SENSOR IoT PARA MONITORIZACIÓN DE CONSUMO DE ENERGÍA EN CONTINUA

Ferreras Astorqui, Ignacio María

GRADO EN INGENIERÍA DE COMPUTADORES. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería de Computadores

Madrid, 1 de junio de 2016

Directores:

Igual Peña, Francisco
Piñuel Moreno, Luis

Autorización de difusión

Ferreras Astorqui, Ignacio María

Madrid, a 1 de junio de 2016

Los abajo firmantes, matriculados en el Grado de Ingeniería Informática de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: "SENSOR IoT PARA MONITORIZACIÓN DE CONSUMO DE ENERGÍA EN CONTINUA", realizado durante el curso académico 2015-2016 bajo la dirección de Francisco Igual Peña y la co-dirección de Luis Piñuel Moreno en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

"If the statistics are boring, you've got the wrong numbers"

Edward Tufte

Agradecimientos

Muchas gracias a Francisco Igual Peña y a Luis Piñuel Moreno por toda la ayuda prestada.

Índice general

Índice de figuras

Resumen

Hoy en día la mayor parte de los sensores de energía IoT (Internet of Things) están orientados a la medida de corriente alterna (AC). No son aptos para monitorizar equipos que no estén conectados a la red eléctrica (baterías, paneles fotovoltaicos, etc.) o que formen parte de otros equipos más grandes y que estén situados detrás del transformador (ej. aceleradores de cómputo en supercomputadores).

El presente trabajo tiene como objetivo principal construir un sistema, con una instalación sencilla y reducida, que permita la monitorización de consumo de dispositivos conectados a corriente continua. Toda la información recogida será mostrada a través de una interfaz web, que nos permitirá observar los cambios en el consumo en tiempo real con un intervalo de actualización especificado por el usuario. Además el sistema será robusto, con bajo coste de implementación y permitirá una alta escalabilidad, ya que el objetivo del proyecto es que sea escalable a nivel de centro de datos o institución.

Palabras clave

INA219, ESP8266 - NodeMCU, consumo, potencia, voltaje, Node-Red, MQTT, LUA, Freeboard, flujo de datos.

Abstract

Nowadays practically all the energy sensors are oriented to the measuring of alternating current (AC). They are not capable to monitor equipment that is not connected to the power grid (batteries, photo voltaic systems, etc) or devices that are behind the transformer and they are part of bigger system (for example: hardware accelerators).

The main goal of the project is to develop a simple and small system to monitor the energy consumption of any device connected to direct current. A web interface has been set up to show all the gathered information. These data will be shown in real-time at a user-specified refreshing rate. In addition, the system will be robust with low-cost and with great scalability.

Keywords

INA219, ESP8266 - NodeMCU, consumption, current, voltage, Node-Red, MQTT, LUA, Freeboard, data flow.

Capítulo 1

Introducción

IoT (*Internet of Things*) es un concepto que se refiere a la interconexión digital de objetos cotidianos con Internet. En este tipo de infraestructuras se dispone de diferentes elementos, cada uno con un objetivo claro:

- Nodo Sensor: encargado de la captura de datos; en nuestro caso, un nodo de bajo coste ESP8266 y un sensor de corriente continua Texas Instruments INA219.
- Broker/Gateaway: encargado de almacenar los diferentes elementos software usados; normalmente se usan sistemas de bajo coste y prestaciones, como por ejemplo Raspberry Pi o BeagleBone Black.
- Visualización: la forma en la que se mostrarán los datos capturados del sistema.

1.1. Objetivos y visión general del sistema

El objetivo principal de este proyecto es ofrecer un sistema robusto y escalable, capaz de monitorizar sistemas en corriente continua o sub-sistemas sobre los que queramos obtener información concreta. Se pretende que el resultado del proyecto pueda ser utilizado por cualquier persona, empresa o universidad que le vea un uso para sus proyectos. Para ello, el sistema ha de ser lo suficientemente sencillo como para que una persona sin altos conocimientos de informática pueda instalarlo, pero a su vez, una persona con altos conocimientos pueda sacar todo su potencial de manera cómoda y eficaz.

En la misma Universidad Complutense, existen proyectos actuales a los que el sistema completo o parte de él puede ayudar a su desarrollo y monitorización. Además, la posibilidad de personalización le da la capacidad de adaptarse al entorno en que se le necesite sin una gran carga de trabajo. Por último, su visualización da libertad al usuario ya que al ser una interfaz web permite a éste monitorizar el sistema remotamente. A su vez, la interfaz puede ser modificada por el usuario de una manera sencilla a través la propia interfaz, sin necesidad de modificar el código. El software utilizado en el gateway tiene la capacidad de crear alarmas, que nos permite despreocuparnos, ya que en el momento en el que falle podremos saberlo al instante, y con su persistencia (almacenamiento en base de datos) podremos observar qué es lo que ha producido el posible fallo en nuestro estudio.

1.2. Tecnologías del sistema

El sistema dispone de diferentes elementos tanto a nivel hardware como software. En el siguiente diagrama podemos observar el funcionamiento general del sistema, sin preocuparnos de momento, por la forma en la que se comunican entre ellos. Más adelante se completará este diagrama para que la visión sea completa.

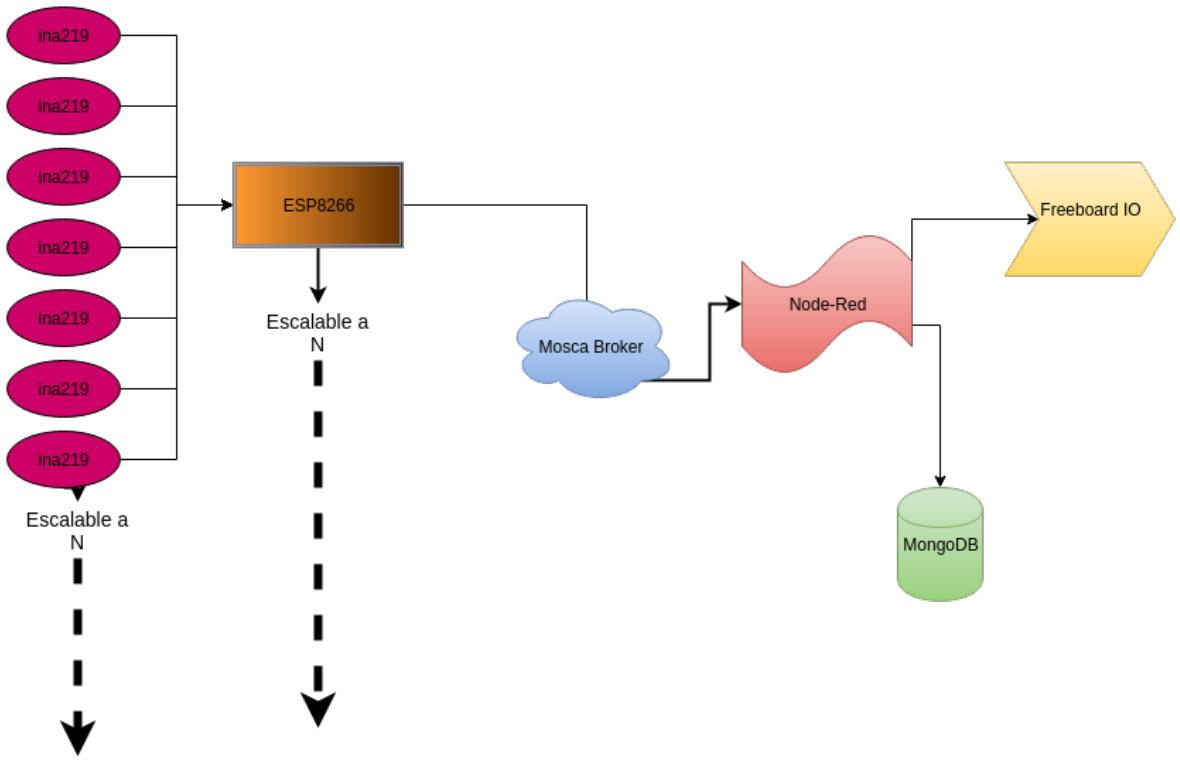

Figura 1.1: *Diagrama de funcionamiento del sistema*

En el diagrama se puede observar que el flujo de datos comienza con los INA219. Estos chips son los que realizan las mediciones de los datos que vamos a estar tratando a lo largo de todo el sistema. La ESP8266 es la encargada de solicitar dicha información a los diferentes INA219 a los que esté conectado. Esta placa permite diferentes métodos de comunicación y a su vez tiene un módulo de conexión WI-FI que le permite transmitir los datos de forma inalámbrica.

Toda esta información es recibida por el *Broker*, en constante ejecución dentro del gateway. Cabe constatar la posibilidad de replicar el sistema descrito en el párrafo anterior hasta N veces si es necesario. Esto permite poder motorizar tantos dispositivos como deseemos. Este Broker redistribuye la información para que ésta llegue a quien desee recibirla.
En este caso es Node-Red el que recibe esta información y es él quien se encarga de que a la información se le de formato y que llegue a sus destinos. Estos son: MongoDB, por la necesidad de persistencia, y Freeboard, que es la encargada de ofrecer una visualización de los datos para el usuario.

1.3. Comunicación en 2 vías

El sistema está diseñado para permitir una configuración en 2 vías. Esto permite la personalización de ciertos factores de configuración. Por ejemplo, el tiempo de refresco, el sensor o sensores concretos a monitorizar, o qué métricas dentro de dichos sensores van a consultarse.

La comunicación en este sentido es breve puesto que se origina desde Node-Red y solo ha de llegar a la ESP8266, como vemos en la siguiente figura. La ESP8266 es la encargada de solicitar los datos a los INA219s conectados a ella y por tanto es ella quien puede modificar los valores de personalización del sistema. Dicha personalización se hace a nivel de ESP8266, lo cual implica que si tenemos múltiples ESP8266 conectadas, cada una puede tener sus propias configuraciones personalizadas.

