DEMOSTRADOR IOT-CLOUD EN TIEMPO REAL

EDUARDO ARIAS MONCHE
MÁSTER UNIVERSITARIO DE INGENIERÍA DE TELECOMUNICACIÓN UOC-URL

Junio 2016

Jose Lopez Vicario

Xavi Vilajosana Guillen

FICHA DEL TRABAJO FINAL DE MÁSTER

Título del trabajo:	Demostrador IoT-Cloud en tiempo real
Nombre del autor:	Eduardo Arias Monche
Nombre del consultor:	Jose Lopez Vicario
Nombre del PRA:	Xavi Vilajosana Guillen
Fecha de entrega:	Junio 2016
Titulación:	Máster Universitario de Ingeniería de Telecomunicación UOC-URL
Área del Trabajo Final:	Telemática
Idioma del trabajo:	Castellano
Palabras clave:	IoT, Cloud computing, Fog computing

Resumen del trabajo:

El crecimiento del uso de dispositivos conectados, consolida el concepto de Internet de las cosas (IoT) dentro del paradigma de la computación en la nube. Paralelamente al uso de la nube aparece el paradigma de la computación en la niebla o *fog computing*, el cual proporciona una capa intermedia entre el dispositivo y la nube.

En este proyecto, además de analizar y conectar un dispositivo OpenMote con diversos servicios en la nube con diferentes características, se ha utilizado esta capa intermedia para optimizar el número de mensajes que se envían a cada uno de estos servicios, con el objetivo de reducir los costes de los mismos.

El sistema implementado se compone a nivel de hardware de placas OpenMote, que usando OpenWSN, se interconectan a través del protocolo IEEE 802.15.4e, para enviar la información a un *gateway* compuesto por una Raspberry Pi donde se almacena la información, y que le permite enviar datos a los servicios en la nube. Para saber que información ha de enviar, se implementan estrategias diferentes según la facturación de los servicios y la precisión de la información que se requiera.

En cuanto a los servicios en la nube utilizados se han elegido tres con diferentes características: AWS IoT, que es un *gateway* de entrada de la información al conjunto de servicios de AWS, thethings.iO, que es una plataforma especifica para IoT y PubNub que ofrece un servicio de mensajería que hay que conectar a otros servicios.

Abstract:

The increase of usage of connected devices, consolidates the concept of Internet of Things (IoT) using a cloud computing model. Concurrently to the use of the cloud there is a new concept called fog computing, which adds an intermediate layer between devices and cloud.

In this project, in addition of analyzing and connect OpenMote devices with several cloud computing services which different features, an intermediate layer has been used to optimize the number of messages that are sent to this services, with the goal of reducing the cost of them.

The developed system use as hardware OpenMote boards, using OpenwSN, to connect each other through protocol IEEE 802.15.4e to send the information to a gateway made by a Reaspberry Pi where the information is stored, and who send this data to cloud services. To know which information has to be sent, different strategies are defined and implemented according to the service pricing styles ad the accuracy of the information needed.

As for the cloud services used on this project, three with different characteristics have been chosen: AWS IoT, that is an entry gateway for the data to the set of services offered by AWS, thethings.iO as an specif platform for IoT and PubNub, that offers a messaging service that needs to be connected to external services.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Contexto y justificación del Trabajo

El IoT (*Internet of Things* - Internet de las cosas) se está convirtiendo en una de las tendencias más relevantes de los últimos años en la industria de las nuevas tecnologías. A medida que la tanto la conectividad, el almacenamiento, como la capacidad de computación se vuelven más asequibles se está viendo un crecimiento exponencial de este tipo de soluciones.

Mientras que plataformas como Arduino, OpenMote o Raspberry Pi que permiten conectar sensores para adquirir información, cada vez es mayor la cantidad de datos a procesar y analizar. De este punto de partida nacen las soluciones *cloud* para IoT, que permiten un control y análisis de la información dada por estos sensores, aportando además la flexibilidad y elasticidad de un sistema *cloud*, y funcionalidades como analítica en tiempo real, conexión desde cualquier ubicación, incluido dispositivos móviles, integración con sistemas corporativos, etc.

Dado la cantidad de datos que pueden generar las redes de sensores y que muchas de ellas pueden comunicarse con los sistemas *cloud*, a través de redes móviles o inalámbricas, y los costos que pueden tener los diferentes sistemas *cloud* sobre el envío de datos, es apropiado definir políticas y mecanismos de decisión sobre cómo, qué y cada cuanto ha de enviarse esta

información, de forma que nos permita hacer un uso racional de los recursos, bien sean de coste de la plataforma *cloud*, computacionales, de ancho de banda o conexión de datos.

Para este proyecto se cuenta con unos nodos OpenMote, cuya conexión se realiza usando protocolos IEEE 802.15.4e, y se envía la información de los sensores a una placa Raspberry Pi, esta será la encargada de establecer los mecanismos de decisión necesarios para saber que información y con que frecuencia se debe enviar a las plataformas *cloud*, para así evitar el envío continuo y ahorrar en los costes de estos servicios, cosa que debería impactar también en el consumo de datos de red. Para ello se habrá de tener en cuenta que cada empresa de las que ofrece servicios en la nube aplica distintos tipos de tarificación, con lo cual el sistema se ha de adaptar al servicio escogido.

1.2. Objetivos del Trabajo

El objetivo principal de este Trabajo Final de Máster es el de implementar y analizar un demostrador de IoT conectado a plataformas *cloud* que tenga en cuenta mecanismos de decisión y estrategias para enviar información de una red de sensores a servicios de *cloud* para IoT, buscando un equilibrio adecuado entre la relevancia de la información a transmitir y los costes de enviar esta información a la nube.

Para poder asumir el objetivo principal del proyecto habrá que:

- Estudiar y entender la plataforma de hardware abierta OpenMote, así como conocer sus diferentes componentes y la tecnología de comunicación inalámbrica IEE 802.15.4e.
- Aprender la implementación abierta de OpenWSN y su pila de protocolos y herramientas de software que permitan el conectar con la red de sensores.
- Desarrollar una plataforma de recogida de información basada en OpenMote, OpenWSN y Raspberry Pi que permita recoger la información de los sensores de forma periódica.

- Analizar diferentes plataformas *cloud* para IoT. Sus características, costes, ventajas e inconvenientes.
- Implementar conectores que permitan enviar la información recogida a servicios en la nube.
- Implementar y analizar estrategias para conectar la información recogida por la red de sensores en la Raspberry Pi con los servicios de *cloud computing* para IoT y cuantificar los ahorros obtenidos.

1.3. Enfoque y método seguido

Para la realización del proyecto, determinar que contenido se desarrolla y cuales herramientas existentes utilizar, se ha seguido un doble enfoque, por un lado el de desarrollar aquello que forme parte intrínseca de los objetivos del proyecto, y usar servicios y aplicaciones existentes para lo demás.

Con esta idea se ha desarrollado una aplicación que toma las lecturas de las placas OpenMote y las envía a diferentes servicios, *cloud* o base de datos, utilizando diversos mecanismos de decisión. Se han usado las API (*Application Programming Interface* - interfaz de programación de aplicaciones) de los propios servicios en la nube, siempre que dispusieran de una.

Obviamente, se han usado servicios en la nube existentes, además de TSDB (*Time Series Database* - base de datos de series temporales), herramientas de monitorización, paneles de control que enlacen con esos servicios *cloud*, realizando previamente análisis sobre los componentes a utilizar, como se verá en el capítulo 3.

1.4. Planificación del trabajo

Para la realización de este proyecto se ha elegido el uso de metodologías ágiles, de forma que se se ha podido entregar en los términos establecidos, con las máximas garantías de calidad.

Dado que el sistema de evaluación es continuo, se definieron hitos para cada una de las entregas que se habían de realizar y se definieron que tareas entraban en cada uno de estos hitos (PAC2, PAC3, Memoria final y entrega de la presentación).

Para la primera entrega del proyecto, se planificaron iteraciones semanales, mientras que de cara la segunda se ha varió a periodos de dos semanas. Esto es debido a lo ajustado del calendario de cara a la primera entrega. Al finalizar el primer hito se replantearon algunas tareas ya que con el conocimiento adquirido y la evolución del desarrollo se podían mejorar y afinar más, tanto las estimaciones en puntos como los criterios de aceptación. La planificación de tareas está reflejada en la tabla 1.2.

Para poder analizar el coste de cada una de las tareas se han usado puntos de historia, los cuales según las prácticas ágiles son una medida arbitraria utilizada para medir el esfuerzo y complejidad requerido para implementar una tarea. Aún siendo una medida arbitraría, la idea es que guarden relación entre ellas. Para este proyecto se ha elegido el método de la secuencia de Fibonacci (1, 2, 3, 4, 5, 13, 21, . . .) para definir esta puntuación. Hay que tener en cuenta que la asignación de puntuación al principio del proyecto es muy estimada y se ha de ir refinando a medida que avanza el proyecto para hacer más realista la carga de trabajo antes de comenzar cada una de las iteraciones.

Una tarea se considera terminada cuando se cumpla la siguiente DoD (*Definition of Done* – definición de acabado):

- El código necesario esté desarrollado.
- La funcionalidad está probada sobre el hardware.
- Cada clase o método del código esté documentado.
- Si requiere documentación específica, que esté completa.
- Las guías de instalación están actualizadas.

Los hitos definidos en las tareas son:

Tarea	Criterios de aceptación	Puntos	Iteración	Hitos
Aprendizaje de la plataforma OpenMote y OpenWSN	- Adquirir unos conocimientos suficientes para poder realizar los desarrollos requeridos sobre estas plataformas.	5	1	PAC2
Análisis de plataformas *Cloud*	- Conocer las principales plataformas *Cloud IoT* y sus características. - Análisis de costes de cada una de ellas. - Recomendación de uso para este proyecto.	5	1	PAC2
Adquisición de datos desde plataforma OpenMote	- El hardware ha de tener sensores conectado y ser capaz de tomar la información de los sensores.	5	3	PAC2
Recogida de datos de sensores en Raspberry Pi	- La placa Raspberry Pi deberá ser capaz de recibir la información enviada por las placas OpenMote. - Esa información deberá ser almacenada y clasificada dentro de la Raspberry Pi.	5	3	PAC2
Análisis de mecanismos de decisión de envío de información	- Esta es una tarea de investigación sobre mecanismos de decisión de envío de información y cuales serían más apropiados para los objetivos de este proyecto.	5	2	PAC2
Conexión de Raspberry Pi con servicio de *Cloud IoT*	- La Raspberry Pi deberá ser capaz de enviar información almacenada a los servicios de *Cloud IoT*.	5	4	PAC2
Implementación de los mecanismos de decisión	- La información recogida en la *Raspberry Pi* ha de enviarse a la plataforma *Cloud* siguiendo los mecanismos especificados.	13	4	PAC3
Monitorización en Raspberry Pi	- Montar un sistema de monitorización de recursos y datos insertados en *InfluxDB* en la Raspberry Pi, en principio con *Grafana*. - Diseñar algunos *dashboards* para tener un seguimiento visual de estos datos.	5	4	PAC3
Pruebas y análisis de resultados de los mecanismos de decisión	- Obtener y comparar métricas de los resultados obtenidos con los mecanismos de dicisión.	13	5	PAC3
Presentación de datos en plataforma *Cloud*	- La plataforma *Cloud* ha de tener un *Dashboard* o forma de presentar la información que se ha subido.	8	5	PAC3
Elaboración de la memoria del trabajo	- La memoria debe explicar el trabajo realizado en el proyecto, siguiendo las recomendaciones de la UOC y el tutor.			Entrega memoria final
Elaboración de la presentación	- La presentación debe explicar el trabajo realizado en el proyecto, siguiendo las recomendaciones de la UOC y el tutor.			Entrega presentación
Defensa del proyecto				Tribunal del proyecto

Tabla 1.2: Listado de tareas

PAC2 20 de abril.

PAC3 25 de mayo.

Entrega de la memoria final 12 de junio

Entrega de la presentación 19 de junio

Tribunal del proyecto 20-24 de junio

La fecha para cada una de las iteraciones es:

Iteración	Inicio	Final
1	28 de marzo	3 de abril
2	4 de abril	10 de abril
3	11 de abril	17 de abril
4	25 de abril	8 de mayo
5	9 de mayo	22 de mayo

1.5. Publicación del código fuente

El código utilizado en este proyecto se ha publicado bajo licencia GPL3 en repositorios de GitHub de acceso público.

Se ha creado un repositorio con los *scripts* para crear los paquetes de Debian usados para InfluxDB y Telegraf, accesible en la dirección `https://github.com/eduarias/influx_rpi_deb_builder`.

El segundo repositorio contiene el código fuente desarrollado para conectar OpenMote con los distintos servicios en la nube, cuyos detalles de implementación están explicados en la sección 3.3. La dirección de este repositorio es `https://github.com/eduarias/iot_fog`.

1.6. Métricas de evolución del proyecto

Para la realización del trabajo se ha utilizado GitLab como repositorio de información, tanto para código como para la documentación, lo que permite tener algunas métricas sobre la evolución del proyecto y la continuidad del mismo. En la figura 1.1 está la evolución del proyecto por días, mientras que la figura 1.2 y 1.3 los muestra por horas o días de la semana respectivamente.

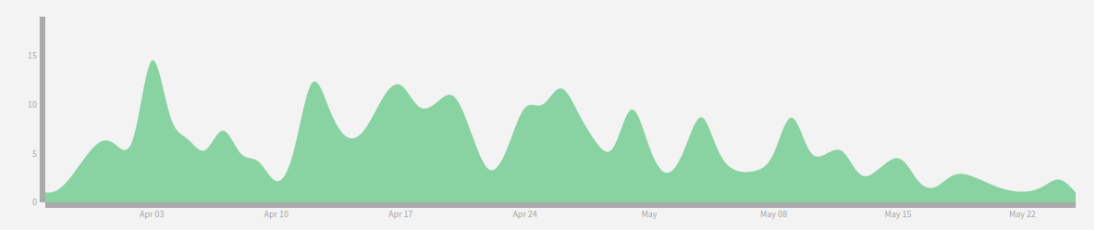


Figura 1.1: *Commit*s enviados por días

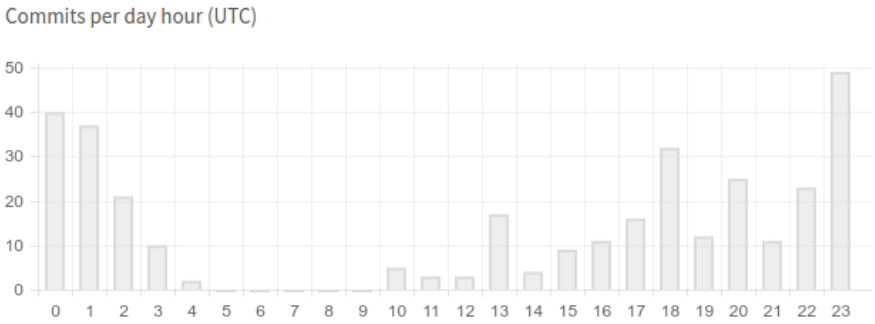


Figura 1.2: *Commit*s enviados por horas

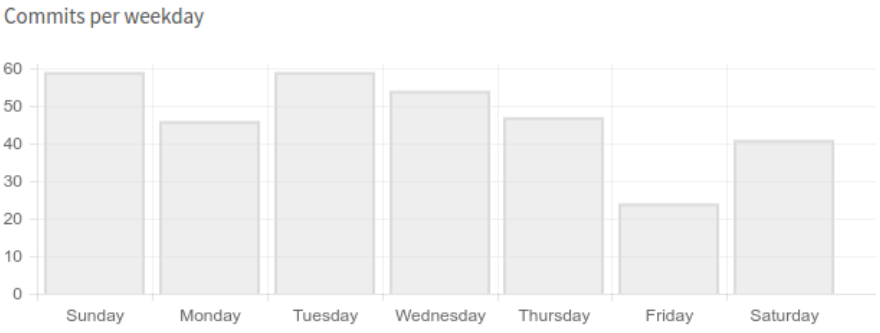


Figura 1.3: *Commit*s enviados por día de la semana

Por otro lado la figura 1.4 muestra los lenguajes de programación usados en este repositorio. *Python* se usa principalmente para aplicaciones en *Raspberry Pi*, *bash* para los scripts de instalación y *Jupyter notebook* para pruebas

de concepto sobre *python*. Aquí no se incluyen los cambios efectuados sobre OpenWSN para intentar conectar con las OpenMote.

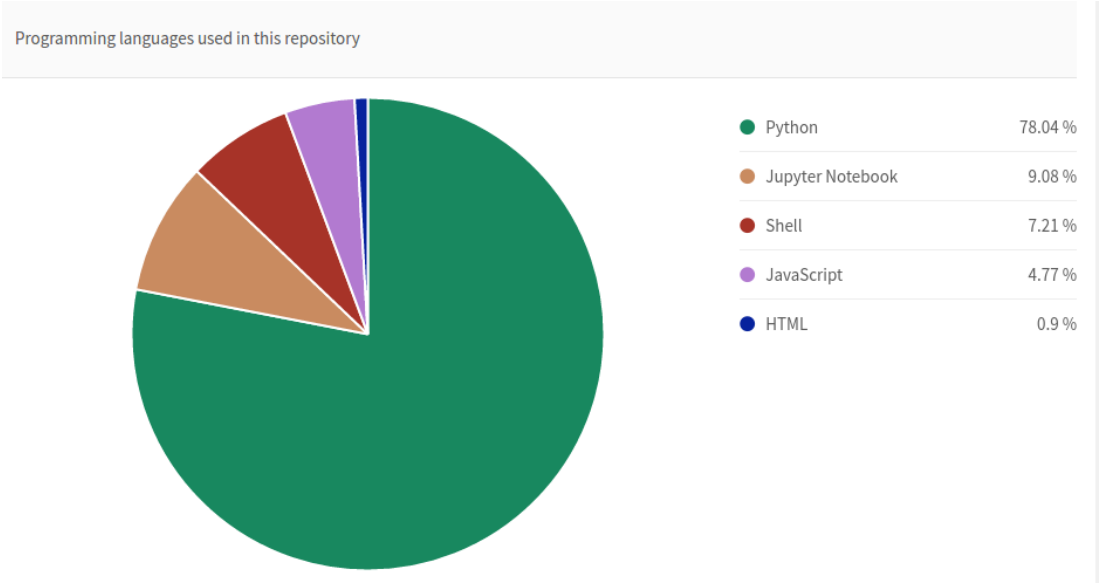


Figura 1.4: Lenguajes de programación utilizados

1.7. Breve sumario de los productos obtenidos

Como producto de este TFM se ha realizado una aplicación para conectar las OpenMote con una base de datos y con servicios en la nube, tal y como se detalla en la sección 3.3. Así mismo durante el desarrollo se han creado *scripts* para facilitar la instalación, tal y como se explica en la sección 3.4.

1.8. Breve descripción de los capítulos de la memoria

En el capítulo 2 se hace una introducción a IoT, explicando los protocolos involucrados, algunos servicios en la nube de referencia y el encaje del concepto de *fog computing*.

En el capítulo 3 se explican los componentes del sistema, las decisiones de diseño e implementación, la instalación de los componentes y los diferentes servicios en la nube utilizados.

En el capítulo 4, se explica como se ha utilizado el sistema, los valores y estrategias óptimas para cada cloud y se presentan los resultados obtenidos.

Capítulo 2

Internet de las cosas (IoT)

El concepto de IoT es el de la interconexión de objetos cotidianos con Internet. El primero en usar este término fue Kevin Ashton en 1999 [1], que se dio cuenta que en aquel tiempo la mayor parte de los datos en Internet eran originados o introducidos en el sistema por personas.

Según la visión de Ashton, desde el punto de vista de un sistema, una persona no es más que una forma entrada lenta, propensa a errores e ineficiente para introducir los datos y que limita la cantidad y calidad de información disponible. De esta forma, sería más eficiente que los sistemas se pudiesen conectar directamente a los sensores que miden los eventos y propiedades del *mundo real* de forma directa, sin personas de por medio. De ahí que en un primer momento se hablara de comunicaciones M2M (*machine to machine* - máquina a máquina) o D2D (*device to device* - dispositivo a dispositivo).

Actualmente, el término Internet de las cosas se usa con una denotación de conexión avanzada de dispositivos, sistemas y servicios que va más allá del tradicional M2M y cubre una amplia variedad de protocolos, dominios y aplicaciones. Es un término que no está relacionado a tecnologías concretas, sino que usa un gran número de tecnologías existente y cuyos casos de uso tampoco se puede decir que tengan una definición clara o sencilla.

