Desarrollo de proyectos IoT utilizando Raspberry Pi como plataforma

D. Miguel Ángel Martínez Sánchez

Desarrollo de proyectos IoT utilizando Raspberry Pi como plataforma

D. Miguel Ángel Martínez Sánchez

Tutorizada por

D. Pedro Sánchez Palma
D. Diego Fernández Álvarez

Índice general

Resumen

El objetivo de este proyecto es desarrollar casos canónicos de mini-proyectos de Internet de las Cosas utilizando la plataforma Raspberry pi y hacer una propuesta de despliegue para un caso de estudio agronómico inicial. Para realizar la propuesta se ha procedido al estudio de diferentes plataformas-middleware IoT. El estudio se ha realizado tanto desde un punto de vista teórico como desde un punto de vista práctico. Con el análisis realizado en este documento se pretende dar respuesta a qué middleware/plataforma IoT es el más adecuado para el escenario IoT planteado.

1 Introducción

El escenario inicial planteado se muestra en la figura 1.1. Tal y como se describe en [1], la red se compone de tres tipos distintos de nodos, cada uno de ellos con sus propias funciones y posición en la jerarquía: los Nodos Locales en la parte inferior de la jerarquía, los Nodos de Coordinación Locales en la parte intermedia y los Nodos de Coordinación en la parte más alta de la jerarquía.

- Nodos Locales (LNs): Estos nodos físicos son los más básicos y su principal función es la de monitorizar el entorno y reaccionar de acuerdo al estado del mismo. Podrían así mismo intercambiar mensajes con otros nodos o solicitar servicios web, aunque no es su objetivo principal. Están directamente subordinados a los Nodos de Coordinación Locales.
- Nodos de Coordinación Locales (LCNs): Situados en el medio de la jerarquía, la principal misión de estos nodos físicos es la de coordinar las diferentes redes de LNs y actuar como brokers ante agentes y servicios externos. Aunque no son obligadas, también podrían disponer de capacidades de monitorización y actuación.
- Nodos de Coordinación (CNs): Estos nodos, ubicados en la cima de la jerarquía, no disponen de la capacidad para monitorizar y actuar sobre el entorno, y se encargan exclusivamente de coordinar las diferentes redes de LCN, actuar como brokers ante agentes y servicios externos, y conectar con la Nube. Algunos de estos nodos podrían ser virtuales en lugar de físicos en función de los requisitos de la aplicación.

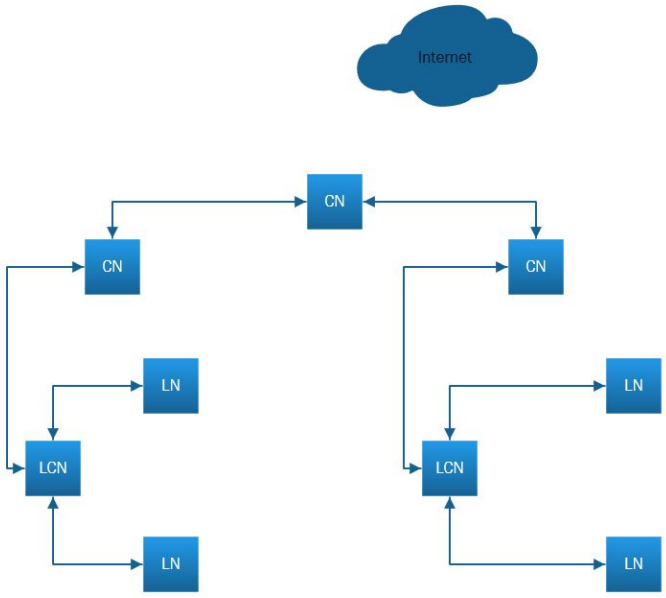
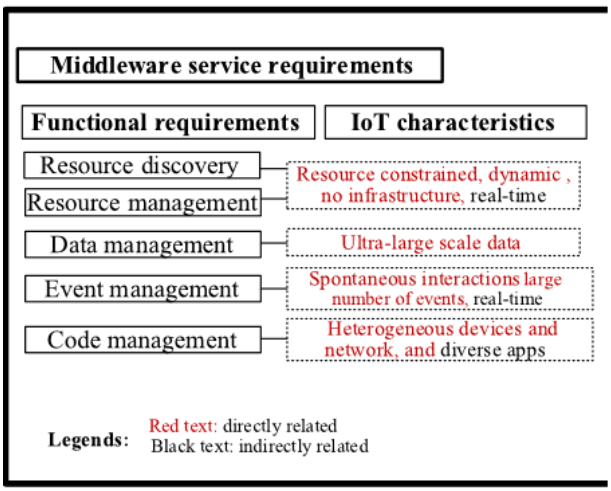
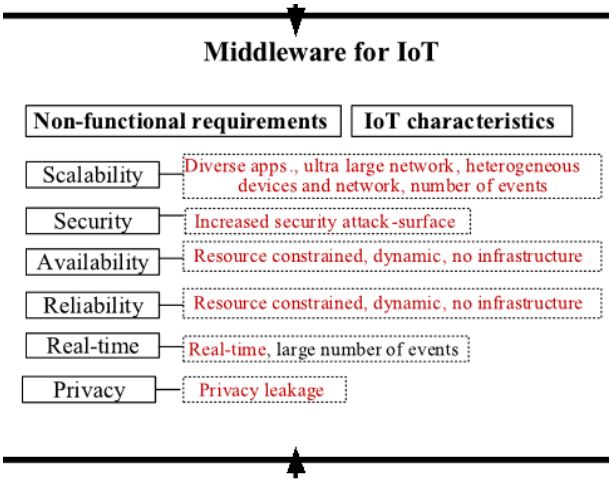


Fig 1.1. Escenario inicial planteado.

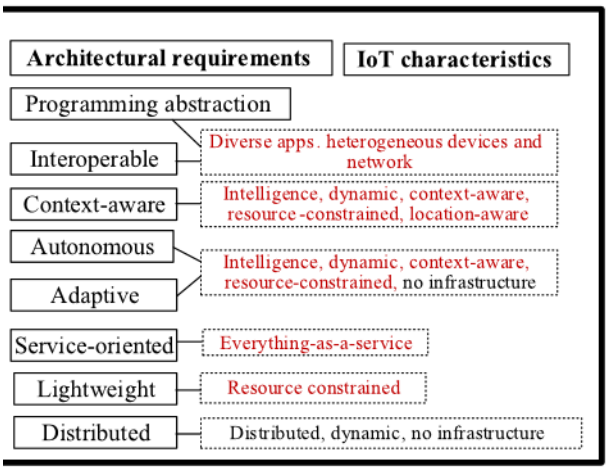
Se ha realizado un detallado análisis de diferentes plataformas middleware open-source para determinar cuál es la más idónea para esta arquitectura. Para la realización de este análisis se han seguido las pautas del artículo "Middleware for Internet of Things" [2] en el que se describe los requisitos funcionales, no funcionales y arquitectónicos que un middleware IoT debe satisfacer. Estos requisitos quedan recogidos en la tabla 1.1.



(a) Requisitos funcionales.



(b) Requisitos no funcionales.



(c) Requisitos arquitectónicos.

Tab 1.1. Requisitos para un middleware IoT.

Se ha investigado qué requisitos son satisfechos por el middleware analizado y, en en el caso de los requisitos no satisfechos, cómo suplir sus carencias.

2 El Internet de las Cosas

El concepto del Internet de las Cosas fue propuesto en 1999 por Kevin Ashton, en el Auto-ID Center del MIT, donde se presentó un sistema de sensores e identificadores de radiofrecuencia (RFID).
El nombre de Internet de las cosas (Internet of Things (IoT), en inglés)[3][4][5] hace hincapié en la interconexión digital de objetos cotidianos con internet.
Un frigorífico que avise de la fecha de caducidad de los alimentos, unas zapatillas de deporte que registren en la nube las estadísticas de la velocidad y distancia a la que el usuario corre, un inodoro que analice la orina para recomendar una dieta u otra, en definitiva, que cualquier cosa u objeto tenga conexión a Internet.
Según la empresa Gartner [6], en 2020 habrá en el mundo más de 20 mil millones de dispositivos con un sistema de conexión al internet de las cosas. Los mencionados objetos se valen por un sistema embebido que permite dotar de inteligencia al objeto conectado, ayudándose de sensores para captar información; éste es uno de los conceptos que acompañan al IoT, el denominado Big Data: datos, datos y más datos, estructurados o no estructurados, para ser analizados. El IoT puede controlar desde aplicaciones domésticas como las mencionadas anteriormente, a aplicaciones profesionales, como por ejemplo, un cultivo en el que cada planta tenga su propio sensor y el agricultor sea capaz en todo momento de recibir información sobre ellas conociendo cómo crecen y si existen problemas en algunas de ellas.
El coste que podría tener monitorizar el sistema anterior, en el que cada planta tiene un sensor enviando información a ciertos nodos para su procesamiento, podría ser elevado. Por ello se busca hardware de bajo coste, cuyas capacidades se ven potenciadas gracias a la conexión con Internet.

Fig 2.1. Internet de las cosas.

2.1 Barreras en el crecimiento del IoT

Existen algunos inconvenientes que impiden el avance del IoT. Uno de ellos es el problema de direccionamiento, bien es cierto que ya existe una propuesta firme para cambiar a IPV6. Otro de los principales problemas que impiden el crecimiento del IoT es la seguridad y privacidad. El hecho de compartir información privada en la nube nos hace mucho más vulnerables a los ataques por parte de hackers. Existen casos preocupantes que hacen que la interconexión de todo con todo aún tenga que esperar.

En 2013, Hugo Teso, un experto en seguridad, demostró, en una maqueta, que era posible tomar el control de un avión desde un dispositivo Android (entre otras cosas), haciéndose valer de información compartida por los aviones para determinar su posición.

En 2015 un grupo de investigadores descubrió que era relativamente sencillo manipular remotamente la aceleración y el sistema de frenado en los coches autónomos. Ejemplos que nos hacen pensar en la importancia de un sistema de seguridad en IoT y por qué aún no ha dado ese paso definitivo para su expansión.

Por otro lado, la ausencia de un estándar común para facilitar la interoperabilidad entre distintas aplicaciones y dispositivos es otro de los obstáculos que impiden la expansión del IoT. Estos problemas será cuestión de tiempo que se solucionen para que IoT acabe siendo una realidad [7].

2.2 Características

IoT no es solo una tecnología, es un concepto que integra varias tecnologías: Redes de Sensores Inalámbricas, RFID, Cloud, Edge y Fog Computing, Big Data, etc. Los dispositivos IoT cuentan con sensores y actuadores que ayudan a interactuar con el medio físico. Los datos recopilados por los sensores, esto es, cualquier dispositivo capaz de proporcionar información sobre el medio físico, se tienen que almacenar y procesar. Esta información se suele enviar a un servidor para que la procese.

2.2.1 Arquitectura del IoT

Existe un gran conjunto de tecnologías de comunicación que necesitan ser adaptadas para las aplicaciones IoT, como la eficiencia energética, seguridad o confidencialidad. Al contrario que ocurre con TCP/IP, en IoT no hay un consenso en una única arquitectura. Se han propuesto varias arquitecturas. En la figura 2.2 se puede ver una arquitectura de IoT.

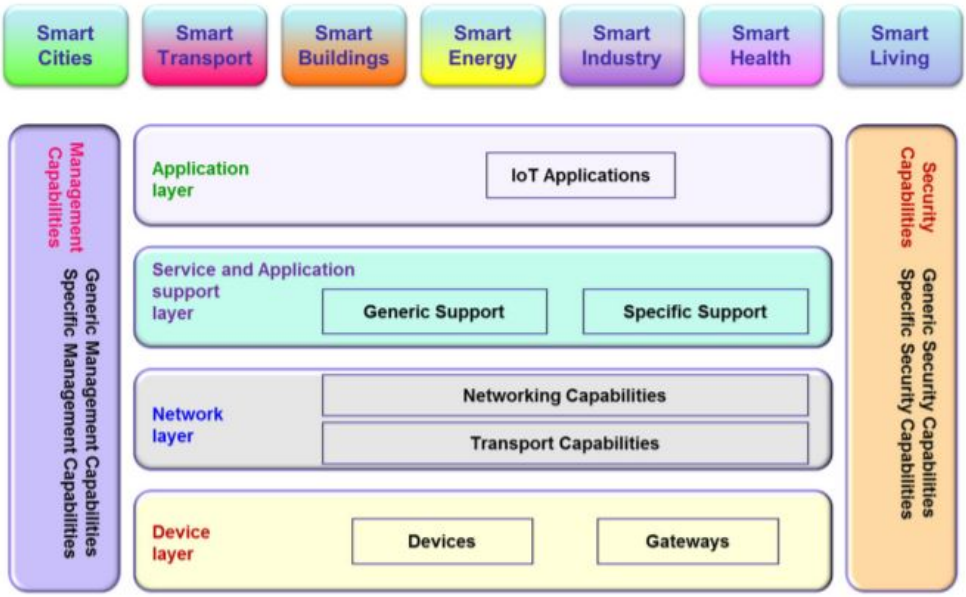


Fig 2.2. Arquitectura de IoT. [4]

Con IoT se pasa del modelo tradicional y centralizado a un modelo distribuido y de interconexión de todo con todo.

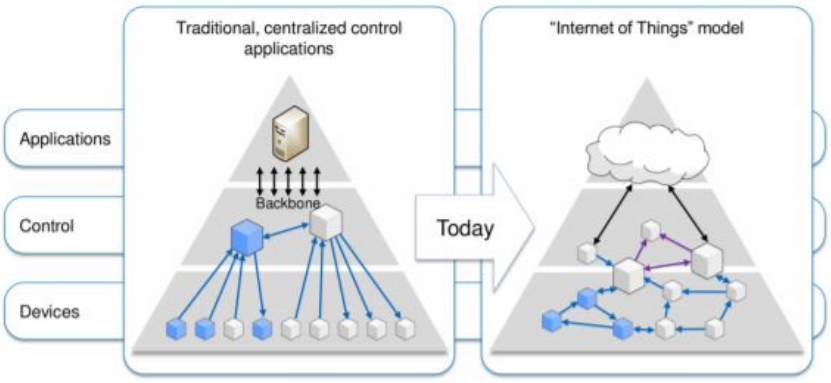


Fig 2.3. Modelo de IoT. [4]

2.2.2 Áreas de aplicación

Las aplicaciones de IoT son muchas y muy diversas. El IERC (European Research Cluster on the Internet of Things) ha identificado las principales áreas de aplicación en IoT: smart energy, smart heatlh, smart buildings, smart transport, smart industry, smart city and smart farming. Esto es: energía, salud, construcción, transporte, industria, ciudades inteligentes y agricultura inteligente.

Smart cities

Es un concepto emergente que aún no tiene una definición aceptada. Su objetivo es integrar el uso de tecnologías para proporcionar servicios a sus ciudadanos y mejorar su estancia en las ciudades. Sus aplicaciones van desde el Smart Parking, indicando al conductor dónde hay un hueco libre para poder aparcar, el Smart Traffic indicando en tiempo real el tráfico en la ciudad y rutas alternativas, pasando por una gestión eficiente del alumbrado de la ciudad.
Esto son solo unos ejemplos de lo que se pretende implementar con esta tecnología, su futuro es muy prometedor sobre todo viendo como muchas ciudades y empresas están apostando claramente por convertir las ciudades en un ecosistema más inteligente y conectado.

Fig 2.4. Smart City.

Smart Transport

La conexión de los vehículos a internet hará el trasporte más fácil y seguro. Sus objetivos son, entre otros, reducir el consumo y emisiones de vehículos, proporcionar soluciones de parking inteligentes que identifiquen los espacios disponibles y se lo hagan saber al usuario, mejorar la seguridad en motoristas, viandantes y ciclistas o reducir la congestión del tráfico y actuar, por ejemplo, con el control de semáforos.

Smart Home

La inteligencia en las casas hará la vida diaria más cómoda y sencilla. Poder controlar las tareas de casa desde cualquier punto es el objetivo principal de un sistema Smart Home. Éste es, posiblemente, uno de los sectores de mayor crecimiento en IoT en los últimos años.

Fig 2.5. Smart Home.

Smart Health

La salud es otra de las posibles aplicaciones que puede tener IoT. Los sistemas Smart Health proporcionan servicios relacionados con la salud usando la red como conexiones entre agentes. Estos agentes pueden ser teléfonos móviles, pulseras de actividad deportiva, dispositivos de medición de la actividad cerebral... Entre sus objetivos está conectar múltiples agentes, recopilar diferentes datos y tomar acciones. Por ejemplo, un monitor de bebés que mida ciertos parámetros que puedan avisar si ha dejado de respirar durante la noche.

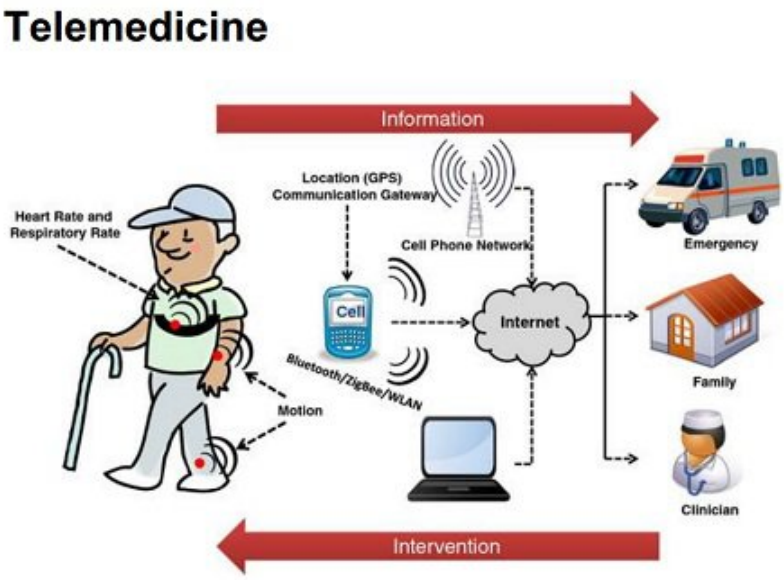


Fig 2.6. Smart Health.

Smart Farming

A pesar de que le está costando arrancar en este medio, probablemente por ser un área en el que la evolución tecnológica nunca ha sido una prioridad, se está intentando revolucionar la forma en la que trabajan los agricultores.
Sus aplicaciones van desde monitorizar los cultivos, herramientas de soporte para la toma de decisiones, controlar automáticamente riego, protección de heladas, fertilización, etc. La agricultura inteligente se convertirá en el campo de aplicación más importante en los países predominantemente agrícolas.

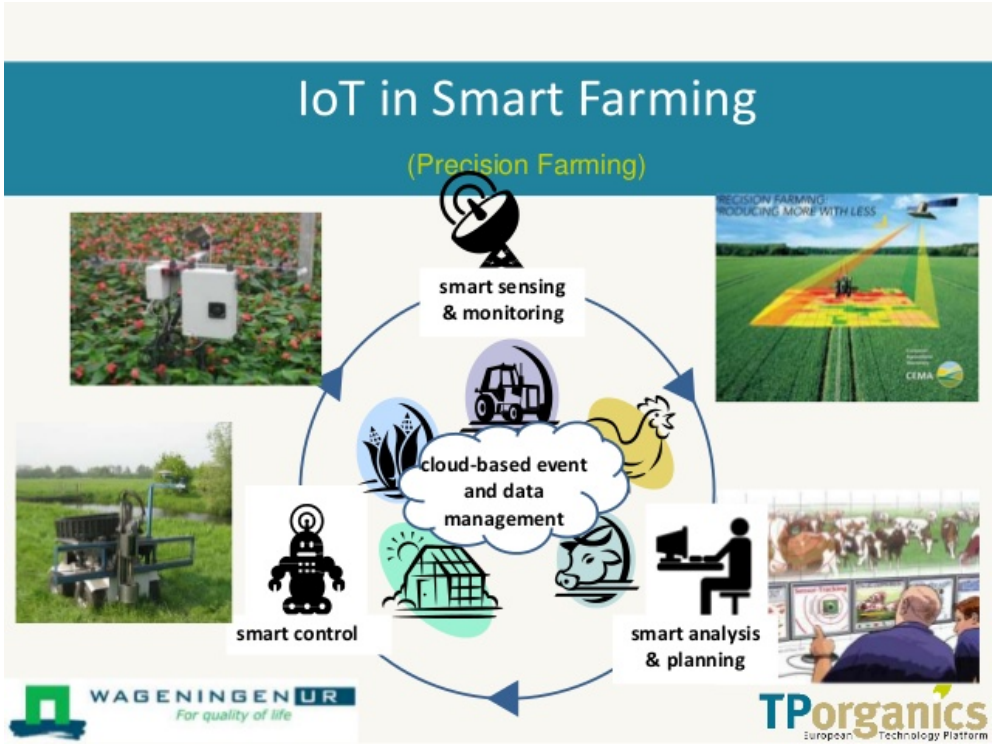


Fig 2.7. Smart Farming.

2.3 Conclusiones

El IoT se presenta como un nuevo paradigma que parece predestinado a cambiar por completo el escenario socioeconómico tal y como lo conocemos hoy en día, pues el hecho de que cualquier objeto pueda ser monitorizado nos ayudaría a administrar mejor los recursos. Estas nuevas tecnologías están permitiendo la obtención de mayores cantidades de información e incluso la obtención de datos sobre elementos de los que antes ni tan siquiera se podía pensar que fuera posible o sobre los que se pudiera influir.

Aún con el gran abanico de posibilidades que se abre gracias a la utilización de estas nuevas tecnologías, este nuevo paradigma no se encuentra exento de problemas. Uno de ellos, posiblemente el más grave, lo constituye su seguridad, en sus múltiples vertientes. Así, según los expertos, la seguridad es un asunto de especial trascendencia, ya que la mayoría de las medidas de seguridad tradicionales no pueden aplicarse al IoT debido principalmente a las restricciones y/o características que comporta su diseño: bajo procesamiento, entornos propensos a ataques malintencionados, etc.

Los estudios relacionados con el Internet de las Cosas están todavía en un punto muy temprano de desarrollo. Como resultado, carecemos de una definición estandarizada para este término.

3 Middleware en IoT

3.1 Introducción

Como se ha visto, la esencia del IoT es conectar y comunicar cualquier cosa mediante internet. Conseguir interconectar millones de dispositivos diferentes, trabajando a niveles de red diferentes y generando una gran cantidad de eventos no es una tarea sencilla. Un middleware IoT facilita la integración entre la comunicación de los dispositivos, además de soportar una interoperabilidad dentro de los diferentes servicios y aplicaciones.
El desarrollo de middleware en IoT es un área activa de investigación. Las líneas de investigación en middleware han ido por las redes de sensores inalámbricas (WSN) , por las comunicaciones máquina a máquina (M2M) o también por etiquetas de identificación de radio frecuencia (RFID), componentes esenciales en el IoT. En este trabajo se hablará de algunos middleware IoT y finalmente qué middleware es el elegido para su análisis.

3.2 CoAP

CoAP [8] es un protocolo que trabaja en el dominio de las redes de sensores (WSN) y comunicación M2M. Estas redes requieren de protocolos de transmisión de datos con un ancho de banda reducido y con un consumo muy bajo de energía. CoAP se ha diseñado como una "traducción" sencilla de HTTP para simplificar la integración con la web con requisitos especiales como baja carga, bajo consumo o simplicidad. CoAP implementa el modelo REST de HTTP funcionando sobre UDP y con mecanismos de seguridad específicos para ser usados en este tipo de redes.

3.2.1 Características CoAP

- Traduce su integración simplificada a HTTP de forma sencilla.

- Intercambio de mensajes asíncrono.
- Baja sobrecarga de cabeceras para reducir la complejidad al analizar el mensaje.