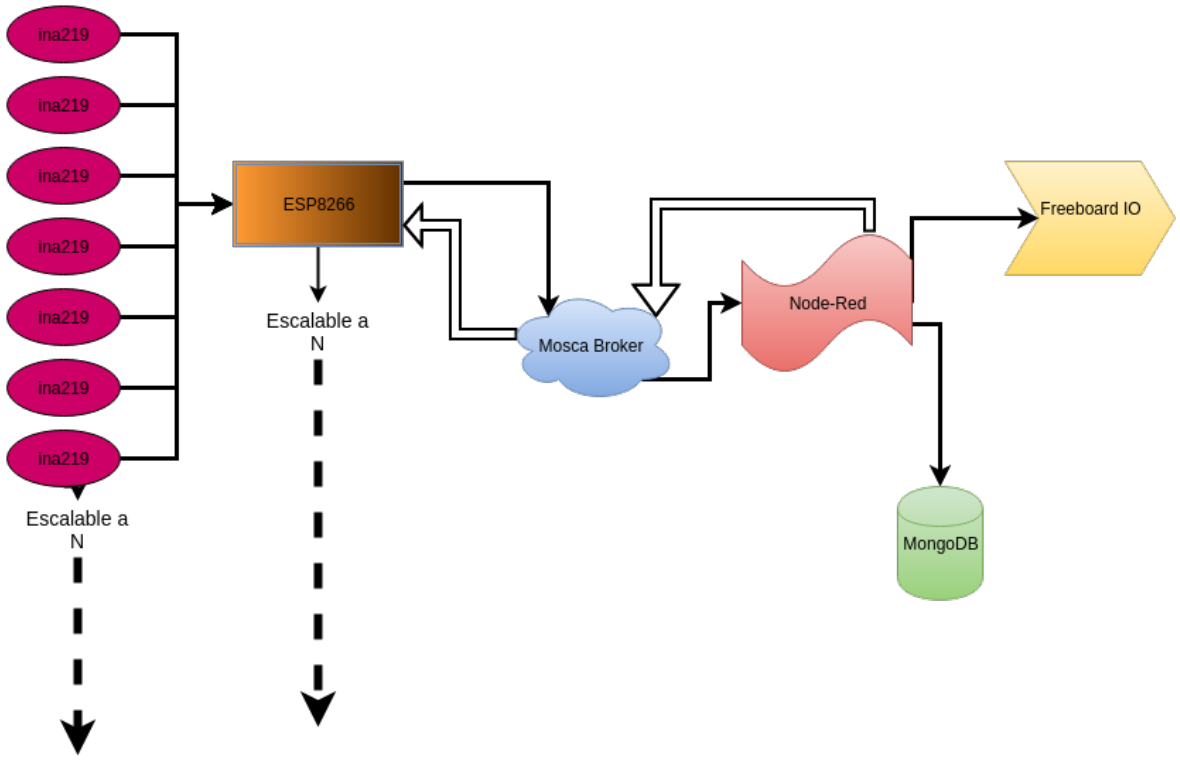

Figura 1.2: *Diagrama de comunicación en 2 vías.*

1.4. Protocolos de Comunicación

En el diagrama de la figura 1.3 es posible observar todo el sistema con sus 2 vías de comunicación así como los protocolos de comunicación usados para pasar los mensajes en ambos sentidos. Este es el sistema que he desarrollado al completo.

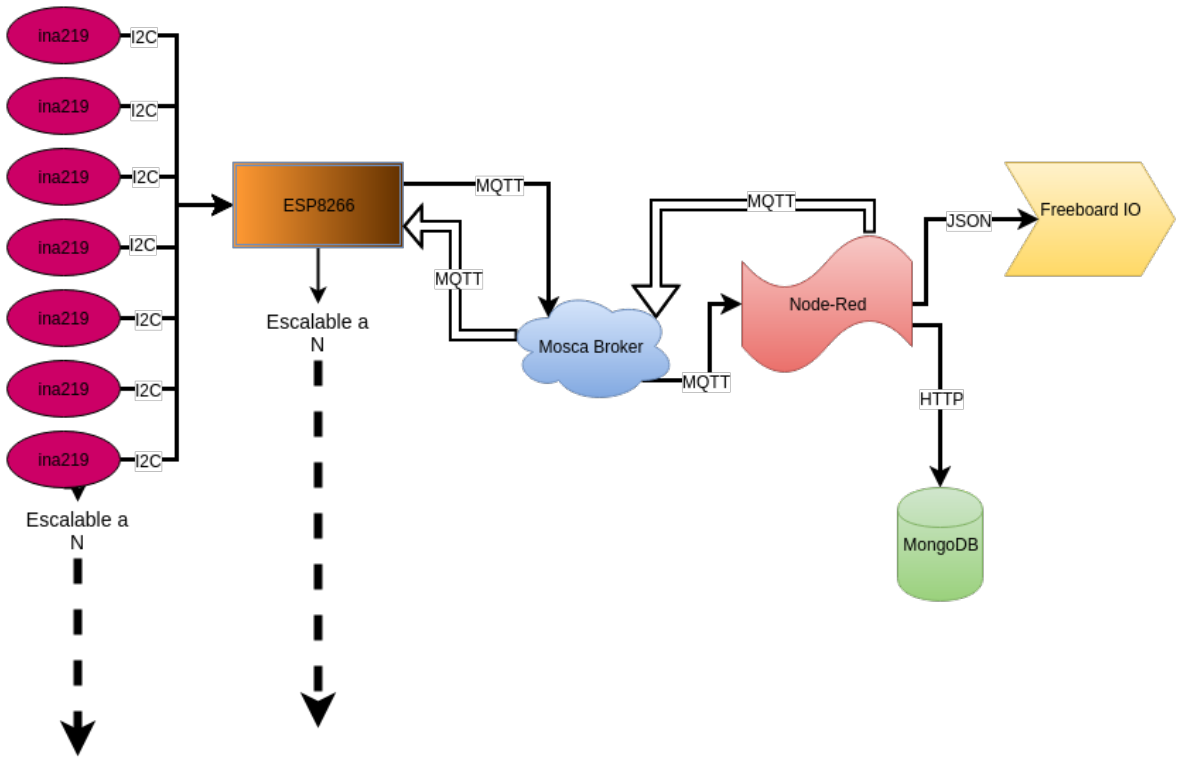

Figura 1.3: *Diagrama completo del sistema.*

Como se puede observar la comunicación a bajo nivel, que es la que conecta a los INA219 con la ESP8266, se realiza a través del protocolo I^2C. Para las siguientes partes se decidió usar el protocolo MQTT, que es un protocolo de comunicación ligero basado en publicación/suscripción, muy utilizado en infraestructuras IoT. Por últimom Node-Red usa HTTP para comunicarse con la base de datos MongoDB, lo cual permite que se encuentre en otro dominio si fuese necesario. Para Freeboard se utiliza un protocolo basado en JSON.

Capítulo 2

Obtención de los Datos

En este capítulo estudiaremos más a fondo la forma de obtención de los datos. A su vez, coincide con la primera etapa de desarrollo del sistema. Principalmente nos centraremos en el INA219, la ESP8266 y la comunicación entre ellos a través de I^2C.

2.1. INA219

El INA219 es un chip desarrollado por Texas Instruments que permite la lectura de datos de corriente y potencia. La comunicación con este chip es realizada a través del protocolo de comunicación I^2C. Además el precio de éste dispositivo es realmente reducido (1.90€[1]). El INA219 está compuesto por un amplificador para medir el voltaje que circula a través de una resistencia de precisión de valor seleccionable. La máxima diferencia de entrada del amplificador es de ±320 mV lo que significa que puede medir hasta ±3.2 Amps. La capacidad descrita es la estándar del dispositivo, pero ésta (con los conocimientos adecuados) puede ser modificada para un mayor rango de medida.

Para este sistema hemos usado una placa diseñada por Juan Diego González como Trabajo de Fin de Grado en 2015. Esta placa tiene como objetivo ofrecer una forma más cómoda de uso y comunicación con los ocho INA219 que posee. La placa está diseñada para su uso en conjunción con una Beagle Bone Black, aunque en nuestro caso, utilizaremos una ESP8266 para gestionar la comunicación tanto desde el sensor como hacia el exterior.

[1] http://goo.gl/9qVlqw

Para nuestro diseño solo necesitamos los 4 pines principales involucrados en la comunicación I^2C: SDA, SCL, VDD y GND. Cabe destacar que resultaría trivial utilizar cualquier otra placa utilizando los mismos o similares sensores, con mínimos cambios en el desarrollo.

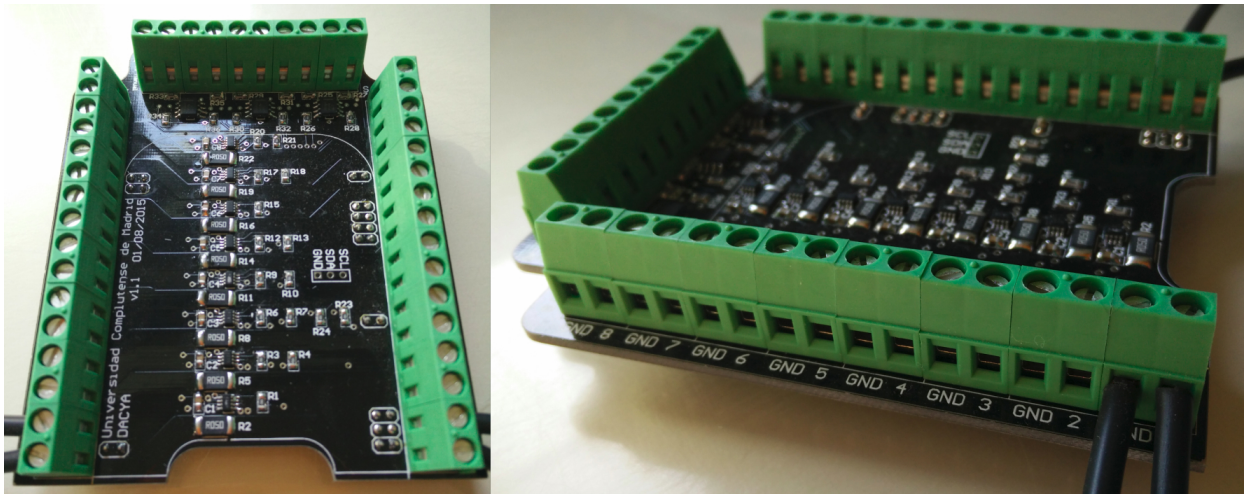

Figura 2.1: *Placa con los INA219*

Como podemos observar esta placa tiene 8 INA219s, lo cual permite que cada placa monitorice hasta 8 dispositivos diferentes. Cada entrada irá conectada por el lado derecho (INPUT) y saldrá por el izquierdo (OUTPUT) una vez se hayan realizado las mediciones. Los pines usados están en la parte inferior, ya que estos se conectaban directamente a la Beagle Bone Black. De esta forma, lo único necesario para poder leer de todos los INA219 de la placa es cambiar la dirección I^2C de cada dispositivo, en este caso desde la 0x40 al 0x47. Todos estos parámetros son configurables en el programa desarrollado para ejecutarse en la ESP8266.

Un solo INA219 nos proporciona el voltaje, la potencia y la corriente del sistema al que esté conectado, previa calibración. La forma de calibración de un INA219 consiste en la introducción de un valor específico en un registro del INA219, en este caso el 0x05. Dicho valor es calculado a través de una formula que podemos encontrar en el datasheet[2] del INA219 provisto por Texas Instruments.

[2] http://www.ti.com/lit/ds/symlink/ina219.pdf

En nuestro caso el valor que hemos usado es 0x16DB, que fue calculado previamente por Juan Diego González en su TFG del 2015 y que se adapta tanto a la resistencia utilizada como a los valores específicos del circuito a medir.

2.2. ESP8266

La ESP8266 es una placa cuya principal característica es el módulo Wi-Fi que tiene incorporado. Para este proyecto se ha usado un NodeMCU v1.0. Esta placa tiene integrada la ESP8266. Lo único extra que añade es la salida de los pines para que su uso sea más sencillo. Como podemos observar en la siguiente imagen, a la izquierda vemos un módulo ESP8266 y a la derecha un NodeMCU v1.0.