Estas *cosas*, o dispositivos, se conectan a la red para enviar información que obtienen a través de sensores o para permitir que otros sistemas interactúen con el mundo a través de actuadores. Estos pueden ser tanto dispo-

sitivos personales, construidos para este propósito o incluso venir incluido en otros objetos como una manera de monitorizar y actuar con ellos.

Los casos de uso y las oportunidades que IoT proporciona a las diferentes industrias son numerosas, lo que hace que se vayan definiendo un conjunto de retos y patrones a resolver. Algunos de estos retos tienen que ver con la interconexión de diverso hardware, sistemas operativos, software o requerimientos de los *gateways* de la red. [2]

2.1. Arquitectura básica

En un sistema de IoT conectado a la nube podemos diferenciar tres elementos básicos: el dispositivo, el *gateway* y la nube.

El dispositivo incluye el software y hardware conectado con el mundo. Para ello deben conectarse en red para comunicarse entre ellos, hacia dispositivos centralizados o directamente a servicios en la nube. Por lo que pueden estar conectados directa o indirectamente a Internet.

El *gateway* permite que los dispositivos que no están directamente conectados a Internet puedan alcanzar los servicios en la nube, de esta manera, se trata de un tipo de dispositivo que procesa los datos en nombre de un grupo de dispositivos o *cluster* y enviarlos a la nube para ser tratados.

El servicio en la nube o *cloud* permite procesar y combinar los datos de los diferentes dispositivos, además de analizarlos, mostrarlos, tomar decisiones o actuar sobre ellos.

Cada dispositivo está compuesto de hardware y software y provee cuatro tipos fundamentales de datos [4]:

Metadatos Datos sobre los datos.

Estado La condición del dispositivo.

Acciones Comandos que indican una acción a realizar por el dispositivo.

Telemetría Datos sobre lo que rodea al dispositivo.

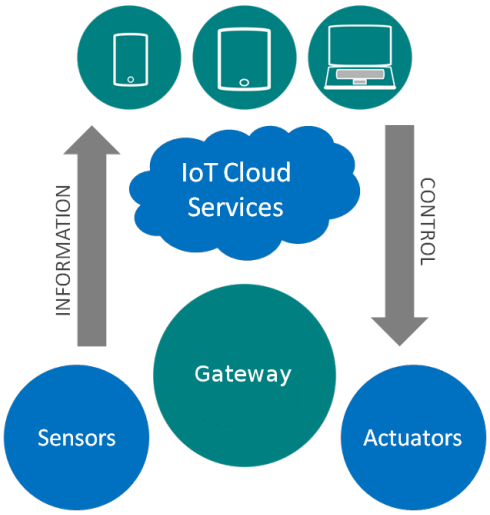


Figura 2.1: Arquitectura básica de un sistema IoT [3]

2.2. Protocolos

Para IoT se usan tanto protocolos de amplio espectro como HTTP y más recientemente WebSockets, como protocolos diseñados para este tipo de soluciones, como puede ser MQTT o CoAP.

2.2.1. HTTP (Hypertext Transfer Protocol)

Es el protocolo de comunicación con el que se realizan las transferencias de información Web. Está orientado a transacciones y sigue el esquema de un cliente, que hace una petición, y un servidor, que le envía un mensaje de respuesta.

Los mensajes que usan HTTP (*Hypertext Transfer Protocol* - protocolo de transferencia de hipertexto) son en texto plano, que por un lado lo hace más legible y fácil de depurar, pero por contra supone mensajes más largos. Están compuestos por una línea inicial, donde se define el método de petición o código de respuesta, una cabecera y el cuerpo del mensaje. [5]

Actualmente la versión más utilizada es HTTP/1.1, aunque en mediados de 2015 se aprobó HTTP/2. Esta nueva versión no modifica la semántica de

HTTP, sin embargo incluye mejoras como el uso de una única conexión, compresión de cabeceras o servicios *server push*. [6]

Para aportar seguridad a este protocolo existe HTTPS (*Hypertext Transfer Protocol Secure* - protocolo seguro de transferencia de hipertexto), que permite cifrar los mensajes HTTP sobre SSL/TLS, utilizando un sistema de clave pública/clave privada.

REST (Representational State Transfer)

Se trata de un tipo de arquitectura de desarrollo web apoyada en el estándar HTTP, que nos permite describir un interfaz entre sistemas que utilice HTTP para obtener información o realizar operaciones con los datos, por lo que supone una arquitectura muy natural para crear servicios orientados a Internet.

Uno de los principios básicos de esta arquitectura es que nunca se debe guardar el estado en el servidor, sino que toda la información que se pide ha de estar en la petición del cliente. Esto facilita el tema de la escalabilidad, ya que no requiere de mantener variables de sesión. [7]

2.2.2. WebSockets

WebSocket es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket en TCP (*Transmission Control Protocol* - protocolo de control de transmisión), de manera que existe una conexión persistente entre el cliente y el servidor, con lo que ambas partes pueden empezar a enviar datos en cualquier momento. Forma parte de la especificación HTML5.

Debido a que las conexiones TCP comunes sobre puertos diferentes al 80 son habitualmente bloqueadas por los administradores de redes, el uso de esta tecnología proporciona una solución a este tipo de limitaciones con una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP, a costa de una pequeña sobrecarga del protocolo. [8]

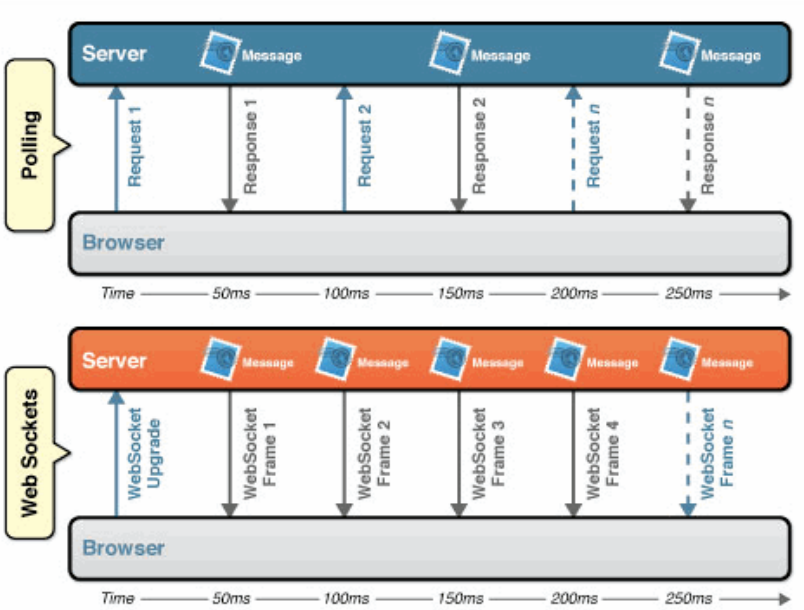


Figura 2.2: Peticiones Polling y WebSockets [9]

En el ámbito de IoT nos permite tener cada sensor conectado a un servidor determinado en el cual debemos de utilizar el WebSocket con el fin de obtener esta información en tiempo real. Como la conexión es permanente no se necesita enviar cada vez una petición al servidor, como suele ser necesario en otros protocolos, lo cual ofrece una reducción del tráfico innecesario y de la latencia. [10]

2.2.3. MQTT

Es un protocolo de mensajes *publish-suscribe*, muy simple y ligero, diseñado para sistemas con recursos muy limitado, bajo ancho de banda, alta latencia o redes poco fiables. Sus principios de diseño son minimizar el ancho de banda y los requerimientos de recursos mientras garantiza fiabilidad y calidad en la entrega del mensaje. Estas características lo convierten en un protocolo ideal para dispositivos móviles o bien dispositivos conectados M2M o IoT.[11]

La arquitectura de MQTT (*MQ Telemetry Transport*) sigue una topología de estrella, donde existe un *broker* o nodo principal con hasta 10.000 clientes conectados. El *broker* se encarga de enrutar, filtrar y distribuir los mensajes a los

clientes apropiados, y los clientes publican y se suscriben a estos mensajes. Esta comunicación puede ser cifrada.

La comunicación se basa en unos temas (*topics*), de manera que el cliente publica el mensaje y los nodos que desean recibirlos han de suscribirse a él, de manera que esta comunicación puede ser uno a uno o uno a *n*. Estos temas se representan mediante una cadena separada por '/'que define la estructura jerárquica, de esta manera se pueden definir jerarquías de clientes que publican y reciben datos, tal y como vemos en la figura 2.3.

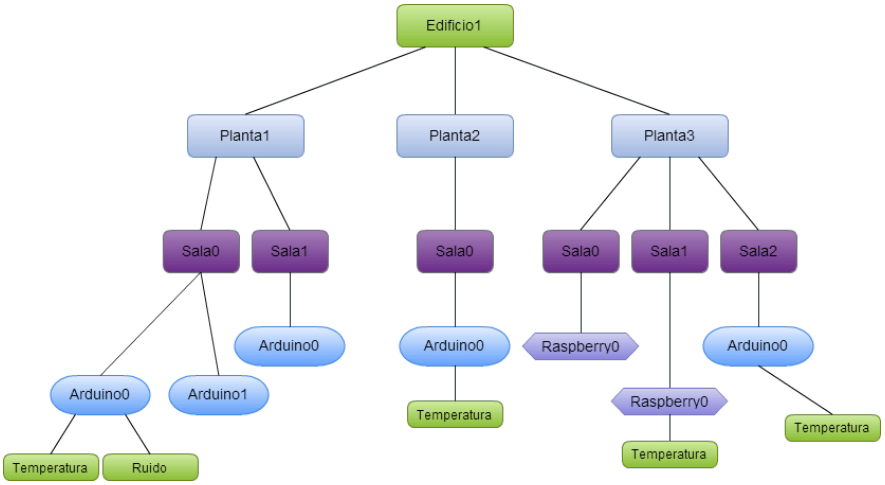


Figura 2.3: Jerarquía MQTT [12]

Uno de los atractivos principales de MQTT son los niveles de QoS (*Quality of Service* - calidad de servicio). El protocolo define tres niveles:

QoS 0. Solo envía una vez Este método es el nivel por defecto y no garantiza la recepción del mensaje, este se envía una sola vez y el receptor no envía ninguna respuesta, por lo que en caso de no recibirse no hay ningún reintento.

QoS 1. Entrega al menos a uno Este nivel garantiza que el mensaje es entregado, ya que el receptor envía un mensaje de confirmación. Si el mensaje de confirmación no se recibe, se vuelve a enviar el mensaje original marcado como duplicado hasta recibir confirmación.

QoS 2. Entrega exactamente una vez En este nivel la entrega se realiza en dos pasos, en el primero se envía el mensaje y una vez recibida la confirmación se envía un segundo mensaje diciendole al receptor que puede

publicarlo, del cual también habrá de dar confirmación. Este nivel es el más seguro, pero también el más lento, ya que requiere de al menos dos pares de comunicaciones.[11]

2.2.4. CoAP

Este protocolo está diseñado para ser utilizado en dispositivos electrónicos simples, con un mínimo de 10KiB de RAM y 100 KiB de código, permitiendo que estos se comuniquen con la web. Fue aprobado como RFC 7252 por IETF en 2014, tras tres años como borrador.

Está basado en el modelo HTTP REST (*Representational State Transfer* - Transferencia de Estado Representacional), de forma que a los servidores tienen una URL, a la que se accede usando métodos como GET, PUT, POST y DELETE. Usa cabeceras reducidas, y limita el intercambio de mensajes, añadiendo soporte UDP y otras modificaciones como mecanismos de seguridad específicos.

Una de sus características destacadas de CoAP (*Constrained Application Protocol*) son Los servicios de descubrimiento integrados, que permiten encontrar las propiedades de otros nodos de la red. En cuanto a seguridad, utiliza parámetros DTLS, equivalente a usar claves RSA de 3072 bits. [13]

2.2.5. Conclusiones

Por la naturaleza de los dispositivos IoT, donde suele ser importante mantener un bajo consumo de recursos, lo normal es intentar dentro de lo posible utilizar protocolos optimizados como son CoAP y MQTT, tal y como se refleja en la tabla 2.1. Por desgracia no siempre podremos elegir una alternativa, ya que esto depende del soporte que de cada plataforma a cada uno de ellos.

	CoAP	**HTTP Rest**	**MQTT**	**WebSocket**
Protocolo de transporte	UDP	TCP	TCP	TCP
Seguridad	DTLS	SSL/TLS	SSL/TLS	WSS(TLS)
Con estado	No	No	Si	Si
Tipo de mensajes	Request/Response	Request/Response	Publish/Suscribe Request/Response	Publish/Suscribe Request/Response
Características principales	- Bajo consumo de recursos	- Ha de establecer la comunicación cada vez. - Alto consumo de ancho de banda. - Alta estandarización para servicios web.	- Bajo consumo de recursos. - Minimiza ancho de banda. - QoS	- Mantiene una conexión permanente, lo que implica menos tráfico y baja latencia.

Tabla 2.1: Resumen de protocolos

2.3. Servicios IoT en la nube

2.3.1. Amazon AWS IoT

En el mes de octubre de 2015 Amazon anunció su nuevo servicio cloud de IoT, que permite a los dispositivos conectar e interactuar con facilidad y seguridad con el resto de servicios de AWS (*Amazon Web Services*), además de procesar y actuar sobre datos de dispositivos y habilitar las aplicaciones para que interactúen con dispositivos aunque no estén conectados. AWS IoT admite miles de millones de dispositivos y billones de mensajes, y es capaz de procesar y enrutar dichos mensajes a puntos de enlace de AWS y a otros dispositivos de manera fiable y segura.

Para poder conectar los dispositivos, AWS IoT ofrece un SDK (*Software Development Kit* - kit de desarrollo de software) que permite conectar, autenticar e intercambiar mensajes mediante MQTT, HTTP REST o WebSockets. Actualmente se soportan oficialmente las SDK para C, JavaScript y Arduino Yún [14]. En la figura 2.4 se muestra el esquema básico de AWS IoT.

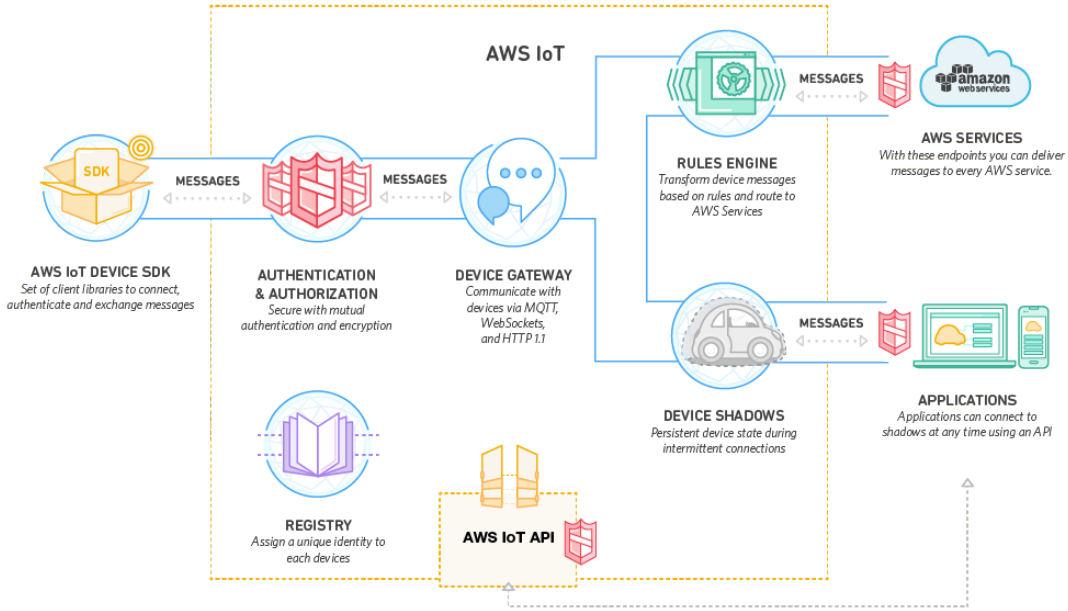


Figura 2.4: Esquema y conexión de componentes en AWS IoT [15]

Este servicio está formado por los siguientes componentes:

***Broker* de mensajes** Proporciona un mecanismo seguro para que los dispositivos y las aplicaciones publiquen y reciban mensajes entre ellas. Puede utilizar tanto MQTT para publicar y suscribirse como HTTP REST para publicar.

Motor de reglas Permite procesar mensajes e integrarlos con otros servicios de AWS como AWS Lambda, Amazon Kinesis, Amazon S3, Amazon Machine Learning o Amazon DynamoDB. Este motor evalúa los mensajes entrantes publicados en AWS IoT y los transforma y entrega a otros dispositivos o servicios basándose en una serie de reglas definidas. Además se puede utilizar para republicar mensajes a otros suscriptores.

Registro de dispositivos Organiza los recursos asociados con cada dispositivo a través de metadatos como atributos y capacidades de los mismos. El registro permite almacenar metadatos acerca de los dispositivos sin un costo extra. Se pueden asociar también certificados e identificadores de cliente MQTT de cara ha gestionar y resolver problemas. Además, los metadatos del registro no caducan, siempre que obtenga acceso o actualice su entrada en el registro al menos una vez cada 7 años.

Sombras del dispositivo Este componente crea una versión persistente, virtual o *sombra* de cada uno de los dispositivos, que proporciona el último estado del mismo, pudiendo las aplicaciones u otros dispositivos interactuar con él. Las sombras del dispositivo hacen que persista el último estado registrado y el estado que se desea en el futuro de cada uno de los dispositivos, incluso aunque estén desconectados. Este estado persiste ilimitadamente si se actualiza al menos una vez al año, si no caducan.

***Gateway* para dispositivos** Permite a los dispositivos conectarse de forma segura y eficiente con AWS IoT, utilizando un modelo de publicación o suscripción, lo que permite las comunicaciones individuales o múltiples. Con el patrón de comunicación múltiple, AWS IoT hace posible que un dispositivo conectado difunda datos a varios suscriptores sobre un tema concreto. El *gateway* para dispositivos admite los protocolos MQTT y HTTP 1.1; y de igual forma permite implementar la compatibilidad con protocolos de propietario o heredados. Además se amplía automática-

mente para admitir más de mil millones de dispositivos sin infraestructura de aprovisionamiento.

Servicio de seguridad e identidad AWS IoT provee de autenticación mutua y cifrado para que no se intercambien datos sin una identidad probada. Esto se hace mediante un mecanismo de responsabilidad compartida, de manera que los dispositivos han de mantener sus credenciales seguras de cara a poder enviar de forma segura al *broker* de mensajes, que a su vez enviará la información también de forma segura a los dispositivos, servicios o aplicaciones. Las conexiones que utilizan HTTP pueden hacer uso de SigV4 o certificados X.509, las que utilizan MQTT usan una autenticación basada en certificado y, por último, las conexiones con WebSockets pueden utilizar SigV4.

El coste de este servicio es de 5$ por millón de mensajes (publicados o enviados) y dispone de una capa gratuita que permite comenzar con 250.000 mensajes gratuitos al mes durante 12 meses [16]. Esto solo incluye el flujo de mensajes, en caso de querer almacenar la información se deberá recurrir a otros servicios de AWS como DynamoDB, cuyo coste se divide en capacidad de almacenamiento, rendimiento previsto y transmisiones, costando 0,283$ por GB a partir de 25 GB, 0,00735$ por hora por cada 10 unidades de capacidad de escritura (36.000 escrituras/hora) y la misma cantidad por 50 unidadaes de capacidad de lectura [17].

2.3.2. Google Cloud Platform

La apuesta de la plataforma *cloud* de Google para IoT es la de usar su servicio *Cloud Pub/Sub* como plataforma de mensajes para publicar y suscribirse los datos de múltiples dispositivos. Entonces se pueden utilizar el *Google Cloud Dataflow* para procesar esta información en tiempo real, tal y como muestra la figura 2.5.

