UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

PROYECTO PARA LA MONITORIZACION DEL PASO DEL AGUA DE RIEGO MEDIANTE APLICACIÓN MOVIL.

Israel Baeza Domínguez

Tutor: José Enrique López Patiño

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingeniería de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2020-21

Escuela Técnica Superior de Ingeniería de Telecomunicación

Universitat Politècnica de València

Edificio 4D. Camino de Vera, s/n, 46022 Valencia

Tel. +34 96 387 71 90, ext. 77190

www.etsit.upv.es

VLC/ CAMPUS VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS Gestión ambiental verificada REG.NO. ES-CV- 000030

Valencia, 12 de septiembre de 2021

Resumen

En el siguiente trabajo se explica el proceso de desarrollo de una aplicación para dispositivo móvil capaz de monitorizar en tiempo real el flujo del agua por los canales de irrigación de la zona bajo estudio. Se diseñará específicamente para esta función un dispositivo que recogerá datos en campo y los retransmitirá hasta la aplicación en destino, utilizando para este proceso servidores y bases de datos alojadas en la nube.

La aplicación está diseñada enfocada hacia un grupo especifico de usuarios, como las comunidades de regantes. El objetivo principal del proyecto es intentar reducir en lo posible las esperas sufridas por los usuarios a la hora de emplear el sistema de irrigación, proporcionando para ello mayor control sobre el agua de riego utilizada.

La aplicación contara con visualización de mapas donde poder observar el estado de los dispositivos sensores, junto con otras funcionalidades añadidas, como mensajería instantánea entre usuarios y sección de avisos de la comunidad o colectivo de usuarios.

Resum

En el següent treball s'explica el procés de desenvolupament d'una aplicació per a dispositiu mòbil capaç de monitoritzar en temps real el flux de l'aigua pels canals d'irrigació de la zona sota estudi. Es dissenyarà específicament per a aquesta funció un dispositiu que recollirà dades en camp i els retransmetrà fins a l'aplicació en destinació, utilitzant per a aquest procés servidors i bases de dades allotjades en el núvol.

L'aplicació està dissenyada enfocada cap a un grup específic d'usuaris, com les comunitats de regants. L'objectiu principal d'el projecte és intentar reduir en el possible les esperes sofertes pels usuaris a l'hora d'emprar el sistema d'irrigació, proporcionant per això major control sobre l'aigua de reg utilitzada.

L'aplicació comptés amb visualització de mapes on poder observar l'estat dels dispositius sensors, juntament amb altres funcionalitats afegides, com missatgeria instantània entre usuaris i secció d'avisos de la comunitat o col·lectiu d'usuaris.

Abstract

The following work explains the process of developing an application for a mobile device capable of monitoring in real time the flow of water through the irrigation channels of the area under study. A device will be specifically designed for this function that will collect data in the field and relay it to the application at the destination, using servers and databases hosted in the cloud for this process.

The application is designed focused on a specific group of users, such as irrigation communities. The main objective of the project is to try to reduce as much as possible the waits suffered by users when using the irrigation system, providing for this greater control over the irrigation water used.

The application will have a visualization of maps where you can observe the status of the sensor devices, along with other added functionalities, such as instant messaging between users and the community or user group notifications section.

Índice

Capítulo 1. Introducción

El trabajo realizado surge tras un estudio de observación basado en el colectivo de operarios de los canales de irrigación de la parte noroeste de la ciudad de Valencia. Tras investigar el sistema de uso de las acequias, se pudo verificar uno de los puntos críticos de este sistema, las intensas demoras que usualmente sufrían los operarios a la hora de hacer uso de los canales. El estudio previo reveló que existían deficiencias en las funciones de monitorización, mediante las cuales el operario pueda conocer en tiempo real el estado de las acequias relevantes para desempeñar sus funciones. La idea fundamental de este proyecto es el desarrollo de un dispositivo capaz de registrar el estado del flujo del agua de riego, con función de envío de datos hasta recepción en aplicación. El operario podrá observar en su dispositivo móvil el estado de sus ubicaciones de riego de manera remota y en tiempo real.

Las dificultades técnicas principales aparecen al situar el dispositivo en campo abierto. Estas son, la alimentación del dispositivo y la elección del sistema de envío de datos. Es necesario que el dispositivo sensor tenga autonomía energética y que el sistema de transmisión de datos resulte eficiente, asequible y fiable.

Se usarán diferentes tecnologías para el desarrollo del proyecto:

- Para la captación de datos se trabaja con sensores y placas de desarrollo propias del sistema Arduino.
- Para la transmisión de datos desde campo hasta el servidor se utiliza la red Sigfox.
- Para el almacén de los datos en la nube se utiliza la base de datos Firebase Realtime Database.
- Los datos obtenidos se muestran con una aplicación para dispositivo móvil desarrollada mediante el software Android Studio.

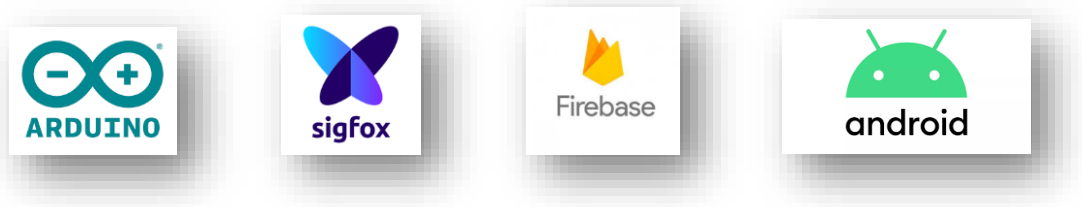


Figura 1. Tecnologías implicadas en el proyecto.

Capítulo 2. Objetivos

El resultado final que se espera conseguir es una aplicación para dispositivo móvil que proporcione en tiempo real el estado de la acequia donde esté situado el dispositivo. Se pretende con este proyecto minimizar las demoras sufridas por el colectivo de operarios para el cual está enfocado el diseño y creación de la aplicación.

Se ha realizado un prototipo, utilizando la herramienta FigmaTools, previo al proceso de creación de la aplicación misma, para partir de una idea sobre las funcionalidades y diseño gráfico del proyecto. Se detallan a continuación algunos bocetos de la futura aplicación móvil.

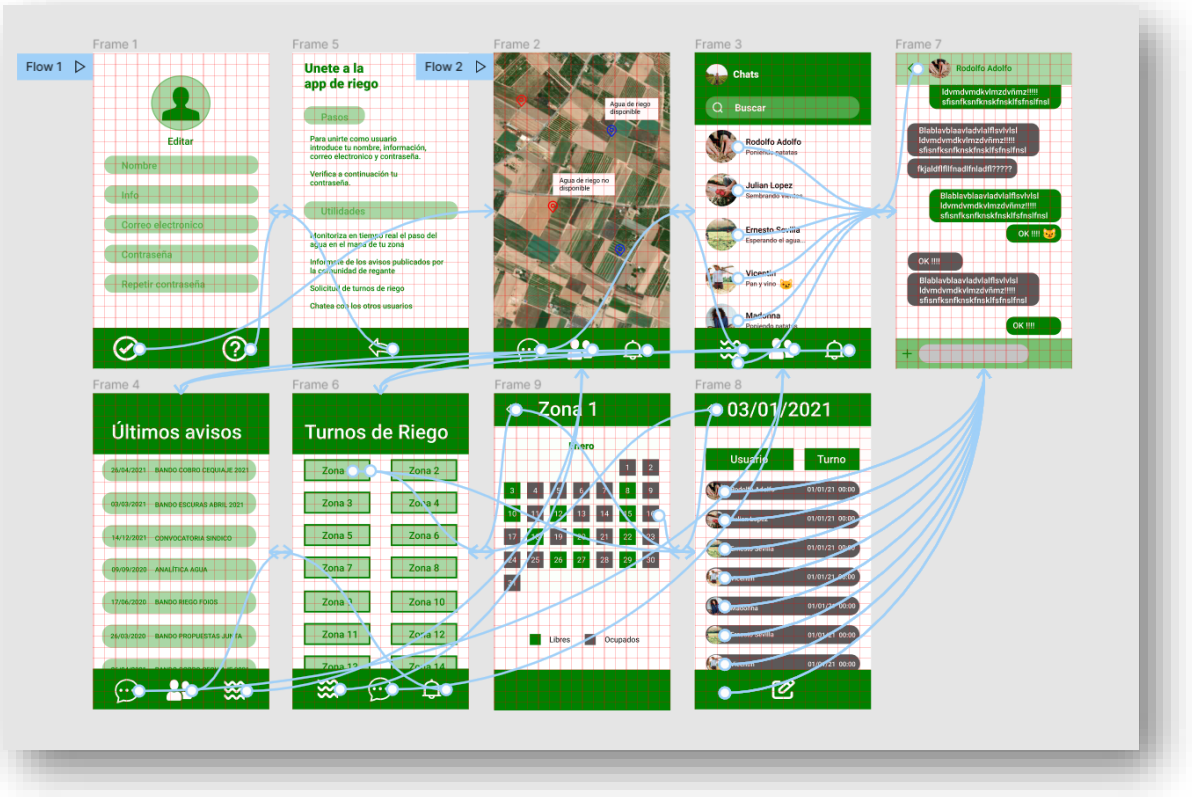


Figura 2. Prototipo de la futura aplicación.

Podemos observar el funcionamiento de este prototipo accediendo al siguiente enlace:

https://www.figma.com/proto/IDOd5RejSqe4MkIif9hXSr/Uno?page-id=0%3A1&node-id=26%3A27&viewport=241%2C48%2C0.5&scaling=min-zoom&starting-point-node-id=26%3A27

Para conseguir el resultado final, la aplicación móvil, se deben cumplir unos objetivos separados en secciones:

I- **Tomar datos validos utilizando el caudalímetro y la placa de Arduino.** Este objetivo es primordial, puesto que, sin recogida de datos válidos, no podemos continuar con el proyecto.
II- **Transmitir estos datos desde la placa hasta el backend de Sigfox.** Debemos configurar correctamente el programa que implementa el envío de datos a la nube.
III- **Enviar los datos desde el backend de Sigfox al backend de Firebase.** El objetivo de esta sección es conectar Sigfox con una base de datos, en este caso, Firebase Realtime Database.
IV- **Crear la aplicación con Android Studio. (Modelo Básico).** Crearemos una primera aplicación para unir Firebase con Android y obtener los datos.
V- **Crear la aplicación con Android Studio. (Modelo Final).** Finalmente, implementaremos el resto de las funcionalidades de la aplicación.

Como objetivo transversal al desarrollo experimental del proyecto, se podría incluir el objetivo de encontrar una solución a un problema o una situación dada, utilizando para ello las herramientas tecnológicas y los conocimientos teóricos, la cual redunde en el beneficio colectivo.

Capítulo 3. Metodología

3.1 Gestión del proyecto

Se ha realizado un proceso previo de investigación, tanto en campo, observar y conocer el sistema de irrigación y sus operarios, así como un proceso de estudio de los dispositivos y sistemas necesarios para conseguir el resultado. Una vez resuelta la viabilidad del proyecto, se ha procedido a estudiar detenidamente estos dispositivos y tecnologías con tal de poder obtener de ellas las funcionalidades que requiere este proyecto. Familiarizados con los sistemas a usar, se ha procedido a configurarlos para que exista transmisión de datos entre ellos. Paralelamente trabajamos con el sensor en laboratorio, con el fin de captar datos válidos. Posteriormente se pasó a probar el sensor en campo. Una vez conseguida la transmisión de datos entre todos los sistemas, se inicia la creación de la aplicación.

3.2 Distribución en tareas

Las tareas en las que se ha dividido el proyecto son las siguientes:

I- **Investigación.** Fase de estudio de las condiciones necesarias para llevar a buen puerto el proyecto. En esta fase se incluye la realización del prototipo con FigmaTools.

II- **Sensor y Arduino.** En esta etapa se trabajará con el sensor, en este caso un caudalímetro, conectado al microcontrolador MKRFOX-1200.

III- **Arduino y Sigfox.** En esta parte se configurará la transmisión de datos entre el microcontrolador y el backend de Sigfox.

IV- **Sigfox y Firebase.** Sección dedicada a la conexión entre estos dos sistemas. Transmisión de datos desde el backend de Sigfox a la base de datos de Firebase.

V- **Firebase y Android.** Creación de una aplicación básica con tal de priorizar la configuración de Android Studio con Firebase y poder obtener los datos del sensor.

VI- **Aplicación móvil.** Una vez obtenidos los datos, se dotará a la aplicación de otras funcionalidades y de un diseño más adecuado.

3.3 Diagrama temporal

A continuación, se adjunta una tabla con la distribución de las tareas a lo largo del tiempo. Como se puede observar algunas de estas tareas se han realizado paralelamente. En el último tramo se ha retomado el trabajo con el sensor y el microcontrolador debido a cambiar la configuración a funcionamiento en campo abierto.

MAYO	JUNIO	JULIO	AGOSTO
Investigación			
	Sensor y Arduino		Sensor y Arduino
	Arduino y Sigfox	Arduino y Sigfox	
		Sigfox y Firebase	
			Firebase y Android
		Aplicación móvil	Aplicación móvil

Tabla 1. Distribución temporal de las tareas.

Capítulo 4. Desarrollo y resultados

El trabajo de campo se ubicará en la parte noroeste de la ciudad de Valencia, concretamente en la localidad de Foios. Esta población posee una gran tradición agrícola, teniendo grandes extensiones de tierras de cultivo. Esta zona de la huerta valenciana es conocida como L´Horta Nord y parte de ella es regada por la Real Acequia de Moncada, uno de los sistemas de riego más antiguos de la Comunidad Valenciana.

En esta localidad se utiliza el riego por inundación o "riego a manta", como sistema de irrigación, método que consiste en inundar el campo en toda su extensión. Para hacer llegar el agua hasta la parcela a regar, se utiliza un laberintico sistema de acequias, divididas en brazos principales y secundarios, todos ellos partiendo de la acequia madre, que a su vez surge de una desviación o azud de la margen izquierda del rio Turia a su paso por la localidad valenciana de Paterna.

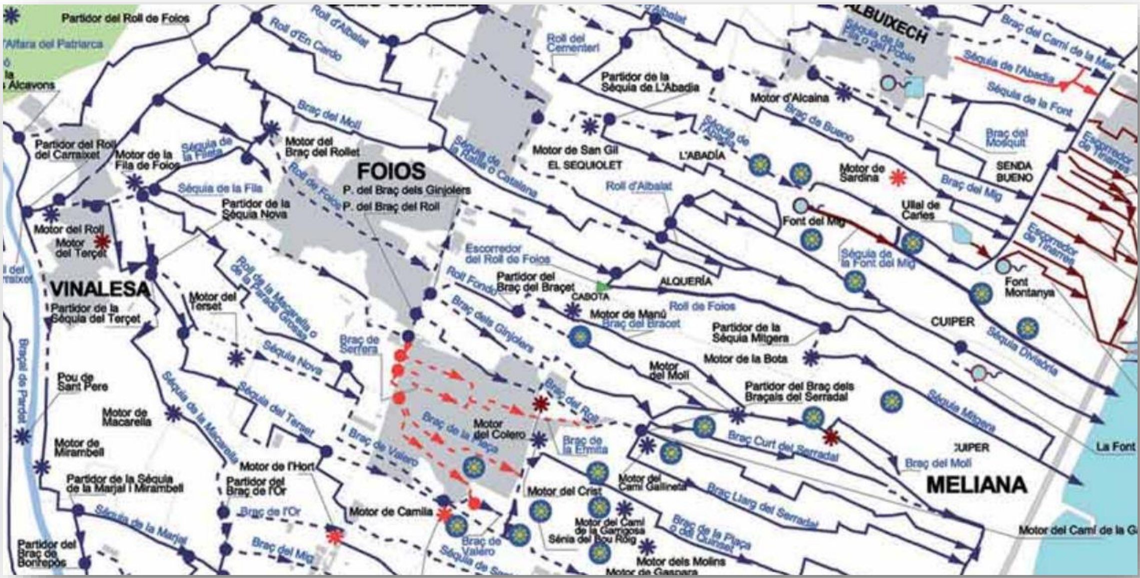


Figura 3. Detalle extraído del mapa "Sistema de Regadío Tradicional en l´Horta Nord: Reial Sequia de Montcada". Colección: Cartografía de los Regadíos Históricos Valencianos.

En la imagen superior se pueden observar algunas de las ramas principales que surgen de la acequia madre a su paso por Foios. Las acequias importantes que riegan estas tierras son principalmente, el Roll de Foios, la acequia de La Fila y la acequia de Alcavons. De estos brazos principales surgen otros ramales que a su vez alimentan otros canales menores, todos ellos divididos mediante compuertas, unas veces metálicas y con sistemas de abertura mediante volante giratorio, y otras veces una simple placa de metal. En la siguiente sección se aporta algo más de información acerca del sistema de riego bajo estudio, así como de sus formas de uso.

4.1 Investigación.

El trabajo de investigación comienza por recabar información acerca de la Real Acequia de Moncada, concretamente de la parte que riega los campos de Foios y Meliana. Se ha realizado una búsqueda de información sobre los brazos importantes que parten del canal principal, su trazado, nacimiento y divisiones. El estudio de campo incluye las ubicaciones de las compuertas principales y seguimiento del trazado a través de los campos de cultivo. De la información encontrada se ha podido extrapolar que las tierras bajo riego se dividen en partidas, al menos en la zona en la que estamos trabajando. Se podría utilizar esta información para nombrar a las ubicaciones en la aplicación móvil, ya que es posible que los operarios estén familiarizados con el uso de estos nombres para denominar a ciertas zonas de las tierras de cultivo.

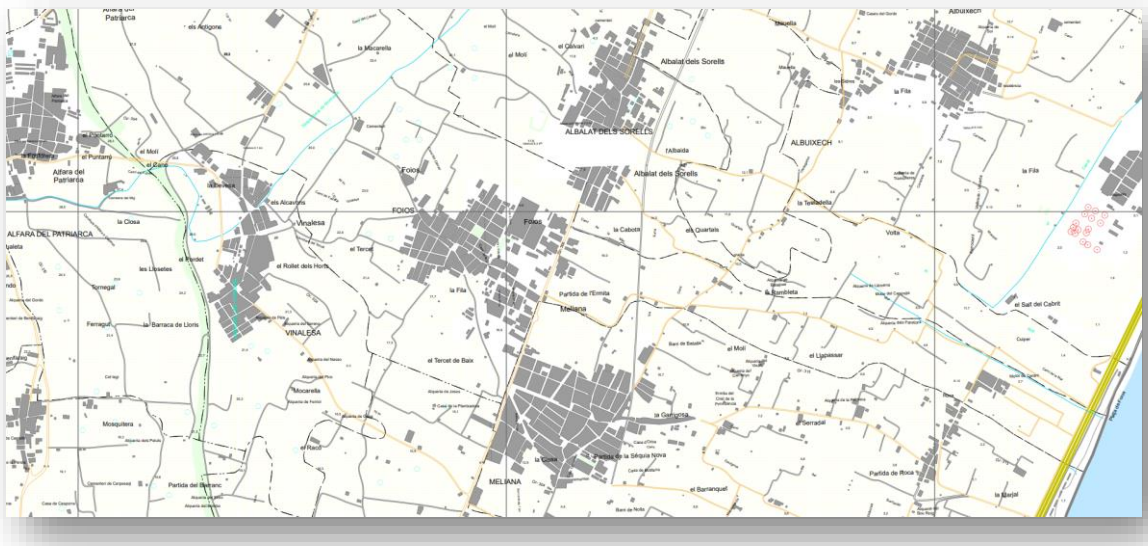


Figura 3. Detalle del mapa "Plan de Infraestructuras. Red de Áreas Cortafuegos". Plan de prevención de incendios forestales de la demarcación de Llíria.

Debido a que el sistema de riego de la zona de estudio constituye una red bastante grande de acequias y canales, el trabajo se centrará en las acequias principales, que nutren a todas las demás. A continuación, se refleja un pequeño análisis de estas ramas principales, mostrando sus principales características, tierras a las que riegan, breve descripción, origen y final, así como otros parámetros importantes.

I. **Roll de Foios.** Posee rumbo Noroeste-Sureste, luego cambia rumbo Sur-Norte. Finalmente adquiere rumbo Este-Oeste hasta su final. Se alimenta de otras acequias menores.

II. **La Fila.** Circula con rumbo Oeste-Este hasta que entra en Meliana y toma rumbo Sur-Norte.

III. **Séquia de Alcavons.** Canal con dirección Suroeste-Noreste. Nace de la margen izquierda del canal principal, caso poco frecuente en todo el recorrido.

	Longitud (metros)	Origen	Final	Partidas regadas	Acequias derivadas
Roll de Foios	5.500	Partida de la Devesa, Vinalesa	Vierte sus aguas directamente en el Mediterráneo, en la partida de Cuiper	La Macarella, LÁlqueria, La Rambleta y Cuiper.	Brac del Moli y Séquia Mitgera
La Fila	2.300	Partidor emplazado al Oeste de Vinalesa, junto a la Avenida principal	Concluye en el Roll de Fondo, otra acequia en el término de Meliana	Norte de la partida del Terset y la partida de la Fila de Foios	Macarella, Terset, Sequia Nova, Parada Grossa, Brac del Roll, Brac dels Ginjolers y Roll de Fondo
Séquia de Alcavons	3.000	Partidor situado en el sector noroccidental del casco urbano de Vinalesa	Partida de la Huitena. Desemboca en la acequia del mismo nombre	Alcavons y la Huitena	

Tabla 2. Acequias principales y sus características.

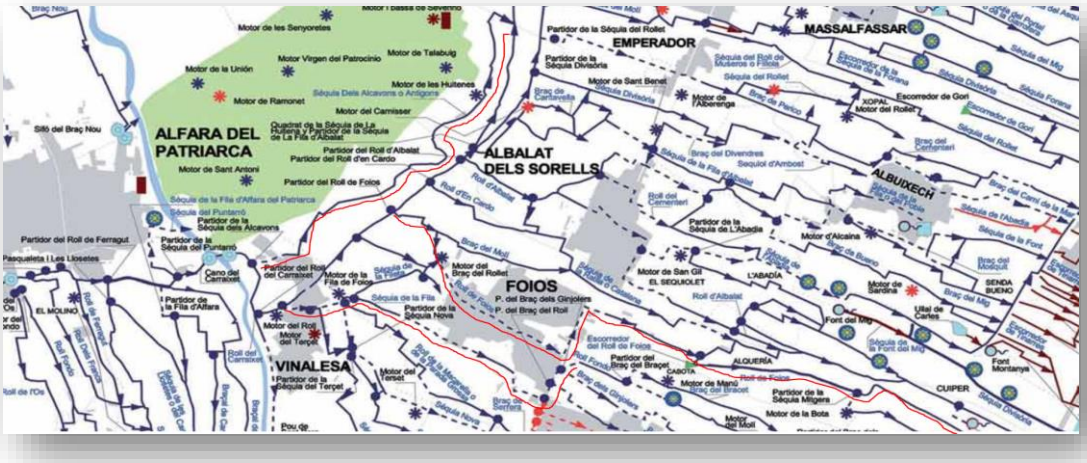


Figura 4. Recorrido de las acequias encargadas de regar el termino de Foios. Extraído del mapa "Sistema de Regadío Tradicional en l´Horta Nord: Reial Sequia de Montcada". Colección: Cartografía de los Regadíos Históricos Valencianos.