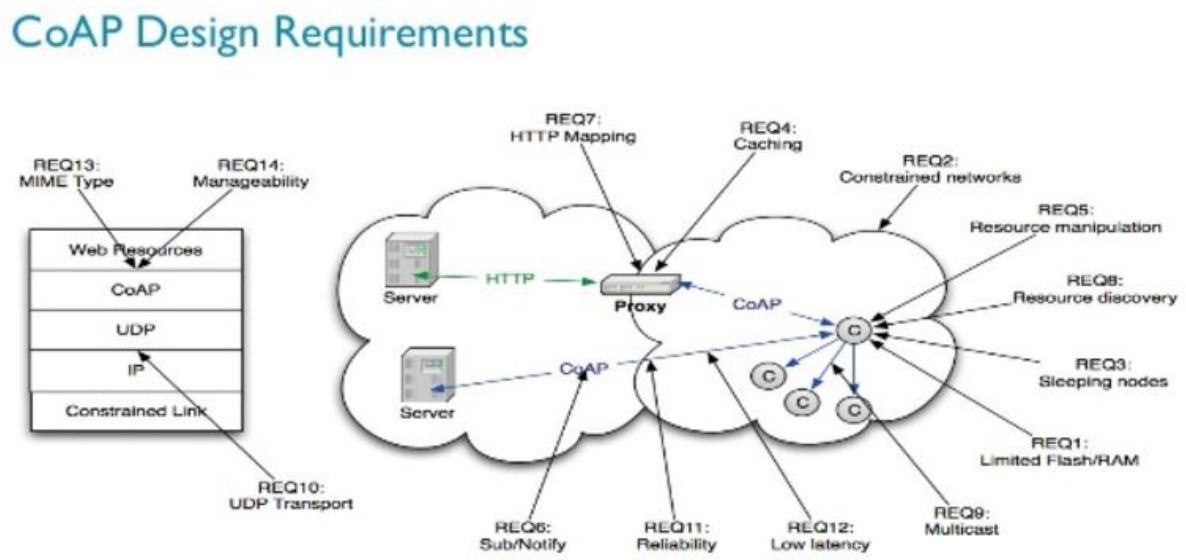


Fig 3.1. Diseño CoAP

CoAP es un protocolo muy usado en IoT, permite descubrimiento de recursos, seguridad, soporte IPv6. En definitiva, un protocolo moderno y con muchos beneficios.

3.3 UbiROAD

UbiROAD [9] es una propuesta de middleware inteligente conocedor del contexto, esto es, capaz de decidir por él mismo ofreciendo al usuario los servicios necesarios. Se usa en vehículos para el control del tráfico inteligente.

Fig 3.2. Middleware UbiROAD.

3.4 Agilla

Agilla [10] es un middleware basado en agentes diseñado para aplicaciones auto-adaptables en redes de sensores. Las redes de sensores están expuestas a un mundo dinámico, muchas de sus aplicaciones tienen que adaptarse a los cambios en el ambiente. Por ejemplo, una red WSN usada por un robot, tiene que adaptarse a los cambios del robot. En definitiva, se necesitan middleware y nuevos modelos de programación para soportar la auto-adaptabilidad de las redes de sensores. Para tal fin, se desarrolló Agilla
La estructura de Agilla está basada en agentes. Un agente no es más que un nodo dotado de inteligencia. Un agente puede clonarse desde un nodo a otro manteniendo su estado y así poder coordinar con otros agentes. La estructura de Agilla se muestra en la figura 3.3. Agilla funciona sobre TinyOS, un sistema operativo para WSN.

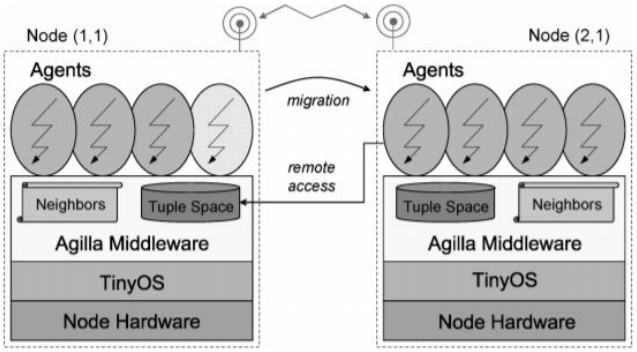


Fig 3.3. Modelo Agilla.

3.5 TinySOA

TinySOA [11] es un middleware usado en redes de sensores. Proporciona un conjunto de servicios Web que facilita el envío y recepción de información por parte de los sensores. La arquitectura se puede ver en la figura 3.4. Está formado por cuatro partes principales: Nodos, gateway, registro y servidor.
El nodo reúne toda la información de un nodo sensor. Tiene algunos servicios como el descubrimiento de recursos o la comunicación con el gateway. El gateway en las redes WSN, tiene la función de conectar la red WSN con la red externa, es decir con Internet. Entre otras funcionalidades, usa la información del nodo sensor para registrarlo en el servicio de registro. Una vez registrados, el gateway se subscribe a los servicios proporcionados por el sensor. El servidor es otra de las partes principales de este middleware, y su función es usar el servicio de registro para establecer un servicio web

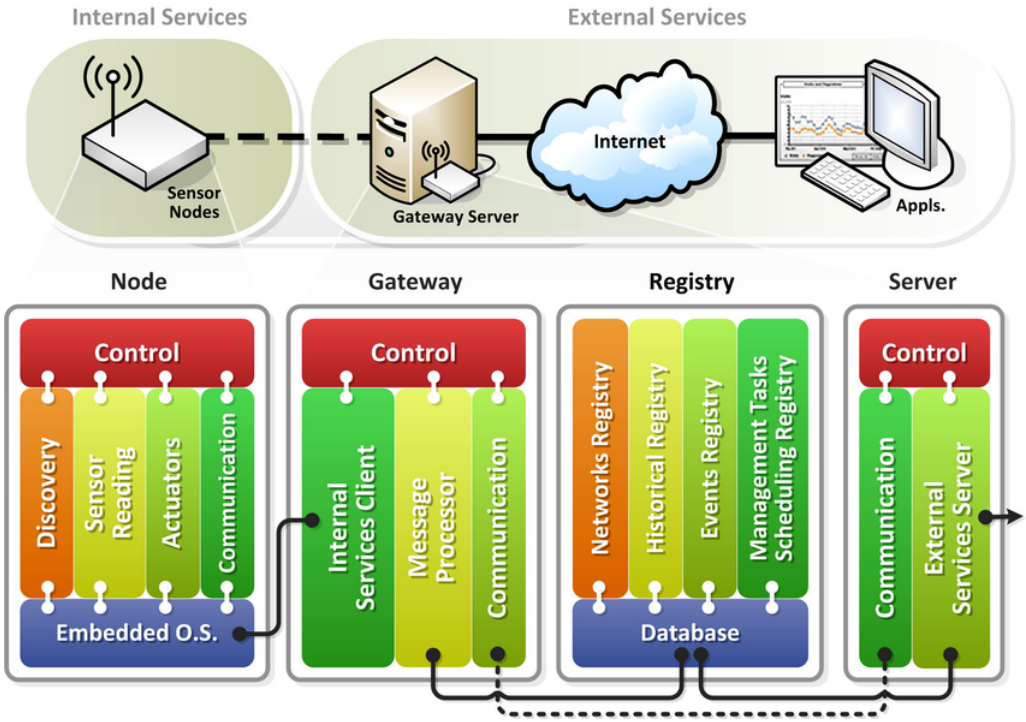


Fig 3.4. Arquitectura de TinySOA.

3.6 MQTT

Message Queue Telemetry Transport (MQTT) [12] es un protocolo de código abierto que se desarrolló y optimizó para dispositivos restringidos y redes de bajo ancho de banda, alta latencia o poco confiables. Es un transporte de mensajería de publicación/suscripción que es extremadamente ligero e ideal para conectar dispositivos pequeños a redes con ancho de banda mínimo. MQTT es eficiente en términos de ancho de banda, independiente de los datos y tiene reconocimiento de sesión continua, al usar TCP. Tiene la finalidad de minimizar los requerimientos de recursos del dispositivo y, a la vez, tratar de asegurar la confiabilidad y cierto grado de seguridad de entrega con calidad del servicio.

3.7 Conclusión

Con la cantidad de dispositivos existentes y diferentes entre ellos, conectarlos sobre internet es una tarea complicada. Por ello, es necesario un software que facilite la interconexión entre ellos, es aquí donde entra en juego el uso de un middleware. Hay cientos de middleware IoT

existentes, cada uno desarrollado para cumplir una cierta función. Elegir uno entre tantos para ser usado en un escenario IoT puede ser una tarea muy tediosa; IoT es un entorno complejo. Tras presentar algunos de ellos, se ha optado por elegir MQTT como protocolo/middleware IoT para ser analizado en mayor profundidad y, en caso de éxito, ser usado en el escenario Smart Farming que planteamos.

4 MQTT

4.1 Introducción

MQTT (Message Queue Telemetry Transport) [12] es un protocolo usado para la comunicación M2M (Machine to machine) en el internet de las cosas. Fue creado por IBM en 1999. Está orientado a las redes de dispositivos pequeños, como sensores, debido a que consume muy poco ancho de banda además de que no requiere de mucho software para implementar un cliente, lo cual es perfecto para dispositivos como Arduino. MQTT se ha convertido en un protocolo muy usado en IoT, aunque también es ideal para aplicaciones móviles por su envío eficiente. Un ejemplo de uso es Facebook Messenger para iPhone y Android.
Se basa en el estandar publish/subscribe usado sobre TCP/IP. Esta comunicación publish/subscribe hace uso de un "broker" o servidor centralizado para la gestión de los paquetes.

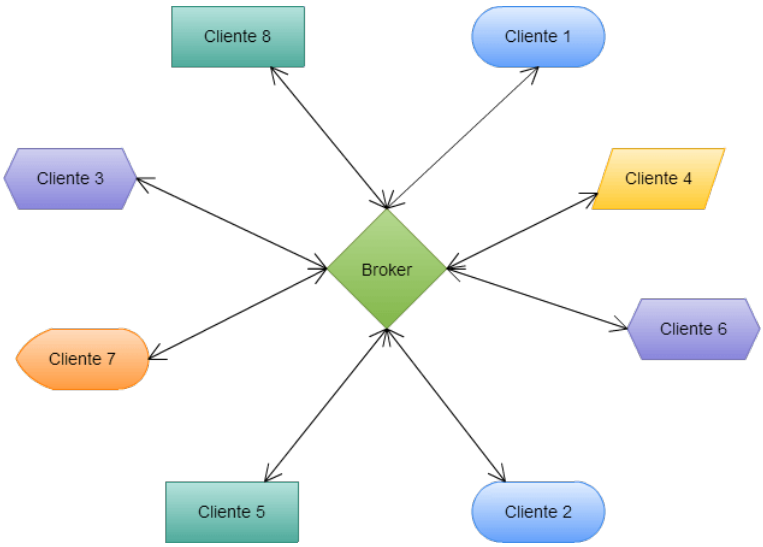


Fig 4.1. Topología MQTT.

La comunicación se basa en "topics" (temas) que el cliente que publica el mensaje envía y los nodos que desean recibirlo deben subscribirse a él. Un topic se representa mediante una cadena de caracteres separada por "/". De esta forma se pueden crear jerarquías como se puede ver en la figura 4.2

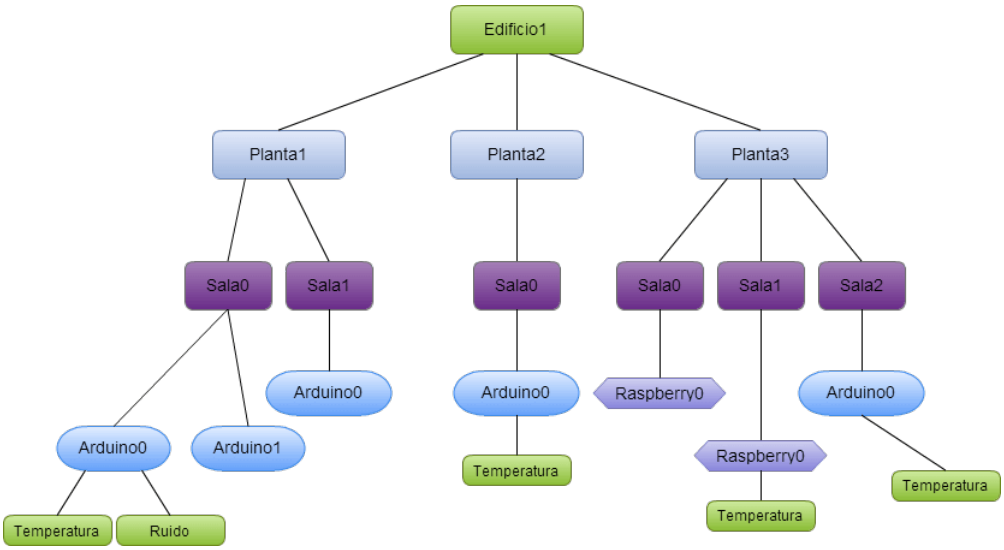


Fig 4.2. Jerarquía Topics.

4.2 Protocolo

MQTT funciona sobre TCP y, como cualquier protocolo, tiene un esquema definido para el envío y recepción de paquetes. El código y tipo de paquete se definen en la siguiente tabla.

Paquete	Enumeración	Descripción
Reservado	0	Reservado
CONNECT	1	Cliente solicita conectarse al servidor
CONNACK	2	Confirmación del mensaje CONNECT
PUBLISH	3	Mensaje PUBLISH
PUBACK	4	Confirmación del paquete PUBLISH
PUBREC	5	PUBLISH recibido ()
PUBREL	6	PUBLISH release ()
PUBCOMP	7	Publish completado
SUBSCRIBE	8	Petición del cliente para subscribirse a un tópico
SUBACK	B	Confirmación del paquete SUBSCRIBE
UNSUBSCRIBE	10	Petición del cliente para desuscribirse
UNSUBACK	11	Confirmación de UNSUBSCRIBE
PINGREQ	12	PING
PINGRESP	13	Respuesta PING
DISCONNECT	14	Cliente se va a desconectar
Reservado	15	Reservado

Tabla 4.1. Enumeración de paquetes

Una vez realizada la conexión a nivel de red entre cliente y broker mediante el envío de los paquetes CONNECT y CONNACK, el cliente se subscribe a un topic. El flujo de datos es el que se muestra en la figura 4.3. El cliente envía un paquete "SUBSCRIBE" con el que el cliente le dice a qué topics quiere subscribirse. Cada suscripción será confirmada por el broker mediante el envío de un paquete "SUBPACK". El broker envía en el paquete un código de respuesta por cada topic al que el cliente se ha suscrito, en el mismo orden que el cliente lo envió. El código de respuestas se puede ver en la tabla 4.2. Una vez recibida la confirmación, el cliente puede empezar a enviar mensajes.

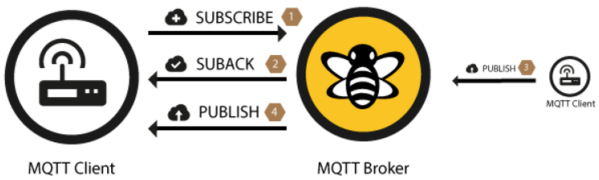


Fig 4.3. Envío de paquetes en MQTT.[13]

Return Code	Return Code response
0	Success – Maximum QoS 0
1	Success – Maximum QoS 1
2	Success – Maximum QoS 2
128	Failure

Tabla 4.2. Código de respuesta

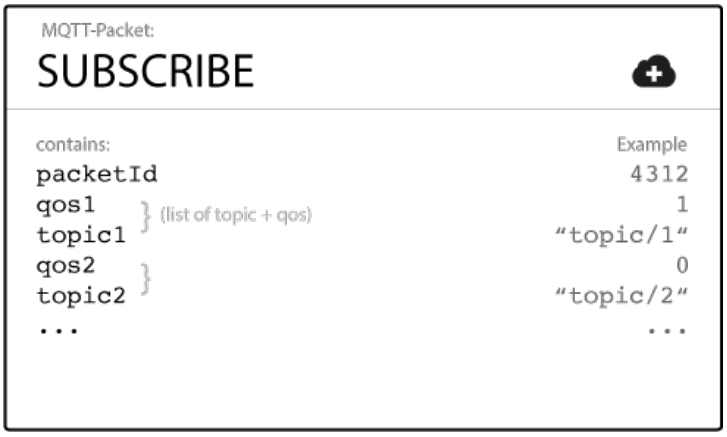


Fig 4.4. Paquete SUBSCRIBE.[13]

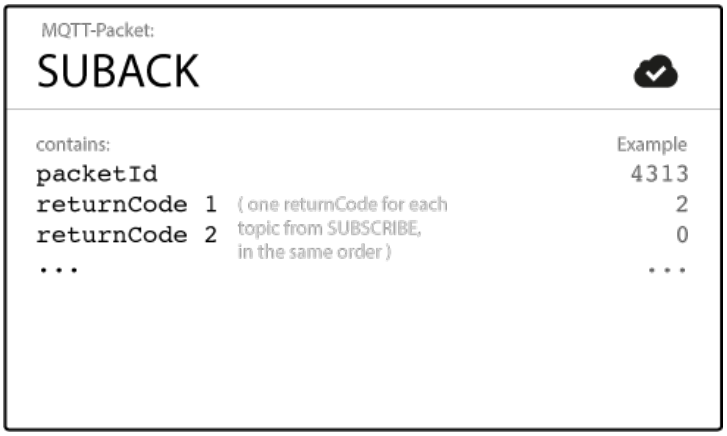


Fig 4.5. Paquete SUBACK.[13]

Por otro lado, si el cliente quiere desuscribirse a un topic deberá enviar un paquete "UNSUBSCRIBE" y esperar la confirmación por parte del broker mediante el paquete "UNSUBACK"

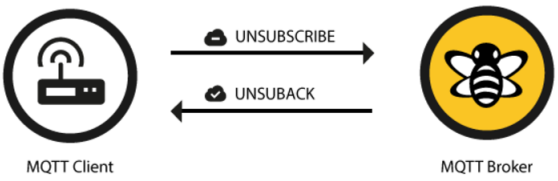


Fig 4.6. Envío de paquetes para desuscribirse de un topic.[13]

Fig 4.7. Paquete UNSUBSCRIBE.[13]

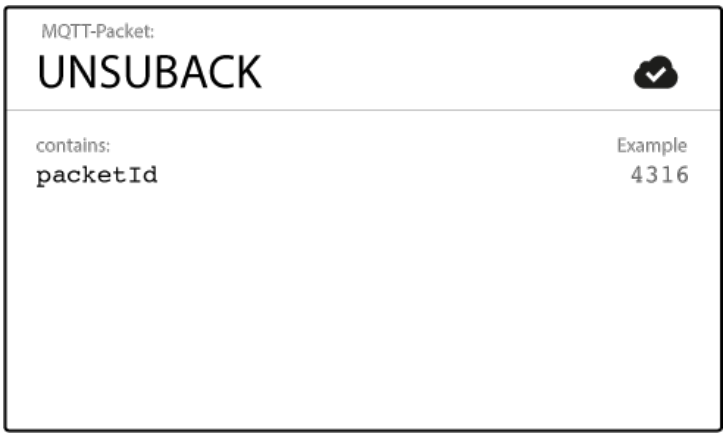


Fig 4.8. Paquete UNSUBACK.[13]

Brokers

Es el servidor de MQTT. El broker recibe los mensajes por parte de los clientes y se encarga de procesar y enviar el mensaje a los subscriptores de ese topic. La gestión de la red es otra

de las funcionalidades del broker. Los clientes envían periódicamente un paquete (PINGREQ) y esperan la respuesta del broker (PINGRESP) para mantener activo el canal.
Existen varias empresas como HiveMQ [13][14] que comercializan un broker MQTT diseñado por ellos mismos. Otro broker comercial es CloudMQTT [15], propiedad de Amazon. En este documento se trabajará con el broker Mosquitto [16], el cual es de código abierto y es uno de los más usados y completos que existen. En la siguiente tabla se muestra una comparación entre alguno de los brokers más usados.

Broker	QoS0	QoS1	QoS2	Autenticación	Bridge	SSL	Clustering	WebSocket
Mosquitto	✓	✓	✓	✓	✓	✓	✗	✓
Mosca	✓	✓	✗	✓	✗	✓	✗	✓
HiveMQ	✓	✓	✓	✓	✓	✓	✓	✓
ActiveMQ	✓	✓	✓	✓	✗	✓	✓	✓
VerneMQ	✓	✓	✓	✓	✓	✓	✓	✓
JoramMQ	✓	✓	✓	✓	✓	✓	✓	✓

Tabla 4.3. Comparación entre brokers MQTT

MQTT Websocket surge para usar MQTT sobre HTTP y poder usar un cliente MQTT en un navegador. Esto es útil cuando se quiera enviar los datos a dispositivos situados en el exterior de la red, como por ejemplo servidores. Estos dispositivos tendrán que tener un nivel de procesamiento más elevado que los de la red interna, pues HTTP necesita de dispositivos más potentes.

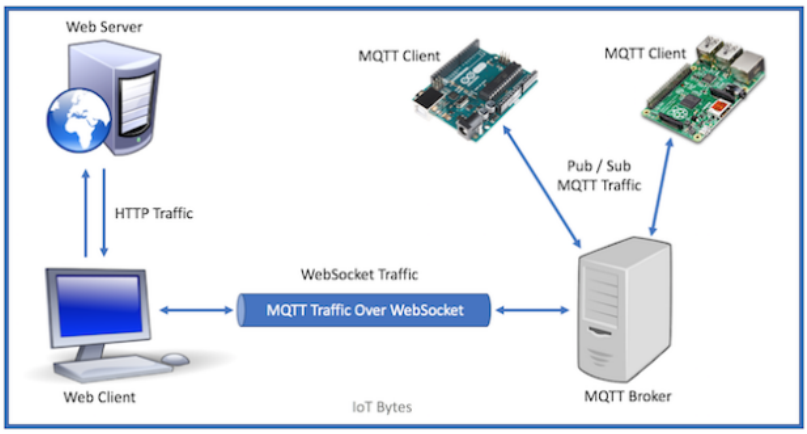


Fig 4.9. MQTT sobre WebSockets

Clientes

Se encargan de enviar al broker la información recibida, normalmente por parte de sensores. Los clientes también reciben mensajes de los topics subscritos. Al igual que los brokers, hay muchos tipos de clientes MQTT. El cliente Open-Source más usado es Eclipse Paho Java Client [17]

4.3 MQTT-SN

4.3.1 Redes de sensores inalámbricas

Las redes de sensores o Wireless Sensor Networks (WSN) en inglés, se han incrementado en los últimos años. Estas redes tienen diferentes aplicaciones, como puede ser la vigilancia y seguridad, medicina, domótica o aplicaciones militares. Una red de sensores está formada por sensores y gateway para conectar con una red de datos. En estas redes es importante la comunicación de forma inalámbrica entre sensores, puesto que el número de nodos (sensores, actuadores...) es muy grande, y una infraestructura cableada tendría un coste muy elevado. Las características de estas redes son:

- Tolerancia a fallos
- Coste
- Ausencia de infraestructura de red
- Bajo consumo

A diferencia de las redes convencionales, los nodos no tienen conocimiento de la topología de la red, por lo que el enrutamiento cambia con respecto a éstas. Aquí es el nodo el que se informa de los nuevos nodos a su alcance y de la manera de encaminarse hacia ellos. En la figura 4.10 se puede ver la estructura de una red de este tipo.

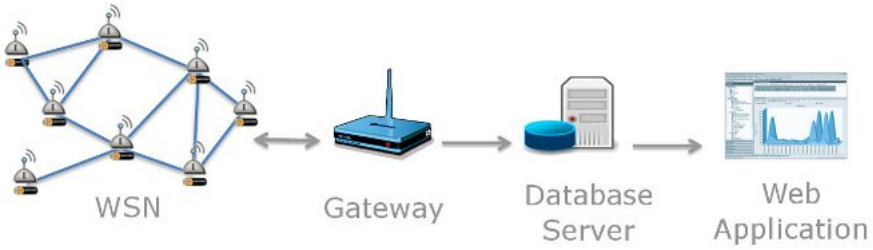


Fig 4.10. Arquitectura de las redes WSN

4.3.2 MQTT-SN

MQTT es un protocolo usado sobre redes donde el ancho de banda es limitado. Sin embargo, MQTT requiere del protocolo TCP/IP, útil para redes con dispositivos de aceptable procesamiento, puesto que ofrece una entrega correcta de los paquetes pero es demasiado complejo como para ser usado en redes de sensores, en las que los dispositivos son de poca memoria y de bajo procesamiento y batería.
Es por ello que surge MQTT-SN, un protocolo pub/sub específico para redes de sensores.