El servicio *Cloud Pub/Sub* no es exclusivo para IoT, sino que se usa para otros servicios de Google como Ads o Gmail. Entre sus funcionalidades más destacadas están: [19]

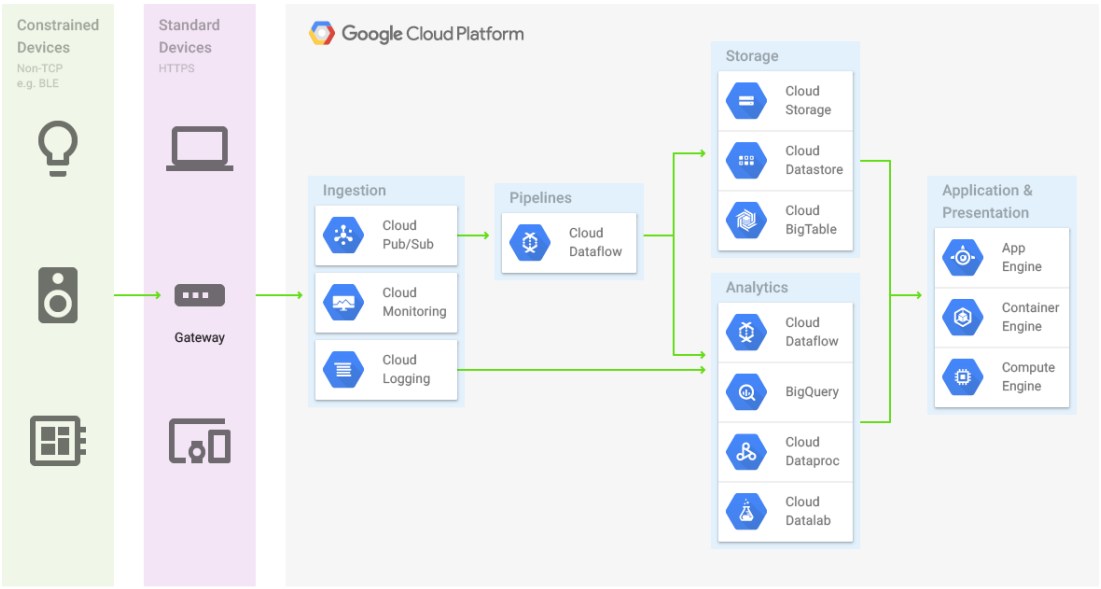


Figura 2.5: Diagrama de servicios *cloud* para IoT de Google [18]

Escalabilidad Permite hasta 10.000 mensajes/segundo por defecto, y varios millones bajo petición.

Entregas push y pull Los suscriptores tiene opciones flexibles de recepción.

Cifrado Todos los mensajes de datos van cifrados y se provee una capa de seguridad.

Almacenamiento replicado Almacena los datos en varias localizaciones.

Cola de mensajes Proporciona colas de mensajes a las que suscribirse.

Acuse de recibo punto a punto Provee de mensajes de acuse de recibo (*acknowledge*) a nivel de aplicación.

Fan-out Soporta patrones *once-to-many* o *many-to-many* a través de suscripciones a temas.

REST API Dispone de una librería REST que permite interactuar con diversos lenguajes de programación.

El coste de este servicio varía entre 0.40$ a 0.05$ por millón de operaciones. En este coste no se incluyen los servicios de almacenamiento de información o análisis. [20]

Una vez se ha establecido las comunicaciones entre los dispositivos y la plataforma, se pueden usar servicios como *BigQuery* para almacenar y procesar esta información, en el que caso que necesitáramos servicios de *Analytics*, o *Firebase* para crear aplicaciones para hacer uso de estos datos y proveer de actuadores.

Los servicios de *Google Cloud Platform* disponen de una versión de prueba que permiten obtener 300$ para utilizarlos en su plataforma durante 60 días en cualquiera de los servicios.

2.3.3. Oracle IoT Cloud Service

La apuesta de Oracle por los servicios *cloud* para IoT es similar a los anteriormente analizados. En este caso la solución de Oracle para conectar los dispositivos a la nube se hace a través de la librería cliente que proporciona el servicio. Esta librería proporciona un sistema de mensajes bidireccional mediante el *cloud* y el dispositivo, como se aprecia en la figura 2.6.

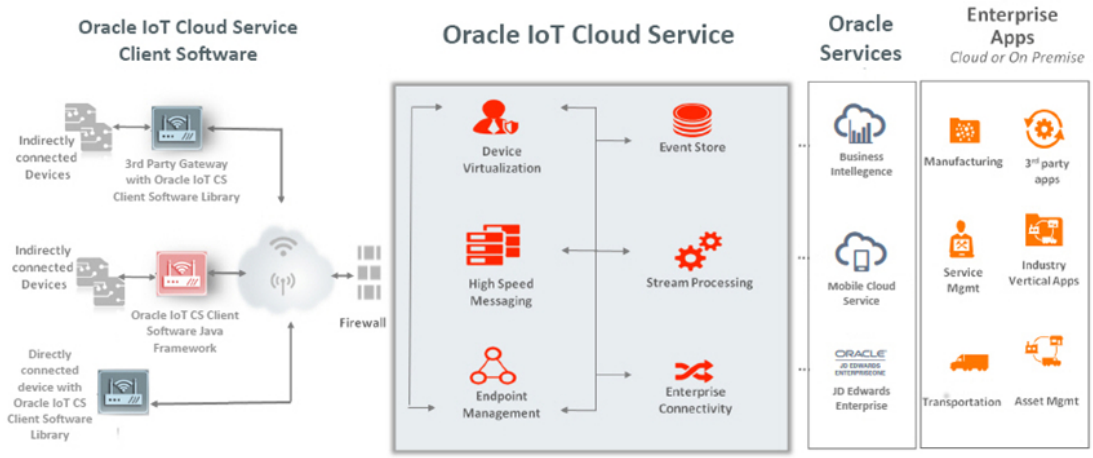


Figura 2.6: Diagrama de servicios *cloud* para IoT de Oracle [21]

Hay tres maneras de conectar los dispositivos con la nube. La primera es que esten conectados directamente a través de *IoT Cloud Service Device*

Client Software Library. Las otras dos se basan en la utilización de un *gateway* para conectar los dispositivos, una solución es usar directamente *Oracle IoT Cloud Service Gateway*, que proporciona un sistema operativo con todo el software para conectar con la nube de Oracle de forma rápida. La última propuesta es que el *gateway* utilice el *IoT Cloud Service Device Client Software Library* para conectar con el servicio.

Algunas de las funcionalidades que ofrece Oracle IoT Cloud Service son [22]:

Virtualización de dispositivos Cada dispositivo se expone a como un conjunto de recursos con propiedades y mensajes que se pueden definir, de forma que se vayan haciendo llamadas REST para acceder a la información que el dispositivo ha enviado a la nube como si se estuviese accediendo directamente al dispositivo.

Gestión de extremos(*endpoints*) A través de la API y la consola de gestión se pueden gestionar todos los dispositivos conectados, lo que incluye desplegar software o gestionar las políticas de seguridad.

Almacenamiento de eventos Dispone de una base de datos persistente donde se almacenan todos los mensajes que se intercambian con os dispositivos.

Proceso de flujos de datos Se pueden realizar análisis en tiempo real de flujos de datos recibidos, soportando operaciones como correlación, agregación, filtrado y analíticas sobre datos basados en tiempo.

Conectividad empresarial Dispone de un canal seguro para conectarse con aplicaciones empresariales.

Producto	Precio mensual de dispositivo	Mínimo de dispositivos	Mensajes incluidos por mes
Wearable	0.40$	100.000	1.500
Dispositivos de consumo	0.80$	50.000	15.000
Dispositivos telemáticos	2.00$	20.000	100.000
Dispositivos comerciales / industriales	3.00$	10.000	100.000

Tabla 2.2: Precios Oracle IoT Cloud Service

El precio de este servicio va por dispositivos conectados, habiendo un mínimo para cada uno de ellos, los detalles están en la tabla 2.2.

2.3.4. Xively

Es una división de LogMeIn Inc. que desde 2013 ofrece una plataforma *cloud* para IoT que incluye entre otros servicios de directorio, servicios de datos, un motor de seguridad y aplicaciones de gestión web.

Figura 2.7: Diagrama de servicios *cloud* para IoT de Xively [23]

Su sistema de mensajes está basado en el protocolo MQTT y su API soporta HTTP REST, WebSockets y MQTT [24]. Su diagrama de servicios se muestra en la figura 2.7.

Entre los servicios disponibles, aparte de la transmisión de datos con los dispositivos, destacan:

Identity Management Gestiona identidades y autenticación de forma segura.

Blueprint & Messaging Controla como los elementos y entidades se representan.

Time Series Guarda y analiza las series de datos históricos.

En cuanto a el precio, Xively no lo hace público, sino que se ha de contactar con su servicio de ventas de cara a fijar presupuesto. Algunas fuentes hablan de un coste aproximado de 0.10$ por dispositivo [25], aunque depende de los servicios contratados. También dispone de cuentas para desarrolladores de forma gratuita, que permite conectar un número ilimitado de dispositivos, con un límite de 25 llamadas a la API por minuto y los datos están disponibles únicamente un mes.

2.3.5. thethings.iO

Esta empresa con sede en Barcelona se fundó con el lema “Tú creas cosas chulas. Nosotros te las conectamos a Internet” y pretende convertirse en un referente mundial en el *cloud* para empresas IoT.

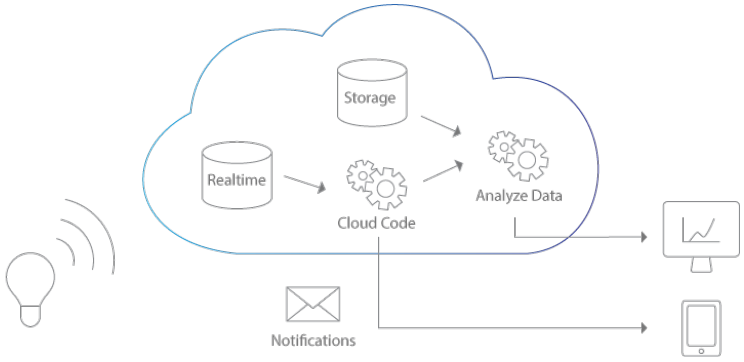


Figura 2.8: Esquema thethings.io [26]

Ofrece una plataforma en tres capas, la de conexión, la de monitorización y gestión y la de análisis de datos. La capa de conexión ofrece sencillas APIs en JavaScript, Python y Arduino, además de soportar HTTP REST, MQTT, WebSockets y CoAP. La capa de monitorización y gestión permite, entre otras cosas, visualizar los datos en tiempo real, obtener datos históricos, gestionar los dispositivos o crear aplicaciones para visualizar los datos. En capa de análisis de datos se pueden generar *dashboards* de forma sencilla haciendo *drag & drop* de *widgets*, tomar métricas de los dispositivos, analizar la información y ejecutar funciones a partir de los datos de entrada.

El costo de esta solución es de 1 € por dispositivo al año, hasta 1.000 dispositivos, y ofrece mensajes, funciones y almacenamiento ilimitados, así como soporte básico de análisis de datos. También dispone de una cuentas gratuitas para usar un único dispositivo, con almacenamiento de un mes y hasta 10.000 mensajes.[27]

2.3.6. Temboo

Esta compañía con sede en Nueva York, fundada en 2013, ofrece una interesante perspectiva sobre el tema de *cloud* para IoT. Su producto se basa en

ofrecer además de 10 SDKs para diversas plataformas y lenguajes de programación, lo que denomina Coreografías (*Choreos*), que son pequeños trozos de código que permiten desarrollar tareas, como conectar a API públicas via HTTP, enviar correos o actualizar una base de datos.

Posee un extenso repositorio de APIs, denominado *Choreos Library*, que permiten interactuar con más de 200 servicios, entre ellos otras plataformas en la nube, como AWS, Google o Dropbox, de forma muy sencilla. Lo que la convierte en una plataforma muy interesante si se ha de interactuar con diversas plataformas.

Los protocolos soportados son MQTT, CoAP y HTTP. Esta plataforma viene instalada de serie en algunos productos de Texas Instruments y Samsung, además de Arduino Yún.

En cuanto a costes, disponen de tres planes de precios. El *Enterprise* cuesta 49$ al mes y permite 16 GB de transferencia de datos, 100.000 ejecuciones de *Choreo* y 20 perfiles permanentes, mientras que el plan individual cuesta 7$ al mes para 1 GB de transferencia de datos, 25.000 ejecuciones de *Choreo* y 5 perfiles permanentes. A estos precios se añaden costes extras por usos adicionales. También disponen de un plan gratuito para explorar, con 250 ejecuciones de *Choreo*, 1 GB de transferencia de datos y tres perfiles mensuales. [28]

2.3.7. PubNub

Esta empresa ofrece servicios IaaS (*Infrastructure as a Service* - infraestructura como servicio) orientados a redes de transmisión de datos. Su producto principal es *Publish/Suscribe Messaging*, que proporciona una plataforma para publicar y suscribirse a mensajes. A diferencia de los servicios anteriores, PubNub no almacena la información, así que su funcionamiento es el de un *gateway* en la nube para distribuir los mensajes de los dispositivos.[29]

Como protocolo utiliza en sus SDK WebSocket sobre TCP, aunque también soporta MQTT, COMET, BOSH y están trabajando en soporte para SPDY, HTTP 2.0. Su objetivo es que los protocolos a utilizar sean transparentes para

el desarrollador, siendo su propio SDK el encargado de utilizar el más apropiado. [30]

El precio de este servicio viene marcado por dispositivos activos y número de mensajes mensuales. Dispone de diferentes planes de precios que van desde el gratuito, que ofrece 100 dispositivos y un millón de mensajes, hasta el *Scale*, que por 799$ ofrece hasta 20.000 dispositivos y 40 millones de mensajes al mes.

En este caso hay que tener en cuenta que no se cuenta como mensaje solo el enviado, si no que para cada dispositivo que esté suscrito al canal y que reciba un mensaje este se cuenta también. El caso más llamativo es el propio panel de control del servicio, que al tenerlo abierto recibe los mensajes enviados para poder contarlos, por lo que cada mensaje enviado a PubNub y recibido por su propio panel de control cuenta como dos.

2.3.8. Conclusiones

Se pueden observar varios patrones en los servicios en la nube analizados. Uno de ellos es el de servicios generales existentes que aportan un módulo de IoT específico que actúa como *broker* de entrada de mensajes en su *cloud*, a este grupo pertenece AWS, Google Cloud o Oracle IoT. Por otro lado los otros servicios analizados son específicos para el mundo IoT. Esto se refleja en la tabla 2.3, que resume las características de los servicios en la nube.

Si se analizan por su política de precios existen dos mecanismos, cobrar por mensaje enviado (o por millón de mensajes) y cobrar por dispositivo con un límite de mensajes por unidad de tiempo (generalmente día o mes), aunque hay servicios como Temboo que tienen su propio mecanismo.

Para el desarrollo de este proyecto se ha elegido AWS IoT que ofrece una integración con el servicio de *cloud computing* más importante que existe en estos momentos [31], esta solución es similar a la que ofrece Google Cloud o Oracle, pero a diferencia de esta última factura por mensajes. Por otro lado se ha escogido thethings.iO como un ejemplo de una plataforma específica para IoT, que no requiere de interconectar servicios para explotar los datos

y que además usa una facturación por dispositivo. Y por último PubNub, que gestiona únicamente la mensajería en tiempo real. Con estos tres servicios se pueden analizar mecanismos de decisión de envío de datos a la nube para distintos tipos de facturación.

Servicio	**Protocolos**	**Características**	**Política de precios**	**Servicios de facturación adicional**
AWS IoT	MQTT, HTTP, WebSocket	Reglas que permiten interactuar con el resto de servicios disponibles en AWS	Por mensajes	Almacenamiento, procesamiento, *Big Data*, etc...
Google Cloud	MQTT, HTTP	Reglas que permiten interactuar con el resto de servicios disponibles en Google Cloud	Por mensajes	Almacenamiento, procesamiento, *Big Data*, etc...
Oracle IoT	HTTP	Reglas que permiten interactuar con el resto de servicios disponibles en Oracle Cloud Service	Por dispositivo	Procesamiento, conectividad empresarial, *Big Data*, etc...
Xively	MQTT, HTTP, WebSocket	Diseñado para integrarse con servicios corporativos	Por dispositivo	-
thethings.io	MQTT, HTTP, WebSocket, CoAP	Incluye almacenamiento, panel de control basado en métricas, funciones, ...	Por dispositivo	-
Temboo	MQTT, HTTP, CoAP	Interconexión con diversos servicios	Por transferencia de datos + uso de reglas	Servicios de terceros con los que se opera
PubNub	WebSocket, MQTT, COMET, BOSH	Sistema de gestión de mensajes	Por mensajes mensuales	-

Tabla 2.3: Comparativa servicios cloud IoT

2.4. Fog Computing

En los últimos años se ha escrito mucho sobre el modelo de computación en la nube o *cloud computing* y dado que IoT suele requerir de dispositivos con ciertas limitaciones, como el consumo de energía, hace que estos suelan tener como único propósito recoger información o realizar acciones, lo que los hace ideal para relegar el procesado, análisis y almacenamiento de la información a servicios disponibles en la nube.

A medida que mejoran las capacidades de los dispositivos que se usan como *gateway* aparece un nuevo modelo de computación en la niebla o *fog computing* que extiende el paradigma del *cloud computing*, el cual es un modelo en el cual el procesado de datos y las aplicaciones se concentran en dispositivos en el borde de la red. El término *fog* o niebla viene a referirse a que es una nube pegada al suelo. De esta forma los datos son procesados y pueden realizar acciones localmente,aunque luego esa información se suba a la *nube*.

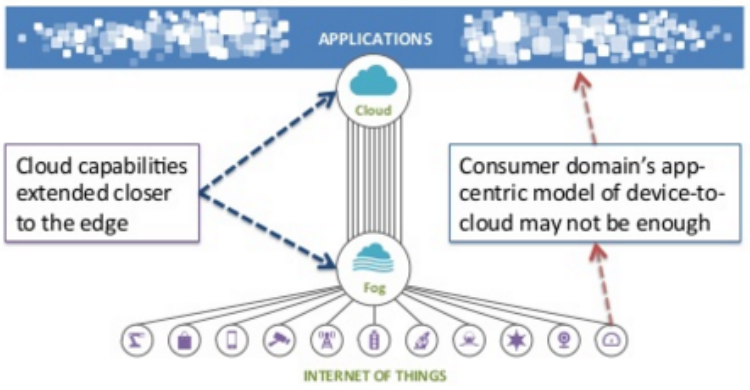


Figura 2.9: Esquema de *fog computing* [32]

Hay ciertas ventajas en usar este modelo. Una de ellas es que dado que los dispositivos IoT pueden generar gran cantidad de datos, el propio *gateway* de acceso puede realizar parte del procesamiento, descargando así a la nube de parte del trabajo para tener una experiencia de uso más fluida, cercana e inmediata.

Hay que tener en cuenta también que el uso de los servicios en la nube suele tener un coste variable asociado, que suele depender del número de

dispositivos o mensajes, además de los costes que puede tener a nivel de red, especialmente en comunicaciones móviles. Colocando una capa intermedia entre los dispositivos y la nube se pueden aplicar políticas para el envío de información, pudiendo disponer de la información completa dentro de la niebla, en caso que se necesitara.

Además hay casos en que se puede requerir una baja latencia para que la respuesta a ciertas acciones detectadas por los sensores sea muy rápida, por ejemplo en el caso de un vehículo conectado hay acciones que no pueden esperar a que un servicio en la nube reciba la información, la procese y devuelva la acción necesaria [33], en estos casos hace falta una capa próxima a los dispositivos que tome este tipo de decisión.

El uso combinado de computación el niebla y en la nube proporciona las ventajas de ambos mundos, por ejemplo, actuaciones rápidas en la niebla, diseñadas para interacciones M2M, mientras que la nube proporciona centralización de la información, que puede venir de varias nieblas, y análisis a nivel de *Big Data* o visualización y reportes en interacciones HMI (*Human to Machine Interactions* - interacción humano máquina).[33]

Un tema importante de este modelo es ser capaces de decidir la interacción entre la niebla y la nube. Para ello se han de definir unos mecanismos de decisión sobre la información recogida a nivel de niebla.

2.4.1. Mecanismos de decisión de interacción entre *Cloud* y *Fog*

En un sistema basado en IoT, hay muchas variables a optimizar, y las soluciones a aportar dependen enormemente de los objetivos que se quieran conseguir. Dado que suelen ser sistemas con capacidades limitadas, lo primero que se ha de identificar es la problemática específica del sistema y ver como impactan las soluciones sobre el resto de variables.

En sistemas que requieran altos niveles de computación, es posible encontrase con limitaciones energéticas, ya que los dispositivos pueden estar alimentados por baterías. En estos casos, una solución es relegar los cálculos

a servicios en la nube para reducir el uso de procesador y con ello el consumo de energía [34]. Para ello hay que analizar las diferencias de coste entre procesar localmente la información y el coste de enviarla a la nube. Esto conlleva una reducción en la latencia a la hora de tener que actuar el sistema, ya que requiere enviar la información, procesarla de forma remota y devolver resultados, además de tener consecuencias sobre el ancho de banda necesitado.

Para reducir el uso de ancho de banda en casos de requerir mucha computación, una opción es usar técnicas de multi computación [35], donde se particionan los cálculos necesarios y una parte se realizan de forma local y otra de forma remota en servicios en la nube [36] [37]. Esto requiere estudiar los tipos de cálculos requeridos y la cantidad de información necesaria para cada uno, de forma que se puedan separar obteniendo una reducción real de esta ancho de banda consiguiendo mantener el consumo dentro de unos niveles aceptables.