En la imagen superior se ha destacado en rojo el recorrido de las tres acequias principales. A continuación, se muestran unas imágenes recogidas de Google Maps que muestran el recorrido real de estos canales.

Figura 5. Recorrido de la acequia de Alcavons. Imagen de la izquierda extraída de Google Maps. Imagen de la derecha extraída del mapa "Sistema de Regadío Tradicional en l´Horta Nord: Reial Sequia de Montcada". Colección: Cartografía de los Regadíos Históricos Valencianos.

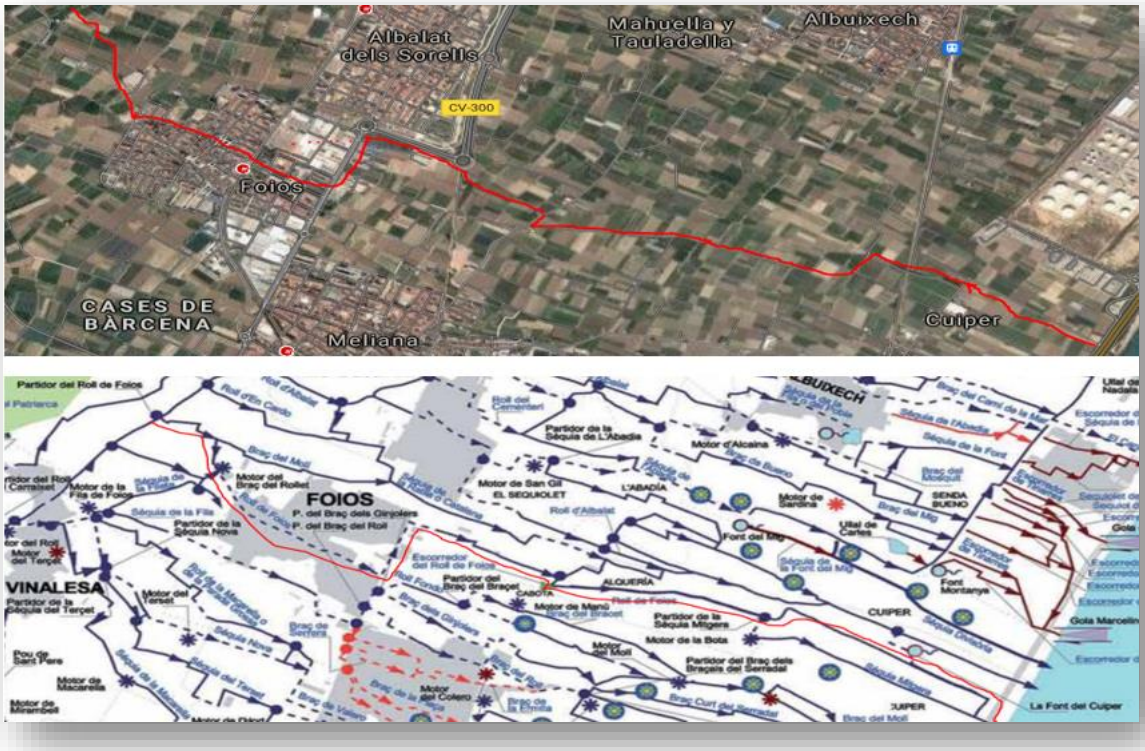


Figura 6. Recorrido de la acequia del Roll de Foios. Imagen superior extraída de Google Maps. Imagen inferior extraída del mapa "Sistema de Regadío Tradicional en l´Horta Nord: Reial Sequia de Montcada". Colección: Cartografía de los Regadíos Históricos Valencianos.

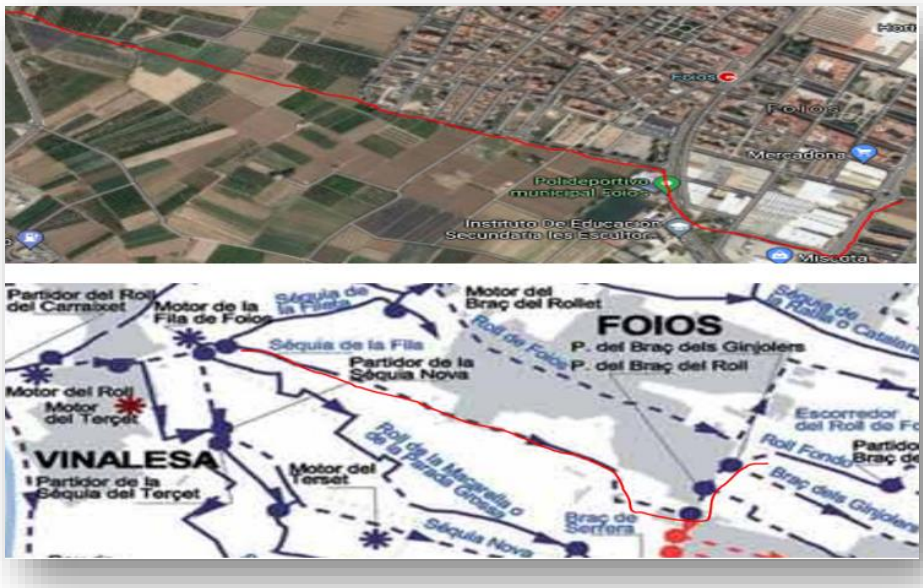


Figura 7. Recorrido de la acequia de La Fila. Imagen superior extraída de Google Maps. Imagen inferior extraída del mapa "Sistema de Regadío Tradicional en l´Horta Nord: Reial Sequia de Montcada". Colección: Cartografía de los Regadíos Históricos Valencianos.

Por cuestiones de cercanía, el experimento centrara sus pruebas de campo en el canal del Roll de Foios. Este canal tiene una bifurcación a cielo abierto importante, que posibilita poder observar directamente las variaciones en el caudal de la acequia. Tiene fácil acceso a la hora de poder realizar mediciones y es punto de encuentro de operarios. Tras una serie de observaciones, se ha verificado que el agua fluye con fuerza cuando el flujo proviene directamente del canal principal y muy lentamente cuando es agua residual, información que será punto importante a la hora de implementar el algoritmo de decisión.

Tras la investigación previa se ha llegado a la conclusión de que el sistema de turnos empleado por los operarios es bastante rudimentario. En la página web de la Real Acequia de Moncada se publica un aviso regularmente, el cual anuncia cuando arribara el agua de riego a ciertas localidades.

Figura 8. Detalle la sección de Turno de Riego de la web de la Real Acequia de Moncada.

Este mismo aviso se puede encontrar en la cuenta oficial de WhatsApp utilizada por cada ajuntamiento de las zonas por las que discurre el canal principal de la Real Acequia de Moncada. La información aportada por este anuncio tan solo facilita el conocimiento de la apertura de las compuertas principales de la acequia madre. El operario cuenta con una información demasiado escasa, debido a que no puede calcular de manera eficiente el tiempo de espera hasta que su turno se haga efectivo. Otro claro ejemplo de la precariedad de este sistema es la imagen que se muestra a continuación, una pizarra para apuntar los turnos.

Figura 9. Pizarra donde se apuntan los regantes

Puesto que solamente se tiene conocimiento de cuando se abren las compuertas principales y que el sistema de turnos no deja suficientemente claro los horarios de uso, solo se llega a tener una idea aproximada de cuándo va a aparecer el agua de riego por las proximidades de cada usuario. Esto desemboca en esperas muy prolongadas.

El trabajo de investigación incluye entrevistas con los operarios para recabar información, intercambio de impresiones y estudio de aceptación del producto. Gracias a la información aportada por un operario, se dedujo que no había ningún sistema para el usuario, ni aplicación móvil, ni página web de internet, donde pudieras obtener la posición del agua de riego en tiempo real. Que las esperas eran, como se había supuesto, muy largas, y que un producto semejante tendría buena aceptación por parte del colectivo de usuarios.

Se pudo averiguar que no existía un sistema de turnos como tal. Se pagaba una cuota por el uso del agua. Este pago otorga el derecho a usar las acequias que conducen el agua hasta tus tierras. El turno de cada usuario es simplemente cuando el agua de riego llega a sus tierras. Con este sistema las esperas están aseguradas.

Una vez familiarizados con el sistema de regadío y su uso, se inició una investigación acerca de los sistemas y dispositivos necesarios para desarrollar la aplicación. El estudio no partía de cero, debido a realizaciones anteriores de proyectos similares basados en Arduino, Firebase y Android. En uno de estos proyectos el usuario podía ver datos de sensores en tiempo real en su dispositivo móvil y en base a esta información, activar ciertos sistemas. El proceso debía ser similar, las principales diferencias surgirían a la hora de enviar los datos, debido a que en otros proyectos se habían utilizado placas de desarrollo con conexión WIFI para acceder a la red del usuario, lo cual obliga a que haya una red disponible y que esta permita el acceso del dispositivo a internet.

En el caso bajo estudio, es poco probable que, en la ubicación del dispositivo, una acequia, exista acceso WIFI a internet, por lo que no serviría este sistema de comunicación. Tras una consulta con un experto en la materia, se llegó a la conclusión de que para poder dotar de comunicación con internet al dispositivo necesitaría incorporar un modem 4G y una tarjeta SIM, en caso de querer usar las redes móviles comerciales. Añadir un modem y pagar por una línea de datos aumentaría de manera considerable el coste del equipo y en el caso de implementar un proyecto en el que se utilicen muchos dispositivos, este sistema no sería factible. Esta información incluía soluciones que no usaban las redes móviles comerciales, como Sigfox, una compañía que ha creado su propia red inalámbrica, especialmente diseñada para dispositivos IoT, que utilizan poco tráfico de datos. Esta empresa ofrece conexión a su red pagando una cuota de 3.5 euros al año, permite un máximo de 140 mensajes al día con una tasa de 100 bytes por segundo como máximo. Más adelante se expondrán con más detalle las características principales del sistema Sigfox.

Una vez escogido el sistema que se debía utilizar para la transmisión de datos, se dio paso a investigar el tipo de sensor necesario para este proyecto. Debía ser un dispositivo capaz de discernir si existía o no agua de riego. En un primer momento se investigó un medidor de humedad del suelo. Este dispositivo trabaja con la conductividad del medio. Son básicamente dos conductores situados paralelamente, unidos a una placa de circuito la cual proporciona un valor en función de la conductividad del ambiente donde este situado. Cuando este dispositivo se encuentra dentro del agua, da un valor máximo de 1000. En un ambiente completamente seco, el valor sería cero. Normalmente al aire libre, sin estar sumergido en agua, da valores inferiores a 1000. Este sistema funcionaria en el caso de que en la acequia solo existieran dos estados, llena de agua o sin agua, y esto no es siempre el caso. Con frecuencia después del riego, o tras una tormenta, puede quedar agua residual, que este sistema identificaría como agua de riego. Se debían buscar otras soluciones que fueran capaces de diferenciar entre agua residual y agua de riego. El dispositivo más acertado sería una especie de detector del flujo del agua, cuando esta fluya rápido y con fuerza significa que es agua de riego, en caso de discurrir lentamente, o con menor velocidad que la registrada en un rango, no es agua de riego. Un dispositivo que presenta un comportamiento parecido es el caudalímetro el cual, en función de la velocidad del agua, proporciona una respuesta en forma de pulso cuadrado con una frecuencia determinada, a mayor velocidad del agua, mayor es la frecuencia. Mediante unas formulas se puede calcular el caudal en función de esta frecuencia, pero para este caso, con la frecuencia podría ser suficiente.

Como microcontrolador se investigó qué posibilidades ofrecía Arduino. La búsqueda debía centrarse en algún modelo que ofreciera compatibilidad con el sistema Sigfox. Tras una primera revisión su pudo comprobar que existía en el mercado una placa de Arduino que incorporaba el sistema de conexión de Sigfox, el modelo MKRFOX-1200. Esta placa de Arduino incorpora todas las funcionalidades propias del sistema Atmel-SAMD21, microcontrolador incluido en las placas de Arduino MKR, más un circuito integrado para la conexión a la red Sigfox. Este dispositivo esta especialmente diseñado para soluciones IoT y presenta unas características que lo convierten en el candidato perfecto para este proyecto.

Como base de datos se dirigió la atención hacia Firebase Realtime Database, ya que se había trabajado con este sistema en proyectos anteriores. Este contenedor en la nube almacena los datos en formato Json y permite la lectura y escritura de estos mediante peticiones.

Finalmente, como destino final de los datos, se diseñará una aplicación móvil con Android Studio. En esta aplicación el usuario podrá observar en un mapa el estado actual de la acequia o acequias donde estén situados los dispositivos. Esta aplicación será unidireccional en este aspecto, solo recibirá datos de los sensores, no enviará datos a estos. Contará también con funciones extras, como mensajería instantánea entre usuarios y posibilidad de avisos en caso de que fuera gestionada por una comunidad de regantes o asociación de usuarios.

4.2 Sensor y Arduino

Las especificaciones técnicas de cada dispositivo aparecerán explicadas detalladamente en el capítulo de materiales. En este apartado, se detalla únicamente la configuración necesaria de cada dispositivo para poder desarrollar el experimento.

Para este proyecto se trabajará con el caudalímetro YF-S201, de conexión de media pulgada. Su funcionamiento se basa en una rueda giratoria accionada por el paso del agua. Esta rueda lleva incorporado un imán en su centro. Un sensor Hall alojado en la carcasa detecta las variaciones en el campo magnético producidas por el imán al girar y responde con un pulso cuadrado, con frecuencia proporcional al giro de la rueda, el cual se envía a través del cable de señal hasta uno de los pines de entrada de la placa.

El sensor estará conectado a la placa MKRFOX-1200, la cual utiliza un microcontrolador Atmel-SAMD21 de 32 bits, especialmente diseñado para trabajar con dispositivos IoT, más un módulo Sigfox ATA8520, para conectarse a su propia red. Esta placa se alimenta de tres formas posibles, a través de conector USB, una entrada Vin, que admite 5 voltios de una fuente de alimentación regulada, más dos bornes para conectarle dos pilas AA o AAA en serie, resultando 3.3 voltios. Este dispositivo esta especialmente diseñado para funcionar mucho tiempo alimentado con pilas. Configurándolo de la manera adecuada, el dispositivo puede alcanzar una autonomía energética cercana a los seis meses.

El sensor presenta tres conectores, dos que serán la alimentación, más la señal de datos, por la cual se transmitirá el pulso cuadrado. En la siguiente imagen podemos ver el sistema de conexiones.

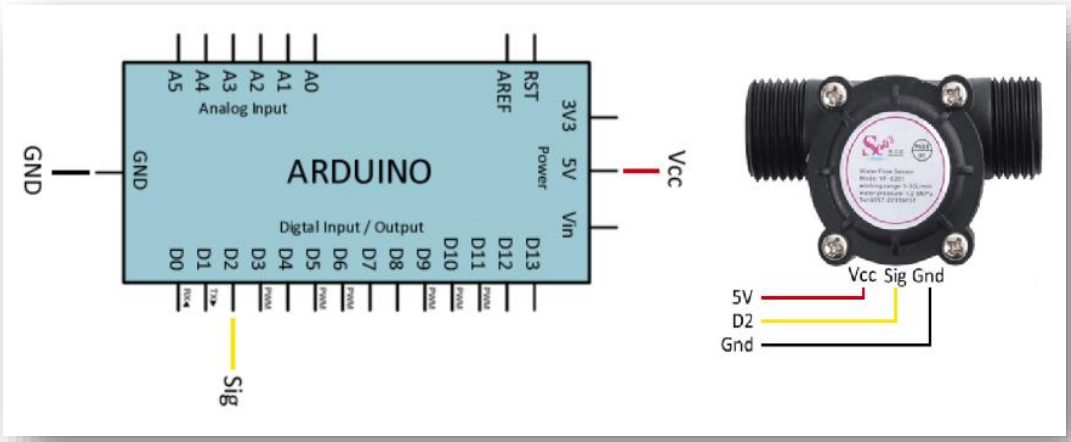


Figura 10. Conexiones entre el sensor y la placa. Imágenes extraídas del tutorial "Medir caudal y consumo de agua con Arduino y caudalímetro", del autor Luis Llamas"

Una vez realizadas las conexiones debemos implementar el código en el entorno de programación de Arduino. El primer paso será configurar el entorno del IDE de Arduino para trabajar con la placa MKRFOX-1200. En la pestaña Herramientas seleccionamos la placa que vamos a utilizar, en este caso debemos elegir MKRFOX-1200. También debemos seleccionar el puerto de comunicación del dispositivo con el ordenador. Normalmente, al conectar el dispositivo, deberá aparecer reflejado este en el puerto en el que se encuentre.

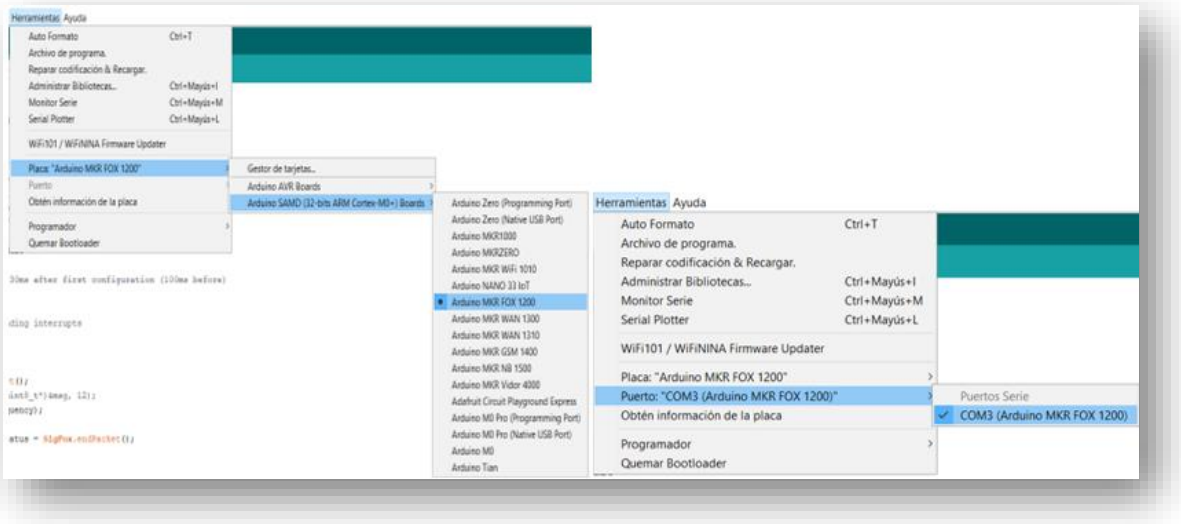


Figura 11. Configuración del entorno de programación de Arduino.

Una parte del código utilizado se ha extraído del tutorial "Medir caudal y consumo de agua con Arduino y caudalímetro", del autor Luis Llamas. Podemos encontrar este código, así como el tutorial para realizar el experimento que nos muestra el autor en el enlace incluido en la sección de bibliografía. Este código implementa una interrupción, la cual utilizamos para contar los pulsos que nos transmite el sensor en un determinado tiempo. Una vez tenemos el número de pulsos, lo dividimos por el intervalo de tiempo utilizado y conseguimos la frecuencia. El ejemplo también nos muestra como calcular el caudal una vez conseguida la frecuencia. Para nuestro caso, no utilizaremos esa parte, tan solo trabajaremos con la frecuencia.

El código utilizado para conseguir la frecuencia del paso del agua es el siguiente:

```
#include <SigFox.h>

#include <ArduinoLowPower.h>
```

Librerías con las que vamos a trabajar. La primera nos permitirá el uso de dispositivos con tecnología Sigfox. La librería LowPower nos permite activar el modo descanso, ahorrando energía.

```
const int sensorPin = 1;

const int measureInterval = 2500;

volatile int pulseConter;
```

En estas tres instrucciones definimos el pin de entrada del pulso del sensor, el intervalo de tiempo en el que se contaran los pulsos y una variable pulseCounter donde se almacenara el número de pulsos.

```
void ISRCountPulse(){
  pulseConter++;
}
float GetFrequency(){
  pulseConter = 0;
  interrupts();
  delay(measureInterval);
  noInterrupts();
  return (float)pulseConter * 1000 / measureInterval;
}
```

En el primer método se implementa un contador. En el segundo método se cuentan los pulsos. La respuesta de sensor, una onda cuadrada hace que el pin seleccionado alterne entre HIGH y LOW. Cada cambio de estado crea una interrupción y cada interrupción aumenta el contador. Una vez calculado el número total de interrupciones en el intervalo de tiempo, se traduce en el número de pulsos obtenido, se divide entre el intervalo de tiempo seleccionado y se obtiene la frecuencia.

```
void setup(){
  Serial.begin(9600);
  attachInterrupt(digitalPinToInterrupt(sensorPin), ISRCountPulse, RISING);
}
```

En esta parte implementamos la comunicación serie con el ordenador a una velocidad de 9600 baudios y configuramos el pin para que produzca interrupciones. Los parámetros que pasamos a la función son el pin antes inicializado, el método al que se llama cuando se produce la interrupción y, por último, el modo de funcionamiento, en este caso RISING, que significa que implementa la interrupción cuando el estado del pin pasa de bajo a alto.

```
void loop(){
  float frequency = GetFrequency();
  Serial.print("Frecuencia: ");
  Serial.println(frequency, 0); }
```

Finalmente, en el bucle creo una variable float donde cargo el valor de la frecuencia obtenida en con el método anterior y la muestro por la consola del IDE de Arduino, para una primera comprobación del resultado.

Una vez implementado este código, el cual solo cuenta con la parte para la obtención de la frecuencia, había que ejecutarlo y comprobar que el sensor y la placa se entendían correctamente. Aquí surgió el primer problema, una vez configurados ambos dispositivos, no se obtuvieron datos. Comenzó entonces un proceso de verificaciones e investigaciones para encontrar la causa de la ausencia de datos. Como primer recurso, se decidió probar el código con otra placa de Arduino, en este caso una placa similar a Arduino Uno, pero de marca blanca. Obtuvimos resultados válidos. En este caso, accionábamos las aspas del caudalímetro mediante soplido y podíamos observar cómo se obtenían diversos valores para la frecuencia, la cual aumentaba a medida crecía la intensidad del soplido, lo que era una buena señal. El código funcionaba bien, pero no con la placa MKRFOX-1200. Tras una breve investigación por la red pudimos encontrar información acerca de los pines en los que esta placa puede realizar interrupciones, que no son los mismos que los de otras placas de Arduino. En la configuración inicial estábamos utilizando el pin 2 o 3, los cuales producen interrupciones en placas de Arduino que no llevan el microcontrolador SAMD21, en estas últimas, se han cambiado por el 0 y el 1. Una vez subsanado el problema, cargamos el nuevo código en la placa y obtuvimos resultados satisfactorios.

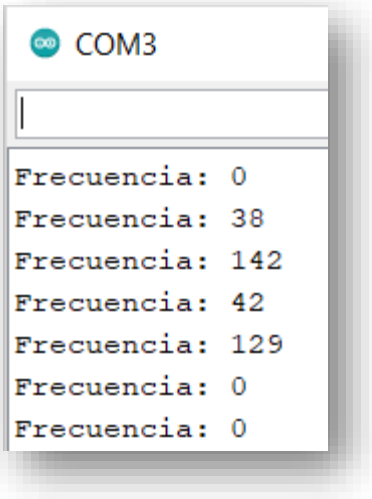


Figura 12. Resultados obtenidos. Podemos apreciar como aumenta y disminuye la frecuencia según la intensidad del soplido.

