Universidad de Valladolid

Máster universitario

Ingeniería Informática

Trabajo Fin de Máster

Desarrollo de un dispositivo IoT para la trazabilidad de residuos

Realizado por **Daniel López Martínez**

✠✠✠

Universidad de Valladolid

15 de julio de 2025

Tutor: Diego Rafael Llanos Ferraris

Agradecimientos

Quiero agradecer a mi tutor, Diego Rafael Llanos Ferraris, por su inmensa paciencia y dedicación a lo largo del desarrollo de este trabajo.

Y también, a mi familia y a mi pareja, Miriam, por el apoyo constante que me han brindado a lo largo de todo el máster, y a mis amigos por los ánimos y ayuda que me han proporcionado en todo momento.

Resumen

La creciente generación de residuos, especialmente en entornos urbanos y sectores como la agricultura o la medicina, plantea retos importantes en cuanto a su gestión y trazabilidad. En este trabajo se presenta el diseño e implementación de un sistema basado en tecnologías *IoT* (*Internet of Things*) para mejorar la trazabilidad de residuos urbanos, permitiendo monitorizar en tiempo real parámetros como la posición geográfica, la temperatura y la humedad de los contenedores durante su transporte. La solución propuesta consiste en un dispositivo "suicida" con sensores de bajo coste, fácilmente replicable, que envía los datos recopilados a un servidor en un formato de mensaje fiable y fácil de analizar. El sistema se ha probado en un entorno controlado, demostrando ser una opción fiable, económica y funcional, mostrando también sus posibilidad de evolución hacia soluciones aún más compactas y baratas. Todo el proceso de desarrollo ha sido exhaustivamente documentado, permitiendo su reproducción y mejora por parte de cualquier persona con los conocimientos técnicos necesarios.

Descriptores

Generación de residuos, IoT, Trazabilidad de residuos, Dispositivo "suicida", Sensores.

Abstract

The increasing generation of waste, especially in urban environments and sectors such as agriculture or medicine, poses significant challenges in terms of waste management and traceability. This work presents the design and implementation of a system based on *IoT* (*Internet of Things*) technologies to improve the traceability of urban waste, allowing real-time monitoring of parameters such as geographical position, temperature and humidity of the containers during transport. The proposed solution consists of a "suicidal" device with low-cost sensors, easy replicability, which sends the collected data to a server in a reliable and easy-to-analyze message format. The system has been tested in a controlled environment, proving to be a reliable, economical and functional option, showing also its possibility of evolution towards even more compact and cheaper solutions. The entire development process has been thoroughly documented, allowing it to be reproduced and improved by anyone with the necessary technical knowledge.

Keywords

Generation of waste, IoT, Waste traceability, "Suicidal" device, Sensors.

Índice general

Índice de figuras

Índice de tablas

1: Introducción

El aumento de la generación de residuos se ha convertido en un reto importante en los países en desarrollo debido a un crecimiento demográfico y una urbanización sin precedentes [1]. De hecho, se calcula que la generación mundial de residuos sólidos urbanos alcanzará los 3.400 millones de toneladas en 2050, más del doble del crecimiento demográfico en el mismo periodo, tal como se afirma en [2]. En lo que se refiere, por ejemplo, a la agricultura, se intenta minimizar la generación de residuos, optimizar las técnicas de eliminación y promover el reciclaje y la reutilización, véase [3].

Por otro lado, en lo que se refiere a los residuos médicos, resultó de vital importancia durante la pandemia de COVID-19. En [4], se presenta una propuesta de gestión de este tipo de residuos mediante un sistema inteligente de gestión de restos médicos con monitorización en tiempo real de los datos de residuos médicos y una aplicación para el usuario final.

En España la gestión de residuos y el reciclaje de los mismos requiere de la coordinación de diversos grupos y empresas, que no siempre cumplen con los acuerdos establecidos. Sin embargo, es difícil llevar un control exhaustivo de los distintos sistemas de transporte de residuos. A pesar de que existen diversos contenedores para distintos tipos de residuos, estos pueden no acabar en el destino adecuado, ya sea por descuidos o por una falta de control sobre los medios de transporte de los mismos.

La falta de trazabilidad de los residuos puede llevar a vertidos ilegales, ineficiencias logísticas y pérdida de materiales potencialmente reciclables. En este último caso, es bien sabido que el gobierno español invierte una buena cantidad de recursos para el tratamiento de residuos en plantas de reciclaje. Sin embargo, esta inversión tiene poca relevancia si los residuos reciclables acaban en vertederos en vez de en las plantas de reciclaje correspondientes, lo que actúa como incentivo para el desarrollo de sistemas de trazabilidad más efectivos.

En este trabajo se describe el desarrollo de una solución al problema de la trazabilidad de residuos de distinto tipo, permitiendo controlar el camino y destino de las basuras depositadas en los distintos contenedores. El dispositivo propuesto tiene como misión obtener datos sobre su posición geográfica, temperatura y humedad en diversos momentos,

para enviar dicha información a un servidor seguro donde poder analizarlos. Al introducir este dispositivo en un contenedor de basura se podrá observar cómo y a dónde se transportan los residuos que se encontrasen en dicho contenedor.

Por lo tanto, podríamos resumir los objetivos de este proyecto en los siguientes puntos clave:

- Diseñar un dispositivo capaz de registrar y enviar datos de posición geográfica, temperatura y humedad en tiempo real.
- Iterar el diseño del dispositivo hacia implementaciones más sencillas y menores en tamaño para mejorar su manejabilidad.
- Implementar y construir el dispositivo para su puesta en marcha.

La implementación de un sistema de estas características podría mejorar la eficiencia en la gestión de residuos urbanos, facilitar el control por parte de las autoridades, y contribuir a una mayor transparencia en los procesos de reciclaje.

2: Análisis de requisitos

Para el correcto desarrollo del proyecto se ha realizado un análisis de requisitos de usuario que nos permita entender el alcance del proyecto y las características indispensables del dispositivo a desarrollar. Estos requisitos junto con la definición de otros detalles clave que se dan en este capítulo, servirán de base para el diseño, desarrollo y validación del dispositivo y su sistema asociado.

2.1. Requisitos funcionales

En la Tabla 2.1 se presentan los requisitos funcionales que describen qué debe hacer el sistema para cumplir con los objetivos del proyecto.

Tabla 2.1: Requisitos Funcionales

ID	Requisito	Descripción
RF01	Localización GPS precisa	El dispositivo debe ser capaz de registrar su posición geográfica mediante un sistema de posicionamiento global (GPS) con una precisión mínima de 20 metros.
RF02	Medición ambiental	El sistema debe medir y registrar la temperatura y humedad relativa del entorno.
RF03	Almacenamiento en desconexión	El dispositivo debe almacenar temporalmente los datos en caso de pérdida de conectividad, y enviarlos al recuperar la conexión.
RF04	Envío periódico de datos	Los datos registrados deben enviarse periódicamente a un servidor remoto mediante un protocolo de comunicación fiable (como MQTT).
RF05	Frecuencia configurable	La frecuencia de envío de datos debe poder ser configurada manualmente.
RF06	Seguridad en la transmisión	El sistema debe garantizar la seguridad de los datos transmitidos mediante autenticación.

ID	Requisito	Descripción
RF07	Identificador único	El sistema debe registrar un identificador único que permita asociar los datos a un dispositivo específico.
RF08	Exportación de datos	El sistema debe permitir exportar los datos registrados en formatos estándar (como CSV o JSON) para su análisis.
RF09	Funcionamiento autónomo	El dispositivo debe operar de manera completamente autónoma, sin requerir interacción manual durante su funcionamiento habitual.
RF10	Registro de movimiento	El dispositivo debe ser capaz de determinar si se encuentra en movimiento.

2.2. Requisitos no funcionales

En la Tabla 2.2 se presentan los requisitos no funcionales que describen cómo debe comportase el sistema para cumplir con los objetivos del proyecto.

Tabla 2.2: Requisitos No Funcionales

ID	Nombre Corto	Descripción
RNF01	Autonomía energética mínima	El dispositivo debe contar con una autonomía energética mínima de 24 horas en funcionamiento normal sin recarga.
RNF02	Resistencia ambiental	El dispositivo debe estar diseñado para resistir golpes, vibraciones y condiciones ambientales adversas (humedad, polvo, variaciones térmicas).
RNF03	Diseño compacto y compatible	El diseño físico del dispositivo debe permitir su inserción en el interior de distintos tipos de contenedores sin modificar su funcionalidad ni el acceso a su contenido.
RNF04	Consumo eficiente	El dispositivo debe estar diseñado con componentes de bajo consumo para maximizar la eficiencia energética.

2.3. Funcionamiento general del dispositivo

El comportamiento fundamental del dispositivo se describe en el Algoritmo 1, en donde se puede observar como la idea principal consiste en recopilar distinta información durante varias iteraciones para enviarla tras un número específico de datos recopilados. Como parte del intento de envío de datos se plantea la reconexión de servicios necesarios como la conexión a internet y con el servidor de destino, ya que esta puede no mantenerse activa entre envíos. Este comportamiento se repite mientras el dispositivo se mantenga alimentado por una corriente, teniendo en cuenta que antes se habrán realizado las inicializaciones necesarias. Una descripción más completa del funcionamiento del dispositivo y de los mecanismos de comunicación se detalla en el Capítulo 5.

Algorithm 1 Algoritmo General de Operación del Dispositivo IoT

```
Inicio
inicializarDispositivo()
while dispositivoEncendido do
    recolectarDatos()
    procesarDatos()
    if datosRecopilados > N then
        if conexiónEstablecida then
            enviarDatosAlServidor()
        else
            conectarARed()
            enviarDatosAlServidor()
        end if
    else
        almacenarDatosLocalmente()
        datosRecopilados++
    end if
    esperarXTiempo()
end while
Fin
```

2.4. Estructura general del servidor

Si bien el diseño e implementación de un servidor no forma parte de los objetivos de este proyecto es importante definir la estructura del mismo para contextualizar la solución completa, y que se describe con más detalle en la contribución para las Jornadas Sarteco 2025 [5].

La arquitectura propuesta, ilustrada en la Figura 2.1, integra de manera estratégica IoT, blockchain y sistemas de almacenamiento escalables. El objetivo principal es asegurar la trazabilidad y la seguridad de los datos que los dispositivos IoT transmiten mediante el protocolo MQTT. Esta arquitectura se estructura en varios módulos interconectados, cada uno con una función específica en la recolección, procesamiento, almacenamiento y validación de los datos dentro de un entorno distribuido y seguro. Si bien este documento se centra en el desarrollo del dispositivo IoT, se asume que este utiliza comunicaciones MQTT para el envío de información. Los componentes clave del sistema son:

- **API Gateway**: Actúa como el punto de entrada centralizado para todas las interacciones. Se encarga de gestionar la autenticación, la autorización y el enrutamiento del tráfico hacia los diferentes servicios de la arquitectura.
- **Collector (MQTT)**: Compuesto por un bróker MQTT y un proceso worker, este módulo es capaz de escalar según la demanda. Su misión fundamental es recibir

los datos procedentes de los dispositivos IoT, validarlos y transformarlos antes de enviarlos a las siguientes fases de procesamiento.

- **Persistence Controller**: Este módulo es el responsable de la gestión del almacenamiento de datos en el Data Lake, así como de su posterior procesamiento y recuperación. Para ello, se emplea Delta Lake, que ofrece una gran compatibilidad con diversas soluciones de almacenamiento, tanto locales como en la nube.
- **Blockchain Controller**: Su función es interactuar directamente con los contratos inteligentes desplegados en la red blockchain. Este módulo garantiza la integridad de los datos almacenados y facilita la obtención del identificador de la transacción (hash) para su registro junto con los datos en el Data Lake.

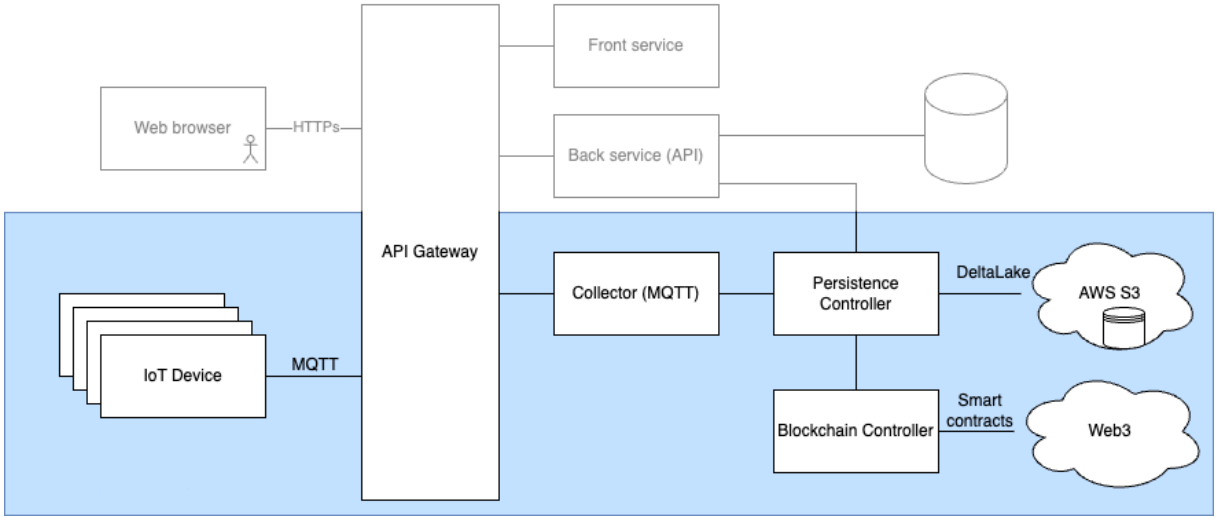

Figura 2.1: Propuesta de arquitectura del servidor

En la propuesta de arquitectura se pueden ver otros módulos que están más relacionados con la visualización de los datos por parte del usuario. Sin embargo, dado que esta propuesta de arquitectura se encuentra fuera del alcance de este proyecto no se detallará más ninguna de sus partes.

2.5. Resumen

En este capítulo se ha presentado el resultado del análisis de requisitos junto con un resumen general del funcionamiento del dispositivo y una vista de la propuesta de arquitectura del servidor, que está fuera de los objetivos del servidor. Todo esto permite contextualizar y comprender las características y estructura de la solución propuesta en este proyecto, y asentar una base necesaria para el diseño y posterior implementación del

dispositivo que se presentan en capítulos posteriores. Sin embargo, como paso anterior al desarrollo del dispositivo se debe plantear una planificación de las tareas a realizar, los posibles riesgos a enfrentar y el presupuesto necesario para realizar el proyecto completo. Todo este análisis y planificación se detalla en profundidad en el Capítulo 3.

3: Modelo de análisis

En este capítulo se detalla el modelo de análisis que ha sustentado el desarrollo de este proyecto. En este se desglosa en primer lugar, la planificación inicial, describiendo las fases y tareas fundamentales que guiaron el proyecto. Posteriormente, se presenta un análisis de riesgos, donde se identifican posibles situaciones y problemas y las estrategias de mitigación implementadas para asegurar la viabilidad y robustez del sistema. Posteriormente se examinará la desviación de la planificación inicial, explicando los ajustes realizados y los motivos de los mismos. Finalmente, se expondrá el presupuesto y los costes asociados al proyecto, ofreciendo una visión clara de los recursos económicos gestionados durante su ejecución.

3.1. Planificación del proyecto

Este proyecto se organizó para comenzar su desarrollo a finales de 2024, en paralelo a las asignaturas del Máster en Ingeniería Informática, con la intención de documentar parte del mismo en una contribución para las Jornadas Sarteco de 2025. Por lo tanto, parte de la organización implicó tener una parte del trabajo terminado para las fechas de envío de contribuciones, que originalmente estaban marcadas para finales de marzo de 2025. Con este objetivo el proyecto se dividió en iteraciones de desarrollo como se explica en el Capítulo 4, siendo aconsejable que las ultimas tareas de la segunda iteración coincidan con la contribución para Sarteco.

Cada iteración ha implicado un desglose del trabajo en tareas características de un desarrollo IoT, las cuales, en gran medida, se corresponden con los capítulos de esta memoria. Esto facilita la consulta del esfuerzo dedicado a cada una de ellas. Concretamente, las tareas son:

- **Diseño del dispositivo**: Este es el primer paso a realizar para cada iteración, ya que consiste en elegir los componentes hardware a utilizar para cumplir con las funcionalidades de dicha iteración y diseñar el circuito eléctrico del sistema. Esta fase requiere revisar los distintos componentes que existen actualmente en el mercado y

consultar los distintos requisitos de alimentación de cada uno de ellos para poder diseñar un circuito eléctrico funcional.

- **Implementación del sistema**: Esta tarea está más relacionado con el software que se ejecutará en el dispositivo diseñado y que deberá cumplir con los requisitos de cada iteración. Esta fase también requiere de analizar y consultar la documentación de distintas librerías a utilizar para saber su compatibilidad con los componentes elegidos, sus limitaciones y ejemplos que pueden ayudar al desarrollo del propio código.
- **Pruebas y mediciones del sistema**: Como parte del desarrollo del sistema se deberá verificar que todas las funcionalidades implementadas son correctas y dan los resultados esperados. Además, como parte de un desarrollo IoT, se deben comprobar los distintos niveles de consumo tanto eléctrico como de ancho de banda.
- **Documentación del desarrollo**: En paralelo a las anteriores tareas se realiza una documentación de todos los procesos y actividades realizadas como parte de la memoria de trabajo fin de máster. Además, parte de la documentación se hará en la contribución para las Jornadas Sarteco, por lo que se tendrá en cuenta también su desarrollo en la planificación.

