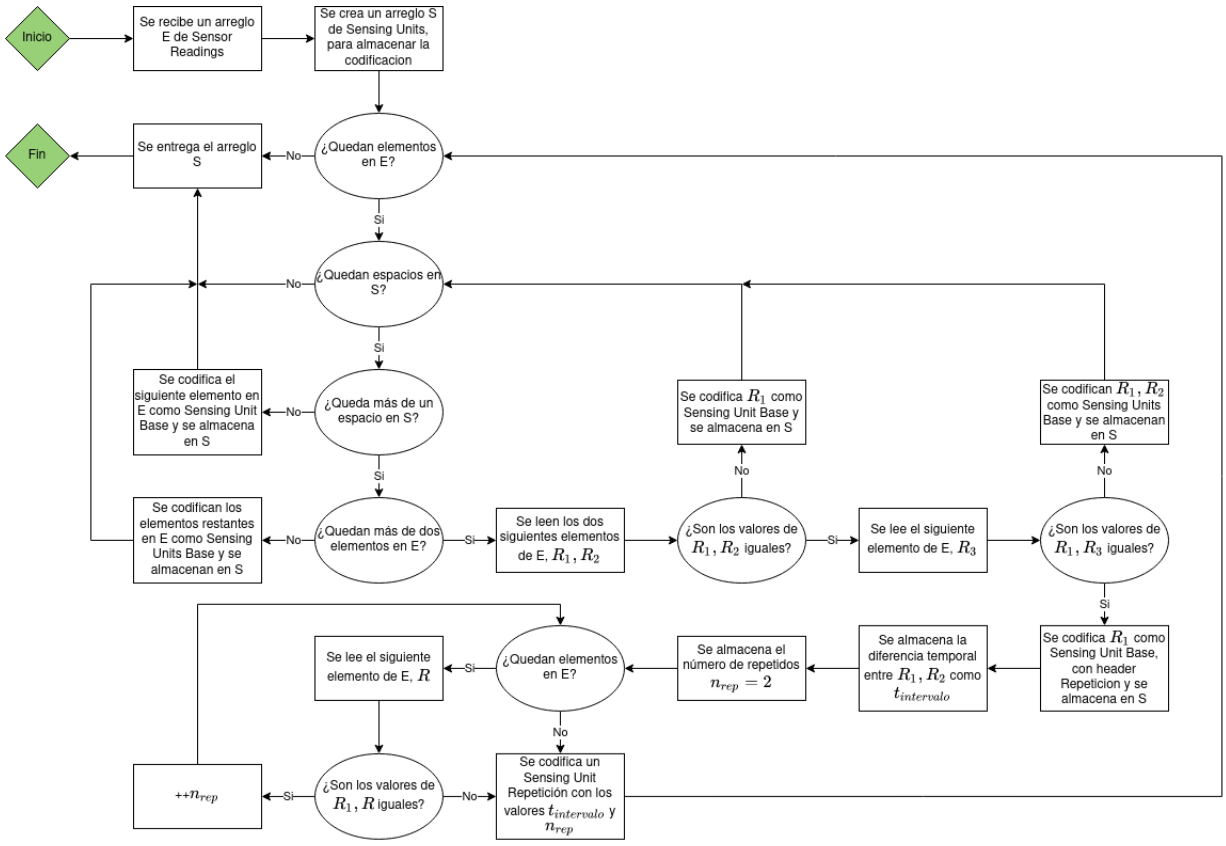

Figura 3.30: Algoritmo de codificación Repetición para un conjunto de *Sensor Readings* dado

Como se puede ver, el algoritmo de codificación implementado recorre el arreglo de *Sensor Readings* a codificar de manera ordenada, generando una cantidad constante de consultas en cada paso. De esto, podemos ver que el anterior algoritmo es de orden $\mathcal{O}(N)$ para un arreglo de N *Sensor Readings* como entrada.

Mejor y peor caso

Esta codificación depende fuertemente de la repetición adyacente de un valor en el tiempo. Si denotamos por C_R a un conjunto de N *Sensor Readings* a ser codificadas, el **peor caso** de la codificación Repetición se presenta cuando todos los valores de las lecturas adyacentes son distintos entre sí. De esta manera, la codificación genera un conjunto de N *Sensing Unit*, entregando el mismo resultado que la codificación Base.

Por el contrario, si consideramos el **mejor caso** de la codificación, el cual sería la repetición de un mismo valor a lo largo de todo el conjunto C_R, genera un conjunto de 2 *Sensing Unit*, reduciendo el tamaño total de las medidas representadas de $9N$ bytes a 14 bytes.

3.4.4. Codificación Diferencial

La codificación Diferencial, basada en la técnica propuesta en [18], es una codificación pensada para representar lecturas de sensores cuyas medidas varían poco a lo largo del tiempo. Para esto, se plantea la representación de los valores mediante el uso de diferencias con respecto a un valor de referencia.

A modo de ejemplo, supongamos que se desea representar la secuencia de valores

$$260,\ 261,\ 262,\ 256$$

Estos valores podrían representarse usando como referencia el primer valor de la secuencia y utilizando la diferencia numérica que el resto de los valores tenga con éste. De esta forma, la secuencia se representaría como

$$260,\ +1,\ +2,\ -4$$

Ahora, si pensamos en el tamaño necesario para almacenar la secuencia original, vemos que necesitamos 2 bytes para representar cada número, sumando un total de 8 bytes para la secuencia completa. Por otra parte, la secuencia generada con diferencias puede almacenarse en 5 bytes, destinando 2 bytes para la representación del primer número, y 1 byte para las diferencias numéricas con éste.

Con lo anterior presente, se diseña un *payload* para esta codificación que almacena cuatro valores, los cuales representan diferencias numéricas de dos lecturas, tanto de tiempo como de valor, con respecto a un *Sensing Unit* de referencia. La Figura 3.31 muestra la estructura de este *payload*, donde los campos `DifTime` representan diferencias temporales de segundos y los campos `DifVal` representan diferencias numéricas de valores medidos por un sensor.

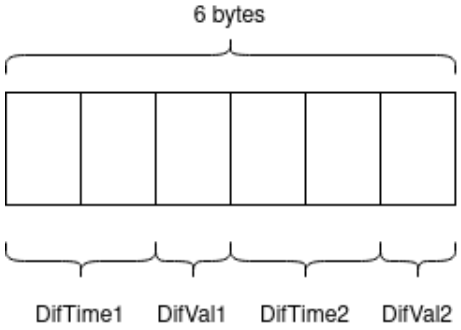

Figura 3.31: Estructura de un *payload* Diferencial

Cabe señalar que los campos `DifTime` y `DifVal` están restringidos en su capacidad de representación por el tamaño que poseen.

De esta manera, el campo `DifTime` está pensado para representar diferencias temporales

en el rango de 0 y 65535 segundos, lo cual permite representar una diferencia máxima de aproximadamente 18 horas de distancia. Además, la codificación considera que las lecturas a codificar estarán ordenadas de manera creciente con respecto al tiempo en que se tomaron, por lo que no considera la opción de representar diferencias negativas.

Por otra parte, el campo `DifVal` está pensado para representar valores en el rango de -127 y 127. Además, el valor almacenado en este campo sigue la misma regla de codificación que la utilizada en la codificación Base, la cual consiste en representar un número real con un decimal de precisión. Así, por ejemplo, el número 127 debe interpretarse como 12.7 y el número -1 debe interpretarse como -0.1. Con esto, la codificación se restringe a representar valores que no estén más distantes en magnitud que 12.7.

Al igual que la codificación Repetición, la codificación Diferencial utiliza una *Sensing Unit* con un *payload* Base y un *header* Diferencial para representar a la lectura que servirá como referencia para la codificación. La diferencia de esta codificación con la de Repetición radica en el uso de una cantidad cualquiera de *Sensing Unit* Diferenciales adyacentes, las cuales almacenan las diferencias necesarias para representar las *Sensor Readings* codificadas. Para explicar esto, en la Figura 3.32 se representa la estructura de un mensaje con codificación Diferencial. En naranjo se representa la *Sensing Unit* utilizada como referencia y en verde se representan las *Sensing Unit* Diferenciales adyacentes.

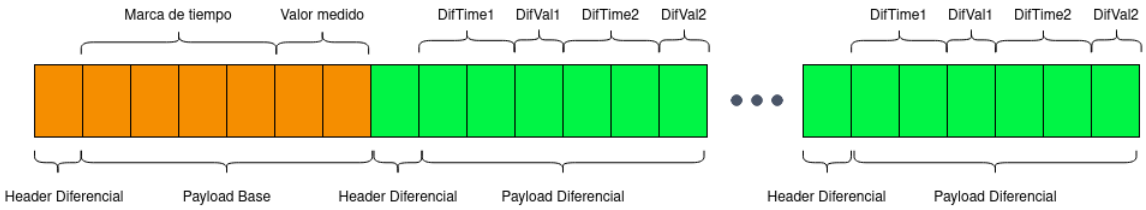

Figura 3.32: Estructura de un mensaje generado con codificación Diferencial

De esta manera, la codificación representa una serie de *Sensor Readings* a partir de las diferencias que éstas presentan con un valor de referencia dado al principio de la codificación.