Una vez conseguida esta parte, teníamos que probar si ambos dispositivos funcionaban correctamente al ser alimentados con baterías AA o AAA. Sabíamos que en este tramo podía presentar dificultades, ya que el sensor, según sus especificaciones, funciona con una tensión de trabajo mínima de 4.5 voltios, valor que cumplíamos al alimentar la placa mediante USB directo al portátil. Cuando alimentamos la placa con pilas, el valor nominal de salida se establece en 3.3 voltios, valor que igual no era suficiente para alimentar el sensor. Las especificaciones del sensor hablan de una tensión mínima de 4.5 voltios y las especificaciones de la placa nos dicen que está especialmente diseñada para funcionar con tensiones cercanas a los 3 voltios, ya que está pensada para trabajar en campo abierto con alimentación de baterías. Sus pines de entrada no admiten más de 3 voltios, pueden llegar a dañarse en caso de superar ese límite.

En una primera configuración, con la placa alimentada con dos pilas AAA en serie conectando el sensor a través de los pines 5V y GND, no obtuvimos resultados. Aquí podíamos tener un problema ya que, si el sensor no se activaba con esa tensión, deberíamos incluir alimentación extra para que funcionara. En un primer momento se utilizó una placa de alimentación que incluía el kit de Arduino adquirido en proyectos anteriores. Esta placa puede alimentarse mediante una batería de 9 voltios y sacar una tensión directa de 5 voltios en uno de sus bornes. La inclusión de esta placa de alimentación no estaba programada, aumentaría el coste y el tamaño del experimento. El resultado funcionaba durante una primera obtención de resultados, pero después bloqueaba la placa y debíamos reiniciarla. No se pudo encontrar porque se producía este bloqueo. De todas formas, la idea de incluir otro componente no estaba contemplada en el proceso. Se procedió a analizar las salidas cuando la placa estaba alimentada con USB y cuando lo estaba con baterías. Mediante el uso de un voltímetro se pudo comprobar que, al estar conectado con USB, la salida 5V proporcionaba una tensión directa de 5 voltios y cuando estaba alimentada mediante pilas, el valor era de 1.3 voltios, demasiado bajo para alimentar el sensor. Probamos a medir la salida Vcc, la cual proporciona 3.3 voltios regulados, aunque se alimente la placa con 5 voltios. No se había contemplado el uso de este pin debido a que se daba por hecho que la placa contaría con algún tipo de conversor de voltaje interno y que, aunque fuera alimentada con una tensión de 3 voltios, devolviera 5 voltios en el pin de 5V. También se pensaba que el sensor no funcionaría alimentado con una tensión de 3.3 V, así que se había desechado el uso de este pin como alimentación para el sensor. Comprobamos que por el pin de Vcc salían 3.3 voltios cuando la placa estaba alimentada a pilas, lo mismo que si lo alimentabas por USB. Realizamos una primera prueba para comprobar si el sensor admitía esta tensión y el resultado fue óptimo. Una vez comprobado que el sensor y la placa podrían funcionar óptimamente alimentados mediante baterías, comenzamos a trabajar en la parte de transmitir los datos obtenidos al backend de Sigfox.

4.3 Arduino y Sigfox

La ventaja de trabajar con la placa MKRFOX-1200 es que en ella podemos encontrar trabajando conjuntamente la tecnología de Atmel y de Sigfox. Por lo tanto, existe código ya creado para ejecutar ejemplos y conducir al usuario inexperto en sus primeros pasos con Sigfox. Los primeros pasos que debemos realizar son dar de alta el dispositivo adquirido. Este dispositivo incluye en su precio la cuota de suscripción a la red de Sigfox, la cual nos concede el uso durante un año. Para registrar nuestro dispositivo en el backend de Sigfox debemos ejecutar un pequeño programa que nos permitirá obtener por la consola del IDE de Arduino dos códigos de identificación, el ID y el PAC. El ID es un número de identificación para cada dispositivo conectado a la red de Sigfox. El PAC o Código de Autorización de Portabilidad es un identificador asociado a la titularidad del dispositivo. Estos códigos permiten que solo el propietario pueda gestionar sus dispositivos dentro de la red. El código utilizado para conseguir estos identificadores se ha extraído de la página web “Introducción Arduino MKRFOX-1200 Sigfox y redes LPWAN “. Podemos encontrar toda la información del autor Luis del Valle, referente a este ejemplo en el enlace adjunto en la sección de bibliografía. Este código nos permite obtener los dos identificadores y realizar un pequeño testeo de la comunicación del dispositivo. A continuación, incluyo las líneas de código utilizadas:

```
#include <SigFox.h>
#include <ArduinoLowPower.h>
void setup() {
  // CONFIGURAMOS MONITOR SERIE
Serial.begin(9600);
delay(1000);
// INICIALIZACION DEL MODULO SIGFOX
if(!SigFox.begin()){
  Serial.print("Error placa");
    return;
}
```

Si el módulo de Sigfox no arranca, mostramos que ha habido un error.

```
// OBTENER ID Y PAC
Serial.print("ID = " + SigFox.ID());
Serial.print("PAC = " + SigFox.PAC());
// APAGAR MODULO SIGFOX
SigFox.end();
}
```

Realizamos dos peticiones al módulo Sigfox para que nos proporcione el ID y el PAC y los mostramos por pantalla. Una vez mostrados los números apagamos el módulo.

```
void loop() {
  // DE momento no hacemos nada aqui
}
```

Una vez obtenidos los identificadores, en la dirección https://buy.sigfox.com/activate, nos pedirá introducir estos dos valores. Rellenados estos campos, podremos configurar el grupo, nombre de dispositivo y otros parámetros, así como visualizar los mensajes recibidos y enviarlos a través de CALLBACKS, una función muy interesante de Sigfox que nos permite enviar los datos a otras plataformas, como correo electrónico o bases de datos. Una vez configurado el dispositivo, ejecutaremos un pequeño código de muestra para probar el envío de datos al backend de Sigfox. Este programa consiste en obtener la temperatura de la placa MKRFOX-1200 y enviar este dato al servidor. Antes de mostrar el código, incluyo unas imágenes del proceso de identificación del dispositivo y de la consola de trabajo de Sigfox una vez registrado el producto.

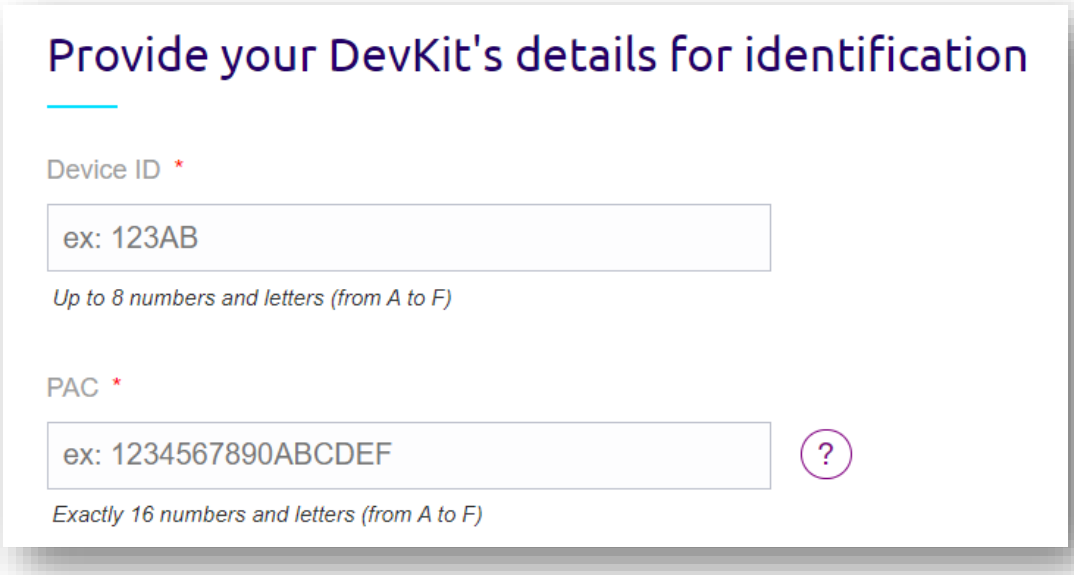


Figura 13. ID y PAC, identificadores de nuestro dispositivo.

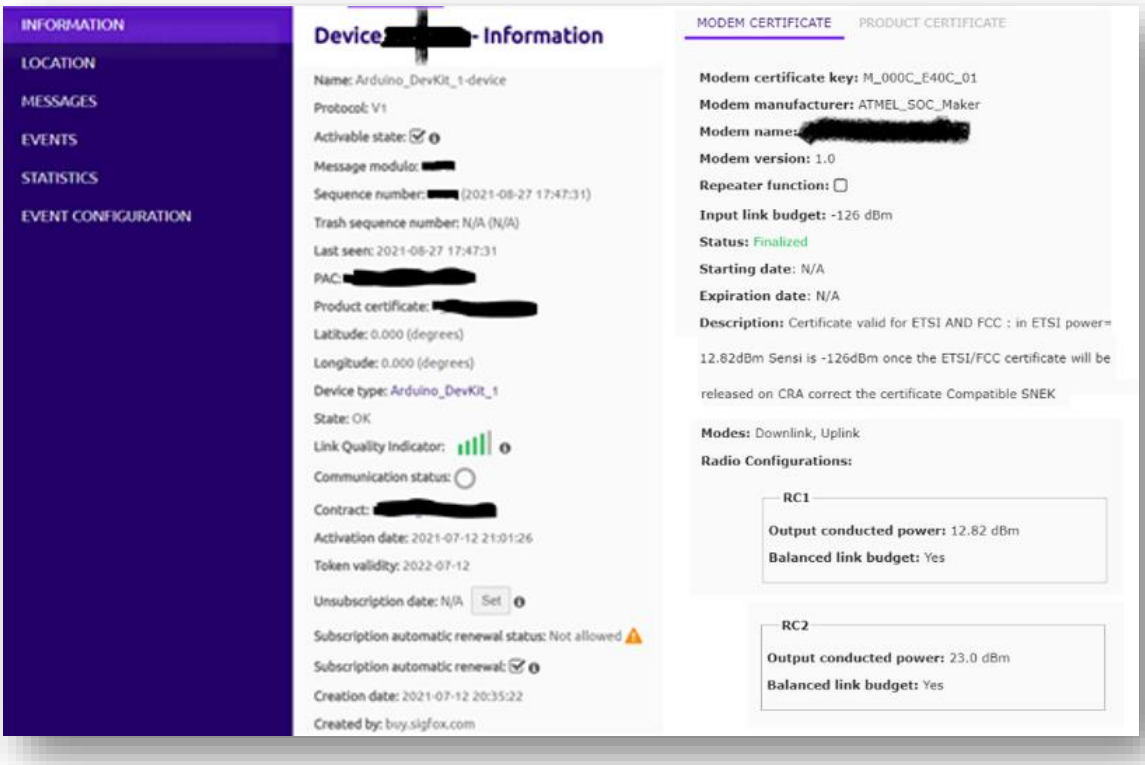


Figura 14. Backend de Sigfox, pestaña de información del dispositivo.

En esta ventana podemos visualizar algunas características del dispositivo validado, como sus modos de funcionamiento, Downlink y Uplink, parámetros radio, como la potencia de entrada y salida, ubicación, etc.

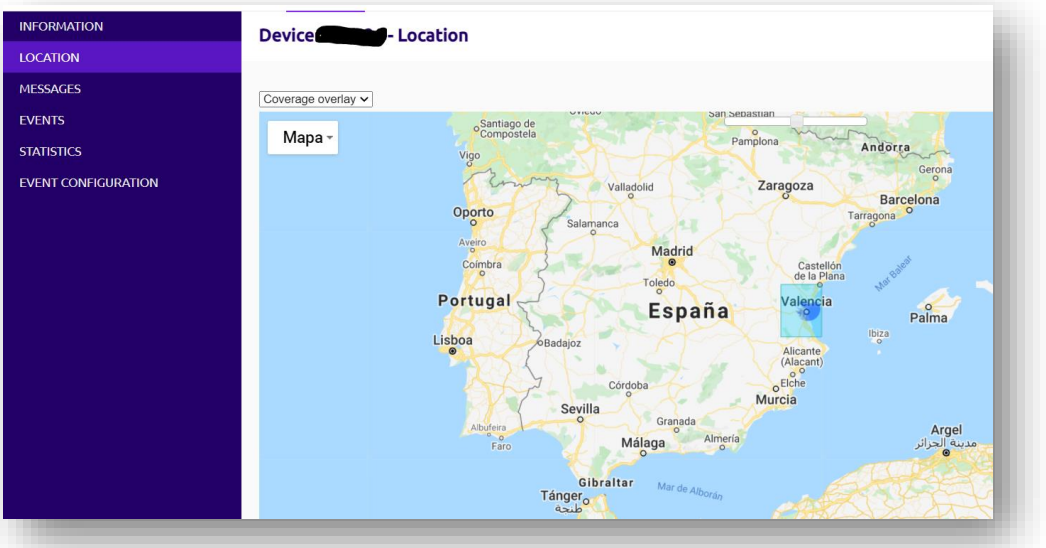


Figura 15. Backend de Sigfox, pestaña de localización del dispositivo.

En esta pestaña podemos observar la ubicación de nuestro dispositivo.

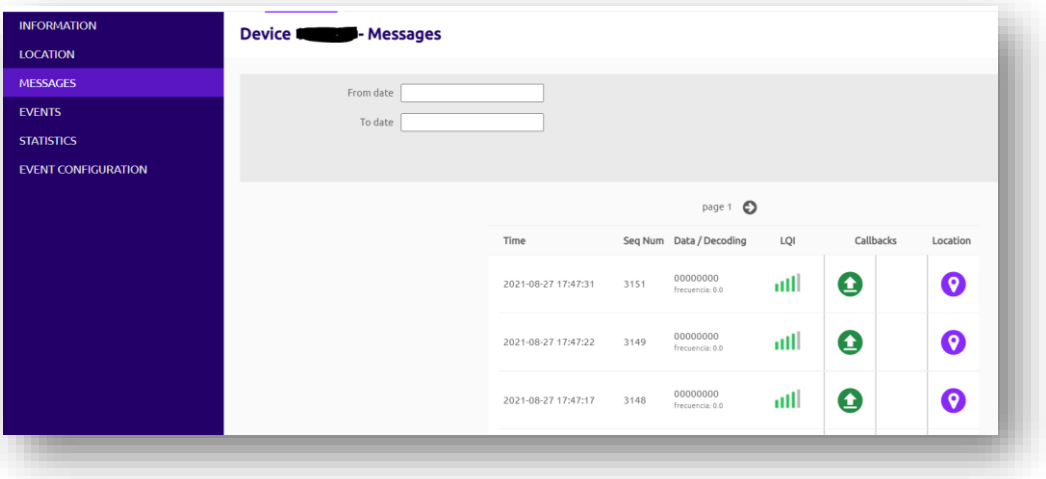


Figura 16. Backend de Sigfox, pestaña de mensajes del dispositivo.

La imagen superior nos muestra la parte dedicada a los mensajes recibidos. El valor del mensaje viene en formato hexadecimal por defecto, pero podemos configurar el formato en que queremos visualizar los datos, esta configuración se explicara más adelante. En esta sección podemos encontrar también un icono referente a los CALLBACKS, verde si el envío ha sido optimo, rojo en caso de error. Pulsando en este icono obtendremos información del envío mediante CALLBACKS. En esta información aparecerá la causa del error en caso de envío fallido. También cuenta con un indicador de señal recibida y un apartado de localización, que nos traslada a la pestaña de localización.

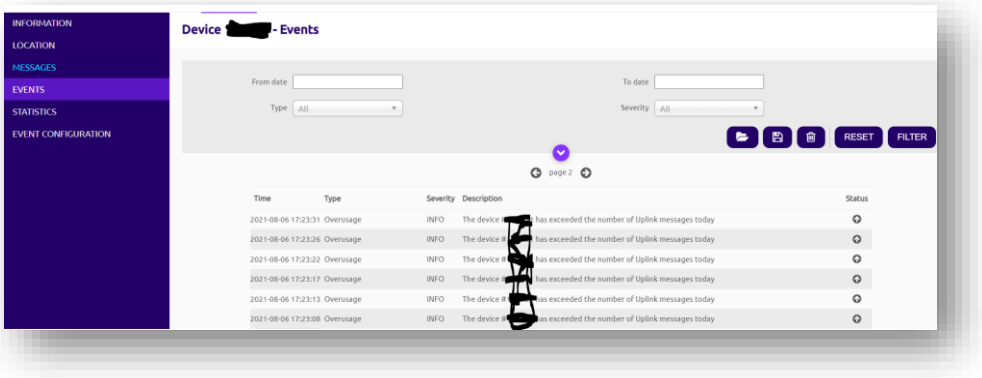


Figura 17. Backend de Sigfox, pestaña de eventos del dispositivo.

En esta pestaña podemos ver avisos del estado de la comunicación entre el dispositivo y el servidor. En caso de superar el máximo de mensajes para un día, recibiremos información sobre ello, así como otro tipo de información, la cual configuraremos en la pestaña de configuración del evento

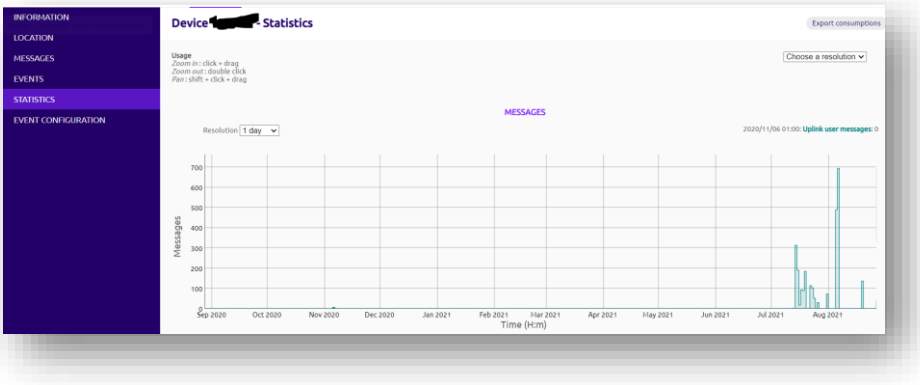


Figura 18. Backend de Sigfox, pestaña de estadísticas del dispositivo.

En este apartado podemos encontrar estadísticas sobre el número de mensajes, el total de bytes enviados, la relación señal/ruido y otros parámetros de interés. Finalmente, en la pestaña de configuración de evento, podemos seleccionar el tipo de aviso que queremos que se adjunte en la información del evento.

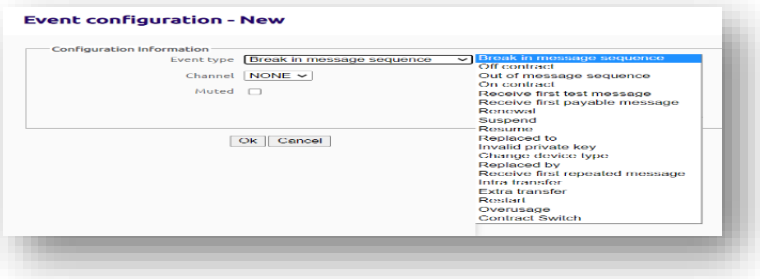


Figura 19. Backend de Sigfox, pestaña de configuración del evento del dispositivo.

Una vez tenemos debidamente configurado el dispositivo, podemos probar el código de ejemplo de transmitir el valor de la temperatura de la placa. A continuación, las instrucciones empleadas para ello:

```
#include <SigFox.h>
#include <ArduinoLowPower.h>
void setup() {
  Serial.begin(9600);
  delay(1000);
  if(!SigFox.begin()){
    Serial.print("Error placa");
    return;
  }
  SigFox.end();
  SigFox.debug();
}
```

Librerías, inicialización de la comunicación serie, apagado del módulo y activación del modo debug.

```
void loop() {
  SigFox.begin();
  delay(200);
  byte temperatura = (int)SigFox.internalTemperature();
  Serial.print("La temperatura interna es de:  ");
  Serial.print(temperatura);
```

En la parte del bucle, inicializamos el módulo Sigfox, cargamos dato de temperatura de la placa en variable “temperatura” y mostramos por pantalla.

```
  SigFox.status();
```

Esta instrucción se encarga de limpiar las interrupciones que puedan encontrarse pendientes.

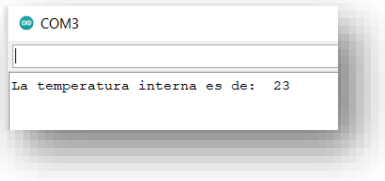
```
  delay(1);
  SigFox.beginPacket();
  SigFox.write(temperatura);
```

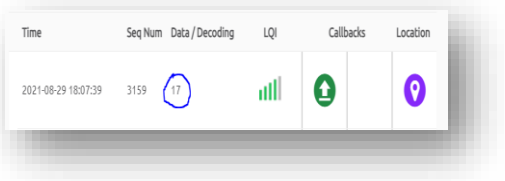
Tras la instrucción de retardo encontramos la función que inicializa el proceso de transmisión de mensajes. La función SigFox.write nos permite el envío de mensajes. Cargamos como argumento la variable con el dato de la temperatura interna de la placa.

```
SigFox.endPacket();
 SigFox.end();
  delay(900000);
}
```

Terminamos el código con estas instrucciones. La primera tiene como función finalizar el proceso de transmisión de mensajes. Apagamos el módulo y creamos un retardo de quince minutos. Hay que tener cuidado con las instrucciones que dejan a la placa en estado de reposo, ya que bloquean el funcionamiento y hacen que la placa se quede en modo durmiente. Solo salimos de este estado transcurrido el tiempo o reseteando.

Con este código verificáremos que el envío de datos desde el sensor y la placa es posible y pasaremos a complementar el anterior programa que nos permitía encontrar la frecuencia de paso del agua, añadiéndole las instrucciones de envío de este valor al backend de Sigfox.





Figuras 20 y 21. Dato de temperatura interna en la consola del IDE de Arduino y el mismo dato en el apartado de mensajes recibidos en el backend de Sigfox.

En la imagen de la derecha podemos ver que el valor de temperatura interna de la placa es de 23 grados. El valor del dato en el backend de Sigfox viene expresado en sistema hexadecimal. Realizando la conversión obtenemos que el valor coincide con el obtenido en la consola del IDE de Arduino.

El código final une la parte de la obtención del dato del sensor junto con la transmisión de este valor al backend de Sigfox. También se implementará una función para dejar al dispositivo en estado de reposo durante 10 minutos. Esta será la frecuencia de envío de mensajes. Como se ha comentado antes, las instrucciones que sitúan al dispositivo en estado durmiente, para ahorro de batería, provocan que el dispositivo no salga de este estado hasta que transcurra el tiempo seleccionado o se reinicie la placa. Al implementar el código utilizando el dispositivo conectado al ordenador, la instrucción de la librería LowPower `LowPower.sleep();` situará al dispositivo en estado de reposo durante el tiempo que pasemos como argumento. Esta función deshabilita la conexión por el puerto serie y a la hora de intentar visualizar los resultados por consola produce error. Esto está justificado ya que esta función está creada para funcionar con el dispositivo alimentado por baterías, con lo que no es necesaria la comunicación por el puerto serie. Al trabajar con el dispositivo en el laboratorio estaba alimentado a través del ordenador, por ahorro de baterías. Al ejecutar el programa, como ultima instrucción se situaba al dispositivo en estado de reposo inmediatamente. Se enviaba el mensaje correctamente, pero al intentar verificar si el dato coincidía con el valor que aparecía en la consola de Arduino, se producía un error de puerto serie no encontrado. La solución fue deshabilitar esta función cuando el trabajo fuera en laboratorio.