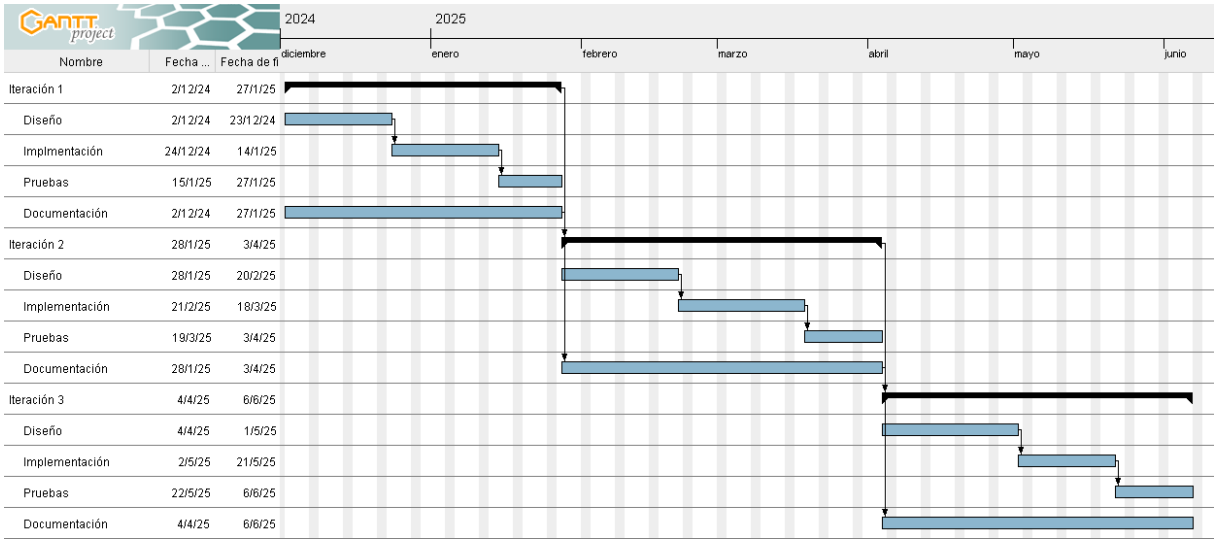

Figura 3.2: Diagrama de Gantt de la planificación inicial

Como ya se ha comentado este proyecto se empezó en diciembre de 2024, con la intención de poder tener resultados para una contribución en las Jornadas Sarteco. Sin embargo, la carga de horas semanales aplicadas al proyecto no se planificaron igual para esos primeros meses de desarrollo, ya que además de este trabajo se realizaban a tiempo completo asignaturas y un contrato de empleo de 37 horas y medias semanales. Por lo tanto, los meses de diciembre, mitad de enero, mitad de febrero, marzo y primera mitad de abril se planificó una dedicación semanal de 4 horas. Mientras que el resto del tiempo,

es decir, segunda mitad de enero, primera mitad de febrero, segunda mitad de abril, mayo y primera mitad de junio, se planificó una dedicación semanal de 8 horas, ya que en esas etapas no habrá carga de asignaturas. Teniendo en cuenta la dedicación en horas anteriormente mencionada, y que la fecha de deposito de trabajos en convocatoria ordinaria empieza el 23 de junio, se plantea en total una dedicación de aproximadamente 170 horas de trabajo, tratando de ajustar las horas a los seis créditos de dedicación asignados por la Escuela de Ingeniería Informática. En la Figura 3.2 se puede observar una imagen de la planificación inicial realizada mediante GanttProject, teniendo en cuenta la dedicación de horas anteriormente explicada.

3.2. Análisis de riesgos

En todo proyecto de desarrollo hay que tener en cuenta que pueden surgir situaciones inesperadas o fallos no contemplados, por ello es importante realizar un análisis de los posibles riesgos que se pueden producir. Si bien gran parte de los problemas que pueden darse tienen relación con la elaboración de este proyecto en paralelo a asignaturas y a un empleo a tiempo completo, hay otros riesgos más relacionados con los conocimientos que se poseen de este campo y los problemas más típicos de desarrollo de sistema IoT. A continuación desde la Tabla 3.3 hasta la Tabla 3.7 se detallan los riesgos analizados y contemplados durante la planificación inicial, junto con un posible plan de contingencia:

Identificador	R01 - Mala planificación del proyecto
Descripción	La duración real de las tareas no se ajusta correctamente al tiempo originalmente planificado para su realización
Impacto	Alto
Probabilidad	Media
Plan de contingencia	Replanificación del proyecto

Tabla 3.3: Riesgo de mala planificación

Identificador	R02 - Retraso en el envío de componentes
Descripción	Un componente llega más tarde de lo esperado según las tareas relacionadas con este
Impacto	Medio
Probabilidad	Alta
Plan de contingencia	Replanificación del proyecto y avanzar con otras tareas

Tabla 3.4: Riesgo de retraso en envío de componentes

Identificador	R03 - Componente defectuoso o roto
Descripción	Un componente ha llegado defectuoso o se ha roto durante su uso
Impacto	Alto
Probabilidad	Media
Plan de contingencia	Pedir más de un componente del mismo tipo a la vez

Tabla 3.5: Riesgo de componente defectuoso o roto

Identificador	R04 - Falta de conocimientos de electrónica
Descripción	La falta de conocimientos básicos de electrónica está retrasando el diseño y montaje de los circuitos
Impacto	Medio
Probabilidad	Alta
Plan de contingencia	Pedir ayuda a personas con conocimiento en electrónica y dedicar tiempo del proyecto a aprender lo necesario

Tabla 3.6: Riesgo de falta de conocimientos de electrónica básica

Identificador	R05 - Fallos de cobertura y/o detección satélite
Descripción	Las antenas GPS y/o de red móvil fallan mucho o pierden la conexión con frecuencia
Impacto	Medio
Probabilidad	Baja
Plan de contingencia	Comprar nuevas antenas si se consideran rotas o implementar un sistema adecuado de reintentos de conexión

Tabla 3.7: Riesgo de cobertura y/o detección satélite

3.3. Desviación de la planificación inicial

Sobre la planificación inicial planteada se han tenido que realizar cambios y ajustes debido a la sucesión de algunos riesgos esperados e inesperados. Algunos de los riesgos esperados que han ocurrido se han podido contener correctamente gracias a los planes de contingencia planteados. Sin embargo, otros riesgos como el riesgo **R04** no se ha podido contener tan fácilmente, debido a que el problema se centraba en la mala soldadura de componentes a lo largo de varias iteraciones, y dado que no se encontró una solución rápida a este problema, se tuvo que hacer una replanificación más drástica.

Otros riesgos no contemplados también han causado problemas, específicamente el análisis incorrecto de la documentación de una librería causó que una primera implementación funcionase de manera inestable y no se encontrase el problema hasta pasado un tiempo del planificado. También como consecuencia de estos problemas no se pudo proporcionar

mediciones de consumo eléctrico para la contribución en las Jornadas Sarteco, aunque sí se pudo dar resultados aceptables para esta contribución.

La paralelización de este proyecto con el resto de las actividades lectivas del máster y el contrato laboral también causó algún retraso, aunque lo esperado dentro del riesgo **R01**. Por último, el recorrido de la tercera iteración se vió afectado por múltiples situaciones fuera de lo esperado que se describen en más detalle a lo largo de la memoria. Todo esto forzó la modificación de la planificación inicial, hasta el punto de trasladar la fecha de final del proyecto al periodo extraordinario de deposito de trabajos fin de máster de julio. Como consecuencia en la Figura 3.3 se puede observar la modificación del diagrama de Gantt que representa la planificación final realizada. En esta planificación final se ve cómo la mayoría de fases se han alargado en el tiempo, sobretodo las correspondientes a la tercera iteración, y la fase de documentación final se alargó más para poder documentar y revisar el trabajo realizado correctamente.

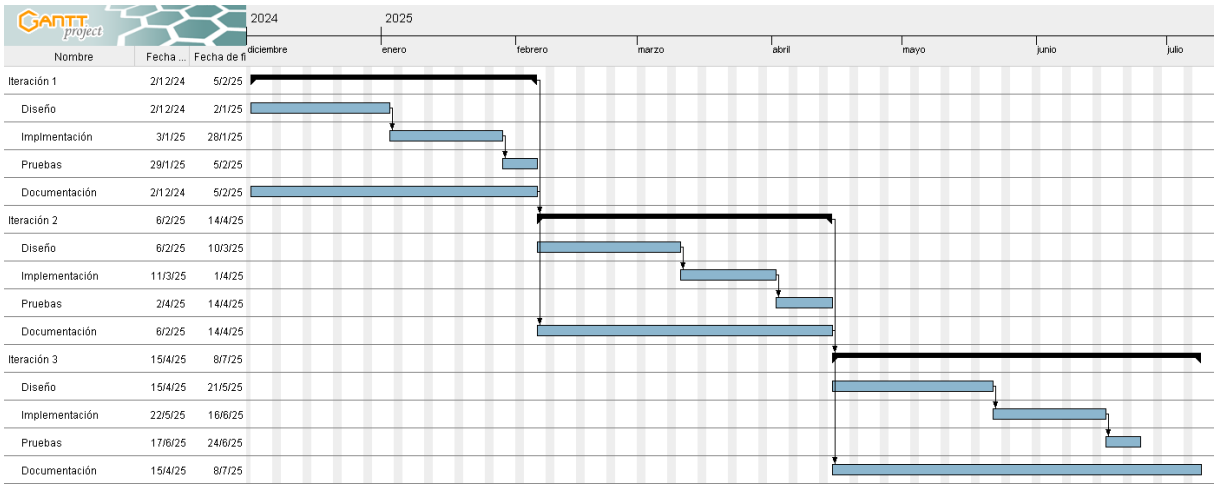

Figura 3.3: Diagrama de Gantt de la planificación final

3.4. Presupuesto y costes

En esta sección se muestra un presupuesto estimado del proyecto, teniendo en cuenta el gasto en componentes hardware, el gasto en maquinas y otros tipos de hardware, y el número de horas de trabajo realizadas. Dado que no se ha estimado necesario ningún software con licencia de pago se excluye del presupuesto la parte relativas a licencias.

Como parte del plan de contingencia del riesgo **R03**, se determinó comprar más de un componente del mismo tipo para los componentes más susceptibles a roturas. Cabe destacar que entre las máquinas y otros hardware solo se tiene en cuenta el uso de una fuente de alimentación. Por último, el número de horas de trabajo se ha calculado, como se ha explicado en las anteriores secciones, teniendo en cuenta que la jornada de trabajo ha sido variable y flexible a lo largo de los distintos meses de trabajo. Por lo tanto, se ha calculado el coste por horas a partir del sueldo anual bruto de un ingeniero informático

junior que actualmente ronda los 28.000€, que se traduciría a 14€ la hora. Todo ello suponiendo una jornada completa de 40 horas semanales y 250 días laborables al año. En la Tabla 3.8 se puede observar un resumen del presupuesto completo estimado.

Concepto	Cantidad	Coste por unidad	Coste total
NodeMCU V1.0	1	9,00€	9,00€
Módulo GPS GY NEO 6MV2	3	3,00€	9,00€
DHT11 de tres pines	1	2,00€	2,00€
Módulo GRPS SIM800L	3	3,00€	9,00€
Arduino Nano Every	1	14,50€	14,50€
Arduino Nano	1	27,10€	27,10€
Kit de cables y resistencias	1	3,00€	3,00€
Fuente de alimentación regulable	1	15,00€	15,00€
Trabajo en horas del alumno	180	14,00€	2.520,00€
			2.608,60€

Tabla 3.8: Coste estimado

3.5. Resumen

Este capítulo ha detallado el modelo de análisis que fundamenta el desarrollo del proyecto. Se ha detallado la planificación inicial, explicando las distintas fases y tareas que guiaron el trabajo como son: el diseño, la implementación, las pruebas y la documentación. Además, se ha realizado un exhaustivo análisis de riesgos, identificando posibles contratiempos y las estrategias de mitigación necesarias para asegurar la viabilidad del proyecto. Se ha mostrado también la desviación de la planificación original, justificando los ajustes realizados, y se ha presentado un presupuesto detallado junto con los costes asociados al proyecto. Con esta base de análisis y planificación, se describe en el Capítulo 4 el primer paso del desarrollo del sistema, el Diseño del Dispositivo IoT, especificando los componentes y la estructura que materializan los requisitos definidos.

4: Diseño del dispositivo IoT

En este capítulo se describe el modelo de diseño seguido para el desarrollo del dispositivo, y las distintas fases por las que se ha transcurrido, dando más detalles sobre los distintos componentes IoT utilizados y su funcionamiento.

El desarrollo del dispositivo IoT propuesto en este proyecto se ha llevado a cabo siguiendo un *modelo de diseño iterativo*. Este enfoque permite construir el sistema de forma incremental mediante una serie de versiones sucesivas, cada una de las cuales mejora o amplía la anterior de alguna forma. A diferencia de modelos lineales como el modelo en cascada, el modelo iterativo permite validar parcialmente el sistema en cada fase y detectar errores tempranos.

Se ha optado por este modelo de diseño con el objetivo de realizar un proceso de desarrollo flexible, permitiendo centrar los esfuerzo en una funcionalidad del sistema en cada iteración. De esta manera se reducen los riesgos asociados al diseño de sistemas IoT compuesto por múltiples componentes que dependen entre si.

El diseño del sistema se ha organizado en tres iteraciones con los siguientes resultados finales en cada una:

- **Primera iteración**: creación de un sistema IoT que permita la captura de datos de posicionamiento global (GPS), además de temperatura y humedad, y que envíe dicha información mediante una red Wi-Fi externa utilizando protocolo MQTT para el envío.
- **Segunda iteración**: creación de un sistema IoT que permita la captura de datos de posicionamiento global (GPS), además de temperatura y humedad, y que envíe dicha información mediante una conexión a internet que no dependa de redes Wi-Fi locales, utilizando protocolo MQTT para el envío.
- **Tercera iteración**: cambio en el diseño del dispositivo de la segunda iteración a un dispositivo más sencillo, compacto y más barato en términos generales, para ello se decidirá primero qué partes del dispositivo merece la pena cambiar.

En las tres iteraciones el funcionamiento general del dispositivo obtenido es el mismo, salvo por ligeras diferencias en cómo es capaz de realizar sus tareas. El dispositivo deberá obtener información de la posición geográfica, la temperatura y humedad ambiente en distintos momentos temporales, y guardará dicha información de manera que pueda consultarse el momento temporal en el cual se tomaron dichas instancias. Tras un número de instancias de datos guardadas se enviará toda la información a un servidor MQTT mediante algún método de conexión a internet. Este número de instancias por conjunto de datos y el tiempo entre instancias o mediciones dependerá del modo de funcionamiento deseado para el dispositivo, por ejemplo, un número bajo de instancias y un tiempo entre mediciones bajo servirá mejor para poder localizar el dispositivo cuando está en movimiento. Por el contrario, un número alto de instancias con un tiempo entre mediciones alto sería más adecuado para los casos en los que el dispositivo no esté en movimiento. En el Capítulo 6 se definen tres modos de funcionamiento especificando el número de instancias y el tiempo entre mediciones, para realizar pruebas y comparar el rendimiento del dispositivo en cada caso.

4.1. Descripción de la primera iteración

En esta primera iteración se partirá desde cero para diseñar un sistema que recoja datos GPS, de temperatura y de humedad, permitiendo enviar dicha información mediante una red Wi-Fi ajena al sistema. Por lo tanto, se eligieron los componentes que a continuación se exponen para el desarrollo de la primera versión.

Placa de desarrollo NodeMCU 1.0 [6], es un kit de desarrollo que contiene un microcontrolador ESP8266 de 32 bits y cuenta también con un transceptor Wi-Fi, para conexiones a internet. Este kit tiene conexión micro-USB para poder cargar el programa que se quiere ejecutar fácilmente desde un ordenador, y estabilización de corriente de 5 voltios a 3.3 voltios, lo cual nos permite conectar componentes que trabajen con ambas medidas de voltaje. Contiene una memoria de 4MB de los cuales aproximadamente 1MB se puede utilizar para almacenar el programa que se desea ejecutar.

Cabe destacar que el NodeMCU tiene 11 pines digitales y un pin analógico, y que también consta de dos interfaces UART (Universal Asynchronous Receiver/Transmitter). UART es un protocolo de comunicación serie asíncrono muy utilizado en dispositivos IoT, ya que permite el intercambio de datos entre dispositivos sin necesidad de compartir un reloj común. El NodeMCU puede ser alimentado por medio de la conexión USB o de manera externa con una batería conectada al pin de alimentación VIN, aunque en el primer caso la corriente máxima que se le puede suministrar no suele pasar de los 250mA.