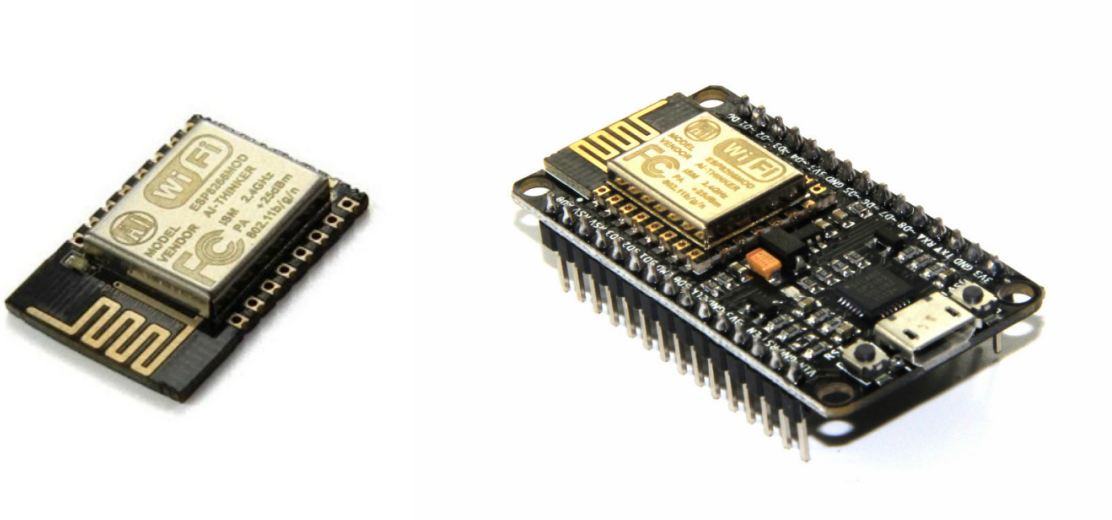

Figura 2.2: *ESP8266 vs NodeMCU*

El dispositivo NodeMCU fue desarrollado por la empresa Adafruit[3]. Ahora mismo existen 3 versiones diferentes, aunque su funcionalidad no ha cambiado mucho a lo largo de ellas. En gran parte se eligió este dispositivo por su reducido precio, ya que se puede encontrar por menos de 4€[4].

[3] https://www.adafruit.com
[4] http://goo.gl/vF8Xma

El NodeMCU puede ser programado en tres lenguajes diferentes:

- ***MicroPython:*** Este lenguaje consiste en una versión reducida de todo el potencial de Python en un lenguaje más ligero que permite su uso en dispositivos centrados en IoT. A pesar de su potencial, cabe constatar que actualmente se encuentra en desarrollo, después de conseguir suficientes fondos para completarlo a través de una plataforma de crowdfunding. Para el desarrollo de este sistema se intentó el uso de MicroPython pero se decidió dejarlo puesto que se descubrió que muchas de las librerías que íbamos a usar en el proyecto aún no habían sido desarrolladas. Se espera que MicroPython ofrezca muchas de las librerías que Python contiene, lo cual ofrecería grandes posibilidades de mejora en el proyecto.

- ***LUA:*** LUA es un lenguaje desarrollado por LabLUA, un laboratorio de investigación en Río de Janeiro, y el Departamento de Ciencias de la Computación de PUC Río. NodeMCU está realizando una adaptación de este lenguaje para sus dispositivos. Aún encontrándose en desarrollo ya tienen varias versiones estables. Al final se optó por este lenguaje ya que tiene una comunidad muy grande y la cantidad de errores o fallos que pudiésemos encontrar era menor que en MicroPython. Además ya disponen de múltiples librerías funcionando y con documentación, aunque ésta sea bastante escasa.

2.3. Comunicación y desafíos encontrados

La comunicación entre estos dos dispositivos se realiza a través del protocolo de comunicación I^2C. Este protocolo en serie funciona con la estructura maestro-esclavo. En nuestro caso los esclavos son los INA219 y el maestro la ESP8266. La recepción de datos es muy sencilla ya que simplemente se necesita una calibración y una lectura desde el registro correspondiente para obtener el dato.

Durante el desarrollo de esta etapa nos encontramos con los siguientes desafíos/problemas:

- El primer problema con el que nos encontramos al comenzar el proyecto fue la instalación y aprendizaje del lenguaje a tratar. Después de investigar los diferentes lenguajes y decidirnos por LUA, necesitábamos instalar la versión correcta. Para ello se actualizó el firmware de serie y el SDK asociado. Para ello NodeMCU permite descargar el firmware con las librerías que se deseen a través de esta web[5]. Las librerías que se decidieron incluir fueron bit, cjson, gpio, i2c, mqtt, node, tmr, uart y wifi. Una vez que pudimos instalar el nuevo firmware, actualizamos el SDK. Cabe constatar que la instalación en la placa resultó algo complicada, ya que las herramientas que hay para su instalación dependen mucho del sistema operativo. Por ejemplo: la herramienta de Linux (esptool) provoca problemas en múltiples ocasiones, lo que nos forzó a usar la herramienta de Windows (ESP8266Flasher).

- Una vez instalado y funcionando el firmware, empezamos a realizar pruebas de conexión para ser capaces de medir el voltaje, la corriente y la potencia de un INA219 no conectado, lo cual nos debería dar 0. Para ello antes tuvimos que calibrarlo de forma correcta. Esto llevó bastante tiempo puesto que los tutoriales y/o guías de uso de las diferentes llamadas de la librería I^2C de LUA eran escasas en explicaciones. Cada uno de ellos tenían formas diferentes de hacer las cosas y después de múltiples pruebas conseguimos establecer la calibración correcta. Una vez que ya recibíamos ceros, la siguiente etapa fue obtener los datos de un sistema reducido. El sistema era una simple fotoresistencia, que al variar provocaría un cambio en las salidas de éste.

- Uno de los grandes desafíos con el que nos encontramos, fue a la hora de mejorar el formato de los scripts de LUA que usamos para la solicitud de los datos a los INA219. Al principio, teníamos un simple script en el que llamábamos todo de forma secuencial.

[5] http://nodemcu-build.com

Pero dado el hecho de que teníamos que tratar con ocho INA219 diferentes, nos llevó a simplificar el script. Dicha simplificación fue mediante el uso de orientación a objetos. Ésto permitió la separación del script en dos, siendo uno de ellos una clase y el otro un script, que hace uso de la clase anterior a través de múltiples objetos. LUA soporta clases y orientación a objetos, pero no da una plataforma específica para su uso. Para conseguir una funcionalidad de clase tuvimos que buscar una estructura definida por uno de los desarrolladores de LUA.

- Una vez se consiguió un sistema funcionando a pequeña escala, decidimos llevar a cabo una prueba real para comprobar comó funcionaba el sistema bajo estrés. Ahí nos dimos cuenta de que el número de mensajes que enviábamos al Broker era demasiado grande, ya que enviábamos un mensaje por cada uno de los elementos (corriente, voltaje, potencia) de cada INA219, lo cual era terriblemente ineficiente. Al final se optó por englobar todos los datos en una sola tabla y convertirla a JSON. Esta compactación hizo que pasásemos de 27 mensajes a 1 mensaje por cada ESP8266. Además esto ayuda de manera increíble a la persistencia ya que simplemente hemos de guardar estos mensajes (que ya están en JSON), en MongoDB que funciona bajo JSON.

- Una vez que teníamos el sistema funcionando, nos pusimos como objetivo secundario reducir el consumo del sistema para que pudiese funcionar con una batería. La idea era usar el comando node.sleep. Este comando tiene como objetivo dormir la ESP8266 entre comunicaciones MQTT. En la fase de test el comando funcionaba de forma inestable, reseteando el sistema de forma aleatoria. Investigamos la razón y descubrimos que el node.sleep tenía un conocido bug. La solución aún no ha sido encontrada y los desarrolladores no parece que vayan a solucionarlo en las siguientes versiones.

Una vez solucionados los errores y teniendo unos scripts capaces de ser usados en múltiples ESP8266, pudimos pasar a la próxima etapa de desarrollo que se encuentra en el siguiente capítulo.

Capítulo 3

Manejo de los datos

En este capítulo veremos la forma de comunicación de los datos, desde su salida de la ESP8266 hasta su llegada a los correspondientes destinos. En concreto hablaremos del protocolo de comunicación MQTT, del Broker usado y del programa de manejo de flujos Node-Red.

3.1. *MQTT*

MQTT es un protocolo de comunicaciones dispositivo-a-dispositivo. Su objetivo es ofrecer una plataforma de comunicación ligera basada en publicación/suscripción. Ésto permite que dispositivos de muy poca potencia puedan comunicarse de manera rápida y ligera. Para que este sistema funcione, es necesaria la existencia de un Broker. Los mensajes publicados a través de MQTT están compuestos por dos partes principales: Tema y Mensaje. La visualización de los mensajes es de estilo JSON lo que permite una lectura cómoda y limpia. La versión utilizada para este proyecto es la 3.1 (la más reciente).

El Broker es el software encargado de hacer llegar a los clientes los mensajes a los que se han suscrito. Una vez establecida la conexión con el Broker ya podemos publicar o suscribir sin problema alguno. La suscripción está relacionada a un Tema específico. Por ello, para poder recibir las publicaciones deseadas, hemos de conocer el Tema con el que van a marcarse los mensajes asociados. El contenido del mensaje puede tener múltiples estructuras. Para este trabajo al final nos decantamos por el uso de JSON, ya que es limpio y ligero.

Para el diseño del sistema hemos establecido los siguientes Temas específicos:

- **/ESP8266/<num esp>/** Este Tema es el establecido para el envío de los datos obtenidos desde los INA219. En caso de que hubiese múltiples ESP8266 simplemente hemos de aumentar el "num esp". En nuestro sistema, éste comienza desde 0.

- **/Node-Red/sleep/<num esp>/** Este Tema es uno a los que la ESP8266 correspondiente se suscribirá. En este caso lo que hace este Tema es establecer el tiempo que ha de dormir la ESP8266 entre envío y envío de datos. El tiempo establecido se envía en segundos. El valor por defecto es 100 pero se puede reducir o aumentar cuanto se desee.

- **/Node-Red/current-enable/<num esp>/** Con current-enable podemos establecer si queremos o no recibir el valor de la corriente de un INA219 especificádo en el cuerpo del mensaje. Su configuración es tal que si está desactivado se reactiva con un mensaje igual que el de desactivación.

- **/Node-Red/power-enable/<num esp>/** Con Power-enable podemos activar o desactivar la recepción de la potencia de un INA219 que se encuentre dentro de "num esp". Para la desactivación o activación simplemente hemos de publicar un mensaje con este Tema y como mensaje el INA219 que queremos cambiar.

- **/Node-Red/voltage-enable/<num esp>/** Con voltage-enable podemos activar o desactivar la recepción del voltaje de un INA219 que se encuentre dentro de "num esp". Para la desactivación o activación simplemente hemos de publicar un mensaje con este Tema y como mensaje el INA219 que queremos cambiar.

La comunicación hacia la ESP8266 la podemos realizar, por el momento, a través de cualquier nodo conectado a la misma red que el sistema. Para ello simplemente tenemos que publicar con uno de los tags explicados previamente. Para las publicaciones necesitaremos tener un cliente MQTT (el usado para este proyecto es *Mosquitto*).

Este cliente puede ser descargado a través del gestor de paquetes de Ubuntu, Debian y otros muchos. Permite realizar publicaciones y suscripciones al Broker que esté funcionando en un puerto conocido.
Contemplamos otras formas de comunicación, como HTTP, pero estas otras no eran ni tan rápidas, ni tan ligeras como MQTT. Al fin y al cabo el consumo energético es un factor importante en nuestro sistema y está directamente relacionado con la ligereza del protocolo de comunicación utilizado.