Cabe señalar que esta codificación se vuelve de utilidad siempre y cuando la cantidad mínima de *Sensor Readings* a codificar sea 3, codificando estas en 14 bytes en lugar de los 21 bytes que se usarían según la codificación Base. En caso contrario, el tamaño resultante será el mismo que el de la codificación Base.

Algoritmo de codificación

La lógica del algoritmo de codificación Diferencial se asemeja a la presente en la codificación Repetición, con la diferencia de que la restricción de igualdad se reemplaza por la restricción de distancia numérica. La Figura 3.33 ilustra el algoritmo implementado.

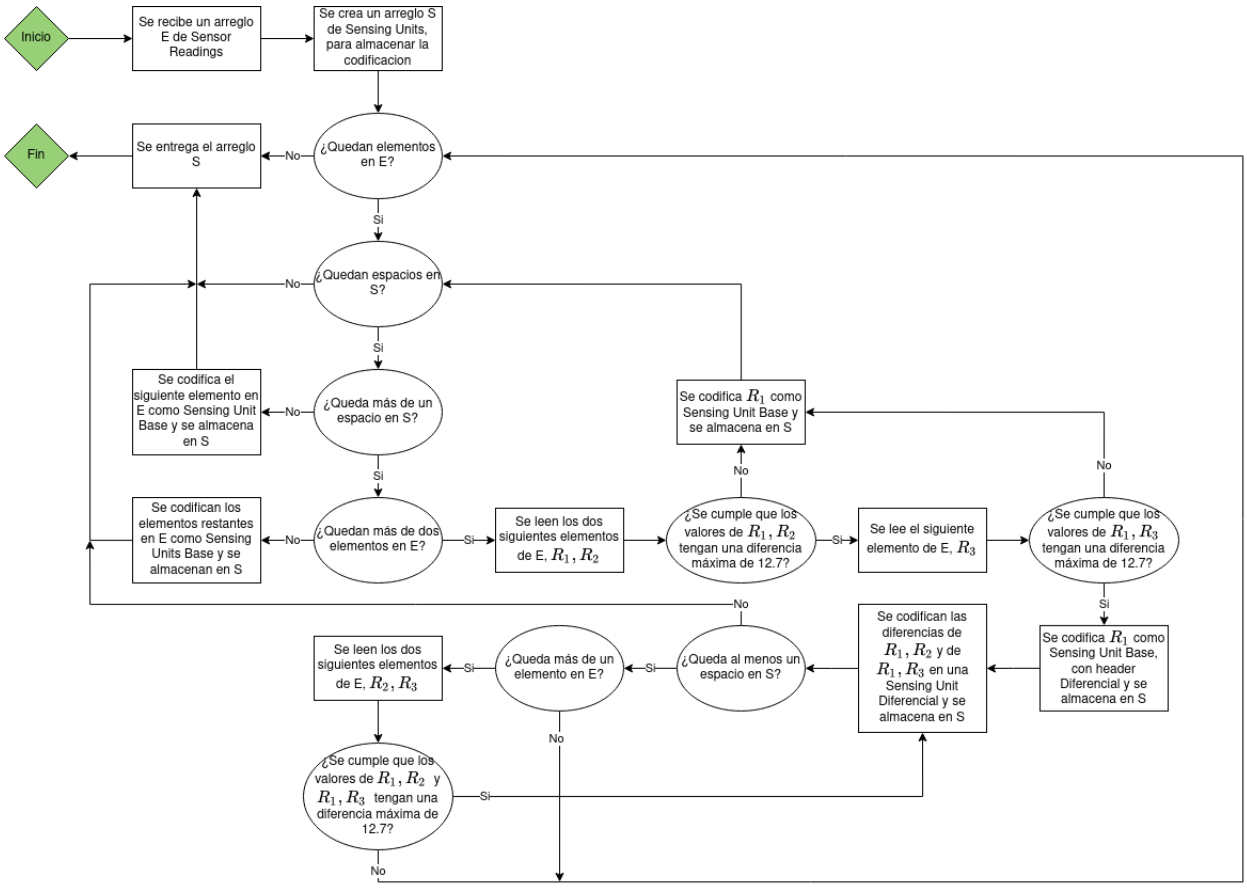

Figura 3.33: Algoritmo de codificación Diferencial para un conjunto de *Sensor Readings* dado

Como puede verse, la estructura del algoritmo es muy similar a la estructura del algoritmo usado para la codificación Repetición, recorriéndose el arreglo de *Sensor Readings* de manera ordenada y generando un número de consultas constante a medida que se avanza a través del arreglo. De esta manera, el algoritmo implementado es de orden $\mathcal{O}(N)$ para un arreglo de N *Sensor Readings* como entrada.

Mejor y peor caso

La codificación Diferencial depende de los valores que contengan las *Sensor Readings* a codificar, debido a la restricción de las diferencias numéricas representables por su *payload*. Además, esta restricción se aplica a través de los valores presentes en *Sensor Readings* adyacentes entre sí.

Con esto, si se tiene un conjunto ordenado C_D de N *Sensor Readings* a codificar, el **peor caso** de la codificación se dará cuando todos los elementos adyacentes del conjunto a codificar no cumplan con la restricción de diferencia numérica de la codificación. En otras palabras, si $C_D = c_1, c_2, ..., c_N$, con sus respectivos valores de lectura $v_1, v_2, ..., v_N$, entonces se cumplirá que $\forall \mathrm{i} \in [0, N-1], |v_\mathrm{i} - v_{\mathrm{i}+1}| > 127$. En este caso, la codificación genera un conjunto de N *Sensing Unit*, igualando al resultado generado por la codificación Base.

Por otra parte, el **mejor caso** de la codificación se obtiene cuando todos los elementos adyacentes cumplen con la restricción de diferencia numérica. Para este caso, la codificación del conjunto C_D genera un conjunto de tamaño $\frac{N}{2}$ para N par y $\frac{N-1}{2}$ para N impar. Así, se codifica el conjunto de lecturas pasando de $9N$ bytes de tamaño a $7\left\lfloor\frac{N}{2}\right\rfloor$ bytes de tamaño.

Capítulo 4

Experimentos

4.1. Pruebas de Conectividad

En la etapa inicial del trabajo realizado, posterior a la definición del hardware a utilizar para el desarrollo del sistema, se realizan dos pruebas para verificar la capacidad de conexión del dispositivo de comunicación inalámbrica LoPy4.

4.1.1. Prueba de conexión a distancia

Como se mencionó en el Capítulo 3, el dispositivo de hardware que se usa para la transmisión inalámbrica en el sistema implementado es la placa de desarrollo LoPy4, la cual presenta dentro de sus características conectividad LoRa, siendo ésta la técnica de modulación usada bajo el protocolo LoRaWAN.

Dado que el contexto del proyecto considera comunicaciones bajo el protocolo LoRaWAN, se plantea la posibilidad de realizar un experimento con el fin de validar la conectividad del dispositivo LoPy4 utilizando modulación LoRa. Un aspecto importante a considerar es la necesidad de una red LoRaWAN previamente establecida, a la cual la LoPy4 pueda tener acceso. Por otra parte, una característica deseable de este experimento considera la realización de comunicaciones a larga distancia, simulando así un escenario cercano a las condiciones en que el proyecto será implementado.

En condiciones normales, estas pruebas de conectividad se hubiesen realizado en las dependencias de la Facultad de Ciencias Físicas y Matemáticas y sus alrededores, pero como el trabajo se desarrolla en un contexto de pandemia, se tuvo que adaptar los métodos de experimentación utilizados.

Así, se da una coincidencia entre la comuna de residencia de un integrante del equipo de trabajo del proyecto y del estudiante, por lo que es posible realizar una prueba de conexión de larga distancia de manera remota. Para esto, el integrante del equipo instala un *gateway* LoRa en las dependencias de su hogar y levanta una red LoRaWAN y una aplicación dentro de ésta para recibir datos. Luego el estudiante implementa una rutina básica de envío de

datos a través de la placa de desarrollo LoPy4, con el objetivo de conectarse a la aplicación montada y enviar datos hacia ésta. Finalmente, se procede a registrar en la aplicación la placa de desarrollo LoPy4 en poder del estudiante, de manera que ésta pueda efectivamente conectarse a la red y comunicarse con la aplicación.

La Figura 4.1 muestra en un mapa la disposición geográfica de los elementos antes mencionados, siendo la distancia de separación de éstos de aproximadamente 1.4 kilómetros. El círculo rojo muestra la zona donde se dispone la LoPy4 en poder del estudiante, mientras que el círculo celeste muestra la zona donde se encuentra el *gateway* LoRa.