El objetivo dentro de este proyecto, en cuanto a mecanismos de decisión, es el de optimizar los costos de los servicios en la nube. Como se ve en el análisis de las diferentes plataformas de *cloud*, en la sección 2.3, no están unificados los criterios de precios que se aplican, sino que depende de cada proveedor. Generalmente depende de el número de dispositivos conectados, el número de mensajes o capacidad de almacenamiento. Dependiendo del sistema que use cada servicio en la nube y de los requerimientos de ancho de banda se pueden escoger diferentes opciones para enviar mensajes.

En los sistemas que usan una facturación por número de mensajes, lo habitual es fijar un cierto número de tiempo entre mensajes, fijando a los valores razonables de actualización según los requisitos e negocio.[38]

Generalmente en los sistemas que toman como facturación el número de dispositivos, suele ir asociado también un número de mensajes máximos por periodo de tiempo. Dado que el número de dispositivos es algo que no se puede ajustar desde el *gateway*, el foco se ha de poner en ajustar el número de mensajes enviados a los límites del servicio, siempre dependiendo de las capacidades que haya en el transporte de datos.

Además la otra solución planteada en este proyecto es la de actualizar los valores de la red en base a una cierta variabilidad, de forma que se defina para

cada lectura de los sensores cuando se considera que el valor ha variado de forma significativa. Esta aproximación va bien para parámetros ambientales, como temperatura y humedad, cuya variabilidad en periodos cortos de tiempo es baja, pero puede tener más impacto a unas horas del día que a otras.

Capítulo 3

Diseño e implementación de la solución

3.1. Diseño del sistema

Para implementar el demostrador de IoT, conectado a plataformas *cloud*, que tenga en cuenta mecanismos de decisión y estrategias para enviar información de una red de sensores a servicios de *cloud computing* para IoT; necesitamos un *gateway* que implemente esta lógica, aquí es donde encaja el paradigma de *fog computing*. El *gateway* necesitará una lógica para decidir que información debe actualizar en los servicios en la nube y cual solo es de relevancia para la niebla.

El sistema estará basado en el modelo de integración de computación en la niebla, integrándose con la nube, de esta forma los datos enviados por los se recibirán en una *Raspberry Pi*, que los almacenará e implementará los mecanismos de decisión necesarios para discernir la información que se debe subir a la nube.

En la figura 3.1 están detallados cada uno de los componentes y servicios que integran el sistema, tanto a nivel de hardware con las placas OpenMote montadas sobre OpenBase y OpenBattery y Raspberry Pi, como a nivel de software detallando cada una de las aplicaciones utilizadas y servicios en la nube. La interacción para observar los resultados se realiza a través de los paneles de visualización de las distintas soluciones.

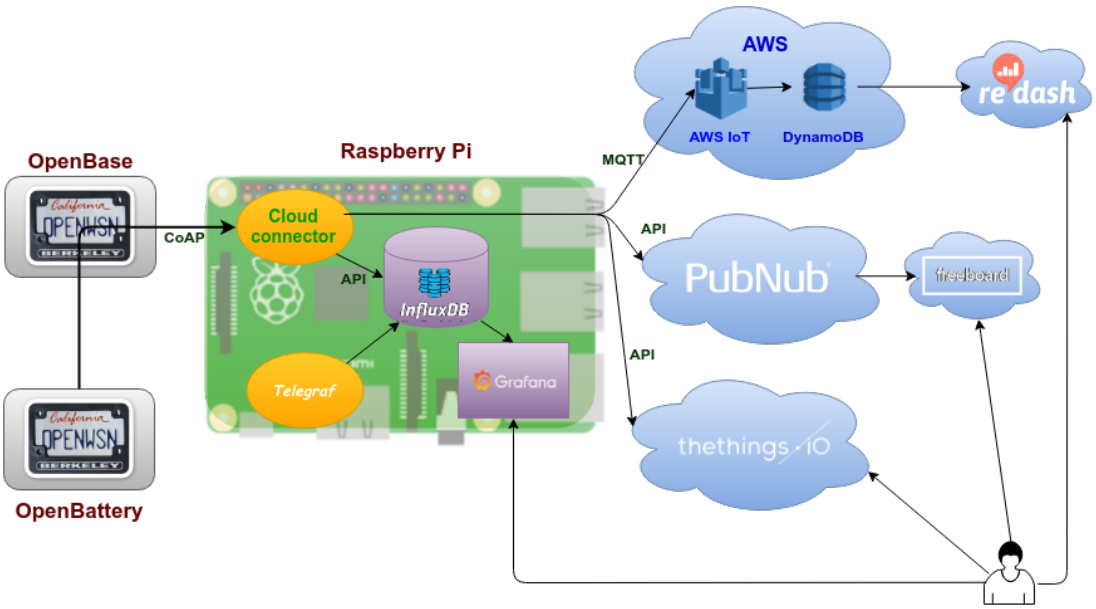


Figura 3.1: Esquema del proyecto

3.1.1. Hardware

Para este proyecto se han dispuesto de dos OpenMote, una conectada a OpenBattery, que además de proporcionar energía, dispone de sensores de humedad, temperatura, aceleración y luminosidad. La otra OpenMote está conectada a una OpenBase, que provee de conexión USB, la cual usaremos para enviar los datos al *gateway*.

La función de *gateway* se realizará con una *Raspberry Pi 2*, conectada vía Ethernet, con acceso a la red donde está conectada la OpenBase y a Internet.

OpenMote

El componente principal es el OpenMote-CC2538, que contiene un chip CC2538 de Texas Instruments con un microcontrolador Cortex-M3 de 32 bits que funciona hasta 32 MHz y un receptor de radio tipo CC2520, que opera con el estándar IEEE 802.15.4-2006 [39] sobre la banda de 2,4 GHz. Va montado sobre una placa con formato *XBee* que permite conectarlo tanto a OpenBase como a OpenBattery.[40]

OpenBase es una placa que permite una conexión serie por USB con el OpenMote-CC2538 a través de un chip FTDI FT232R, lo cual permite programar y conectarse con el CC2538. Además incorpora comunicaciones Ethernet 10/100 Mbps.

El tercer componente de esta familia es OpenBattery, que es una placa con interfaz para conectar una OpenMote-CC2538 usando *XBee*. Contiene un porta pilas 2xAAA para alimentarse de forma autónoma y tres sensores, un SHT21 que mide temperatura y humedad, un ADXL346 que mide aceleración y un MAX44009 para medir luminosidad, todos ellos se conectan usando un bus *I2C*.

Raspberry Pi

Es un ordenador de placa única de bajo coste desarrollado por la fundación Raspberry Pi con el objetivo de estimular la enseñanza de ciencias de la computación. Existen varios modelos de esta placa, el usado para este proyecto es el Raspberry Pi 2, que contiene un *System-on-a-chip* Broadcom BCM2836, compuesto por una CPU de cuatro núcleos ARM Cortex A7 a 900 MHz, una GPU VideoCore IV doble núcleo y 1 GB de RAM. [41]

A nivel de conectividad dispone entre otras de 4 puertos USB, 40 pines GPIO (*General Purpose Input/Output* - Entrada/Salida de propósito general), tarjeta microSD y Ethernet 10/100. Estas características, además de un consumo de entre 200 y 450 mA [42] más de los periféricos conectados, y su bajo costo (35$ [43]) la hacen ideal para este tipo de proyectos.

3.1.2. Software

Motes

Como software para OpenMote se usa OpenWSN, que es un proyecto de código abierto, optimizado para esta plataforma, que implementa la pila de protocols IEE802.15.4e y que permite crear redes en malla, además de otros

estandares para IoT como CoAP, RPL (*IPv6 Routing Protocol for Low-Power and Lossy Networks*) o 6LoWPAN (*IPv6 over Low power Wireless Personal Area Networks*). Incluye herramientas de monitorización, como *OpenVisualizer*. [40]

Gateway

Esta función la realiza la Raspberry Pi 2, ejecutando un sistema operativo Raspbian Jessie Lite, que es la distribución oficial de esta placa y no requiere de entorno gráfico.

La funcionalidad de este componente se puede descomponer en varias funciones:

Recogida de información de OpenMote y conector con la nube Para integrar la información de las OpenMotes se ha implementado un servicio en el *gateway*, que utilizando el protocolo CoAP consulte la información obtenida por los sensores, suba la información a la nube según unas estrategias definidas y además actualice la TSDB con la información y a que servicios en la nube se ha subido.

Almacenamiento de información La información recibida por las OpenMotes se guardará en una base de datos, ya que la información obtenida está formada por datos tomados en un instante de tiempo, una buena proximación es la de usar una base de datso específica para este tipo de información, como son las bases de datos de series temporales (TSDB). Para este proyecto se han analizado varias TSDB y se ha optado por InfluxDB como la mejor opción disponible.

Obtención de información del sistema De cara a monitorizar también el estado del *gateway* y poder analizar el rendimiento del sistema, es conveniente monitorizarlo, ya que se utiliza una TSDB, lo apropiado es un sistema que cargue esta información en la base de datos y usar las mismas herramientas de monitorización que para el resto de funcionalidad. De las herramientas analizadas se ha optado por Telegraf.

Monitorización La información obtenida y almacenada ha de ser fácilmente visualizable, para ello se han analizado algunas opciones y se utiliza Grafana.

3.2. Análisis de herramientas

3.2.1. Bases de datos de series temporales (TSDB)

Dado que se va a trabajar con métricas tomadas en diferentes instantes de tiempo, el uso de una base datos NoSQL (*Non Structured Query Language*) especializada en series temporales es bastante recomendable. Para decidir cuál utilizar se han analizado algunas de las alternativas existentes actualmente y sus posibles formas de monitorización y uso.

Son bases de datos NoSQL optimizadas para secuencias de datos medidos en determinados momentos y con un orden cronológico. Cuando se trabaja con grandes series de datos, el uso tradicional de bases de datos relacionales era el de hacer informes a partir de unos ciertos intervalos de tiempo, ya que el coste transaccional de explotar la información en forma de series numéricas era muy alto. Al ritmo que se pueden generar datos en la actualidad, especialmente con IoT, incluso con unas pocas semanas o meses sería suficiente para sobrecargar un sistema usando métodos tradicionales de bases de datos.

En cambio estos sistemas de TSDB permiten rápidamente analizar esta información, en mayor detalle, y realizar modelos de análisis predictivos, detección de anomalías, tendencias a largo plazo, etc. [44]

InfluxDB

Es un proyecto *open source*(MIT) gestionado por InfluxData, que almacena métricas y eventos en forma de series temporales usando LevelDB como sistema de almacenamiento clave-valor y no requiere el uso de esquemas.

Soporta cientos de miles de escrituras por segundo, además de sistemas de alta disponibilidad y *clustering*.

Se pueden hacer búsquedas utilizando un lenguaje parecido a SQL, aunque orientado a funciones temporales, el cual permite una rápida adaptación a quien ya este familiarizado con el uso de bases de datos relacionales. Los datos pueden ser etiquetados, lo que facilita búsquedas flexibles.

Soporta sistemas Linux, OS X y dispositivos ARM. Dispone de una API REST, además de 17 clientes para diferentes lenguajes.[45]

OpenTSDB

Funciona ejecutando unos TSD (*Time Series Daemon* - demonio de series temporales) y utilidades de linea de comandos, cada instancia puede ejecutar uno o varios de estos demonios, dependiendo de la carga que se necesite, tal y como se aprecia en su esquema en la figura 3.2. Esta información se guarda usando *HBase*.

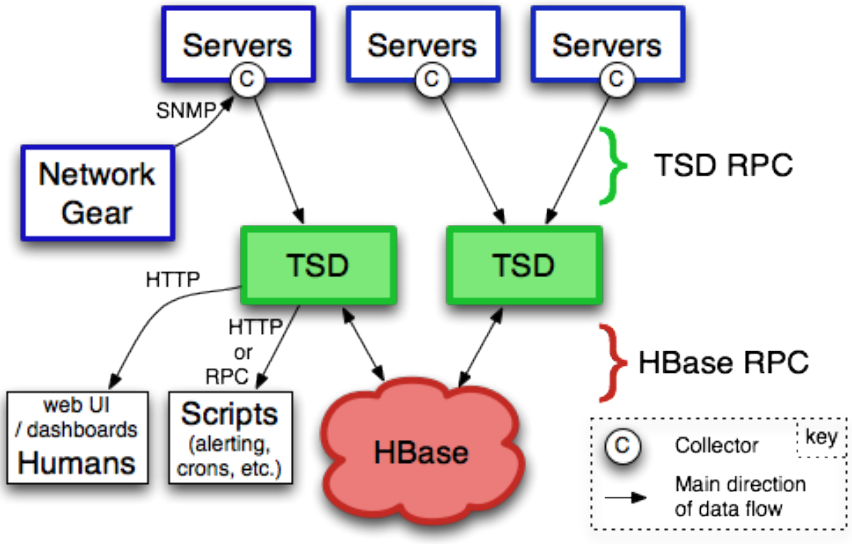


Figura 3.2: Arquitectura de OpenTSDB [46]

Los usuarios acceden a la información a través de los TSD, nunca directamente de HBase, usando una API HTTP REST.[46]

Prometheus

Ofrece un servicio de monitorización y alertas sobre una base de datos de series temporales. Su modelo de datos son series temporales identificadas por el nombre de la métrica y los datos en formato clave/valor usando un modelo *pull* via HTTP.

Tiene su propio sistema de monitorización, que permite crear paneles de control y gráficas y un sistema de alertas experimental.

Graphite

Se trata de una herramienta de monitorización capaz de ejecutarse en hardware sencillo. Esta compuesta por tres partes: *Carbon*, un demonio que escucha series de datos. *Whisper*, una librería que contiene una base de datos para series temporales. *Graphite Webapp*, una aplicación web que renderiza gráficos a demanda, utilizando *Django* y *Cairo*.

3.2.2. Herramientas de monitorización

Para visualizar la información recogida es conveniente usar alguna herramienta que permita crear gráficas y explotar los datos almacenados en la TSDB.

Grafana

Se utiliza para monitorizar series de datos, permitiendo crear paneles de control en tiempo real, con métricas y gráficas. Puede tomar datos de diferentes fuentes como Graphite, Elasticsearch, Prometheus, InfluxDB, OpenTSDB o KairosDB, de forma nativa u otras a través de plugins. Es una plataforme *open source* con una comunidad bastante activa

Chronograf

Se trata de la herramienta de monitorización desarrollada por InfluxData para InfluxDB. Su integración con esta base de datos es muy sencilla, simplemente se configura el origen de datos y se crean los paneles de control utilizando las *queries* de InfluxDB. Además permite el uso de plantillas de visualización. [47]

Pese a la fácil integración de esta herramienta con InfluxDB, las opciones para explotar datos están muy limitadas actualmente, solo permite crear paneles de control a partir de gráficas, estas bastante simples, donde no se pueden definir, por ejemplo, unidades de medida. El estado actual de esta herramienta es muy inicial.

3.2.3. Obtención de información

Es conveniente poder tomar métricas del sistema para visualizar el rendimiento en tiempo real.

Telegraf

Es un agente, escrito en Go, para recoger métricas y escribirlas en InfluxDB, de forma nativa, u otras posibles salidas mediante *plugins*. Este sistema de *plugins* se extiende no solo a las salidas, sino también a las posibles entradas, que pueden ser sistemas, como bases de datos, *ping* a máquinas o colas (*RabbitMQ*) o también pueden ser servicios en MQTT, StatsD o TCP Listeners.[48]

3.2.4. Conclusiones

InfluxDB ofrece una base de datos que necesita de muy pocos recursos de memoria, y es fácil de instalar, dispone de una pila integrada (TICK - Telegraf,

InfluxDB, Chhronograf, Kapacitor) que puede instalarse o bien en un equipo o de forma distribuida.

Para monitorizar está información, Grafana es una herramienta que permite conectarse a múltiples tipos de TSDB y es ampliamente usada con InfluxDB. La mayor parte de la carga de creación de gráficas se hace en el cliente Web, lo que la hace ideal para instalar en hardware simple.

Para obtener métricas del propio sistema, una buena opción es Telegraf, se integra muy fácilmente con InfluxDB, y permite que, además de monitorizar los datos de entrada, ver el consumo de recursos del sistema.

3.3. Implementación del conector entre OpenMote y los servicios en la nube

La aplicación central del sistema es la que extrae la información de OpenMote y la conecta tanto con InfluxDB como con los servicios en la nube. Para esta implementación se ha utilizado Python 2.7.9, que es la versión que trae Raspbian Jessie por defecto, se ha evaluado la posibilidad de usar Python 3.4 en este diseño, pero la librería CoAP desarrollada para OpenWSN solo soporta Python 2.7. Por otro lado el protocolo TLS 1.2 necesario para conectar por MQTT a AWS IoT [49] solo está soportado por la librería ssl de Python a partir de la versión 2.7.9 [50].

El diseño de la aplicación está basado en la idea de mantener separadas las responsabilidades de cada tipo de clase, de manera que sea lo más modular posible, cada una de estos tipos se ha desarrollado con un patrón de factoría abstracta [51] de la que heredan las diferentes clases. Hay factorias para dispositivos(*DeviceBase*), servicios en la nube(*CloudServiceBase*), TSDB(*TSDatabase*) y estrategias para subir datos a la nube(*StrategyBase*). Esto se aprecia en más detalle en el diagrama UML de la figura 3.4. Cada una de estas clases proporciona una patrón de diseño de fachada [51] para cada uno de los servicios utilizados, de manera que que a todas clases que derivan de la factoría se pueda llamar a los mismos métodos, independientemente de

su implementación, para realizar la acción correspondiente en cada uno de ellos.

Con este diseño, para incorporar, por ejemplo, un nuevo servicio en la nube a la aplicación, simplemente hay que heredar la clase *CloudServiceBase* y sobreescribir los métodos necesarios.

Por ejemplo, en el caso de la clase que sirve de base para los servicios en la nube, mostrada en el código 3.1, se puede ver como implementa la lógica genérica para enviar datos al sistema en la nube. En el constructor de la clase se define la estrategia, de manera que sean utilizables por los diferentes servicios en la nube, lo mismo ocurre en el método *insert_data*, cuya implementación es independiente del servicio. Lo que se ha de sobreescribir para cada servicio es el método *_send_data*, utilizando código específico para uno y en el constructor hacer la inicialización del mismo antes de llamar al constructor del padre. Destacar que a esta clase no se le puede llamar directamente, ya que la propia clase está definida como abstracta (*__metaclass__ = ABCMeta*) y tiene métodos definidos como abstractos (*@abstractmethod*), por lo que se ha de usar siempre mediante herencia y sobreescriendo al menos el método *_send_data*.

De los servicios que se han implementado, el más claro donde se aprecia este concepto es el usado para thethings.iO, ya que no requiere de ningún método más para establecer la conexión, tal y como se aprecia en el código 3.2. En él, en el constructor (*__init__*) se llama a la clase padre con la llamada a *super*, donde se le pasa la estrategia, y después se definen los *tokens* que se utilizarán para cada dispositivo. En el método *_send_data* de definen los pasos necesarios para enviar la información recogida al servicio *cloud*.

```
1  class CloudServiceBase(object):
2      """
3      Factory pattern class for cloud services
4      """
5      __metaclass__ = ABCMeta
6
7      def __init__(self, strategy=None):
8          """
9          :param strategy: Strategy object to send data to cloud.
10         :type strategy: StrategyBase.
11         """
12         self.name = 'Unknown'
13         if not strategy:
14             self.strategy = All()
15         else:
```

```
16              self.strategy = strategy
17
18      def insert_data(self, data, device_name):
19          """
20          Insert data into cloud service if strategy allows to.
21          :param data: Data to be inserted
22          :type data: dict.
23          :param device_name:
24          :type device_name:str.
25          :return: Class name
26          :rtype: str.
27          """
28          if self.strategy.has_to_send_data(data):
29              self._send_data(data, device_name)
30              return self.__class__.__name__
31          else:
32              logging.debug('Data is not going to be updated to {}
                    cloud service.'.format(self.name))
33              return False
34
35      @abstractmethod
36      def _send_data(self, data, device_name):
37          """
38          Send the data to the cloud
39          :param data: Data to be inserted
40          :type data: dict.
41          :param device_name:
42          :type device_name:str.
43          """
44          raise NotImplementedError
```

Código fuente 3.1 Clase base para servicios cloud

```
1  class CloudThingsIO(CloudServiceBase):
2      """
3      Configures thethings.io as cloud service
4      """
5
6      def __init__(self, tokens, strategy=None):
7          """
8          Initialize class
9          :param tokens: Dictionary of devices and tokens.
10         :type tokens: dict.
11         :param strategy: Strategy object to send data to cloud.
12         :type strategy: StrategyBase.
13         """
14         super(CloudThingsIO, self).__init__(strategy=strategy)
15         self.name = 'thethings.io'
16         self._tokens = tokens
17         self._thethings_connector = {}
18         for device_name, token in tokens.iteritems():
19             self._thethings_connector[device_name] = thethingsiO(
                   token)
20
21     def _send_data(self, data, device_name):
22         """
23         Insert data into thethings.io
24         :param data: Data to be sent.
25         :param device_name: dict.
26         :return: Class name.
27         :rtype: str.
```

```
28          """
29          tt = self._thethings_connector[device_name]
30          for variable, info in data.iteritems():
31              tt.addVar(variable, info)
32          try:
33              logging.debug('Sending data to thethings.iO')
34              response_code = tt.write()
35              logging.info('Sent to thethings.io with response code
                    {}: {}:{}'.format(response_code, device_name,
                   data))
36          except Exception:
37              logging.error('Unable to send data to thethings.io')
```

Código fuente 3.2 Clase para thethings.iO

Se usan clases bases, además de para los servicios en la nube, para los dispositivos, TSDB y estrategias. A pesar de que solo hay una implementación de TSDB es uso de este tipo de patrón de diseño permite adaptar fácilmente el sistema para usar otro tipo de bases de datos que no sea InfluxDB.