A continuación, el código final:

```
#include <SigFox.h>
#include <ArduinoLowPower.h>
const int sensorPin = 1;
const int measureInterval = 2500;
volatile int pulseConter;
void ISRCountPulse(){
  pulseConter++;
}
float GetFrequency(){
  pulseConter = 0;
  interrupts();
  delay(measureInterval);
  noInterrupts();
  return (float)pulseConter * 1000 / measureInterval;
}
void setup(){
  Serial.begin(9600);
  attachInterrupt(digitalPinToInterrupt(sensorPin), ISRCountPulse, RISING);
    if (!SigFox.begin()) {
  // Something is really wrong, try rebooting
    reboot();
```

Si ocurre algún error al inicializar el módulo, reiniciamos.

```
  }
  SigFox.end();
  SigFox.debug();
}
void loop(){
  float frequency = GetFrequency();
  Serial.print("Frecuencia: ");
  Serial.println(frequency, 0);
```

Las líneas superiores corresponden básicamente con el código de la obtención de la frecuencia. A partir de aquí se incluyen las instrucciones para configurar el envío de mensajes al backend.

```
SigFox.begin();
delay(100);
SigFox.status();
 delay(1);
```

Inicializamos el módulo Sigfox y limpiamos las interrupciones pendientes. Retardo para dar tiempo al dispositivo a que se configure.

```
 SigFox.beginPacket();
 SigFox.write(frequency);
 int lastMessageStatus = SigFox.endPacket();
```

Activamos el proceso de envío de mensajes con la primera instrucción. Como argumento de la función SigFox.write para enviar mensajes, cargamos la variable portadora del valor de la frecuencia. Finalmente terminamos el proceso de envío de mensajes. Es posible cargar el resultado del envío de mensajes en una variable int, para visualizar en la consola de Arduino si el envío ha sido óptimo. En caso afirmativo se nos mostrará un cero, en caso contrario, uno.

```
 SigFox.end();
 LowPower.sleep(10 * 60 * 1000);
}
```

Con estas dos instrucciones situamos al dispositivo en estado de reposo durante diez minutos. Con esto conseguiremos más o menos 140 mensajes al día y según tutoriales, un ahorro de batería importante, que nos resulta en una vida útil de las baterías de cerca de seis meses.

```
void reboot() {
  NVIC_SystemReset();
  while (1) ;}
```

Método de reinicio, en caso de que al inicializar el dispositivo se produzca algún error, salta la llamada a este método que ejecuta un reinicio de la placa.

Después de subsanar varios errores, como usar el pin equivocado e intentar visualizar los datos por la consola del puerto serie con la placa en modo durmiente, los cuales supusieron algunos retrasos en el desarrollo de la aplicación, pudimos obtener resultados satisfactorios, tanto en la captura de datos del sensor, como en la transmisión de estos al backend de Sigfox. Una vez comprobado que el mensaje era transmitido, debía configurar el formato de recepción de datos de manera adecuada para una mejor comprobación. A continuación, se incluyen algunas imágenes donde se puede ver el mensaje obtenido, en formato hexadecimal y la pestaña de configuración del formato de los mensajes.

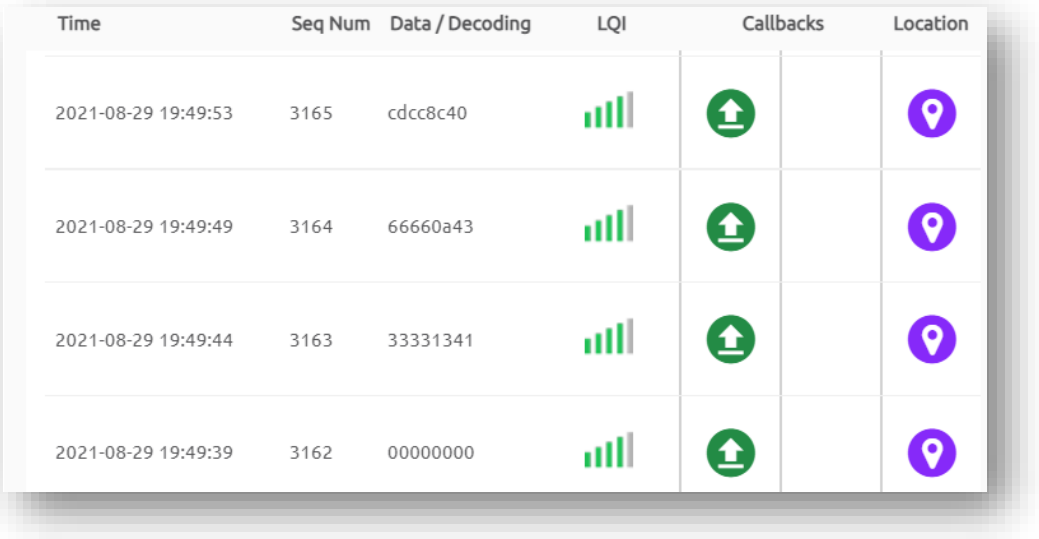


Figura 22. Backend de Sigfox. Mensajes recibidos con formato hexadecimal.

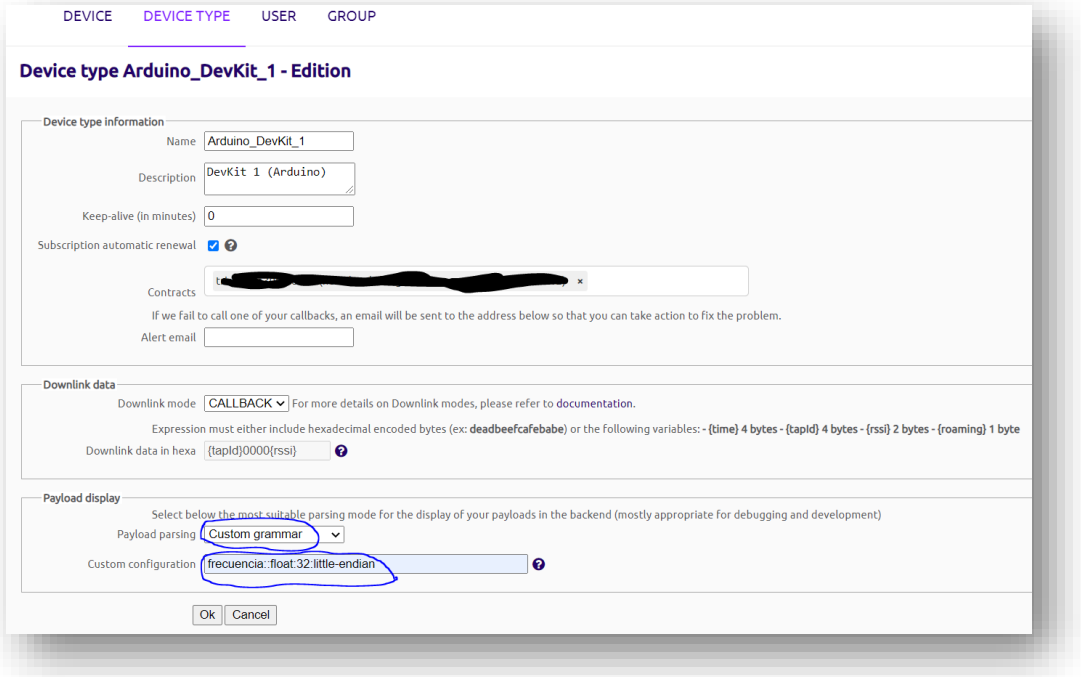


Figura 23. Sección de configuración del tipo de dispositivo.

En este apartado podemos configurar el formato del mensaje. En nuestro caso, elegiremos la opción de gramática personalizada y configuraremos el formato para que lea una variable tipo float, de 32 bits, en formato Little-endian. El resultado podemos visualizarlo en la imagen inferior.

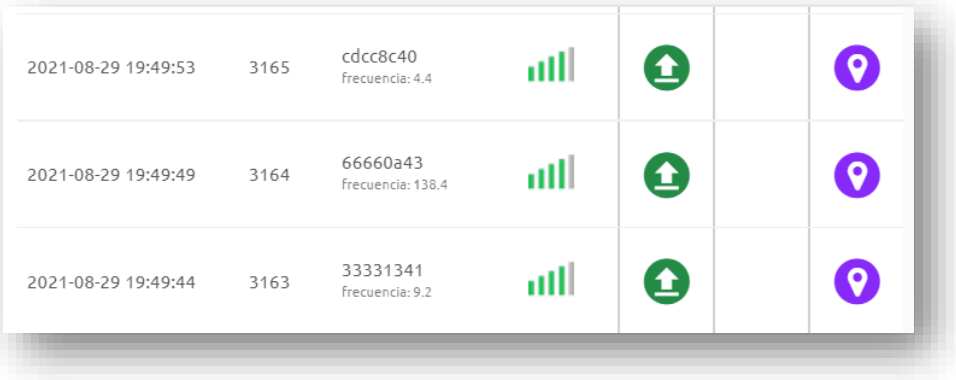


Figura 24. Mensajes con el formato seleccionado.

Podemos ver que también aparece el mensaje en formato hexadecimal, pero se incluye el valor en el formato seleccionado, lo cual facilita la verificación del dato. Una vez conseguido el dato en el servidor, debemos configurar la función de CALLBACKS para que envíe los datos obtenidos a la base de datos. Este proceso se explica en el siguiente capítulo.

4.4 Sigfox y Firebase

En el siguiente apartado se procede a explicar el proceso de configuración de la plataforma Firebase para que tenga comunicación con el backend de Sigfox. Lo primero es realizar un proceso de registro y crear una cuenta en Firebase. Lo más rápido es accediendo desde la consola de Firebase, escribiendo esto en el buscador de Google. Es necesario tener activa una cuenta de Google, sino deberemos crear una.

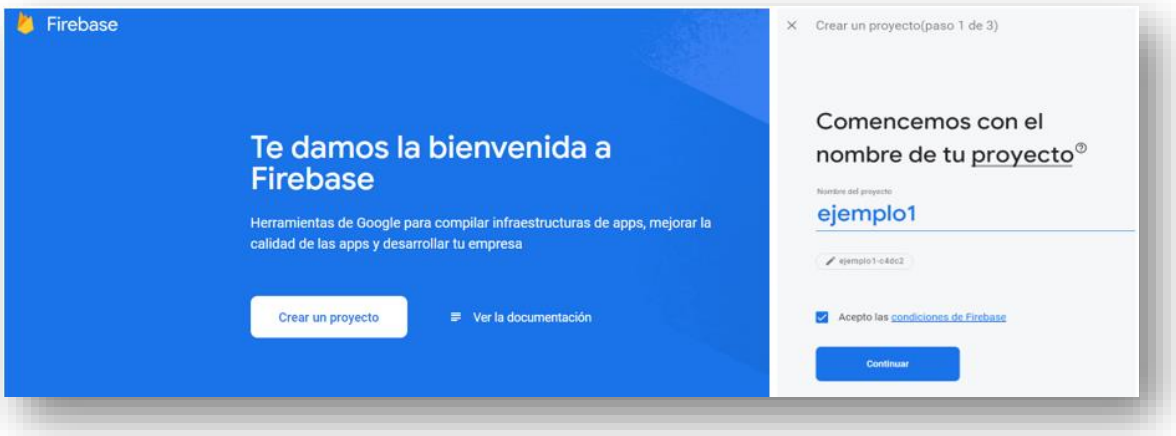


Figura 24. Consola de Firebase y creación de nuevo proyecto.

La imagen anterior es la que nos aparecerá una vez estemos registrados en la plataforma y seleccionemos crear nuevo proyecto. Una vez creado el proyecto volveremos a la consola de Firebase, la cual nos mostrara nuestro proyecto y las diversas funciones de las que disponemos.

Nos centraremos en la pestaña de Compilación. Una vez desplegada esta sección, encontraremos la opción Realtime Database, accederemos para crear una base de datos.

Figura 25. Opciones del nuevo proyecto.

En la imagen superior podemos ver que una de las opciones que se ofrecen en primer plano es incluir Firebase en nuestra futura aplicación, información que más adelante se tendrá en cuenta.

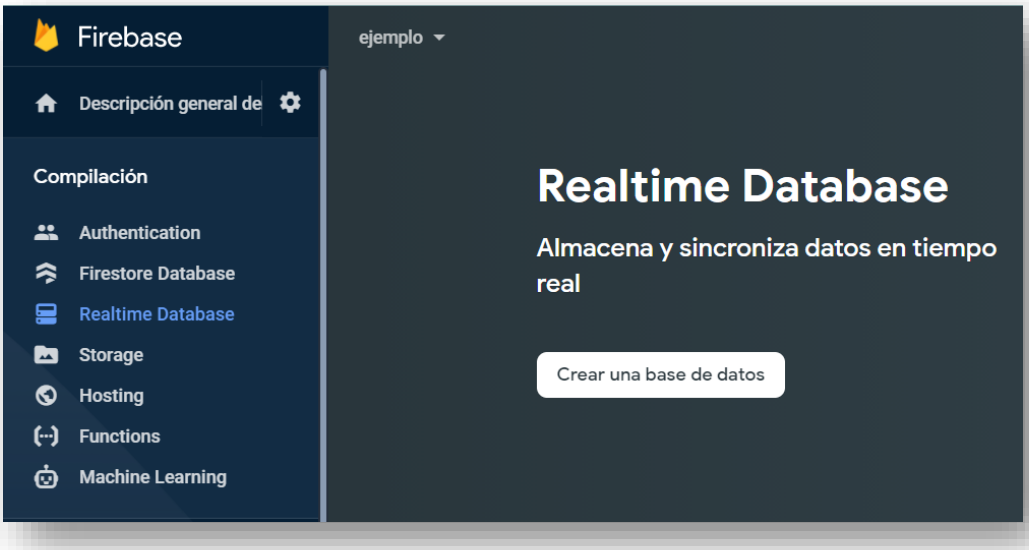


Figura 26. Realtime Database. Creación de la base de datos.

Al seleccionar crear la base de datos nos preguntara donde queremos que se guarden los datos que vamos a subir y las opciones de privacidad.

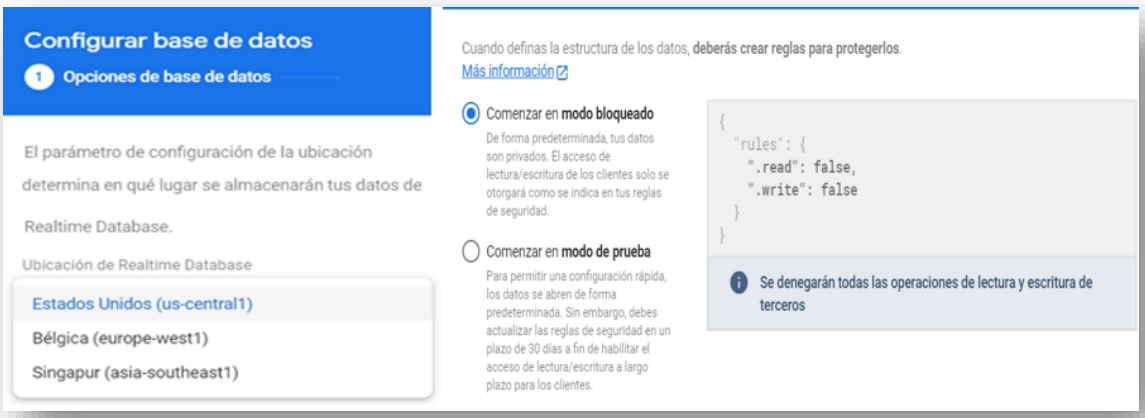


Figura 26. Realtime Database. Almacén de datos y opciones de privacidad.

Una vez cumplimentados estos pasos, tendremos nuestra base de datos creada. La base de datos de Realtime funciona con formato Json para el intercambio de datos, pero no necesitaremos por ahora crear los campos donde almacenaremos los valores del sensor. Centraremos la atención en la dirección http que nos proporciona Firebase.

Figura 27. Realtime Database. Base de datos en formato Json.

Copiaremos este enlace y lo reservaremos para el momento de configurar el backend de Sigfox para que establezca comunicación con Firebase. Ahora volvemos al backend de Sigfox y procedemos a configurar el apartado de CALLBACKS para transmitir los datos a nuestra base de datos.

De vuelta al backend de Sigfox, accedemos a la sección de Device Type, pulsamos en el nombre de nuestro dispositivo y entraremos a la subsección de CALLBACKS, desde donde podemos crear un nuevo CALLBACK o editar uno ya creado. Esta función, la cual es una petición http, nos permite retransmitir el mensaje recibido a otra plataforma o servidor. Es automática, se activa con cada recepción de mensaje.

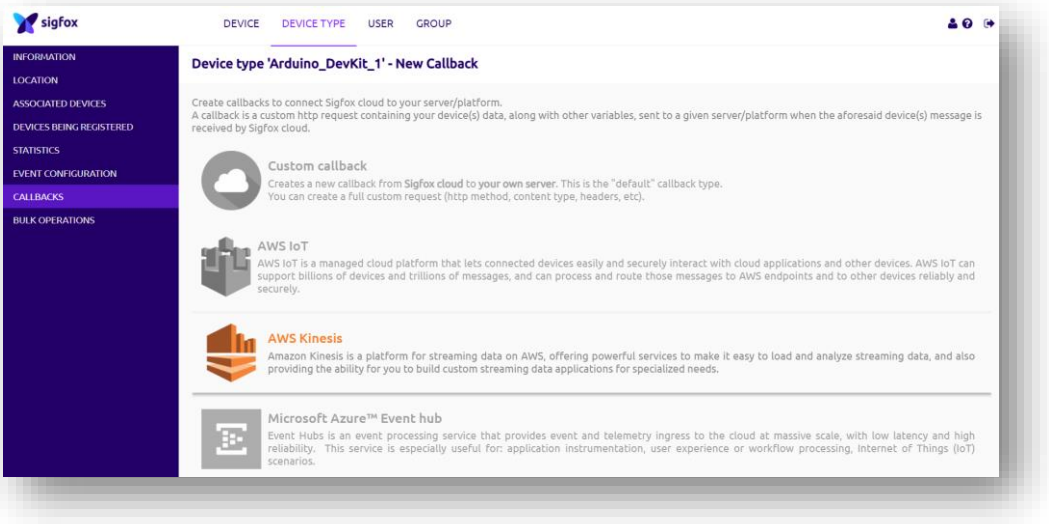


Figura 28. Backend de Sigfox. Configuración de CALLBACKS.

Escogeremos la opción de CALLBACK personalizado. Procederemos a configurar el envío para que el dato se transmita a la base de datos de Firebase.

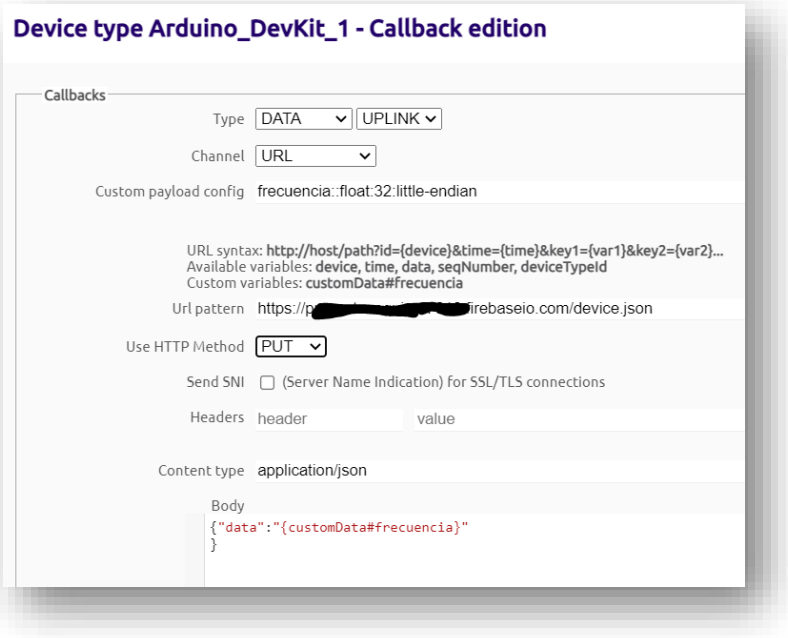


Figura 29. Backend de Sigfox. Edición de CALLBACKS.

Como podemos observar en la imagen anterior, la configuración es un proceso sencillo. A continuación, detallo brevemente cada una de las opciones que nos permite configurar esta sección:

1. **Tipo.** Podemos seleccionar entre DATA, donde podremos cargar variables las cuales serán sustituidas por su valor, SERVICE, para mensajes operativos y ERROR, para notificar la perdida de comunicación entre el dispositivo y el backend de Sigfox. En nuestro caso elegiremos la opción de DATA, ya que lo que pretendemos es transmitir el valor de una variable. Seleccionaremos en este mismo apartado la opción UPLINK.
2. **Canal.** En este apartado escogeremos la opción de direccionamiento mediante URL. Otras opciones son BATH URL y EMAIL, para direccionar el CALLBACK a una dirección de correo electrónico. En este apartado aparece reflejado el formato de texto de los mensajes, que se utilizara en la transmisión del CALLBACK.
3. **Tipo de método HTTP.** Podemos escoger entre POST, GET y PUT. Utilizaremos la opción PUT ya que esta función actualiza el dato existente en el receptor.
4. **Enviar SNI.** Habilitamos esta función en caso de indicar el nombre de Host. En nuestro caso no es necesario.
5. **Tipo del contenido.** En este apartado seleccionamos transmitir los datos en formato Json y creamos el cuerpo del mensaje, que aparecerá de la siguiente forma: {"data":"{customData#frecuencia}"
   }

Por último, guardamos los cambios y ya tendremos configurada la transmisión del mensaje a la base de datos de Firebase. Solo nos queda probar que la transmisión es correcta.

Tras un primer intento el resultado fue negativo. No obtuvimos datos en Firebase. Tocaba revisar la configuración de ambos sistemas y ver porque el valor que aparecía en la base de datos era “null”. En una primera revisión al backend de Sigfox se pudo verificar que el icono de CALLBACK aparecía en color rojo, lo que indicaba que la transmisión del mensaje había sido errónea.