3.2. *Broker*

En este apartado vamos a ver los diferentes tipos de Broker que hay y cuál elegimos para su uso en el sistema. Ya sabemos que el Broker es el encargado de redirigir las diferentes publicaciones a sus correspondientes suscriptores. La forma de conexión a un Broker es muy sencilla, simplemente tenemos que saber en qué puerto el Broker está escuchando y solicitar conexión para empezar a publicar o suscribir. Los tipos de Broker con los que hemos trabajado son los siguientes:

- **Mosquitto.** Este fue el primer Broker con el que estuvimos trabajando. Este Broker es un proyecto de código libre desarrollado por el proyecto Eclipse[1]. Su descarga e instalación en Linux es muy sencilla ya que el propio gestor de paquetes de Debian lo tiene. También es posible incluir los clientes MQTT, que permiten publicar y suscribir desde consola. Por desgracia decidimos cambiar ya que generaba un retardo mayor que el otros Brokers. Además, la visualización del paso de mensajes era inexistente por tanto en la etapa de desarrollo/testing necesitábamos programas de terceros para poder visualizar el flujo. Es importante saber que Mosquitto de forma nativa solo funciona a través de sockets, esto implica que sólo se puede conectar al Broker en cuestión si está conectado a la misma red que él.

[1] http://mosquitto.org/

Esto se puede solucionar añadiendo la comunicación a través de web-sockets pero ello genera un aumento de inseguridad. Este Broker siempre se puede proteger con usuario y contraseña o cambiar el puerto de escucha.

- **Mosca.** Este Broker de código libre basado en Node.js, permite la creación de un Broker como servidor unitario, "standalone", o como parte de una aplicación más grande. Para nuestro sistema lo usaremos como "standalone" ya que lo vamos a tener funcionando a través de consola con una Raspberry Pi. Este Broker ofrece una gran comodidad para visualización de mensajes aunque en el instalador se recomienda el añadir una herramienta llamada bunyan para observar los mensajes de depuración. Esta herramienta, en unión con mosca muestra toda la información de paso de mensajes con un código de colores asociado para facilitar su lectura. La razón principal por la que al final elegimos este Broker y no Mosquitto es por unidad de lenguajes, puesto que estamos trabajando con Node-Red y él también esté basado en Node.js. Esto facilita la instalación de nuestro programa en cualquier parte puesto que ya no depende del sistema, lo único requerido para su instalación en cualquier sistema es tener Node.js instalado. Mosca puede ser protegido con usuario y contraseña y también es posible cambiar los puertos de escucha.

El Broker es una de las piezas principales del sistema ya que sin MQTT no tendríamos una forma tan eficiente de comunicación. Cabe constatar la sencillez con la que se puede instalar el sistema y tenerlo funcionando de forma instantánea. El Broker que usamos escucha por el puerto 8266 y no está protegido con usuario y contraseña.

3.3. *Node-Red*

Node-Red es una herramienta para conectar diferentes elementos hardware, APIs y aplicaciones software de forma sencilla a través de una interfaz web. Cada uno de los diferentes elementos que podemos conectar son llamados nodos. En la imagen siguiente vemos la conexión entre dos nodos de ejemplo, con ello podemos ver la sencillez con la que se unen dos nodos cualquiera.

Figura 3.1: *Ejemplo de flujo de Node-Red*

Una de las características más importantes que tiene Node-Red es la capacidad de añadir nuevos nodos para aumentar las posibilidades de la aplicación. Para el proyecto se han usado dos nodos que se encuentran fuera del paquete oficial de Node-Red. Los paquetes que usamos fueron los siguientes:

- **node-red-node-mongodb.** Este paquete nos permite realizar inserciones a colecciones específicas o realizar lecturas de base de datos a través de Node-Red.
- **node-red-contrib-freeboard.** Este paquete nos permite enviar flujos de datos a Freeboard y que éste los reciba como un input normal de su aplicación.

La instalación de ambos fue sencilla ya que estos han sido desarrollados por la misma empresa que Node-Red, por tanto con usar el centro de descargas que npm (gestor de paquetes de node.js) tiene ya los podemos tener instalados en nuestro sistema. El potencial de esta aplicación es realmente grande, pero en nuestro sistema solo hemos necesitado crear flujos de complejidad media. Sin embargo este programa permite realizar un gran número de tareas de manera sencilla e intuitíva.

Creemos que es en esta parte de la aplicación en la que podemos generar una mayor automatización, ya que mediante la creación de alarmas o el envío de mensajes controlados por Node-Red podemos realizar cambios en la configuración de las ESP8266, sin necesidad de estar en la misma red que el sistema. Los mensajes enviados a Node-Red pueden ser enviados a través de plataformas conocidas como mail o Twitter.

En la siguiente imagen vemos la configuración que se ha utilizado para la ejecución del sistema completo con una sola ESP8266 proveyendo información.

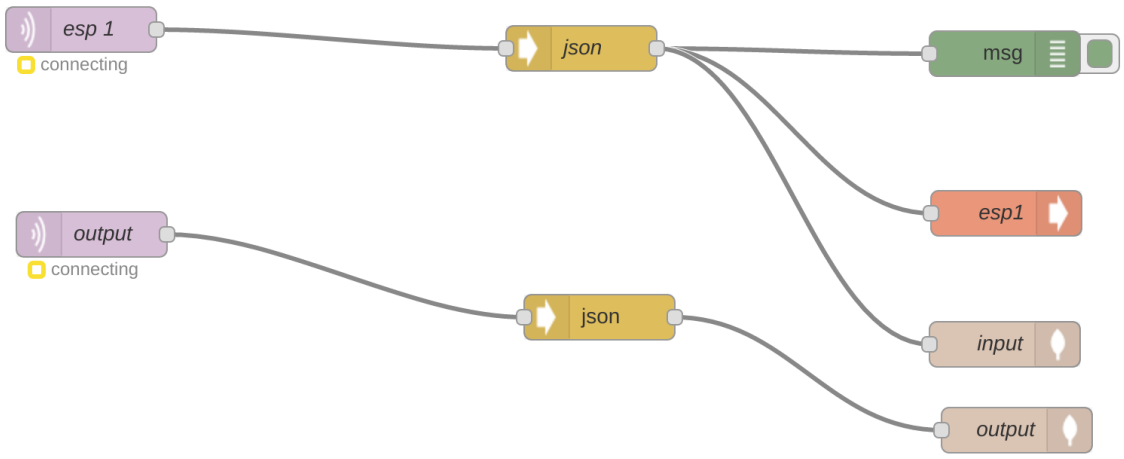

Figura 3.2: *Flujo de Node-Red del sistema*

Como podemos ver tenemos dos nodos de entrada. Las entradas son de tipo MQTT y se suscriben a las publicaciones que deseemos. En este caso el esp1 se suscribe al Tema "/ESP8266/<num esp>/". Este Tema, como ya sabemos, es el encargado de enviar los datos capturados de los INA219. En la imagen se ve que antes de ser pasado a Freeboard o a MongoDB, éste pasa previamente por un conversor a JSON. Sabemos que el mensaje MQTT enviado está en formato JSON pero el cuerpo del mensaje es convertido a string durante la transmisión, así que simplemente lo volvemos a su estado inicial.

La segunda entrada que tenemos es aquella que está suscrita al Tema "/Node-Red/#/". El símbolo # es el indicador que tiene MQTT para poder suscribirse a múltiples publicaciones a la vez. En este caso nos suscribimos a todos los Temas que comienzan por "/Node-Red/". Con ello nos cercioramos de que los mensajes que mandemos a la ESP8266 queden almacenados en la base de datos en caso de necesitar consultarlos.
Después de la conversión de ambas entradas a JSON, éstas ya son conectadas a sus correspondientes nodos de salida. Con el fin de mejorar la seguridad de nuestro sistema se decidió proteger Node-Red con un usuario y contraseña. Es importante proteger esta parte del sistema puesto que es aquí es donde configuramos los datos que recibirá la interfaz. Además al publicar la IP para que la página web sea pública, Node-Red queda descubierto. Por tanto la protección con usuario y contraseña es vital.

3.4. *Desafíos encontrados*

Uno de los principales desafíos que nos encontramos al comenzar a usar MQTT fue hacerlo lo más seguro posible. Para ello cambiamos el puerto de escucha del Broker, del 1883 al 8266, después lo protegimos con usuario y contraseña. Durante la fase de testing el sistema se comportó de forma estable. Sin embargo, ante una prueba real, se observó que al estar protegido con contraseña había problemas en la recepción de los mensajes. El problema era que el Broker no recibía todos los mensajes que la ESP8266 enviaba. Después de múltiples pruebas nos dimos cuenta que la cause eran las contraseñas. Una vez identificado el problema se investigó una solución. Se descubrió que se trataba un fallo conocido y que se esperaba solucionar en las siguientes versiones del firmware. Así que aunque actualmente no esté protegido con usuario y contraseña, sabemos que esta característica estará disponible en un futuro cercano.

Node-Red no fue empleado durante todo el desarrollo del sistema; en las primeras etapas no usábamos Node-red ya que la interfaz que usábamos en el momento no lo necesitaba y no teníamos la persistencia (almacenamiento en base de datos) en desarrollo. Una vez que la persistencia fue el siguiente objetivo ya si era obligado el uso de Node-Red, ya fuese por su sencillez o por la posibilidad de expandir el sistema si fuese necesario.

Capítulo 4

Visualización y Almacenamiento de los datos

En este capítulo se explica como mostramos los datos obtenidos de los INA219 que hemos recibido a través de Node-Red. También se explican las diferentes formas estudiadas para el muestreo de los datos y las diferentes bases de datos utilizadas.

4.1. Muestreo de los datos

Con el fin de mostrar los datos debemos saber exactamente los requisitos necesarios para que la interfaz funcione correctamente. Dichos requisitos que necesitamos que cumpla la interfaz son los siguientes:

- La primera necesidad es una interfaz que permita de manera cómoda, la recepción de datos estructurados en tiempo real. Los datos que recibiría serían los recogidos por la ESP8266. Toda página web puede ser codificada para que reciba datos en tiempo real, pero requiere un desarrollo específico adaptado a la aplicación. Por se decidió usar plantillas definidas para IoT. Estas plantillas han sido desarrolladas por diferentes empresas con el fin de poder visualizar los datos llegados de los múltiples dispositivos.

- Otro de los requisitos importantes es la capacidad de personalización. Queremos una interfaz que se pueda editar de forma cómoda sin necesidad de modificar el código. También deseamos que haya múltiples formas de mostrar nuestros datos, ya sea con mediante gráficas o de forma numérica. Las plantillas de IoT de las que hemos hablado antes tienen varias formas de representación de datos. Muchas de ellas además permiten crear plug-ins en caso de que no muestren los datos de la forma deseada por el usuario.

- Unos de los requisitos opcionales que nos gustaría tener en nuestra aplicación es seguridad. Es conveniente proteger el acceso a la aplicación a través de la creación de un usuario o contraseña. En las diferentes plantillas encontramos diferentes formas de seguridad.

Con los requisitos bien especificados, se realizó una búsqueda de aplicaciones ya desarrolladas que los cumpliesen. Después de un largo tiempo de investigación se encontraron tres posibles candidatos, Emoncms[1], Graphene[2] y Freeboard[3]. Estas aplicaciones fueron aprobadas por el tutor y presentan diferentes capacidades, así como el estar desarrolladas en varias tecnologías. Ahora estudiaremos las herramientas seleccionadas y su proceso de selección.

A partir de ahora nos referiremos a la aplicación de muestreo de datos como *Dashboard*. Los diferentes dashboards que nos encontramos en Internet presentan una serie de capacidades comunes, como la visualización de datos en tiempo real.

[1] https://emoncms.org
[2] http://jondot.github.io/graphene/
[3] https://freeboard.io

4.1.1. Graphene

Este Dashboard desarrollado por jondot[4] ha sido codificado en Ruby. La plataforma tiene potencial ya que Ruby permite una gran variedad de mejoras en todos los campos web. El mayor problema que presenta el Dashboard es la necesidad de tener un gran conocimiento de Ruby. Dada la falta de tiempo y la falta de soporte, y que no existe una comunidad detrás de este, se decidió rechazar esta opción.