Módulo GPS GY NEO 6MV2 [7], es un módulo GPS que permite consultar información precisa sobre la geolocalización del mismo. Este módulo viene con un conector para antenas y una antena pequeña ideal para sistemas compactos. Funciona en voltajes entre 3.3V y 5V y con un consumo típico de 45mA, por lo que se puede alimentar sin problema con la conexión de 3.3V del NodeMCU. Utiliza el protocolo UART para

comunicación de los datos de posición que recoge con una precisión de aproximadamente 2.5 metros, lo que nos permite cumplir con el requisito **RF01**.

Además de los pines de recepción y trasmisión de datos, cuenta con un pin PPS (Pulse Per Second) que proporciona una señal de pulso de 1 Hz, altamente precisa, que se sincroniza con el tiempo GPS. Sin embargo, no se utilizará este pin, ya que no requerimos una sincronización temporal tan precisa. El módulo GPS viene con un led interno que nos indicará si está recibiendo corriente y si está recibiendo datos de la posición. Si el led esta encendido sin parpadear, significa que está tratando de fijar la señal GPS, mientras que si el led parpadea, significa que ha encontrado al menos 4 satélites y está recibiendo información de su posición actual que enviará a través de su pin de trasmisión TX. Este led es muy importante, ya que nos puede indicar fallos en el propio componente y fallos en nuestro propio circuito, en particular a lo largo del desarrollo sirvió para detectar múltiples fallos en la soldadura del componente, que retraso la consecución de los objetivos del proyecto.

Sensor DHT11 [8], es un sensor digital que permite medir la temperatura y la humedad relativa del entorno. Este sensor funciona en voltajes entre 3V y 5.5V y con un consumo que rara vez supera 0.3mA, por lo que se puede alimentar sin problema con la conexión de 3.3V del NodeMCU. El DHT11 utiliza una interfaz de comunicación digital de un solo hilo, lo que simplifica su conexión con microcontroladores. La precisión de sus mediciones es de $\pm 2^{o}$C de temperatura y $\pm 5\,\%$ de humedad. Gracias a su diseño compacto y bajo consumo energético, este sensor es ideal para cualquier sistema IoT que se desee diseñar, además de permitirnos cumplir con el requisito **RF02** al incorporarlo al dispositivo.

Para poder trabajar con este prototipo se dispusieron los componentes y los cables necesarios en una placa de prototipado. En la Figura 4.4 se puede observar un esquema del cableado creado con la aplicación *Fritzing* [9], para que sea más fácil entender el circuito, y el prototipo real se puede observar en la Figura 4.5. Como se puede ver en el esquema, el cable amarillo va conectado al pin de trasmisión (TX) del componente GPS y el verde al pin de recepción (RX). Este detalle es importante a la hora de establecer la comunicación entre el NodeMCU y el componente, ya que el pin de trasmisión deber ir conectado a un pin de recepción y al contrario. Cabe destacar también que todas las conexiones a tierra acaban en el NodeMCU, de esta manera se asegura que todas las tomas de tierra estén conectadas al mismo lugar, ya que sino el dispositivo no funcionaría correctamente.

Como ya se ha comentado, el dispositivo se podría alimentar tanto con una batería externa conectando el cable de voltaje a la entrada VIN del NodeMCU, como con la conexión micro-USB conectada a un ordenador. De esa manera la salida de 3V3 da alimentación tanto al componente GPS como al sensor de temperatura y humedad.

El dispositivo diseñado en esta iteración es capaz de almacenar información GPS, de humedad y temperatura ambiental correctamente y enviarla a un servidor MQTT siempre y cuando se disponga de una conexión Wi-Fi adecuada, y la antena de componente GPS sea capaz de obtener señal satélite. Con la intención de mejorar el rendimiento y autonomía del dispositivo actual se plantea para la siguiente iteración eliminar la necesidad de una

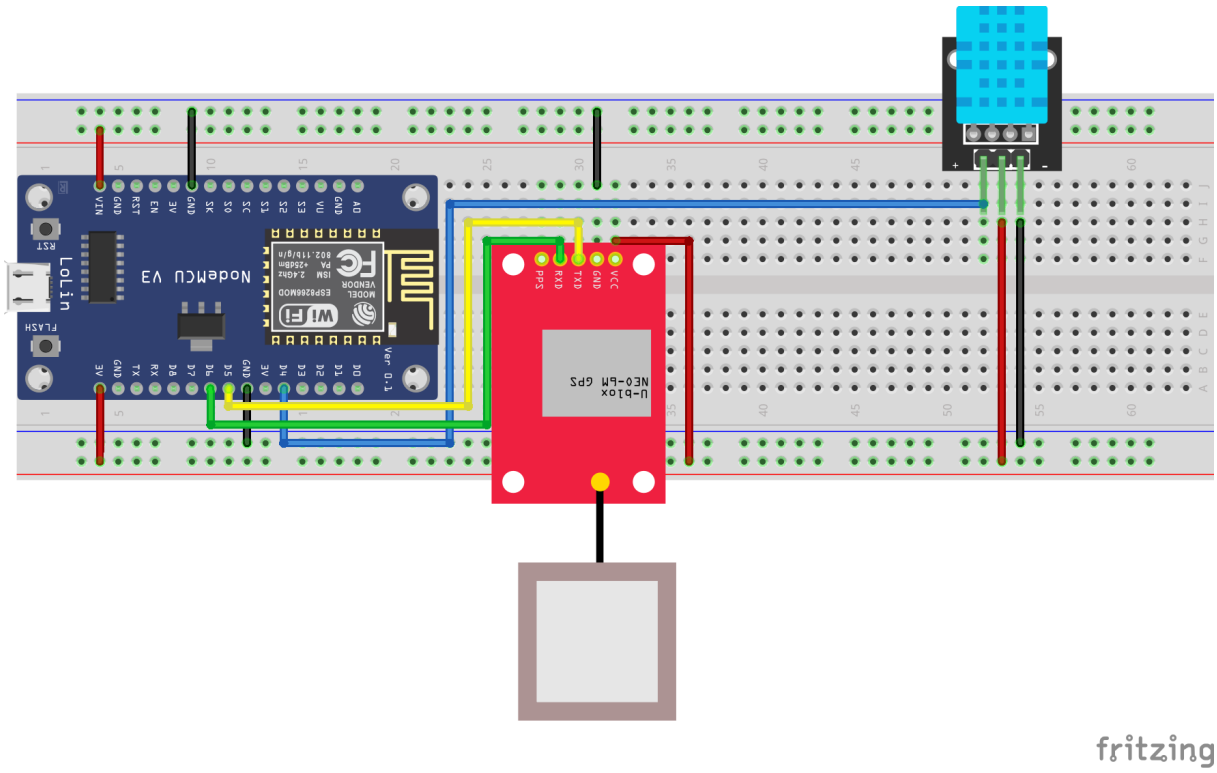

Figura 4.4: Esquema del sistema 1

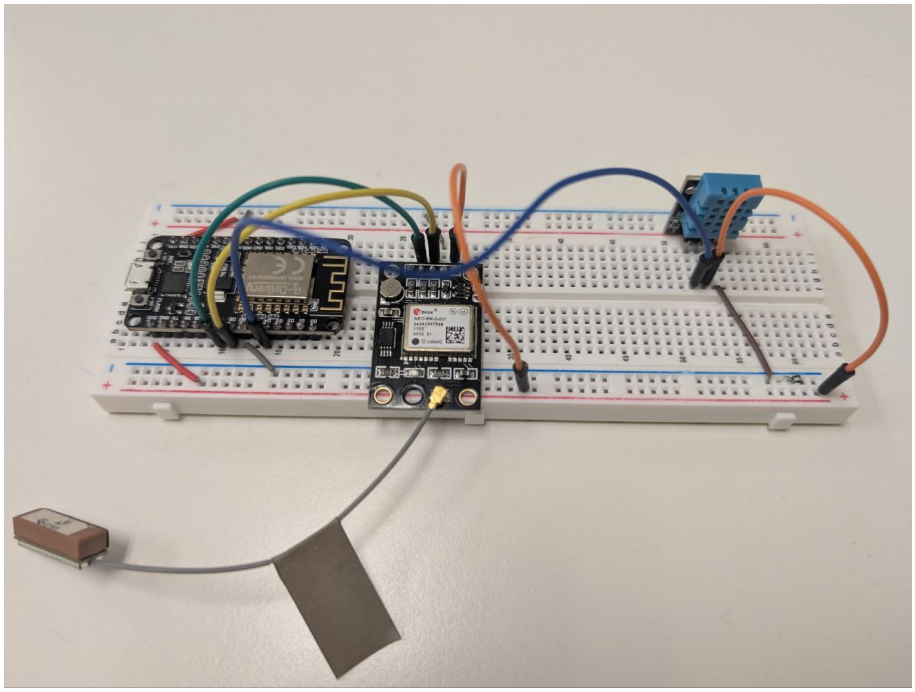

Figura 4.5: Imagen del prototipo 1

conexión Wi-Fi mediante el uso de un componente capaz de conectar el dispositivo a internet en cualquier momento, gracias a la red móvil. De esta manera el dispositivo solo requerirá de una cobertura móvil estable para el envío de datos, en lugar de un punto de conexión Wi-Fi, cumpliendo así con el requisito **RF09**.

Formato de mensajes MQTT

En el diseño propuesto para la primera iteración se transmiten datos de posición geográfica, temperatura y humedad mediante el protocolo MQTT, un estándar ampliamente utilizado en sistemas IoT por su eficiencia y bajo consumo de ancho de banda. Como parte de esta iteración, también se define la estructura de los mensajes MQTT, con el objetivo de garantizar la trazabilidad, integridad y utilidad de los datos transmitidos. Para ello, se ha optado por el envío de mensajes en formato JSON, siguiendo la estructura que se muestra en la Figura 4.6, y que a continuación se detalla.

```
{
  "sha256": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "mac": "1E:E5:89:44:FE:19",
  "IMEI": "123456789012345",
  "support": [
    "GPS",
    "TEMP"
  ],
  "data": [{
    "timestamp": "1736875157",
    "lat": "41.662739",
    "long": "-4.705044",
    "humid": "34.00",
    "temp": "18",
    "altitude": "680.00",
    "speed": "0.00"
  },
  {
    "timestamp": "1736876157",
    "lat": "41.662749",
    "long": "-4.705044",
    "humid": "34.00",
    "temp": "18",
    "altitude": "680.00",
    "speed": "0.00"
  }]
}
```

Figura 4.6: Estructura de ejemplo del mensaje JSON

- **sha256**: Hash criptográfico que permite validar la integridad del mensaje en el lado del servidor.
- **mac**: Dirección MAC de la red Wi-Fi a la que se conecte el dispositivo IoT que generó el mensaje, pudiendo así identificar el origen de los datos.
- **IMEI**: Dirección IMEI del módulo GPRS que se utiliza en la segunda iteración como sustitutivo para la conexión Wi-Fi, de tal manera que se pueda identificar el origen de los datos cuando no se dispone de una dirección MAC.
- **support**: Lista de tipos de datos que puede proporcionar el dispositivo de origen. En este caso incluye GPS y TEMP (temperatura y humedad).
- **data**: Array de datos recolectados cuya cantidad de elementos dependerá de la frecuencia de recogida y envío de datos. Por cada elemento se registran siempre los siguientes datos:
  - **timestamp**: Marca de tiempo UNIX que indica el momento exacto de la recolección de los datos, obtenida del componente GPS.
  - **lat y long**: Coordenada geográficas (latitud y longitud) que representan la posición del dispositivo al recoger los datos.
  - **humid**: Nivel de humedad relativa del ambiente, en porcentaje, del lugar donde se encuentra el dispositvo al recoger los datos.
  - **temp**: Temperatura ambiente, registrada en grados centigrados, del lugar donde se encuentra el dispositivo al recoger los datos.
  - **altitude**: Altitud, en metros, a la que se encuentra el dispositivo al recoger los datos.
  - **speed**: Velocidad, en kilometros por hora, a la que va el dispositivo al recoger los datos.

El formato JSON ofrece múltiples ventajas para la transmisión de datos en sistemas IoT. Su compatibilidad lo convierte en un estándar ampliamente reconocido y fácilmente interpretable por distintos lenguajes de programación y plataformas. La incorporación de un campo de verificación como el sha256 garantiza la integridad del mensaje a lo largo de todo el proceso de transmisión, lo que nos permite cumplir con los requisitos **RF06** y **RF07**. Además, su estructura es escalable, lo que permite incorporar fácilmente nuevos tipos de datos o sensores si el dispositivo lo requiere. Por último, su eficiencia se ve reforzada por el uso de arrays, que permiten agrupar varios puntos de recolección en un único mensaje, reduciendo el uso del ancho de banda y pudiendo ajustar la frecuencia de envíos sin mayor problema.

4.2. Descripción de la segunda iteración

Para la segunda iteración se partirá del sistema diseñado en la primera iteración, añadiendo un componente que permita al dispositivo conectarse a internet sin necesidad de una red Wi-Fi local. Por ello, se ha decidido utilizar el módulo GSM/GPRS **SIM800L** junto con una tarjeta de telefonía móvil que permita la conexión a internet a través de un proveedor.

El **módulo GSM/GPRS SIM800L** [10], es un componente de comunicación que puede dotar a nuestro sistema IoT de conectividad móvil mediante redes GSM y GPRS. GSM (Globas System for Mobile Communications) es un estándar para redes móviles desarrollado en Europa en los años 80 y 90, comúnmente conocido como segunda generación o 2G, y que se enfocaba en proporcionar comunicaciones de voz digital y otros servicios básicos como SMS [11]. GPRS (General Packet Radio Service) es una extensión del sistema GSM que introduce el uso de datos por paquetes, lo que lo convierte en una tecnología más eficiente para el acceso a internet y transmisión de datos, para más información véase [12], a veces se la referencia como la 2.5G ya que es una evolución entre la 2G y la 3G.

De la misma manera que el módulo GPS GY NEO 6MV2, el SIM800L se comunica a través de una interfaz UART, utilizando sus pines de trasmisión y recepción de datos, mediante lo que se denomina comandos AT que permiten enviar y recibir SMS, enviar paquetes de datos vía GPRS, o incluso realizar una llamada de voz. Existen distintos comandos AT que sirven a diversos propósitos, realizando los comandos adecuados en el orden indicado se puede llevar a cabo las distintas funcionalidades de este tipo de módulo, véase [13] para más información sobre los comandos AT. Si bien este módulo tiene muchas funcionalidades, para este dispositivo solo se utilizará el envío de datos por conexión GPRS, específicamente utilizando el protocolo MQTT, por lo que hay una fila entera de pines que no se soldarán. Esta fila de pines aporta funcionalidades como la posibilidad de conectar un altavoz o micrófono para llamadas telefónicas.

El SIM800L funciona con voltajes de entre 3.4V y 4.4V, mandando por su pin de trasmisión mensajes de advertencia cuando se sobrepasa este último. Si bien su consumo en reposo es de aproximadamente 20mA, este consumo puede tener picos de hasta 2A durante las trasmisión de datos, por lo que necesitaremos una fuente de alimentación externa para alimentar este componente. Este módulo viene con una antena helicoidal externa que se debe soldar al pin NET, para que el componente funcione correctamente. Dispone también de una entrada para tarjetas de telefonía móvil, que se deberán colocar en una posición específica para su correcto funcionamiento, y un led interno que nos indicará en que estado se encuentra el componente. Hay distintas frecuencias de parpadeo del led con sus correspondientes significados plasmados en el datasheet del componente [10], a continuación se explican en detalle:

- Tiempo entre parpadeos de 800 milisegundos, significa que el SIM800L está recibiendo corriente y está intentando conectarse a la red móvil, aunque aún no esté conectado.

- Tiempo entre parpadeos de 3 segundos, significa que ha logrado conectares a la red móvil (GSM) y puede realizar llamadas y enviar o recibir SMS.
- Tiempo entre parpadeos de 300 milisegundos, significa que se ha establecido la comunicación GPRS, y por lo tanto tiene conexión a internet.
- 7 parpadeos en intervalos de 800 milisegundos seguidos de 5 segundos sin parpadeos, significa que el SIM800L se esta reseteando cada 7 segundos, probablemente porque no está recibiendo la corriente adecuada, ya sea por voltaje o amperaje inadecuados.