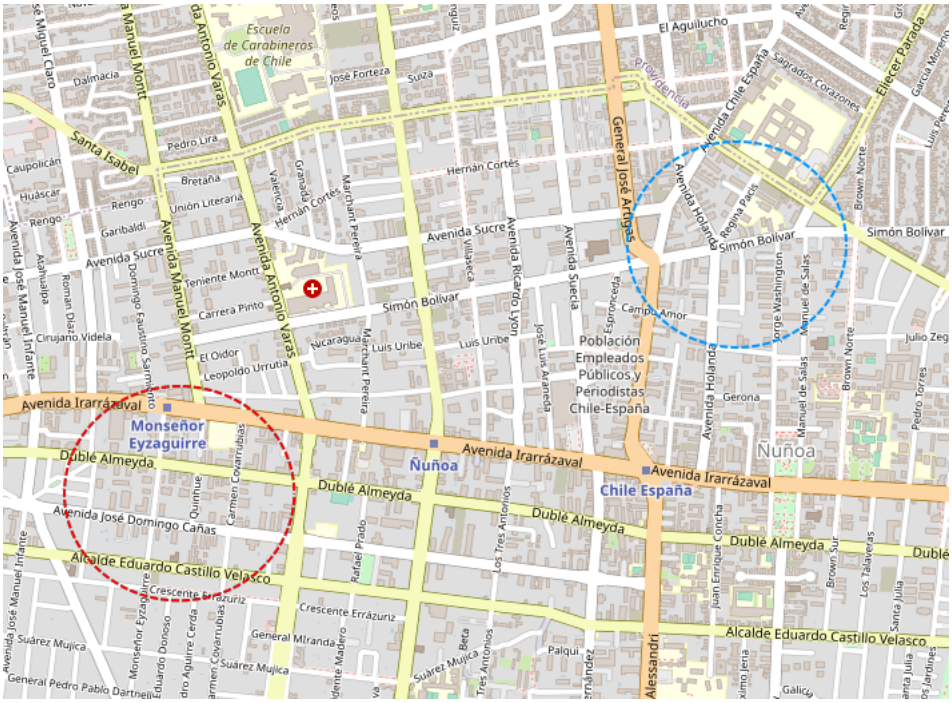

Figura 4.1: Locación geográfica del *gateway* LoRa y de la placa LoPy4 para la prueba de conectividad a distancia.

Como resultado de este experimento, se determina que el dispositivo del estudiante logra conectarse a la red LoRaWAN montada, pero los mensajes enviados a la aplicación jamás llegan. Dentro de las posibles causas de falla destacan la interferencia que podría existir entre los dos puntos de conexión y la incapacidad de la onda de llegar a destino debido a no tener visibilidad del punto de llegada, considerando la existencia de edificios de gran envergadura en el camino, al encontrarse ambos puntos inmersos en una zona urbana de Santiago.

Para descartar posibles fallas de software o hardware, se traslada la placa LoPy4 del estudiante a un punto más cercano al *gateway* instalado, quedando dentro de la zona demarcada en azul en la Figura 4.1. Al transmitir desde esa posición, aproximadamente a 100 metros de la antena, la aplicación montada recibe los datos enviados desde el dispositivo del estudiante, por lo que se descartan los problemas antes presupuestados. De esta experiencia entonces surgió la necesidad de montar una red LoRaWAN accesible por el dispositivo del estudiante.

4.1.2. Prueba de conexión puertas adentro

Con los resultados negativos de la prueba de conexión anterior, se decide montar una red LoRa en el domicilio del estudiante, para asegurar la conexión del dispositivo. Para esto, se instala un *gateway* LoRa, montando una red LoRaWAN local, y se registra una aplicación usando los servicios de The Things Network [1]. Con esto dispuesto, se realiza una prueba de conexión utilizando la misma implementación usada en la prueba de conexión remota, registrando al dispositivo LoPy4 en la nueva red. El resultado obtenido es una correcta conexión entre el dispositivo y la aplicación registrada, junto con un envío de datos exitoso. La Figura 4.2 muestra el registro de la aplicación para la llegada de datos, donde se ve la llegada de datos desde el dispositivo registrado con ID `"bicho"`.

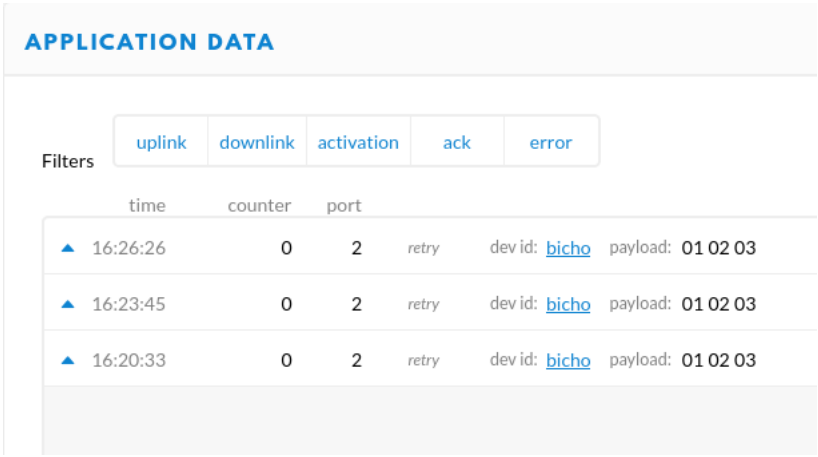

Figura 4.2: Tablero de la aplicación registrada en The Things Network para la prueba de conexión puertas adentro

Cabe mencionar que la rutina de envío de datos implementada consiste simplemente en el envío de un arreglo de bytes equivalentes a los números 1 2 3, como puede verse en el campo *payload* de la anterior figura.

4.2. Evaluación energética

En la etapa final del trabajo, y ya con el sistema implementado y funcionando, se busca conocer el consumo energético asociado al sistema, así como la posible injerencia que las estrategias de codificación implementadas podrían tener sobre este consumo. Para esto, se hace necesario conocer la cantidad de corriente eléctrica que circula por el sistema a la hora de su ejecución.

4.2.1. Instrumento de medición

Para conocer la cantidad de corriente que pasa por el sistema se debe definir un instrumento adecuado que permita realizar esta medición. Además, considerando el contexto de pandemia en el cual se desarrolla el trabajo, se considera que la realización de éstas medi-

[1] https://www.thethingsnetwork.org/

ciones debe hacerse puertas adentro, descartando el acceso a sensores de alta gama como los que podrían usarse en un laboratorio de electrónica.

Una primera idea para realizar estas mediciones es el uso de un amperímetro de alta precisión, el cual sea capaz de leer valores de corriente del orden de los mA, orden de magnitud esperado para el consumo del sistema implementado. Esta idea se descarta debido a que los amperímetros presentes en el mercado no presentan una forma de automatizar la toma de datos, imposibilitando la medición continua de una ejecución del sistema.

Considerando lo anterior, se decide buscar una alternativa que permita realizar lecturas de corriente del sistema y transformarlas a datos que fuesen almacenables para su posterior análisis. De esta manera, se llega al sensor de corriente INA219. Este sensor, que se muestra en la Figura 4.3, es un sensor de corriente de alta precisión, capaz de medir corrientes de hasta 3.2A con una precisión de 0.8mA, o corrientes más pequeñas de hasta 400mA con una precisión de 0.1mA. Además, su versión de mercado se presenta con una placa integrada, la cual permite conectarlo directamente a un microcontrolador y realizar mediciones, comunicándolas a través de una interfaz I2C, lo cual lo convierte en un sensor ideal para las necesidades antes planteadas.

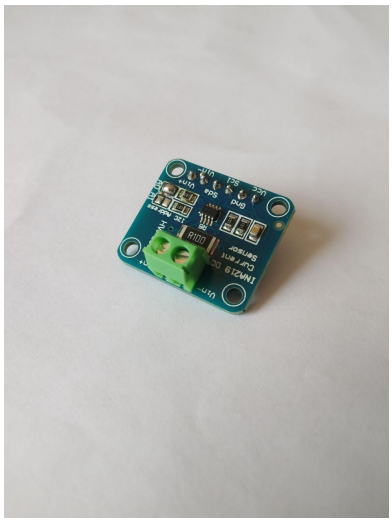

Figura 4.3: Sensor de corriente para Arduino INA219.

Ahora bien, como el sensor por sí solo no es suficiente para realizar las lecturas de corriente, éste se utiliza en conjunto con una placa de desarrollo Arduino Mega. Para realizar la medición de corriente, se implementa una rutina simple de lectura de datos desde el sensor INA219, los cuales se transmiten hacia un computador mediante comunicación secuencial a través de un puerto USB. En la Figura 4.4 se adjunta una imagen del módulo utilizado como sensor de corriente, donde se ve tanto la placa de desarrollo Arduino como el sensor INA219.