Figura 4.1: *Dashboard de ejemplo Graphene*

[4] https://github.com/jondot

4.1.2. Emoncms

Este dashboard pertenece a Emon, una empresa que tiene por objetivo crear una plataforma propietaria que permita monitorizar toda la información de los sensores que uno puede colocar en una casa. La empresa proporciona todo lo necesario para la creación del sistema, ya sea Hardware o Software. Toda la información que los diferentes elementos generan son redirigidos a Emoncms Dashboard.

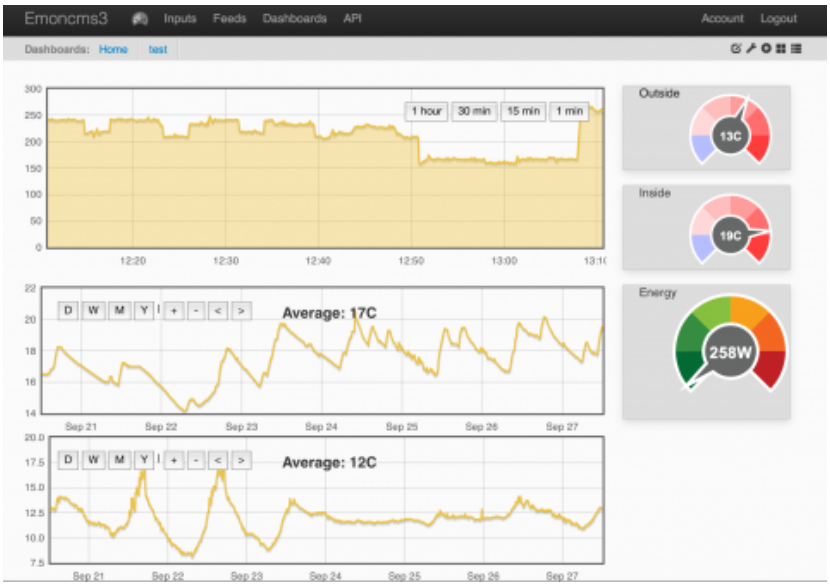

Figura 4.2: *Dashboard de ejemplo en Emoncms*

Como podemos observar en la imagen anterior este Dashboard nos permite la creación de diferentes formas de representación de datos. Este fue el elegido al principio puesto que tiene una muy buena reputación de representación de datos relacionados con consumo. Permite la creación de plug-ins personalizados, gracias a él tuvimos la oportunidad usarlo de forma sencilla. Para ello usamos PahoMQTT que es una librería que ofrece un cliente MQTT a través de websockets. Éste conectado a Emoncms, permite la suscripción del dashboard a los Temas que deseemos.

Con este dashboard hicimos pruebas de estudio para comprobar la robustez de nuestro sistema y no alcanzó nuestros estándares. El sistema, a pesar de ser funcional, generaba una serie de problemas como perdida de algunos datos en tiempo real.
Otro de los grandes problemas que vimos fue que la comunidad que daba soporte a este dashboard estaba completamente centrada en los dispositivos de Emon y esto generaba una gran dificultad a la hora de solucionar problemas de compatibilidad con sistemas externos. Además para dar uso a Emoncms no era necesario el uso de Node-Red, lo cual dejaba la persistencia sin asignar. Aunque lo que nos hizo rechazar finalmente este dashboard fue la vulnerabilidad del sistema, puesto que no ofrecía ninguna forma de protección del dashboard en sí ni de los flujos de datos entrantes.

4.1.3. Freeboard

Este dashboard desarrollado en Javascript ofrece una plataforma de muestreo de datos en código libre y a su vez permite acceder a través de su pagina web para que sea accesible a través de Internet. Este dashboard acepta la creación de nuevos plug-ins con los que mejorar la forma de muestreo de datos o la forma de recibirlos. Lo único que no ofrece es la creación de usuarios y la protección del dashboard completo, sin embargo más adelante se explicará cómo protege las entradas de datos.
Una vez determinamos que Emoncms no cumplía con los requisitos que buscamos, comenzamos la investigación de Freeboard. Como entrada de datos empezamos a usar PahoMQTT, que como hemos explicado es un plug-in que permite la comunicación con el broker a través de web-sockets. Una vez testado con resultados positivos, decidimos hacer una prueba de medición de un computador real, con el fin de probarlo bajo estrés. Durante la realización de la prueba comprobamos el gran retardo entre el envío de los datos por parte de la ESP8266 y su recepción en Freeborad. Éste era debido a la conexión a través de web-sockets. Dicho retardo, sumado a la posible inseguridad del sistema al tener una conexión a través de web-sockets, llevó a buscar una nueva forma de comunicación con Freeboard.

Figura 4.3: *Dashboard final Freeboard*

En la imagen anterior podemos ver el dashboard que observamos en la fase de testing. Tenemos tres paneles principales y dentro de ellos existen varios bloques en los que podemos proyectar nuestra información de varias maneras. También podemos ver el panel de configuración en el que visualizamos las entradas de datos, así como la posibilidad de guardar el dashboard o cargarlo.

Después de un tiempo investigando se encontró una nueva forma de comunicación con Freeboard, esta vez a través de Node-Red. Esta forma de comunicación no era con un plug-in, sino que era otra instalación de Freeboard. Dicha instalacion viene incluida en un nuevo nodo que añadimos a Node-Red. De esta forma conseguíamos una conexión segura.

La diferencia con PahoMQTT, es que la entrada de datos se hace de forma diferente, provocando una mejora en la seguridad. En PahoMQTT el cliente de MQTT se configura para su conexión a través de web-sockets. El problema es que cualquier persona viendo el dashboard puede acceder al panel de configuración, y como consecuencia podrían borrarse las entradas y sería necesario volver a configurarlas por completo. Con el nodo de Node-Red, como hemos visto antes, al añadir un nuevo nodo a nuestro flujo, éste aparece en las entradas sin necesidad de configuración en Freeboard.

4.2. Persistencia de los datos

Una vez todo el sistema ya estaba funcionando, nos centramos en establecer una forma de almacenamiento de datos con el fin de guardar la toda la información intercambiada en el sistema. El objetivo de éste es tener un log que nos permita revisar los datos intercambiados en caso de necesidad. Si se detectase algún fallo, tenemos la oportunidad de saber el momento exacto en el que se ha producido y cuales son los datos previos al fallo.
Antes de pensar que tipo de base de datos vamos a usar, tuvimos que decir en que parte de nuestro sistema lo alojaríamos. El almacenamiento podría haber sido en una de las tres partes de nuestro sistema:

- En la primera parte del sistema no es conveniente, ya que para la ESP8266 no tendría mucho sentido el enviar los datos sin haber sido tratados. Además el consumo de la placa aumentaría sustancialmente, haciendo que el sistema consumiese más de lo debido. La razón principal para no usarlo, fue por el hecho de que esta parte de nuestro sistema es la que se replica para poder monitorizar cuantos sub-dispositivos queramos.

- La parte de nuestro sistema que contiene el Broker es un buen candidato para ser encargado de la persistencia. Ambos Brokers tenían la capacidad de persistencia a múltiples bases de datos diferentes, el único problema que se planteaba era la limpieza de los datos introducidos.

Un broker recibe más mensajes de los que nos interesa almacenar, desde los de keep-alive hasta los mensajes de envío. Éstos aumentan en gran cantidad el número de mensajes almacenados, dificultando futuras búsquedas en caso de problema.

- En cada una de las plataformas de muestreo de datos hemos buscado formas de almacenarlos. Cuando se evaluó Emoncms, se investigó cómo realizar envíos a las bases de datos desde el propio dashboard y funcionaba a la perfectamente. Cuando estuvimos probando Freeboard sin Node-Red, se buscaron formas de establecer persistencia, pero este dashboard no nos lo ofrecía. Por último, cuando nos decidimos a usar Node-Red, la idea de la persistencia se hizo mucho más sencilla. Node-Red tiene múltiples nodos para su uso o inserción en múltiples bases de datos. Dada la comodidad de tener datos estructurados en Node-Red, podemos insertarlos en la base de datos deseada.

Una vez teníamos decidido que Node-Red iba a ser el encargado de almacenar en una base de datos, estudiamos en que formato nos llegaban éstos. Puesto que el mensaje recibido por MQTT es parseado en Node-Red a JSON, nos encontramos con un bloque de datos con un formato conocido. La base de datos que trabaja mejor los datos de tipo JSON es MongoDB[5].

Node-Red tiene un conjunto de nodos específicos para operaciones en MongoDB. Para el sistema estamos usando solamente el nodo de inserción en base de datos. MongoDB es una base de datos documental, esto significa que todas las entradas se almacenan en documentos, en este caso JSON. Con el fin de reducir el espacio de almacenamiento en realidad se usa BSON, es decir JSON en binario. MongoDB es una base de datos basada en colecciones, cada una de ellas almacena datos que normalmente tengan una relación entre ellos, como las tablas de una base de datos relacional.

Otra de las características importantes que ofrece MongoDB es no necesitar un esquema previo. Esto permite al usuario poder introducir en una misma colección, datos que aún estando relacionados tengan diferentes campos entre ellos.

[5] https://www.mongodb.com/es

Para nuestro sistema, ofrece la posibilidad de modificar el formato de datos recibidos sin necesidad de preocuparnos por alterar MongoDB.

En el sistema lo que verdaderamente nos interesa es almacenar los datos y poder visualizarlos en caso de necesidad. Con MongoDB, el almacenamiento lo hemos conseguido de una forma sencilla ya que el formato de entrada coincidía con el de esta base de datos. Con respecto a la visualización, se puede realizar a través de la consola de MongoDB. Con el fin de comprobar que los datos estaban siendo correctamente insertados, hemos usado una herramienta externa llamada RoboMongo[6].

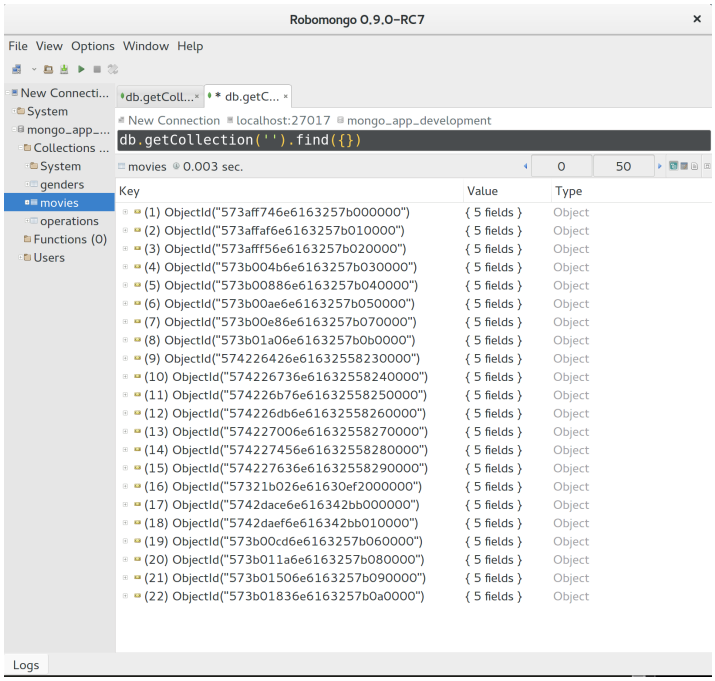

Figura 4.4: *Ejemplo de visionado de RoboMongo*

En la imagen anterior podemos ver un ejemplo de la interfaz de RoboMongo. Esta aplicación te permite realizar múltiples búsquedas así como editar o insertar datos en caso de necesidad. Podemos ver todas las posibles colecciones de la base de datos, así como conectarse a múltiples bases de datos sin problemas.