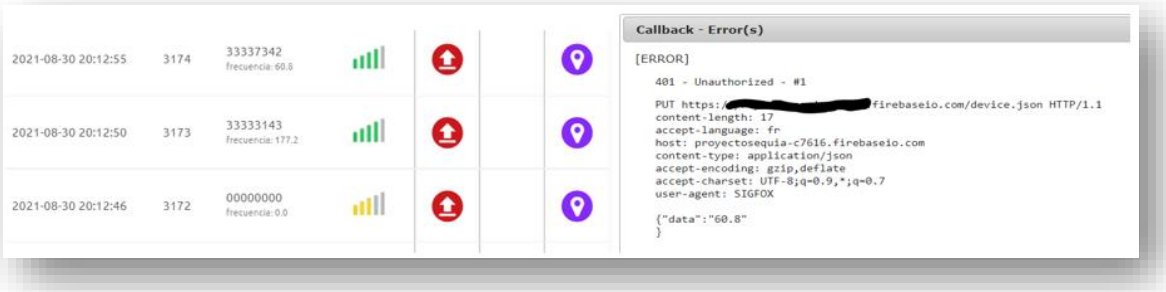
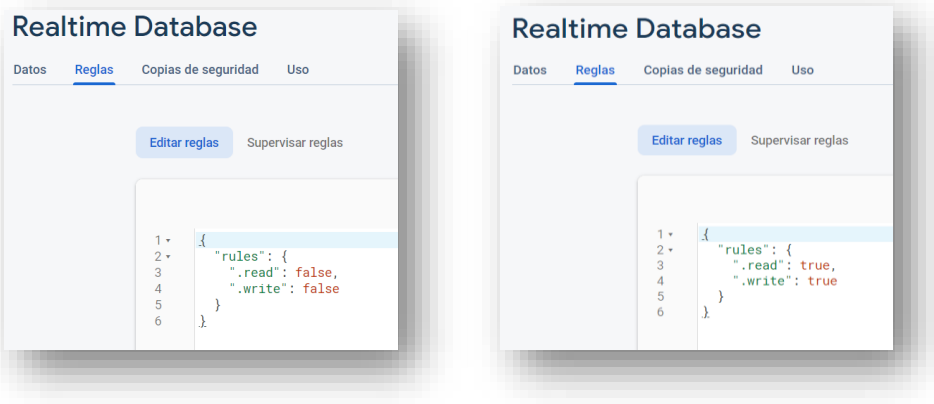


Figura 30. Backend de Sigfox. Mensaje de envío fallido.

En el mensaje de error podemos ver que se incluye el motivo de fallo. El mensaje no estaba autorizado. Encontré el fallo en la configuración de la privacidad de la base de datos. Al crear la base de datos en Realtime Database, había seleccionado esta opción como uso privado y la plataforma estaba rechazando la escritura y la lectura.



Figuras 31 y 32. Base de datos de Firebase. Habilitamos lectura y escritura.

Tras habilitar la lectura y la escritura, realizamos otro proceso de pruebas y el resultado fue satisfactorio. Finalmente teníamos el dato de la frecuencia en la base de datos. Ahora nos restaba enviar este dato desde la base de datos de Firebase hasta la aplicación Android.

Figura 33. Dato transmitido desde el sensor hasta Firebase.

4.5 Firebase y Android

El primer paso en esta sección será crear un nuevo proyecto en el programa de diseño de aplicaciones para dispositivos móviles, Android Studio. Después de investigar diversos ejemplos y tutoriales, se llegó a la conclusión que, para implementar la parte referente a la recogida y visualización de los datos provenientes del sensor, el modelo más adecuado sería un mapa de Google. Implementar este modelo es un proceso sencillo. Directamente, en el proceso de creación del nuevo proyecto, podemos escoger como punto de partida una plantilla de mapa de Google. Si elegimos esta opción, automáticamente nos cargará una configuración que nos permitirá utilizar la API de Google Maps. Si que deberemos seguir unas instrucciones para conseguir la API key de Google. Estos pasos los podemos encontrar en el archivo "Google_maps_api.xml", el cual se carga directamente en nuestro proyecto al escoger como plantilla de inicio la opción de mapa de Google.

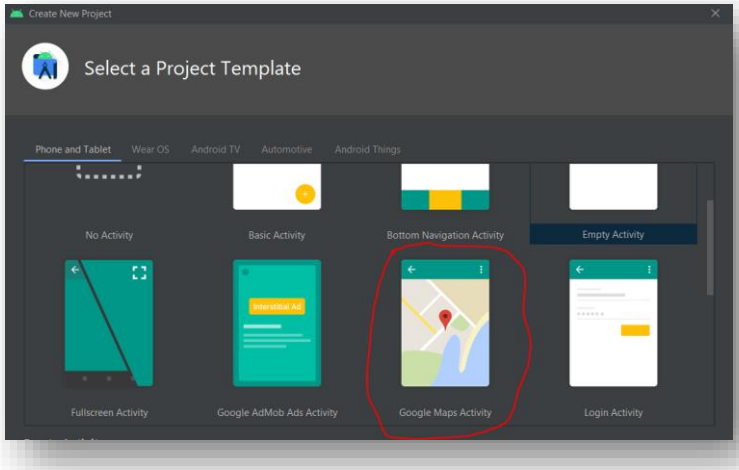


Figura 34. Selección de plantilla en el proceso de creación de un nuevo proyecto Android.

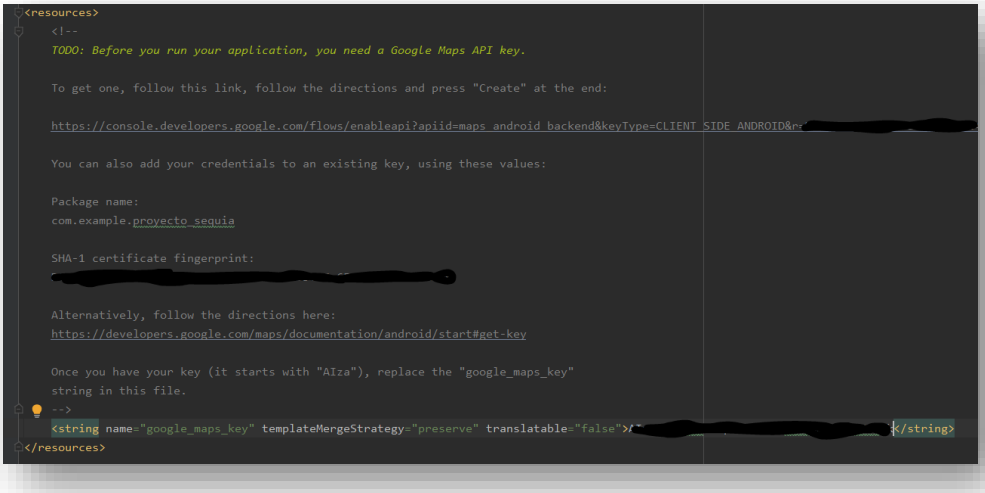


Figura 35. Instrucciones para conseguir la API key de Google maps.

Debemos seguir los pasos que nos indican en el enlace. Primero, crearemos una cuenta en Google Cloud Plattform. Una vez configurada la cuenta para que trabaje con nuestro proyecto, habilitaremos el SDK para Android y obtendremos la API Key, la cual debemos introducir en el archivo "Google_maps_api.xml".

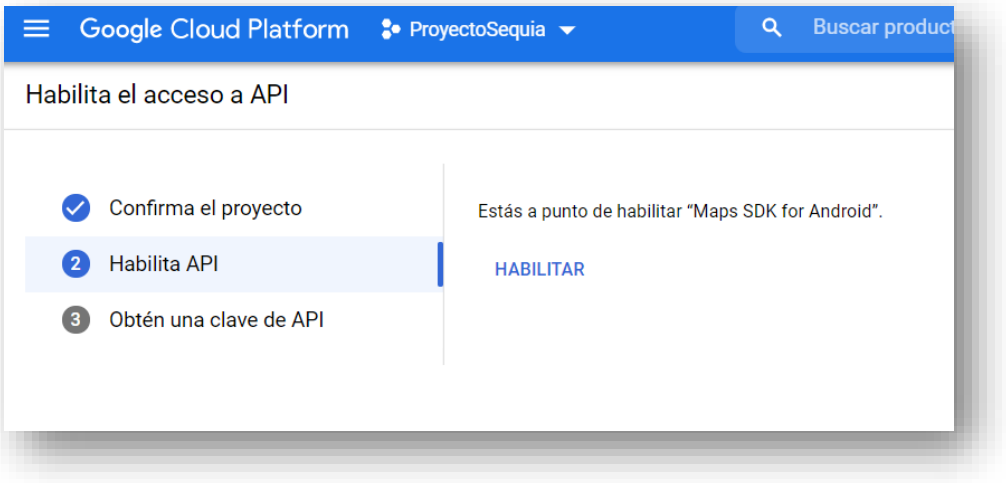


Figura 36. Instrucciones para conseguir la API key de Google maps.

Al escoger como plantilla de inicio la opción de Google maps activity, también se cargarán por defecto dos códigos para hacer funcionar la aplicación. Uno será el archivo "activity_maps.xml", con el código xml de lo que será la interfaz de usuario, más otro archivo, "Maps_activity.java", con la clase que implementa la configuración del mapa de Google. El resultado, una vez se han completado todos los pasos de configuración, ya que no tenemos que introducir todavía código extra, es que la aplicación muestra un mapa de Google con una ubicación, la cual hemos configurado en el código java. Las instrucciones encargadas de implementar el mapa son las siguientes:

```
package com.example.proyecto_sequia;
```

Paquete con el nombre del proyecto.

```
import androidx.annotation.NonNull;
import androidx.fragment.app.FragmentActivity;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;
import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.SupportMapFragment;
import com.google.android.gms.maps.model.BitmapDescriptorFactory;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.Marker;
import com.google.android.gms.maps.model.MarkerOptions;
```

Librerías que se incluyen al configurar el proyecto como mapa de Google.

```
public class MapsActivity extends FragmentActivity implements
OnMapReadyCallback, GoogleMap.OnMarkerClickListener  {
    private GoogleMap mMap;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_maps);
    // Obtain the SupportMapFragment and get notified when the map is ready
to be used.
    SupportMapFragment mapFragment = (SupportMapFragment)
getSupportFragmentManager()
            .findFragmentById(R.id.map);
    mapFragment.getMapAsync(this);
```

Clase que implementa la llamada al mapa de Google.

```
@Override
public void onMapReady(GoogleMap googleMap) {
    mMap = googleMap;
     LatLng foios = new LatLng(39.536, -0.334);
    mMap.addMarker(new MarkerOptions().position(foios).title());
    mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
```

Método para crear un marcador o ubicación en el mapa de Google. Cargamos en el objeto de la clase LatLng las coordenadas del punto que queremos observar en el mapa. Posteriormente se introducirán como valores las coordenadas de cada dispositivo en funcionamiento. La función "setMapType" nos permite elegir el tipo de visualización del mapa, en este caso tipo satélite.

Hasta este punto llegaría la configuración de la aplicación para que trabaje con Google Maps. A continuación, debemos encontrar el sistema adecuado para que la aplicación solicite los datos a Firebase.

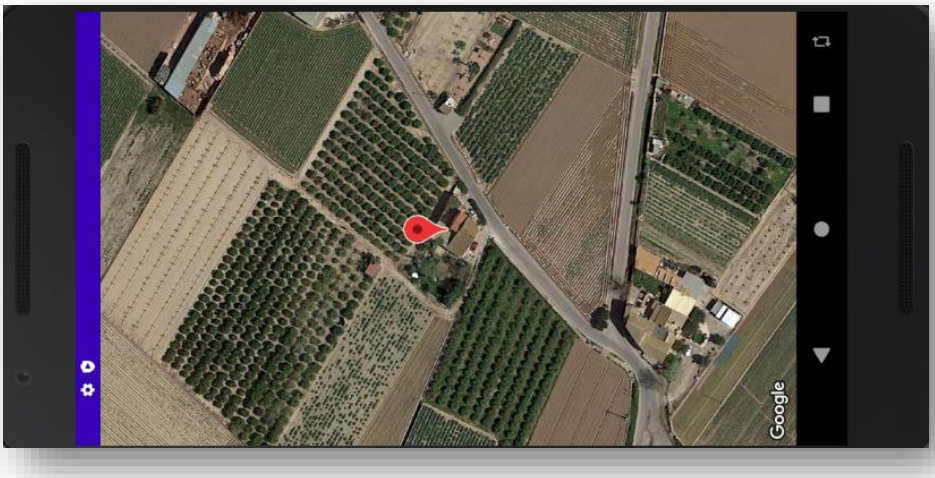


Figura 37. Mapa de Google con ubicación en la aplicación Android.

Al buscar información por la red al respecto del funcionamiento de Firebase con Android Studio, pudimos comprobar que el proceso se había vuelto mucho más rápido y sencillo. Las últimas versiones de Android Studio incluyen entre sus opciones la posibilidad de trabajar con Firebase en tus nuevos proyectos. De esta manera, se facilita mucho la configuración. En la pestaña de Herramientas, en el entorno de trabajo de Android Studio, encontramos la opción de usar Firebase en nuestro proyecto.

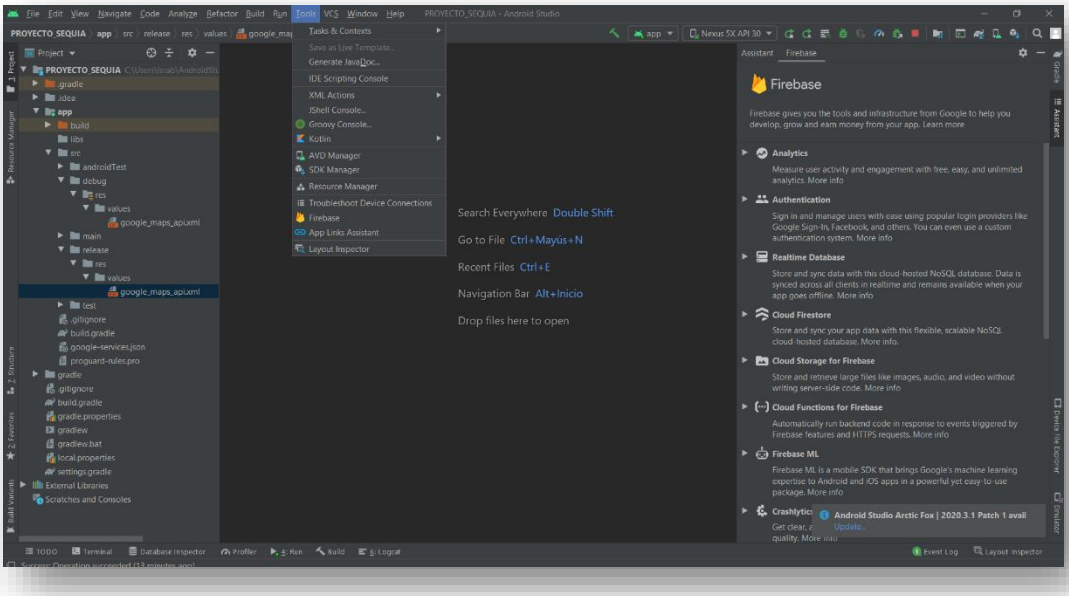


Figura 37. Herramienta Firebase incluida en Android Studio.

Si pulsamos en la pestaña de Realtime Database, encontraremos una serie de ítems para configurar el proyecto con Firebase. También se incluyen los comandos necesarios para poder realizar peticiones de lectura y escritura a la base de datos.

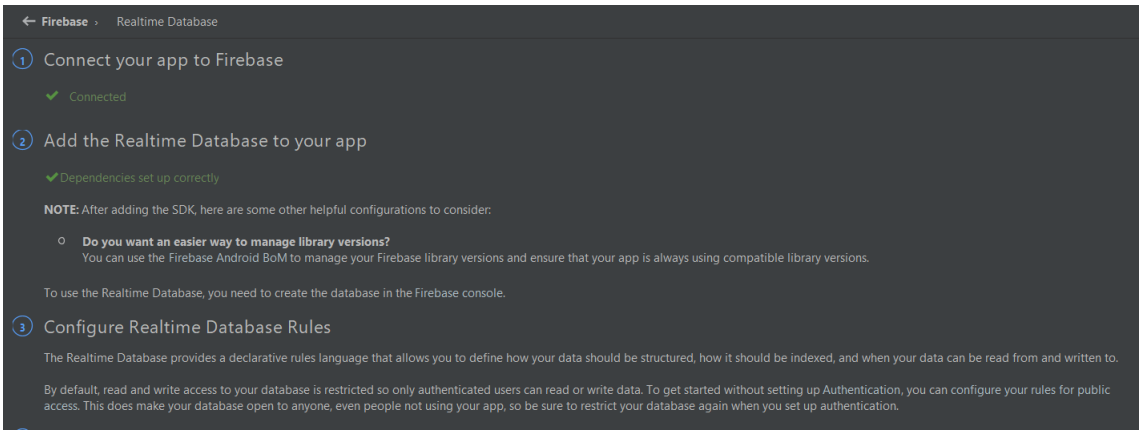


Figura 38. Opciones de configuración de Firebase en Android Studio.

Se deberá configurar la parte de Firebase para obtener conectividad con Android Studio. Proceso sencillo, ya que Firebase nos ofrece configurar el proyecto para que funcione con aplicaciones que estemos desarrollando. Tendremos que configurar el SDK de Firebase para que trabaje con nuestro proyecto de Android y descargar el paquete Google-services.json, el cual es específico para el proyecto con el que estamos trabajando.

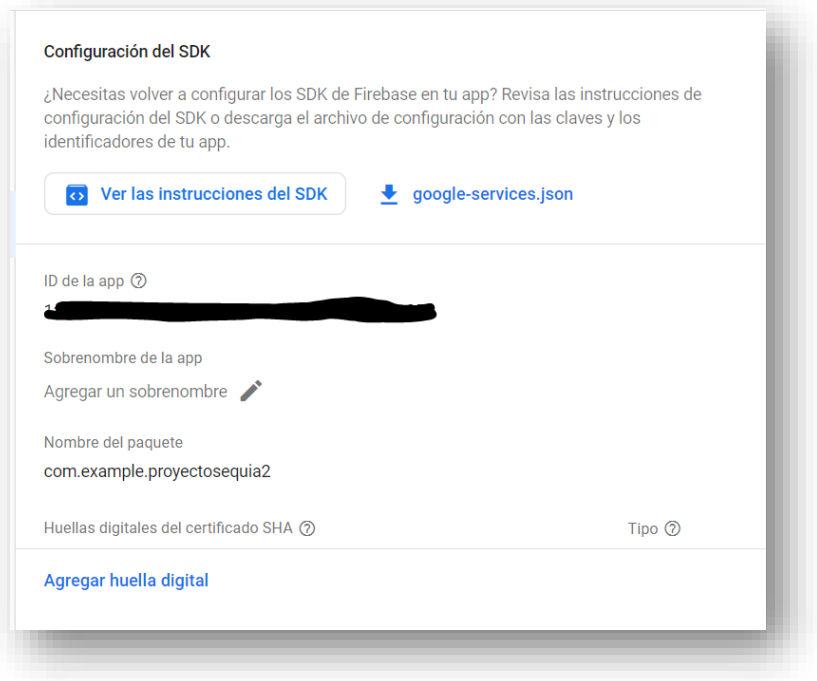


Figura 39. Configuración del SDK de Firebase para vincularlo al proyecto de Android.

La idea para esta parte de la aplicación es que el programa solicite el dato de frecuencia a Firebase y según el valor del dato, cree un aviso de si existe o no agua de riego. Se ha decidido utilizar el mismo icono de ubicación que utiliza la plantilla por defecto, pero en caso de agua de riego, presentarlo de color azul. En caso de no existir agua de riego, presentarlo de color rojo. En ambos casos, al pulsar sobre el icono aparecerá un cuadro de texto con el aviso. El código utilizado para poder extraer el dato de Firebase es el siguiente:

```
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;
```

Librerías que debemos incorporar al código anterior.

```
public class MapsActivity extends FragmentActivity implements
OnMapReadyCallback, GoogleMap.OnMarkerClickListener  {
    private GoogleMap mMap;
    private DatabaseReference bbdd;
    private String frequency;
    private Marker markerPrueba;
```

Dentro de la clase MapsActivity anterior, crearemos la instancia bbdd de la clase DatabaseReference, para leer o escribir datos en una ubicación especifica de la base de datos. Crearemos también el objeto markerPrueba, de la clase Marker, para crear nuestro propio marcador de ubicación en el mapa. En la variable frecuency cargaremos el valor del dato recogido de Firebase.

Dentro del método oncreate, añadiremos la siguiente instrucción para recuperar la instancia de nuestra base de datos.

```
bbdd = FirebaseDatabase.getInstance().getReference();
```

Finalmente, dentro del método onMapReady, donde se creaba el mapa de Google, incluiremos las siguientes instrucciones:

```
bbdd.child("device").addValueEventListener(new ValueEventListener() {
```

Con esta instrucción nos aseguramos de que los datos se actualicen en cada petición.

```
bbdd.child("device").addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        if(snapshot.exists()){
            frequency = snapshot.child("data").getValue(String.class);
```

Método que implementa la lectura en la base de datos. Buscará en el apartado “device” el valor del dato “data” y lo cargará en la variable frecuency.

```
float f1 = Float.parseFloat(frequency);
```

Puesto que la variable frecuency se recoge como String, debemos convertirla a valor numérico.

```
if (f1<20){
    LatLng foios2 = new LatLng(39.53502, -0.33988);
    markerPrueba = googleMap.addMarker(new
MarkerOptions().position(foios2).title("Dispositivo 1 ").snippet("No hay agua
de riego")
.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED))
);
    mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(foios2,18));
}
else {
    LatLng foios2 = new LatLng(39.53502, -0.33988);
    markerPrueba = googleMap.addMarker(new
MarkerOptions().position(foios2).title("Dispositivo 1 ").snippet("Agua de
riego disponible")
.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_BLUE)
));
    mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(foios2,18));
```

En estas líneas implementamos el algoritmo que decidirá si existe o no agua de riego. Es realmente sencillo, si el valor de la frecuencia obtenida es inferior a cierto valor, no existe agua de riego y el icono de ubicación aparecerá de color rojo. Pulsando sobre este icono aparecerá el aviso de que no existe agua de riego.

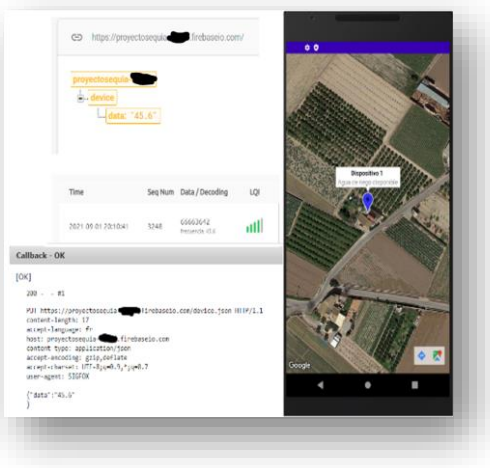
Después de realizar diversas pruebas de campo, se ha llegado a la conclusión de que existe una clara diferencia entre la frecuencia del agua para riego y la obtenida con la que no sirve para regar, como la que puede quedar tras el riego. Se ha comprobado que los valores de frecuencia obtenidos cuando el agua transcurre con velocidad moderada, siendo agua de riego, no bajan de 30 Hz, por lo tanto, se ha decidido que el valor que defina el límite tiene que encontrarse un poco por debajo de 30 Hz, dejando un pequeño margen. Fijaremos el valor en 20 Hz.