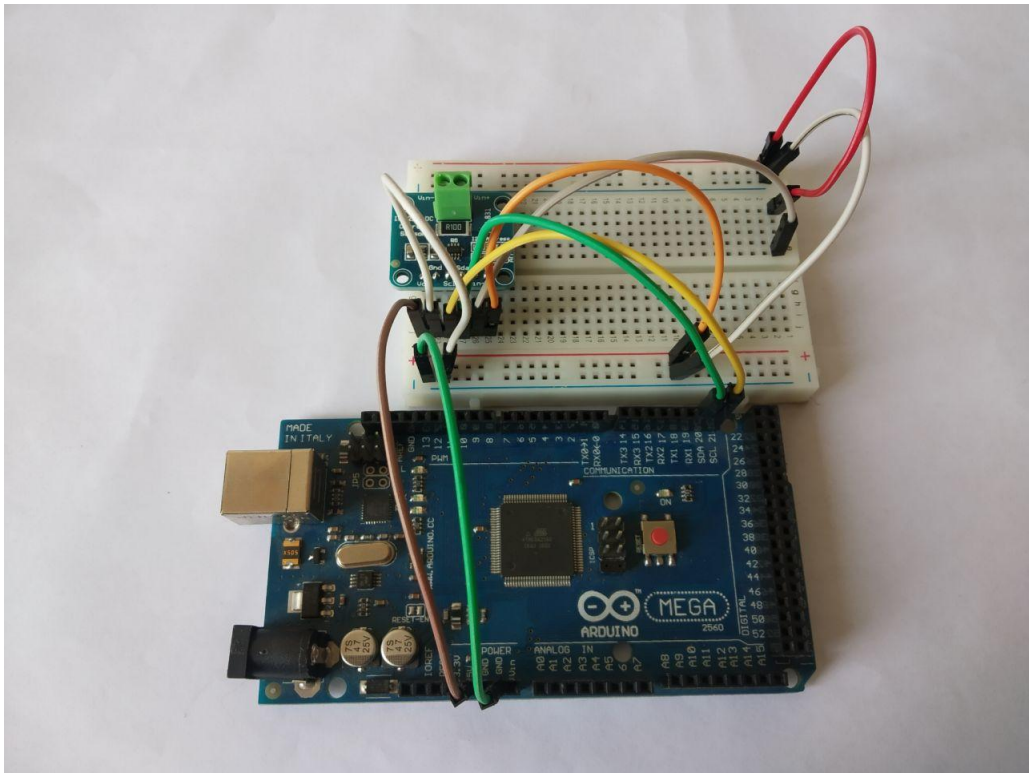

Figura 4.4: Módulo de medición de corriente compuesto por un Arduino Mega y un sensor INA219.

De esta manera, se logra implementar un instrumento de medición de corriente funcional para realizar los experimentos deseados.

4.2.2. Medición de consumo de Bloque Sensor

Teniendo ya un instrumento de medición de corriente, se procede a realizar un primer experimento para estudiar el consumo energético del Bloque Sensor. Para esto se realiza un montaje experimental, mostrado en la Figura 4.5, consistente en el Bloque Sensor, una fuente de poder regulada, el módulo de medición de corriente y un computador para recibir los datos generados por éste.

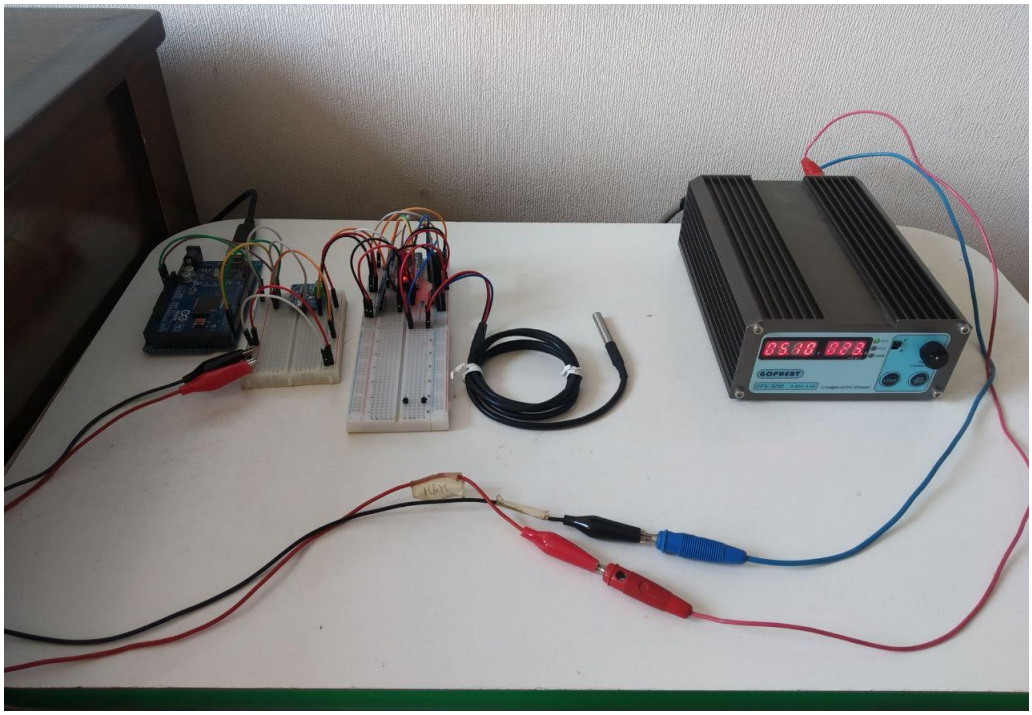

Figura 4.5: Montaje experimental utilizado para la medición de corriente del Bloque Sensor.

El objetivo del experimento es evaluar el consumo energético del Bloque Sensor, y en particular el consumo asociado al microcontrolador Arduino en presencia de los distintos algoritmos de codificación implementados. Esta última consideración se toma en cuenta debido a que el microcontrolador es el encargado de realizar las computaciones asociadas a los distintos tipos de codificación. Por esta razón, si las codificaciones representan esfuerzos computacionales muy distintos entre sí, esa diferencia debería verse reflejada en el consumo energético del microcontrolador. Ahora bien, dado que el orden de los algoritmos de codificación implementados es el mismo para los tres casos, se espera que los consumos energéticos asociados a éstos sean similares.

4.2.2.1. Condiciones generales

La medición de corriente se realiza con el Bloque Sensor configurado para ejecutar lecturas del sensor de temperatura cada 3 segundos y codificarlas cada 168 segundos, dando un total de 56 medidas codificadas. Además, se configura el sistema para que los archivos binarios generados contengan dos bloques de memoria, almacenándose así 112 lecturas en cada archivo.

El tiempo de muestreo total es de aproximadamente 400 segundos, para que de esta manera existan dos eventos de codificación y un evento de escritura de archivo en la tarjeta SD.

4.2.2.2. Medición de consumo Bloque Sensor completo

La primera medición de corriente realizada se hizo sobre el Bloque Sensor completo, utilizando la codificación Base, y midiendo la corriente consumida por todo el circuito desde la fuente de poder. La Figura 4.6 muestra los resultados obtenidos tras este muestreo.

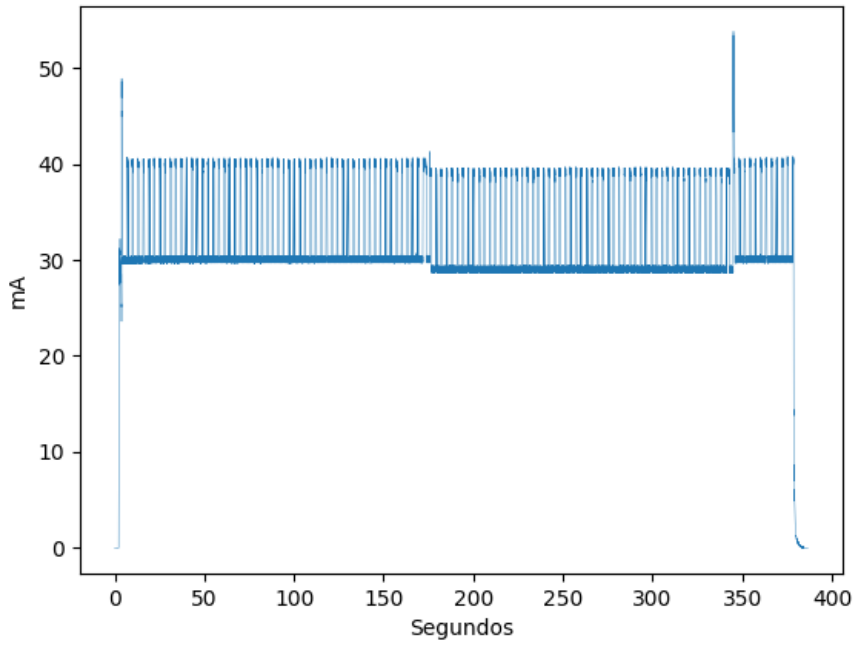

Figura 4.6: Corriente medida para el Bloque Sensor completo, utilizando la codificación Base.

En el gráfico anterior pueden verse dos picos de consumo, uno al principio de la rutina y otro previo a llegar a los 350 segundos. El primer pico corresponde a la configuración inicial del sistema, momento en que el microcontrolador comienza la comunicación con los componentes, genera un nuevo archivo en la memoria microSD, entre otras cosas. El segundo pico corresponde al momento en que se escribe en la memoria microSD un nuevo archivo binario, ya que el archivo de trabajo se llenó.