De la misma manera para el resto de tipos de clases. Cada una de estos tipos se encuentra en un módulo, tal y como representa el diagrama de módulos de la figura 3.3.

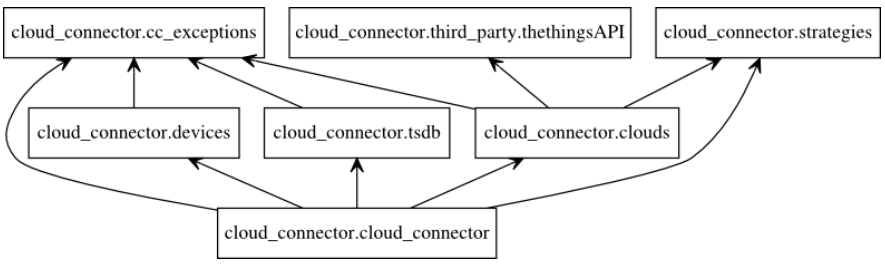


Figura 3.3: Diagrama de módulos

Están definidos como dispositivo OpenMote y un simulador que devuelve valores aleatorios dentro de un rango, como TSDB está InfluxDB y como servicios en la nube AWS IoT, PubNet y thethings.io.

Con OpenMote se utiliza la librería CoAP de OpenWSN [52], a través de la cual se obtienen los valores de temperatura, humedad y luminosidad de la placa OpenBattery. Los valores se obtienen tal y como vienen de los sensores, de manera que se ha de realizar la conversión, tal y como indican la hoja de especificación de los sensores [53] [54]. Para configurar una OpenMote hará falta darle un nombre y una dirección IPv6 a la que hacer las peticiones.

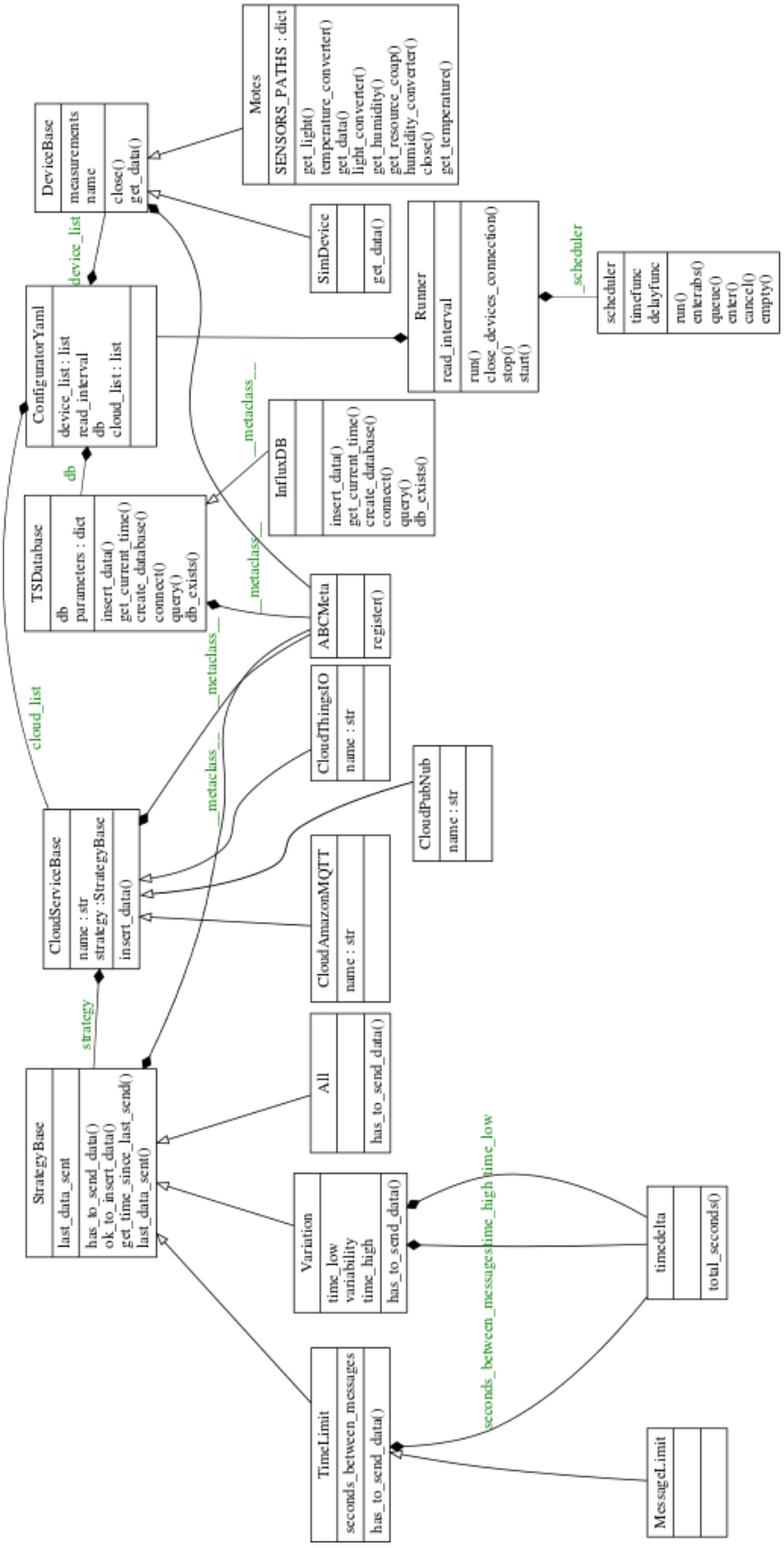
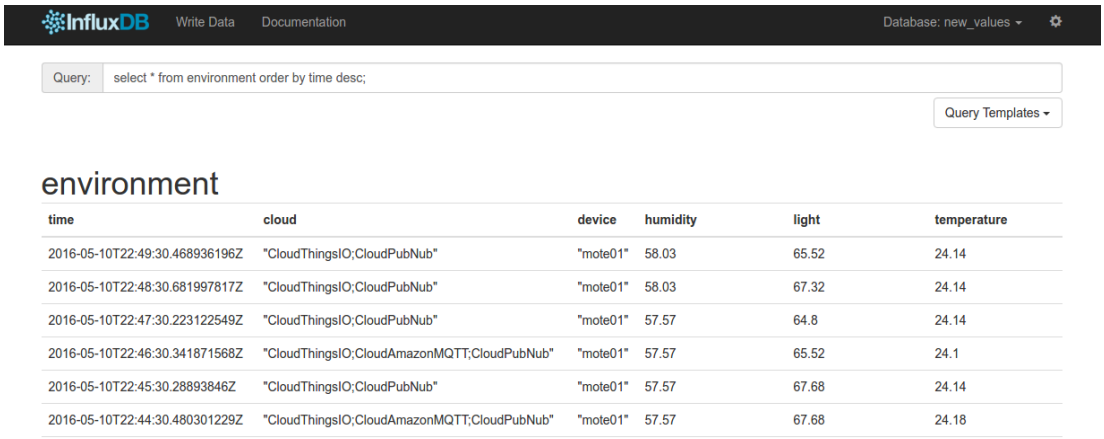


Figura 3.4: Diagrama UML de clases de la aplicación.

Para enviar la información a InfluxDB, se utiliza el cliente Python oficial, el cual se conecta a la base de datos usando HTTP REST. Los parámetros necesarios para utilizarla son la dirección IP o *hostname*, el puerto, el usuario y contraseña de InfluxDB y la base de datos donde almacenar los valores de los sensores, esta de no existir, se crea automáticamente. La información se guarda usando como medición(*measurement*) *environment* y con sus valores de tiempo(*time*), dispositivo(*device*), temperatura(*temperature*), humedad(*humidity*), luminosidad(*light*) y los servicios en la nube a los cuales se ha subido la información(*cloud*), separados por punto y coma, tal como refleja la figura 3.5.



time	cloud	device	humidity	light	temperature
2016-05-10T22:49:30.468936196Z	"CloudThingsIO;CloudPubNub"	"mote01"	58.03	65.52	24.14
2016-05-10T22:48:30.681997817Z	"CloudThingsIO;CloudPubNub"	"mote01"	58.03	67.32	24.14
2016-05-10T22:47:30.223122549Z	"CloudThingsIO;CloudPubNub"	"mote01"	57.57	64.8	24.14
2016-05-10T22:46:30.341871568Z	"CloudThingsIO;CloudAmazonMQTT;CloudPubNub"	"mote01"	57.57	65.52	24.1
2016-05-10T22:45:30.28893846Z	"CloudThingsIO;CloudPubNub"	"mote01"	57.57	67.68	24.14
2016-05-10T22:44:30.480301229Z	"CloudThingsIO;CloudAmazonMQTT;CloudPubNub"	"mote01"	57.57	67.68	24.18

Figura 3.5: Información de dispositivos en InfluxDB

En cuanto a los servicios en la nube, para los casos de thethings.iO y PubNub se han utilizados las API de cada uno de ellos. En el caso de thethings.iO esta se proporciona como un módulo Python, que se ha incorporado al código dentro del módulo *third_party* y que se conecta mediante HTTP REST. Mientras que PubNub dispone de un cliente oficial que se instala a través de los repositorios, este paquete usa actualmente WebSocket, aunque los desarrolladores pretenden que los protocolos sean lo más agnósticos posibles al desarrollo, de manera que puedan cambiarlo según sus necesidades. En el caso de AWS IoT no dispone de una API Python específica, por lo que se usa *Paho Python Client* [55], que proporciona una librería con soporte MQTT con seguridad TLS necesaria para conectarse a AWS IoT.

Los parámetros a configurar dependen de los requisitos y funcionamiento de cada uno de los servicios *cloud*. Para thethings.iO son los *tokens* para

cada uno de los dispositivos, PubNub necesitará de las claves de publicación y de suscripción, mientras que AWS IoT necesita del *hostname* del servidor AWS asociado, el puerto y la ruta para los ficheros de certificados tanto de la autoridad certificadora, como de cliente y claves privadas.

Cada uno de los servicios en la nube configurado puede incorporar su propia estrategia entre las definidas, estas son inyectadas en la clase que modela los servicios *cloud* como colaborador, usando el patrón de diseño de estrategia [51]. Las estrategias definidas son:

All Todo dato leído de los dispositivos se sube a la nube.

TimeLimit Si hace cierto tiempo definido que no envía información, se procede a enviar a la nube.

MessageLimit Igual que el anterior pero el tiempo va definido para no sobrepasar un límite de mensajes diarios.

Variation Se define un tiempo mínimo bajo el cual no se envía el mensaje(*time_low*), un tiempo máximo bajo el cual se envía siempre(*time_high*) y un diccionario con las variaciones permitidas para cada valor, cuando se sobrepase esta variación los datos se enviarán(*variability*).

La clase que orquesta la lógica de la aplicación es el *Runner*, que realiza una ejecución completa del ciclo de leer los datos de los dispositivos, enviarlos a los servicios en la nube, que según su configuración aplicará un mecanismo de decisión para subirlo o no, e introduce los datos en la TSDB, etiquetando a que servicio en la nube se han subido para poder tomar métricas sobre la base de datos. Esto se ejecuta de forma periódica a través de un *scheduler*, que repite la operación en cada ciclo de lectura definido en la configuración como *read_interval*.

Para configurar el sistema se usa un fichero YAML [56] donde se define la configuración de dispositivos, servicios en la nube y sus estrategias y TSDB, tal y como se refleja en el código 3.3.

```
devices:
  read_interval: 60
  mote01:
    name: mote01
    ipv6: bbbb::12:4b00:0615:a557
  mote02:
    name: mote02
    ipv6: bbbb::12:4b00:0615:a558

tsdb:
  influxdb:
    host: localhost
    port: 8086
    user: root
    password: root
    database: new_values

cloud:
  aws:
    host: A2KYAWFNYZU0I0.iot.eu-west-1.amazonaws.com
    port: 8883
    ca_path: ./keys/aws-iot-rootCA.crt
    cert_path: ./keys/cert.pem
    key_path: ./keys/privkey.pem
    strategy:
      type: Variation
      parameters:
        time_low: 60
        time_high: 300
        variability:
          temperature: 0.5
          humidity: 2
          light: 2
```

Código fuente 3.3 Fichero de configuración *config.yml*

También se incluye un módulo para gestionar las excepciones, cuyo diagrama UML está representado en la figura 3.6.

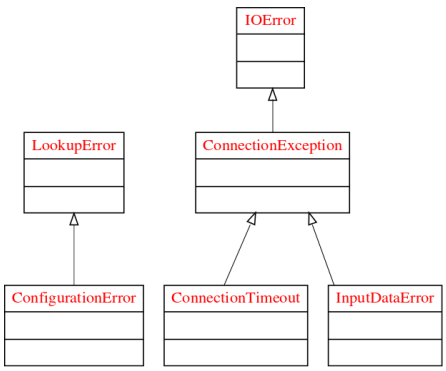


Figura 3.6: Diagrama UML de clases de excepciones

3.3.1. Políticas de calidad

Durante el desarrollo del proyecto se han mantenido unas buenas prácticas de calidad, para asegurar tanto el buen funcionamiento de la aplicación durante el desarrollo, que garantice un resultado final satisfactorio.

La primera de ellas es la existencia de pruebas unitarias, que se escriben en paralelo al código de la aplicación, esto permite tanto validar en el momento que se crea el código, como realizar regresiones automatizadas, en este caso usando el servidor de integración continua de GitLab para ejecutar las pruebas cada vez que se sube código. La cobertura de pruebas unitarias es de un 83 % de las líneas de código.

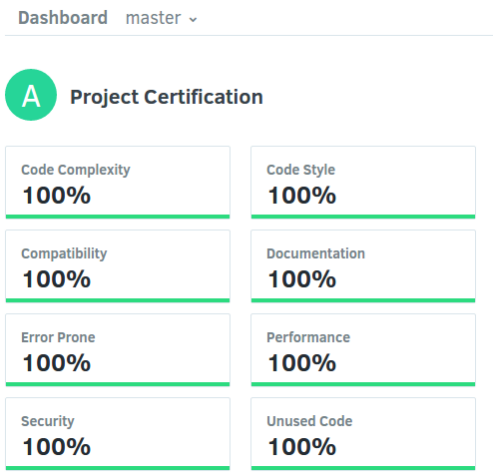


Figura 3.7: Certificación Codacy

GRADE	FILENAME	ISSUES	DUPLICATION	COMPLEXITY
A	Code/rpi/cloud/cloud_connector/tsdb.py	0	0	5
A	Code/rpi/cloud/cloud_connector/strategies.py	0	0	7
A	Code/rpi/cloud/cloud_connector/setup.py	0	0	-
A	Code/rpi/cloud/cloud_connector/devices.py	0	0	2
A	Code/rpi/cloud/cloud_connector/clouds.py	0	0	3
A	Code/rpi/cloud/cloud_connector/cloud_connector.py	0	0	8
A	Code/rpi/cloud/cloud_connector/cc_exceptions.py	0	0	1
A	Code/rpi/cloud/cloud_connector/__init__.py	0	0	-

Figura 3.8: Calificación y complejidad reportada por Codacy

Otra política ha sido usar un servicio de revisión de código, en este caso Codacy [57], que permite de forma automática hacer revisiones del código sobre estilo, documentación, complejidad, buenas practicas, etc. Con esto se intenta obtener un código claro, mantenible y bien documentado. En la figura 3.7 se ve el el resultado final del análisis. En la figura 3.8 se pueden ver los valores de complejidad ciclomática para cada uno de los módulos, manteniéndose por debajo de 10, lo que significa que el código esta escrito de forma simple y mantenible. [58]

Una vez publicado el código en GitHub se han mantenido estas mismas políticas, usando Travis CI como servicio de integración continua y Codacy como servicio de revisión de código para el nuevo repositorio. Al estar disponible el código bajo licencias libres, se pueden consultar las métricas de forma abierta en cada uno de estos servicios.

3.4. Instalación del sistema

A partir del diseño del sistema hay que proceder a instalar los diversos componentes.

3.4.1. Adquisición de datos desde plataforma OpenMote

Para adquirir la información de las placas OpenMote se ha de cargar el firmware, en este caso OpenWSN, habilitando la aplicación *csensors*, que es la encargada de, bajo una petición CoAP, hacer la lectura de los sensores de la placa OpenBattery. Para ello el primer paso es descargarse el firmware de OpenWSN:

```
1 git clone https://github.com/openwsn-berkeley/openwsn-fw.git
```

Dentro de la carpeta *openapps* se modifica el fichero *openapps.c* y con los demás *include* se añade:

```
1 #include "csensors.h"
```

Para habilitar *csensors* hace falta activarla en el fichero de compilación *openapps/SConscript*, como aplicaciones a construir y que deben estar presentes, modificando las siguientes listas:

```
# apps which should always be built
defaultApps    = [
    'c6t',
    'cinfo',
    'cleds',
    'cwellknown',
    'techo',
    'uecho',
    'uinject',
    'rrt',
    'csensors',
]

# apps which should always be present
defaultAppsInit    = [
    'c6t',
    'cinfo',
    'cleds',
    'cwellknown',
    'techo',
    'uecho',
    'rrt',
    'csensors',
]
```

Por último se compila el firmware y para cada OpenMote se conecta esta a la OpenBase y esta a un ordenador usando el puerto USB FTDI y se carga el firmware sobre ella:

```
# Compila el firmware
sudo scons board=OpenMote-CC2538 toolchain=armgcc oos_openwsn
# Carga el firmware sobre OpenMote, siento ttyUSBX el puerto USB
    al cual se conecta OpenBase
sudo scons board=OpenMote-CC2538 toolchain=armgcc oos_openwsn
    bootload=/dev/ttyUSBX
```

Para conectar OpenMote con Raspberry Pi se utiliza el puerto USB FTDI de OpenBase. A nivel de software se usa el *OpenVisualizer* que está disponible en los repositorios de OpenWSN. Esta aplicación crea un túnel IPv6 que permite conectar con la red *bbbb::/64*, además de una interfaz web, accesible por el puerto 3000, que permite monitorizar y configurar algunos de los parámetros de las OpenMote. Para lanzarlo se siguen los siguientes pasos:

```
git clone https://github.com/openwsn-berkeley/openwsn-sw.git
cd openwsn-sw/software/openvisualizer
sudo scons runweb
```

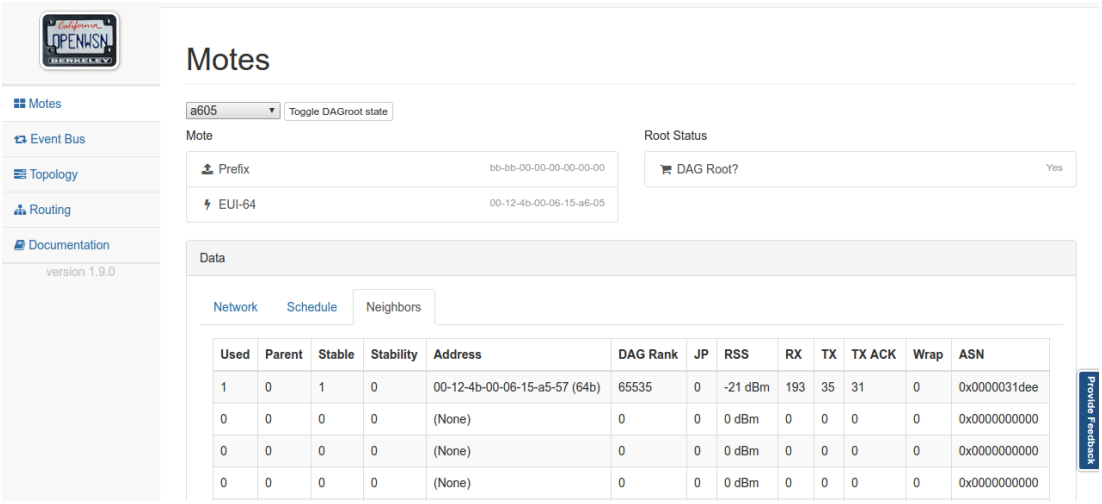


Figura 3.9: OpenWSN sobre plataforma OpenMote

Para que la OpenMote montada sobre OpenBase se vea con la que está montada sobre OpenBattery se ha de activar el *DAGroot* mediante el botón *Toogle DAGroot state*.