[6] https://robomongo.org/

En nuestro sistema hemos usado dos colecciones diferentes: una para los datos recibidos a través de la ESP8266 que hemos usado para nuestros test. Y la otra para los mensajes enviados por el usuario a la ESP8266 con el fin de configurarla. Gracias a Node-Red tenemos la posibilidad de elegir el número de colecciones que deseemos utilizar. Por ejemplo, cuando el sistema sea escalado a gran nivel, podremos usar diferentes colecciones para almacenar datos de las ESP8266 según la relación entre ellos. Con Node-Red almacenar datos en múltiples colecciones es tan sencillo como añadir más nodos y configurarlos de forma que realicen inserciones en las colecciones que deseemos.

Capítulo 5

Conclusiones

En este proyecto se ha desarrollado un sistema capaz de monitorizar un dispositivos en corriente continua y/o sub-dispositivos dentro de grandes computadores. Ha sido desarrollado al completo y hemos cumplido con los objetivos establecidos al comienzo del proyecto. El sistema desarrollado es capaz de:

- Capturar datos de potencia, corriente y voltaje de todo dispositivo alimentado por corriente continua o sub-sistema a través de un INA219.
- Una vez recogidos los datos, éstos son publicados en los correspondientes destinos de manera segura a través de MQTT, en un puerto específico.
- Los datos son mostrados en una pagina web de forma segura, lo que evita que alguien pueda dañar las entradas de éstos. A su vez, son almacenados con el fin de tener una persistencia habilitada en caso de necesitar un análisis a posterior.

El proyecto no sólo es funcional sino que se ha conseguido que cumpla los siguientes objetivos principales:

- El sistema es robusto, puesto que usamos tecnologías en desarrollo, lo que implica una comunidad detrás con actualizaciones para que el sistema se mantenga al día.

- La escalabilidad se consiguió gracias a la posibilidad de multiplicar la cantidad de ESP8266 que puede manejar nuestro sistema sin necesidad aumentar el resto de dispositivos Hardware.

- La instalación del sistema es muy sencilla; existen unas instrucciones con todo lo necesario para un correcto funcionamiento del proyecto.

5.1. Componentes Hardware

El sistema está compuesto de tres elementos Hardware:

- ***INA219***: Dispositivo que obtiene los datos que serán mostrados posteriormente. Precio: 2€.

- ***ESP8266***: Placa encargada de recibir los datos por I^2C y transmitirlos por MQTT. Precio: 4€.

- ***Raspberry Pi 2b***: Este ordenador de placa única es el encargado de realizar todo el manejo de los datos. Aquí tendremos el Broker MQTT, Node-Red y Freeboard. Para acceder a ello fuera de la red, simplemente habría que publicar la IP. Precio: 35€.

5.2. Conocimientos adquiridos y usados

Durante la realización de este proyecto hemos tenido la oportunidad de aplicar múltiples conocimientos adquiridos en la carrera.

- La asignatura de Ingeniería del Software nos ha ayudado para realizar una documentación clara y eficiente. Sistemas Web también aumentó nuestros conocimientos con respecto a la redacción de documentos.

- Con respecto a la programación, las diferentes asignaturas que hemos cursado a lo largo de la carrera me han provisto de la base necesaria para poder aprender cualquier lenguaje de una manera más sencilla. Esto se debe a que hemos trabajado con los diferentes tipos de programación, ya sea funcional, declarativa u orientada a objetos.

Con respecto a los conocimientos adquiridos, hemos tenido la oportunidad de aprender sobre los siguientes campos de la informática:

- ***Protocolos de comunicación***: Para este proyecto hemos aprendido dos protocolos de comunicación importantes, I^2C y MQTT. El protocolo I^2C lo conocíamos de la universidad pero solo teóricamente. Aquí tuvimos la oportunidad de trabajar con él y eso nos permitió su comprensión de forma rápida y efectiva. Con MQTT descubrimos un protocolo de comunicación que permite publicar y suscribirse a mensajes (muy usado en el ecosistema IoT). Esto permite una recepción y envío de mensajes rápida y breve, que reduce el gasto de consumo por comunicación. El uso de este protocolo también nos brindó la oportunidad, aunque fuese por necesidad, de aprender a usar un Broker (encargado de redirigir los mensajes publicados hasta los suscritos).

- ***Lenguajes de programación***: En este proyecto no hemos necesitado usar una gran cantidad. Principalmente ha sido LUA, un lenguaje de programación adaptado para programar pequeños dispositivos, en nuestro caso una ESP8266. A pesar de que no hemos usado más lenguajes, Node-Red es una tecnología que permite programar flujos entre aplicaciones de forma muy sencilla y efectiva. Gracias al formato usado para el envío de los datos hemos podido trabajar con JSON.

- ***Tecnologías de almacenamiento y muestreo***: Este proyecto nos ha brindado la oportunidad de investigar diferentes formas de mostrar y almacenar datos. Hemos tenido la oportunidad de trabajar con MongoDB para almacenar y con Freeboard y Emoncms para el muestreo de los datos.

5.3. Posibles mejoras y objetivos futuros

Una vez que MicroPython haya alcanzado un mayor nivel de madurez, nos gustaría probarlo como sustitutivo de LUA en nuestro sistema. Ya que Python tiene un gran potencial. También nos gustaría mejorar el código de LUA una vez se solucione el error de node.sleep puesto que creemos que eso mejoraría la utilidad del proyecto, permitiendo usar una batería como fuente de alimentación sin preocuparnos por su rápido consumo.

Uno de los objetivos futuros es la mejora de comunicación de Node-Red con cada una de las ESP8266. La idea es crear alarmas que permitan la comunicación en esa segunda vía sin necesidad de encontrarse en la misma red.

Por último, estamos esperando una mejora en el firmware de LUA para poder volver a introducir seguridad en las comunicaciones con MQTT. A pesar de que la comunicación se realice en local y con puertos personalizados, nos gustaría mejorar la seguridad lo máximo posible.

Bibliografía

[1] *INA219 - DataSheet*. http://www.ti.com/lit/ds/symlink/ina219.pdf.

[2] *NodeMCU - Documentación*. https://nodemcu.readthedocs.io/en/dev/.

[3] *MQTT - Documentación*. http://mqtt.org/documentation.

[4] *Mosquitto - Broker*. http://mosquitto.org/.

[5] *Mosca - Broker*. https://github.com/mcollina/mosca.

[6] Node-Red. http://nodered.org/.

[7] I^2C - Module. https://nodemcu.readthedocs.io/en/dev/en/modules/i2c/.

[8] MQTT - Module. https://nodemcu.readthedocs.io/en/dev/en/modules/mqtt/.

[9] Emoncms - Dashboard. https://emoncms.org/.

[10] Freeboard - Dashboard. https://freeboard.io/.

Apéndice A

Introduction

IoT (Internet of Things) is the network of physical objects. In these systems we see different elements, each of them with a clear objective:

- Sensor Node: in charge of the data gathering, in this case the Hardware devices used are the INA219 and the ESP8266.
- Broker/Gateaway: Hardware in charge os hosting the necessary software for the project. In this case a Raspberry Pi was used.
- Visualization: The way the captured data is showed. For this project Freeboard Dashboard was used.

A.1. Objectives

The main objective of this project is to deliver a robust and scalable system, capable of monitoring any system running in direct current or sub-systems on which we want to obtain specific information.
My objective is for this project to be used by any person, company or university that could have a use for it. To that end, the system has to be as easy to use as possible so a person with an entry level of informatics could install it. In addition a high skilled person has to be able to get the maximum possible out of the project in a comfortable and efficient way.

In this university there are real projects in which this system or at least a part of it would prove useful to its development or monitorization. In addition the possibility of personalization gives the system the ability to adapt to any external system without much struggle. At last but not least the web interface allows to access it from anywhere. Also this interface can be modified from within, without the need of altering its code. This system uses Node-Red so alarms can be created to automatize the resolution of a problem, that plus the persistence deliver a system capable of be checked in case of an error.

A.2. Technologies of the System

In this system we have various Hardware and Software elements. In the next diagram we can observe how the system works, not focusing in how the different parts communicate with each other. Futher down the document i will complete this diagram so the idea is complete.

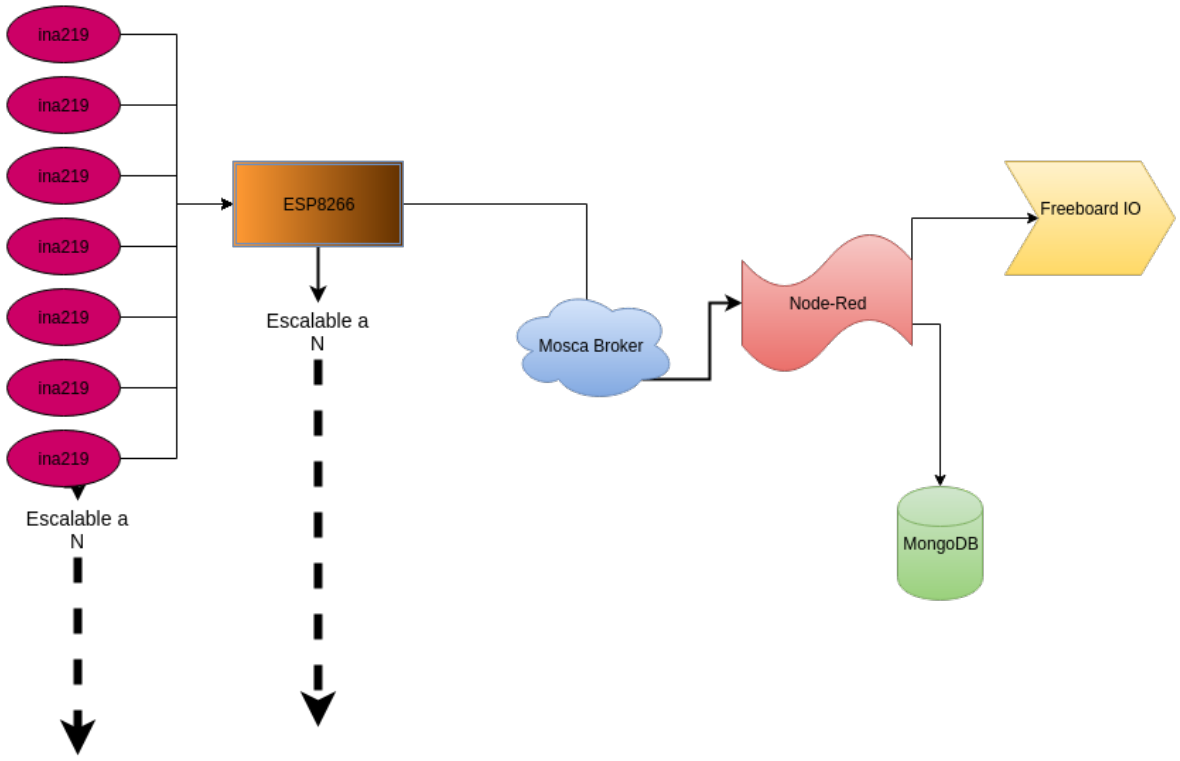

Figura A.1: *One-way Communication diagram*

In this diagram the flow starts with the INA219. This chips are the ones tasked with doing the measurements of all the data that we are going to be managing during the whole system. The ESP8266 is the one that asks for all the information to all the different INA219 that it is connected to. This board allows different communication methods , in addition to a WI-FI module to transmit data wirelessly.