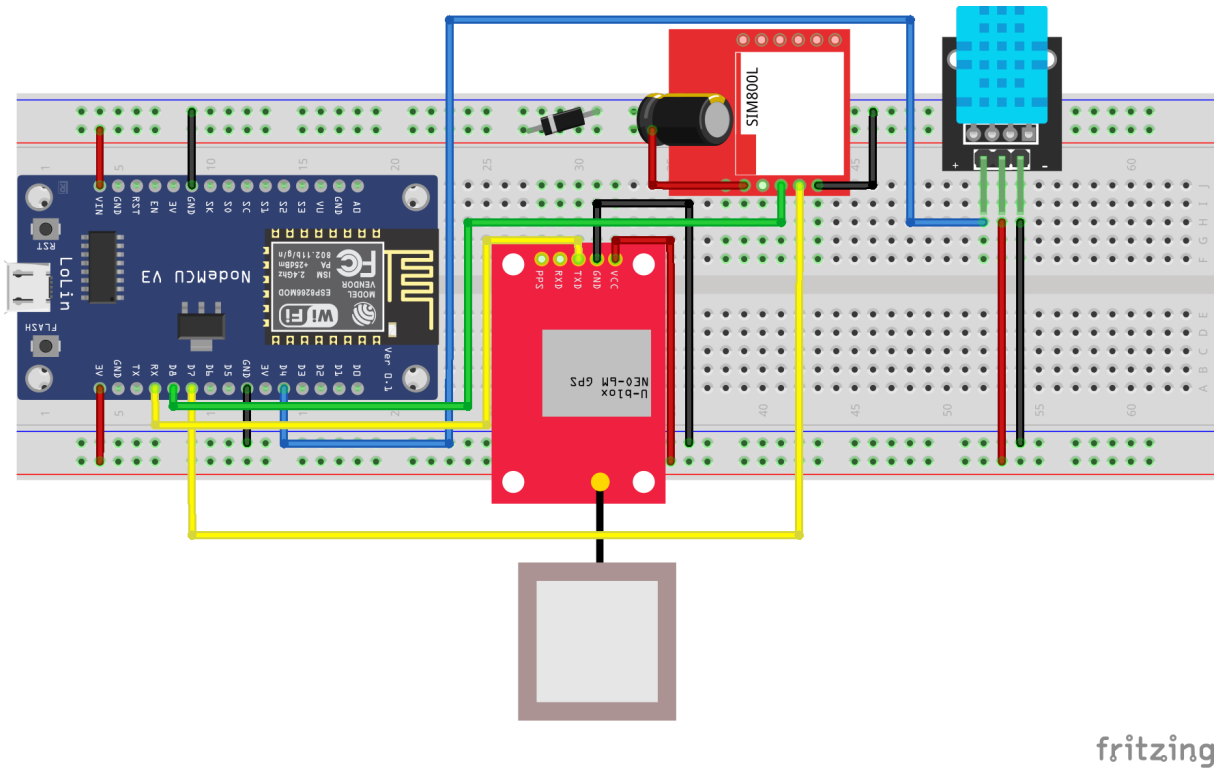

Figura 4.7: Esquema del sistema 2

A la hora de integrar el nuevo componente en el sistema desarrollado en la primera iteración se deben hacer algunos ajustes, ya que se necesita manejar dos interfaces UART distintas, una para el módulo GPS y otro para el SIM800L. Dado que el módulo GPS trasmite información de manera automática sin necesidad de realizar peticiones, se ha optado por mantener solo la conexión al pin de trasmisión que irá conectado al pin de recepción hardware del NodeMCU, de esta manera se pueden leer los datos que va trasmitiendo el GPS sin problema. Por otro lado, el SIM800L se debe conectar al NodeMCU con ambos pines, trasmisión y recepción, para poder darle instrucciones de qué tipo de conexión llevar a cabo y qué datos enviar. Todo esto se puede observar en el esquema del cableado representado en la Figura 4.7 y en la imagen del prototipo ya montado de la Figura 4.8.

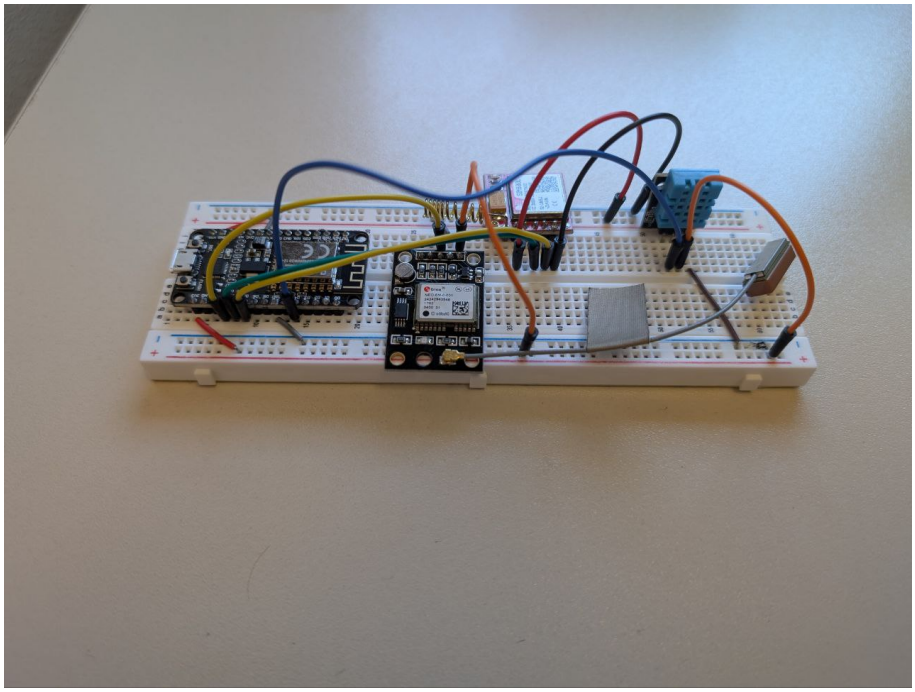

Figura 4.8: Imagen del prototipo 2

Como ya se ha explicado, el módulo SIM800L necesita obligatoriamente recibir corriente mediante una fuente de alimentación externa, ya que la corriente que puede extraer de la conexión micro-USB del NodeMCU no es suficiente en amperaje para su funcionamiento. Además, el datasheet del componente recomienda añadir un condensador de desacoplo cerca del módulo y un diodo de protección en el circuito para filtrar el rizado de la tensión y suprimir los picos de corriente, lo que ayuda a estabilizar el voltaje de alimentación y a prevenir la desestabilización del circuito. Sin embargo, se ha podido comprobar en diversas pruebas con la fuente de alimentación, que el sistema funciona sin estos elementos añadidos al circuito, aunque estos serían muy recomendables como medida de seguridad en el dispositivo final.

Como en el caso de la primera iteración el sistema diseñado podría ser alimentado externamente conectando la fuente al pin VIN del NodeMCU. De nuevo es importante destacar que la todas las tomas de tierra deben estar conectadas, por lo que al añadir una fuente de alimentación externa se deberá conectar su respectiva toma de tierra a uno de los pines GND del NodeMCU.

El dispositivo diseñado en esta iteración es capaz de almacenar información GPS, de humedad y temperatura ambiental correctamente, y realizar el envío de esta mediante un componente GPRS capaz de conectarse a internet por red móvil, siempre y cuando exista cobertura y señal satélite adecuadas. Con esto se ha logrado una mayor autonomía y rendimiento del dispositivo, con respecto a la iteración anterior, siendo un buen candidato de producto final para este proyecto. Sin embargo, la solución obtenida es más compleja, más cara y de mayor tamaño con respecto a la anterior iteración, por lo que se puede

buscar soluciones que permitan reducir estas tres características del dispositivo final en la siguiente iteración, ya sea mediante sustitución del hardware como simplifcación del software.

4.3. Descripción de la tercera iteración

Para esta última iteración se ha tratado de analizar el dispositivo obtenido hasta el momento para reducir tres factores clave para el sistema final:

- **Reducción de complejidad**: en términos de funcionalidad tanto en el software como en el hardware, prescindiendo de utilidades innecesarias como es el caso de la conexión a redes Wi-Fi. En este factor también entraría la sustitución de componentes por otros más sencillos.
- **Reducción de costes**: realizar cambios en el hardware que abarate el sistema final. Hay que tener en cuenta que el sistema deberá seguir realizando las mismas funcionalidades, por lo que es importante elegir qué partes sustituir.
- **Reducción de tamaño físico**: de cara a cumplir el requisito **RNF03** sería deseable reducir el tamaño final del sistema para su posterior fabricación, de nuevo sustituyendo componentes por otros más pequeños.

Como se puede observar, los tres factores descritos implican en gran parte la sustitución de componentes del sistema actual, por lo que es imprescindible detectar sustituciones que no aumenten innecesariamente el tiempo de desarrollo, pero que cumplan con las reducciones estipuladas y las funciones hasta ahora logradas.

Analizando el sistema desarrollado en la segunda iteración se pudo observar que el componente que más espacio general ocupa es la propia placa NodeMCU, y dado que algunas de sus funcionalidades adicionales no se utilizan en este sistema, como es el caso de la conexión a redes Wi-Fi, es un componente que merece la pena sustituir. Por otro lado, existen módulos GSM que tienen también capacidad de geolocalización, con lo cual sustiuiríamos dos componentes, SIM800L y GPS GY NEO 6MV2, por uno solo reduciendo así el tamaño general del sistema. Sin embargo, estos componentes a menudo se encuentran a precios bastante altos para un componente IoT, cerca de 30 euros, mientras que el SIM800L se puede encontrar fácilmente por 2 o 3 euros y el módulo GPS utilizado por un precio similar. Por lo que esta sustitución encarecería bastante el sistema final, y dado que los dispositivos pretenden ser desechables, será más importante reducir costes que tamaño.

Con el análisis realizados se decidió plantear un cambio de placa del sistema, sustituyendo el NodeMCU por otra placa más sencilla y barata. Sin embargo, NodeMCU es una placa bastante barata de por si, principalmente porque utiliza el chip ESP8266, que es un microcontrolador de muy bajo coste con conectividad Wi-Fi integrada, además de ser una plataforma de código abierto que se beneficia de la producción masiva. Se puede encontrar por aproximadamente 9 euros en tiendas oficiales y alrededor de 3 euros en tiendas no

oficiales, aunque su precio varía bastante. Con esto en mente lo ideal sería buscar una placa cuyo microcontrolador sea más barato en términos generales que el ESP8266, lo cual implica buscar un microcontrolador más sencillo. Para evitar que el desarrollo se puede alargar demasiado, se trató de buscar opciones que en principio mantengan compatibilidad con el resto de componentes y también con las librerías utilizadas hasta el momento, si es posible.

Sustitución de placa: Arduino Nano Every

Dado que la mayoría de librerías son compatibles con todas las placas de Arduino según su documentación, se revisaron las distintas opciones y se eligió inicialmente la placa **Arduino Nano Every**. Esta placa utiliza el microcontrolador ATmega4809, el cual presenta características más limitadas en comparación con el ESP8266. Por ejemplo, el ATmega4809 dispone de una memoria flash de 48KB, frente a los 4MB del ESP8266. En cuanto a la velocidad, el ATmega4809 opera a una frecuencia máxima de 20 MHz, mientras que el ESP8266 alcanza los 160 MHz. Además, la SRAM del ATmega4809 es de 6KB, muy inferior a los 64KB de RAM del ESP8266. Un punto clave es que el Arduino Nano Every carece de conectividad Wi-Fi, algo que el ESP8266 sí incorpora. Sin embargo, el precio del Arduino Nano Every tanto en tiendas oficiales como no oficiales supera al del NodeMCU, siendo este de unos 7 euros en tiendas no oficiales, aunque sea poco común. Esto se debe principalmente a su margen de beneficio, investigación y desarrollo, y el soporte técnico asociado a la marca Arduino. Aún así, cuando se compara con el NodeMCU, el Arduino Nano Every no oficial es más económico por sus componentes más simples. Si bien el espacio de memoria de programa es más reducido que el del ESP8266, se pudo comprobar que la compilación del programa desarrollado en la anterior iteración, con el *toolkit* específico para la placa Arduino Nano Every, ocupaba un 80 % del espacio disponible.

Una de las características distintas al NodeMCU que hacen del Arduino Nano Every una opción más interesante es que tiene varias interfaces UART disponibles. Esto simplifica la implementación de la solución, ya que permitiría utilizar estas interfaces UART en lugar de utilizar la librería SoftwareSerial. Además, utilizar comunicación serial hardware es siempre más fiable que emularla mediante software. Por otro lado, este módulo trabaja con 5V en sus distintos pines y requiere de un voltaje de alimentación de entre 7V y 21V, por lo que para componentes como el SIM800L se debe reducir el voltaje de la señal de trasmisión para evitar dañar el componente.

Inicialmente se asumió que el voltaje de trasmisión del resto de componentes sería suficientemente alto como para que el Arduino Nano Every sea capaz de interpretar correctamente la señal. Sin embargo, tras varias pruebas con el SIM800L se pudo observar que las respuestas que este daba a distintos comandos AT contenían caracteres corruptos. En las especificaciones del SIM800L se puede comprobar que el voltaje máximo del pin de trasmisión es de 2.8V [10], mientras que según las especificaciones del ATmega4809 [14], para poder detectar correctamente la señal, requiere un voltaje mínimo de entrada de $0.7 \cdot VCC \approx 3.5V$, siendo VCC el voltaje con el que trabajan los pines, es decir,

5V. La opción más fiable para resolver este problema es utilizar un "logic level shifter" el cual permite la comunicación entre componentes que operan con diferentes voltajes lógicos, adaptando las señales para que sean compatibles. Sin embargo, esto encarecería y complicaría la nueva solución, por lo que finalmente se decidió descartar esta placa.

Sustitución de placa: Arduino Nano

Como segunda opción para reemplazar el NodeMCU, se consideró el Arduino Nano, que utiliza el microcontrolador ATmega328p. Este es muy similar al Arduino Nano Every (basado en el ATmega4809), con algunas diferencias clave: el ATmega328p cuenta con una memoria flash de 32KB y una SRAM de 2KB. Además, el Arduino Nano suele encontrarse por, aproximadamente, 1 euro en tiendas no oficiales. Los requisitos de alimentación y voltaje de trabajo de los pines son los mismos que en el Arduino Nano Every, por lo que tendremos que tenerlo en cuenta de la misma manera en el circuito nuevo del dispositivo. En este caso las especificaciones del microcontrolador nos indican que el voltaje de entrada mínimo requerido se encuentra en $0.6 \cdot VCC \approx 3V$, siendo de nuevo VCC 5V, [15], por lo que se asumió que esta vez no debería ser necesario un "logic level shifter" para hacerlo funcionar. Esto es algo que se confirmó posteriormente al ver que la comunicación con el SIM800L no devolvía respuestas con caracteres corruptos.

Como se puede ver, el Arduino Nano cuenta con mucho menos espacio de memoria para el programa lo cual es un problema al utilizar varias librerías distintas para el funcionamiento de la anterior iteración. Esto se puede comprobar compilando el programa desarrollado en la anterior iteración con el *toolkit* para Arduino Nano. Esto indica el porcentaje de espacio de memoria de programa ocupado por la compilación realizada, que en primera instancia mostraba un valor del 124 %, lo que implica que se esta ocupando un 24 % extra del espacio disponible. Esto requeriría una reducción significativa del programa, lo cual representa un desafío considerable dado que la mayoría de las librerías empleadas son esenciales. Otra característica distinta con respecto al Arduino Nano Every es que solo tiene una interfaz de comunicación UART, igual que en el caso del NodeMCU, lo cual implica que se tendrá que seguir utilizando la librería de SoftwareSerial.

Si bien la trasmisión de datos hacia el Arduino Nano no debería ser un problema, como ya se ha explicado, la trasmisión en el sentido contrario si podría dañar el componente SIM800L, ya que en sus especificaciones se menciona que está pensado para voltajes de entrada de datos de entre 2.5V y 2.8V. Esto requiere de una bajada de tensión del pin que se utilizará como trasmisión en el Arduino Nano, mediante un puente de resistencias. Este puente consiste en conectar dos resistencias: una resistencia R_1 a la salida del pin de trasmisión del Arduino de 5V y otra resistencia R_2 a tierra, con la salida de 2.8V tomada entre ambas. La relación para el divisor de voltaje se obtiene mediante la siguiente ecuación:

$$V_{out} = V_{in} \cdot \frac{R_2}{R_1 + R_2}$$

Utilizando una resistencia R_1 de $10k\Omega$ con la intención de obtener un voltaje de salida $V_{out} = 2.8V$ y con un voltaje de entrada de $V_{in} = 5V$, se puede despejar la segunda resistencia, $R_2 \approx 12.7k\Omega$. Siguiendo los valores de resistencia comerciales más comunes lo más sencillo sería utilizar dos resistencias en serie para llegar a los $12.2k\Omega$, mediante una resistencia de $10k\Omega$ y otra de $2.2k\Omega$. Esto debería dar un valor de voltaje de salida entre los 2.5V y 2.8V. Se puede revisar el esquema del circuito para el Arduino Nano en la Figura 4.9 y el sistema montado sobre el que se realizaron pruebas en la Figura 4.10.