Una vez realizados estos pasos, la Raspberry Pi está conectada a el hardware OpenMote y se pueden hacer llamadas a través de CoAP a los sensores.

3.4.2. Base de datos de series temporales y recolección de datos del sistema

Para facilitar la instalación y despliegue de InfluxDB y Telegraf, se han creado paquetes *deb* para *Raspbian Jessie* a partir de los binarios con la versión v0.12.1, de manera que sea fácilmente instalable y actualizable.

Estos *scripts* se han subido a GitHub y se pueden ejecutar e instalar en la Raspberry Pi usando:

```
git clone https://github.com/eduarias/influx_rpi_deb_builder.git
./influxdb_deb_packaging.sh
./telegraf_deb_packaging.sh
sudo dpkg -i influxdb_0.12.1-1.deb
sudo dpkg -i telegraf-0.12.1-1.deb
sudo service influxdb start
sudo service telegraf start
```

La información del sistema que extrae Telegraf se guarda en una base de datos de InfluxDB con el nombre *telegraf*.

3.4.3. Monitorización de la información

Para monitorizar la información de la TSDB, tanto de los sensores de OpenMote como del sistema, se ha optado por usar Grafana, que permite crear paneles de control a partir de la información disponible en InfluxDB. Se ha utilizado la versión 3.0beta, ya que la versión 2.6 no permite conectarse a InfluxDB v0.12, a partir de un paquete compilado se instala con los siguientes comandos:

```
sudo dpkg -i grafana_3.0.0-beta41460625790_armhf.deb
sudo /bin/systemctl daemon-reload
sudo /bin/systemctl enable grafana-server
sudo /bin/systemctl start grafana-server
```

Una vez seguidos estos pasos Grafana está accesible a través del puerto 3000 de la Raspberry Pi, y se han creado paneles de control para monitorizar tanto el rendimiento de la placa como los valores extraídos de los sensores y a que servicio *cloud* se han subido.

3.4.4. Conector entre dispositivos y cloud

La aplicación que conecta las placas OpenMote con los servicios en la nube se puede instalar directamente desde GitHub, requiere instalar las librerías necesarias para el proyecto y configurar la aplicación usando el fichero *config.yml*, tal y como se especifica en la sección 3.3.

```
git clone https://github.com/eduarias/iot_fog.git
sudo pip install -r requirements.txt
cd cloud_connector
vim config.yml
python cloud_connector.py
```

3.5. Servicios en la nube

Cada uno de los servicios de *cloud computing* requiere de una configuración diferente. AWS IoT al ser un servicio dentro del catálogo de AWS, se ha de conectar con la base de datos DynamoDB para almacenar la información,

Figura 3.10: Panel de control de Grafana para monitorizar Raspberry Pi

esto requiere configurar reglas, permisos y la propia base de datos. Por el contrario thethings.iO al ser un servicio para IoT integrado, solo hay que crear una cuenta y registrar los equipos. Mientras PubNub es un sistema de mensajes, por lo que hace falta algún otro servicio que los recoja y los presente.

3.5.1. AWS

Amazon ofrece el servicio más completo de los analizados, ya que permite integrar los datos obtenidos de los dispositivos con la amplia red de servicios de AWS, pero eso supone también más complexidad para definir el ciclo de trabajo. En este caso ha habido, además de crear una cuenta en AWS, seguir los siguientes pasos, usando AWS CLI [59].

Crear el dispositivo IoT en su red Habiendo configurado apropiadamente AWS

CLI, para crear la relación de dispositivo con la red de AWS IoT se ejecuta:

```
$ aws iot create-thing --thing-name "mote_01"
{
        "thingArn": "arn:aws:iot:eu-west-1:111111111111:
           thing/sim_02",
        "thingName": "sim_02"
}
```

Crear los certificados necesarios para los dispositivos Para establecer la conexión con seguridad se han de crear un conjunto claves públicas y privadas para la comunicación, que junto con la clave para la autoridad certificadora se usa para configurar el cliente MQTT, además se ha de adjuntar los certificados a los dispositivos que harán uso de ellos:

```
$ aws iot create-keys-and-certificate --set-as-active --
   certificate-pem-outfile cert.pem --public-key-outfile
   publicKey.pem --private-key-outfile privkey.pem
$ aws iot attach-thing-principal --thing-name "mote_01" --
   principal "certificate-arn"
```

Crear una política para dispositivos Para ello se ha de definir en un fichero JSON con los permisos para esta política, crearla y adjuntarle los certificados:

```
$ cat iotpolicy.json
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action":["iot:*"],
        "Resource": ["*"]
    }]
}
$ aws iot create-policy --policy-name "PubSubToAnyTopic" --
   policy-document file://iotpolicy.json
{
    "policyName": "PubSubToAnyTopic",
    "policyArn": "arn:aws:iot:eu-west-1:111111111111:policy/
       PubSubToAnyTopic",
    "policyDocument": "{\n    \"Version\": \"2012-10-17\", \
       n    \"Statement\": [{\n        \"Effect\": \"Allow\"
       ,\n        \"Action\":[\"iot:*\"],\n        \"
       Resource\": [\"*\"]\n    }]\n}\n",
    "policyVersionId": "1"
}
$ aws iot attach-principal-policy --principal "certificate-
   arn" --policy-name "PubSubToAnyTopic"
```

Crear un perfil en AWS con permisos en AWS IoT y AWS DynamoDB Se ha de crear un rol en AWS IAM(*Access and Identity Management*), crear

una política de permisos y establecer el vínculo entre ellas:

```
$ cat role_policy.json
{
"Version": "2012-10-17",
"Statement": [{
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
            "Service": "iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
  }]
}
$ aws iam create-role --role-name iot-actions-role --assume-
   role-policy-document file://path-to-file/role_policy.json
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "sts:AssumeRole",
                    "Principal": {
                        "Service": "iot.amazonaws.com"
                    },
                    "Effect": "Allow",
                    "Sid": ""
                }
            ]
        },
        "RoleId": "AROAJND6WOTIONS77NNNN",
        "CreateDate": "2016-04-17T20:43:42.264Z",
        "RoleName": "iot-actions-role",
        "Path": "/",
        "Arn": "arn:aws:iam::111111111111:role/iot-actions-
           role"
    }
}
$ cat role_permissions.json
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "dynamodb:*", "lambda:InvokeFunction"],
        "Resource": ["*"]
    }]
}
$ aws iam create-policy --policy-name iot-actions-policy --
   policy-document file:///path-to-file/role_permissions.
   json
{
    "Policy": {
        "PolicyName": "iot-actions-policy",
        "CreateDate": "2016-04-17T20:51:01.873Z",
        "AttachmentCount": 0,
        "IsAttachable": true,
        "PolicyId": "ANPAINOPO545JVAI4NNNN",
        "DefaultVersionId": "v1",
        "Path": "/",
        "Arn": "arn:aws:iam::111111111111:policy/iot-actions
```

```
            -policy",
56          "UpdateDate": "2016-04-17T20:51:01.873Z"
57      }
58  }
59  $ aws iam attach-role-policy --role-name iot-actions-role --
      policy-arn "arn:aws:iam::941711456493:policy/iot-actions-
      policy"
```

Crear la regla que envíe los datos del dispositivo a DynamoDB Se ha de crear una regla que envíe parte de la información recibida a la base de datos, la selección se realiza a modo de búsqueda SQL, en este caso se ha seleccionado que envíe todo y se use como claves de DynamoDB el nombre de dispositivo y la marca de tiempo:

```
1   $ cat dynamo_iot_rule.json
2   {
3     "sql": "SELECT * FROM '#'",
4     "ruleDisabled": false,
5     "actions": [{
6         "dynamoDB": {
7           "tableName": "IoT_data",
8           "hashKeyField": "topic",
9           "hashKeyValue": "${topic()}",
10          "rangeKeyField": "timestamp",
11          "rangeKeyValue": "${timestamp()}",
12          "roleArn": "arn:aws:iam::111111111111:role/iot-
              actions-role"
13        }
14      }]
15  }
16  $ aws iot create-topic-rule --rule-name saveToDynamoDB --
      topic-rule-payload file://path-to-file/dynamo_iot_rule.
      json
```

Crear la tabla en DynamoDB Desde la consola de administración de DynamoDB se crea, mediante un asistente, la base de datos necesaria.

Visualización de la información

AWS no dispone de un panel de control desde el que visualizar la información, por lo que se debe hacer desde una aplicación externa, en este caso se ha optado por conectar un servicio web como Re:dash [60], que en su última versión permite hacer consultas a una base de datos DynamoDB y extraer la información en una tabla. Como lenguaje de consultas usa DQL, que es un lenguaje que permite hacer consultas tipo SQL a una base de datos DynamoDB que es NoSQL. La figura 3.11 muestra la información de DynamoDB en formato de tabla.

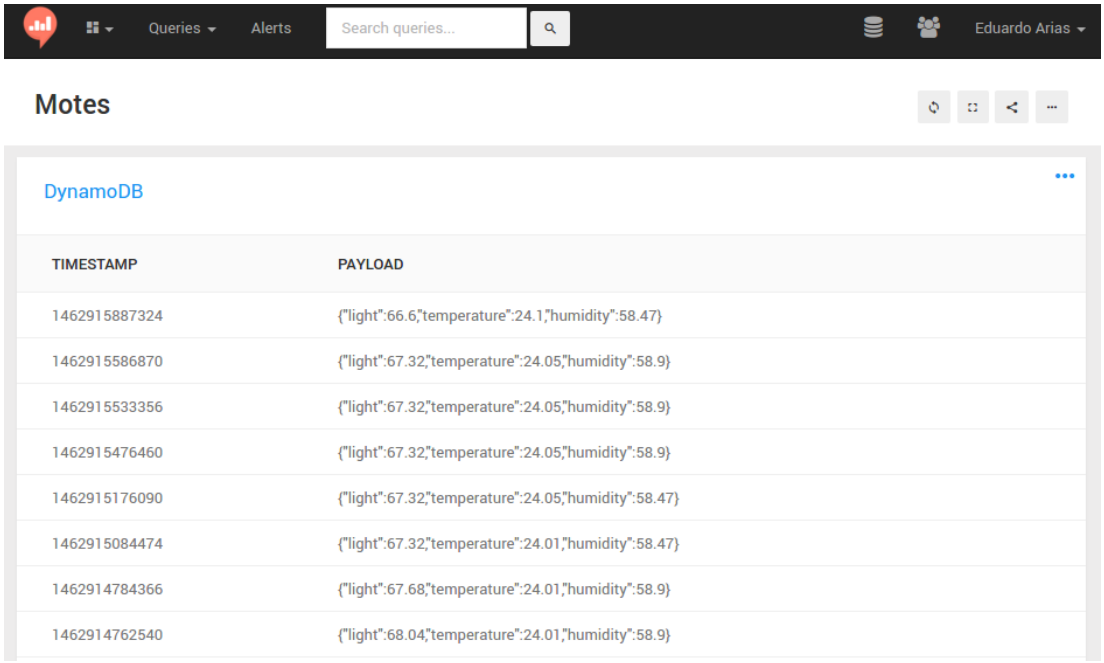


Figura 3.11: Panel de control de Re:dash sobre DynamoDB

3.5.2. thethings.iO

Al tratarse de una plataforma IoT específica, la configuración es muy sencilla y se realiza a través de su página Web [27], consiste en crear un nuevo dispositivo tal y como muestra la figura 3.12, con lo que se obtiene el token para ese dispositivo, que es el que se usa para configurarlo.

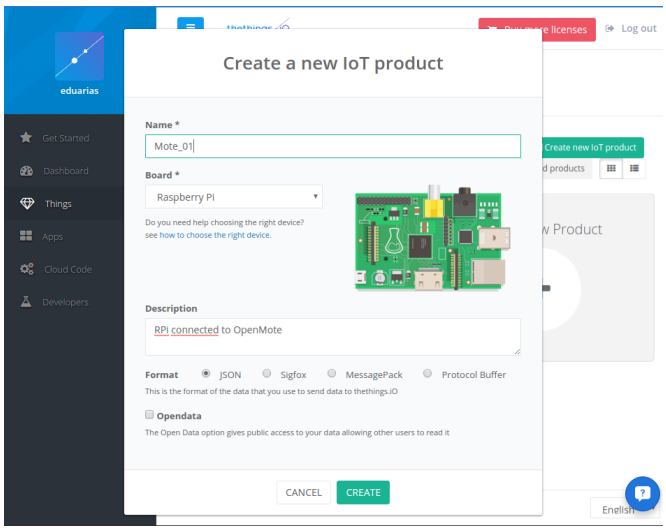


Figura 3.12: thethings.io: Crear nuevo dispositivo

El propio servicio thethings.iO se ocupa de almacenar la información recibida y proporciona un panel de control donde visualizarla de forma sencilla a través de *widgets*, entre los que dispone gráficos históricos, últimos valores, etc, como el de la figura 3.13.

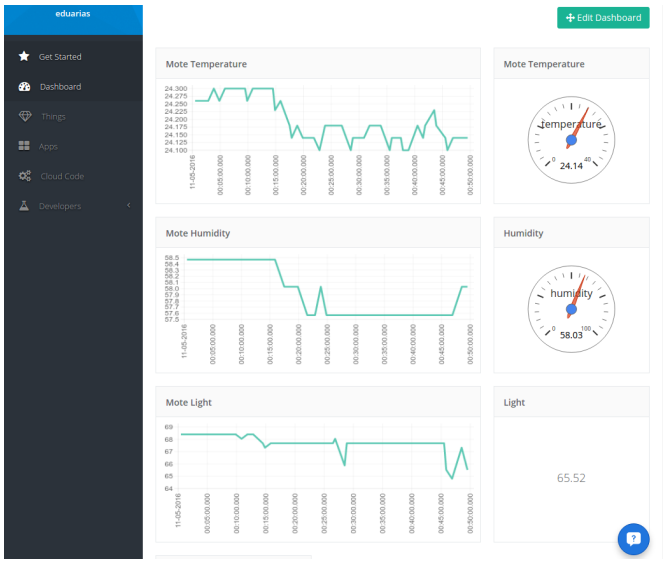


Figura 3.13: thethings.io: Panel de control

3.5.3. PubNub

En este caso, al ser un mero sistema de intercambio de mensajes, se necesita por otro lado un receptor de esos mensajes y que presente la información, para este diseño se ha escogido FreeBoard [61], que es una herramienta de visualización pensada para IoT.

Para crear un canal de comunicaciones en PubNub, desde su web, será necesario crear una nueva aplicación(*app*) y crear juegos de claves para publicar y suscribirse. A partir de ese momento ya se puede configurar, tanto en la Raspberry Pi como el origen de datos de FreeBoard para enviar y recibir la información.

En FreeBoard habrá que configurar el origen de datos como PubNub y usar la clave de suscripción y el nombre del canal. El panel de control de FreeBoard se configura a partir de *widgets* que se añaden a los diferentes paneles. Entre los *widgets* disponibles hay valores, gráficas, mapas, fotos, textos, etc. En la

figura 3.14 se ve el panel configurado con los datos ambientales extraídos de OpenBattery.

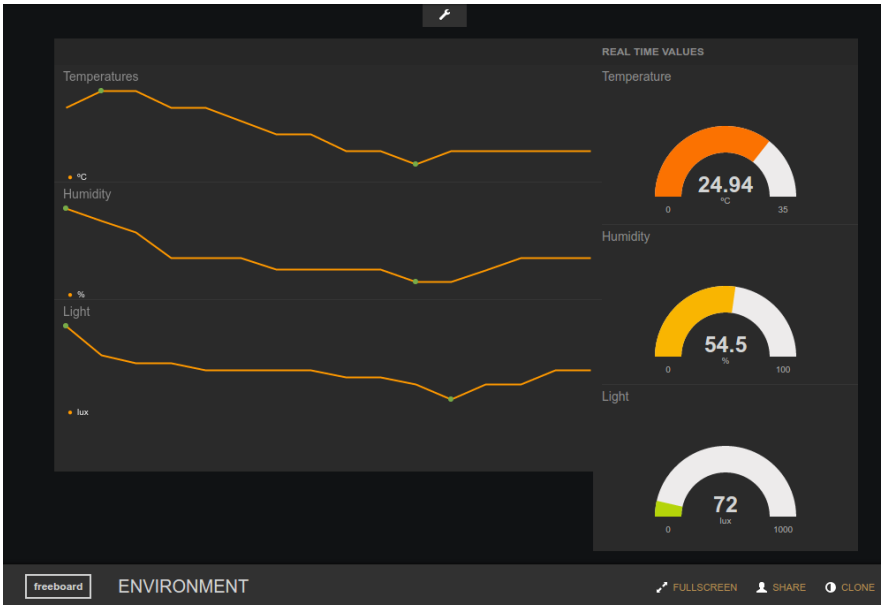


Figura 3.14: FreeBoard reprentando los valores de OpenBattery

Capítulo 4

Funcionamiento

Una vez completada la instalación de los servicios que componen el sistema, es el turno de establecer una configuración óptima para el sistema.

4.1. Configuración del sistema

Para configurar el sistema es necesario conocer los valores óptimos para enviar la información a cada servicio en la nube. Dado que la aplicación usa para su lógica principal un solo hilo, el tiempo mínimo de lectura está marcado por el valor medio de ejecución, ya que en caso de tener varias ejecuciones, estas se van poniendo en cola. Algunas de las librerías utilizas para subir valores a la nube si que funciona con multihilo, como el caso de PubNub, ese tiempo extra en el hilo no influye a la hora de marcar un tiempo mínimo de lectura.

Teniendo el cuenta el caso presentado, en el se leen los valores ambientales y como máximo se suben a los tres servicios en la nube (AWS, PubNub y thethings.iO), de forma empírica sobre el hardware comentado y usando una red Ethernet conectada a a Internet mediante una conexión de fibra óptica, el tiempo medio de ejecución es de unos 2,5 segundos, de forma aproximada. Por lo tanto un valor apropiado para tomar lecturas de los sensores sin tener problemas de colas, en un caso donde solo se ha de leer una placa OpenMote, es de 5 segundos.

Los parámetros óptimos para conectar con los servicios en la nube, dependen de las características de estos. El caso más sencillo es el de thethings.iO, donde se permiten 10.000 mensajes por dispositivo al día, por lo que el mecanismo de decisión más apropiado es *MessageLimit*, para mantener una posición conservadora el valor se puede fijar a 9.000 mensajes al día, lo que supone un máximo de uno cada 9,6 segundos.

En el caso de AWS y PubNub, dado que facturan por mensajes, sería conveniente usar un mecanismo de decisión variable(*Variation*) para reducir el número de mensajes que se envían a la nube, enviando solo aquellos en los que se produzcan variaciones significativas o que haya pasado un tiempo máximo desde el último envío.

Si se quiere conseguir un ahorro de mensajes enviados habrá que ajustar correctamente los valores utilizados para una estrategia variable. El valor más importante es la definición de la variabilidad de cada parámetro medido, para ello ahí que conocer tanto la la precisión de los sensores, como la que se quiere alcanzar con el sistema. Los otros dos valores a ajustar son el tiempo mínimo de envío, que se utiliza para controlar el máximo de mensajes que se envía si se da una situación de alta variabilidad y el tiempo mínimo de envío, que asegura que al menos se envía un mensaje cada cierto tiempo, incluso si no hay variabilidad, con lo que se fija unos ciertos mensajes mínimos.