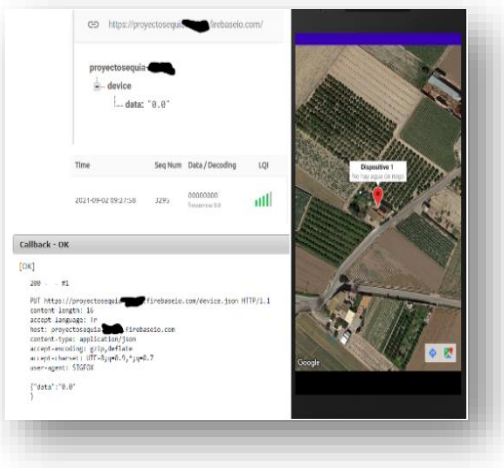
```
@Override
public void onCancelled(@NonNull DatabaseError error) {

}
```

Por último, implementamos un método de cancelación en caso de error con la base de datos.

Solo nos queda probar si la configuración es correcta. En caso afirmativo, el dato habría viajado desde su posición en campo, hasta la aplicación móvil, pasando por el backend de Sigfox y la base de datos de Firebase.





Figuras 40 y 41. Funcionamiento correcto de la aplicación.

La configuración ha sido exitosa. El dato ha viajado entre los diferentes sistemas y ha sido manipulado correctamente por el algoritmo. Se puede apreciar al ejecutar la aplicación que el estado del icono se actualiza en tiempo real. El tiempo de actualización definitivo será cada 10 minutos, por cuestiones de ahorro de batería.

Conseguido esto, tendríamos superada la parte importante del experimento. El siguiente paso, antes de desarrollar la parte de la configuración de las otras funciones de la aplicación, será el proceso para configurar el sistema para que trabaje con un grupo de dispositivos, suponiendo el caso de monitorización de un entorno similar al utilizado por una comunidad de regantes.

Explicare las modificaciones que hay que implementar en cada sistema de los utilizados, separadas en secciones.

1. **Arduino.** En este caso, las modificaciones consisten en añadir más dispositivos. Cada dispositivo es autónomo, no necesitan control central.
2. **Sigfox.** Crearemos un grupo de dispositivos. Cada dispositivo aparecerá incluido dentro de este grupo, identificado con su ID y su PAC.

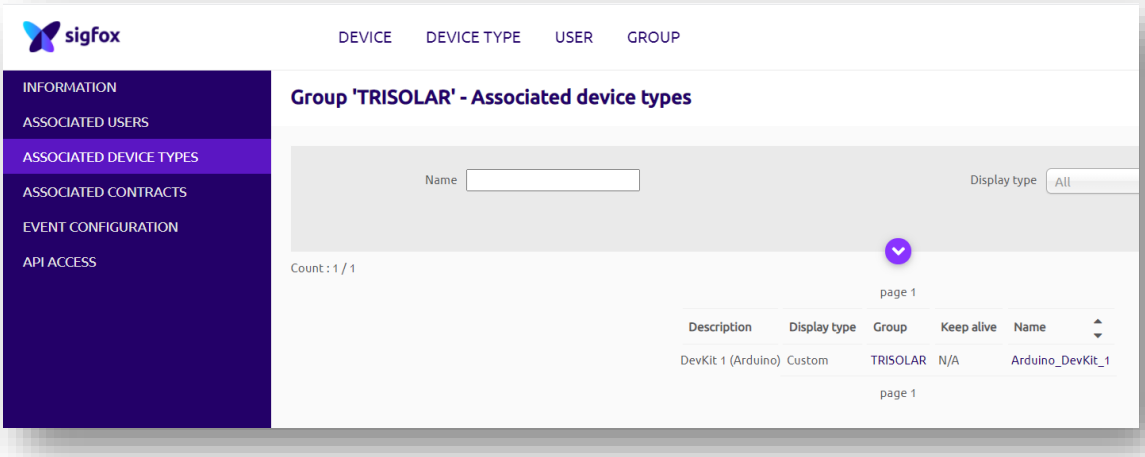


Figura 42. Creación de un grupo dentro del backend de Sigfox.

Deberemos configurar tantos CALLBACKS como dispositivos vayamos a usar en la nueva configuración grupal. Cada CALLBACK nombrará el dispositivo con su nombre o número de identificación. Cambiaremos el nombre resaltado en la siguiente dirección para crear tantos objetos en la base de datos como dispositivos usados.

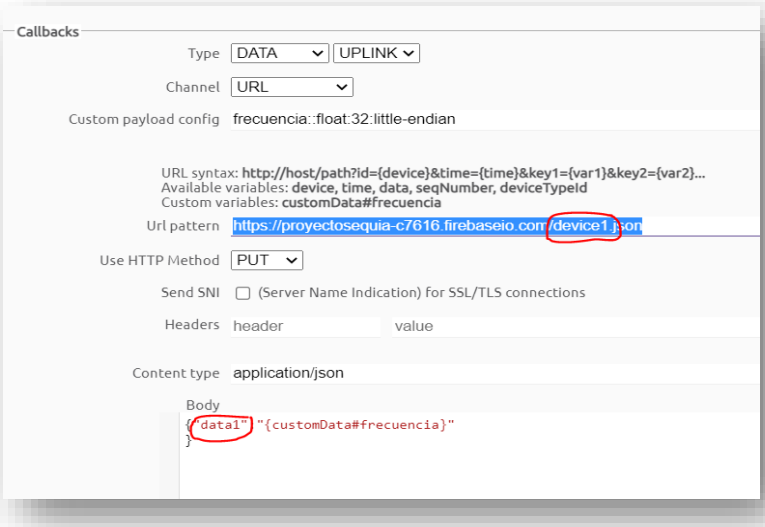


Figura 43. Configuración de CALLBACKS.

3. **Firebase.** Al configurar los CALLBACKS de Sigfox, automáticamente se refresca nuestra base de datos y aparecen todos los dispositivos asociados.

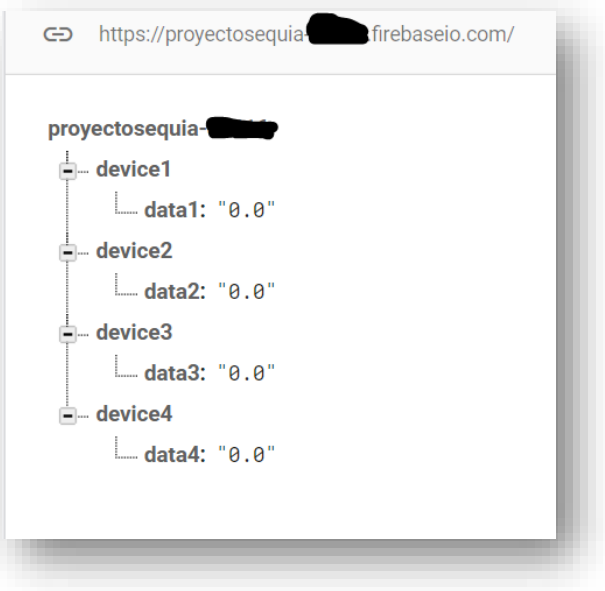


Figura 44. Árbol de la base de datos actualizada.

4. **Android.** Para este apartado cambiaremos el programa anterior para que trabaje con grupos de dispositivos. A continuación, las modificaciones realizadas en el código:

```
double latitud  = 0;
double longitud = 0;
int i = 0;
int NumeroDeDispositivos = 4;
String NombreDevice = "device";
```

Crearemos unas variables para almacenar las coordenadas de cada dispositivo.

```
while (i < NumeroDeDispositivos){
    String NumDevice = String.valueOf(i);
    String Dispositivo = NombreDevice + NumDevice;
```

Utilizaremos la sentencia while para que el código anterior se ejecute tantas veces como dispositivos existan en nuestra red.

```
switch (i){
    case 0:
       latitud = 39.536;
       longitud = -0.3396;
        break;
    case 1:
        latitud = 39.536119;
        longitud = -0.337856;
        break;
    case 2:
        latitud = 39.535399;
        longitud = -0.341547;
        break;
    case 3:
        latitud = 39.536946;
        longitud = -0.339798;
        break;
}
```

Dentro del while utilizaremos la sentencia swich case para cargar las coordenadas en función del valor de la variable i. El resto de las instrucciones que se ejecutaran pertenecen al código anterior. Serán las encargadas de buscar los valores de cada dispositivo y crear los marcadores. Puesto que se encuentran dentro del bucle while, se crearán tantos marcadores como dispositivos existan en la red. Cada marcador se actualizará cada 10 minutos, que será la frecuencia de envío de mensajes desde el sensor hasta la aplicación.

```
LatLng latlong = new LatLng(latitud, longitud);
bbdd.child(Dispositivo).addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot snapshot) {
        if (snapshot.exists()) {
        // Resto de instrucciones del codigo anterior
 } i++;
```

Podemos observar como ahora pasamos el String Dispositivo como referencia para buscar en la base de datos. Esta instrucción mantendrá actualizado el valor del dato de cada dispositivo.

Finalmente, compilamos el código para ver el resultado y verificamos que el funcionamiento es el adecuado. Solo resta incorporar las otras funcionalidades de la aplicación, como un chat para los usuarios y un apartado para avisos y mensajes de la comunidad de regantes.

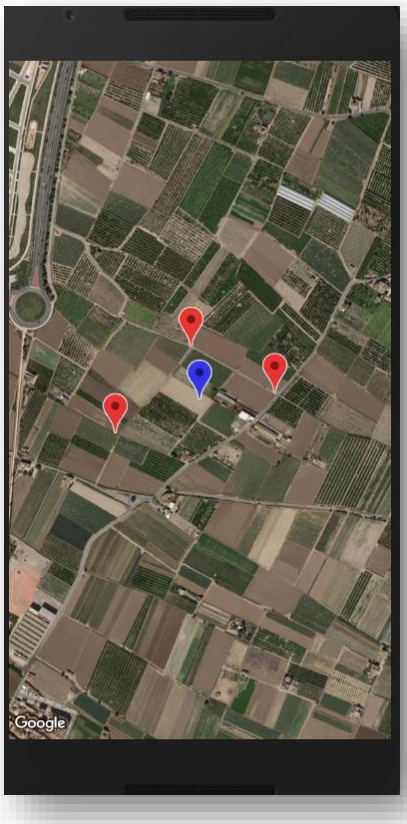


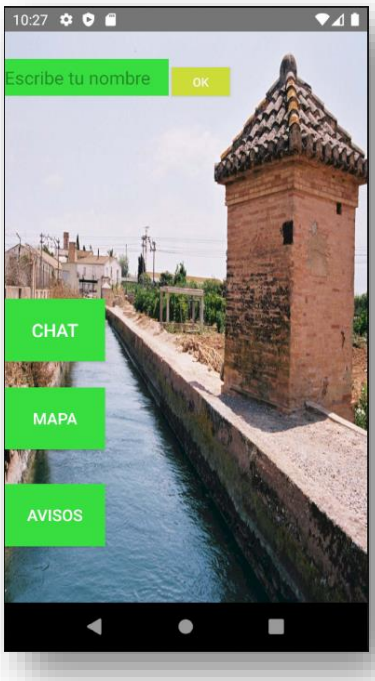
Figura 44. Simulación de la aplicación.

4.6 Aplicación móvil

En esta sección detallaremos el proceso de creación de una función de chat y una zona de avisos de la compañía directiva, administradores, etc. Investigando las ventajas que ofrece Firebase, encontramos entre sus funciones la posibilidad de crear una aplicación de chat en un proyecto de Android Studio, así que trabajaremos sobre este modelo. También crearemos aquí lo que será la visualización inicial de la aplicación, desde donde podremos llamar a las otras funciones. Firebase también será la base de datos que utilizaremos para la sección de avisos. La aplicación finalmente estará formada por las siguientes Activities o secciones:

- **Pantalla de inicio**. Sección de acceso a las funciones principales.
- **Pantalla de bienvenida.** Breve explicación de la funcionalidad de la aplicación.
- **Pantalla de Mapa.** En este mapa se ubicarán los puntos de riego.
- **Pantalla de Chat.** Sección de mensajería instantánea entre usuarios.
- **Pantalla de Avisos.** Aquí encontraremos los avisos que se introduzcan en la base de datos de Firebase.

Comenzaremos explicando la adición de la pantalla de inicio de la aplicación. Esta pantalla dará paso a las diferentes partes de la aplicación. Consta de tres botones de acceso a las diversas funciones, más un cuadro de texto para incluir el nombre, junto con un botón de aceptar, el cual nos traslada a la pantalla de bienvenida.



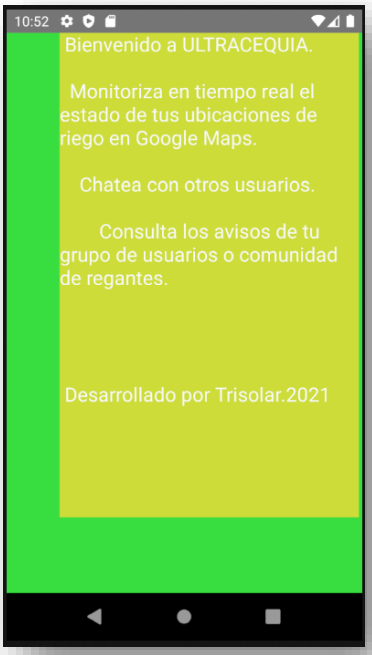


Figura 45 y 46. Pantallas de inicio y bienvenida.

Como el código utilizado en estas secciones es extenso, se ha decidido incluirlo todo en un anexo al final del documento. En este anexo, se incluirá tanto la parte del código java como la parte xml.

Respecto a esta sección, diseño sencillo y funcional. Al iniciar la aplicación los botones de CHAT, MAPA y AVISOS están deshabilitados, tan solo se podrá acceder a estas secciones al introducir el nombre y pulsar OK. Al pulsar este botón se accede a la sección de bienvenida y se habilitan todos los botones. Cuando se retorna de la pantalla de bienvenida a la pantalla de inicio, se ha incluido el nombre escrito más un mensaje de bienvenida. Una vez realizado este proceso, se deshabilita el botón de OK. Cada botón de la pantalla de inicio tiene asociado un método setOnClickListener que se queda a la escucha de que el botón registre actividad. Cuando pulsamos cada botón, se inicia la sección pertinente utilizando un objeto de la clase Intent, con funciones para comunicación entre diferentes secciones. Utilizando este mismo objeto, pasamos el nombre introducido a la sección CHAT, para que aparezca cada vez que el usuario envía un mensaje.

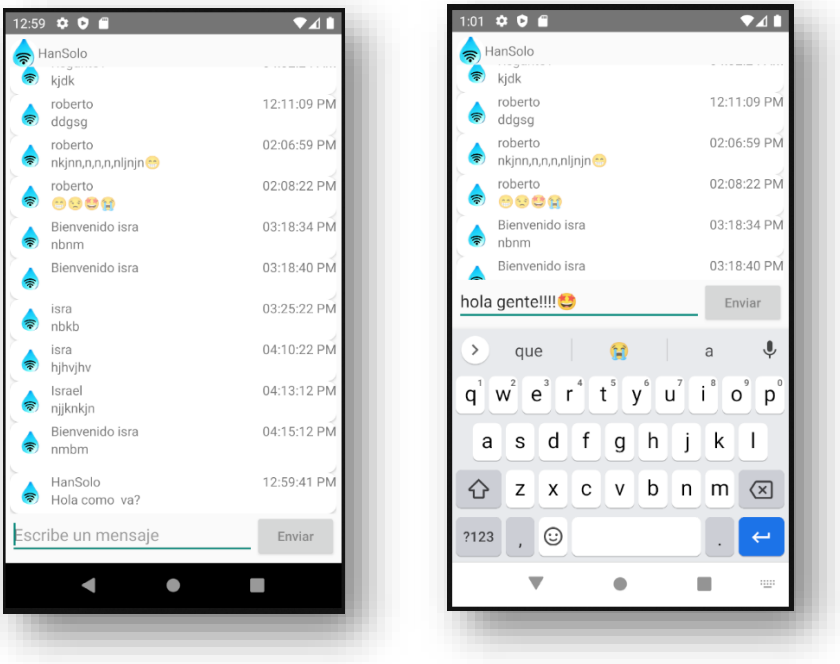
Continuamos con la sección o pantalla de avisos. Esta parte esta pensada para el caso de que la aplicación este supervisada por un órgano central, como una junta administrativa, comunidad de regantes, etc. En esta sección la administración incluiría los avisos a los usuarios o socios. Básicamente, el funcionamiento de esta parte es leer datos de Firebase y mostrarlos por pantalla. Utilizaremos la misma base de datos de Realtime Database utilizada para registrar los datos de los dispositivos sensores.

Figuras 47 y 48. Visualización en la aplicación y nodo CarpetaAvisos de la base de datos de Firebase.

Utilizaremos el método onDataChange de la clase DatabaseReference para extraer los datos de Firebase. Primero cargaremos la función getChildrenCount() para realizar un conteo del total de avisos que existan en la base de datos. Una vez conseguido este dato, implementamos un bucle que muestre en un textView todos los avisos existentes. Por el lado XML, se hace uso de la función android:scrollbars="vertical" para poder desplazar la pantalla verticalmente. También se ha utilizado el comando textNoSuggestions para inhabilitar las sugerencias de corrección, ya que el texto aparecía subrayado en rojo.

Por último, se detalla el proceso para implementar la función de mensajería entre usuarios. Este proceso ha sido extraído del tutorial de Youtube “Crear Un Chat En Menos De Una Hora | Firebase y Android Studio” del autor KAD. Podemos encontrar el enlace para explorar el contenido en el apartado de bibliografía. Esta función permitirá el intercambio de mensajes en tiempo real entre usuarios que tengan instalada esta aplicación.

El proceso creativo empieza por crear lo que será la interfaz de intercambio de mensajes. Crearemos una pantalla de visualización similar a la de que cualquier aplicación Messenger. Incluiremos un layout adicional, llamado card_view_mensajes, que dará forma a lo que serán los datos de usuario en cada mensaje enviado a otro dispositivo, a modo de tarjeta de visita.



Figuras 49 y 50. Visualización de la sección CHAT.

Por otro lado, tenemos la sección con la configuración java, para crear todas las acciones que deberá realizar este apartado. Su función principal será interactuar con la base de datos de Firebase. El programa principal creara en Firebase un nodo, al que llamaremos “chat”, donde se almacenarán los mensajes enviados por los usuarios. Este mismo código recogerá estos mensajes y los mostrará en un cuadro de texto por pantalla. Para ello se deben implementar clases adicionales encargadas de gestionar los datos enviados.

Las partes que forman el programa principal que implementa el chat son las siguientes:

- **MainActivity.** Encargada de crear el almacén en Firebase y de recoger los mensajes de esta misma base de datos. Posee toda la configuración de las acciones de los componentes del diseño xml. Implementa las llamadas a las otras clases necesarias en el proceso de envío y recepción de mensajes.
- **HolderMensaje.** En esta clase configuraremos y almacenaremos los datos de las tarjetas de visita en cada mensaje.
- **Mensaje.** Clase que contendrá la configuración de cada mensaje de usuario. Los parámetros que se incluyen en cada mensaje.
- **AdapterMensaje.** Extenderemos esta clase desde la clase RecyclerView, para mostrar listas de datos con función de actualización (o reciclado). Incluiremos en la extensión la clase **HolderMensaje**. En esta clase configuraremos los objetos de la clase **Mensaje**, como el nombre, texto y hora.
- **MensajeEnviar y MensajeRecibir.** En estas dos clases implementaremos las funciones para enviar la hora en cada mensaje. Utilizaremos un método para conectarnos con el servidor que nos proporcionará la hora correcta.

Hay que tener especial atención a la hora de crear los datos en la base de datos de Firebase, ya que un formato equivocado puede generar conflicto en la recepción de estos en la aplicación Android. Un descuido semejante acarreo dificultades que persistieron varios días hasta que se encontró dónde estaba el problema. Al introducir en Firebase los datos de los sensores manualmente, para realizar pruebas de comunicación entre Firebase y Android, se escribía el valor decimal utilizando un punto, en vez de una coma, con lo que Firebase estaba interpretando los datos con formato double y la aplicación estaba intentando recoger datos con formato String.

Debido a ciertos conflictos entre esta sección y la sección del mapa, opte por reconfigurar el apartado encargado de recoger los valores de los sensores y mostrarlos por el mapa, explicación que he incluido en un anexo al final del documento. Y con esta explicación, termino el apartado de desarrollo del producto bajo estudio.

Capítulo 5. Pliego de condiciones

5.1 Materiales

En esta parte incluiremos todos el material necesario, incluyendo en este termino tanto a los dispositivos físicos como a los sistemas de software utilizados. Se incluirán en cada apartado de materiales las especificaciones técnicas, datasheet, en caso de existir y funcionalidades que ofrece cada producto. El orden de aparición de cada material será según su función en el proceso de la aplicación, empezando con el sensor y terminando con el programa de diseño de aplicaciones Android Studio.

5.1.1 Caudalímetro YF-S201

Como ya se ha explicado en capítulos anteriores, este sensor funciona mediante una turbina accionada por el paso del agua y un sensor hall instalado en la carcasa. Este sensor detecta las variaciones en el campo magnético producido por el imán instalado en la turbina y genera el pulso cuadrado que es enviado por el cable de señal hasta el microcontrolador. Según podemos leer en sus especificaciones, este sensor puede llegar a ofrecer una buena precisión si esta calibrado adecuadamente. Como en el caso que nos ocupa no necesitamos calcular el caudal, tan solo obtener la frecuencia de giro del rotor, no es necesaria una calibración previa. En la imagen inferior podemos encontrar las especificaciones y modos de funcionamiento de este dispositivo.

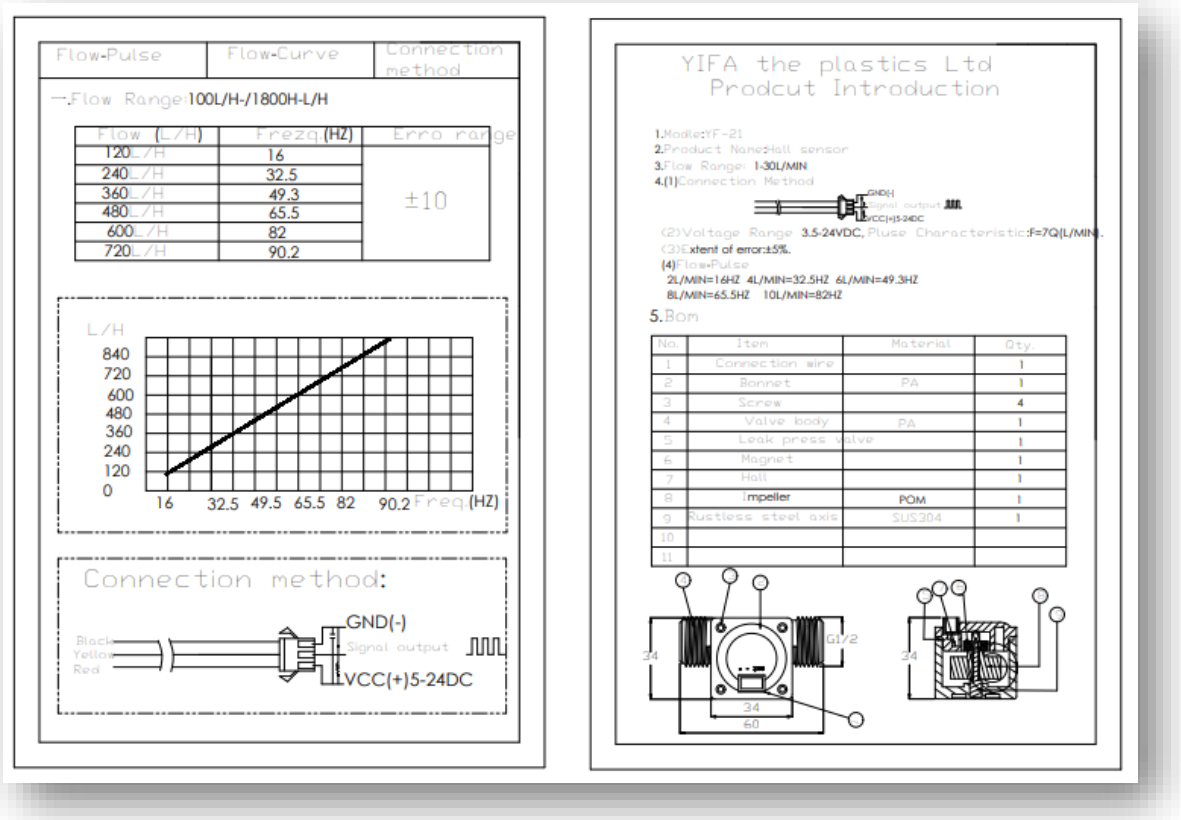


Figura 51. Datasheet del caudalímetro YF-S201.