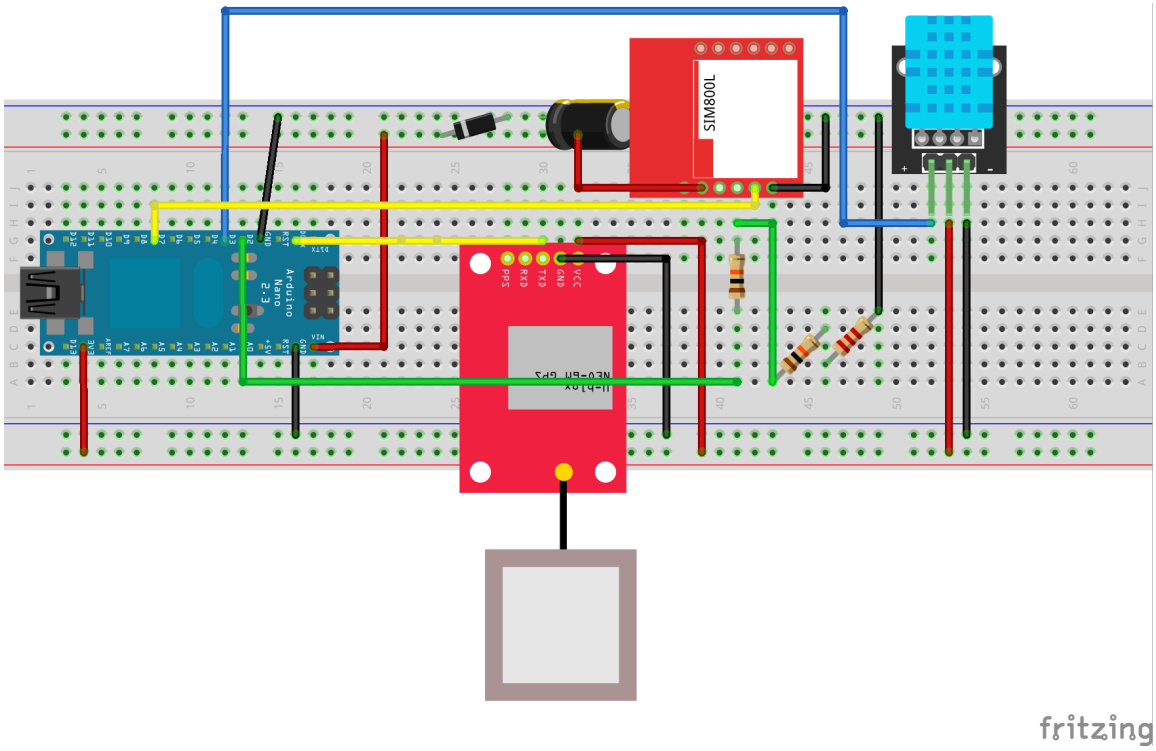

Figura 4.9: Esquema del sistema 3

Como vemos en el esquema de la Figura 4.9 se deben colocar las resistencias mencionadas para evitar daños en el componente SIM800L. Hay que tener en cuenta que el Arduino Nano, igual que el Arduino Nano Every, requiere de un voltaje mínimo de entrada de 7V para su correcto funcionamiento, aunque este puede ser sustituido por la alimentación dada por el puerto mini-usb. Sin embargo, para el sistema final independiente se requeriría de algún método de bajada de tensión, ya que el SIM800L trabaja con voltajes entre 3.4V y 4.4V, siendo lo ideal 4V. Esto revela un factor en contra para la sustitución del NodeMCU por un Arduino Nano, ya que el NodeMCU es capaz de trabajar independientemente con 5V según sus especificaciones, lo cual hace más sencillo una bajada de tensión para el SIM800L. Además, se llegó a comprobar que el NodeMCU es capaz de funcionar correctamente con un voltaje de 4.2V al mantenerlo conectado a la fuente de alimentación regulable. Igual que en el circuito de la iteración anterior, se recomienda el uso de un condensador y diodos específicos para alimentar al SIM800L según su propia especificación.

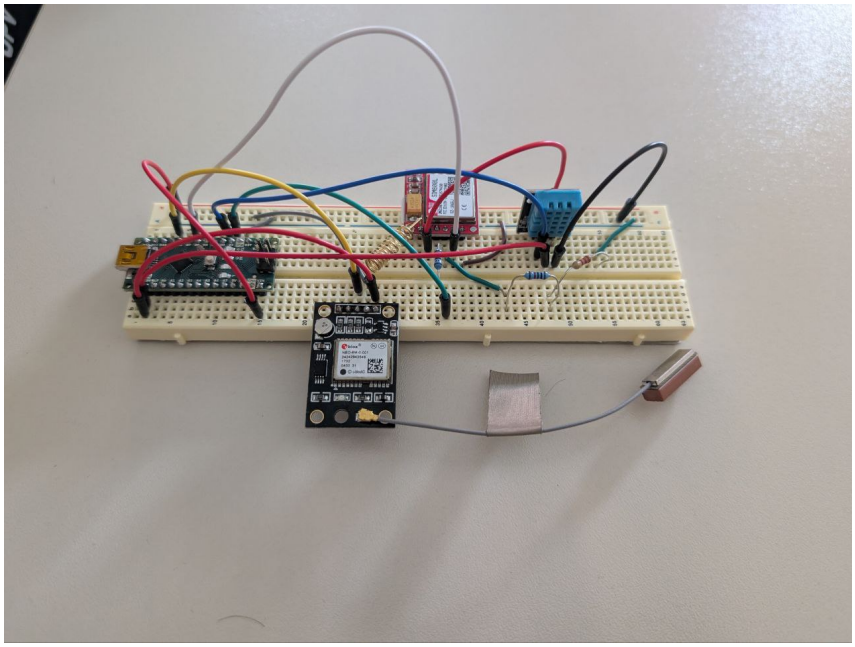

Figura 4.10: Imagen del prototipo 3

4.4. Resumen

Este capítulo ha profundizado en el diseño del dispositivo IoT, detallando el modelo de diseño iterativo adoptado para su desarrollo, el cual permitió construir el sistema de forma incremental mediante versiones sucesivas. Se han descrito las distintas fases de esta evolución, comenzando con una primera iteración centrada en componentes esenciales como la placa de desarrollo NodeMCU, un componente GPS y un sensor DHT11, que permitirán el funcionamiento general del dispositivo bajo la premisa de una conexión a red Wi-Fi. Además se define el formato de lo mensajes MQTT que se enviarán al servidor. Posteriormente, se define la segunda iteración en la que se incorporó un módulo GPRS (SIM800L) para asegurar la autonomía del dispositivo, eliminando así la necesidad de una red Wi-Fi para su funcionamiento. En esta iteración se abordaron los distintos requisitos de alimentación que añade el uso del nuevo componente presentado. Finalmente, la tercera iteración exploró la sustitución de los distintos componentes para reducir la complejidad, tamaño y coste de la solución anterior, proponiendo un diseño nuevo en el que se sustituye la placa NodeMCU por la placa Arduino Nano. En conjunto, el capítulo ha cubierto los detalles de diseño necesarios para llevar a cabo la siguiente fase de implementación, que se define en el Capítulo 5, y que describe el código desarrollado para cada una de las iteraciones ya presentadas.

5: Detalles de la implementación software

En este capítulo se procede a describir detalles más específicos sobre la implementación y el desarrollo del software para el dispostivo IoT. Igual que en el Capítulo 4, aquí se explicarán las modificaciones realizadas a nivel de software en cada iteración del desarrollo, junto con los detalles técnicos que se han tenido que considerar al añadir nuevos componentes.

Para el desarrollo del software de este proyecto se ha utilizado a lo largo de todas las iteraciones el entorno Arduino IDE [16], a pesar de que el principal microcontrolador empleado, NodeMCU, no forme parte de la gama oficial de productos Arduino. Esta decisión se debe a la amplia compatibilidad del IDE con diferentes placas de desarrollo, así como a su entorno intuitivo, la facilidad para gestionar bibliotecas de terceros y la extensa comunidad de soporte que ofrece. Además, su simplicidad para cargar programas directamente a las placas y su integración con herramientas de depuración básicas como el "Serial monitor" lo convierten en una opción práctica y eficiente para el desarrollo de sistemas IoT.

El código se ha desarrollado en archivos con extensión `.ino`, comúnmente conocidos como *sketches*. Estos archivos están escritos en un subconjunto simplificado de los lenguajes de programación C y C++, adaptado específicamente para facilitar la programación de microcontroladores en placas Arduino. Este tipo de archivos tienen una estructura particular que consta de dos funciones principales en su ejecución: `setup()` y `loop()`. La función `setup()` se ejecuta al iniciar el programa una sola vez y se utiliza para realizar la configuración inicial del dispositivo, inicialización de variables y de comunicaciones seriales y otros aspectos propios del inicio de una ejecución. Por otro lado, la función `loop()` contiene la parte del código que se ejecutará repetidamente mientras el dispositivo este encendido, funciona similar a un bucle *while*, aunque en este caso la condición es que el dispositivo reciba corriente.

El código desarrollado a lo largo del proyecto se puede encontrar en un repositorio de GitHub que también se ha proporcionado en la contribución para las Jornadas Sarteco [17].

5.1. Implementación de la primera iteración

En esta iteración se desarrollará desde cero el código, utilizando distintos ejemplos y librerías. Por lo tanto en esta fase se incluyeron las siguientes funcionalidades:

- Comunicación con el componente GPS y extracción de datos del mismo.
- Comunicación con el sensor de temperatura y humedad, y extracción de datos del mismo.
- Encapsulado de los datos obtenidos en un formato JSON específico.
- Conexión a internet vía Wi-Fi y envío de los datos mediante protocolo MQTT.

Para el desarrollo de esta iteración se han utilizado diversas librerías, algunas de las cuales no están incluidas por defecto en el entorno de desarrollo Arduino IDE. No obstante, pueden ser instaladas fácilmente mediante el *Library Manager* del propio IDE. En ciertos casos, ha sido necesario obtener las librerías directamente desde sus repositorios oficiales, e incluso modificar algunos de sus archivos internos para garantizar el correcto funcionamiento del programa. A continuación, se detallan dichas librerías y los ajustes realizados:

- **TinyGPSPlus** en [18]: proporciona una interpretación comprensible y orientada a objetos de las sentencias GPS (NMEA). NMEA es el formato estándar que utilizan los dispositivos GPS para informar de la ubicación, la hora, la altitud, etc. Esta librería nos permite acceder a distintos datos adicionales de posición como la altitud o la velocidad, lo que nos permite cumplir con el requisito **RF10**.
- **SoftwareSerial** en [19]: permite la comunicación serial en otros pines digitales, replicando dicha funcionalidad mediante software. Cabe destacar que, aunque es posible tener dos SoftwareSerials activos, solo uno de los dos podrá recibir datos al mismo tiempo, esto se observó durante el desarrollo de la segunda iteración, cuando tras varios problemas durante la implementación se volvió a consultar la documentación en profundidad.
- **ArduinoJson** en [20]: permite la serialización y deserialización de datos JSON en proyectos Arduino de manera eficiente.
- **Crypto** en [21]: librería genérica con múltiples funcionalidades de criptografía para proyectos Arduino. En este proyecto nos centraremos en la parte correspondiente al algoritmo SHA256. Si bien esta librería aparece en el Library Manager del IDE de

Arduino, este método de instalación ha dado problemas varias veces y se ha tenido que instalar manualmente a partir de su repositorio oficial.

- **ESP8266Wi-Fi** en [22]: permite acceder y utilizar la funcionalidad de conexión Wi-Fi que posee el chip ESP8266. Para instalarlo se debe primero instalar el core de ESP8266, las instrucciones de instalación se pueden encontrar en [22].
- **Adafruit MQTT Library** en [23]: proporciona soporte para la conexión y envío de datos mediante protocolo MQTT a los servidores de Adafruit. Los mensajes enviados se pueden comprobar en la propia página de Adafruit, siempre y cuando se disponga de una cuenta. Esta librería tiene una limitación en el envío de datos MQTT, permitiendo paquetes de no más de 256 bytes. Sin embargo, los mensajes que se envían en nuestro caso sobrepasan fácilmente este límite, por lo que se ha optado por cambiar el tamaño máximo de paquetes de envío, modificando el código interno de la librería. A mayores se han modificado algunas funciones de la misma para poder consultar el tamaño de los paquetes justo antes del envío, permitiendo así tomar mediciones durante las pruebas.
- **DHT sensor library** en [24]: permite leer fácilmente la temperatura y humedad de los sensores DHT11, DHT21 y DHT22. En nuestro caso el sensor utilizado es un DHT11, por lo que se debe especificar en el código el typo de sensor mediante la directiva de preprocesador `#define DHTTYPE DHT11`.

En la Figura 5.11 se puede apreciar un diagrama de flujo del código desarrollado para esta iteración. En el se describe el flujo habitual de un *sketch* de Arduino, distinguiendo entra la función `setup()` y la función `loop()`. Como se puede observar al encender el dispositivo se procede a inicializar la comunicación con el componente GPS mediante SoftwareSerial, la conexión con la red Wi-Fi previamente especificada y el sensor de temperatura y humedad. A partir de este punto el dispositivo irá comprobando si hay datos GPS que decodificar y guardarlos, de esta manera se va creando el mensaje que se enviará al servidor MQTT. Entre guardados de datos se introduce una espera de una cantidad de tiempo ajustable, para así recopilar datos en distintos momentos. Finalmente, cuando el mensaje está completado se realiza la conexión al servidor MQTT de Adafruit, se serializa el mensaje en formato JSON y se publica.

Dificultades encontradas

Durante el desarrollo de esta primera iteración se trató por primera vez con muchos de los componentes ya descritos. Esto significa que durante esta iteración se tuvo que soldar por primera vez muchos de estos componentes, ya que no todos los componentes se venden con sus respectivos pines ya soldados. Si bien soldar parece una tarea trivial, es importante realizar una buena soldadura de los pines, ya que sino no recibirán corriente o mandarán y recibirán información adecuadamente. Específicamente el módulo GPS GY NEO 6MV2 tuvo que ser soldado y revisado en repetidas ocasiones, ya que debido a malas soldaduras, no recibía corriente e incluso no era capaz de recibir y trasmitir información.

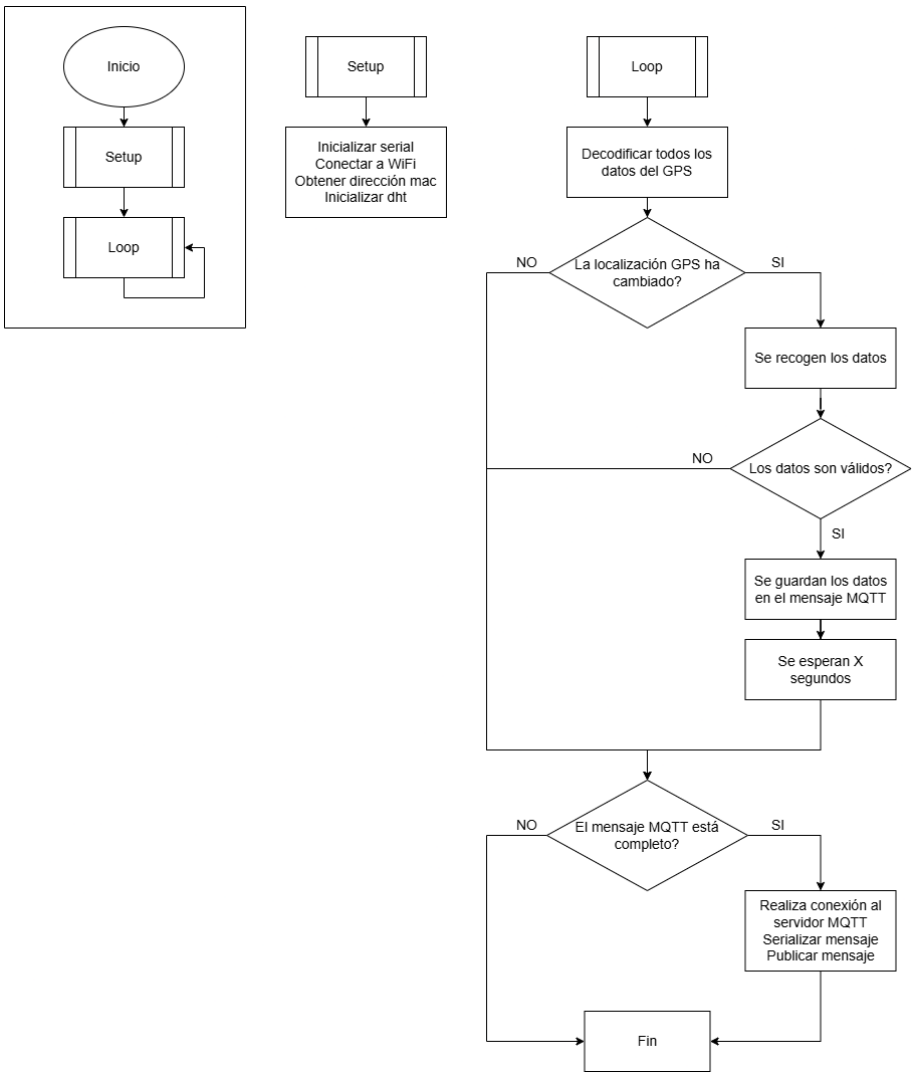

Figura 5.11: Diagrama de flujo de la iteración 1

Como ya se explicó, el led interno puede indicarnos si está recibiendo corriente y si está recibiendo señal y trasmitiendo información por su pin de trasmisión, TX.

Si no se está seguro de si el componente GPS está enviando información por su pin de trasmisión, se puede prescindir de la librería anteriormente mencionada e imprimir directamente la información que reciba el NodeMCU. Sin embargo, en este caso se mostrarán datos en formato NMEA, mensajes que por si mismos pueden ser difíciles de interpretar, pero que puede indicarnos qué está ocurriendo con nuestros componente [25]. Este detalle ayudó a comprender a lo largo del desarrollo que el componente GPS tiene una sensibilidad un tanto alta, con lo que en ocasiones puede tardar hasta 15 minutos empezar a trasmitir datos, aunque pueda parecer que no está funcionando correctamente.

Como ya se ha comentado, la librería MQTT de Adafruit tiene una limitación en el tamaño de los paquetes que se pueden enviar mediante dicho protocolo. Esto es algo que

no se menciona en la documentación de la librería y que también dió problemas a lo largo del desarrollo, ya que comprobando los mensajes enviados al servidor de Adafruit se pudo comprobar que los JSONs llegaban incompletos. Sin embargo, se pudo descubrir revisando el código interno de la librería, el cual, no solo se modificó para poder enviar mensajes MQTT más grandes, sino también para consultar el número de bytes reales que se enviaban.