All the information is received at the Broker. Is important to acknowledge the capability to replicate the system described in the previous paragraph as many times as it is needed. This allows the monitorization of as many devices as we deemed fit. The Broker is the one who redistributes the information so anyone who wants to received does. In this case Node-Red is the one who receives all this information, parses it and finally ensures that this information arrives at their destinations. These are MongoDB, given the need for persistence, and Freeboard, because this one is tasked with giving a visualization of the data to the user.

A.3. 2-way communication

The system has been designed to permit a 2-way communication. This allows the user to edit certain values of the ESP8266, like the refresh time or the information . The communication on the other way is short, given the origin is in Node-Red an it is received by the ESP8266. The ESP8266 is the one in charge of making the requests for the data to the INA219 connected to it, that is why he is the one who can modify the values of the system. If we have a system with multiple ESP8266 we can modify each with different configurations.

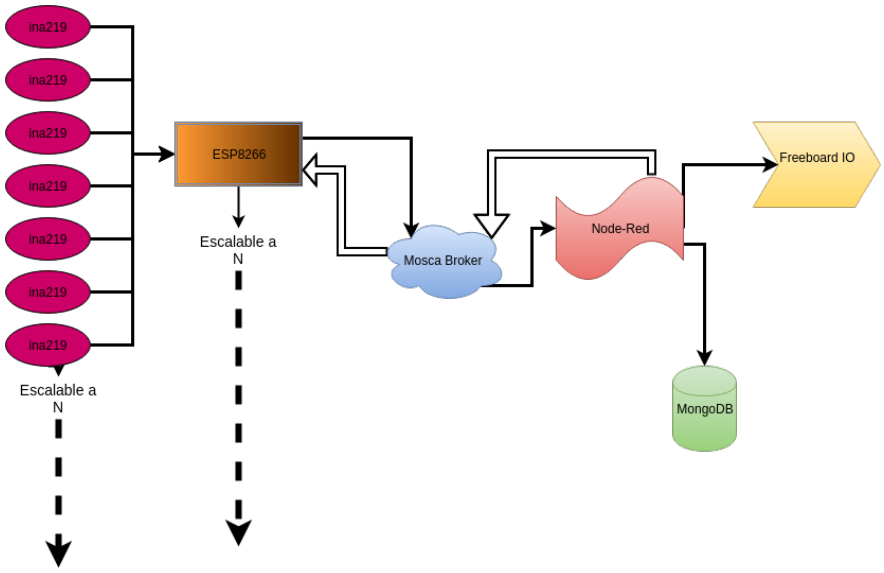

Figura A.2: *2- way Communication Diagram.*

A.4. Communication Protocols

In the next diagram we can observe the whole system with it 2-way communication and the different protocols used to achieve that communication. This system has been developed to completion.

As you can see the low-level communication is the one connecting the INA219's with the ESP8266. The protocol used in this case is I^2C. For the rest of the communications it was decided to use MQTT, which is a communication protocol based on the publication/subs-

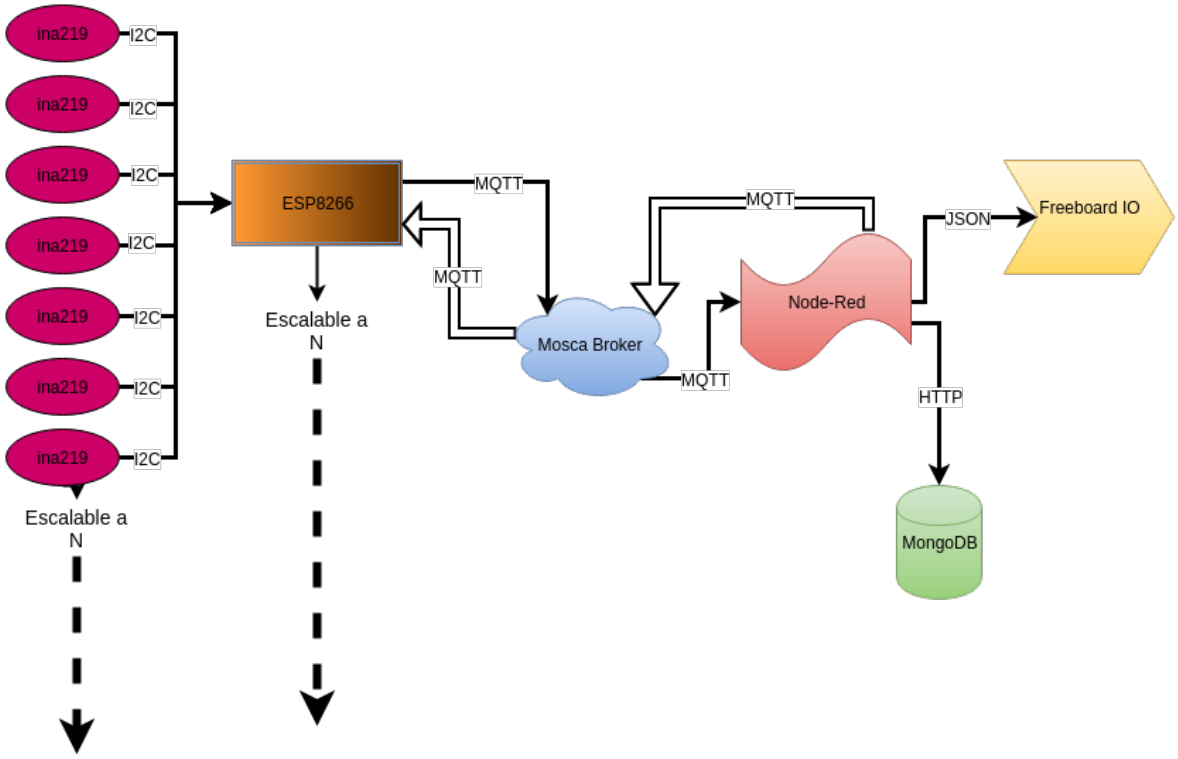

Figura A.3: *Complete system diagram.*

cription method. To sum up Node-Red uses HTTP to communicate with the MongoDB database, which allows for it to be hosted somewhere else. For Freeboard a JSON based protocol is used, taking into account that both of them work in localhost.

Apéndice B

Conclusions

In this project I have been capable of developing a system which monitors a system connected to direct current and/or sub-systems inside large computers. The system has been fully developed and we have accomplished all the objectives we established at the beginning of the project.
The system developed is capable of:

- Capture data of power, current and voltage from every device powered by direct current. Data captured through INA219s.
- Once the data has been retrieved, it is published to their pertinent destinations through a secure MQTT connection in an specific port.
- The data is shown in a web-page with secure inputs so there cant be any tampering with it. In the meantime a copy of the data is being stored on an external database.

The project is not only functional, also I have accomplished the following main objectives:

- The system is robust, thanks to the use of technologies in development. That implies the existence of a community behind, helping with updates to keep up to date the programs.

- The scalability for this system was accomplished thanks to the capability of multiplying the quantity of ESP8266 with INA219s attached without the need of increasing the number of the rest of Hardware devices.
- The installation of the system is quite easy, there are instructions with all the necesary information to have the system up and running.

B.1. Hardware Components

The system is composed of three Hardware elements

- **INA219**: Chip that calculates the data that will be shown later in the project. Price $\tilde{2}$€.
- **ESP8266**: Board in charge of asking for the data, and recieveng through I^2C, and transmiting them through MQTT. Price 4€.
- **Raspberry Pi 2b**: This board computer is in charge of managing the data. In here we have to have installed the MQTT Broker, Node-Red and Freeboard. To access it we would just need to publish the IP. Price 35€.

B.2. Acquired and used knowledge

During the development of the project I have had the opportunity to give use to the knowledge I gained during my degree.

- For the redacting of the documentation, the class "Software Engineering"with all the documentation needed to be written it was a great help to redact clearly and efficient documents. Web systems helped in the same way given the project we had to develop and the documentation that came with it.

- Related to programming, all the different subjects I have taken during the degree may have not taught me the specific languages I have used in this project. But it has taught me great ways to learn a new language faster than normal, starting with easier identification of the language as functional, declarative or object oriented.

In relation to the acquired knowledge,I have had the opportunity to learn about every area of informatics:

- **Communication protocols**: During the project I have learned two important communication protocols I^2C and MQTT. The I^2C protocol was known to me from different Hardware related subjects, but only in theoretical applications. However in this project I had the chance to work with it. With MQTT I discovered a communications protocol that allows you to publish and subscribe to certain topics. This makes the transmission fast, brief, which overall can reduce the consumption of our system. The use of this protocol requires a broker. Which is the one that redirects the published data to their subscriptions.

- **Programming Languages**: In this project we have not needed to use a lots of coding. The main language has been LUA, a programming language used to program low-end devices , in this case an ESP8266. Even though we haven used any more languages, Node-Red is a technology that allows to program flows between applications easily and effectively.

- **Technologies of visualization and storing**: This project has given me the opportunity of researching the different ways to showcase the data and store data. I have worked with MongoDB to store, Freeboard and Emoncms for the showcase of the data.

B.3. Enhancements and possible future objectives

During the development of the project I have had to make decisions about the development that I would like to re-examine in the future. In the selection of the programming language to use, I would like to try MicroPython once its development has been concluded or at lest matured. In relation to that i am waiting for LUA to resolve the node.sleep bug so the board can sleep between publications. That will undoubtedly reduce the consumption hence giving the chance for a use of a battery as a power supply.

One of the future objectives is the enhancement of the communications from Node-Red to every ESP8266 on the systems. The idea is to create alarms that allow the program to operate more seemingly in the second way of communication so you will not have the need to be in the same network.

At last but not least im waiting for an update in the LUA language so we will be able to use again password protection in MQTT communications. Even though the communication is happening in localhost and with personalized ports, I would like enhance the security as much as possible.

Apéndice C

Instrucciones de instalación

Se proporcionan, como documentación adicional, los pasos básicos necesarios para la instalación de los distintos elementos utilizados en el desarrollo del trabajo.

C.1. Introducción

El proyecto tiene como objetivo la creación de un sistema de monitorización de dispositivos en corriente continua. Para ello usamos los siguientes recursos Hardware:

- INA219
- NodeMCU ESP8266 con Lua
- Raspberry Pi 2B

Para que la implementación de este sistema sea lo más intuitiva y sencilla posible usamos los siguientes recursos software:

- Node-Red
- Mosca
- MongoDB
- Freeboard

C.2. Instalación

Para la instalación de todo el sistema necesitaremos nodejs, se recomienda una versión 0.12 o superior.

C.3. Node-Red

Ejecutamos el siguiente comando en home:

```
$ sudo npm install -g --unsafe-perm node-red
```

Una vez instalado, nos vamos al directorio de node-red y ejecutamos los siguientes comandos para instalar los nodos necesarios en la conexión con Freeboard y MongoDB.

```
$ sudo npm install node-red-contrib-freeboard
$ sudo npm install node-red-node-mongodb
```

C.4. Mosca

Mosca es el Broker que vamos a utilizar para el manejo de los mensajes MQTT de nuestro sistema. En home ejecutamos:

```
$  sudo npm install mosca bunyan -g
```

Bunyan es un programa que permite visualizar los logs de mosca de manera más guiada y cómoda.