5.1.2 Placa de desarrollo MKRFOX-1200

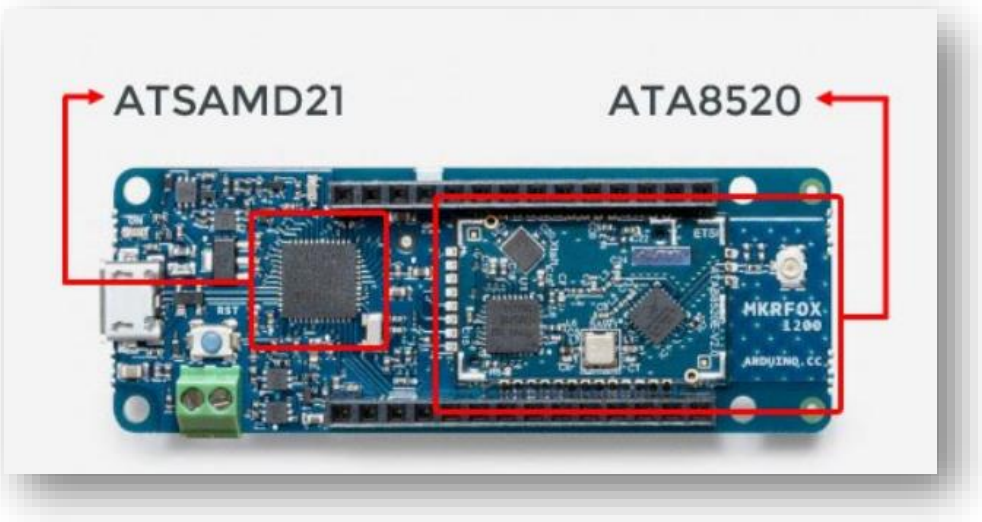


Figura 52. Placa MKRFOX-1200.

Como podemos ver en la imagen, esta placa cuenta con dos chips integrados. Por un lado, tenemos el microcontrolador ATSAMD21, encargado de controlar la interfaz de los pines GPIO y donde cargaremos el programa de trabajo. Luego encontramos el microcontrolador ATA8520 encargado de la conexión con la red de Sigfox. A continuación, incluyo una imagen con las especificaciones técnicas extraídas del datasheet. No se incluye todo el datasheet ya que tiene una extensión de más de 1000 páginas y es fácilmente accesible desde internet.

	SAM D21J	SAM D21G	SAM D21E
Pins	64	48	32
General Purpose I/O-pins (GPIOs)	52	38	26
Flash	256/128/64/32KB	256/128/64/32KB	256/128/64/32KB
SRAM	32/16/8/4KB	32/16/8/4KB	32/16/8/4KB
Timer Counter (TC) instances	5	3	3
Waveform output channels per TC instance	2	2	2
Timer Counter for Control (TCC) instances	3	3	3
Waveform output channels per TCC	8/4/2	8/4/2	6/4/2
DMA channels	12	12	12
USB interface	1	1	1
Serial Communication Interface (SERCOM) instances	6	6	4

Figura 52. Parte de la hoja de especificaciones de la placa MKRFOX-1200.

Serial Communication Interface (SERCOM) instances	6	6	4
Inter-IC Sound (I^2S) interface	1	1	1
Analog-to-Digital Converter (ADC) channels	20	14	10
Analog Comparators (AC)	2	2	2
Digital-to-Analog Converter (DAC) channels	1	1	1
Real-Time Counter (RTC)	Yes	Yes	Yes
RTC alarms	1	1	1
RTC compare values	1 32-bit value or 2 16-bit values	1 32-bit value or 2 16-bit values	1 32-bit value or 2 16-bit values
External Interrupt lines	16	16	16
Peripheral Touch Controller (PTC) X and Y lines	16x16	12x10	10x6
Maximum CPU frequency	48MHz		
Packages	QFN TQFP UFBGA	QFN TQFP WLCSP	QFN TQFP WLCSP
Oscillators	32.768kHz crystal oscillator (XOSC32K) 0.4-32MHz crystal oscillator (XOSC) 32.768kHz internal oscillator (OSC32K) 32kHz ultra-low-power internal oscillator (OSCULP32K) 8MHz high-accuracy internal oscillator (OSC8M) 48MHz Digital Frequency Locked Loop (DFLL48M) 96MHz Fractional Digital Phased Locked Loop (FDPLL96M)		
Event System channels	12	12	12
SW Debug Interface	Yes	Yes	Yes
Watchdog Timer (WDT)	Yes	Yes	Yes

Figura 53. Parte 2 de la hoja de especificaciones de la placa MKRFOX-1200.

Explicaremos brevemente las especificaciones más relevantes de este dispositivo:

- La placa cuenta con 15 pines digitales. Permiten leer o escribir dos estados, HIGH Y LOW. Soportan un voltaje de entrada de 3.3 voltios como límite superior. Hay que tener especial cuidado con este detalle, ya que de aplicar más tensión podemos quemar los pines. El pin 6 lleva incorporado un led integrado, para realización de pruebas.
- Como pines analógicos tenemos un total de 7. Permiten señales de hasta 3.3 voltios. Podemos configurar la resolución de estos pines en 8,10 o 12 bits, consiguiendo hasta un rango de 4095 valores.
- Pines PWM o modulación de ancho de pulso. Tenemos un total de 12 pines PWM, en los que podemos escribir valores analógicos.
- Interrupciones externas. Contamos con 8 pines capaces de soportar esta función.
- BUS SPI. Para la comunicación serie entre dos dispositivos.
- BUS I2C. Prácticamente tiene la misma función que el BUS SPI, pero tiene menos pines dedicados.
- UART. Transmisor-Receptor Asíncrono Universal. Control y gestión de la comunicación serie.

- AREF o pin de voltaje de referencia. Utilizado para designar el valor de referencia para la conversión A/D.
- Además del botón de RESET, la placa cuenta con un pin de reinicio, el cual reinicia el sistema si le damos el valor de 0.
- La placa cuenta con cuatro pines de alimentación, que detallo a continuación:
    - Pin 5V. Este pin proporciona 5 voltios si alimentamos la placa por el puerto USB o por el pin Vin. No cuenta con regulador de tensión. Si alimentamos la placa mediante pilas, este pin proporciona un nivel muy bajo de tensión, por lo que queda fuera de uso.
    - Pin Vin. El único pin de entrada de alimentación de la placa. Funciona en un rango de 5V-6V.
    - Pin Vcc. Con esta salida obtenemos 3.3 voltios regulados.
    - Pin GND. Ground o toma de tierra (0 V).

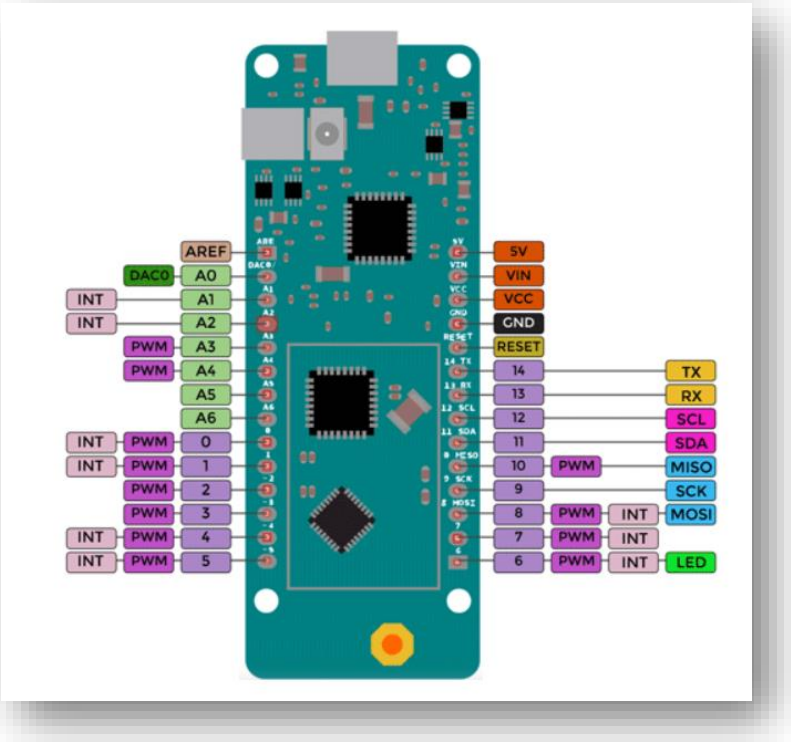


Figura 54. Esquema de pines de la placa MKRFOX-1200.

- Una de las ventajas que ofrece este dispositivo frente a otras placas es que permite la alimentación mediante baterías AA o AAA. Utilizando una configuración de optimización del gasto energético podemos conseguir una autonomía de baterías de seis meses.
- El chip SAMD21 funciona a 32-bit, con una velocidad de 48 MHz. Memoria Flash de 256 Kb y SRAM de 32 Kb.
- Microcontrolador ATA8520. Tiene integrado un microcontrolador AVR con el firmware y módulo de radiofrecuencia. Opera en el rango de 868.0 MHz y 868.6 MHz.

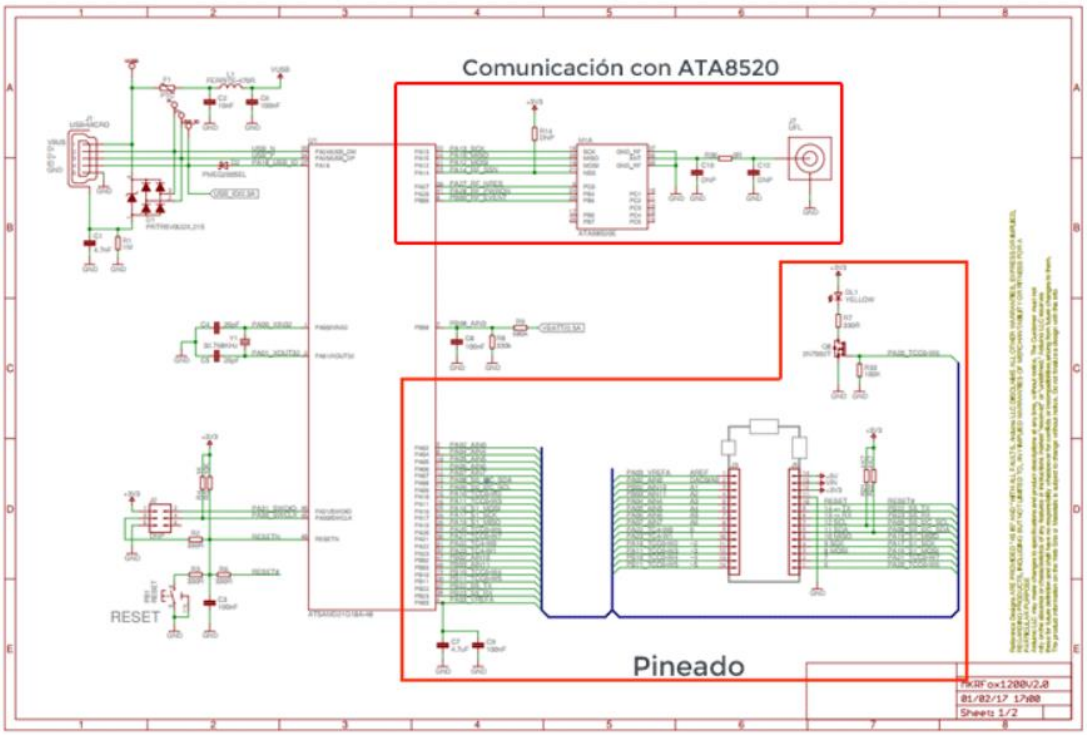


Figura 55. Esquemático de la placa MKRFOX-1200.

5.1.3 Sigfox

El sistema Sigfox se basa en una red de conexión para dispositivos IoT de bajo consumo con independencia de las redes móviles comerciales. Pensada para comunicaciones de baja velocidad. Sigfox ha desarrollado una red de largo alcance para este tipo de dispositivos.

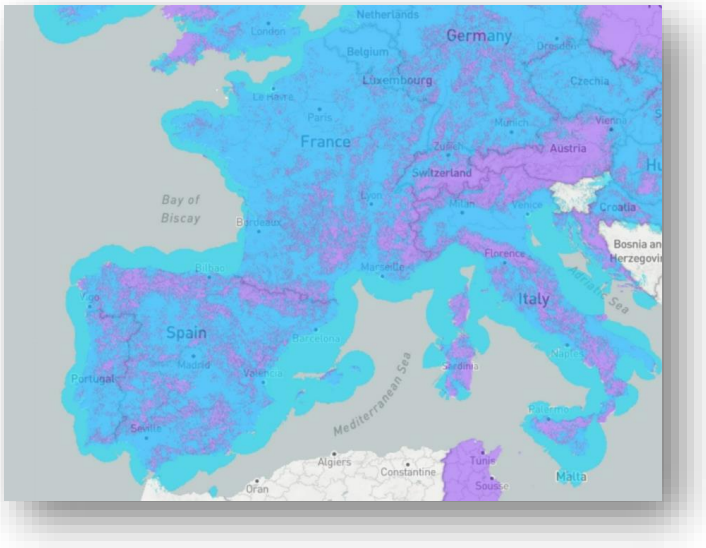


Figura 56. Mapa de cobertura de Sigfox en parte de Europa.

Utiliza la banda de radio conocida como Ultra Narrow Band, en la banda de 868 MHz, con un rango de frecuencias muy reducido (<1 KHz), lo que por otro lado le confiere el largo alcance en las comunicaciones, 5 Km en zona urbana y 25 Km en zonas rurales. Trabaja con frecuencias no licenciadas, como las bandas ISM. Este aspecto repercute en que solo puede ocupar el espectro un tiempo reducido conocido como duty cycle, lo que permite un número de 140 mensajes al día.

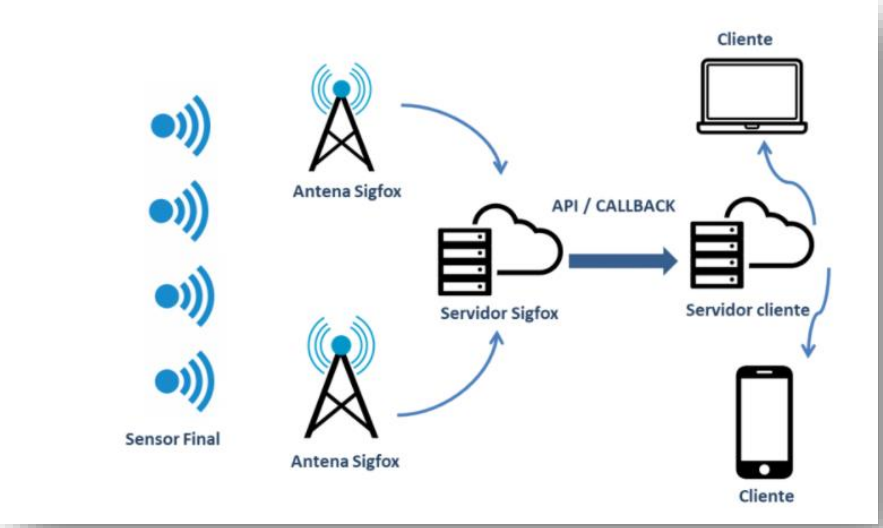


Figura 57. Esquema de funcionamiento de la red Sigfox.

Sigfox posee su propio sistema en la nube donde puedes registrar los dispositivos que van a estar incluidos en la red, mediante su número de identificación o ID, haciendo innecesario el uso de tarjetas SIM. Esta plataforma permite el envío de mensajes desde su sistema en la nube hasta el dispositivo, pero no son mensajes fiables, son asíncronos y no garantiza que sean recibidos. La velocidad de transmisión que ofrece oscila entre 0.3 a 50 Kbyte por segundo, lo cual se puede definir como bastante lenta, pero suficiente para los datos que se van a enviar desde los sensores. Otra de sus principales características es la función de reenvío de los mensajes recibidos a otras plataformas o servidores, empleando para ello la función de CALLBAKS. Estos permiten la configuración del modo de envío y formato de datos. El peso máximo de los mensajes soportado por este sistema no puede superar los 12 bytes. Para el envío de mensajes utiliza modulación PSK y GFSK.

Figura 58. Logotipo de Sigfox.

5.1.4 Firebase

Plataforma de Google para el desarrollo de aplicaciones web y móvil. Nació como base de datos en tiempo real, pero con el tiempo se incluyeron más funcionalidades, como la agrupación de los SDK de productos de Google. Su finalidad principal es facilitar el proceso de creación de aplicaciones. Las funcionalidades más destacadas que ofrece son herramientas para el desarrollo, crecimiento, monetización y análisis de nuestros proyectos. Entre las herramientas de desarrollo encontramos Realtime Database, la base de datos de Firebase en tiempo real. También se encuentran entre estas herramientas: sistema de autenticación de usuarios, almacenamiento en la nube, seguimiento de errores, laboratorio de pruebas, configuración remota, más funciones de cloud messaging y hosting. Por otro lado, cuenta con herramientas para el control del uso de la aplicación por parte de los usuarios, captación de nuevos clientes, estadísticas de crecimiento, publicidad y otras funcionalidades centradas en la parte comercial de los proyectos. A continuación, explicare más detalladamente la herramienta de Firebase, Realtime Database, la base de datos en tiempo real.

Firebase Realtime Database es una base de datos incluida en la nube. Almacena los datos en formato Json y los sincroniza con cada petición de cliente en tiempo real. Entre sus funciones principales se encuentran:

- **Tiempo real.** Sincronización de datos con los dispositivos. Cuando cambian los datos, los dispositivos se actualizan en milisegundos.
- **Sin conexión.** El SDK hace que los datos persistan en el disco. Al restablecerse la conexión, el dispositivo se actualiza.
- **Acceso desde dispositivo cliente.** No es necesario el uso de un servidor de aplicaciones. Reglas de seguridad para trabajar con los datos.
- **Escalamiento en varias BBDD.** Se satisfacen las necesidades de datos masivos.
- Realtime Database es una base de datos **NoSQL**, configurada para aceptar operaciones de alta velocidad.
- Permite el acceso seguro a la base de datos directamente desde el código del cliente.
- Los datos se almacenan persistentemente de forma local.
- Utiliza un lenguaje flexible para configurar las reglas de acceso y permite al cliente diseñar la estructura de los datos.
- Puede ofrecer servicio a millones de usuarios sin afectar a la capacidad de respuesta.

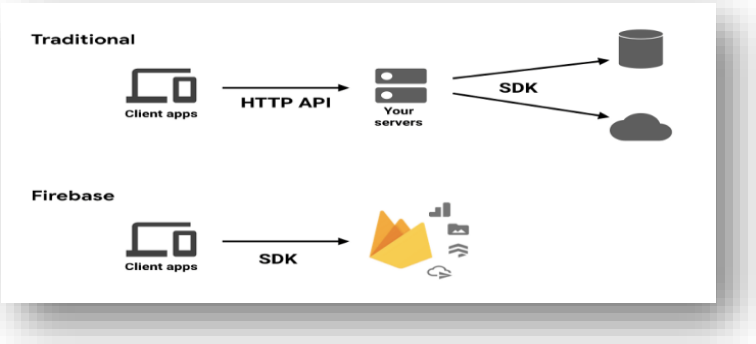


Figura 59. Comparación entre sistema clásico utilizando servidor y Firebase.

5.1.5 Android Studio

Android Studio es el IDE oficial del sistema Android. Esta basado en lenguaje IntelliJ IDEA de JetBrains, diseñado específicamente para el desarrollo de aplicaciones Android. Admite los lenguajes de programación: Kotlin, Java y C++. Podríamos dedicar un trabajo entero para explicar este sistema, con lo que pasaremos a explicar las características más importantes.

- Android Studio proporciona soporte para construcción basada en Gradle.
- Refactorización del código y arreglos.
- Conjunto de herramientas Lint, para la detección de problemas.
- Integración de ProGuard y firma de aplicaciones.
- Plantillas de diseño de aplicaciones.
- Editor de diseño para la interfaz de usuario.
- Soporta Android Wear, hoy Wear OS.
- Google Cloud Platform.
- Dispositivo virtual para pruebas.
- Renderizado en tiempo real.
- Consola de desarrollador.

Los requisitos necesarios para poder integrar este sistema en nuestro ordenador se muestran en la siguiente tabla:

	Windows	**OS X/macOS**	**Linux**
Sistema Operativo	10, 8, 7 (32 o 64-bit)	10.10 o superior	GNOME o KDE
RAM	4 GB mínimo, 8 recomendado, más 1 GB extra para el simulador.		
Almacenamiento	2 GB mínimo, 4 recomendado.		
Java	Java Development Kit (JDK) 8		
Pantalla	1280 x 800 mínimo, 1440 x 900 recomendado.		

Tabla 3. Requisitos para la versión 3.x.

El emulador presenta unos requisitos propios para poder trabajar correctamente:

- SDK Tools 26.1.1 o superior.
- Procesador de 64 bits.
- Si trabajamos con Windows, CPU con soporte UG.
- HAXM 6.2.1 o posterior.