5.2. Implementación de la segunda iteración

Continuando con el desarrollo obtenido en la primera iteración se plantea la implementación del uso de un tipo de conexión a internet distinto, de tal manera que el dispositivo sea independiente de la existencia o no de una red Wi-Fi disponible, cumpliendo así con el requisito **RF09**. Para ello se ha dispuesto de un componente que permite la conexión a internet mediante la red móvil, de tal manera que el dispositivo solo requiera de una tarjeta de telefonía móvil habilitada por un proveedor y cobertura.

Si bien la intención en esta iteración es reutilizar la mayor cantidad de código desarrollada en la anterior iteración, por compatibilidad con el componente SIM800L se ha tenido que desarrollar de nuevo el código correspondiente a la conexión a internet y envío de los datos mediante protocolo MQTT. Con ello se ha prescindido de las librerías: **Adafruit MQTT Library** ya que no se pudo compatibilizar con el SIM800L, y **ESP8266Wi-Fi Library** ya que solo era necesaria para la conexión a redes Wi-Fi. Por otro lado, para implementar la funcionalidad comentada se ha utilizado las siguientes librerías:

- **TinyGSM** en [26]: proporciona una manera comprensible y orientada a objetos de acceder a las distintas funcionalidades de los módulos GSM/GPRS, a las que normalmente se acceden mediante comandos AT. Principalmente, se ha utilizado para realizar la conexión a la red móvil y con ello a internet, pudiendo comprobar cómodamente y en cualquier momento el estado de dicha conexión.
- **PubSubClient** en [27]: proporciona soporte para realizar conexiones a distintos servidores MQTT y enviar datos mediante dicho protocolo. Se utilizó como sustituto de la librería **Adafruit MQTT Library**, ya que si bien funcionan de manera similar, esta última fue creada para conexión a internet vía Wi-Fi o Ethernet, por lo que no se consiguió hacer funciona con la red GPRS. Esta librería también tiene una limitación de envío de datos MQTT de 256 bytes, por lo que de nuevo se ha modificado el código interno de la librería para aumentar este limite. También se han modificado funciones internas de la librería para poder obtener el tamaño de paquete que se envía como resultado de la función `publish()`, lo que permite tomar mediciones de consumos de datos durante las pruebas. Los archivos modificados de esta librería se pueden encontrar en una carpeta del repositorio del proyecto en [17].

Estas modificaciones en la librerías utilizadas no solo implicó cambiar las funciones de comunicación con el servidor MQTT, sino que se tuvo que optar por un nuevo broker

MQTT, ya que la librería **PubSubClient** no es la más adecuada para conectar con los servidores Adafruit. Por ello, se optó por broker MQTT de Mosquitto [28], el cual dispone de un servidor de pruebas disponible de manera gratuita [29], aunque se podrían plantear otras opciones tanto gratuitas como de pago distintas a Mosquitto.

Como mejora adicional de esta iteración se planteó una modificación del código que en principio podría disminuir el consumo eléctrico general del dispositivo, aumentando así la autonomía del mismo. Para ello se exploraron las distintas opciones de puesta en reposo o *sleep modes* de los componentes del sistema. En muchos casos los componentes se desarrollan con una opción que les permite ponerse en un estado de espera en el que su actividad de reduce permitiendo así disminuir el consumo eléctrico. En el caso de nuestro sistema los únicos componentes que tiene un consumo eléctrico suficientemente alto como para considerar su puesta en reposo son tres: el módulo GPS, el módulo GPRS y el NodeMCU (placa de desarrollo del dispositivo). Analizando estos tres componentes y su contribución en el funcionamiento del sistema se pudo concluir que tanto el módulo GPS como el módulo GPRS no se pueden poner en estado de reposo en ningún momento de su funcionamiento.

Por un lado, el módulo GPS tiene el defecto de tardar entre 5 y 15 minutos en captar inicialmente la señal de suficientes satélites para poder empezar a enviar datos de posición. Por lo tanto, si se privase de corriente en cualquier momento a este componente o se activase algún modo de ahorro de energía podría perder su señal, lo que podría causar un retraso en la recolección de datos del dispositivo e incluso en el envío de los mismos. Por otro lado, el módulo GPRS tiene un consumo de energía típico más bajo (por debajo de los 20mA), aunque en los momentos de reconexión y envío de datos su consumo se puede disparar durante unos pocos segundos. El puerto serial del SIM800L se desactiva mientras cualquiera de los dos modos de bajo consumo del módulo estén activados, por lo que no se puede realizar reconexiones o envíos de datos de ningún tipo con estos modos de reposo activados. Se puede consultar más información sobre estos modos en el DataSheet del SIM800L que se aporta en [10]. Tampoco interesa activar este modo en los momentos de recolección de datos, ya que si se pierde la conexión, en la fase de envío se deberá reconectar a la red GPRS y al servidor MQTT. Esta tarea en condiciones normales puede tardar unos pocos segundos, pero en algunos casos puede llevar más tiempo debido a repetidos intentos fallidos de conexión, lo que implicaría unos picos de consumo que durarían más tiempo de lo habitual.

Por su parte el NodeMCU también dispone de varios modos de reposo o "soluciones de bajo consumo" [30]. En este caso las más utilizadas son tres y algunas engloban a otras:

- **Modem-sleep**: esta configuración lo único que hace es desactivar la funcionalidad de conexión a redes Wi-Fi con la que cuenta el NodeMCU. Si bien esta configuración no afecta al resto de funciones del microcontrolador, si que reduce levemente el consumo típico de la placa.
- **Light-sleep**: en este caso no solo se desactiva la funcionalidad de conexión a redes Wi-Fi, sino que también se desactiva el reloj interno del sistema del NodeMCU y

el procesador interno se queda en un estado de suspensión. Esto resulta en una reducción del consumo mucho mayor que en el Modem-sleep mode.

- **Deep-sleep**: este último modo es más brusco, ya que en vez de desactivar funcionalidades, desactiva el procesador interno por completo, con lo que el programa y la memoria guardada se resetean. Como consecuencia este modo es el que más reduce el consumo del NodeMCU.

Estos modos de reposo se pueden activar durante un periodo de tiempo a partir del cual el NodeMCU vuelve a su modo habitual de trabajo, por lo que se podría considerar un sustitutivo de la instrucción `delay()`. Sin embargo, en el caso del modo Deep-sleep, el NodeMCU resetea su estado, comenzando de nuevo el programa y borrando los datos guardados en memoria, lo cual hace que los datos de posición recolectados se pierda. Esto hace que el modo Deep-sleep no sea adecuado para el sistema que se está desarrollando.

La implementación del modo Light-sleep habitualmente se realiza utilizando un pin como entrada para la señal externa que despertará al dispositivo, ya que en este modo el procesador se encuentra en suspensión. Si bien se encontraron ejemplos de código no oficiales que supuestamente despertaban al dispositivo tras un tiempo utilizando la función `delay()`, al probarlos se observó que el NodeMCU hacía la espera de tiempo programada, pero el consumo energético no disminuía. Por lo tanto, se descartó el modo Light-sleep, ya que no parece funcionar sin utilizar un método externo al dispositivo para salir de dicho modo.

Por último, para activar el modo Modem-sleep solo se necesita llamar a una función interna del ESP8266, y dado que en esta iteración no se utiliza la conexión a redes Wi-Fi, no tiene sentido utilizar este modo, ya que por defecto el módem ya parece estar apagado. Es por esto que finalmente no se vió adecuado utilizar ninguno de los modos de reposo disponibles.

El flujo de código final de esta iteración es muy similar al representado en la anterior iteración en la Figura 5.11, aunque hay algunas diferencias que destacar. En primer lugar durante la función de `setup()` ya no se conecta a una red Wi-Fi, ya que ahora se utiliza el módulo GPRS para conectarse a internet. Además, al estar utilizando un módulo GPRS, ya no se puede obtener la dirección MAC, ya que esta dirección es exclusiva de dispositivos red, pero un módulo GPRS es una tecnología de red móvil. En su lugar se puede obtener una dirección IMEI, código numérico de 15 dígitos normalmente único que sirve para identificar un dispositivo móvil. Otro cambio con respecto a la anterior iteración se encuentra en el momento del envío de datos, el cual ahora se realiza mediante el componente GPRS. Para poder asegurar el envío se debe comprobar que el componente SIM800L está conectado a la red, para ello se debe comprobar por orden: si se ha registrado en la red móvil y ha encontrado un operador válido (`isNetworkConnected()`), si se ha conectado al servicio GPRS (`isGprsConnected()`), y por último si la conexión al servidor MQTT está operativa. Si alguna de dichas conexiones no está habilitada, se realizará una reconexión de los servicios necesarios antes del envío de los datos. Esto permite mantener

los datos almacenados hasta tener una conexión estable para el envío de los mismos, cumpliendo así con el requisito **RF03**.

Dificultades encontradas

En esta iteración se añadió un nuevo componente, que como en la iteración anterior, se tuvo que soldar. Durante el desarrollo de esta iteración se tuvo que utilizar distintos componentes SIM800L, por problemas con la soldadura y porque uno de ellos resultó defectuoso e incapaz de conectarse a la red móvil, a pesar de que todo parecía correctamente soldado. Todo esto se puede comprobar en gran parte atendiendo al led acoplado al componente, el cual debería parpadear cada segundo si recibe corriente, y tras un par de minutos, si tiene correctamente insertada una tarjeta SIM y la antena está bien soldada empezará a parpadear cada 3 segundos. Sin embargo, la posición en la que debe introducirse la tarjeta SIM no es intuitiva, aunque venga indicada de manera muy sutil en la carcasa del módulo, por lo que es fácil introducirla erróneamente.

Para la comunicación con los componentes GPS y GPRS se utiliza el protocolo UART de comunicación serial. En la primera iteración se utilizó la librería SoftwareSerial para emular una interfaz UART y en esta iteración originalmente se pensó utilizar la misma librería para emular dos interfaces de comunicación serial distintas. Sin embargo, esta opción resultó funcionar de manera inestable, ya que cuando se se habilitan dos SoftwareSerial distintos, solo uno de ellos puede recibir datos al mismo tiempo. Por ello se optó por mantener un SoftwareSerial para el componente GPRS y utilizar la interfaz UART hardware del propio NodeMCU para recibir los datos del componente GPS.

5.3. Implementación de la tercera iteración

Si bien se realizaron diversas pruebas sobre el Arduino Nano Every, no se pudo hacer funcionar este sin un “logic level shifter”, por lo que su uso se descartó en la fase de diseño de la tercera iteración. Continuando con la sustitución del NodeMCU por el Arduino Nano, se realizaron distintas pruebas que mostraron buena compatibilidad del dispositivo con los componentes GPS y DHT11 y las librerías que estos requieren. Sin embargo, se hallaron dificultades para realizar la comunicación entre el Arduino Nano y el componente SIM800L, que se detallan a continuación.

Mediante el uso de la librería SoftwareSerial se pudo realizar la comunicación por prtocolo UART entre SIM800L y Arudino Nano, observando cómo los comandos AT se enviaban correctamente al componente GPRS y este enviaban respuestas sin caracteres corruptos. Esto indicaba que los voltajes de trasmisión y recepción eran correctos en ambos sentidos, ya que de otra manera se obtendría los mismos resultados que para el Arduino Nano Every. Sin embargo, utilizando las librerías TinyGSM y PubSubClient de la misma manera que en la iteración anterior, se pudo observar que los mensajes MQTT nunca llegaban al servidor.

La librería TinyGSM cuenta con una opción de debug, que mediante el uso de la librería **StreamDebugger** [31], permite obtener retroalimentación por el monitor serial sobre los distintos comandos AT que se envían y las respuestas obtenidas. Con ello se pudo observar que los comandos se enviaban correctamente, aunque en algunos casos eran respondidos con mensajes de error o con una tardanza notable, siendo la mayoría de respuestas de error remediadas mediante el reenvío del mismo comando sucesivas veces. Sin embargo, el mayor punto de fallo se encontraba en el envío del comando `AT+CIPSEND`, comando que se utiliza para enviar un paquete de datos mediante TCP o UDP según el manual de comandos AT [13]. Las librerías utilizadas esperan como respuesta el carácter "`>`", que indica que se puede enviar el contenido del paquete o mensaje. Sin embargo, la respuesta esperada no parecía llegar a tiempo y el programa cerraba la conexión TCP y volvía a intentar todos los comandos continuamente.

Se realizaron múltiples pruebas con un código en el que se trataba de enviar mensajes por TCP a un servidor que escuchaba en un puerto específico, para lo cual se tuvo que abrir dicho puerto a la red externa. Estas pruebas se realizaron prescindiendo de la librería PubSubClient, dado que utilizarla para este propósito era inviable. Con ello, se pudo comprobar que efectivamente el carácter "`>`" de respuesta era enviado por el SIM800L, aunque a menudo con un retraso en el tiempo bastante grande, lo que en muchos casos hacía que el programa diese por fallido el envío y volviera a intentarlo una y otra vez. Esto forzó a prescindir completamente de la librería PubSubClient para la implementación de esta iteración. Sin embargo, no se encontraron otras librerías de mensajería MQTT que fueran compatibles con Arduino Nano, el SIM800L y la librería TinyGSM, ya que en principio PubSubClient es la única que mantiene compatibilidad con esta última.

Solución obtenida

Tomando como punto de partida el código desarrollado en la segunda iteración, se realizaron modificaciones para enviar los datos recolectados por el componente GPS a un servidor TCP en lugar de al broker MQTT utilizado hasta ahora. Como ya se comentó, la compilación del código de la segunda iteración, con el *toolkit* específico de Arduino Nano, ocupa un 24 % por encima del espacio de memoria de programa. Sin embargo, al prescindir de la librería PubSubClient este pasó de ocupar un 24 % por encima a ocupar un 8 % por encima del espacio de memoria de programa, por lo tanto, se requirió de más optimizaciones para reducir aún más el espacio de memoria de programa ocupado. Analizando las librerías utilizadas, se pudo concluir que la librería ArduinoJson podía ser sustituida por una implementación manual que se encargase de crear únicamente la estructura del JSON específica para esta solución, en lugar de permitir la serialización y deserialización de cualquier estructura JSON. Esta implementación se realizó mediante la función `snprintf()` que permite formatear y guardar texto en un array de caracteres de forma segura, controlando su tamaño máximo para evitar desbordamientos, de tal manera que se evite el uso de la clase String.

Como resultado tras prescindir de las librerías **ArduinoJson** y **PubSubClient**, y otras optimizaciones realizadas sobre el código, se pudo reducir el espacio de memoria de

programa ocupado de 124 % a 96 %, siendo posible una reducción aún mayor al prescindir de los mensajes de retroalimentación. Además, se desarrollo un script de python que permitiese recibir los mensajes TCP enviados por el dispositivo, encapsulando y redirigiendo el mensaje al broker MQTT deseado. Este script utiliza la librería `paho-mqtt` para reenviar el mensaje por MQTT al broker de Mosquitto, abriendo primero un puerto de escucha TCP, para así poder recibir los mensajes. El código desarrollado en esta iteración se puede encontrar en el repositorio del proyecto [17].

Si bien con la solución planteada se pudieron recibir mensajes al servidor MQTT, se pudo observar que los retrasos afectaban bastante al buen funcionamiento del sistema. Finalmente, debido a los retrasos en los envíos, la complejidad añadida a la nueva solución que ahora requería de un servidor adicional para el reenvío de mensajes, y otros factores negativos detallados en el diseño de la implementación, se decidió descartar la sustitución del NodeMCU por el Arduino Nano como mejora del dispositivo. Esto nos lleva a considerar el dispositivo obtenido en la segunda iteración como la mejor opción que cumple con los requisitos propuestos inicialmente.

5.4. Resumen

Este capítulo ha descrito en detalle la implementación software del dispositivo IoT, explicando su desarrollo a través de las sucesivas iteraciones del proyecto. Se ha abordado la programación de la primera iteración, centrada en el esqueleto principal del funcionamiento del dispositivo, incluyendo el uso de librerías clave como TinyGPS++ y Adafruit MQTT necesarias para la interpretación de los datos GPS y la conexión a un servidor MQTT de Adafruit, respectivamente. Posteriormente, se detallaron los ajustes necesarios en el software para la segunda iteración, que incorporó la conectividad GPRS mediante el módulo SIM800L, lo que obligó a utilizar distintas librerías para manejar el nuevo componente y realizar una conexión a un nuevo servidor MQTT, esta vez de Mosquitto. Finalmente, se explicaron las múltiples adaptaciones de software necesarias para la tercera iteración, que involucró el cambio de placa por la de Arduino Nano, lo que concluyó con la elección del dispositivo desarrollado en la segunda iteración como solución final de este proyecto. Con esto todo la parte relativa al desarrollo del dispositivo queda cubierta, siendo necesaria una última fase de pruebas descritas en el Capítulo 6, en dónde se exponen distintas conclusiones sobre los casos probados sobre las distintas iteraciones.