4.3.3 Arquitectura MQTT-SN

La arquitectura MQTT-SN se muestra en la figura 4.11

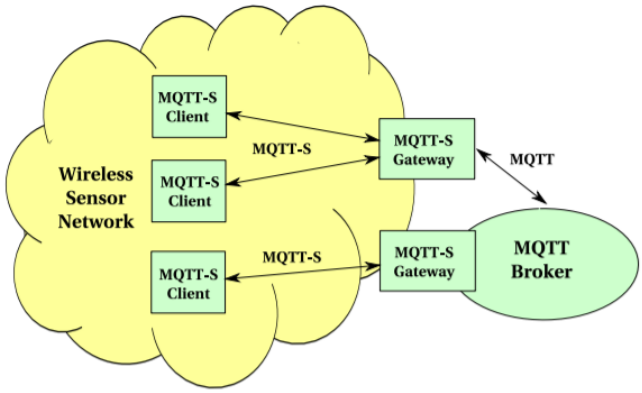


Fig 4.11. Arquitectura MQTT-SN

Hay 2 componentes: MQTT-SN clientes y MQTT-SN gateways. Los MQTT-SN clientes son

los nodos en la red WSN. La comunicación entre clientes se hace a través del Gateway mediante el protocolo MQTT-SN. Los clientes realizan la comunicación pub/sub con un broker, localizado en una red tradicional, a través del Gateway. La comunicación entre el broker y el MQTT-Gateway se hace mediante el protocolo MQTT.

4.3.4 MQTT vs MQTT-SN

MQTT-SN se diferencia en MQTT en varios aspectos. A nivel de red, no funciona sobre TCP pero necesita de Gateways para su funcionamiento. MQTT-SN soporta ID de topics en lugar de nombres, siempre es menos costoso enviar el ID en cada mensaje que no el nombre completo ("home/livingroom/socket2/meter").
El descubrimiento es una de las mayores ventajas de MQTT-SN. Los clientes no necesitan saber la dirección IP o DNS del broker. El anunciamiento ayudará a descubrir los nodos. Los gateways de la red envían paquetes cada cierto periodo de tiempo advirtiendo de su presencia. Los clientes tienen que guardar una lista de los gateways activos junto con su dirección de red. Esta lista se forma a partir de los paquetes ADVERTISE y GWINFO enviados por los gateways. El cliente puede usar esta lista para conocer la disponibilidad de los gateways. Por ejemplo, si no recibe el mensaje ADVERTISE durante un tiempo determinado puede considerar que está caído y actualizar la lista. Conociendo la dirección de red del gateway el cliente puede conectarse mediante el envío de paquetes, como se hace en MQTT, aunque en este caso el paquete CONNECT, se divide en varios paquetes.

4.4 Requisitos

MQTT como middleware IoT, necesita cumplir una serie de requisitos. En el escenario que se plantea hay algunos que no son indispensables y otro que sí lo son, como es la seguridad. Aquí se definen algunos de los requisitos.

4.4.1 Requisitos funcionales

Descubrimiento de recursos

En MQTT no hay un mecanismo para el descubrimiento de recursos. El cliente se tiene que conectar al broker para su comunicación. Con MQTT-SN sí que hay un mecanismo de descubrimiento de nuevos dispositivos.

Control de recursos

MQTT realiza un control básico a nivel de red. Si el broker percibe (mediante el temporizador "keep alive" en el envío de PINGREQ) una desconexión inesperada del cliente, el broker publica un mensaje "Last Will and Testament" (LWT) a todos los subscriptores de ese topic. Los clientes cuando se conectan al broker, tienen la opción de definir este mensaje y pedir al broker que lo almacene.
Existen herramientas que realizan un control y monitorización más avanzados como MQTTSpy. MQTTSpy se ejecuta sobre Java y permite obtener estadísticas sobre los tópicos.

Control de datos

MQTT proporciona un control de datos como son el almacenamiento o el filtrado. De todo ello se encarga el broker. El broker puede almacenar los paquetes de un tópico mientras el cliente, subscrito a dicho tópico, esté offline. El broker también puede aplicar un filtrado en los tópicos, para que se aplique a un nivel o a todos los niveles de ese tópico:

- **Single level : +**
  Para que el mensaje llegue a un cliente, éste debe de estar subscrito a un tópico en el que solamente cambie el nivel indicado por el signo +. En el ejemplo de la figura 4.12, el mensaje destinado a **myhome/groundfloor/livingroom/temperature** o **myhome/groundfloor/kitchen/temperature** llegaría al cliente. Sin embargo, un mensaje destinado a **myhome/groundfloor/kitchen/brightness** o **myhome/firstfloor/kitchen/temperature** no llegaría al cliente.

Fig 4.12. Single level

- **Multi level: #**

Fig 4.13. Multi level

En este caso, llegaría a cualquier cliente que esté subscrito a partir de ese nivel.
Los tópicos que empiecen por el símbolo $ están reservados para las estadísticas internas del broker MQTT. Un ejemplo puede ser:

- $SYS/broker/clients/connected
- $SYS/broker/clients/disconnected
- $SYS/broker/clients/total
- $SYS/broker/messages/sent

4.4.2 Requisitos no funcionales

Escalabilidad

MQTT tiene, en general, una alta escalabilidad, que va a depender en gran medida de la calidad de los nodos y de la topología que se use. La existencia de uno o más brokers dota al sistema de gran escalabilidad ya que la solución puede crecer solo con aumentar recursos en un único elemento. Para añadir un nuevo cliente, tan solo es necesario suscribirse a un tópico.

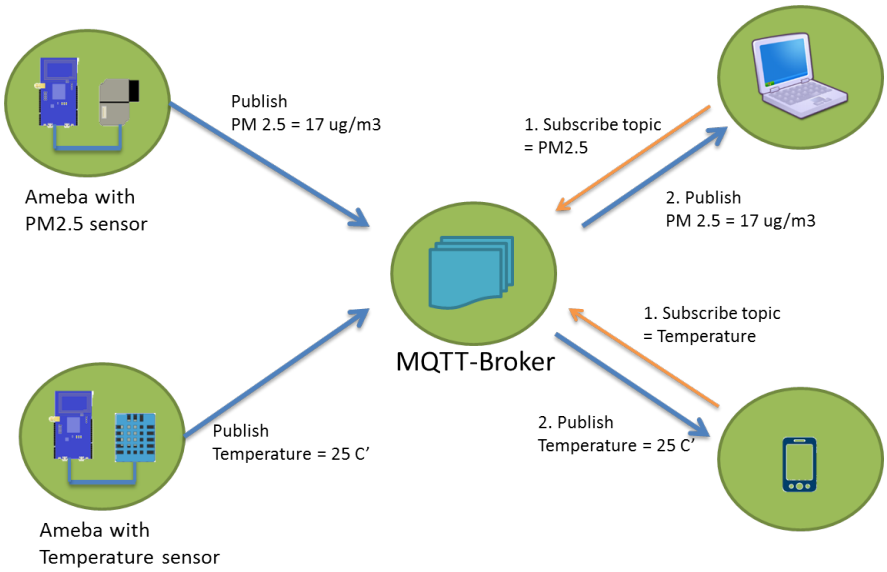


Fig 4.14. Escalabilidad MQTT

Seguridad y privacidad

La seguridad es vital en un entorno IoT y para ello MQTT usa TLS junto con una autenticación de usuario/contraseña. Usa una autenticación usuario/contraseña a nivel de aplicación para establecer la conexión entre cliente y broker. Si TLS se resuelve correctamente, entra en juego dicha autenticación. Además, también existen políticas de autorización para denegar un determinado topic o para permitir una operación

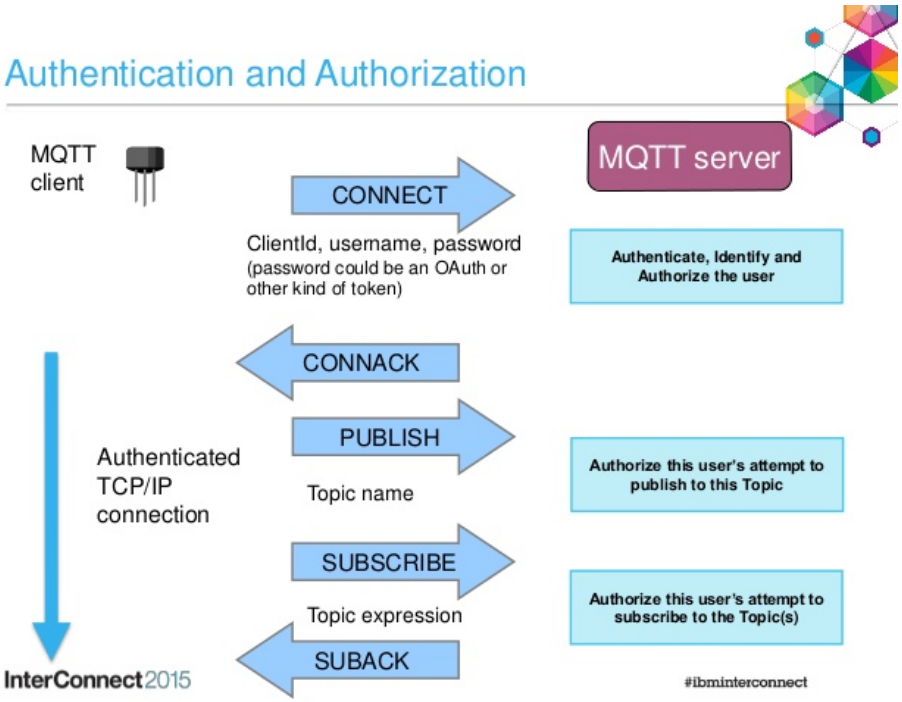


Fig 4.15. Autenticación MQTT

Disponibilidad

Debido a su arquitectura "brokerizada", los sistemas MQTT tiene un único punto de fallo: el broker. La disponibilidad del sistema es baja al basarse únicamente en la disponibilidad del broker.
Con herramientas como balanceadores de carga, o el uso de los brokers en modo “bridge” se puede conseguir una buena disponibilidad. La disponibilidad permitirá poder seguir usando la red en caso de fallo en alguno de sus nodos.
Con el denominado MQTT clustering se incrementa la disponibilidad. MQTT clustering consiste en usar varios brokers (cluster) MQTT como si de uno solo se tratase. Esto se consigue usando brokers en diferentes dispositivos físicos y conectándolos a nivel de red. A vista del cliente MQTT, estos brokers se comportan como un único broker. Se usan balanceadores de carga para tener un único punto de entrada y mejorar así el consumo en la red. En la siguiente imagen se puede ver un ejemplo de clustering con el broker comercial HiveMQ.

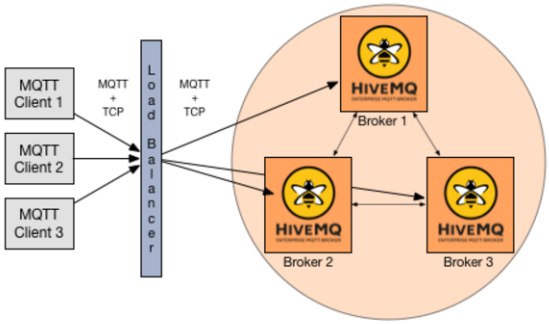


Fig 4.16. MQTT Clustering.[13]

Mosquitto no tiene esta característica pero se podría mejorar la disponibilidad utilizando el broker en modo "bridge". Con el modo "bridge" configurado, se puede enviar información entre los brokers y así eliminar un único punto de fallo, mejorando la disponibilidad.

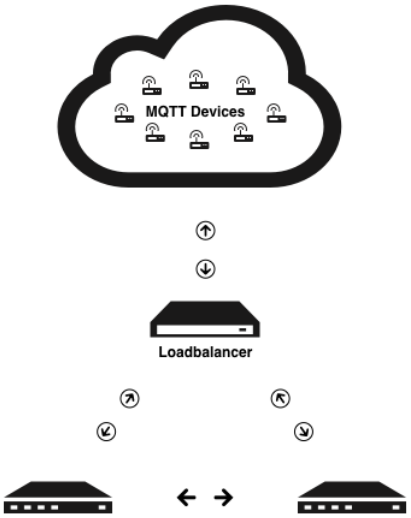


Fig 4.17. MQTT en modo bridge.

Confiabilidad

MQTT proporciona 3 grados de QoS para la entrega de los paquetes:

- **QoS 0: Como mucho una vez**
  El mensaje PUBLISH se envía y el broker no manda ningún reconocimiento. El mensaje llega al broker una vez o ninguna y si llega, llegará a los subscriptores una vez o ninguna.

Fig 4.18. QoS 0.[13]

- **QoS 1: Al menos una vez**
  Este nivel de calidad de servicio asegura que llega al broker al menos una vez. El broker tiene que reconocer la recepción con un paquete PUBACK. Si no recibe un acuse de recibo se vuelve a enviar de nuevo el mensaje PUBLISH con otro identificador distinto hasta que se reciba el reconocimiento. En este nivel no se garantiza que los paquetes no lleguen duplicados.

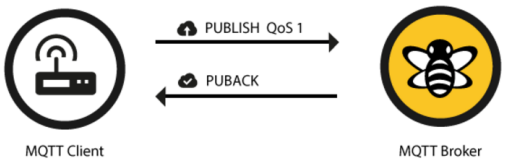


Fig 4.19. QoS 1.[13]

- **QoS 2: Exactamente una vez**
  Este nivel de calidad asegura que el paquete llega una y solo una vez. Para ello se deben enviar además los paquetes PUBREC y PUBREL.

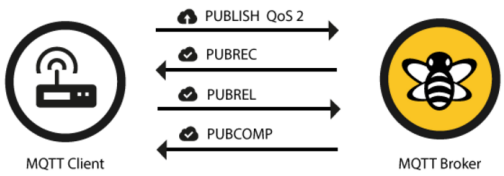


Fig 4.20. QoS 2.[13]

Tiempo real

Si tiempo real lo definimos en nanosegundos, MQTT no cumple este requisito ya que una de las principales características de MQTT es que funciona sobre TCP. En el escenario que se plantea, estos nanosegundos no son necesarios por lo que no es un requisito indispensable.

Fácil de desplegar

MQTT es un protocolo que destaca por su sencillez a la hora de su despliegue.

Popularidad

MQTT es un middleware muy usado debido a su consumo y facilidad de uso. Tiene un gran soporte así como empresas que comercializan sus propios brokers añadiéndole mejoras y usándolo en aplicaciones IoT.

4.4.3 Requisitos arquitectónicos

Interoperabilidad

MQTT es un protocolo heterogéneo en cuanto a dispositivos y tecnologías se refiere. Puede funcionar prácticamente en cualquier dispositivo ya que es un protocolo muy sencillo que requiere poco procesamiento. Puede funcionar en un dispositivo móvil, PC, Raspberry…

Distribuido

Esta característica dependerá del broker a usar, algunos como HiveMQ o JoramMQ, pueden actuar como sistemas distribuidos, en los que se interpreta un cluster de brokers como un único broker. El broker Open-Source Mosquitto no cumple con este requisito.

Ligero

El bajo consumo de MQTT es una de las claves de su uso extendido en IoT. MQTT está diseñado para trabajar con dispositivos de bajo procesamiento como sensores o dispositivos móviles. Además, el consumo de ancho de banda que realiza en la red es mínimo.

4.4.4 Documentación

MQTT es un middleware de código abierto perfectamente documentado. Es sencillo de usar y hay una gran cantidad de ejemplos para facilitar su despliegue.

4.5 Cloud Computing

En IoT se busca un procesamiento en la nube. MQTT nos ofrece el envío de datos y pequeños controles sobre los mismos pero para obtener una solución IoT completa se necesita de servidores y bases de datos. Algunos de los escenarios típicos con MQTT hacen uso de herramientas como Node-RED[18] junto con MongoDB [19]. MQTT se encargaría del envío de datos (como se ha explicado más arriba) y éstos se enviarían al servidor, donde Node-RED recopilaría esos datos y los almacenaría en MongoDB. Se pueden usar herramientas como Google Chart para analizar los datos y poder llevar un control en tiempo real sobre los mismos.

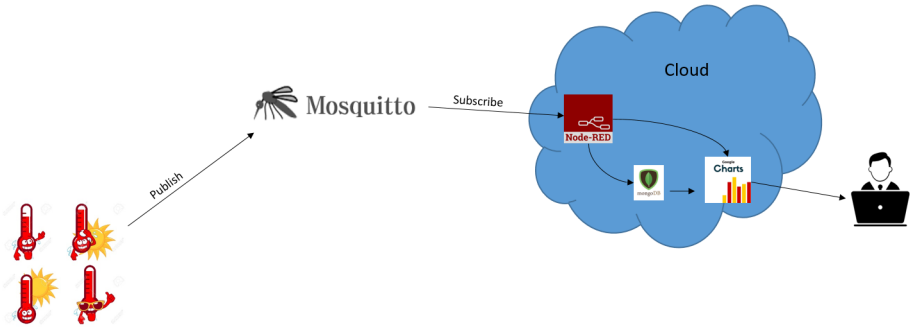


Fig 4.21. Cloud Computing haciendo uso de MQTT

4.6 Conclusiones

MQTT como middleware IoT, puede dar solución a múltiples problemas. MQTT cumple con lo que buscamos en cuanto a requisitos se refiere pero no nos ofrece una solución completa. Por ello hemos optado por buscar plataformas-middleware Open-Source que nos ofrezca una solución más completa. Es decir, un middleware que unifique almacenamiento en la nube, procesamiento y análisis de datos, entre otros.

5 Plataformas IoT

5.1 Introducción

Hoy en día, se ha incrementado el número de soluciones middleware para IoT que proporcionan una interconexión entre los dispositivos y los datos. Estas soluciones middleware se aplican a escenarios en concreto, como el caso de MQTT que se aplica a nivel de transporte para facilitar la comunicación M2M o, el caso de Agilla para redes de sensores en las que se busca un ambiente de auto-adaptación. Es decir, no existe un middleware que ofrezca una solución completa, desde el control de dispositivos y la recolección de datos hasta el procesamiento y visualización de los mismos. Es aquí donde surge la idea de una plataforma-middleware IoT [20] que integre esos servicios. Este panorama, como cualquier concepto de IoT, inexistente hace unos años, es complejo y cambiante de manera rápida. Estas plataformas residen en la nube y permiten al usuario reunir los datos de sensores y actuadores y procesarlos, bien para almacenarlos, bien para visualizarlos o bien para analizarlos y extraer información útil de ellos.

Fig 5.1. Plataformas IoT

5.2 Características

Las plataformas IoT son componentes claves para conectar dispositivos, coleccionar y procesar datos, y hacer uso de interfaces web. Controlan un gran número de dispositivos mientras ofrecen servicios de seguridad y privacidad, además de solventar problemas de interoperabilidad.
Una posible estructura de IoT se muestra en la figura 5.2. El objetivo de una plataforma IoT es facilitar la integración de esos niveles: conectividad, servicios y nube, aplicaciones y análisis.

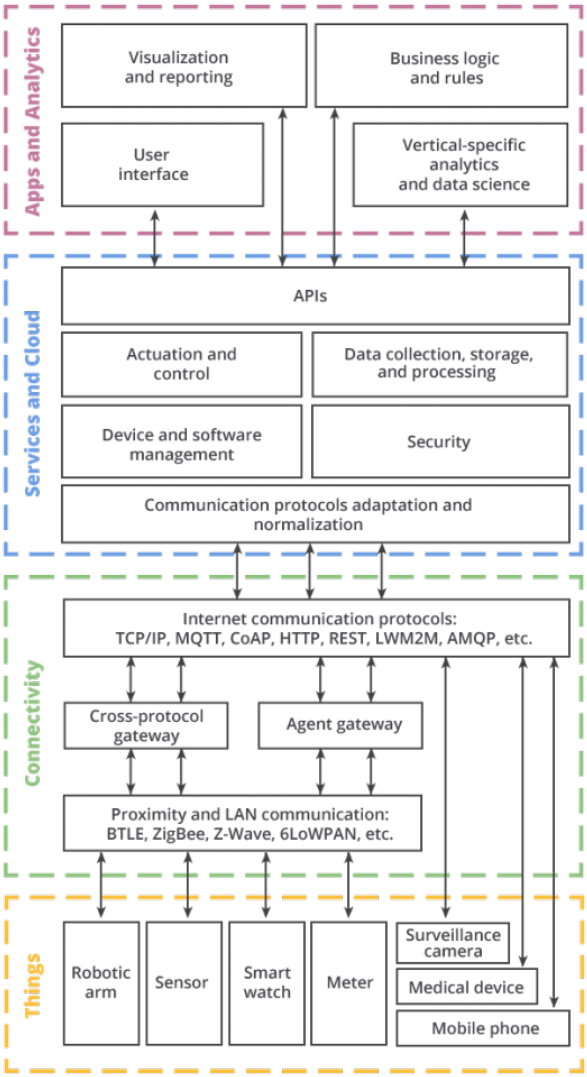


Fig 5.2. Pila de tecnologías en IoT

Algunas características que poseen las plataformas son:

Seguridad y privacidad

Un criterio fundamental para las plataformas IoT es el uso de mecanismos eficientes de seguridad y privacidad

Basada en nube

Una plataforma IoT es una solución que "vive" en la nube. Pueden ser controladas y accesibles desde cualquier lado.

Integración de tecnologías

La heterogeneidad de diferentes tecnologías es la esencia de las plataformas IoT. Una plataforma ideal, ofrecería un conjunto de protocolos de comunicación donde el usuario elige el protocolo apropiado.

Propiedad de los datos

Esta es una de las preocupaciones de las plataformas IoT, pues en un entorno en el que se envía enormes volúmenes de datos a la nube, es difícil garantizar la propiedad total de los datos. Solo soluciones donde se almacenan los datos localmente son las que ofrecen una completa privacidad de los datos. El desarrollo de algoritmos y mecanismos que garanticen esta propiedad es una de las tareas pendientes de las plataformas IoT.

Análisis de datos

El análisis de datos es una de las características principales de las plataformas IoT. Una vez subidos los datos a la nube se analizan bien con herramientas propias de la plataforma o bien con herramientas de análisis de datos externas a ella.

5.2.1 Plataformas

Existen cientos de plataformas IoT tanto Open-Source como con licencia comercial. Algunas empresas buscan hacerse un hueco en el mercado IoT con el desarrollo de estas plataformas.

- **Amazon Web Services:** Amazon domina el mercado de la nube. Fueron los primeros en traer el Cloud-Computing allá por el 2004. Su plataforma es extremadamente escalable soportando millones de interacciones entre los dispositivos y el servidor. Usa sus propios servicios para el funcionamiento de la plataforma.
- **IBM Watson:** IBM es otro gigante IT que ha fijado su objetivo en el IoT. Su plataforma busca la facilidad para acceder a sus servicios en la nube. Se puede, entre otros, almacenar datos por un periodo de tiempo definido u obtener información de los dispositivos conectados.
- **Salesforce:** Salesforce es una plataforma impulsada por Thunder, la cual se basa en alta velocidad y tiempo real en las decisiones en la nube.

En la siguiente tabla se muestran las características de algunas plataformas IoT.