Con esta medición se tiene la forma de la corriente consumida por el Bloque Sensor para un funcionamiento continuo durante aproximadamente 400 segundos. De esta manera, tenemos una base para comparar el consumo del microcontrolador del sistema para distintas codificaciones.

4.2.2.3. Medición de consumo microcontrolador Arduino

Ya teniendo la curva de consumo para el sistema completo, se procede a medir el consumo eléctrico sólo para el microcontrolador del Bloque Sensor. Para esto, se modifica la forma de medición utilizada previamente, moviendo el sensor de corriente desde la fuente directa hacia el cable de alimentación ligado al microcontrolador Arduino. De esta manera, la lectura de corriente reportará exclusivamente el consumo del microcontrolador del sistema.

La primera medición realizada se da para el caso de uso de la codificación Base. La Figura 4.7 ilustra el consumo energético de este caso. En el gráfico pueden verse los dos picos de corriente identificados en la medición del Bloque Sensor completo, los cuales no están relacionados a los procesos de codificación. Además, aparecen tres picos de corriente

entre los segmentos de tiempo de 0 a 50 segundos, de 150 a 200 segundos y de 300 a 350 segundos. El primer pico de este conjunto parece ser ruido del sensor de corriente, pues también presenta valores que se disparan hacia abajo de la curva. Los dos siguientes picos coinciden con el tiempo en que el sistema ejecuta la codificación de las lecturas realizadas, en los segundos 168 y 336 de la ejecución, por lo que podrían asociarse a la ejecución del algoritmo de codificación.

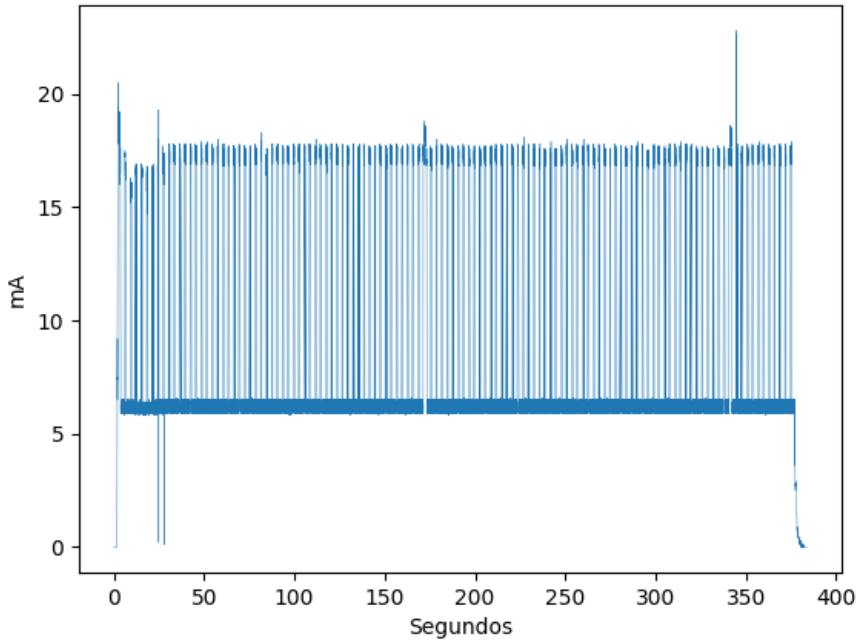

Figura 4.7: Corriente medida para el microcontrolador Arduino, utilizando la codificación Base.

Para validar las observaciones realizadas de la medición de corriente anterior, se compara el gráfico obtenido para el caso de la codificación Base con los gráficos obtenidos para los casos de codificación Repetición y Diferencial. Las Figuras 4.8 y 4.9, muestran el consumo medido para el microcontrolador Arduino utilizando las codificaciones Repetición y Diferencial respectivamente.

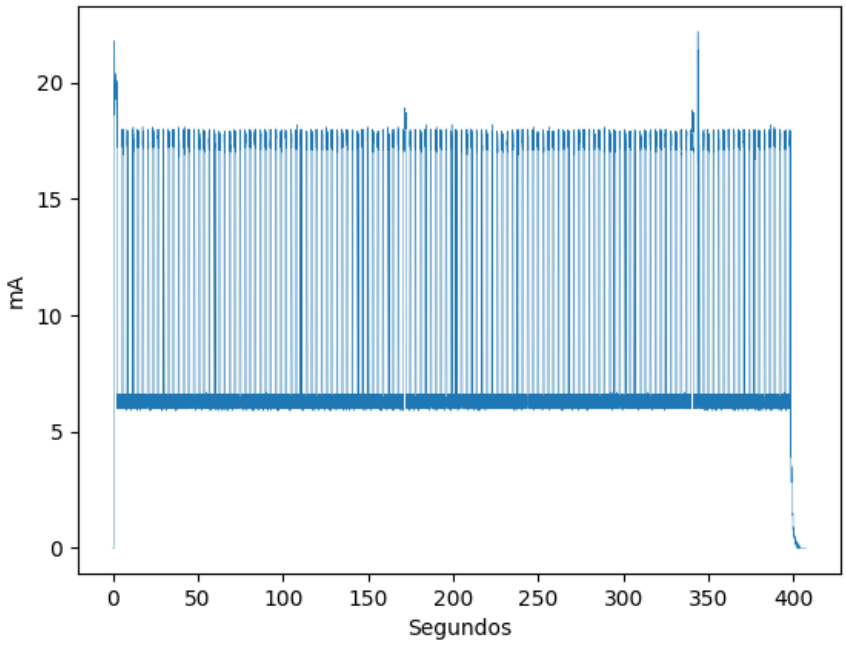

Figura 4.8: Corriente medida para el microcontrolador Arduino, utilizando la codificación Repetición.

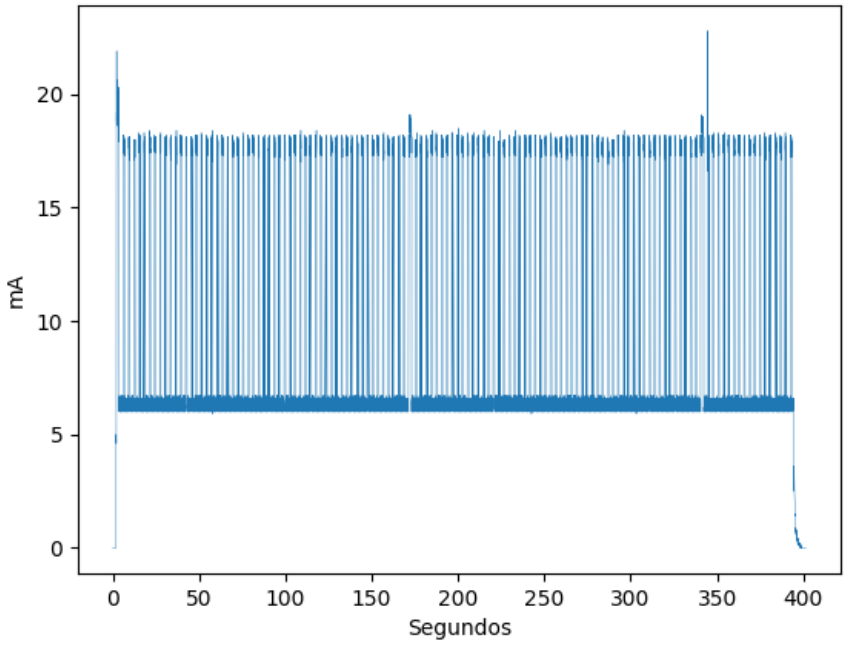

Figura 4.9: Corriente medida para el microcontrolador Arduino, utilizando la codificación Diferencial.

En los gráficos mostrados anteriormente, se puede observar que las figuras obtenidas son altamente parecidas, pudiéndose identificar los mismos picos presentes en la medición realizada para el caso de codificación Base, lo cual significa que los picos identificados en los segmentos de tiempo de 150 a 200 segundos y de 300 a 350 segundos, representan el consumo del sistema a la hora de codificar las lecturas realizadas previamente.

Además, vemos que no existe una diferencia sustancial entre los picos obtenidos para cada una de las tres codificaciones, por lo que se confirma la suposición realizada con respecto al consumo esperado de los algoritmos. De esta manera, podemos concluir que las tres codificaciones generan un consumo energético equivalente, siendo indiferente la elección de la estrategia de codificación a usar en el sistema, a nivel consumo de energía.

4.2.3. Medición de consumo Bloque Transmisor

Posterior a la medición de consumo del Bloque Sensor, se procede a medir el consumo asociado al envío de los datos hacia el *gateway* LoRa instalado localmente. Para esto, y de igual forma que para la medición realizada sobre el Bloque Sensor, se dispone de un montaje experimental consistente de una fuente de poder regulada, el módulo de medición de corriente, el Bloque Sensor conectado de manera secuencial con el Bloque Transmisor y un computador para recibir los datos del experimento. La Figura 4.10 muestra la configuración antes descrita.