Para determinar los valores mínimos de variabilidad del sistema, el primer paso es conocer el margen de error y la repetitividad de cada una de las medidas. En el caso del sensor SHT21 que monta la placa OpenMote, según la hoja de especificaciones [53], para la medición de temperatura existe una precisión de $\pm 0.3\,^{\circ}\mathrm{C}$ para valores entre 0 y $60\,^{\circ}\mathrm{C}$, con una repetitividad de $\pm 0.1\,^{\circ}\mathrm{C}$. Por lo tanto para temperatura no conviene fijar el valor de variabilidad por debajo de $0.1\,^{\circ}\mathrm{C}$ por la alta posibilidad de encontrar falsos cambios. Un valor mínimo que mantiene cierta precisión con baja probabilidad de falsos cambios es $0.2\,^{\circ}\mathrm{C}$.

Realizando el mismo proceso sobre los valores de humedad del sensor SHT21, este tiene una precisión de $\pm 2\,\%$ dentro del margen de $20\,\% - 80\,\%$ y una repetitividad de $\pm 0,1\,\%$. En este caso la hoja de especificaciones proporciona el valor de histéresis, que es de $\pm 0,1\,\%$, por lo que cambios por debajo de este valor difícilmente serán detectados. Por lo tanto el valor mínimo recomendado de variación de humedad es de $\pm 1\,\%$.

Para los valores de luminosidad el tema es más complejo, ya que no se disponen de valores fijos en la hoja de especificaciones [54] que nos den estos valores, porque dependen de factores como el espectro de luz detectado. Dado que el ojo humano puede detectar cambios de luminosidad de entre 4-10 lux [62], 4 lux puede ser un valor mínimo de variabilidad adecuado.

Otros valores a tener en cuenta para la configuración pueden ser la desconexión por inactividad, por ejemplo en el caso de AWS IoT, si se usa MQTT los clientes se desconectan tras 30 minutos de inactividad o 5 minutos en caso de usar WebSocket [63]. Por lo que si se está limitado en ancho de banda puede ser un factor a tener en cuenta.

4.2. Fase de pruebas del sistema

Basado en las recomendaciones de configuración anteriores, se ha configurado la aplicación para recoger información cada 5 segundos, que serán almacenadas en la TSDB. El servicio de thethings.iO se configura a un máximo de 9.000 mensajes. Dado que tanto AWS IoT como PubNub tienen un coste por mensaje, en ambos casos se usa un mecanismo de decisión variable, para probar dos combinaciones diferentes en AWS IoT se ajustan los envíos entre 60 y 1.800 segundos, con variaciones de temperatura de $\pm 0.2\,^{\circ}\mathrm{C}$, $\pm 1\,\%$ de humedad relativa y ± 4 lux de luminosidad. Para PubNub se apuesta por enviar datos en un periodo más corto de tiempo, entre 10 y 300 segundos, pero con unos margenes de variación superiores, una temperaturas de $\pm 0.5\,^{\circ}\mathrm{C}$, $\pm 2\,\%$ de humedad relativa y ± 10 lux de luminosidad.

Por lo tanto, se tiene una configuración de la que se obtienen datos de forma continua, que son los que se guardan en InfluxDB, y se envían a tres servicios en la nube, uno de ellos con una frecuencia de envío fija, thethings.iO, y otros dos se envían de forma variable, con diferentes parámetros de configuración, que son los casos de AWS y PubNub.

En una primera fase de pruebas se utiliza el dispositivo en interior, lo que supuestamente representa un escenario de pocos cambios de temperatura y humedad y algunos más de luminosidad.

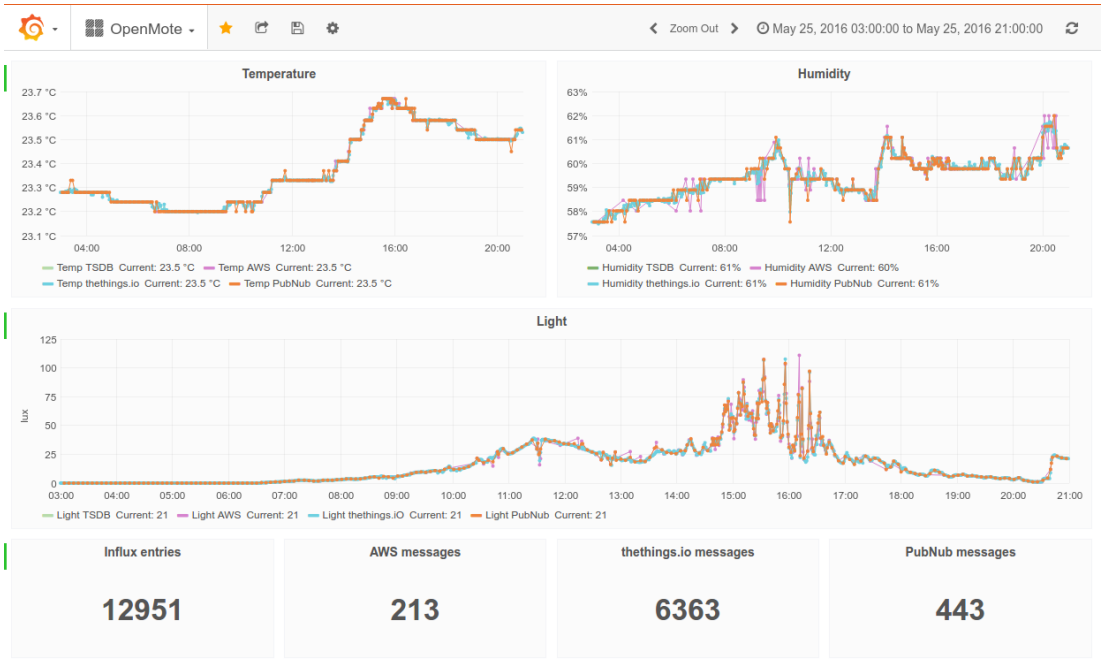


Figura 4.1: Datos recogidos en interior y enviados a los servicios en la nube durante 18h.

Como se aprecia en la figura 4.1 en un escenario de pocos cambios como el mostrado, las gráficas están bastante solapadas, dado las escasas variaciones, especialmente de temperatura y humedad durante este periodo, prácticamente cualquier sistema que envíe unas pocas muestras sería eficaz. Pero en cambio, analizando las dos horas donde se producen cambios en la luminosidad, en la figura 4.2, se refleja que más de casi la mitad de los mensajes enviados a PubNub y AWS IoT son durante este periodo de tiempo, donde estos adquieren mayor relevancia.

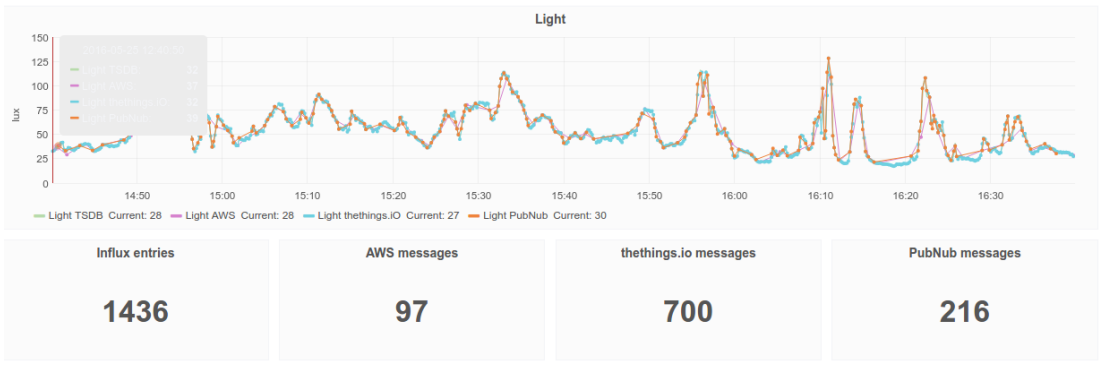


Figura 4.2: Datos recogidos en interior y enviados a los servicios en la nube durante las 2h con mayores variaciones.

Realizando la misma prueba en un ambiente exterior, con los mismos

parámetros de configuración que en el caso anterior, se aprecia en la figura 4.3 como el número de mensajes enviados hacia los servicios con mecanismos de decisión variables se ha prácticamente triplicado, debido especialmente a la luminosidad, ya que se pasa de valores entre 0-125 lux del caso de interior, a valores entre 0-10.000 lux sin variar los parámetros de variabilidad.

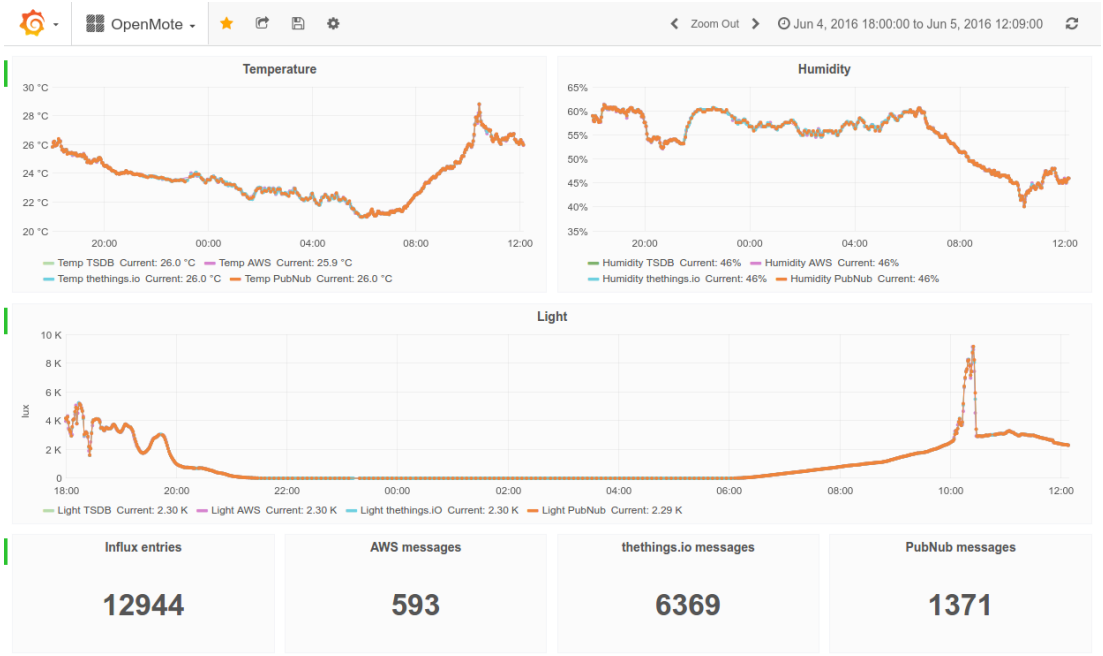


Figura 4.3: Datos recogidos en exterior y enviados a los servicios en la nube durante 18h.

Hay que tener en cuenta que la luminosidad es una magnitud con una función potencial [64], mientras que la variabilidad está definida usando valores fijos, por lo que al mantener los valores de variabilidad usados en interior, mientras que durante la noche se detecta cualquier variabilidad relevante, durante el día, con valores de varios ordenes de magnitud más altos esta misma variabilidad es de poca relevancia.

Si se analizan los datos tomados durante la noche, como se aprecia en la figura 4.4, los mensajes enviados a AWS y PubNub se ven reducidos enormemente por el mecanismo de decisión basado en la variación de la información.

En todos los casos analizados se puede ver claramente como las gráficas de los cuatro tipos de entradas están prácticamente superpuestas, y se ha conseguido una enorme reducción del número de mensajes al usar mecanismos de decisión basados en la variabilidad de los parámetros medidos.

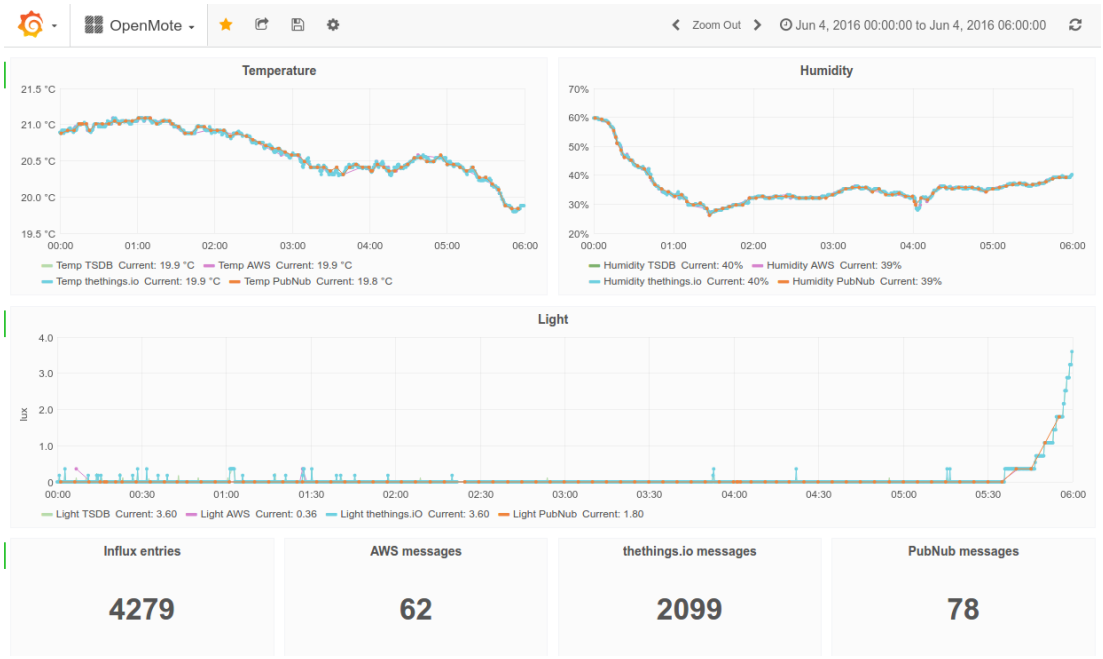


Figura 4.4: Datos recogidos en exterior y enviados a los servicios en la nube durante las 6h con menores variaciones.

Otro aspecto a valorar es que en el caso de AWS el número de mensajes enviados es inferior al de PubNub, en el primer caso la configuración se basa en tiempos de envío más largos y parámetros de variabilidad más pequeños, lo que viene a señalar que se están midiendo parámetros ambientales, donde la variación es relativamente baja en el tiempo, por lo que fijar parámetros de envío más largos suele suponer una mayor reducción del número de mensajes.

Para la tercera prueba del sistema se ha configurado la toma de datos de los sensores y almacenamiento en InfluxDB cada 5 segundos, igual que en los casos anteriores. Para los servicios en la nube se opta por variar los valores, para thethings.iO se mantiene una estrategia de envío constante, pero cambiándola a dos minutos entre envíos. AWS se configura con un rango de tiempo amplio, entre 10 y 3.600 segundos, mientras que para PubNub se define un rango temporal más acotado, de entre 60 y 600 segundos entre muestras, manteniendo la variabilidad de las medidas de ambos igual que en los casos anteriores.

Usando como muestra un periodo de 24 horas, como en la figura 4.5, sigue habiendo un ahorro de la más de la mitad de los mensajes para los dos casos

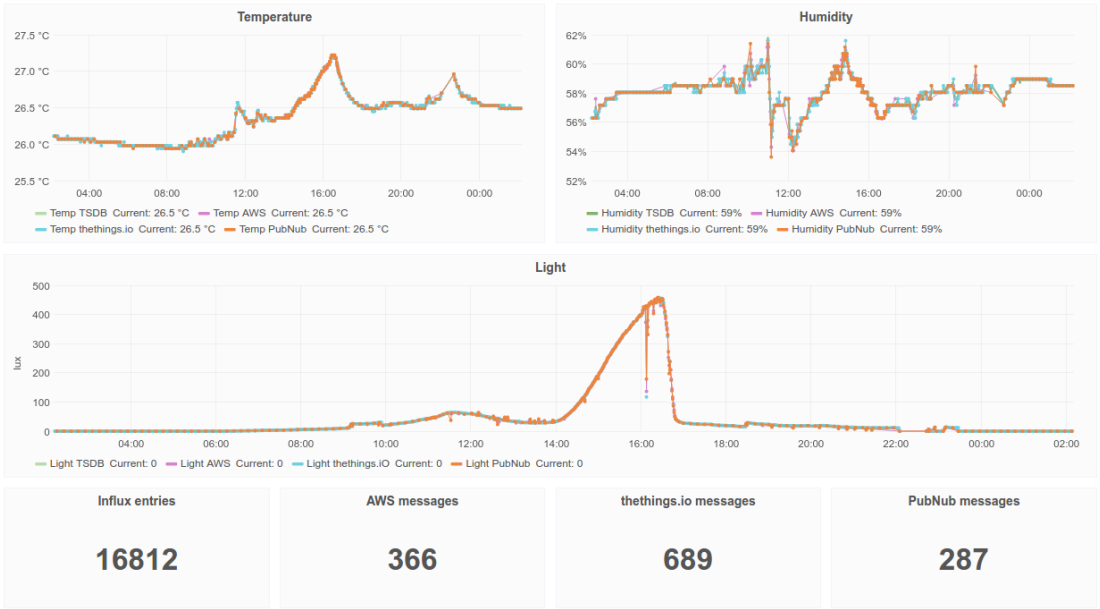


Figura 4.5: Datos recogidos en interior y enviados a los servicios en la nube durante 24h.

variables, AWS y PubNub, sobre las pruebas sobre envíos fijos, thethings.iO. Pero analizando esta serie en detalle se pueden encontrar tramos donde se envíen un número similar de mensajes, como el mostrado en la figura 4.6, donde AWS y thethings.iO envían 316 y 317 mensajes respectivamente.

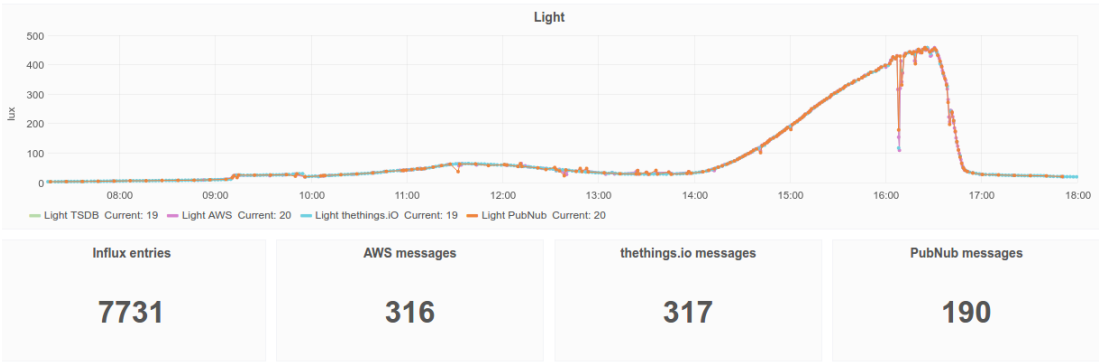


Figura 4.6: Datos recogidos en interior y enviados a los servicios en la nube con un número similar de mensajes.

Analizando diversas tramas de tiempo dentro de este margen, y dibujando la gráfica solo para estos dos servicios en la parte más estable de la recogida de información, como en la parte superior de la figura 4.7, se aprecia que usando un mecanismo de decisión variable se envían menos de la mitad de mensajes, y si en algún momento hay una medida que se desvía esta es enviada. En cambio, si se analiza la parte inferior de la misma figura ocurre el

efecto contrario, ya que en base al mecanismo de decisión variable esta parte requiere de mayor precisión al tener una mayor variabilidad. Esto significa que en caso que el sistema en la nube tenga que tomar decisiones a partir de la información recibida, usando un mecanismo variable se obtiene una respuesta más rápida que usando tiempos de envío fijos.