Platforms	a) Support of heterogeneous devices	b) Type	c) Architecture	d) Open source	e) REST	f) Data access control	g) Service discovery
AirVantage™	Needs gateway	M2M PaaS	Cloud-based	Libraries only (Apache v2, MIT and Eclipse v1.0	Yes	OAuth2	No
Arkessa	Yes	M2M PaaS	Cloud-based	No	n.a.	Facebook like privacy settings	No
ARM mbed	Embedded devices	M2M PaaS	Centralized/ Cloud-based	No	CoAP	User's choice	No
Carriots®	Yes	PaaS	Cloud-based	No	Yes	Secured access	No
DeviceCloud	Yes	PaaS	Cloud-based	No	Yes	n.a.	No
EveryAware	Yes	Server	Centralized	No	Yes	4 levels	No
Everyware	Needs gateway	PaaS	Cloud-based	No	Yes	n.a.	No
EvryThng	Yes	M2M SaaS	Centralized	No	Yes	Fine-grained	No
Exosite	Yes	PaaS	Cloud-based	Libraries only (BSD license)	Yes	n.a.	No
Fosstrack	RFID	Server	Centralized	No	No	Locally stored	No
GroveStreams	No	PaaS	Cloud-based	No	Yes	Role-based	No
H.A.T.	Home devices	PaaS	Decentralized	Yes	Yes	Locally stored	Yes
IoT-framework	Yes	Server	Centralized	Apache license 2.0	Yes	Locally stored	Yes
IFTTT	Yes	SaaS	Centralized	No	No	No storage	Limited
Kahvihub	Yes	Server	Centralized	Apache license 2.0	Yes	Locally stored	Yes
LinkSmart™	Embedded devices	P2P	Decentralized	LGPLv3	No	Locally stored	Yes
MyRobots	Robots	Robots PaaS	Cloud-based	No	Yes	2 levels	No
Niagara^{AX}	Yes	M2M SaaS	Distributed	No	n.a.	n.a.	n.a.
Nimbits	Yes	Server	Centralized/ Cloud-based	Apache license 2.0	Yes	3 levels	No
NinjaPlatform	Needs gateway	PaaS	Cloud-based	Open source hardware and Operating System	Yes	OAuth2	No
Node-RED	Yes	Server	Centralized	Apache license 2.0	No	User-based privileges	No
OpenIoT	Yes	Hub	Decentralized	LGPLv3	No	User-based privileges	Yes
OpenMTC	Yes	M2M client/ Server	Centralized/ Cloud-based	No	Yes	Secured access	No
OpenRemote	Home devices	Server	Centralized	Affero GNU Public License	Yes	Locally stored	No
Open.Sen.se	Ethernet enabled	PaaS/SaaS	Cloud-based	No	Yes	2 levels	Limited
realTime.io	Needs gateway	PaaS	Cloud-based	No	Yes	Secured access	No
SensorCloud™	No	PaaS	Cloud-based	No	Yes	n.a.	No
SkySpark	No	SaaS	Centralized/ Cloud-based	No	Yes	n.a.	No
Swarm	Yes	PaaS	Cloud-based	Client is open source (unknown license)	Yes	n.a.	n.a.
TempoDB	No	PaaS	Cloud-based	No	Yes	Secured access	No
TerraSwarm	Yes	OS	Decentralized	n.a.	n.a.	n.a.	Yes

Fig 5.3. Caracterísitcas de algunas plataformas IoT[20].

5.3 Conclusiones

El rápido y cambiante mundo de IoT necesita de plataformas software que ayuden a disminuir la separación existente entre hardware y software, integrando varios servicios como: recolección de datos, seguridad, análisis en tiempo real, en una única plataforma-middleware IoT. En este

capítulo se han descrito algunas características de las plataformas IoT. No existe la plataforma perfecta, cada una ofrece ciertos servicios y es el usuario quien tiene que decidir cuál usar. En los siguientes capítulos se analizarán OpenIoT y Kaa como plataformas IoT Open-Source en mayor profundidad. Se elegirá, según nuestro criterio, la más adecuada para desplegar el escenario planteado al inicio de este documento.

6 OpenIoT

6.1 Introducción

A pesar de la expansión de las aplicaciones IoT en la nube, la ausencia de una semántica que proporcione una interoperabilidad entre las diferentes aplicaciones es una de las principales limitaciones de IoT. Esta ausencia de una unificación se refleja en los diferentes vocabularios y formas existentes para describir las cosas/objetos físicos. No existe un modelo a seguir para integrar todos los servicios. OpenIoT [21][22][23][24][25] surge como un proyecto europeo que busca la unificación de esas soluciones. Se trata de una plataforma de código abierto que proporciona una convergencia entre los diversos sistemas IoT. Mezcla el concepto Cloud-Computing con el concepto de redes de sensores de IoT. OpenIoT se basa en SSN como el modelo para la unificación de los diferentes sistemas IoT y flujos de datos. OpenIoT ofrece una infraestructura versátil para coleccionar datos de cualquier sensor disponible. Se hace uso del concepto "datos enlazados", Linked Data, en inglés. En el que los datos de los sensores se vinculan entre sí, para que puedan ser compartidos y entenderse entre ellos, ampliando así la información. OpenIoT incluye también un middleware que facilita la recolección de datos de cualquier sensor disponible.

6.2 Plataforma

6.2.1 Arquitectura

La arquitectura de la plataforma se puede ver en la figura 6.1

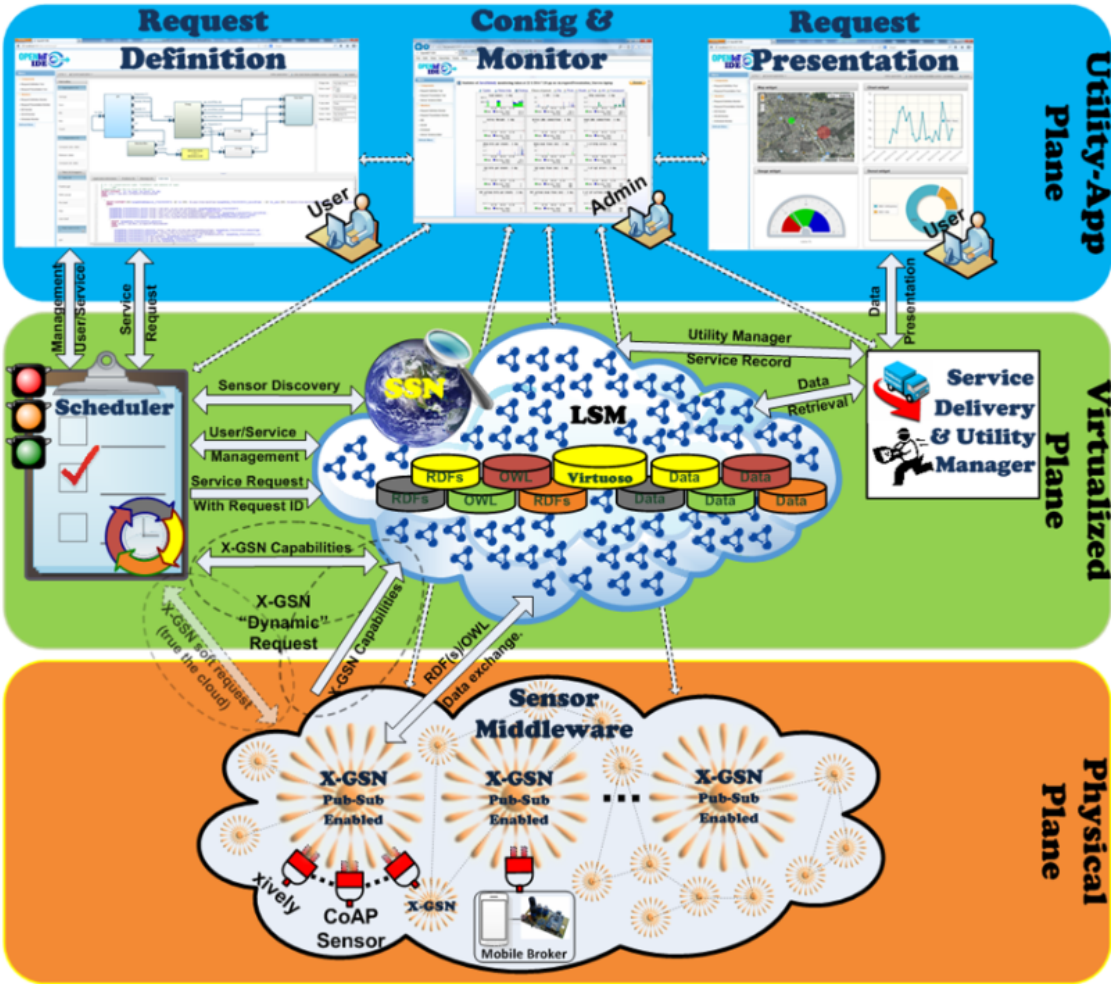


Fig 6.1. Arquitectura OpenIoT

Plano físico

En el plano físico se encuentran los sensores. En OpenIoT se usa X-GSN como middleware para el intercambio de información entre los sensores y la nube. Este middleware se encarga de filtrar, combinar y coleccionar los datos recopilados por los sensores. X-GSN es una versión extendida de GSN, cuya principal característica es el uso de sensores virtuales. Los datos de los sensores están escritos basándose en SSN (Semantic Sensor Network), una semántica para las redes de sensores. Esto proporciona una representación que hace más fácil compartir, descubrir, integrar e interpretar los datos. Un ejemplo de esta semántica se puede ver en la figura 6.2

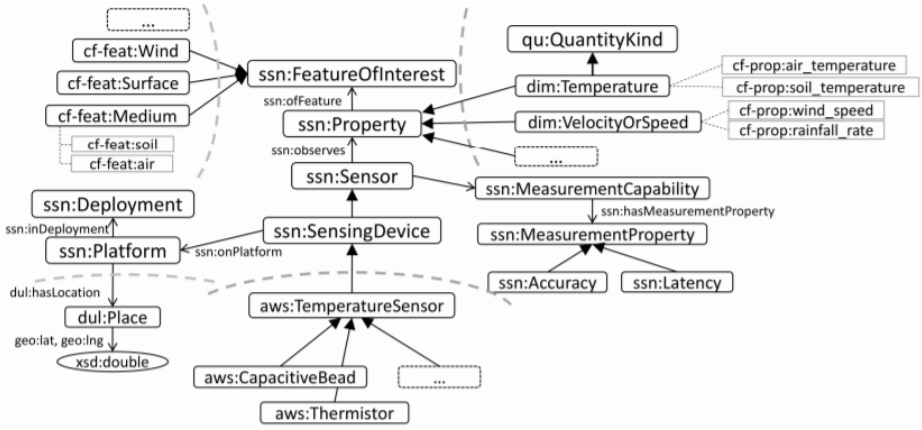


Fig 6.2. Semántica de SSN.

En X-GSN se configuran los sensores usando una descripción en XML, en la que se definen los campos de los sensores. Estos campos están asociados a la semántica SSN. En la figura 6.3 se puede ver un ejemplo de esta descripción.
Para realizar el registro de sensores en la plataforma OpenIoT, es necesario que cada sensor tenga una instancia del mismo almacenada en la nube. Por ello se envía a la plataforma LSM un archivo con los metadatos del sensor, en el que se definen las propiedades del mismo. Este archivo describe los datos definidos en el archivo de configuración XML. Un ejemplo del archivo de metadatos se puede ver en la figura 6.4. En la figura 6.5 se puede ver el proceso para el registro de un sensor en OpenIoT. Inicialmente se crea y se configura el sensor (definición de campos, configuración del tiempo de subida ...), mediante la descripción XML (figura 6.3). Posteriormente, se envía un archivo a la plataforma LSM en la nube conteniendo las propiedades del sensor. Éste se transforma en una descripción RDF, una descripción usada en la Web Semántica, que permite una representación de datos enlazados mediante la identificación de objetos (URIs), lo que facilita el descubrimiento de recursos.

```
<?xml version="1.0" encoding="UTF-8"?>
<virtual-sensor name="demo_weatherstation" priority="10" >
  <processing-class>
    <class-name>org.openiot.gsn.vsensor.LSMExporter</class-name>
    <init-params>
      <param name="allow-nulls">false</param>
      <param name="publish-to-lsm">true</param>
    </init-params>
    <output-structure>
      <field name="temp" type="double" />
      <field name="humidity" type="double" />
    </output-structure>
  </processing-class>
  <description>CSIRO demo station</description>
  <life-cycle pool-size="10"/>
  <addressing>
  </addressing>
  <streams>
    <stream name="input1">
      <source alias="source1" sampling-rate="1" storage-size="1">
        <address wrapper="csv">
          <predicate key="file">data/station_1057.csv</predicate>
          <predicate key="fields">timed, temp, humid, co, co_2, co2_4, no2</predicate>
          <predicate key="formats">timestamp(d/M/y H:m), numeric, numeric,numeric,numeric,numeric,numeric</predicate>
          <predicate key="bad-values">NaN,6999,-6999,null</predicate>
          <predicate key="timezone">Etc/GMT-2</predicate>
          <predicate key="sampling">4000</predicate>
          <predicate key="check-point-directory">csv-check-points</predicate>
        </address>
        <query>select * from wrapper
        </query>
      </source>
      <query>select temp as temp,humid as humidity, timed from source1</query>
    </stream>
  </streams>
</virtual-sensor>
```

Fig 6.3. Configuración de un sensor.

```
sensorName=opensense_1
source="http://planetdata.epfl.ch:22002/gsn?REQUEST=113&name=opensense_1"
author=opensense
sensorType=lausanne
sourceType=lausanne
information=Air Quality Sensors from Lausanne station 1
sensorID="http://lsm.deri.ie/resource/1150805994572850"
feature="http://lsm.deri.ie/OpenIoT/opensensefeature"
fields="humidity,temperature"
field.humidity.propertyName="http://lsm.deri.ie/OpenIoT/Humidity"
field.humidity.unit=Percent
field.temperature.propertyName="http://lsm.deri.ie/OpenIoT/Temperature"
field.temperature.unit=C
```

Fig 6.4. Metadatos de un sensor.

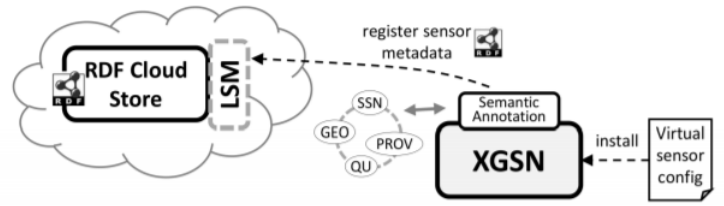


Fig 6.5. Registro de un sensor virtual

La colección de datos por parte de los sensores se basa en "wrappers" que coleccionan datos a través de distintos protocolos, como UDP, puerto serie o HTTP. Esto permite al usuario desarrollar y configurar el protocolo que quiera usar, lo que hace a la plataforma agnóstica respecto al hardware. Los datos se representan como flujos de listas de datos.
El registro de sensores en la plataforma para su posterior descubrimiento, así como la recolección de datos y su posterior almacenamiento en LSM, se hace con unos pocos archivos XML, lo que permite al desarrollador crear grandes soluciones IoT con poco esfuerzo. En la figura 6.6 se puede ver el proceso de colección y envío de datos para su almacenamiento.

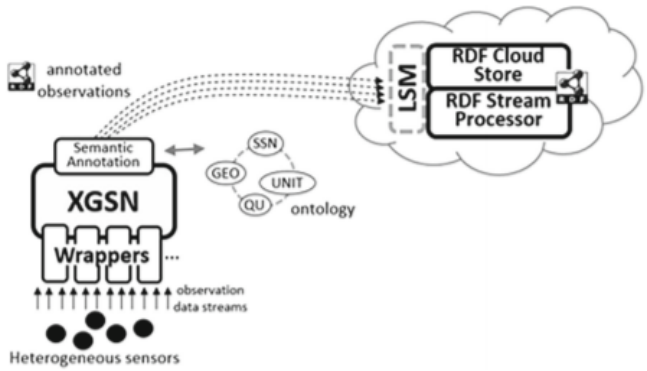


Fig 6.6. Envío de datos en OpenIoT.

Por otro lado, OpenIoT ofrece también soporte para el descubrimiento y recolección de datos por parte de sensores móviles tales como pulseras, gafas, relojes y, en definitiva, sensores incluidos en dispositivos móviles. Todo esto se realiza a través de un middleware publish/subscribe para IoT llamado CUPUS (CloUd-based PUblish/Subscribe middleware).
CUPUS tiene dos componentes principales: 1) un agente (mobile broker, de ahora en adelante) ejecutándose en un dispositivo móvil y 2) un motor de procesado en la nube basado en publish/subscribe (cloud broker, de ahora en adelante), que se encarga del procesamiento de los datos recopilados por los sensores. CUPUS soporta el contenido basado en publish/subscribe. En la

arquitectura OpenIoT, los datos recopilados por los dispositivos móviles se anotan y almacenan en la nube, a través de X-GSN, de la misma forma que con los sensores estacionarios. El mobile broker puede controlar los sensores conectados localmente y realizar un preprocesamiento de los datos adquiridos por los sensores y enviarlos a la nube. Además, también puede recibir publicaciones de la nube y notificar a los clientes que estén subscritos. Como se puede observar en la figura 6.7, el mobile broker recibe los datos de los sensores a través de un mensaje publish y los envía al cloud broker. Pudiendo también conectarse o desconectarse del mismo. El cloud broker puede enviar una notificación al mobile broker. Por otro lado, el cloud broker envía una instancia de los datos del sensor ya procesados al almacenamiento RDF, a través de X-GSN, de la misma forma que en los sensores estacionarios, tal y como se ha visto anteriormente.

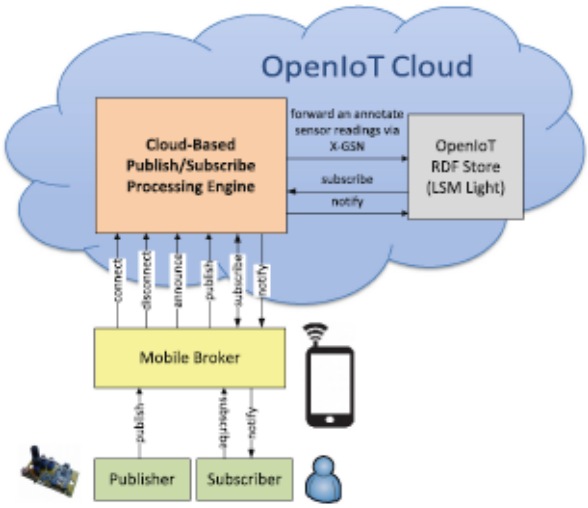


Fig 6.7. Arquitectura publish/subscribe

Plano Virtual

El plano virtual está compuesto por el almacenamiento en la nube (LSM-Light), el Scheduler y el servicio de entrega (SD&UM). LSM-Light (Linked Sensor Middleware Light) es el componente principal de OpenIoT. Es la infraestructura de almacenamiento en la nube. Esta infraestructura, además de almacenar datos y metadatos, también es capaz de ofrecer computación en la nube (software) como por ejemplo Scheduler y SD&UM.
LSM considera los sensores como fuentes de entrada de datos. Los datos provenientes de los sensores se transforman en una representación de datos enlazados, como por ejemplo, RDF. Existen dos formas de importar datos en LSM: pull y push. Una es la fuente de datos (X-GSN,CoAP...) la que se encarga de enviar los datos y la otra es el propio LSM quien obtiene los datos periódicamente. LSM está compuesto por dos módulos, LSM-Client y LSM-Server. LSM ofrece funciona-

lidades como wrappers para la colección y visualización de datos en tiempo real, o un lenguaje SPARQL para las peticiones.
OpenIoT usa Virtuoso, un middleware y motor de almacenamiento, que combina RDF, XML y base de datos virtuales en un solo sistema. Es el corazón de LSM-Light. El scheduler se encarga de formular las peticiones realizadas por los usuarios y, de acuerdo con ellas, interactúa con la plataforma OpenIoT a través de la base de datos en la nube. El Scheduler tiene dos funciones prncipales: descubrir sensores y controlar los servicios. Todo esto se realiza mediante peticiones a la BBDD. Las peticiones se realizan en lenguaje SPARQL, un lenguaje usado en la web semántica para consulta de sentencias RDF. En la figura 6.8 se observa la Cloud DB junto con cada uno de los servicios que realiza peticiones. Request presentation y Request Definition realizan las peticiones a través de otros servicios (Scheduler y SD&UM).
El Scheduler recibe las peticiones del servicio Request Definition, situado en el plano de utilidad. El Scheduler las procesa, y envía la respuesta de vuelta al plano de utilidad. El SD&UM (Service Delivery & Utility Manager), al igual que el Scheduler, también recibe y procesa peticiones del plano de utilidad pero en este caso del servicio Request Presentation, para la presentación de los datos. Las peticiones del Scheduler y SD&UM se realizan a la Cloud DB mediante una API.

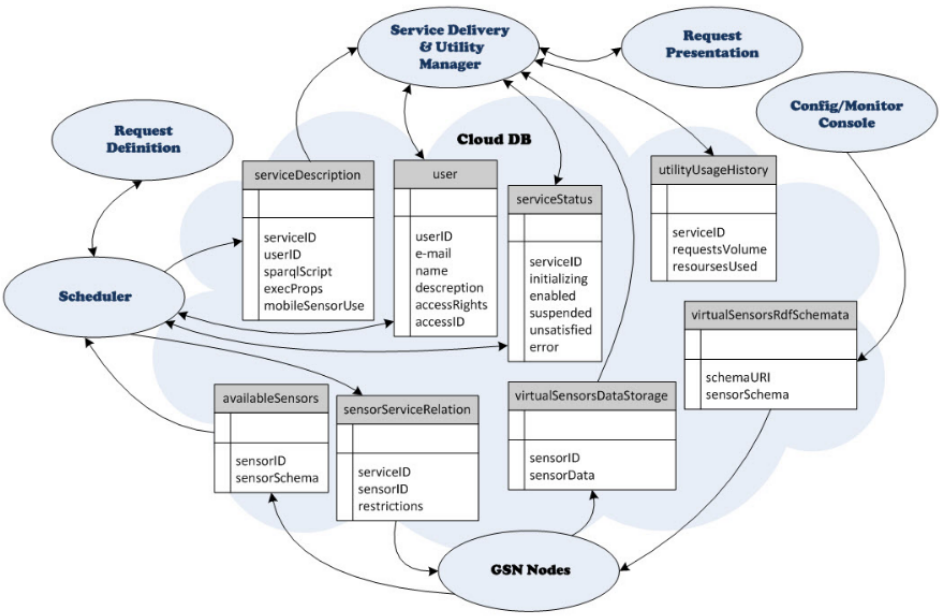


Fig 6.8. Relación de cada servicio con la Cloud DB

Plano de utilidad

En este plano se sitúan las interfaces de usuario. Request Definition es una aplicación que permite a los usuarios crear sus servicios de OpenIoT en una interfaz basada en nodos. Cada mo-

delo de grafos se divide en aplicaciones, siendo cada una de estas aplicaciones un conjunto de servicios que describen a la aplicación. Esto le permite al usuario controlar diferentes aplicaciones desde un solo punto. Todos estos servicios se cargan automáticamente (mediante peticiones) cuando el usuario accede a la web. Un ejemplo de esta interfaz se muestra en la figura 6.9

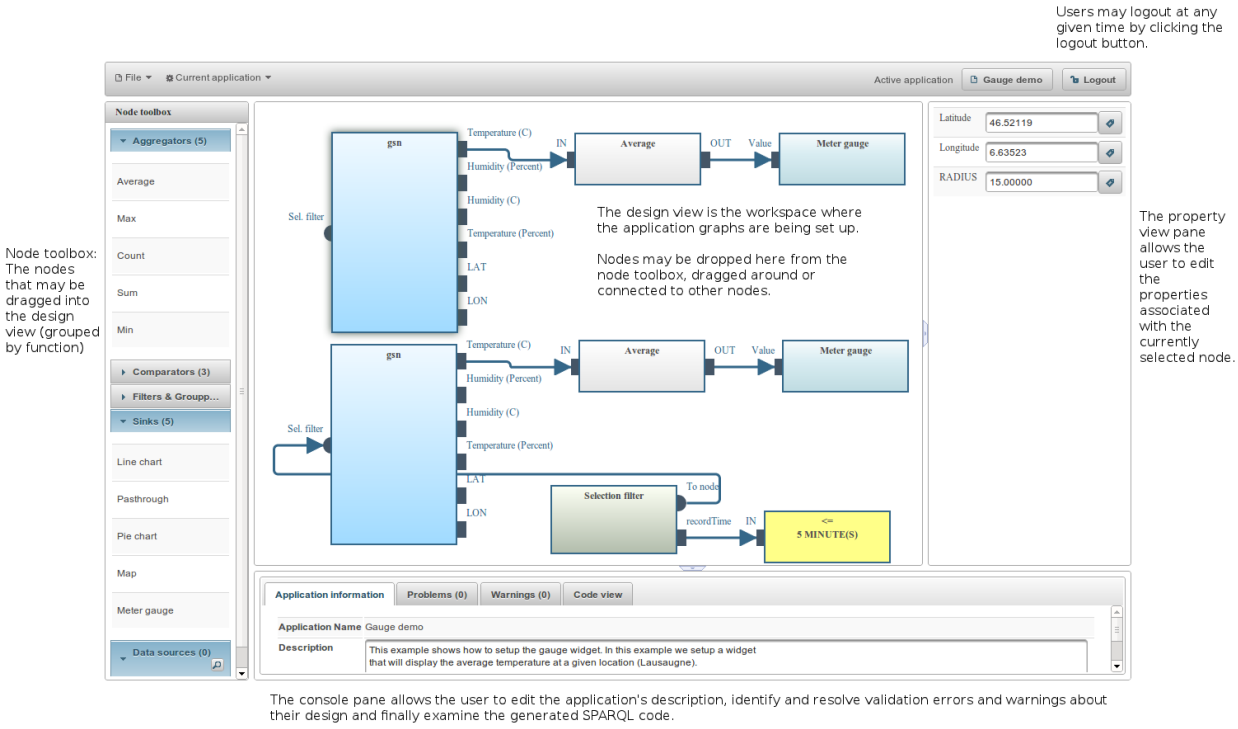


Fig 6.9. Request Definition

Request Presentation es una aplicación web que proporciona al usuario una interfaz visual de los servicios que previamente ha creado en Request Definition. Obtiene la información de los nodos de Request Definition y muestra una interfaz con los datos.
En la figura 6.10 se muestra un ejemplo con la definición del servicio en Request Definition, y en la figura 6.11 la interfaz gráfica de los datos en Request Presentation, una vez recopilados.