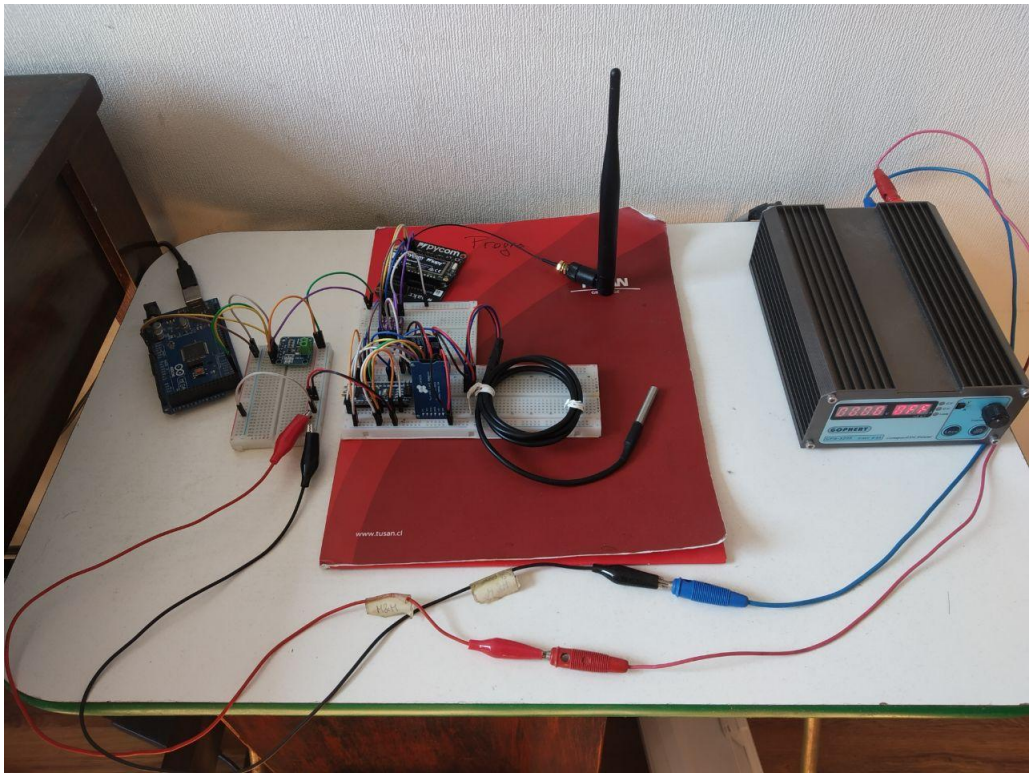

Figura 4.10: Montaje experimental utilizado para la medición de corriente del Bloque Transmisor.

El objetivo de este experimento es evaluar el consumo energético asociado a la transmisión

inalámbrica de los datos desde el Bloque Transmisor hacia el *gateway* LoRa, utilizando las tres diferentes codificaciones implementadas. De esta manera, se busca demostrar que la reducción de tamaño esperada, obtenida gracias a las codificaciones, genera efectivamente un ahorro a nivel del consumo energético del sistema.

4.2.3.1. Condiciones generales

La medición de corriente se realiza utilizando las mismas condiciones aplicadas para la medición realizada sobre el Bloque Sensor. Esto es, configurando el Bloque Sensor para realizar lecturas del sensor cada 3 segundo y codificarlas cada 168 segundos, dando un total de 56 medidas codificadas.

Por su parte, el Bloque Transmisor no presenta algún tipo de configuración específica. De esta manera, después de ejecutar su configuración inicial, el Bloque Transmisor entra en un estado de consumo reducido de energía, hasta ser despertado por el Bloque Sensor. Cabe señalar, que a diferencia del Bloque Sensor, el Bloque Transmisor está compuesto exclusivamente por la placa LoPy4, por lo que no era necesario aislar el componente de un circuito más grande.

El tiempo de muestreo total es de aproximadamente 400 segundos, de igual forma que en el experimento sobre el Bloque Sensor. De esta manera, se generan dos eventos de envío a lo largo de la ejecución, los cuales se registran mediante la aplicación de The Things Network (TTN).

4.2.3.2. Medición de consumo de placa LoPy4

La primera medición, la cual se muestra en la Figura 4.11, se realiza para el uso de la codificación Base. En la figura pueden verse de manera clara los dos eventos de envío, el primero alojado entre los segundos 150 y 250, y el segundo entre los segundos 300 y 400. Además, se puede apreciar un tiempo total de envío del orden de los 50 segundos para cada uno de los eventos. Por otra parte, para cada uno de los picos de corriente puede verse un consumo relativamente similar, alcanzando un máximo por sobre los 175mA.

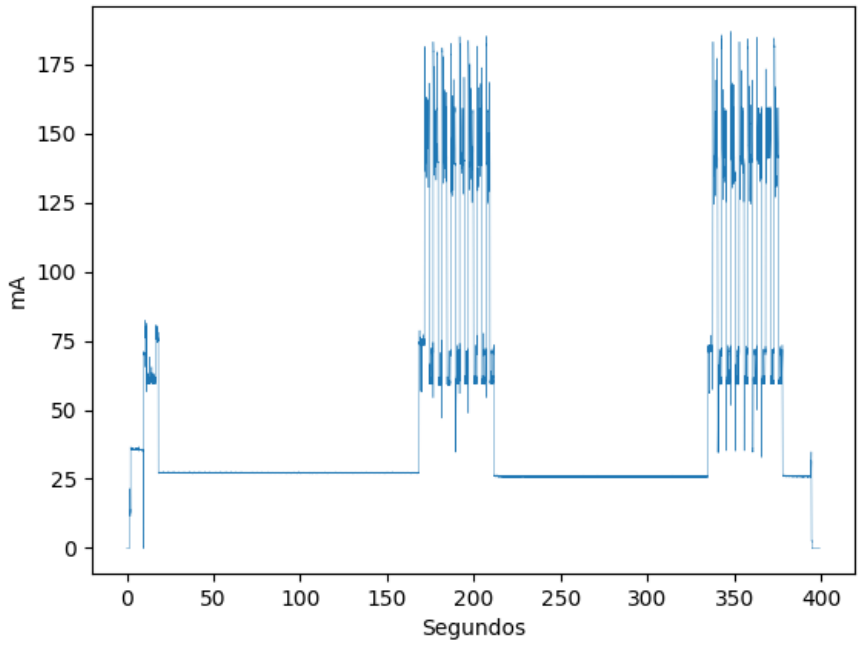

Figura 4.11: Corriente medida para la placa LoPy4, utilizando la codificación Base.

Además, la aplicación de TTN registra un total de 16 paquetes de datos enviados, siendo recibidos en tandas de 8. La Figura 4.12 muestra los paquetes recibidos a través de la aplicación.

time	counter	port			time	counter	port		
18:17:20	15	2	confirmed	payload: F0 60 7A 0D 76 01	18:16:45	8	2	confirmed	payload: F0 60 7A 0C E3 01
18:17:15	14	2	confirmed	payload: F0 60 7A 0D 61 01	18:14:33	7	2	confirmed	payload: F0 60 7A 0C D3 01
18:17:10	13	2	confirmed	payload: F0 60 7A 0D 4C 01	18:14:29	6	2	confirmed	payload: F0 60 7A 0C BE 01
18:17:05	12	2	confirmed	payload: F0 60 7A 0D 37 01	18:14:24	5	2	confirmed	payload: F0 60 7A 0C A9 01
18:17:00	11	2	confirmed	payload: F0 60 7A 0D 22 01	18:14:19	4	2	confirmed	payload: F0 60 7A 0C 94 01
18:16:55	10	2	confirmed	payload: F0 60 7A 0D 0D 01	18:14:14	3	2	confirmed	payload: F0 60 7A 0C 7F 01
18:16:49	9	2	confirmed	payload: F0 60 7A 0C F8 01	18:14:09	2	2	confirmed	payload: F0 60 7A 0C 6A 01
18:16:45	8	2	confirmed	payload: F0 60 7A 0C E3 01	18:14:03	1	2	confirmed	payload: F0 60 7A 0C 55 01
18:14:33	7	2	confirmed	payload: F0 60 7A 0C D3 01	18:13:58	0	2	retry confirmed	payload: F0 60 7A 0C 3E 01

Figura 4.12: Tablero de la aplicación en The Thing Network para el envío de datos usando codificación Base.

Luego de esta medición, se procede a realizar el mismo experimento utilizando la codificación Repetición. La Figura 4.13 muestra la corriente medida para este caso, donde de igual manera pueden identificarse los dos eventos de envío de manera clara, los cuales alcanzan

niveles de consumo máximo similares a los 175mA registrados anteriormente. Además, puede notarse que el tiempo de envío se reduce con respecto a la codificación Base, registrándose aproximadamente en el orden de los 10 segundos para cada evento de envío.