6: Pruebas realizadas

Como parte del desarrollo del dispositivo se han diseñado una serie de validaciones, casos de prueba y métodos de comprobación, para observar el correcto funcionamiento del sistema creado. Para estas pruebas se han utilizado dos brokers MQTT distintos: para la primera iteración se ha utilizado el servicio MQTT gratuito de Adafruit, el cual te permite consultar los mensajes MQTT recibidos en tiempo real a través de su web [32]; y para las siguientes iteraciones Eclipse Mosquitto, que tiene varios *feed* o flujos de datos para realizar pruebas y se puede instalar fácilmente con Docker [28]. Estas dos opciones permiten consultar los mensajes MQTT que se envíen con el dispositivo de manera sencilla y rápida.

Estas pruebas no solo tienen el objetivo de comprobar el buen funcionamiento del dispositivo, sino también poder analizar los requisitos de consumo eléctrico y de ancho de banda necesarios. De esta manera se podría decidir que tipo de baterías y contrato de telefonía móvil serían más adecuados para su puesta en producción.

6.1. Verificación de mensajes MQTT

Con la intención de comprobar y validar que los mensajes MQTT enviados por el dispositivo son accesibles y su contenido correcto, se ha diseñado un código en python que se suscribe al *feed* correspondiente para recibir nuevos datos en tiempo real. Dependiendo de la iteración se conectará a un broker u otro como ya se ha explicado, aunque en el caso de Adafruit se pueden comprobar los mensajes en su propia web, y en el caso de Mosquitto se puede comprobar instalando la propia api y ejecutando un comando para consultar el *feed* de pruebas disponible de manera gratuita [29].

Dado que se utilizan dos brokers distintos, se han creado dos scripts de python distintos con ligeras diferencias, ya que las comprobaciones de los mensajes son las mismas en ambos casos. En ambos scripts se utiliza la misma librería para conectar a servidores MQTT, `paho-mqtt`, el cual además de proporcionar una implementación del protocolo MQTT para python, también nos permite conectarnos a prácticamente cualquier broker MQTT mediante un cliente MQTT. La única diferencia es que en un script se debe proporcionar

el usuario y clave de Adafruit y en el caso de Mosquitto no es necesario, ya que se utiliza un *feed* para pruebas público. Para comprobar la veracidad del hash SHA-256 se utiliza la librería `hashlib` de python, que contiene diversos algoritmos de hash. Ambos scripts se pueden encontrar en una de las carpetas del repositorio del proyecto [17].

Este script de python se conecta en tiempo real al broker correspondiente y comprueba que el contenido del mensaje cumpla lo siguiente:

- El JSON está completo y tiene una estructura de JSON correcta.
- El hash SHA-256 se ha generado correctamente a partir del contenido del JSON con el parámetro de hash vacío.
- Las instancias de datos guardadas tienen timestamps consecutivos y correctos con respecto a la fecha actual.
- Las mediciones de cada instancia de datos no están vacías.

Esta verificación de los mensajes MQTT es fundamental para asegurar la validez de los mensajes MQTT, y a lo largo del desarrollo fue especialmente útil para detectar algunos fallos y problemas.

6.2. Casos de prueba

Dado que los tiempos entre envíos y recogidas de datos son configurables, cumpliendo con el requisito **RF05**, y tienen un gran impacto en el consumo eléctrico y de datos del dispositivo, se han diseñado tres casos de prueba distintos cuyo interés depende de la situación del dispositivo.

- **Caso 1**: se recogen datos del componente GPS cada 10 segundos y se envía el conjunto de datos cada minuto, con lo que se envían mensajes que contienen 6 instancias distintas de datos. Este caso tiene unos tiempos muy cortos de recogidas, con lo que puede ser especialmente interesante para los casos en los que el dispositivo esté en movimiento.
- **Caso 2**: se recogen datos del componente GPS cada minuto y se envía el conjunto de datos cada 10 minutos, con lo que se envían mensajes que contienen 10 instancias distintas de datos. En este caso se plantea unos tiempos más normales entre recogidas, lo que lo hace ideal para un control habitual de los residuos.
- **Caso 3**: se recogen datos del componente GPS cada 10 minutos y se envía el conjunto de datos cada hora, con lo que se envían mensajes que contienen 6 instancias distintas de datos. Los tiempos de entre recogidas en este caso son mucho más largos, con lo que está más pensada para los casos en los que el dispositivo se encuentre parado por varias horas.

Como se ha explicado cada uno de estos casos presenta una variabilidad que se ajusta mejor a distintos supuestos de la trazabilidad de residuos, permitiendo elegir así la configuración más adecuada. Cabe destacar que a nivel de implementación estas configuraciones solo requieren cambiar el valor de dos variables que marcan el tiempo entre recogidas de datos y el número de instancias necesarias para poder enviar el mensaje MQTT, por lo que se podría cambiar entre casos en tiempo de ejecución.

6.3. Resultados obtenidos

Para cada uno de los casos anteriormente planteados se han realizado una serie de pruebas en un ambiente controlado y con una fuente de alimentación en la que se muestra el voltaje del circuito y los amperios de consumo instantáneo. De la misma manera, gracias a las modificaciones en las librerías de conexión MQTT utilizadas, se puede consultar el tamaño de los paquetes enviados, de tal manera que se pueda abstraer de ello el consumo de datos del dispositivo.

Para las mediciones de consumo de corriente se han tomado distintas mediciones en el tiempo de recogida de datos, que en cada caso difiere en longitud, y por otro lado en el tiempo de envío de datos, que por motivos de posibles reconexiones se ha calculado que en todos los casos ronda entre los 5 y 15 segundos. Esto se ha decidido realizar así, porque el consumo en tiempo de recogida de datos es mucho menor a los picos de consumo que se obtienen en los momentos de envío de datos. Por lo tanto, se ha decidido dividir los resultados primero por caso y fase (recogida y envío), y posteriormente dar un resumen de mediciones para cada caso, con la intención de poder elegir un caso de uso dependiendo de la situación.

Durante la fase de recogida se tomaron distintas medidas cada poco tiempo, observando un consumo relativamente estable casi todo el tiempo, por lo que se realizó una media de las mediciones tomadas para obtener el consumo promedio. Sin embargo, en el caso del envío de datos es un poco más complicado, ya que se observó que al principio de esta fase se mantenía estable en una medida durante unos 7 segundos y después tenía un pico de consumo instantáneo mucho mayor durante 3 segundos. Con ello se decidió calcular una media ponderada en función del tiempo que duraban los distintos rangos de medidas, para poder obtener una medida fiable del consumo promedio. La fórmula utilizada se muestra a continuación:

$$C_{promedio} = \frac{(\frac{\sum_{i=0}^{n} C_{1,i}}{n} \cdot t_1) + (\frac{\sum_{i=0}^{n} C_{2,i}}{n} \cdot t_2)}{(t_1 + t_2)},$$

donde $C_{1,i}$ son las mediciones tomadas en el primer tramo de tiempo t_1 y, los $C_{2,i}$ son las mediciones tomadas en el segundo tramo de tiempo t_2. De la misma manera se utilizó una media ponderada para calcular el consumo promedio en el resumen por caso, utilizando los consumos promedios de la fase de envío y recogida:

$$C_{promedio} = \frac{(C_{recogida} \cdot t_r) + (C_{envio} \cdot t_e)}{(t_r + t_e)}.$$

En este caso tenemos que $C_{recogida}$ es el consumo promedio de la fase de recogida que tiene una duración t_r, y C_{envio} es el consumo promedio de la fase de envío que tiene una duración t_e.

Pruebas Segunda iteración

En la Tabla 6.9 se puede observar un resumen del consumo de datos, teniendo en cuenta un promedio de tamaño de paquete para cada caso. Como es obvio el tercer caso es el que menor consumo de datos tendría a lo largo del tiempo, ya que es el que realiza menos envíos a lo largo del tiempo. Se puede observar que siendo el Caso 1 el más extremo, y manteniendo el dispositivo activo las 24 horas del día, un contrato con el proveedor de 50MB al mes sería más que suficiente para cubrir las necesidades de este, lo cual es mucho menos que las ofertas que se suelen publicitar hoy en día. Con esto se podría concluir que el consumo de datos no parece ser un problema.

Caso de prueba	Tamaño por envío	Bytes/hora
Caso 1	910	54600
Caso 2	1404	8424
Caso 3	909	909

Tabla 6.9: Tabla de consumo de datos para los casos de prueba

En la Tabla 6.10 se muestra el consumo eléctrico para cada caso y fase de un ciclo, calculado según las explicaciones anteriores. Nótese que para el promedio de consumo en fases de recogida se revisó el consumo instantáneo a lo largo de varios ciclos en cada caso tomando del orden de 5 mediciones por ciclo, aunque se revisó de manera constante la fuente de alimentación para anotar cómo de estable era el consumo instantáneo. Si bien en los Casos 1 y 2, sí se llegó a tomar anotaciones de por lo menos 5 ciclos, en el Caso 3 solo se anotaron medidas de 3 ciclos consecutivos, ya que más ciclos implicaría probar durante muchas horas el sistema. Se pudo observar en los Casos 2 y 3 una estabilidad mucho mayor en el consumo instantáneo de la fase de recogida de datos que en el Caso 1. Esto puede tener sentido, ya que en el Caso 1 las esperas duraban 10 segundos, por lo que el procesador estaba mucho más tiempo trabajando que en el resto de casos probados. Por otro lado, las mediciones en las fases de envío fueron muy inestables dando en ocasiones picos de 350mA y en otros ciclos llegando solo a los 150mA de pico, lo cual explica las diferencias entre los casos de prueba, cuando deberían tener mediciones similares para las fases de envío. Cabe destacar que este suceso es algo normal, ya que el consumo por parte del SIM800L en momento de envío de datos depende mucho del tráfico de red de dicho momento, por lo que es un tanto impredecible.

Caso	Fase	Duración (seg)	Corriente Pico (mA)	Consumo medio del ciclo (mA)
1	Recogida	60	100	78,60
2	Recogida	600	90	76,00
3	Recogida	3600	80	73,33
1	Envío	10	350	146,38
2	Envío	10	330	154,50
3	Envío	10	350	160,99

Tabla 6.10: Tabla de consumo eléctrico para los casos de prueba y sus fases

En la Tabla 6.11 se recogen los datos para cada caso, de donde se pueden sacar conclusiones más interesantes sobre el consumo eléctrico general. Como era de esperar, al realizar menos envío, el consumo promedio del Caso 3 es el menor de todos, aunque no mucho menor al valor del consumo del Caso 2. También cabe destacar que en todos los casos el consumo al día entra dentro de lo que podría soportar una pila alcalina AA, si no tenemos en cuenta las necesidad de Voltaje, aunque teniendo en cuenta las necesidad de voltaje necesitaríamos tres pilas alcalinas AA colocadas en serie. Pero la cifra de consumo nos indica que cumplir con el requisito **RNF01** sería factible con la fuente de alimentación adecuada. La configuración del dispositivo para el Caso 3 junto con la elección de los componentes ya mencionados nos permite satisfacer también el requisito **RNF04**.

Caso	Duración del ciclo (seg)	Consumo medio del ciclo (mA)	Consumo al día (mA/h)
1	70	88,28	2.118,72
2	610	77,29	1.854,96
3	3610	73,57	1.765,68

Tabla 6.11: Tabla de consumo eléctrico para los casos de prueba

Pruebas Tercera iteración

Durante el desarrollo de la tercera iteración se realizaron múltiples pruebas para determinar el grado de fiabilidad del sistema, ya que como se ha comentado previamente este fallaba a menudo a la hora de realizar el envío de los datos. Esto no fue necesario en iteraciones anteriores, ya que en las pruebas realizadas se pudo observar envíos exitosos en la gran mayoría de los intentos de envío. La estructura del código de la solución permite repetir los intentos de envío hasta que alguno de ellos resulte exitoso. Se fijó el Caso 1 como prueba para observar la fiabilidad del sistema, ya que en los tres casos el tiempo entre intentos de envío es el mismo. Con ello se puso en marcha el sistema durante un tiempo prolongado con lo que se pudo comprobar que de media, 1 de cada 15 intentos de envío era exitoso.

No se realizaron mediciones de consumo eléctrico exhaustivas para esta iteración, ya que el Arduino Nano requiere de 7V de corriente mínima para su correcto funcionamiento y esto requirió de un conversor de corriente DC-DC para bajarla a 4V para el SIM800L, y el uso de este aumentó el consumo de corriente general. No se realizaron más pruebas

para obtener las métricas anteriormente expuestas, ya que el dispositivo desarrollado en esta iteración se consideró peor que el obtenido en la segunda iteración.

6.4. Resumen

En este capítulo se ha documentado las pruebas realizadas para validar el funcionamiento integral del dispositivo IoT desarrollado. En primer lugar se han detallado los métodos de verificación implementados, que han permitido comprobar la integridad de los mensajes MQTT enviados por el dispositivo. Se han presentado los casos de prueba, sobre los que se han tomado métricas específicas que nos permiten analizar el rendimiento de los distintos componentes utilizados, como medidas de consumo de datos enviados por la red y el consumo eléctrico general del dispositivo. Esto permitió analizar aspectos cruciales como la autonomía energética del dispositivo. También se detalla las comprobaciones realizadas sobre la frecuencia de fallo en los envíos de mensajes, que en el caso del dispositivo desarrollado en la tercera iteración fue bastante alta. Los resultados de estas pruebas han permitido verificar la correcta integración de los componentes y la fiabilidad del sistema en su conjunto, lo que nos lleva a las conclusiones y posibles líneas futuras de desarrollo detalladas en el Capítulo 7.

7: Conclusiones y Líneas de trabajo futuras

En este trabajo se ha desarrollado un sistema adecuado para la trazabilidad de residuos mediante el uso de tecnologías IoT y se ha documentado todo el proceso e información necesaria para la fabricación del mismo. Además, como parte del trabajo realizado se ha aportado esta solución para una contribución en las Jornadas Sarteco del año 2025 [5], en el que se presenta un sistema para la trazabilidad de residuos basado en la solución IoT propuesta en este trabajo y un servidor blockchain para la recepción de los datos.

El trabajo desarrollado en el marco de este proyecto ha sido el siguiente:

- Se ha desarrollado una solución fiable y de bajo coste que cumple con los requisitos propuestos para la trazabilidad de residuos. Esta propuesta garantiza la capacidad de monitorizar la ubicación y las condiciones ambientales de los residuos en tiempo real.
- Se ha logrado reducir notoriamente el precio general de la propuesta mediante el uso de componentes electrónicos de bajo coste. Aunque la solución se basa en el concepto de dispositivos "suicida" o desechables, la minimización de los costes de hardware hace que esta aproximación sea viable y sostenible para despliegues a gran escala en el contexto de la trazabilidad.
- Se han explorado diversas opciones alternativas a lo largo de las iteraciones de diseño con el objetivo de reducir aún más el tamaño y el precio final del dispositivo. Esta investigación de distintas configuraciones de hardware y microcontroladores sienta las bases para futuras investigaciones, permitiendo una exploración mucho más acotada hacia una simplificación aún mayor del dispositivo.
- Se ha comprobado que el uso de componentes GPRS es la mejor opción para un problema de trazabilidad de estas características, ya que la conexión a redes Wi-Fi limitaría significativamente el proceso de envío de datos.

- Se ha implementado la obtención de datos adicionales, como la temperatura y la humedad ambiental, junto con otros datos de posición GPS, como la velocidad y la altitud. Estos parámetros nos ofrecen una visión más completa de la situación del dispositivo y el entorno del residuo, lo que abre la posibilidad de implementar funcionalidades avanzadas para cambiar su funcionamiento antes cambios en su entorno.
- Se ha hecho una documentación completa y detallada, abarcando tanto el diseño de hardware como la implementación de software. Esta exhaustiva documentación permite que personas con los conocimientos tecnológicos adecuados puedan replicar la propuesta, comprender su funcionamiento en profundidad, y mejorar y adaptar la solución actual.

En resumen, los resultados obtenidos en este Trabajo Fin de Máster no solo confirman la viabilidad de la propuesta para la trazabilidad de residuos, sino que también demuestran su potencial como solución fiable y económicamente eficiente. La combinación de un diseño iterativo, la selección adecuada de componentes y una implementación software bien estudiada ha permitido desarrollar un prototipo funcional que cumple con los requisitos planteados, sentando una base para futuros desarrollos y para la aplicación de esta tecnología en contextos reales de gestión de residuos.

7.1. Trabajo futuro

Si bien la solución propuesta se ha podido poner en marcha en un ambiente controlado, se propone como trabajo futuro la fabricación de este dispositivo y su puesta en marcha en la situaciones de la problemática expuesta. De esta manera se podría realizar iteraciones de desarrollo adicionales en base a las conclusiones que se puedan obtener de pruebas más realistas, ya que una parte muy importante del desarrollo de los sistemas IoT es la puesta en marcha de los mismos en ambientes realistas. Y, aunque en este trabajo se hayan explorado ya varias sustituciones de componentes del dispositivo, se podría explorar nuevas opciones a partir de los resultados obtenidos con pruebas más realistas del dispositivo. Existen muchas opciones de placas de desarrollo que no se han podido explorar en profundidad durante este proyecto y que podrían ser adecuadas, incluso si ello implicase la modificación del software también desarrollado. También, uno de los puntos débiles del dispositivo es la precisión y sensibilidad del componente GPS, que podría tratar de mejorarse con antenas más potentes, siempre y cuando esto no afecta demasiado el tamaño y precio de la solución.