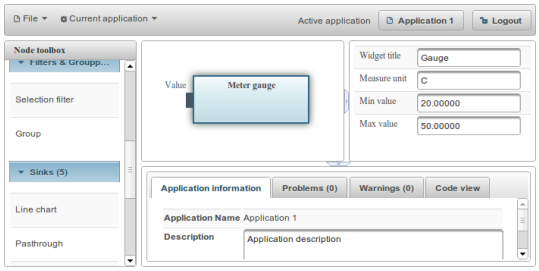


Fig 6.10. Definition

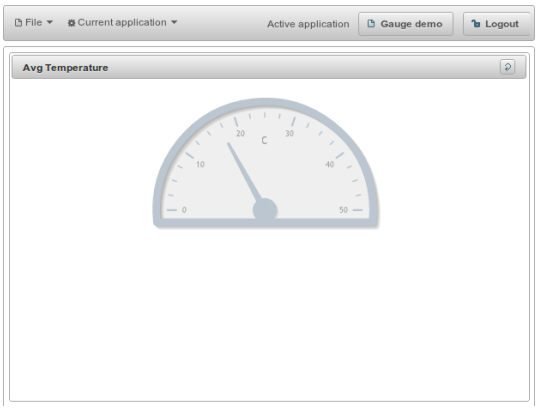


Fig 6.11. Presentation

El IDE es el otro de los servicios situados en el plano de utilidad. Proporciona accesibilidad a los otros módulos o servicios de OpenIoT. Otra de las funcionalidades que soporta IDE es la monitorización. Para implementar dicha funcionalidad se usa JavaMelody. Entre las funciones de monitorización se encuentra proporcionar datos sobre el tiempo medio de respuesta o el número de ejecuciones, la toma de decisiones ante problemas o, mostrar gráficos sobre número de sesiones, consumo de java o número de ejecuciones.

6.2.2 Flujo de datos

En base a la arquitectura que se muestra en la figura 6.1, la figura 6.12 representa un ejemplo del flujo que siguen los datos en la plataforma.

0. X-GSN publica los datos de los sensores virtuales basados en la configuración local de cada nodo (sensor).
1. Los usuarios realizan peticiones al Scheduler de los sensores disponibles con determinados atributos usando la interfaz Request Definition.
2. El Scheduler ejecuta estas peticiones (en lenguaje SPARQL) enviadas por los usuarios.
3. Una vez que tenga la respuesta (los sensores disponibles) se envía de vuelta al Scheduler.
4. El Scheduler la reenvía al módulo Request Definition, mostrándose la información al usuario.
5. El usuario, con ayuda de Request Definiton, define peticiones para realizar determinadas reglas sobre los sensores analizados. Esta información se guarda en un objeto OSDSpec (Figura 6.13). Este objeto se envía entonces al Scheduler con la ayuda de 'registerService.
6. El Scheduler analiza la información recibida y envía la petición al servicio necesario.
7-10. Una vez que se haya configurado, el usuario puede usar el módulo Request Presentation para visualizar los datos del servicio registrado. Con ayuda del SD&UM 'getAvailableAppIDs'

el Request Presentation recupera todos los servicios/aplicaciones registrados de acuerdo a un usuario específico

11. El usuario realiza una petición para recuperar los resultados relacionados con el servicio en concreto. Esto se hace enviando una petición ("pollForReport") desde Request Presentation al SD&UM pasándole el ID de la aplicación ('application ID').
12. El SD&UM realiza una petición ("getService") para solicitar toda la información relacionada al servicio.
13. El servicio proporciona la información al SD&UM.
14. El SD&UM analiza la información, disponible en un objeto OSMO, y reenvía el script SPARQL incluido, el cual ha sido creado por Request Definition (paso 5) y almacenado por el Scheduler (paso 6), a la interfaz SPARQL del servicio.
15. El resultado se envía al SD&UM en formato SparqlResultsDoc.
16. El SD&UM lo reenvía al Request Presentation en un objeto que incluye información de cómo esos datos se deben presentar (Figura 6.14).

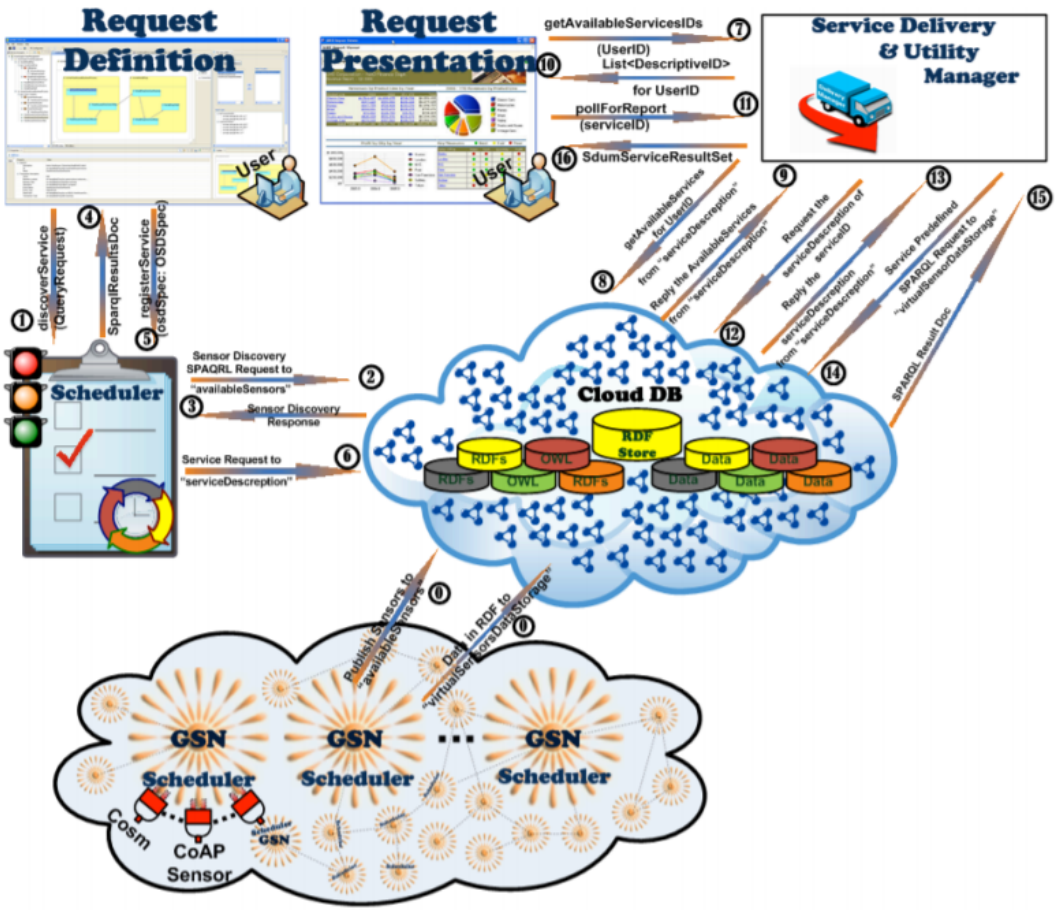


Fig 6.12. Flujo de datos en OpenIoT

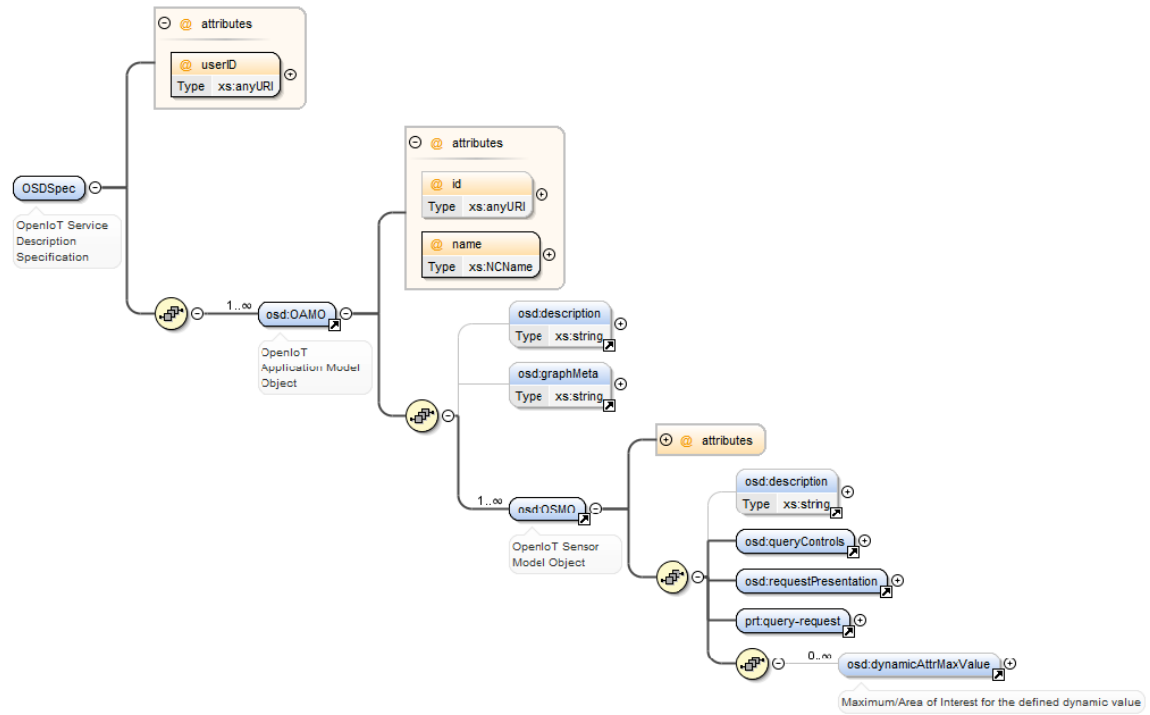


Fig 6.13. Grafo de un objeto OSDSpec.

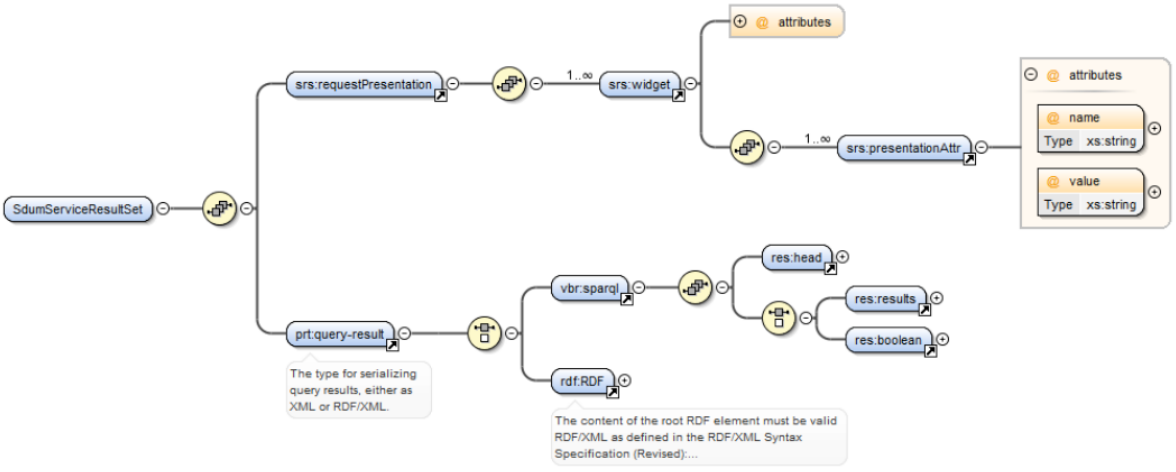


Fig 6.14. Grafo de un objeto SdumServiceResultSet.

Ventajas

En IoT cualquier cosa u objeto (TV,Smartphone...) genera una gran cantidad de eventos. Cada uno de estos objetos usa tecnologías diferentes y, desarrollar aplicaciones y servicios para controlar todo esto puede llegar a ser una tarea muy complicada. Por ello, se usan middlewares con el fin facilitar el desarrollo, proporcionando una integración entre la comunicación y la computación de los dispositivos. Entre los middleware se encuentra MAPS o TinyDB.
A pesar de que con los middleware se facilita el desarrollo de aplicaciones en IoT, cada middleware realiza una determinada función pero no ofrece una solución IoT completa en cuanto a servicios se refiere. Es aquí donde aparece OpenIoT, que permite reunir un conjunto de servicios en una única plataforma. La principal ventaja de OpenIoT es la unificación de los diferentes sistemas o servicios IoT, así como el envío de datos.
A través de la herramienta IDE (Integrated Development Environment) Core (figuras 6.15 y 6.16) que ofrece OpenIoT, se puede controlar las aplicaciones IoT. Esta herramienta integra muchas de las herramientas de OpenIoT en una. Permite configurar los sensores para su integración en X-GSN (Schema Editor), monitorización del estado de los servicios IoT (SD&UM) o definir los servicios IoT (Request Definition). Esto permite el desarrollo de aplicaciones IoT de una manera más rápida y sencilla.

Fig 6.15. OpenIoT IDE

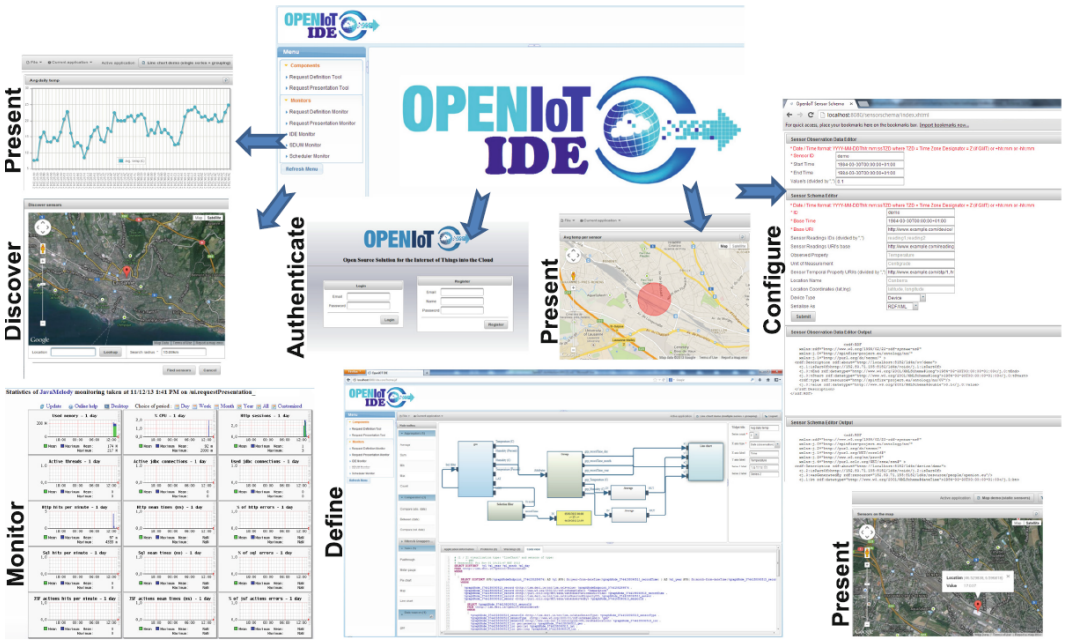


Fig 6.16. Servicios integrados en IDE

6.3 Requisitos

Las aplicaciones IoT deben cumplir una serie de requisitos necesarios en entornos IoT. Algunos de estos requisitos son:

6.3.1 Requisitos funcionales

Descubrimiento de recursos

Una de las características principales de OpenIoT es el descubrimiento de recursos. Una vez que los sensores envían su configuración a la nube a través de X-GSN, OpenIoT, mediante el Scheduler, haciendo uso de la semántica SSN, es capaz de descubrir nuevos sensores o nodos.

Control de datos y recursos

OpenIoT, a través de la herramienta SDUM, permite llevar una constante monitorización sobre los recursos. Esto permite, además de la recopilación y visualización de los datos, llevar

un control sobre los mismos.

Control de código

OpenIoT es un proyecto completamente Open-Source, y todo el código fuente está disponible para descargar. El desarrollador puede usar el código para crear nuevos servicios, extender OpenIoT con nuevos wrappers para sensores , o incluso mejorar la plataforma.

6.3.2 Requisitos no funcionales

Escalabilidad

OpenIoT es una plataforma totalmente escalable, se pueden añadir nuevos nodos tan solo configurando el sensor y enviando la información sobre sus datos (metadatos) al almacenamiento en la nube. Una vez registrado el sensor, el usuario puede acceder al nuevo nodo (sensor).

Disponibilidad

OpenIoT está formado por varios módulos/servicios cuyo funcionamiento es responsable del funcionamiento global de la plataforma. Por ello, un fallo en alguno de los servicios (i.e el Scheduler), implica un fallo en la plataforma. Debido a esta dependencia se puede decir que OpenIoT no dispone de una gran disponibilidad.

Seguridad

La diversidad de aplicaciones interactuando en un entorno IoT hace que la seguridad sea un punto clave para poder proteger los datos. OpenIoT deja la seguridad en manos del servicio CAS (Central authorization service), encargado de la seguridad en la web.
La primera vez se redirecciona a los usuarios a la página de login para que se lleve a cabo una autenticación. Si esta autenticación es correcta, el CAS redirecciona al usuario a la pagina web original enviando un token. Este token se envía de un servicio a otro en cada petición y cada servicio se encarga de comprobar la validez del token.

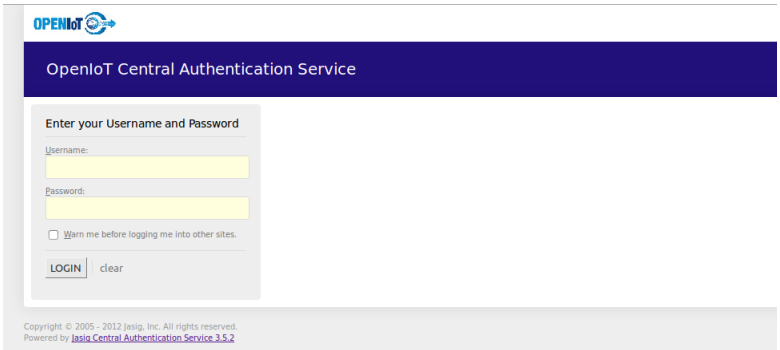


Fig 6.17. Login en CAS

La seguridad en OpenIoT se divide en 3 módulos:

- **Security Server:** En este módulo se usa CAS para la autenticación y autorización. Sus datos se almacenan en LSM-server. Cada cliente se tiene que registrar en CAS.
- **Security Client:** Este modulo proporciona control de acceso y autenticación. Se puede usar en aplicaciones web que interactuan con los usuarios.
- **Security Management:** Se trata de un módulo que se usa para llevar un control sobre servicios, usuarios y permisos.

Tiempo real

OpenIoT proporciona un servicio en tiempo real. El usuario, a través de las interfaces de usuario, puede realizar un seguimiento en tiempo real de los datos recopilados por los sensores.

Facilidad de despliegue

Un middleware IoT debe ser fácilmente desplegable por parte del usuario. OpenIoT llegó a estar entre los 10 mejores proyectos europeos open-source del año 2013. Esta popularidad debe ayudar a mantener la plataforma totalmente actualizada por parte de la comunidad de desarrolladores e investigadores. Sin embargo, OpenIoT no cumple con este requisito ya que es una plataforma con bastantes problemas a la hora de su despliegue y además no tiene el soporte que debería para dar solución a esos problemas.

6.3.3 Requisitos arquitectónicos

Interoperabilidad

Un middleware IoT debería funcionar en distintos dispositivos y con diferentes tecnologías. OpenIoT es una plataforma agnóstica respecto al hardware: los sensores pueden funcionar en cualquier hardware, tan solo es necesario describir los archivos de configuración y registrar los sensores en la plataforma mediante el envío de sus metadatos.

Distribuido

OpenIoT no es una plataforma distribuida, todos sus servicios se ejecutan en un único servidor.

Adaptativo

OpenIoT no es un middleware que se adapte a los cambios de su entorno; tiene el uso de servicios en los diferentes planos bien definidos. Sin embargo, sí que permite usar diferentes protocolos para la colección de datos por parte de los sensores, como UDP, puerto serie o HTTP.

6.4 OpenIoT en la práctica

6.4.1 Casos de éxito con la plataforma

Entre quién ha desplegado la plataforma con éxito se encuentra:

- Universidad nacional de Irlanda - DERI
- Universidad de Zagreb - FER
- SENSAP S.A
- CSIRO
- Universidad de Estambul

OpenIoT en la industria

Cada vez más, en la industria se instalan un gran número de sensores para controlar el proceso de producción de una planta. Estos sensores generan un gran volumen de datos por lo

que es necesario una solución para capturar, almacenar y procesar esos datos. SENSAP S.A ha desarrollado una solución basada en OpenIoT para la monitorización y seguimiento del flujo de materiales en un proceso de producción. Le permite definir y visualizar dinámicamente los KPIs (indicadores de rendimiento en la industria). En este entorno, los KPIs son el flujo de datos necesarios, los cuales son: A) recopilados por sensores físicos situados en la planta, B) transformados en el modelo de datos EPC-IS, un estándar usado en la industria, y C) transmitidos al middleware X-GSN que asegura una anotación semántica de estos datos y su posterior publicación a la nube OpenIoT. Los sensores virtuales son capaces de calcular entre otros:

- Tasa de operación para un proceso específico
- Métricas de utilización de las maquinas
- Tasa de producción por tipo de producto
- Porcentaje de tiempo una operación que ha sido completada

Una vez que se publica la información a la nube, los fabricantes son capaces de obtener y sintetizar la información de los sensores virtuales, con el fin de calcular los KPIs para los determinados procesos.