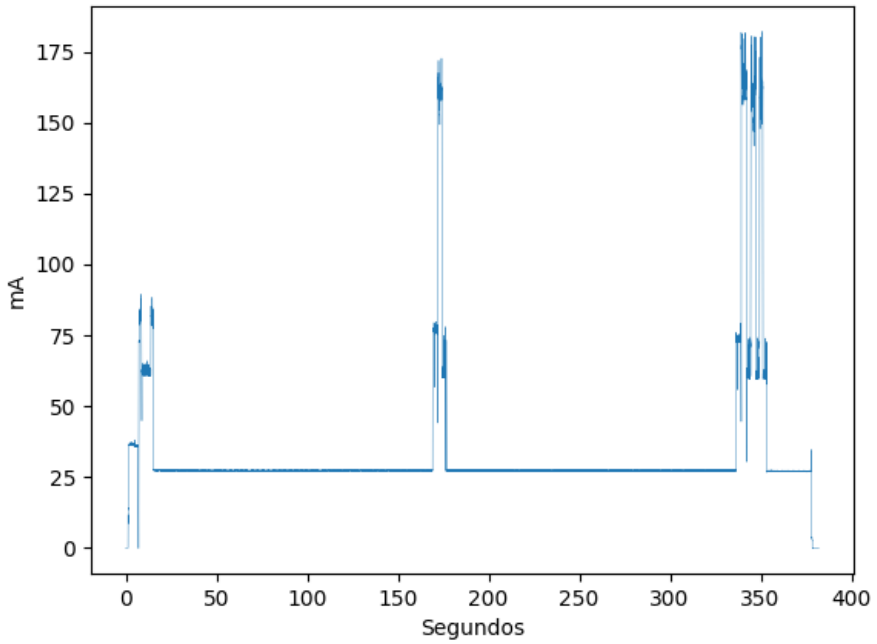

Figura 4.13: Corriente medida para la placa LoPy4, utilizando la codificación Repetición.

Por su parte, la aplicación de TTN registra un total de 4 mensajes, divididos en un primer único mensaje y una segunda tanda de tres mensajes. La Figura 4.14 muestra la consola de la aplicación donde se registran los paquetes recibidos.

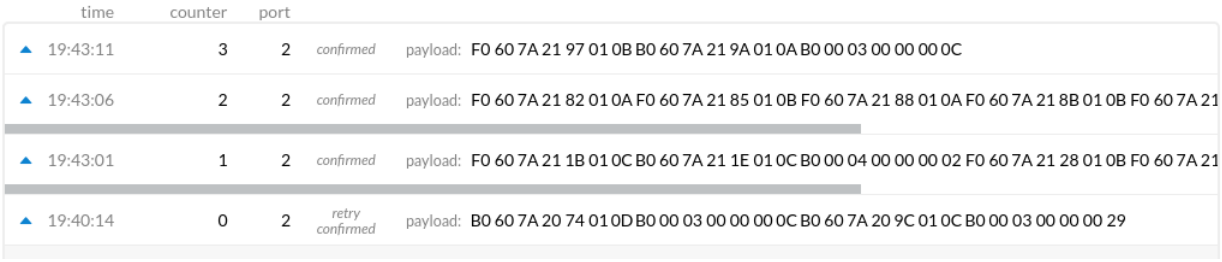

Figura 4.14: Tablero de la aplicación en The Thing Network para el envío de datos usando codificación Repetición.

Por último, se realiza la medición de corriente de la placa LoPy4 utilizando la codificación Diferencial. La Figura 4.15 muestra la corriente medida, diferenciándose nuevamente los dos eventos de envío de manera clara y manteniendo niveles de consumo dentro del mismo rango registrado en los experimentos anteriores. Además, puede verse que ambos eventos de envío

registran una duración del orden de los 30 segundos, reduciendo así el tiempo de envío necesario en comparación con la codificación Base.

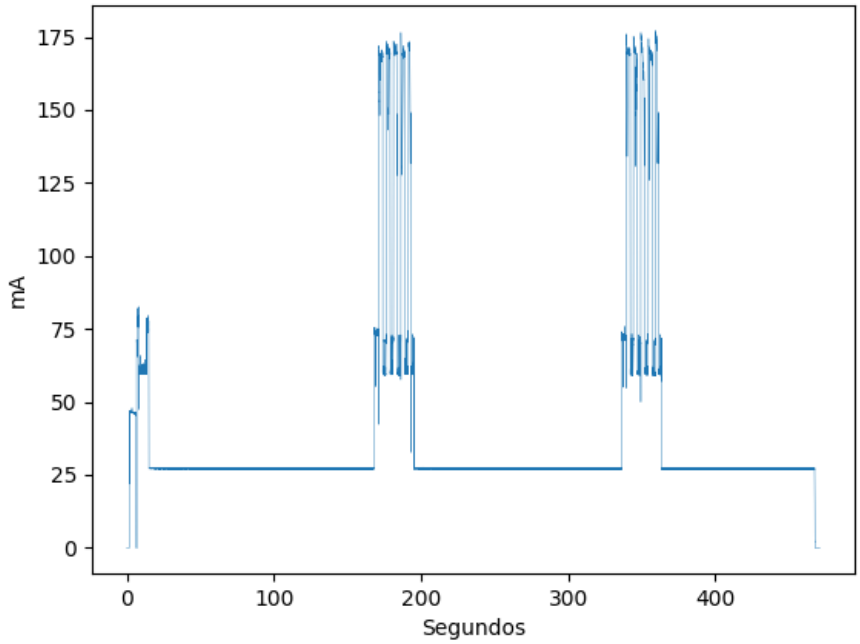

Figura 4.15: Corriente medida para la placa LoPy4, utilizando la codificación Diferencial.

Por otra parte, se registra un total de 10 paquetes de datos recibidos por la aplicación de TTN, enviados en tandas de 5 paquetes cada una. La Figura 4.16 muestra los paquetes recibidos durante el experimento, donde puede apreciarse la separación de las dos tandas gracias al tiempo asociado a cada paquete.

time	counter	port			
18:56:26	9	2	confirmed	payload:	70 60 7A 16 B3 01 0B 70 00 03 80 00 06 80 70 00 09 80 00 0C 80
18:56:22	8	2	confirmed	payload:	70 60 7A 16 8C 01 0B 70 00 03 80 00 06 80 70 00 09 80 00 0C 80 70 00 0F 80 00 12 80 70 00 15 80
18:56:17	7	2	confirmed	payload:	70 60 7A 16 65 01 0B 70 00 03 80 00 06 80 70 00 09 80 00 0C 80 70 00 0F 80 00 12 80 70 00 15 80
18:56:11	6	2	confirmed	payload:	70 60 7A 16 3E 01 0B 70 00 03 80 00 06 80 70 00 09 80 00 0C 80 70 00 0F 80 00 12 80 70 00 15 80
18:56:06	5	2	confirmed	payload:	F0 60 7A 16 1B 01 0C F0 60 7A 16 1E 01 0C 70 60 7A 16 21 01 0C 70 00 03 80 00 06 80 70 00 09 80
18:53:37	4	2	confirmed	payload:	F0 60 7A 16 14 01 0B F0 60 7A 16 17 01 0B
18:53:34	3	2	confirmed	payload:	70 60 7A 15 ED 01 0B 70 00 03 80 00 06 80 70 00 09 80 00 0C 80 70 00 0F 00 00 12 80 70 00 15 80
18:53:29	2	2	confirmed	payload:	70 60 7A 15 C6 01 0B 70 00 03 80 00 06 80 70 00 09 80 00 0C 80 70 00 0F 80 00 12 80 70 00 15 80
18:53:23	1	2	confirmed	payload:	70 60 7A 15 9F 01 0B 70 00 03 80 00 06 80 70 00 09 80 00 0C 80 70 00 0F 80 00 12 80 70 00 15 80
18:53:18	0	2	retry confirmed	payload:	70 60 7A 15 75 01 0A 70 00 03 00 00 08 00 70 00 0B 00 00 0F 00 70 00 12 00 00 15 00 70 00 18 00

Figura 4.16: Tablero de la aplicación en The Thing Network para el envío de datos usando codificación Diferencial.

Con lo anterior, se determina que existe una reducción efectiva del consumo energético necesario para la transmisión de datos utilizando las codificaciones implementadas. Además, se confirma que el proceso de envío de datos de manera inalámbrica es un proceso de alto costo energético, por lo que reducir la cantidad de paquetes a enviar puede generar ahorros de energía significativos para el sistema.

Conclusiones

Del trabajo llevado a cabo y expuesto en esta memoria, es posible concluir que se logró cumplir con los objetivos que se propusieron para su realización.

En primer lugar se cumplió el objetivo principal del trabajo de memoria, lográndose proponer e implementar tres estrategias algorítmicas enfocadas en la reducción del consumo energético de un nodo sensor. Estas técnicas, basadas en la idea de codificación y transformación de la información, cumplían con reducir el tamaño de la información transmitida, ahorrando por tanto la cantidad de paquetes necesarios para enviarla y en consecuencia reduciendo el consumo energético del sistema, al reducirse el tiempo necesario de apertura de la antena de radiofrecuencia usada para la transmisión inalámbrica.