Cada proyecto contiene una serie de módulos con los códigos de los programas y el diseño de la app. Podemos encontrar en un proyecto Android los siguientes módulos:

- **Módulo de aplicación.**
- **Módulos de biblioteca.**
- **Módulos de Google App Engine.**

Cada módulo de aplicación contiene las carpetas:

- **Manifest.** Archivo XML con las funciones y características del proyecto.
- **Java.** Incluye los archivos con los códigos fuente en formato Java.
- **Res.** Diseños XML, Strings de IU, mapas de bits.
-

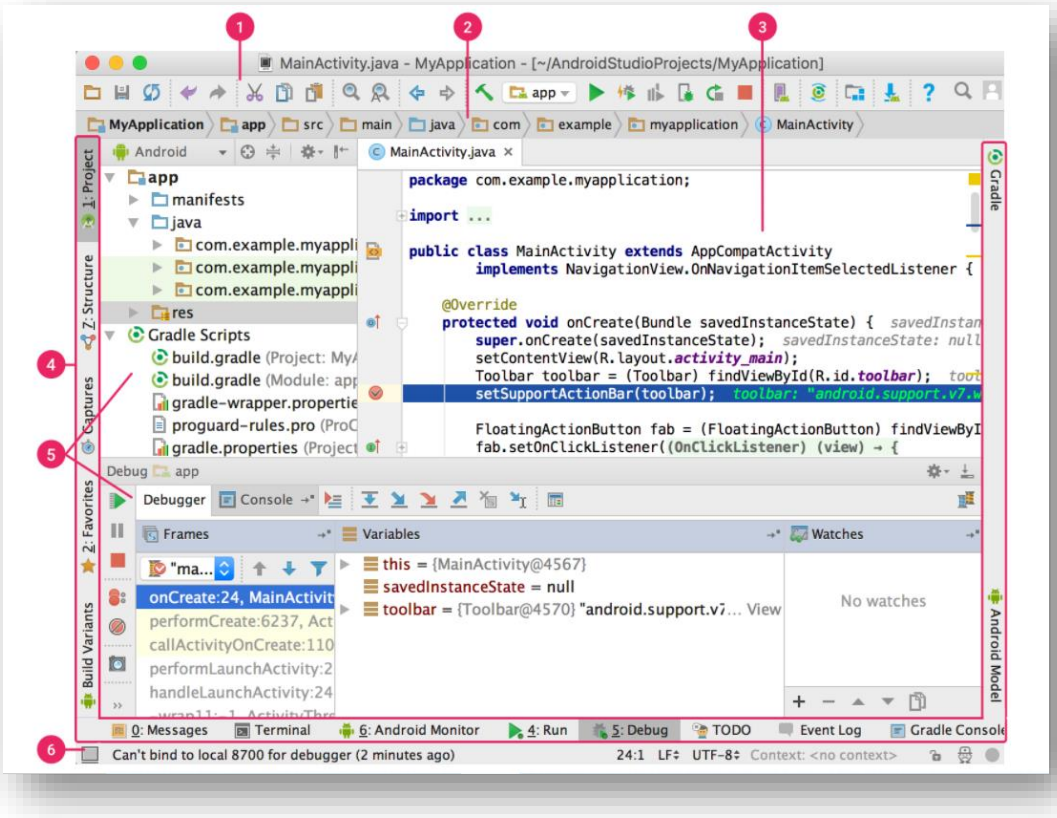


Figura 60. Interfaz de usuario en Android Studio.

En la imagen superior incluyo la vista de la interfaz de usuario del IDE de Android Studio. Detallo en las líneas inferiores los elementos destacados en la imagen:

1. Barra de herramientas.
2. Barra de navegación.
3. Ventana de editor.
4. Barra de la ventana de herramientas.
5. Acceso a tareas específicas.
6. Barra de estado.

5.2 Presupuesto.

El presupuesto ha sido bastante reducido, puesto que parte de los sistemas usados se ofrecen de forma gratuita. Únicamente se ha adquirido mediante compra el sensor y la placa de desarrollo MKRFOX-1200. Se detalla el coste total.

- Sensor caudalímetro YF-S201……Precio = 8.99 euros.
- Placa de Desarrollo MKRFOX-1200. Precio = 33.76 euros.
- Gastos de envío…….. Precio = 4.99 euros.
- Total……...Precio = 47.74 euros.

Capítulo 6. Conclusión

Una de las principales conclusiones extraídas tras el proceso de desarrollo de este proyecto ha sido comprobar que se ha producido un avance importante en el ámbito de las tecnologías referentes al Internet de las cosas, tanto en el ámbito de desarrollo de dispositivos, donde podemos encontrar productos que han sido especialmente diseñados con funcionalidades para trabajar con este sistema, como en temas de conectividad entre diferentes sistemas dedicados, los cuales presentan protocolos de comunicación sencillos de implementar, permitiendo un tránsito de datos bidireccional. Estos avances ofrecen cada día más facilidades a la hora de desarrollar soluciones IoT.

Las dificultades técnicas encontradas en el proceso se han localizado sobre todo a la hora de configurar los protocolos de comunicación entre dispositivos. Pero estas han sido superficiales y no han supuesto un retraso importante, ya que se han podido solventar tras un breve proceso de comprobación de errores. Hoy en día, el proceso de conectividad entre diferentes sistemas esta en un estado avanzado y podemos encontrar multitud de información y ejemplos en internet.

Respecto a la eficacia del prototipo bajo estudio, podemos concluir que cumple con las funciones para las que fue originalmente diseñado. Permite la lectura del estado de los dispositivos sensores, actualiza la información y establece comunicación fiable entre los usuarios de la aplicación. Empleando un conjunto de dispositivos suficiente se podría monitorizar toda una zona de regadío. Un sistema de tales características ofrecería a los usuarios una información global sobre el estado de los canales de irrigación.

Tras una breve mirada al capitulo de presupuestos, podemos observar que para la realización de este experimento no es necesaria una inversión económica importante. Las compañías dedicadas al desarrollo de sistemas similares cuentan entre sus ofertas con productos con precios asequibles. En multitud de ocasiones podemos encontrar que estas compañías permiten el uso de sus sistemas de manera gratuita, como el uso de las bases de datos de Google, o los programas de diseño de aplicaciones utilizados. En el caso de querer ampliar las ventajas ofrecidas con un carácter más profesional, ofrecen cuotas de uso que no suponen gastos desorbitados. El coste total de un sistema con carácter más profesional seria fácilmente asumible por un colectivo de operarios o comunidad de regantes.

Este proyecto podría ser el punto de partida para el desarrollo de un sistema con carácter más comercial.

Capítulo 7. Bibliografía

[1] Luis Llamas. “Medir caudal y consumo de agua con Arduino y caudalímetro”. Tutoriales Arduino. Diciembre 2016.

https://www.luisllamas.es/caudal-consumo-de-agua-con-arduino-y-caudalimetro/

[2]. Luis del Valle Hernández. “Introducción Arduino MKRFOX-1200 Sigfox y redes LPWAN”. Programar Fácil.com.

https://programarfacil.com/blog/arduino-blog/arduino-mkrfox1200-sigfox-lpwan/

[3] KAD. “Como crear un chat con Firebase y Android en una hora”. Septiembre 2017.

https://www.youtube.com/watch?v=DFnxY_PEnYY

[4] “Introducción a Android Studio”. Developers Android. Mayo 2017

https://developer.android.com/studio/intro?hl=es-419

[5] “Android Studio”. Wikipedia, la enciclopedia libre. Septiembre 2021.

https://es.wikipedia.org/wiki/Android_Studio

[6] “Firebase, que es y para qué sirve”. Digital 55. Mayo 2020.

https://www.digital55.com/desarrollo-tecnologia/que-es-firebase-funcionalidades-ventajas-conclusiones/

[7] “Arduino MKRFOX-1200”. Aprendiendo Sigfox. Marzo 2018.

https://www.aprendiendoarduino.com/2018/03/05/arduino-mkrfox1200/

Anexo 1

En este anexo se incluirán las modificaciones realizadas en el código que implementa el mapa de la aplicación, con las ubicaciones que señalizan el agua de riego. Los cambios que se han realizado han sido, entre otros, introducir variables tipo Array para almacenar los valores de los bucles. Estos mismos bucles, que antes estaban implementados con la sentencia while, se han cambiado por bucles for. Parece que el cambio ofrecía un mejor rendimiento en la ejecución del programa. A continuación, se incluye el código empleado para ello.

```
package com.example.firebasemierda2;

import static android.content.ContentValues.TAG;

import androidx.annotation.NonNull;
import androidx.fragment.app.FragmentActivity;

import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;

import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.SupportMapFragment;
import com.google.android.gms.maps.model.BitmapDescriptorFactory;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.Marker;
import com.google.android.gms.maps.model.MarkerOptions;
import com.example.firebasemierda2.databinding.ActivityMapsBinding;
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;

public class MapsActivity extends FragmentActivity implements
OnMapReadyCallback {

    private GoogleMap mMap;
    private ActivityMapsBinding binding;
    private DatabaseReference bbdd;
    private String frequency;
    private Marker markerPrueba;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivityMapsBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        // Obtain the SupportMapFragment and get notified when the map
is ready to be used.
        SupportMapFragment mapFragment = (SupportMapFragment)
```

```
getSupportFragmentManager()
                .findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
        bbdd = FirebaseDatabase.getInstance().getReference();

    }


    @Override
    public void onMapReady(GoogleMap googleMap) {
        mMap = googleMap;
        mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
        Log.d(TAG, "Dispositivo is: " + " pene " );
        int numeroDispositivos=4;
        double[] latitudes=new double[numeroDispositivos];
        double[] longitudes=new double[numeroDispositivos];
        String[] dispositivos=new String[numeroDispositivos];
        for(int i=0;i<numeroDispositivos;i++){
            String stringI=String.valueOf(i);
            dispositivos[i]="device" + stringI;
            Log.d(TAG, "Dispositivo is: " + dispositivos[i]);
            switch (i){
                case 0:
                    latitudes[i] = 39.536;
                    longitudes[i] = -0.3396;
                    break;
                case 1:
                    latitudes[i] = 39.536119;
                    longitudes[i] = -0.337856;
                    break;
                case 2:
                    latitudes[i] = 39.535399;
                    longitudes[i] = -0.341547;
                    break;
                case 3:
                    latitudes[i] = 39.536946;
                    longitudes[i] = -0.339798;
                    break;
            }
            Log.d(TAG, "Long y lat is: " + latitudes[i] +" " +
longitudes[i]);
        }
        // Add a marker in Sydney and move the camera
        //LatLng sydney = new LatLng(-34, 151);
        //mMap.addMarker(new
MarkerOptions().position(sydney).title("Marker in Sydney"));
        //mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));
        for(int i=0;i<numeroDispositivos;i++) {
            LatLng latlong = new LatLng(latitudes[i],longitudes[i]);
            bbdd.child(dispositivos[i]).addValueEventListener(new
ValueEventListener() {
                @Override
                public void onDataChange(@NonNull DataSnapshot
snapshot) {
                    if (snapshot.exists()) {
                       String value =
snapshot.child("data").getValue(String.class);
```

```
                        Log.d(TAG, "Value is: " + value);
                        float f1 = Float.parseFloat(value);

                        if (f1 < 20) {
                            markerPrueba = googleMap.addMarker(new
MarkerOptions().position(latlong).title("dispositivo0").snippet("No
hay agua de riego")

.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HU
E_RED)));

mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(latlong, 16));
                        } else {
                            markerPrueba = googleMap.addMarker(new
MarkerOptions().position(latlong).title("dispositivo0").snippet("Agua
de riego disponible")

.icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HU
E_BLUE)));

mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(latlong, 16));
                        }
                    }
                }

                @Override
                public void onCancelled(@NonNull DatabaseError error)
{

                }
            });
        }
    }
}
```

Anexo 2

En este apartado se incorporan el resto de los códigos creados con el programa Android Studio. Se exponen primero los códigos java, cerrando la sección con los códigos xml relevantes.

- MainActivity (CHAT)

```
package com.example.firebasemierda2;

import static android.content.ContentValues.TAG;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import com.google.firebase.database.ChildEventListener;
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ServerValue;

import de.hdodenhof.circleimageview.CircleImageView;

public class MainActivity extends AppCompatActivity {
    private CircleImageView FotoPerfil;
    private TextView nombre;
    private RecyclerView rvMensajes;
    private EditText txtMensaje;
    private Button btnEnviar;
    private String fotoPerfilCadena;
    private AdapterMensajes adapter;
    private FirebaseDatabase database;
    private DatabaseReference databaseReference;
    public static String nombreDef ;


    private static final int PHOTO_SEND = 1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        nombreDef = getIntent().getStringExtra("miNombreTotal");
        Log.d(TAG, "NOMBRE USUARIO2: " + nombreDef);
        FotoPerfil =
(CircleImageView)findViewById(R.id.fotoPerfil);
        nombre = (TextView)findViewById(R.id.nombre);
        nombre.setText(nombreDef);
```

```
        rvMensajes = (RecyclerView) findViewById(R.id.rvMensajes);
        txtMensaje = (EditText) findViewById(R.id.txtMensaje);
        btnEnviar = (Button) findViewById(R.id.btnEnviar);
        fotoPerfilCadena = "";
        database = FirebaseDatabase.getInstance();
        databaseReference = database.getReference("chat"); //sala
de chat


        adapter = new AdapterMensajes(this);
        LinearLayoutManager l = new LinearLayoutManager(this);
        rvMensajes.setLayoutManager(l);
        rvMensajes.setAdapter(adapter);

        btnEnviar.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                databaseReference.push().setValue(new
MensajeEnviar(txtMensaje.getText().toString(),
nombre.getText().toString(), "", "1", ServerValue.TIMESTAMP));
                txtMensaje.setText("");

            }
        });


        adapter.registerAdapterDataObserver(new
RecyclerView.AdapterDataObserver() {
            @Override
            public void onItemRangeInserted(int positionStart, int
itemCount) {
                super.onItemRangeInserted(positionStart,
itemCount);
                setScrollbar();
            }
        });
        databaseReference.addChildEventListener(new
ChildEventListener() {
            @Override
            public void onChildAdded( DataSnapshot datasnapshot,
String s) {
                MensajeRecibir m =
datasnapshot.getValue(MensajeRecibir.class);
                adapter.addMensaje(m);
            }

            @Override
            public void onChildChanged( DataSnapshot snapshot,
String s) {

            }

            @Override
            public void onChildRemoved( DataSnapshot snapshot) {

            }
```

```
            @Override
            public void onChildMoved( DataSnapshot snapshot,
String s) {

            }

            @Override
            public void onCancelled( DatabaseError error) {

            }
        });
    }
    private void setScrollbar(){
        rvMensajes.scrollToPosition(adapter.getItemCount()-1);
    }

}
```

- MainActivity2. (Pantalla de inicio)

```
package com.example.firebasemierda2;


import static android.content.ContentValues.TAG;

import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

import androidx.appcompat.app.AppCompatActivity;

public class MainActivity2 extends AppCompatActivity {
    Button btnmap;
    Button btnmap2;
    Button btnmap3;
    Button btnOkey;

    private EditText EscribeTuNom;
    private TextView miSalida;

    //@SuppressLint("WrongViewCast")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main2);
        EscribeTuNom = (EditText)findViewById(R.id.textNombre);
        miSalida = (TextView)findViewById(R.id.textNombre);

        btnmap = findViewById(R.id.BTNMAPA);
        btnmap2 = findViewById(R.id.BTNMAPA2);
        btnmap3 = findViewById(R.id.BTNMAPA4);
        btnOkey = findViewById(R.id.btnOK);
```

```
        btnmap.setEnabled(false);
        btnmap2.setEnabled(false);
        btnmap3.setEnabled(false);
        btnmap.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(MainActivity2.this,
MapsActivity.class);
                startActivity(intent);
            }
        });
        btnmap2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String miNombre = null;
                miNombre = EscribeTuNom.getText().toString();
                String soloNombre = miNombre.substring(11);
                Intent intent = new Intent(MainActivity2.this,
MainActivity.class);
                intent.putExtra("miNombreTotal",
String.valueOf(soloNombre));
                startActivity(intent);
            }
        });
        btnmap3.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(MainActivity2.this,
MainActivity3.class);
                startActivity(intent);
            }
        });

        btnOkey.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String miNombre = null;
                miNombre = EscribeTuNom.getText().toString();
                miSalida.setText("Bienvenido " + miNombre);
                Log.d(TAG, "NOMBRE USUARIO: " + miNombre);
                Intent intent = new Intent(MainActivity2.this,
MainActivity4.class);
                startActivity(intent);
                btnmap.setEnabled(true);
                btnmap2.setEnabled(true);
                btnmap3.setEnabled(true);
                btnOkey.setEnabled(false);

            }
        });

    }
}
```

- MainActivity3. (AVISOS)

```
package com.example.firebasemierda2;

import static android.content.ContentValues.TAG;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.text.method.ScrollingMovementMethod;
import android.util.Log;
import android.widget.TextView;

import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;

public class MainActivity3 extends AppCompatActivity {

    private DatabaseReference bbdd2;
    private int countAvisos = 0;
    private TextView pantallaAvisos;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main3);

        pantallaAvisos = (TextView) findViewById(R.id.cuadroAvisos);
        pantallaAvisos.setMovementMethod(new
ScrollingMovementMethod());

        bbdd2 = FirebaseDatabase.getInstance().getReference();

        bbdd2.child("CarpetaAvisos").addValueEventListener(new
ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                if (snapshot.exists()) {
                    countAvisos =(int) snapshot.getChildrenCount();
                    Log.d(TAG, "Numero de avisos " + countAvisos);
                    String[] aviso=new String[countAvisos];
                    for(int i=0;i<countAvisos;i++){
                        String stringI=String.valueOf(i);
                        aviso[i]="Aviso" + stringI;
                        String[] value = new String[countAvisos];
                        value[i] =
snapshot.child(aviso[i]).getValue(String.class);
                        Log.d(TAG, "Value is: " + aviso[i] +
value[i]);
                        pantallaAvisos.append("\n" + value[i]+
```

```
"\n"+"\n");
                        }

                    }
                }

                @Override
                public void onCancelled(@NonNull DatabaseError error) {

                }
            });


    }
}
```

- Clase Mensaje.

```
package com.example.firebasemierda2;

public class Mensaje {
    private String mensaje;
    private String nombre;
    private String fotoPerfil;
    private String type_mensaje;

    public Mensaje() {
    }

    public Mensaje(String mensaje, String nombre, String
fotoPerfil, String type_mensaje) {
        this.mensaje = mensaje;
        this.nombre = nombre;
        this.fotoPerfil = fotoPerfil;
        this.type_mensaje = type_mensaje;
    }

    public String getMensaje() {
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getFotoPerfil() {
        return fotoPerfil;
    }
```

```
    public void setFotoPerfil(String fotoPerfil) {
        this.fotoPerfil = fotoPerfil;
    }

    public String getType_mensaje() {
        return type_mensaje;
    }

    public void setType_mensaje(String type_mensaje) {
        this.type_mensaje = type_mensaje;
    }


}
```

- Clase HolderMensaje.

```
package com.example.firebasemierda2;

import android.view.View;
import android.widget.TextView;

import androidx.annotation.NonNull;
import androidx.recyclerview.widget.RecyclerView;

import de.hdodenhof.circleimageview.CircleImageView;

public class HolderMensaje extends RecyclerView.ViewHolder{

    private TextView nombre;
    private  TextView mensaje;
    private TextView hora;
    private CircleImageView fotoMensaje;

    public HolderMensaje(@NonNull View itemView) {
        super(itemView);
        nombre =
(TextView)itemView.findViewById(R.id.nombreMensaje);
        mensaje =
(TextView)itemView.findViewById(R.id.mensajeMensaje);
        hora =
(TextView)itemView.findViewById(R.id.horaMensaje);
        fotoMensaje = (CircleImageView)
itemView.findViewById(R.id.fotoPerfilMensaje);
    }

    public TextView getNombre() {
        return nombre;
    }

    public void setNombre(TextView nombre) {
        this.nombre = nombre;
    }

    public TextView getMensaje() {
        return mensaje;
```

```
        }

        public void setMensaje(TextView mensaje) {
            this.mensaje = mensaje;
        }

        public TextView getHora() {
            return hora;
        }

        public void setHora(TextView hora) {
            this.hora = hora;
        }

        public CircleImageView getFotoMensaje() {
            return fotoMensaje;
        }

        public void setFotoMensaje(CircleImageView fotoMensaje) {
            this.fotoMensaje = fotoMensaje;
        }
    }
```

- Clase AdapterMensaje.

```
package com.example.firebasemierda2;

import android.content.Context;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import androidx.recyclerview.widget.RecyclerView;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

//import androidx.annotation.NonNull;


public class AdapterMensajes extends
RecyclerView.Adapter<HolderMensaje> {
    private List<MensajeRecibir> listMensaje = new ArrayList<>();
    private Context c;

    public AdapterMensajes(Context c) {
        this.c = c;
    }
    public void addMensaje(MensajeRecibir m){
        listMensaje.add(m);
        notifyItemInserted(listMensaje.size());
    }
```

```
    @Override
    public HolderMensaje onCreateViewHolder(ViewGroup parent, int
viewType) {
        View v =
LayoutInflater.from(c).inflate(R.layout.card_view_mensajes,parent,fals
e);
        return new HolderMensaje(v);
    }

    @Override
    public void onBindViewHolder(HolderMensaje holder, int position) {

holder.getNombre().setText(listMensaje.get(position).getNombre());

holder.getMensaje().setText(listMensaje.get(position).getMensaje());
        Long codigoHora = listMensaje.get(position).getHora();
        Date d = new Date(codigoHora);
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss a"); //
para pm o am
        holder.getHora().setText(sdf.format(d));

    }

    @Override
    public int getItemCount() {
        return listMensaje.size();
    }
}
```

- MensajeEnviar

```
package com.example.firebasemierda2;

import java.util.Map;

public class MensajeEnviar extends Mensaje{
    private Map hora ;

    public MensajeEnviar() {
    }

    public MensajeEnviar(Map hora) {
        this.hora = hora;
    }

    public MensajeEnviar(String mensaje, String nombre, String
fotoPerfil, String type_mensaje, Map hora) {
        super(mensaje, nombre, fotoPerfil, type_mensaje);
        this.hora = hora;
    }

    public Map getHora() {
        return hora;
    }
```

```
    public void setHora(Map hora) {
        this.hora = hora;
    }
}
```

- MensajeRecibir

```
package com.example.firebasemierda2;

public class MensajeRecibir extends Mensaje{
    private Long hora;

    public MensajeRecibir() {
    }

    public MensajeRecibir(Long hora) {
        this.hora = hora;
    }

    public MensajeRecibir(String mensaje, String nombre, String
fotoPerfil, String type_mensaje, Long hora) {
        super(mensaje, nombre, fotoPerfil, type_mensaje);
        this.hora = hora;
    }

    public Long getHora() {
        return hora;
    }

    public void setHora(Long hora) {
        this.hora = hora;
    }
}
```

- Activity_maps. (MAPA xml)

```
<?xml version="1.0" encoding="utf-8"?>
<fragment
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MapsActivity" />
```

- Activity_main. (CHAT xml)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    tools:context=".MainActivity"
    android:padding="5sp"
    android:orientation="vertical">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
        <de.hdodenhof.circleimageview.CircleImageView
            android:id="@+id/fotoPerfil"
            android:layout_width="30sp"
            android:layout_height="30sp"
            android:src="@drawable/gotaapp" />
        <TextView
            android:id="@+id/nombre"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:hint="Regante1"
            android:layout_gravity="center"/>
    </LinearLayout>
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/rvMensajes"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1">
    </androidx.recyclerview.widget.RecyclerView>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
<!--         <ImageButton-->
<!--             android:id="@+id/btnEnviarFoto"-->
<!--             android:layout_width="wrap_content"-->
<!--             android:layout_height="wrap_content"-->
<!--
android:background="@android:drawable/ic_menu_gallery"-->
<!--             android:layout_gravity="center" />-->
        <EditText
            android:id="@+id/txtMensaje"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Escribe un mensaje"
            android:layout_weight="1"/>

        <Button
            android:id="@+id/btnEnviar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:hint="Enviar"/>
    </LinearLayout>
</LinearLayout>
```

FIN