Finalmente, se propone como trabajo futuro la continuidad del desarrollo aquí realizado, junto con la solución de servidor aportada en [5]. La mejora de ambas partes de la misma solución podría llevar a un sistema de trazabilidad aún más robusto y fiable, que incluso podría ponerse en marcha de manera oficial si se consiguiera despertar el interés de las empresas encargadas de la logística de residuos.

Apéndices

Apéndice A

Documentación del programador

En este apéndice se detallan algunos conceptos técnicos sobre la estructura y montaje del proyecto para que una persona con nociones de programación y un mínimo de electrónica pueda modificar y probar el dispositivo presentado en este trabajo. En este sentido se asume que el usuario promedio puede no estar cualificado o interesado en modificar el proyecto presentado, pero sí utilizarlo y probarlo, por ello se plantea una segunda guía en el Apéndice B, en la que se detalla cómo instalar el software necesario y utilizar el dispositivo. Cabe destacar que en esta guía se asume el uso de ArduinoIDE para la programación.

A.1. Estructura del proyecto

Como ya se ha comentado, el proyecto está alojado en una repositorio de GitHub [17], en el que se pueden encontrar el código para el dispositivo, modificaciones de librerías necesarias y código de prueba. En la Figura A.1 se puede observar la estructura de las distintas carpetas y ficheros del repositorio.

Como se puede observar tenemos en primer lugar los códigos o *sketches* del dispositivo en las direcciones `IoT_System_Src/iot_trace/iot_trace.ino` y `IoT_System_Src/iot_trace_TCP_Nano/iot_trace_TCP_Nano.ino`, el primero correspondiente a la segunda iteración y el siguiente a la tercera iteración. Si bien la redundancia en el nombre de la carpeta y el archivo parece tediosa, es necesaria para poder acceder a este desde ArduinoIDE. Estos *sketches* contienen individualmente todo el código necesario para el funcionamiento del dispositivo, y por lo tanto, es la parte más importante si se quiere modificar el mismo. Además, estos dos *sketches* tienen un buen número de dependencias con librerías que pueden ser instaladas de manera sencilla desde el propio IDE, ya que tiene su propio sistema de manejo de librerías. Sin embargo, hay que destacar que una de las librerías utilizadas ha sido modificada para el correcto envío de los mensajes por protocolo MQTT. Los archivos modificados de dicha librería se pueden encontrar en la carpeta `Libraries`, dependiendo de si se esta trabajando con la implementación de

```
iot-trace-system
├── IoT_System_Src
│   ├── iot_trace
│   │   └── iot_trace.ino
│   └── iot_trace_TCP_Nano
│       └── iot_trace_TCP_Nano.ino
├── Libraries
│   ├── PubSubClient_ModifiedFiles
│   │   ├── PubSubClient.cpp
│   │   └── PubSubClient.h
│   └── Adafruit_MQTT_ModifiedFiles
│       ├── Adafruit_MQTT.cpp
│       └── Adafruit_MQTT.h
├── PythonScripts
│   ├── subscriberAdafruit.py
│   ├── subscriberMosquitto.py
│   └── tcp_to_mqtt_server.py
└── README.md
```

Figura A.1: Estructura de ficheros del repositorio

Adafruit o la de PubSubClient, se aportan los distintos archivos a reemplazar en la librería correspondiente.

Para poder comprobar el correcto funcionamiento del dispositivo se aportan unos scripts de python en la carpeta `PythonScripts`, que permiten suscribirse al *feed* de un servidor MQTT y validar los mensajes que se publican en este. De nuevo se distingue entre dos implementaciones, una utilizando Adafruit como broker MQTT y otra utilizando un *feed* gratuito de pruebas del broker Mosquitto. Estos scripts son especialmente interesante en el caso de usar Mosquitto como broker, ya que la instalación y puesta en marcha del propio cliente de Mosquitto es algo más complicada que el uso del script de python proporcionado en este trabajo.

A.2. Detalles del montaje del dispositivo

Si se quiere realizar el montaje actual del dispositivo o modificar el mismo se recomienda primero revisar los esquemas creados con Fritzing [9] que se han ido mostrando a lo largo de este documento, siendo el último realizado el observable en la Figura 4.7. A mayores, sería recomendable realizar cualquier modificación primero sobre un esquema y revisar correctamente los voltajes y amperajes necesarios en cada caso, ya que un error de conexiones podría dañar los componentes utilizados.

A la hora de realizar el montaje en si mismo se debe prescindir de cualquier corriente eléctrica hasta el último momento para evitar daños en el dispositivo. Es aconsejable que

los cables sigan un código de colores acorde a su función, por ejemplo: cables rojos para corriente, cables negros para toma de tierra, cable verdes para recepción de datos serial, cable amarillo para trasmisión de datos serial y cable azul para trasmisión de datos digital. Esto ayudará a identificar mejor los puntos críticos del sistema, como por ejemplo: la conexión de todas las tomas de tierra al mismo punto, la conexión de todos los pines de trasmisión con pines de recepción y viceversa, o la simple conexión de cables de trasmisión digital al pin incorrecto según el código.

Si durante el montaje o modificación del dispositivo se encuentra cualquier dificultad se deben revisar los consejos anteriores o los distintos apartados de dificultades encontradas expuestos en el Capítulo 5.

Apéndice B

Documentación del usuario

En este apéndice se explica una puesta en marcha del dispositivo, asumiendo el previo montaje del circuito requerido y la alimentación correcta del mismo, por lo que esta guía esta destinada a usuarios con conocimiento informáticos mínimos. Esta documentación pretende detallar la instalación de software necesario y como comprobar si el dispositivo funciona correctamente, junto con algunos de los fallos más comunes.

B.1. Instalación del software y carga del programa

Para poder cargar el programa en el dispositivo se necesita el programa ArduinoIDE, que no solo sirve como editor de código, sino que es la opción más sencilla para gestionar las librerías necesarias y cargar el programa. En la página oficial de Arduino se proporcionan los enlaces de descarga del programa para distintos sistema operativos [16]. Una vez instalado se deberá descargar el programa desarrollado directamente del repositorio, y dado que necesitaremos otros archivos del mismo repositorio es mejor descargar todo el proyecto. Para ello se puede clonar el proyecto con git o descargar directamente el zip completo del proyecto en la propia página del repositorio [17], aunque en este caso habrá que descomprimir el proyecto para poder realizar el resto de pasos.

Con ArduinoIDE y el repositorio descargados, se deberán realizar los siguientes pasos en orden para cargar el programa en el dispositivo ya montado:

1. **Descargar la librerías necesarias**: se deberán instalar las dependencias necesarias para el funcionamiento del dispositivo. Para ello deberemos abrir ArduinoIDE y acceder a la pestaña de gestión de librerías situada en la columna de funcionalidades de la izquierda, representada con un dibujo de libros apilados. En esta pestaña deberemos ir buscando las distintas librerías necesarias e instalar la versión específica de las mismas, que son: TinyGPSPlus versión 1.0.3, ArduinoJson versión 7.4.2, Crypto versión 0.4.0, DHT sensor library versión 1.4.6, TinyGSM versión 0.12.0 y PubSubClient versión 2.8.0.

2. **Modificar los archivos de PubSubClient**: dado que la librería PubSubClient fue modificada, se deberán sustituir sus archivos internos por los aportados en el resposito-rio. Para ello se debe acceder a los archivos internos de la librería, estos normalmente están ubicados en `C:/Users/nombredeusuario/Documents/Arduino/libraries/PubSubClient`. En este carpeta deberemos reemplazar los archivos `PubSubClient.cpp` y `PubSubClient.h` por los archivos con el mismo nombre ubicados en el repositorio descargado en la ruta `iot-trace-system/Libraries/PubSubClient_ModifiedFiles`.

3. **Instalar los *drivers* de la placa**: en este caso ArduinoIDE no viene por defecto con el *toolkit* de la placa NodeMCU, por lo que tendremos que instalar esta. Existen dos formas de hacerlo, pero la más sencilla consiste en acceder al menú *Preferences* dentro de la pestaña *File*. En ese menú se debe copiar dentro de la caja que dice "Additional Boards Manager URLs" el siguiente texto:
   `http://arduino.esp8266.com/stable/package_esp8266com_index.json`
   Una vez hecho eso se debe guardar las modificaciones de preferencias y reiniciar la aplicación para asegurarse de que funcione.

4. **Abrir el programa y comprobar que compila**: como paso anterior al envío del programa al dispositivo se puede comprobar si todos los pasos anteriores se han realizado correctamente compilando el programa. Para ello primero deberemos elegir la placa a utilizar en el desplegable situado en la parte de arriba de la interfaz, en este caso se debe buscar NodeMCU 1.0. Para compilar el programa se debe abrir el sketch en ArduinoIDE ubicado en el repositorio descargado en la ruta `iot-trace-system/IoT_System_Src/iot_trace/iot_trace.ino` y clicar en el botón con un "check" para empezar la compilación. Si todo está bien ArduinoIDE debería mostrar un mensaje de compilación realizada sin errores.

5. **Cargar el programa**: para cargar el programa en el dispositivo se debe conectar el dispositivo al ordenador con un cabe USB a micro-USB que permita la transferencia de datos. Se debe comprobar que ArduinoIDE reconoce el dispositivo, para ello se puede revisar el desplegable de la placa a utilizar, en el que debería aparecer la conexión por USB realizada. Para cargar el programa compruebe primero que ningún cable esté conectado a los pines TX o RX del NodeMCU y pulse el botón con el dibujo de una flecha señalando hacia la izquierda que se encuentra en la barra superior. De nuevo si el programa se ha cargado correctamente ArduinoIDE mostrará un mensajes de carga realizada correctamente.

Con el programa cargado al dispositivo se puede poner este en funcionamiento, pero para poder comprobar los mensajes enviados necesitaremos ejecutar el script de python proporcionado en el repositorio. Este requiere de la instalación de python3 y la librería `paho-mqtt` versión 2.1.0 o superior, fácilmente instalable con el comando `pip install paho-mqtt`. Para utilizar el script ubicado en el repositorio en la ruta `iot-trace-system/PythonScripts/subscriberMosquitto.py`, se puede ejecutar el comando `python3 subscriberMosquitto.py` desde la ubicación del archivo. Esto abrirá

una conexión al servidor de Mosquitto y una suscripción al *feed* utilizado en el programa del dispositivo, para poder consultar los mensajes enviados por el mismo.

B.2. Dispositivo en funcionamiento

A la hora de comprobar el correcto funcionamiento del dispositivo se debe tener en cuenta que esté correctamente alimentado como se indica en la fase de diseño de esta memoria, y que puede requerir de iniciar su funcionamiento al aire libre para poder captar señal satélite más eficientemente. Normalmente con colocar la antena del componente GPS en la repisa de una venta debería ser suficiente para que este empiece a parpadear pasados unos minutos, señal de que el componente GPS está recibiendo datos. Si el componente GPS no parpadea puede ser recomendable situarlo en una zona despejada y esperar entre 5 y 15 minutos, ya que al principio puede tardar más en recibir señal.

Si el dispositivo está correctamente alimentado el componente SIM800L debería parpadear cada 3 segundos aproximadamente tras un minuto recibiendo corriente. En los momentos de envío se podrá percibir como el componente SIM800L parpadea en intervalos de medio segundo aproximadamente, y tras pocos segundos debería aparecer el mensaje enviado en la terminal donde se esté ejecutando el script de python mencionado.

Bibliografía

[1] Tariq Ali, Muhammad Irfan, Abdullah Saeed Alwadie, and Adam Glowacz. Iot-based smart waste bin monitoring and municipal solid waste management system for smart cities. *Arabian Journal for Science and Engineering*, 45:10185–10198, 2020.

[2] Amira Henaien, Hadda Ben Elhadj, and Lamia Chaari Fourati. A sustainable smart iot-based solid waste management system. *Future Generation Computer Systems*, 157:587–602, 2024.

[3] Said Gulyamov. Intelligent waste management using iot, blockchain technology and data analytics. In *E3S Web of Conferences*, volume 501, page 01010. EDP Sciences, 2024.

[4] Supriya Pulparambil, Adil Al-Busaidi, Yasmine Al-Hatimy, and Ahmed Al-Farsi. Internet of things-based smart medical waste management system. *Telematics and Informatics Reports*, 15:100161, 2024.

[5] Javier Alonso-Núñez, Daniel López-Martínez, and Diego R. Llanos. Arquitectura segura para la trazabilidad basada en iot y blockchain. In *XXXV Jornadas de Paralelismo (JP2025)*, Sevilla, Spain, 2025. Grupo Trasgo.

[6] Make-It.ca. Nodemcu esp8266 specifications, overview and setting up. https://www.make-it.ca/nodemcu-details-specifications/, 2021. Consultado el 9 de julio de 2025.

[7] Cirkit Designer. How to use gy-neo6mv2: Examples, pinouts, and specs. https://docs.cirkitdesigner.com/component/gy-neo6mv2ww. Consultado el 9 de julio de 2025.

[8] AOSONG. Dht11 temperature and humidity sensor data sheet. https://www.mouser.com/datasheet/2/758/DHT11.pdf. Consultado el 9 de julio de 2025.

[9] Fritzing Team. Fritzing. https://fritzing.org/. Consultado el 9 de julio de 2025.

[10] SIMCom. Sim800l hardware design. https://www.makerhero.com/img/files/download/Datasheet_SIM800L.pdf. Consultado el 9 de julio de 2025.

[11] Michel Mouly and Marie-Bernadette Pautet. *The GSM system for mobile communications*. Telecom publishing, 1992.

[12] Bernhard Walke, Wolf Mende, and Georgios Hatziliadis. Cellpac: A packet radio protocol applied to the cellular gsm mobile radio network. In *[1991 Proceedings] 41st IEEE Vehicular Technology Conference*, pages 408–413. IEEE, 1991.

[13] SIMCom. Sim800 series at command manual. https://www.elecrow.com/download/SIM800%20Series_AT%20Command%20Manual_V1.09.pdf, 2015. Consultado el 9 de julio de 2025.

[14] Microchip Technology Inc. Atmega4809 data sheet. https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega4809-40-Pin-40002104B.pdf. Consultado el 9 de julio de 2025.

[15] Microchip Technology Inc. Atmega328p automotive microcontrollers data sheet. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. Consultado el 9 de julio de 2025.

[16] Arduino. Arduino ide versión 2.x. https://www.arduino.cc/en/software, 2025. Consultado el 9 de julio de 2025.

[17] danipequelangos. IoT Trace System | GitHub del TFM. https://github.com/danipequelangos/iot-trace-system, 2025. GitHub repository, Consultado el 9 de julio de 2025.

[18] Mikal Hart. Tinygpsplus | arduino documentation. https://docs.arduino.cc/libraries/tinygpsplus/, 2022. Consultado el 9 de julio de 2025.

[19] Arduino. Softwareserial library | arduino documentation. https://docs.arduino.cc/learn/built-in-libraries/software-serial/, 2022. Consultado el 9 de julio de 2025.

[20] Benoit Blanchon. Arduinojson | arduino documentation. https://docs.arduino.cc/libraries/arduinojson/, 2025. Consultado el 9 de julio de 2025.

[21] Rhys Weatherley. Crypto | arduino documentation. https://docs.arduino.cc/libraries/crypto/, 2022. Consultado el 9 de julio de 2025.

[22] Ivan Grokhotkov. Esp8266wifi library documentation. https://arduino-esp8266.readthedocs.io/en/latest/esp8266wifi/readme.html. Consultado el 9 de julio de 2025.

[23] Adafruit. Adafruit mqtt library | arduino documentation. https://docs.arduino.cc/libraries/adafruit-mqtt-library/, 2025. Consultado el 9 de julio de 2025.

[24] Adafruit. Dht sensor library | arduino documentation. https://docs.arduino.cc/libraries/dht-sensor-library/, 2023. Consultado el 9 de julio de 2025.

[25] Trimble. Nmea-0183 messages - message overview. https://receiverhelp.trimble.com/alloy-gnss/en-us/NMEA-0183messages_MessageOverview.html. Consultado el 9 de julio de 2025.

[26] Volodymyr Shymanskyy. Tinygsm | arduino documentation. https://docs.arduino.cc/libraries/tinygsm/, 2024. Consultado el 9 de julio de 2025.

[27] Nick O'Leary. Pubsubclient | arduino documentation. https://docs.arduino.cc/libraries/pubsubclient/, 2020. Consultado el 9 de julio de 2025.

[28] Eclipse Foundation. Eclipse Mosquitto. https://mosquitto.org/. Consultado el 9 de julio de 2025.

[29] Eclipse Foundation. test.mosquitto.org - Eclipse Mosquitto. https://test.mosquitto.org/. Consultado el 9 de julio de 2025.

[30] Espressif Systems. Esp8266 low power solutions. https://www.espressif.com/sites/default/files/9b-esp8266-low_power_solutions_en_0.pdf, 2016. Consultado el 9 de julio de 2025.

[31] Volodymyr Shymanskyy. Streamdebugger | arduino documentation. https://docs.arduino.cc/libraries/streamdebugger/, 2016. Consultado el 9 de julio de 2025.

[32] Adafruit Industries. Adafruit IO. https://io.adafruit.com/. Consultado el 9 de julio de 2025.