Para llevar a cabo este proyecto, se usó la siguiente implementación:

- Sensores: Sensores físicos con el fin de calcular KPIs (tasas de operación, información de calidad...). Sensores ópticos, escáneres de códigos de barras, son algunos de los sensores utilizados en este proyecto.
- S-BOX: Productos propios desarrollados por SENSAP, para coleccionar datos de los sensores y transformarlos en eventos EPC-IS, asociados a los procesos de fabricación.
- X-GSN: Ese flujo de datos EPC-IS se envía al middleware X-GSN. Siguiendo el proceso de OpenIoT, el middleware X-GSN convierte ese flujo de datos en la anotación SSN.
- LSM: La información KPIs se envía a la nube (LSM Cloud) a través de X-GSN, tal y como marca el proceso de OpenIoT
- Visualización: Usando la herramienta Request Definition de OpenIoT, se definen servicios que calculan los KPIs asociados al proceso de fabricación. Este cálculo lo realiza a partir de los datos disponibles en el LSM Cloud.

OpenIoT en la agricultura

Otros de los proyectos en los que se ha usado OpenIoT ha sido Phenonet [24], desarrollado por CSIRO. Se trata de un proyecto usado en la agricultura que permite procesar y visualizar datos del terreno en tiempo real. Esto ayuda a la toma de decisiones sobre el cultivo como,

por ejemplo, planificar los recursos de agua o de nitrógeno en el cultivo y así, incrementar la eficiencia y rendimiento. La arquitectura de Phenonet se puede ver en la figura 6.18

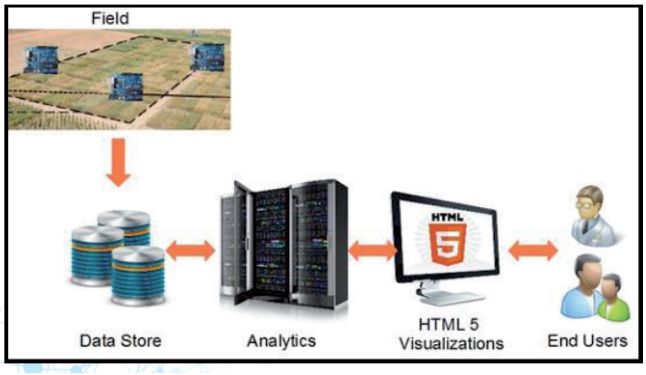


Fig 6.18. Arquitectura de Phenonet

En el campo ('field') se sitúan los sensores para medir temperatura, humedad o velocidad del viento entre otros. Los datos y metadatos se almacenan y posteriormente se procesan y analizan mediante el componente 'Data analysis'. A este componente se accede a a través de una API mediante HTML. Phenonet está basado en los siguientes módulos de OpenIoT:

- X-GSN: Se encarga de los dato provenientes del Data Store/Field.
- Scheduler y SD&UM: Usado para construir un experimento Phenonet en OpenIoT.
- LSM-Light: Se almacenan los datos existentes en Data Store para poder permitir el descubrimiento de sensores.
- Request Definition y Presentation: Estas herramientas se usan para el diseño.

6.5 Problemas

Tras un tiempo intentando desplegar sin éxito la plataforma, se ha optado por descartarla. OpenIoT estuvo entre los 10 mejores proyectos europeos en 2013 y es una plataforma con un potencial alto pero, inexplicablemente, no cuenta con ningún tipo de soporte. Han sido muchos los problemas que se han dado a la hora de la instalación sin llegar en ningún momento a ser 100 % funcional. Tras investigar por la red nos damos cuenta que son problemas comunes, de los que no hemos encontrado solución. Esto nos ha llevado a descartar la plataforma y buscar otra solución para el problema que planteamos.

7 KAA

7.1 Introducción

Kaa [26][27][28][29] es una plataforma middleware de código abierto que, al igual que OpenIoT, busca un control e integración de las aplicaciones IoT. Se trata de un middleware que trabaja entre el hardware y las aplicaciones, ofreciendo una gran comunicación, control e interoperación entre los dispositivos conectados. Recibe soporte de la empresa KaaIoT[27], que se encarga de crear soluciones haciendo uso de la plataforma, algunas de ellas muy completas.

7.2 Arquitectura

Kaa tiene un poderoso back-end que facilita el desarrollo de aplicaciones en un entorno IoT. Kaa soporta múltiples plataformas en el lado del cliente, mediante puntos finales (SDKs), en diferentes lenguajes de programación. El SDK es una librería embebida en el dispositivo conectado. Esto, junto con un lenguaje de definición de datos ("data schema"), hace que Kaa sea una plataforma muy rápida y flexible. Kaa ha sido diseñada como una plataforma robusta que facilita el desarrollo de aplicaciones IoT. La figura 7.1 muestra su arquitectura:

Fig 7.1. Arquitectura Kaa

Despliegue de la plataforma

Kaa ofrece 2 formas de despliegue, que se pueden ver en las figuras 7.2 y 7.3.

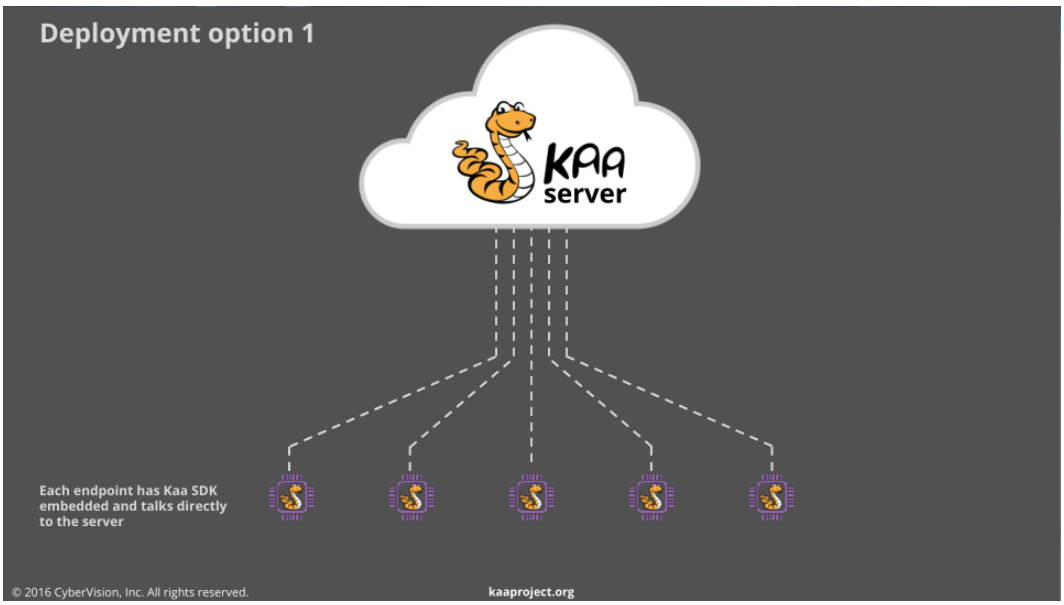


Fig 7.2. Despliegue típico de Kaa

En esta arquitectura cada uno de los endpoints tiene un SDK embebido y se comunican directamente con el servidor. Es la forma más "natural" para la que ha sido diseñada Kaa.

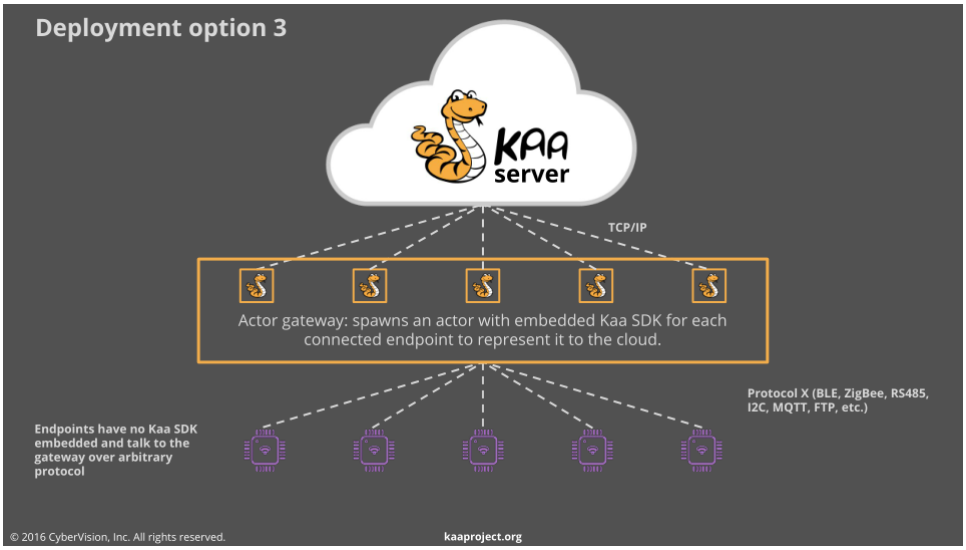


Fig 7.3. Otro despliegue con Kaa

En esta arquitectura, uno o varios gateways o nodos centrales son los que tienen el SDK embebido. Los endpoints se comunican con el gateway a través de protocolos de transporte como Bluetooth, ZigBee o MQTT. Esta arquitectura es útil cuando no se puedan instalar los SDKs en los endpoints por diferentes cuestiones (compatibilidad, bajas prestaciones...) o también cuando se quiere usar un protocolo específico de transporte. El gateway se presenta a la nube como una representación virtual del endpoint. Para versiones posteriores se plantea la inclusión de algún framework para simplificar la integración con Kaa en dispositivos en los que no se pueda ejecutar el SDK. Esta arquitectura tiene algunas ventajas como se verá más adelante.

Cluster Kaa

El servidor Kaa se puede desplegar en un entorno de nodo único o multinodo. Cuando se habla de entorno multinodo, cada nodo representa un servidor Kaa y el conjunto de nodos interconectados representa un cluster Kaa. Por defecto, Kaa funciona como un único nodo ejecutando un servidor Kaa.
Un Cluster requiere bases de datos SQL y NoSQL para almacenar los datos y metadatos de los endpoints. Las bases de datos relacionales se usan para las aplicaciones o grupos de endpoints y también para los metadatos. Las no relacionales almacenan todo la información relacionada con los endpoints. Kaa usa Apache Thrift [30] para la comunicación entre nodos. Cada nodo sube

información sobre la conexión, servicios activados, etc. Permitiendo a otros nodos usar esta información. Para la coordinación entre ellos, Kaa usa Apache Zookeeper [31].
Un nodo en un cluster está ejecutando una combinación de servicios de control, operaciones y bootstrap.

- **Servicio de control:** Es el encargado del sistema de datos, procesa llamadas de las APIs y envía notificaciones. Recibe continuamente información de Zookeeper. Para conseguir una alta disponibilidad, un Cluster Kaa tiene que incluir al menos dos nodos con el servicio de control activado. En el modo de alta disponibilidad, uno de ellos estará activo y el otro en standby, siendo Zookeeper quien se encargue de su control y activación.

- **Servicio de operaciones:** Es el encargado de procesar y enviar peticiones a los endpoints.

- **Servicio bootstrap:** Es el encargado de establecer la conexión con los endpoints. Envía información a los endpoints sobre los parámetros de conexión que pueden ser dirección IP, puerto, protocolos...

La figura 7.4 representa un cluster formado por 3 nodos o servidores.

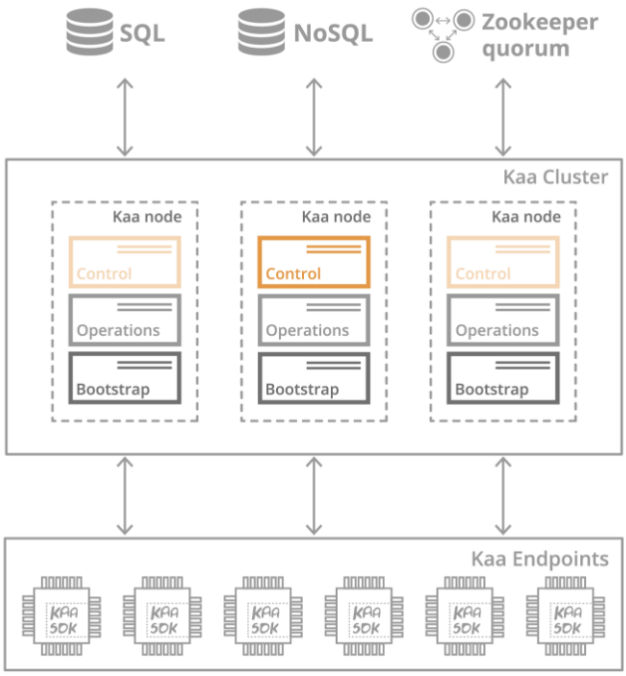


Fig 7.4. Cluster Kaa

Kaa proporciona una versión preconfigurada formada por un único nodo pudiéndose desple-

gar en local haciendo uso de VirtualBox. Otras opciones son exportarla a Amazon Web Services o instalar manualmente un servidor Kaa, pudiéndolo hacer en un entorno de un solo nodo o multinodo.
Kaa es multi-entidad (multi-Tenant) y un cluster puede soportar varias de ellas. Una aplicación en Kaa tiene que pertenecer a un Tenant y los endpoints se registran en las aplicaciones. Para distinguir los endpoints se usan perfiles de endpoints además de un identificador único. Con los perfiles de endpoints se pueden crear grupos de endpoints como se verá más adelante.

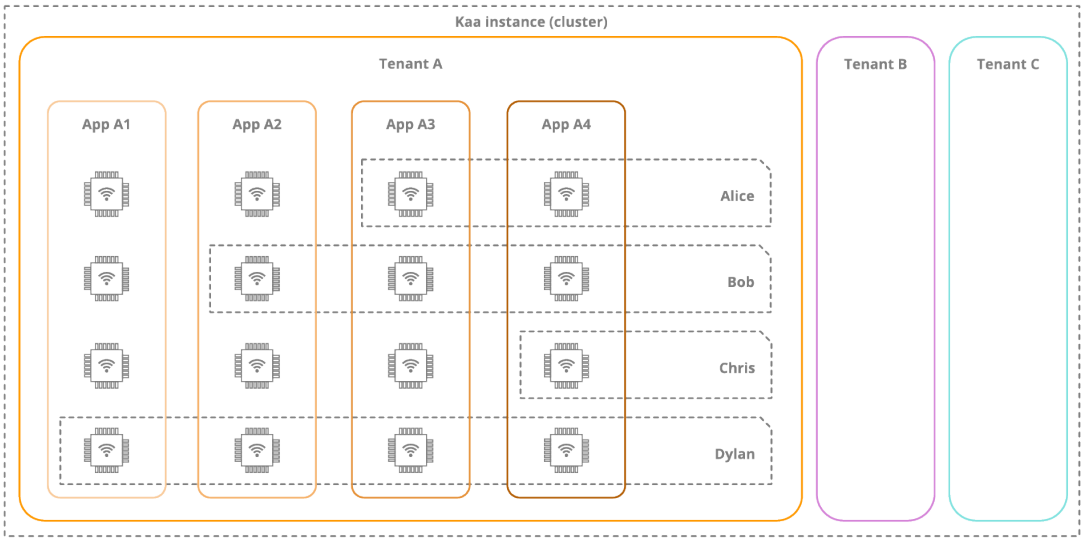


Fig 7.5. Entidades y grupos de endpoints en Kaa

Nota: Se ha analizado la versión 0.10, la última hasta la fecha de publicación de este documento.

7.3 Funcionamiento

La figura 7.6 muestra un flujo de datos en Kaa. Los endpoints recogen datos y, a través del SDK, se envían al servidor. El SDK dependerá de la plataforma y del lenguaje de programación. El envío de datos al servidor desde el SDK se realiza en un formato común, cuyo contenido se define previamente en el servidor mediante los "Log Schemas".
Kaa, como plataforma Cloud-IoT que es, almacena esos datos para que puedan ser procesados y analizados posteriormente. Kaa no ofrece un análisis de los datos como sí ofrecía OpenIoT, si no que tan solo ofrece una persistencia de los datos, sin embargo es posible integrarla con otros sistemas (p. ej. Spark o Zeppelin) para realizar un análisis de los datos. Kaa ofrece tanto bases de datos relacionales como no-relacionales para el almacenamiento. MariaDB y MongoDB

, relacional y no-relacional respectivamente, son las bases de datos por defecto usadas por Kaa. A algunas funciones como las notificaciones no se les puede aplicar una persistencia de datos. En su lugar se define el tiempo de vida (TTL) y así los dispositivos pueden recibirlas en caso de que estén offline en el momento del envío. Una vez pasado ese TTL, se elimina.

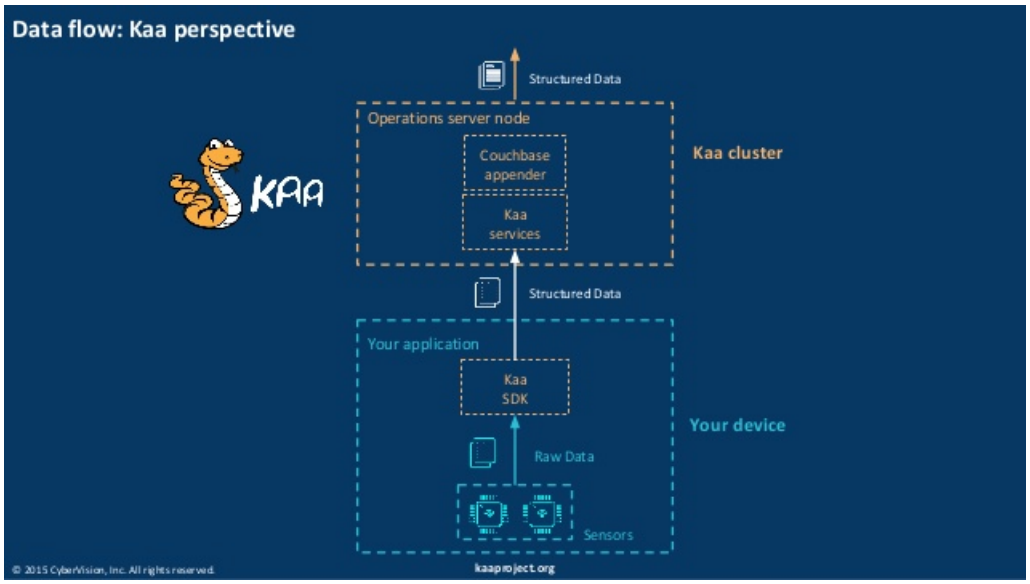


Fig 7.6. Flujo de datos en Kaa

Roles

En Kaa existen 3 roles diferentes que son asignados a los usuarios:

- **Kaa administrador:** Es el administrador más alto en Kaa. Tiene derechos para crear, editar y eliminar Tenant administradores.
- **Tenant administrador:** Usuario con permisos para configurar las aplicaciones, usuarios y familias de clases de eventos dentro de una entidad Tenant.
- **Tenant desarrollador:** Usuario que crea y controla perfiles de aplicaciones. Los usuarios con rol de Tenant desarrollador pueden crear y configurar esquemas, grupos de endpoints y notificaciones. Es quien crea el SDK.

La versión preconfigurada de VirtualBox ofrece unos credenciales por defecto para cada usuario.

Registro de endpoints

Para funcionar en un cluster Kaa, todos los endpoints tienen que registrarse haciendo uso de credenciales de seguridad y del perfil endpoint. En el proceso de registro, los endpoints primero se comunican con los servicios bootstrap para obtener una lista de servicios de operaciones. Una vez recibida, se comunica con el servicio de operación en concreto para enviar los datos del endpoint (esto es, ID y perfil de endpoint). El proceso de registro se muestra en la figura 7.7.

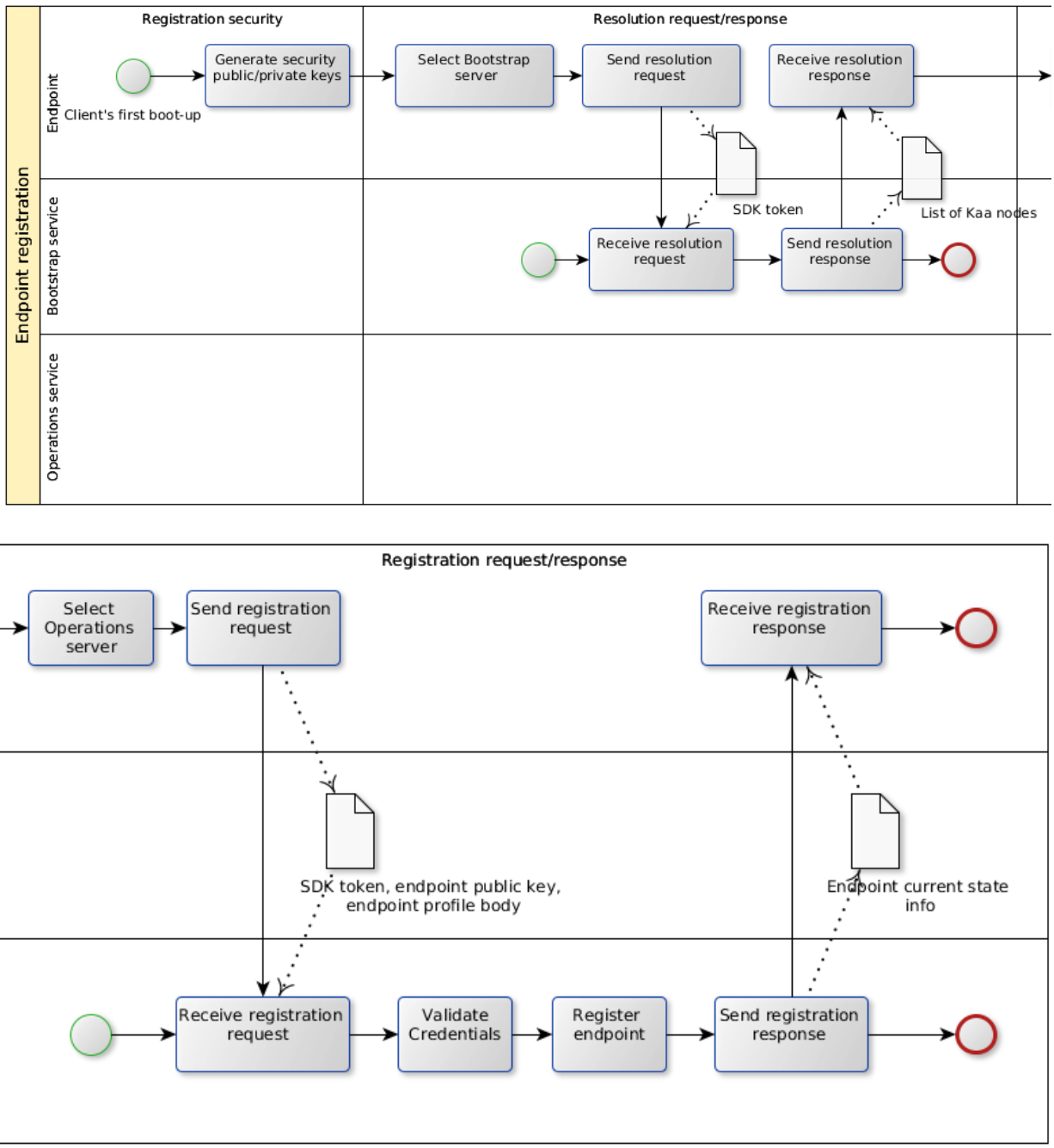


Fig 7.7. Proceso de registro de endpoint en Kaa.

El identificador del endpoint dentro de una aplicación es el Hash de su clave pública, generada automáticamente en el primer registro. Si los credenciales no son válidos, Kaa no iniciará la sesión. Al crear la aplicación, ésta se asigna a un credencial de servicio. Estos credenciales son:

- **Trustful:** Implementación por defecto y permite a cualquier endpoint registrarse y conectarse al cluster.
- **Internal:** Solo permite conectarse al cluster a una lista de endpoints en concreto, previamente suministrada al servidor.

Kaa también permite crear credenciales de servicio personalizados.