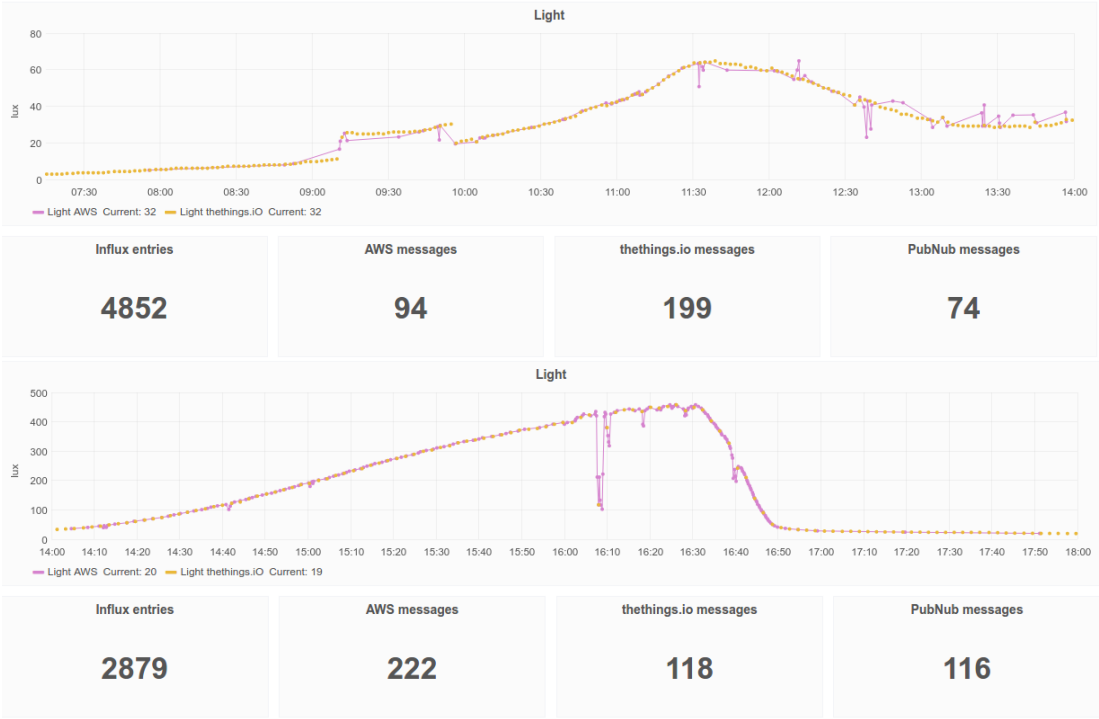


Figura 4.7: Datos recogidos en interior y enviados a los servicios en la nube desglosado en diferentes margenes de tiempo.

Capítulo 5

Conclusiones

Durante el desarrollo de este proyecto se ha hecho un análisis de situación actual de IoT con respecto a los servicios en la nube. Se ha mostrado su arquitectura, protocolos y algunos de los servicios más importantes en la actualidad.

Esta interacción con los servicios en la nube tiene un coste, de ahí que se haya utilizado como intermediario un sistema de computación en la niebla o *fog computing*, que interactúa entre los dispositivos y la *niebla*, proporcionando la capacidad de decidir que información se debe enviar a la nube de forma que se puedan reducir estos costes.

Como existen diferentes formas de facturar por parte de los distintos servicios en la nube se ha desarrollado una aplicación que permita configurar cada servicio con una estrategia de subida distinta. Algunos servicios facturan por dispositivo conectado con un límite de mensajes, para los cuales, si este límite nos permite una resolución suficiente, una forma óptima es buscar el límite.

Mientras que para sistemas donde la facturación es por mensaje (o millón de mensajes) se ha diseñado un mecanismo de decisión variable que permite enviar únicamente la información si hay una variación definida. Esto, tal y como se ve en las pruebas, permite reducir el número de mensajes o bien concentrarlos en los momentos de mayores cambios, actuando de forma similar a un sistema de compresión con perdidas [65].

La solución presentada se ha desenvuelto bien con mediciones de ambiente que tienen características lineales, como la humedad y la temperatura, pero ha tenido más dificultades cuando se trata de variables con funciones no lineales, como la luminosidad en este caso, cuya percepción de variabilidad para el ser humano registra funciones potenciales [64]. Para estos casos se habría de desarrollar una estrategia variable, pero cuya variabilidad viniese dada por una función, en lugar de valores fijos.

Una vez enviada la información a los servicios en la nube, se han mostrado ejemplos de presentación de estos valores, tanto en servicios que ya lo incluían, como el caso de thethings.iO, como dando soluciones a través de otros servicios en la nube como FreeBoard, que es capaz de procesar los mensajes recibidos por PubNub, como re:dash, que permite leer sobre una base de datos DynamoDB de AWS a la que se ha enviado la información recibida por AWS IoT.

El impacto energético de la solución presentada, en el caso de no requerir un *gateway*, es algo mayor que el de una solución donde se envían los datos directamente a la nube, ya que requiere de una capa de *gateway* intermedia, para realizar las funciones de computación en la niebla, que en este caso es la placa Raspberry Pi, que tiene un consumo estimado de unos 200-300 mA en niveles de baja carga [66], como es el caso. A esto sumado al coste energético de los servicios de *cloud computing* [67], donde debido a la naturaleza de los mensajes enviados por dispositivos IoT, lo hace bastante bajo en comparación, ya que requiere anchos de banda del orden de 1 kps.

El uso de metodologías ágiles para la planificación y seguimiento del proyecto ha permitido una planificación flexible, donde se ha podido gestionar riesgos e ir cumpliendo con las entregas requeridas del proyecto, definiendo estas como hitos donde existe menos flexibilidad.

En cuanto a líneas futuras a partir de este trabajo, cabe destacar:

Uso de funciones de variabilidad Ciertos valores, como la luminosidad o la potencia acústica, son percibidos de forma no lineal, por lo que si se definen valores fijos de variabilidad, estos no reflejan la sensación de variabilidad de las personas, por lo que sería conveniente usar funciones como mecanismo de decisión para estos parámetros.

Filtro para posibles errores de medición Al usar sistemas de envío basados en variabilidad, cada medición errónea de forma aislada puede suponer el envío de dos mensajes, uno notificando la variación y otro notificando la vuelta al valor normal. Si bien durante el desarrollo de este proyecto no se han encontrado muchos casos en los que esto sucede, en otro tipo de dispositivos puede ser un problema a solventar.

Post-procesar información recibida en la nube En este trabajo, al utilizar AWS se ha enviado la información recibida a DynamoDB para almacenarla, pero servicios como AWS o Google Cloud ofrecen muchas posibilidades para procesar esa información, como por ejemplo desde AWS Lambda se puede calcular las máximas y mínimas diarias, o pasar la información a procesos de Machine Learning para realizar predicciones, Elasticsearch para para análisis de datos, etc...

Acrónimos

6LoWPAN *IPv6 over Low power Wireless Personal Area Networks*. 44, *ver glosario:* 6LoWPAN

API *Application Programming Interface* - interfaz de programación de aplicaciones. 11, 31, 32, 34, 46, 54

AWS *Amazon Web Services*. 26–28, 35, 36, 61

CoAP *Constrained Application Protocol*. 20, 24, 33, 34, 44, 52, 58, 60

D2D *device to device* - dispositivo a dispositivo. 18

DoD *Definition of Done* – definición de acabado. 12

GPIO *General Purpose Input/Output* - Entrada/Salida de propósito general. 43, *Glosario:* GPIO

HMI *Human to Machine Interactions* - interacción humano máquina. 38

HTTP *Hypertext Transfer Protocol* - protocolo de transferencia de hipertexto. 20, 21, 24, 26–28, 34, 47

HTTPS *Hypertext Transfer Protocol Secure* - protocolo seguro de transferencia de hipertexto. 21

IaaS *Infrastructure as a Service* - infraestructura como servicio. 34, *Glosario:* IaaS

IoT *Internet of Things* - Internet de las cosas. 9–11, 16, 18, 20, 22, 28, 30, 35, 37, 38, 41, 44, 45, 62, 67, 77, *ver glosario:* Internet de las cosas

M2M *machine to machine* - máquina a máquina. 18, 22, 38

MQTT *MQ Telemetry Transport*. 20, 22–24, 26–28, 32–34, 48, 49

NoSQL *Non Structured Query Language*. 45, *ver glosario:* NoSQL

QoS *Quality of Service* - calidad de servicio. 23, 25

REST *Representational State Transfer* - Transferencia de Estado Representacional. 24, 27, 29, 31–33, 46

RPL *IPv6 Routing Protocol for Low-Power and Lossy Networks*. 44, *Glosario:* RPL

SDK *Software Development Kit* - kit de desarrollo de software. 26, 34, 35

TCP *Transmission Control Protocol* - protocolo de control de transmisión. 21, 48

TSD *Time Series Daemon* - demonio de series temporales. 46

TSDB *Time Series Database* - base de datos de series temporales. 11, 44, 45, 47, 49, 52, 55, 61, 71

Glosario

Big Data Concepto que hace referencia al almacenamiento de grandes cantidades de datos y a los procedimientos usados para encontrar patrones repetitivos dentro de esos datos. 36, 38

I2C Bus de datos serie que se utiliza principalmente internamente para la comunicación entre diferentes partes de un circuito, por ejemplo, entre un controlador y circuitos periféricos integrados.. 43

System-on-a-chip Se trata de un chip que integra todos o gran parte de los módulos que componen un ordenador o cualquier otro sistema informático o electrónico.. 43

XBee Es un factor de forma para módulos compatibles con radio.. 42, 43

cloud computing Computación en la nube. Paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es Internet. 11, 35, 37, 41, 61, 78

commit Confirmación efectuada sobre un repositorio de *git*. 7, 15

fog computing Computación en la niebla. Es un modelo que promueve la deslocalización de algunas capacidades de la nube a dispositivos situados en el borde de la red. 7, 16, 37, 41, 77

6LoWPAN Estándar que posibilita el uso de IPv6 sobre redes basadas en el estándar IEEE 802.15.4. Hace posible que dispositivos como los nodos de una red inalámbrica puedan comunicarse directamente con otros dispositivos IP. 44

GPIO Son pines genéricos, cuyo comportamiento, tanto de entrada como de salida, se puede controlar por el usuario en tiempo de ejecución.. 43

IaaS Modelo de servicios en la nube en el que el usuario contrata únicamente las infraestructuras tecnológicas (capacidad de procesamiento, de almacenamiento y/o de comunicaciones).. 34

Internet de las cosas Es un concepto que se refiere a la interconexión digital de objetos cotidianos con internet. 9

JSON *JavaScript Object Notation* es un formato de texto ligero, que usando un subconjunto de la notación literal de objetos de JavaScript, se usa para el intercambio de datos.. 63

NoSQL Amplia clase de sistemas de gestión de bases de datos que difieren del modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS) en aspectos importantes, el más destacado es que no usan SQL como el principal lenguaje de consultas. 45

RPL Protocolo de enrutamiento para dispositivos de bajo consumo, definido en RFC6550 de IETF. 44

Bibliografía

[1] Kevin Ashton. That 'internet of things' thing. 2009. URL http://www.rfidjournal.com/articles/view?4986. Visitado el 03-04-2016.

[2] Peter Waher. *Learning Internet of Things*. Packt Publishing, 2015.

[3] Ubiworx. Bridging sensors and actuators to iot cloud services. 2016. URL http://www.ubiworx.com/ubiworx/. Visitado el 23-04-2016.

[4] Preston Holmes. Gcpnext16 - iot - from small data to big data: Building solutions with connected devices. 2016. URL https://www.youtube.com/watch?v=8NbP07OEGsQ. Visitado el 03-04-2016.

[5] Wikipedia. Hypertext transfer protocol. 2016. URL https://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol. Visitado el 01-04-2016.

[6] Wikipedia. Http/2. 2016. URL https://es.wikipedia.org/wiki/HTTP/2. Visitado el 01-04-2016.

[7] Asier Marqués. Conceptos sobre apis rest. 2013. URL http://asiermarques.com/2013/conceptos-sobre-apis-rest/. Visitado el 01-04-2016.

[8] Malte Ubl y Eiji Kitamura. Introducción a los websockets: incorporación de sockets a la web. 2010. URL http://www.html5rocks.com/es/tutorials/websockets/basics/. Visitado el 31-03-2016.

[9] Peter Lubbers y Frank Greco. Html5 websocket: A quantum leap in scalability for the web. 2016. URL http://www.websocket.org/quantum.html. Visitado el 23-04-2016.

[10] Antonio Teruel. Websocket (html5) y la seguridad en bases de datos: Salto cuántico en iot. 2014. URL http://edriel.com/html5-websockets-y-la-seguridad-en-bases-de-datos-salto-cuantico-en-iot/. Visitado el 31-03-2016.

[11] MQTT.org. Conceptos sobre apis rest. 2016. URL http://mqtt.org/. Visitado el 01-04-2016.

[12] José Antonio Yébenes Gálvez. ¿qué es mqtt? 2015. URL https://geekytheory.com/que-es-mqtt/. Visitado el 01-04-2016.

[13] Carsten Bormann. Coap, rfc 7252 constrained application protocol. 2016. URL http://coap.technology/. Visitado el 04-04-2016.

[14] Amazon. What is aws iot? 2016. URL http://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html. Visitado el 02-04-2016.

[15] Amazon. Cómo funciona la plataforma aws iot. 2016. URL https://aws.amazon.com/es/iot/how-it-works/. Visitado el 24-04-2016.

[16] Amazon. Precios de aws iot. 2016. URL https://aws.amazon.com/es/iot/pricing/. Visitado el 02-04-2016.

[17] Amazon. Precios de amazon dynamodb. 2016. URL https://aws.amazon.com/es/dynamodb/pricing/. Visitado el 03-05-2016.

[18] Google Cloud Platform. Architecture: Real time stream processing - internet of things. 2016. URL https://cloud.google.com/solutions/architecture/streamprocessing. Visitado el 03-04-2016.

[19] Google Cloud Platform. Overview of internet of things. 2016. URL https://cloud.google.com/solutions/iot-overview. Visitado el 03-04-2016.

[20] Cloud Pub/Sub. Gcpnext16 - iot - from small data to big data: Building solutions with connected devices. 2016. URL https://cloud.google.com/pubsub/. Visitado el 03-04-2016.

[21] Cindy Castillo. *Using Oracle Internet of Things Cloud Service Release 16.1.5*. O'Reilly Media, 2016. URL https://docs.oracle.com/cloud/latest/iot/IOTGS/IOTGS.pdf.

[22] Oracle. Using oracle internet of things cloud service. 2016. URL http://docs.oracle.com/cloud/latest/iot/IOTGS/toc.htm. Visitado el 04-04-2016.

[23] Xively. Blueprint and messaging quick start guide. 2016. URL http://developer.stage.xively.us/tutorials/blueprint-and-messaging/. Visitado el 24-04-2016.

[24] Xively. What is xively? 2016. URL https://xively.com/whats_xively/. Visitado el 04-04-2016.

[25] Kate O'Flaherty. What can the internet of things do for smbs? 2014. URL http://www.techradar.com/news/world-of-tech/what-can-the-internet-of-things-do-for-smbs--1218944. Visitado el 04-04-2016.

[26] thethings.io. Features to boost your iot products. 2016. URL https://thethings.io/features/. Visitado el 24-04-2016.

[27] thethings.io. thethings.io. 2016. URL http://thethings.io/. Visitado el 04-04-2016.

[28] Temboo. Temboo. 2016. URL https://temboo.com/. Visitado el 04-04-2016.

[29] PubNub. Pubnub. 2016. URL https://www.pubnub.com/. Visitado el 10-05-2016.

[30] Stephen Blum. Pubnub: Protocol independence. 2014. URL https://www.pubnub.com/community/discussion/71/protocol-independence. Visitado el 10-05-2016.

[31] Barb Darrow. Amazon remains the top dog in cloud by far, but microsoft, google make strides. 2016. URL http://fortune.com/2015/05/19/amazon-tops-in-cloud/. Visitado el 05-05-2016.

[32] Biren Gandhi. Fog computing reality check: Real world applications and architectures. IoT Evolution Expo, 2015. URL http://www.slideshare.net/biren_gandhi/fog-computing-reality-check-real-world-applications-and-architectures.

[33] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, y Sateesh Addepalli. Fog computing and its role in the internet of things. En *MCC '12 Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, págs. 13–16. ACM, 2012.

[34] Nur Idawati, Md Enzai, y Maolin Tang. A taxonomy of computation offloading in

mobile cloud computing. 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, 2014.

[35] K. Sinha y M. Kulkarni. Techniques for fine-grained, multi-site computation offloading, in cluster, cloud and grid computing (ccgrid). *2011 11th IEEE/ACM International Symposium*, págs. 184–194, 2011.

[36] K. Yang. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *IEEE Communications Magazine*, 46(1):56–63, 2008.

[37] Byung-Gon Chun y Petros Maniatis. Dynamically partitioning applications between weak devices and clouds. *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing and Services: Social Networks and Beyond*, (7), 2010.

[38] Fei Li, Michael Vögler, Markus Claeßens, y Schahram Dustdar. Efficient and scalable iot service delivery on cloud. IEEE Sixth International Conference on Cloud Computing, 2013.

[39] WG802.15 Wireless Personal Area Network (WPAN) Working Group. 802.15.4-2006 - ieee standard for information technology– local and metropolitan area networks. *IEEE*, 2008. URL https://standards.ieee.org/findstds/standard/802.15.4-2006.html.

[40] OpenMote. Open hardware for internet of things. 2016. URL http://www.openmote.com/. Visitado el 13-04-2016.

[41] Wikipedia. Raspberry pi. 2016. URL https://es.wikipedia.org/wiki/Raspberry_Pi/. Visitado el 08-05-2016.

[42] Raspberry Pi Dramble. Power consumption. 2016. URL http://www.pidramble.com/wiki/benchmarks/power-consumption. Visitado el 08-05-2016.

[43] Allied Electronics. Raspberry pi raspberry pi 2, model b. 2016. URL http://www.alliedelec.com/raspberry-pi-raspberry-pi-2-model-b/70465426/. Visitado el 08-05-2016.

[44] Ellen Friedman y Ted Dunning. *Time Series Databases*. O'Reilly Media, 2014.

[45] Influxdata. Getting started with influxdb. 2016. URL https://docs.influxdata.com/influxdb/v0.12/clients/api/. Visitado el 07-04-2016.

[46] OpenTSDB. How does opentsdb work? 2016. URL http://opentsdb.net/overview.html. Visitado el 07-04-2016.

[47] Influxdata. Getting started with chronograf. 2016. URL https://influxdata.com/time-series-platform/chronograf/. Visitado el 07-04-2016.

[48] Influxdata. Getting started with telegraf. 2016. URL https://docs.influxdata.com/telegraf/v0.12/introduction/getting-started-telegraf/. Visitado el 06-04-2016.

[49] AWS. Authentication in aws iot. 2016. URL http://docs.aws.amazon.com/iot/latest/developerguide/identity-in-iot.html. Visitado el 09-05-2016.

[50] Python Software Foundation. ssl — tls/ssl wrapper for socket objects. 2016. URL https://docs.python.org/2/library/ssl.html. Visitado el 09-05-2016.

[51] Sakis Kasampalis. *Mastering Python Design Patterns*. Packt Publishing, 2015.

[52] Berkeley's OpenWSN. A coap python library. 2016. URL https://github.com/openwsn-berkeley/coap. Visitado el 15-05-2016.

[53] Sensirion. Datasheet sht21. 2016. URL http://www.farnell.com/datasheets/1780639.pdf.

[54] Maxim Integrated. Datasheet max44009. 2016. URL http://datasheets.maximintegrated.com/en/ds/MAX44009.pdf.

[55] Paho. Python client. 2016. URL https://eclipse.org/paho/clients/python/. Visitado el 15-05-2016.

[56] Clark C. Evans. Yaml: Yaml ain't markup language. 2016. URL http://yaml.org/. Visitado el 09-05-2016.

[57] Codacy. Codacy. 2016. URL https://www.codacy.com. Visitado el 1205-2016.

[58] Wikipedia. Complejidad ciclomática. 2016. URL https://es.wikipedia.org/wiki/Complejidad_ciclomatica. Visitado el 12-05-2016.

[59] AWS. Aws command line interface. 2016. URL http://docs.aws.amazon.com/es_es/cli/latest/userguide/installing.html#install-bundle-other-os. Visitado el 15-05-2016.

[60] Re:dash. Re:dash. 2016. URL http://redash.io/. Visitado el 15-05-2016.

[61] freeboard. Visualize the internet of things. 2016. URL https://freeboard.io/. Visitado el 15-05-2016.

[62] Prashanth Holenarsipur y Arpit Mehta. Ambient-light sensing optimizes visibility and battery life of portable displays. 2011. URL http://www.digikey.com/en/articles/techzone/2011/may/ambient-light-sensing-optimizes-visibility-and-battery-life-of-portable-display Visitado el 24-05-2016.

[63] Amazon Web Services. Aws iot limits. 2016. URL http://docs.aws.amazon.com/es_es/iot/latest/developerguide/iot-limits.html. Visitado el 24-05-2016.

[64] S. S. Stevens. To honor fechner and repeal his law. *Science*, 133(3446):80–86, 1961. URL http://sonify.psych.gatech.edu/~walkerb/classes/perception/readings/Stevens1961.pdf.

[65] Wikipedia. Lossy compression. 2016. URL https://en.wikipedia.org/wiki/Lossy_compression. Visitado el 05-06-2016.

[66] Sandy MacDonald. Raspberry pi 3 - first look. 2016. URL http://blog.pimoroni.com/raspberry-pi-3/. Visitado el 05-06-2016.

[67] Arun Vishwanath, Fatemeh Jalali, Kerry Hinton, Tansu Alpcan, Robert W. A. Ayre, y Rodney S. Tucker. Energy consumption comparison of interactive cloud-based and local applications. *IEEE Journal on selected areas in communications*, 33(4):616–626, 2015.