También se logró cumplir con los objetivos específicos planteados para la resolución del problema principal. Así, por ejemplo, se realizó un estudio de la literatura para conocer distintas estrategias utilizadas para la disminución del consumo energético en redes de sensores inalámbricos, lo cual permitió hacerse una idea de los distintos enfoques con que se podía atacar el problema central de la memoria.

Junto a lo anterior, se logró implementar un nodo sensor funcional, tanto a nivel de hardware como de software, sin el cual no hubiese sido posible llevar a cabo el objetivo principal de la presente memoria. Además, este nodo sensor cumplió con todos los requisitos establecidos en el capítulo 3.3.1 de este informe. Por otra parte, la implementación del software del nodo sensor estuvo condicionada por un enfoque de ahorro energético para el sistema, el cual conjugó de buena manera con la metodología incremental de desarrollo utilizada, facilitando esta última la actualización constante de aquellas tareas que se consideraran ineficientes en su implementación.

Luego, se cumplió el objetivo específico de analizar y explicar las estrategias propuestas para la reducción de consumo energético de un nodo sensor, facilitando con ello tanto la comprensión de las estrategias propuestas como la implementación realizada.

Por último, el objetivo específico orientado a la validación de las estrategias propuestas mediante el uso de un medidor de corriente también fue cumplido, consiguiéndose además realizar los experimentos de medición puertas adentro, adaptándose el trabajo a la situación de pandemia en que se tuvo que desarrollar la presente memoria. Este punto es relevante, pues dado que el proyecto en que se enmarca el trabajo realizado es de carácter abierto y de bajo costo, la generación de sensores propios para la validación del buen funcionamiento del sistema se vuelve muy valiosa, generando así un antecedente para las personas que a futuro

deseen implementar el sistema.

A modo general, considerando los resultados obtenidos luego de las mediciones de consumo realizadas, junto con los análisis realizados sobre los algoritmos de codificación, se puede concluir que la codificación Diferencial es la mejor estrategia para reducir el consumo energético en el nodo sensor implementado. Esto se tiene debido a que esta codificación presenta igual rendimiento, para el peor caso, que la codificación Base y un rendimiento superior para el mejor caso, pues logra reducir el tamaño de los datos codificados a la mitad de lo que la codificación Base logra. Por otra parte, si bien la codificación Repetición presenta un rendimiento superior para el mejor caso, este se genera únicamente con la aparición de valores repetidos, estableciéndose una restricción mucho más fuerte que la necesaria para lograr el mejor caso de la codificación Diferencial. Por último, el análisis de consumo de corriente arrojó una reducción efectiva de la energía necesaria para la transmisión de datos usando esta última codificación, por lo que cumple con el objetivo buscado.

Finalmente, y de manera un tanto anecdótica, se puede concluir que el trabajo con hardware, a pesar de tener similitudes con el trabajo de software, presenta dificultades propias que no comparten las mismas lógicas que los problemas de software, pudiendo esto llegar a afectar las planificaciones temporales de una persona no experta. Esta observación, si bien puede ser anecdótica a este trabajo de memoria, es extrapolable a la labor esperada a ser desarrollada por un ingeniero en computación, pues hoy en día es común ver equipos de trabajo multidisciplinarios, lo cual conlleva a que como profesionales se haga necesario salir de nuestra zona de confort para adquirir nuevos conocimientos que no necesariamente atañen a nuestra área, pero que sí pueden ser importantes de considerar a la hora de trabajar.

En miras a un trabajo futuro, y dadas las limitaciones generadas por haber desarrollado el trabajo en medio de una pandemia, ciertos aspectos experimentales podrían mejorarse. Así, se sugiere realizar los experimentos de medición de corriente de manera automatizada, para así poder repetirlos una cantidad arbitraria de veces y descartar posibles resultados circunstanciales de la medición. Además, se sugiere realizar experimentos de transmisión inalámbrica a larga distancia, para observar posibles influencias que tenga la distancia de transmisión sobre el consumo energético del sistema. Por otro lado, considerando que los experimentos se realizaron solo utilizando un sensor de temperatura, se propone ampliar el espectro de datos codificados mediante la adición de nuevos sensores al sistema, para así ver la efectividad de las codificaciones implementadas sobre otro tipo de datos.

En materia del software asociado al sistema, se sugiere evaluar la restricción de tamaño fijo para los paquetes transmitidos entre los Bloques del sistema. Si esta restricción pudiese eliminarse, se podrían evaluar nuevos métodos de compresión y codificación de los datos, los cuales tendrían la libertad de generar paquetes de tamaño libre, eliminando así la necesidad de trabajar sobre palabras de 7 bytes. En su defecto, si la restricción de tamaño se mantuviese, se podría evaluar la búsqueda de nuevas formas de codificación, las cuales pueden ser adaptativas dependiendo del tipo de dato codificado. Una opción para esto podría ser la de generar una única codificación que mezcle las características de las tres codificaciones aquí implementadas, considerando que éstas no son excluyentes entre sí.

Bibliografía

[1] LoRa Alliance Technical Marketing Workgroup. A technical overview of LoRa and LoRaWAN. `https://lora-alliance.org/resource_hub/what-is-lorawan/`, 2015. [Online, última vez accedido el 7 de Abril de 2021].

[2] P. Koopman and T. Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks. In *International Conference on Dependable Systems and Networks, 2004*, pages 145–154, 2004.

[3] J. de Carvalho Silva, J. J. P. C. Rodrigues, A. M. Alberti, P. Solic, and A. L. L. Aquino. Lorawan — a low power wan protocol for internet of things: A review and opportunities. In *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*, pages 1–6, 2017.

[4] Cisco. Cisco Annual Internet Report (2018–2023). `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`, 2020. [Online, última vez accedido el 19 de Mayo de 2021].

[5] J. Huang, Y. Meng, X. Gong, Y. Liu, and Q. Duan. A novel deployment scheme for green internet of things. *IEEE Internet of Things Journal*, 1(2):196–205, 2014.

[6] F. K. Shaikh, S. Zeadally, and E. Exposito. Enabling technologies for green internet of things. *IEEE Systems Journal*, 11(2):983–994, 2017.

[7] S. Farrell. Low-Power Wide Area Network (LPWAN) Overview. RFC 8376, RFC Editor, May 2018.

[8] LoRa Alliance. What is LoRaWAN Specification. `https://lora-alliance.org/resource_hub/what-is-lorawan/`, 2015. [Online, última vez accedido el 7 de Abril de 2021].

[9] S. D. Infanteena and E. A. M. Anita. Survey on compressive data collection techniques for wireless sensor networks. In *2017 International Conference on Information Communication and Embedded Systems (ICICES)*, pages 1–4, 2017.

[10] Y. Wu, Y. He, and L. Shi. Energy-saving measurement in lorawan-based wireless sensor networks by using compressed sensing. *IEEE Access*, 8:49477–49486, 2020.

[11] J. Haupt, W. U. Bajwa, M. Rabbat, and R. Nowak. Compressed sensing for networked data. *IEEE Signal Processing Magazine*, 25(2):92–101, 2008.

[12] X. Liu, Y. Zhu, L. Kong, C. Liu, Y. Gu, A. V. Vasilakos, and M. Wu. Cdc: Compressive data collection for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 26(8):2188–2197, 2015.

[13] T. Arici, B. Gedik, Y. Altunbasak, and L. Liu. Pinco: a pipelined in-network compression scheme for data collection in wireless sensor networks. In *Proceedings. 12th International Conference on Computer Communications and Networks (IEEE Cat. No.03EX712)*, pages 539–544, 2003.

[14] H. Luo, H. Tao, H. Ma, and S. K. Das. Data fusion with desired reliability in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 22(3):501–513, 2011.

[15] M. Yuanbin, Q. Yubing, L. Jizhong, and L. Yanxia. A data compression algorithm based on adaptive huffman code for wireless sensor networks. In *2011 Fourth International Conference on Intelligent Computation Technology and Automation*, volume 1, pages 3–6, 2011.

[16] S. Renugadevi and P. S. Nithya Darisini. Huffman and lempel-ziv based data compression algorithms for wireless sensor networks. In *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*, pages 461–463, 2013.

[17] Z. Chen, S. Guiling, L. Weixiang, G. Yi, and L. Lequn. Research on data compression algorithm based on prediction coding for wireless sensor network nodes. In *2009 International Forum on Information Technology and Applications*, volume 1, pages 283–286, 2009.

[18] J. F. S. Aquino, E. F. Nakamura, A. A. F. Loureiro, and M. Endler. A differential coding algorithm for wireless sensor networks. In *2008 IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, 2008.

[19] A. Peruzzo and L. Vangelista. A power efficient adaptive data rate algorithm for lorawan networks. In *2018 21st International Symposium on Wireless Personal Multimedia Communications (WPMC)*, pages 90–94, 2018.

[20] D. Magrin, M. Capuzzo, and A. Zanella. A thorough study of lorawan performance under different parameter settings. *IEEE Internet of Things Journal*, 7(1):116–127, 2020.

[21] J. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247–252, 1982.

[22] T. C. Maxino and P. J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Transactions on Dependable and Secure Computing*, 6(1):59–72, 2009.