Esquemas en Kaa

Kaa ha sido diseñada para ser una plataforma sencilla de usar, por ello la configuración en el lado del servidor se realiza mediante esquemas, basados en Apache Avro [32]. El desarrollador (esto es, tenant desarrollador) es el encargado de definir estos esquemas. Son usados para la configuración de notificaciones, eventos, colección de datos... Un ejemplo de estos esquemas es:

```
{
     "type":"record",
     "name":"ExampleNotification",
     "namespace":"org.kaaproject.kaa.schema.sample.notification",
     "fields":[
          {
               "name":"message",
               "type":"string"
          }
     ]
}
```

7.4 Qué ofrece Kaa

Evaluando a Kaa como middleware IoT, se puede decir que cumple con una gran cantidad de requisitos esenciales. Se va a evaluar como el resto de middleware analizados, es decir, mediante la evaluación de los requisitos presentes en la tabla de la figura 2.1

7.4.1 Requisitos funcionales

Descubrimiento de recursos

Kaa desplegada como la arquitectura mostrada en la figura 7.3 permite un descubrimiento de nuevos dispositivos a través del gateway. El gateway es un dispositivo que actúa como puente entre los endpoint y la nube, siendo el gateway quien tiene el SDK embebido. El gateway se puede diseñar para que avise a Kaa cuando se conecte a él un nuevo dispositivo.

Kaa, con la arquitectura típica, no ofrece un descubrimiento de recursos como tal, pero mediante el sistema de eventos se puede conseguir tal propósito. Estos eventos se generan en los endpoints y, al igual que el almacenamiento de datos, se definen mediante esquemas. La figura 7.8 muestra un diagrama de cómo se generan y procesan los eventos. Haciendo uso de ellos, se puede avisar al resto de endpoints de la aparición de un nuevo recurso (endpoint).

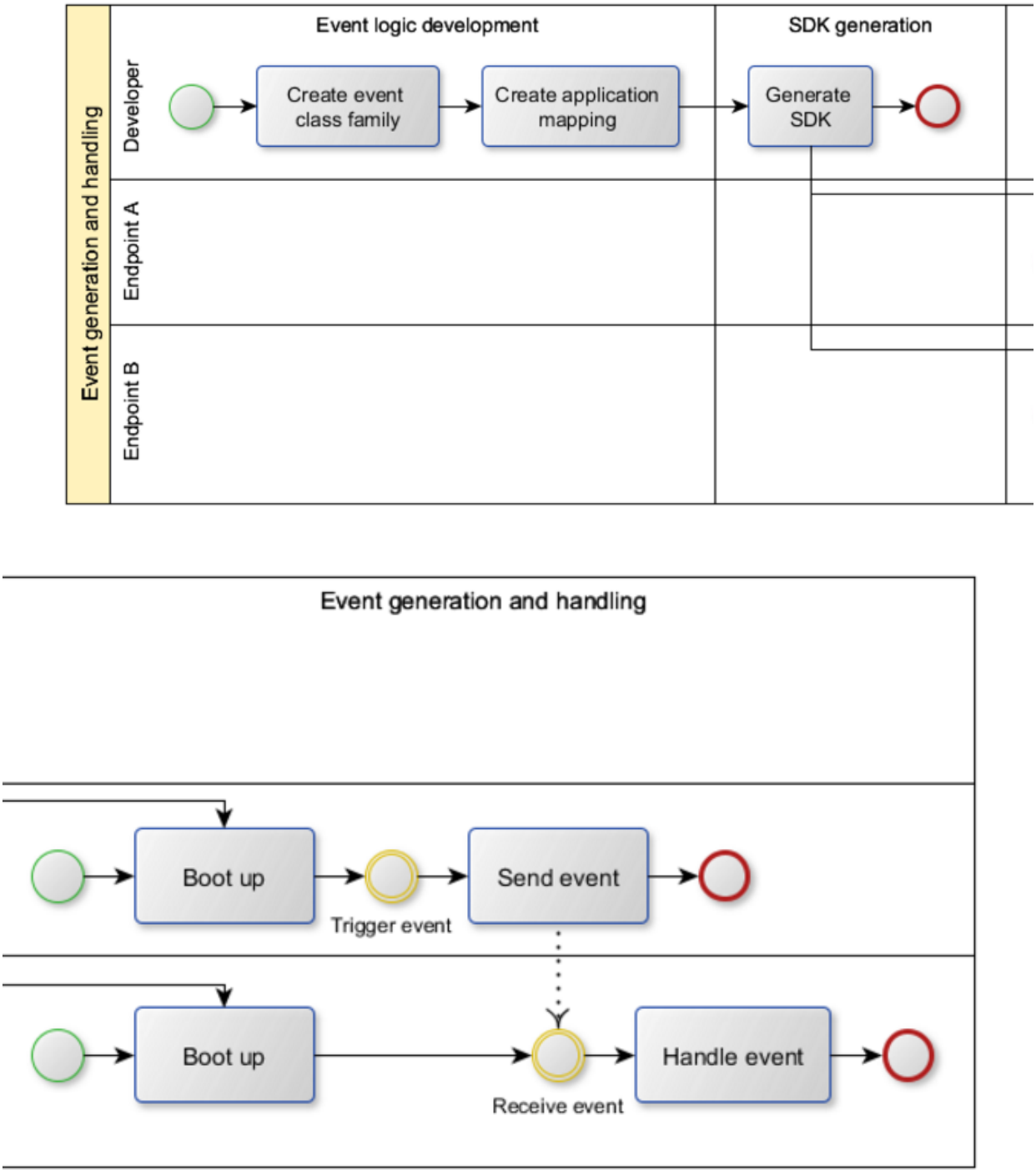


Fig 7.8. Diagrama de eventos en Kaa.

Control de recursos

En IoT se busca que los recursos puedan ser controlados para que, por ejemplo, se pueda reaccionar ante un fallo de éstos. Kaa no ofrece un control de recursos como tal, pero haciendo uso de los eventos y notificaciones se puede conseguir tal propósito. Por ejemplo, una solución sencilla podría ser que un endpoint guarde cada 5/10 min un valor característico, como su timestamp y, mediante un sistema de análisis, analizar los datos y, en caso de fallo, enviar un evento o notificación a un endpoint en concreto, que será el encargado de reaccionar ante tal efecto.
Otro ejemplo podría ser implementar en un SDK una función 'ping' a la que, haciendo uso del sistema de eventos de kaa, tengan que responder todos los endpoints en un determinado tiempo, si no lo hacen se podría considerar como endpoint caído y actuar ante tal efecto.
Por tanto, el sistema de eventos y notificaciones es un sistema completo con el que se pueden realizar muchas funciones. Toda funcionalidad tendrá que ser implementada en los endpoints, donde podrán ser conectados a servidores para realizar funciones más complejas. El motivo por el cual se usa Kaa es, sin duda, su robustez y heterogeneidad con distintas plataformas así como su facilidad de uso.

Control de datos

Los datos son la principal característica de las aplicaciones IoT. Cuando se habla de datos se hace referencia principalmente a los recopilados por los sensores. Un procesamiento, almacenamiento, así como una monitorización o filtrado de datos, es fundamental para poder desarrollar una aplicación IoT.
Kaa, a diferencia de otras plataformas comerciales, no ofrece un procesamiento o visualización de datos. En kaa tan solo se almacenan los datos procedentes de sensores. A pesar de que no ofrezca herramientas de análisis de datos como tal, se pueden usar herramientas externas.
Con estas herramientas se recopilan los datos almacenados en Kaa de manera que sean estas herramientas las que gestionen y analicen los datos. Para comunicarse con los EndPoints se usan las notificaciones Kaa mediante su API. Más adelante se verá un ejemplo práctico sobre ésto, en el que se ha usado Apache Zeppelin como herramienta para el análisis de datos y MongoDB como base de datos de Kaa. Una vez analizados, si supera un cierto umbral, se envía una notificación a Kaa para que encienda un LED en la Raspberry. En la figura 7.9 se puede ver un esquema de uso con Apache Storm.

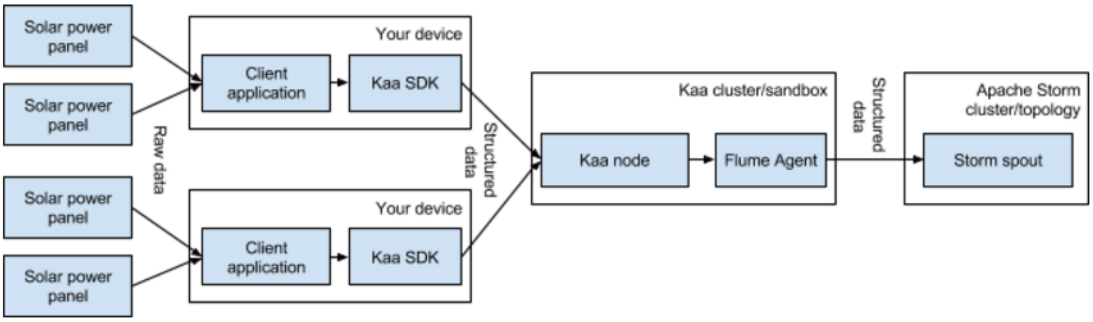


Fig 7.9. Esquema de uso para la monitorización de datos con Apache Storm

Control de código

Kaa es una plataforma Open Source con todo lo que ello implica. El código es accesible por el usuario, pudiendo desplegar su propio clúster Kaa.

7.4.2 Requisitos no funcionales

Escalabilidad

La plataforma Kaa ha sido diseñada para ser escalable horizontalmente. Se pueden añadir miles de endpoints al mismo nivel, es decir, no puede haber varios niveles en la jerarquía. Para poder agrupar distintos endpoints se puede hacer uso de los grupos de endpoints (endpoint group).

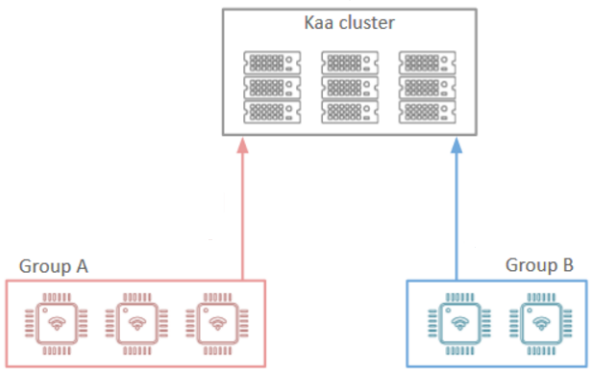


Fig 7.10. Grupos de Endpoints

Tiempo real

Kaa es un sistema en tiempo real, entendiendo tiempo real como unos ms de retraso. Ya que Kaa, por defecto, usa HTTP sobre TCP para la comunicación entre SDK y Cluster, por tanto, por definición, es un servicio con retardos, aunque despreciables para el escenario que se plantea. Kaa es agnóstica respecto al transporte, para un mismo endpoint se pueden usar diferentes protocolos de transporte. Para escenarios en los que la característica de tiempo real pueda marcar la diferencia entre éxito y fracaso, se puede usar UDP como protocolo de transporte.

Disponibilidad

La arquitectura horizonal y lineal de Kaa hace que no cuente con ningún punto de fallo. Esto junto con la conexión entre los cluster, dota al sistema con una alta disponibilidad. Además, los datos de los endpoints se almacenan en bases de datos tolerantes a fallos. Esta interconexión entre los nodos, hace que sea necesario un balanceo de carga para no sobrecargar la red. Kaa usa un balanceo activo para redireccionar el tráfico entre los distintos nodos según el tráfico de cada uno de ellos. Para la coordinación entre los diferentes nodos se usa Apache Zookeeper.

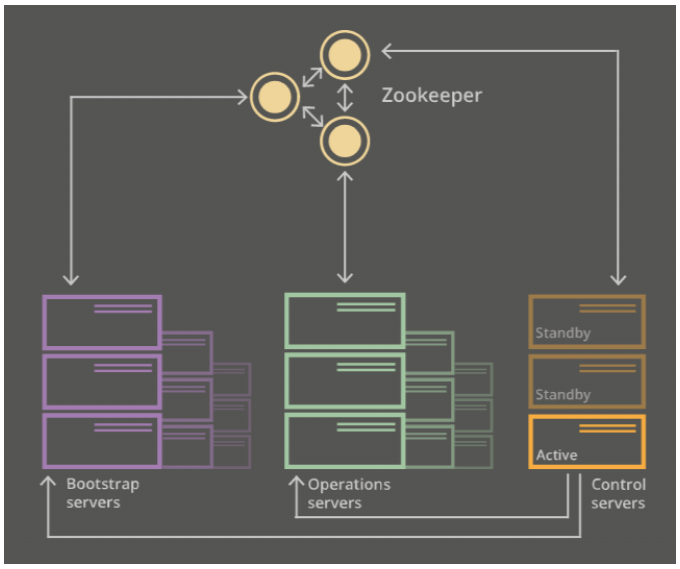


Fig 7.11. Esquema Zookpeer en Kaa

Seguridad y privacidad

Por defecto, en la comunicación entre Kaa y el SDK se usa una encriptación con 2048-bit RSA y AES-256. Kaa también asegura un almacenamiento de datos seguro en la base de datos

así como la autenticación en la plataforma. Cada vez que se inicia una nueva sesión, el endpoint genera un par de claves RSA y envía la clave pública al servidor firmada con la clave pública del nodo que gestiona la sesión. Kaa usa el hash SHA-1 de la clave pública del endpoint como identificador del endpoint en el sistema.

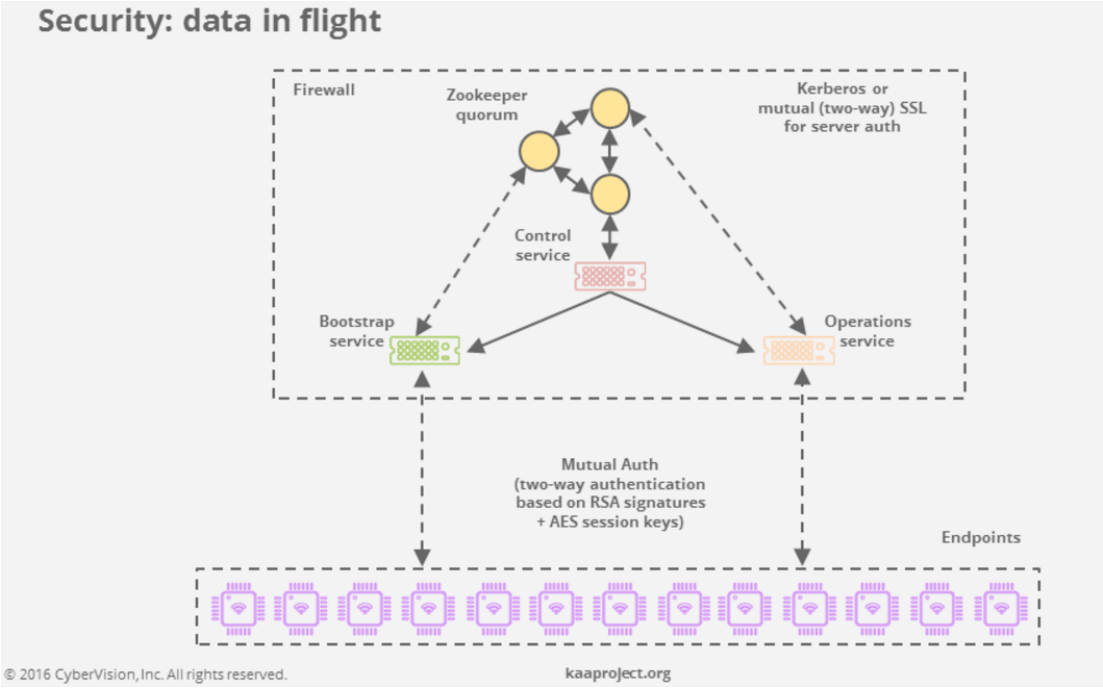


Fig 7.12. Seguridad en Kaa

Facilidad para su despliegue

Kaa es un middleware que ofrece facilidades para su desarrollo. Dispone de una versión en una maquina virtual ya preconfigurada con aplicaciones y ejemplos de uso de la plataforma. El uso de SDKs facilita la interoperabilidad de diferentes plataformas sin la intervención del usuario. Con Kaa se consigue que el desarrollador se 'olvide' de la parte del servidor.

Popularidad

Kaa es un middleware reciente y con mucho soporte y documentación, lo que facilita su despliegue.

7.4.3 Requisitos arquitectónicos

Interoperabilidad

La interoperabilidad entre distintas plataformas es una de las características de Kaa, en la que una misma aplicación puede ser desarrollada para diferentes plataformas, tan solo descargando el SDK generado por el servidor. Para conectar un endpoint a la plataforma será necesario compilar este SDK en el dispositivo junto a la aplicación desarrollada. Esto hace que sea una plataforma totalmente heterogénea en cuanto a plataformas se refiere. Un endpoint puede ser ejecutado en distintos dispositivos, desde un teléfono móvil a una Raspberry o arduino, debido al bajo procesamiento que requiere (10KB de RAM y 40KB de ROM para el caso de C).

Tabla 7.1. Compatibilidad entre SDKs y plataformas

	C	C++	Objetive - C	Java
Linux	✓	✓		✓
Windows		✓		✓
Android				✓
iOS			✓	
Raspberry Pi	✓	✓		

Distribuido

Kaa es una plataforma distribuida, que se organiza en clusters donde cada uno de ellos puede ejecutar una determinada funcionalidad que será controlada como un único sistema. Por defecto, Kaa está formada por un único cluster ejecutando el servidor Kaa.

Abstracción de la programación

Kaa proporciona una interfaz al desarrollador, posibilitando una sencillez a la hora de crear las aplicaciones. El desarrollador tan solo tiene que crear los esquemas de datos, pues la plataforma se encarga de generar el código. Además de la interfaz, Kaa proporciona una API que permite realizar, mediante llamadas a la misma, todas las funciones que se pueden llevar a cabo a través de la interfaz y así nodos ajenos a Kaa, pueden realizar una determinada tarea, como por ejemplo, enviar una notificación desde un sistema de análisis de datos a un endpoint.

Adaptativo

Un middleware IoT debe adaptarse a los cambios que se produzcan en su entorno, como pueda ser el cambio en la red.
Kaa soporta virtualmente cualquier protocolo de red, endpoint o almacenamiento de datos, lo que hace a la plataforma tolerante a cambios.

7.5 Uso de la plataforma

7.5.1 Instalación

Como se ha comentado, existen 2 formas de instalar la plataforma, bien desplegar en local la plataforma haciendo uso de la versión ya preconfigurada para VirtualBox y la que incluye algunos ejemplos, o bien, construir un servidor propio haciendo uso del código fuente disponible en el GIT de Kaa. También se puede desplegar la versión preconfigurada en Amazon Web Services. Todo esto está perfectamente documentado en la web de Kaa.

7.5.2 Características de la plataforma

Eventos

Kaa proporciona un mecanismo para la entrega de eventos (mensajes) a través de los endpoints. Los eventos pueden ser unicast o multicast, eligiendo así el/los destinatario/s. Los endpoints generan los eventos y los envían al Kaa server, éste se encarga de enviarlos a los endpoints acorde al esquema de eventos, previamente configurado por el desarrollador. La figura 7.8 muestra el proceso de envío y recepción de eventos.
En el siguiente ejemplo se muestra la definición de un esquema de eventos. El formato está basado en Avro Schema.

```
{
     "namespace": "com.company.project",
     "type": "record",
     "classType": "event",
     "name": "SimpleEvent2",
     "fields": [
          { "name": "field1", "type": "int"},
          { "name": "field2", "type": "string"}
     ]
}
```

Cada evento se basa en una clase en concreto (EC), definida en el esquema de eventos. Una EC se identifica mediante su nombre completo, en el ejemplo de arriba sería: "com.company.project .SimpleEvent2". El identificador es único y no puede haber 2 ECs con el mismo nombre en el mismo tenat.
Por otra parte, las clases de eventos (ECs) se agrupan en familias de clases de eventos (ECFs). Un ECF se identifica por su nombre y/o nombre de clase, por lo que no puede haber 2 ECFs con el mismo nombre o nombre de clase en un mismo tenat. El siguiente esquema muestra un ejemplo de definición de ECF:

```
[
    {
        "namespace": "com.company.project.family1",
        "name": "SimpleEvent1",
        "type": "record",
        "classType": "event",
        "fields": []
    },
    {
        "namespace": "com.company.project.family1",
        "name": "SimpleEvent2",
        "type": "record",
        "classType": "event",
        "fields": [
            { "name": "field1", "type": "int" },
            { "name": "field2", "type": "string" }
        ]
    }
]
```

Colección de datos

Los endpoints almacenan datos recopilados ("log") siguiendo una estructura predefinida. El SDK implementa la subida de 'logs' desde los endpoints al servidor. El servidor lo almacena en bases de datos. En el siguiente ejemplo se muestra un simple esquema de almacenamiento de datos (log schemas), compatible con Avro Schema:

```
{
    "name": "LogData",
    "namespace": "org.kaaproject.sample",
    "type": "record",
    "fields": [
        {
            "name": "tag",
            "type": "string"
        },
        {
            "name": "message",
            "type": "string"
        }
    ]
}
```

La especificación de la base de datos se hace mediante log appenders usando la interfaz de usuario.

Perfiles y grupos

Kaa permite la agrupación de endpoints pertenecientes a la misma aplicación, en grupos de endpoints. Estos grupos se controlan de forma independiente, para ello se usan filtros de perfiles (profile filter (PF), en inglés) asignados a un determinado grupo. Los endpoints cuyos perfiles coincidan con los filtros de perfiles asociados al grupo, se registrarán automáticamente como miembros de ese grupo. Los filtros son expresiones que definen algunas características de los miembros del grupo. Se pueden asignar varios perfiles a un grupo.
A cada grupo de endpoints dentro de una aplicación se le asignan diferentes pesos. Este valor se usa para la prioridad de cada grupo, siendo el valor más grande el que mayor prioridad tiene. Por defecto, hay un grupo "all" cuyo peso es 0. Este grupo contiene todos los endpoints registrados en la aplicación.
Los endpoints envían sus perfiles al servidor durante su registro. Además de los perfiles enviados por los endpoints (lado del cliente), se pueden definir perfiles en el lado del servidor. Con todos estos datos, el servicio de operaciones de Kaa clasifica los endpoints dentro de grupos basados en los filtros.

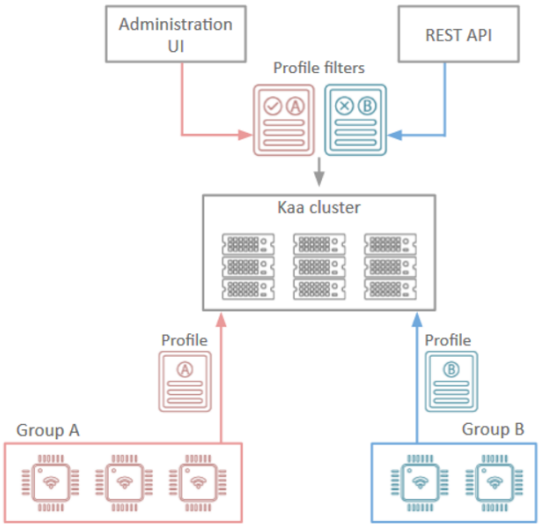


Fig 7.13. Uso de perfiles de grupos de endpoints en Kaa

Los PF están basados en Spring Language [33]. Los filtros se evalúan usando 3 variables de contexto:

- "cp" - Perfil de endpoint en el lado del cliente
- "sp" - Perfil de endpoint en el lado del servidor
- "ekh" - Hash del endpoint

Al definir los filtros se le asignan los esquemas de perfiles de endpoints, tanto del lado del cliente como del servidor. Estos filtros estarán relacionados con dichos esquemas.
Un ejemplo de esquema en el lado del cliente podría ser:

```
[
   {
      "name":"ClientSideEndpointProfileChild",
      "namespace":"org.kaaproject.kaa.common.endpoint.gen",
      "type":"record",
      "fields":[
         {
            "name":"otherSimpleField",
            "type":"int"
         },
         {
            "name":"stringField",
            "type":"string"
         }
      ]
   }
]
```

Y en el lado del servidor:

```
[
   {
      "namespace":"org.kaaproject.kaa.common.endpoint.gen",
      "type":"record",
      "name":"ServerSideEndpointProfile",
      "fields":[
         {
            "name":"simpleField",
            "type":"string"
         },
         {
            "name":"arraySimpleField",
            "type":{
               "type":"array",
               "items":"string"
            }
         }
      ]
   }
]
```

Un ejemplo de filtro asociado a dichos esquemas puede ser:

```
#cp.recordField.otherSimpleField==123
#sp.arraySimpleField[0]=='SERVER_SIDE_VALUE_1'
```

Notificaciones

Kaa dispone de un sistema de notificaciones para la entrega de mensajes desde el clúster Kaa hasta los endpoints. La estructura de los datos se define en el esquema de notificaciones, configurado en el servidor y desplegado dentro de los endpoints. Las notificaciones a tópicos de forma que para recibir una notificación, el endpoint tiene que subscribirse a uno o varios tópicos. También, se puede asignar un tópico a todo un grupo de enpoints. El envío de notificaciones se puede hacer mediante la interfaz de usuario o mediante una llamada a la API.