C.5. MongoDB

MongoDB es el servidor que utilizaremos para la persistencia, éste se puede cambiar por cualquier otro en Node-Red.

```
$  sudo apt-get install -i mongodb
```

Una vez completada la instalación vamos a crear la base de datos y las coleciones que vamos a usar para la persistencia de nuestro sistema. En el shell de mongo ejecutamos los siguientes comandos:

```
$ use ESP8266
$ db.createCollection("input")
$ db.createCollection("output")
```

C.6. Freeboard

Freeboard no requiere instalación específica ya que lo hemos instalado con el paquete del node-red-contrib-freeboard, cuya carpeta se encuentra en la siguiente dirección:

```
$ .node-red/node-modules/node-red-contrib-freeboard/node-modules/freeboard
```

En caso de que se quiera añadir plugins o editar el código.

C.7. Ejecución

Para ejecutar hemos de cargar los archivos del repositorio en la ESP8266. Para ello usamos el ESPlorer [1]. Después hemos de levantar Node-Red y Mosca, en terminales separadas

```
$  node-red
(otra consola)
$  mosca -p 8266 --very-verbose | bunyan
```

[1] http://esp8266.ru/esplorer/

Una vez que ambos sistemas están levantados y funcionando accedemos a Node-Red y cargamos el flujo que está en la carpeta Node-Red. Hay que copiar dicho archivo a

```
"~/.node-red/lib/flows"
```

Cuando tengamos los flujos sólo tenemos que hacer el deploy.

Para acceder a Freeboard nos vamos a la siguiente url " http://127.0.0.1:1880/freeboard ". La configuración de los flujos de Node-Red aparecerán como entradas en Freeboard. Para cargar el dashboard simplemente presionamos al botón de "Load Dashboard" y seleccionamos el .json de la carpeta "freeboard-dashboard"

Apéndice D

JSON. Mensaje de ejemplo

Este es un ejemplo de mensaje JSON enviado desde una ESP8266 y recibido por Node-Red. Todo ello ha sido comunicado a través del protocolo MQTT. Como vemos algunos de los INA219 envían "-1" esto significa que están deshabilitados (realizado a través de Node-Red).


```
{
  "topic": "/ESP8266/0",
  "payload": {
    "ina-1": {
      "current": 656.18,
      "voltage": 11892,
      "power": 7789.61
    },
    "ina-2": {
      "current": 285.18,
      "voltage": 3376,
      "power": 949.61
    },
    "ina-3": {
      "current": -1,
      "voltage": 1,
      "power": -1
    },
    "ina-4": {
      "current": -1,
      "voltage": 1,
      "power": -1
    },
```

```
"ina-5": {
      "current": -1,
      "voltage": 1,
      "power": -1
    },
    "ina-6": {
      "current": -1,
      "voltage": 1,
      "power": -1
    },
    "ina-7": {
      "current": -1,
      "voltage": -1,
      "power": -1
    },
    "ina-8": {
      "current": -1,
      "voltage": -1,
      "power": -1
    }
  },
  "qos": 1,
  "retain": false,
  "_topic": "/ESP8266/0/",
  "_msgid": "353262e.cacd9e"
}
```

Apéndice E

Código LUA. Comunicación I^2C

A continuación se muestra la clase en LUA desarrollada que contiene toda la funcionalidad necesaria para la obtención de los datos de los INA219. Dicha captura de datos se realiza a través de I^2C. En el Apéndice F, se mostrará la parte principal del programa desarrollado, que incluye la gestión de conexión WIFI, suscripción al Broker y demás infraestructura necesaria para el correcto desarrollo del trabajo[1].

[1]El código completo desarrollado puede encontrarse en https://github.com/nachoferre/real-time-energy-monitoring

```
require'Class'
ina219 = {
    id = 0,
    address = 0x41,
    calibration_reg = 0x05,
    power_reg = 0x03,
    voltage_reg = 0x02,
    current_reg = 0x04,
    sda = 6,
    scl = 5,
    current_num = 7.142857,
    power_num = 0.357142
    }
local ina219_mt = Class(ina219)

    function ina219:showdata()
        voltage = ina219:read_voltage()
        current = ina219:read_current()
        power = ina219:read_power()

        print("================")
        print( "Voltage: "..voltage)
        print( "Current: "..current)
        print( "Power: "..power)
    end
```

```
function ina219:init(adr)
        ina219.address = adr
    i2c.setup(self.id, self.sda, self.scl, i2c.SLOW)
        i2c.start(self.id)
        i2c.address( self.id, self.address, i2c.TRANSMITTER )
        number = i2c.write(self.id,self.calibration_reg)
        print("Written "..number)
        number = i2c.write(self.id,0x16)
        print("Written "..number)
        number = i2c.write(self.id,0xDB)
        print("Written "..number)
        i2c.stop(self.id)
end

function ina219:read_voltage()
    i2c.start(self.id)
    i2c.address(self.id, self.address, i2c.TRANSMITTER)
    i2c.write(self.id,self.voltage_reg)
    i2c.stop(self.id)

    i2c.start(self.id)
    i2c.address(self.id, self.address, i2c.RECEIVER)
    number=i2c.read(self.id, 2) -- read 16bit val
    i2c.stop(self.id)

    h,l = string.byte(number,1,2)
    result = h*256 + l
    result = bit.rshift(result, 3) * 4
    --print(result)

    return result
end
```

```
    function ina219:read_current()
        i2c.start(self.id)
        i2c.address(self.id, self.address, i2c.TRANSMITTER)
        i2c.write(self.id,self.current_reg)
        i2c.stop(self.id)

        i2c.start(self.id)
        i2c.address(self.id, self.address, i2c.RECEIVER)
        number=i2c.read(self.id, 2) -- read 16bit val
        i2c.stop(self.id)

        h,l = string.byte(number,1,2)
        result = h*256 + l
        --print(result)

        return result / self.current_num
    end

    function ina219:read_power()
        i2c.start(self.id)
        i2c.address(self.id, self.address, i2c.TRANSMITTER)
        i2c.write(self.id,self.power_reg)
        i2c.stop(self.id)

        i2c.start(self.id)
        i2c.address(self.id, self.address, i2c.RECEIVER)
        number=i2c.read(self.id, 2) -- read 16bit val
        i2c.stop(self.id)

        h,l = string.byte(number,1,2)
        result = h*256 + l
        --print(result)

        return result / self.power_num
    end
    function ina219:set_adr(adr)
        ina219.address = adr
    end
return ina219;
```

Apéndice F

Código LUA. Script principal

Este código es el script principal del sistema. Debe ser nombrado como init.lua para que cada vez que la placa se conecte a una fuente de alimentación se ejecute el script. En él, vemos la conexión por WIFI y el envío y recepción de mensajes a través de MQTT.

```
require ("ina219")


-- here we specified the subscriptions the esp has active
function subs()
    m:subscribe("/Node-Red/sleep/0/", 0, function(conn)
        print("Subscription successful")
    end)
    m:subscribe("/Node-Red/current_enable/0/", 0, function(conn)
        print("Subscription successful")
    end)
    m:subscribe("/Node-Red/power_enable/0/", 0, function(conn)
        print("Subscription successful")
    end)
    m:subscribe("/Node-Red/voltage_enable/0/", 0, function(conn)
        print("Subscription successful")
    end)
end
```

```
function connection_mqtt()
    -- j = 1
    t = {}
    for j=1,8
    do
         local ina_chip = ina219:new()
         local current = -1
         local power = -1
         local voltage = -1
         ina_chip:set_adr(ina_adr[j])
         print(tostring(current_enable[j]))
         if current_enable[j] == 1 then
             print(tostring(current))
             current = ina_chip:read_current()
             print(tostring(current))
         end
         if voltage_enable[j] == 1 then
             voltage = ina_chip:read_voltage()
             print(tostring(voltage))
         end
         if power_enable[j] == 1 then
             power = ina_chip:read_power()
             print(tostring(power))
         end
         t["ina-"..tostring(j)] = {}
         t["ina-"..tostring(j)]["current"] = current
         t["ina-"..tostring(j)]["power"] = power
         t["ina-"..tostring(j)]["voltage"] = voltage
    end
    ok, json = pcall(cjson.encode, t)
    if ok then
        m:publish("/ESP8266/0/",json , 0, 0, function(conn)
            print("Whole data sent.")
            end)
    else
        json = pcall(cjson.encode, t)
        m:publish("/ESP8266/0/",json , 0, 0, function(conn)
            print("Whole data sent.")
            end)
     end
    tmr.alarm(3,time_between_sensor_readings*1000, 0,
                    function() connection_mqtt() end)

end
```

```
--wifi--
wifi.setmode(wifi.STATION)
wifi.sta.config("USER","PASSWORD")
wifi.sta.connect()
connected = false
m = mqtt.Client(mqtt_client_id, 1200, mqtt_username, mqtt_password)

tmr.alarm(1, 1000, 1, function()
    if wifi.sta.getip()== nil then
      print("IP unavaiable, Waiting...")
    else
      print("ESP8266 mode is: " .. wifi.getmode())
      print("The module MAC address is: " .. wifi.ap.getmac())
      print("Config done, IP is "..wifi.sta.getip())
      print(wifi.sta.status())

      a = m:connect( mqtt_broker_ip , mqtt_broker_port, 0, function(conn)
          print("Connected to MQTT")
          print("  IP: ".. mqtt_broker_ip)
          print("  Port: ".. mqtt_broker_port)
          print("  Client ID: ".. mqtt_client_id)
          print("  Username: ".. mqtt_username)
          subs()
          connected = true
          tmr.unregister(1)
          end )
    end
end)
--mqtt--
mqtt_broker_ip = "192.168.1.7"
mqtt_broker_port = 8266
mqtt_username = "" --ESP8266
mqtt_password = "" --INA219
mqtt_client_id = "esp1-nacho"
time_between_sensor_readings = 20
--node.dsleep(0)

local ina_chip = nil
ina_adr = {0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47}
current_enable = {[1]=1, [2]=1, [3]=1, [4]=1, [5]=1, [6]=1, [7]=1, [8]=1}
power_enable = {[1]=1, [2]=1, [3]=1, [4]=1, [5]=1, [6]=1, [7]=1, [8]=1}
voltage_enable = {[1]=1, [2]=1, [3]=1, [4]=1, [5]=1, [6]=1, [7]=1, [8]=1}
```

```
m:on("message", function(conn, topic, data)
    print("message recieved. Topic: "..topic)
    if topic == "/Node-Red/sleep/0/" then
        if type(data) == "string" then
            message = tonumber(data)
            time_between_sensor_readings = data
            print("treated")
        end
--new subs to be treated here
    elseif topic == "/Node-Red/power_enable/0/" then
        if type(data) == "string" then
            message = tonumber(data)
            if power_enable[message] ==1 then
                power_enable[message] = 0
            else
                power_enable[message] = 1
            end
            print("treated")
        end
    elseif topic == "/Node-Red/current_enable/0/" then
        if type(data) == "string" then
            message = tonumber(data)
            print(tostring(message))
            if current_enable[message] == 1 then
                current_enable[message] = 0
            else
                current_enable[message] = 1
            end
            print("treated")
        end
    elseif topic == "/Node-Red/voltage_enable/0/" then
        if type(data) == "string" then
            message = tonumber(data)
            if voltage_enable[message] == 1 then
                voltage_enable[message] = 0
            else
                voltage_enable[message] = 1
            end
            print("treated")
        end
    end
end)
```

```
for j=1,8
do
  ina_chip = ina219:new()
  ina_chip:init(ina_adr[j])
  ina_chip = nil
end

tmr.alarm(3,time_between_sensor_readings, 1, function()
    if connected == true then
        tmr.unregister(3)
        print("starting sends")
        connection_mqtt()
    end
end)
```