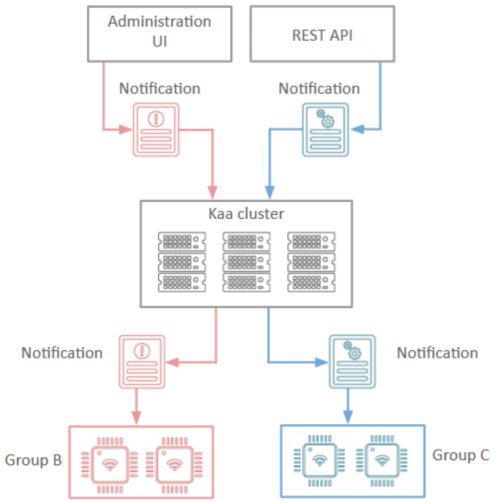


Fig 7.14. Notificaciones en Kaa

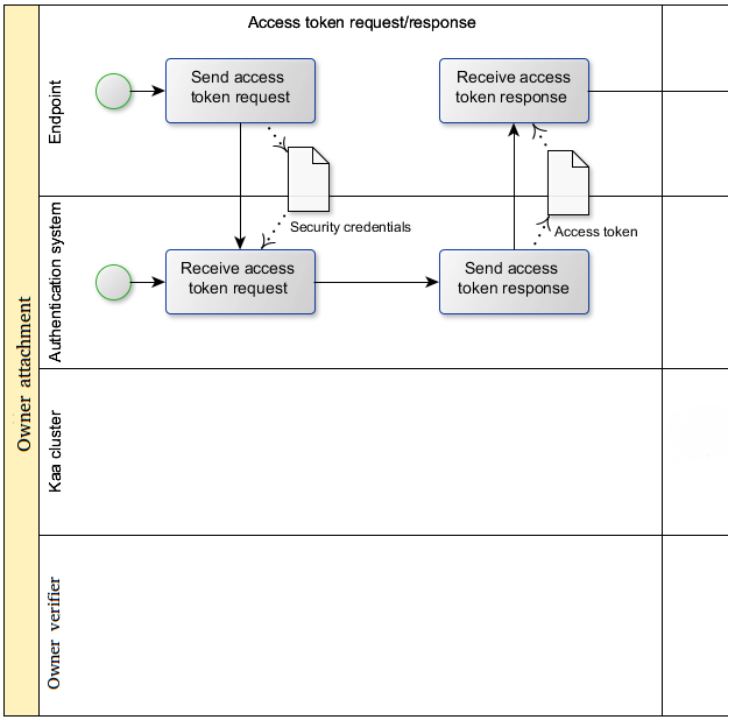
Distribución de datos

La distribución de datos es una de las principales características de Kaa ya que los desarrolladores pueden definir cualquier tipo de datos mediante los esquemas de Kaa.

Propiedad de los Endpoints

En Kaa, los usuarios pueden asociar Endpoints a propietarios. Estos propietarios pueden ser personas, grupos de personas o organizaciones. Para capturar o añadir un endpoint a un propietario se puede hacer bien mediante el token de acceso o bien a través de un endpoint ya añadido a un propietario.
La figura 7.15 muestra el proceso para añadir un endpoint a un usuario. El endpoint obtiene su token usando el sistema de autenticación y lo envía al cluster Kaa. El cluster verifica el token y añade el endpoint al propietario.

Para la verificación existen los verificadores de propietarios. Kaa incorpora 4 verificadores por defecto. El verificador Trustful, el cual acepta cualquier ID y token y los otros 3 verificadores son los verificadores de Facebook, Google + y Twitter. Útiles cuando las aplicaciones estén integradas con Facebook o Google, ya que están implementados para verificar sus cuentas. El usuario puede crear sus propios verificadores.
La figura 7.16 muestra el proceso para añadir un endpoint a un propietario mediante otro endpoint previamente añadido a ese propietario.



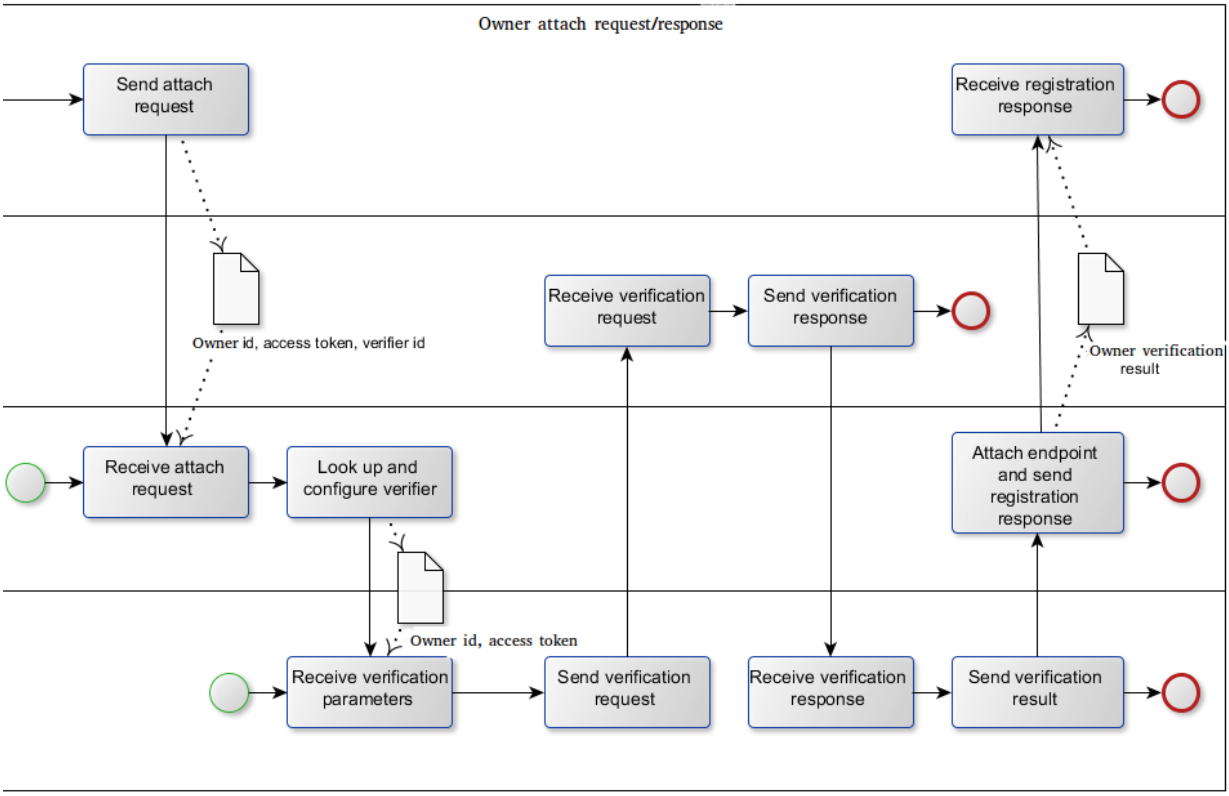


Fig 7.15. Verificación usando el token de acceso.

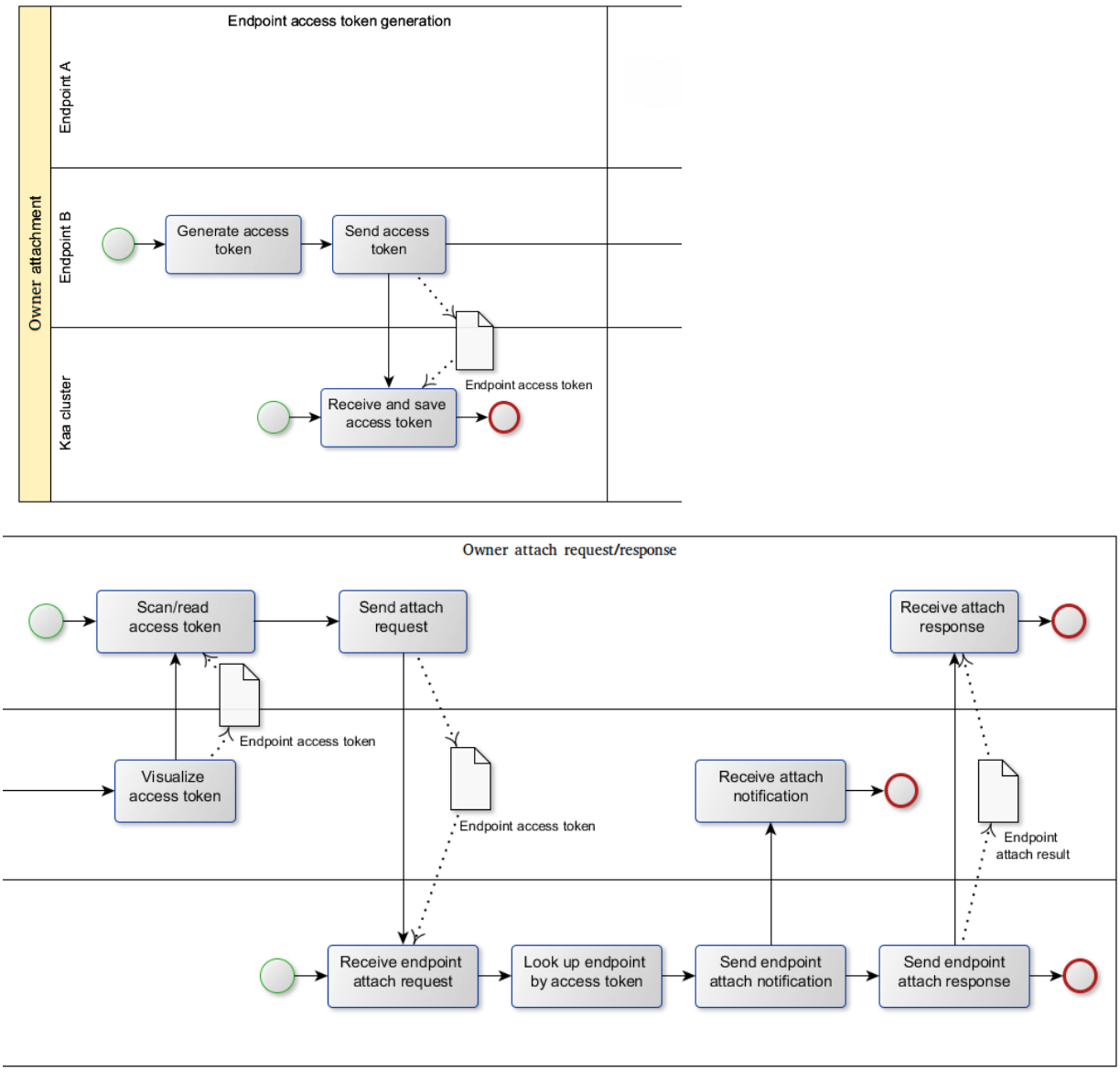


Fig 7.16. Verificación mediante un endpoint previamente añadido al propietario.

Para ciertas funciones como son los eventos en Kaa, es necesario añadir el endpoint a un usuario.

Abstracción en la capa de transporte

Kaa ha sido diseñada para soportar virtualmente cualquier protocolo de transporte de datos. Además se pueden usar diferentes protocolos en un mismo endpoint, por ejemplo las notifica-

ciones mediante SMS y la configuración y datos mediante TCP.

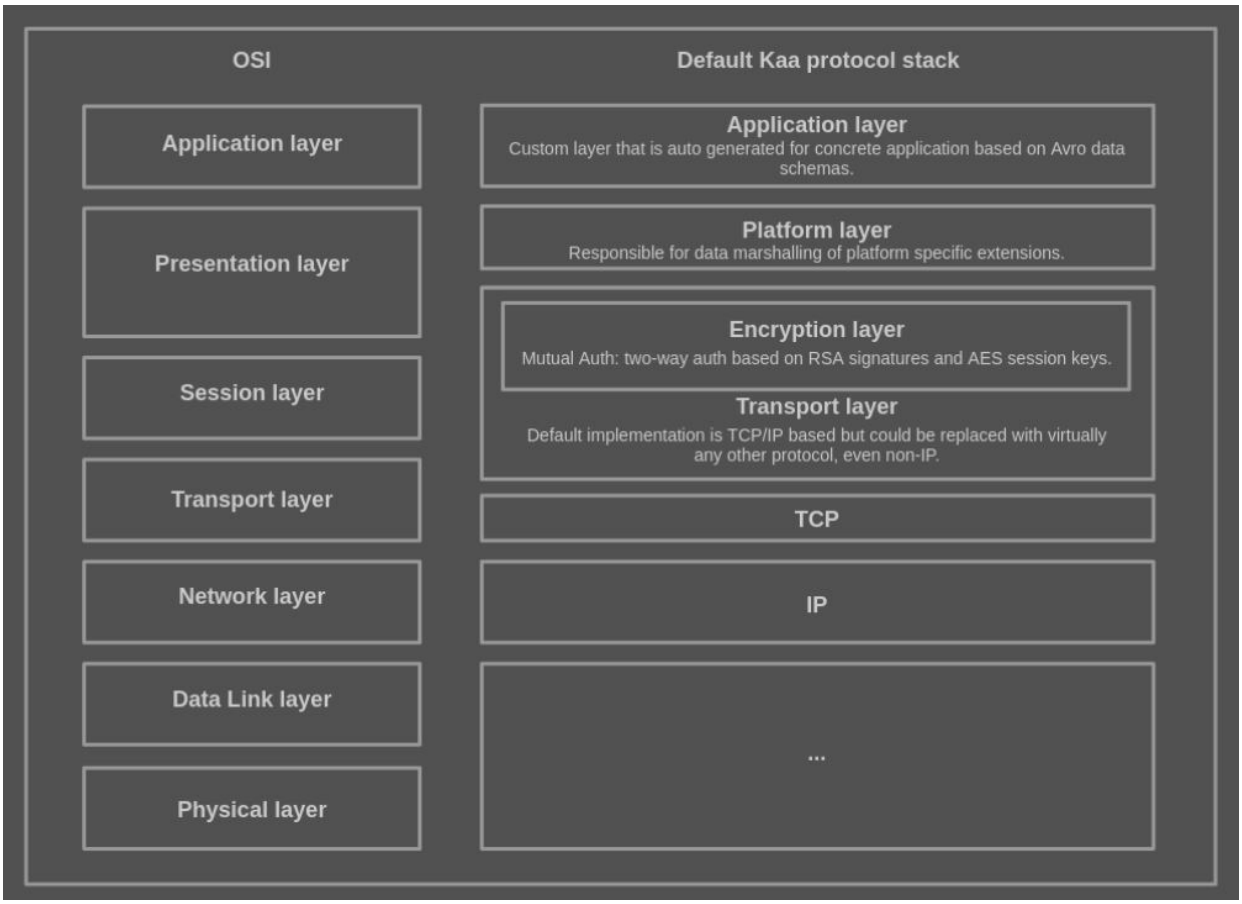


Fig 7.17. Pila de protocolos de Kaa

7.5.3 Edge analytics

Fog-Computing y Edge-Computing

Desde hace unos años el almacenamiento y procesamiento en la nube (en inglés, Cloud Computing) ha ido usándose cada vez más, hasta tal punto que muchas de las tareas se realizan en la nube, quitándoles así protagonismo a los dispositivos y haciéndolos tan solo emisores y receptores de datos. Kaa, como muchas otras, es una plataforma en la que su motor de procesamiento está en la nube. Los endpoints, a vista de Kaa, son dispositivos cuya función es únicamente enviar datos a la nube (servidor Kaa).
Pero desde la proliferación del Internet de las cosas, cada vez hay más dispositivos conectados a Internet, y según algunas predicciones, se estima que para 2030, la cantidad de datos disponibles en Internet crecerá a 1 Yottabyte (1 seguido de 24 ceros) con un tráfico diario de 2GB por usuario.

Seguramente IoT sea uno de los promotores de este crecimiento.
El uso de miles de millones de dispositivos enviando gran cantidad de datos, hace que aparezca problemas de latencia y ancho de banda de la red. A la vista de esto, el uso de Cloud-Computing no parece muy eficiente.
Uno de los primeros artículos sobre este problema fue publicado por Christopher Mims, titulado "Forget 'the Cloud'; 'the Fog' Is Tech's Future" [36]. En él aparecen los conceptos de 'Fog Computing' y 'Edge Computing', que hacen referencia al lugar donde se sitúa el procesamiento de datos.
El concepto de Edge/Fog Computing reducirá el ancho de banda, puesto que el procesamiento de datos se realiza en el punto de generación y tan solo se envía la información resumen al siguiente nivel. Incluso se propone enviar solamente alarmas en muchos casos.

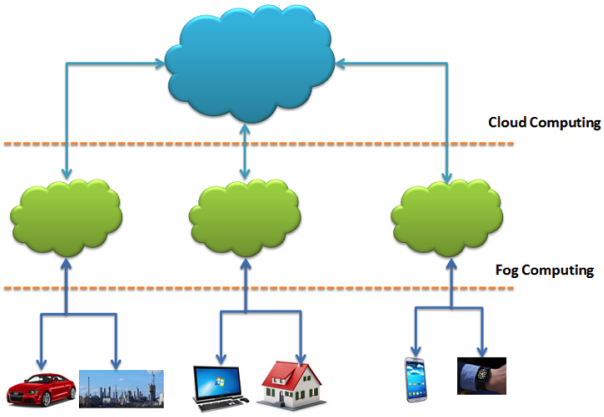


Fig 7.18. Fog Computing

El concepto Edge/Fog aunque es parecido, hay una diferencia y está en que Fog deja el procesamiento en manos de un nodo o servidor dentro de la red, mientras que Edge hace que cada dispositivo en la red juegue su propio papel de procesamiento, de forma que cada dispositivo de manera independiente elige qué información guardar y cuál subir a la nube. Edge tiene la ventaja de que hay menos puntos de fallo en la red, mientras que Fog es más escalable al tener un servidor centralizado.

Edge Computing en Kaa

Este concepto, cada vez más usado, también se ha trasladado a Kaa [37]. Desplegando Kaa como la arquitectura mostrada en la figura 7.2, se puede implementar la funcionalidad del Edge Computing en los gateways. Estos gateways se encargan de realizar un procesamiento de datos antes de enviarlos a la nube. Es el desarrollador quien se encarga de implementar el procesamiento previo de los datos en los gateways, reduciendo así drásticamente el ancho de banda en

la red. Con la forma más "natural" de despliegue de Kaa también se podría conseguir esta funcionalidad pero sería cada endpoint quien analice los datos y envíe a la nube. Esto implica una mayor latencia de red además de realizar un buen uso de los grupos de endpoints. De hecho, Kaa no habla de Edge Analytics con este despliegue [38].
Se plantea para versiones posteriores de Kaa el uso de herramientas de análisis de datos provenientes de los endpoints para su posterior envío al servidor, añadiendo así un nivel de jerarquía en la estructura de Kaa.

7.6 Conclusiones

Tras el análisis de MQTT como middleware, y OpenIoT y Kaa como plataformas, hemos optado por elegir Kaa como plataforma para desplegar el caso de estudio planteado. Aunque se planteó al inicio la posibilidad de crear una jerarquía a distintos niveles, hemos optado por eliminar esa jerarquía ya que Kaa no ofrece ese soporte, pero por contra, ofrece requisitos muy interesantes como la interoperabilidad entre dispositivos, o el uso de mecanismos de transporte de menor consumo para dispositivos con un nivel de procesamiento menor.

8 Caso de estudio

8.1 Ejemplos prácticos

Primero se probaron los ejemplos presentes en la versión de VirtualBox preconfigurada de Kaa. Estos ejemplos me han ayudado a comprender el funcionamiento de Kaa en mayor profundidad y validar de manera práctica algunos de los requisitos que se plantean en este documento para un middleware IoT.

8.1.1 Configuración de datos

En este ejemplo se muestra el funcionamiento del proceso de configuración de datos en Kaa. El esquema de configuración para este ejemplo queda así:

```
{
        "type": "record",
        "name": "Configuration",
        "namespace": "org.kaaproject.kaa.schema.sample",
        "fields": [
                {
                        "name": "samplingPeriod",
                        "type": "int",
                        "by_default": "1"
                }
        ]
}
```

En él hay un valor por defecto para el valor "samplingPeriod". Cuando el cliente Kaa se inicie, se conectará con el servidor y recibirá la información por defecto del grupo "All", grupo al que pertenecen todos los endpoints (al no haber ningún grupo más definido).
Si se cambia la configuración del grupo "All", el endpoint recibirá automáticamente la nueva configuración y éste la usará para cambiar el periodo de muestreo.

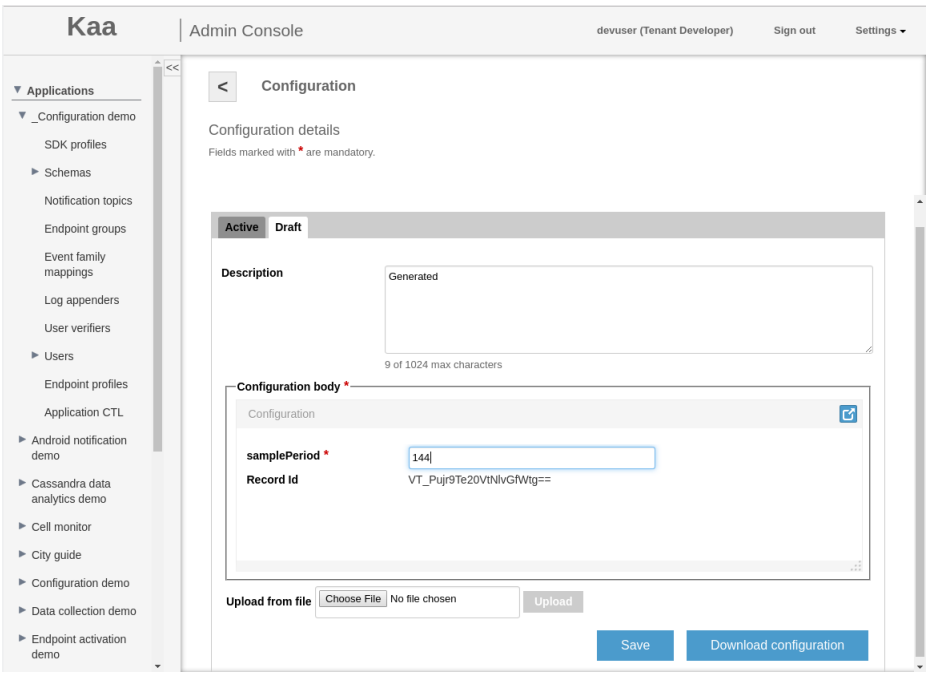


Fig 8.1. Cambio de configuración por defecto

[pool-4-thread-1] INFO o.k.k.d.c.ConfigurationDemo - Configuration was updated [pool-2-thread-1] INFO o.k.k.d.c.ConfigurationDemo - Sampling period is now set to 144 seconds

Con la creación de grupos de endpoints, se podría cambiar la configuración para un grupo de endpoints en concreto, útil cuando se tengan endpoints realizando funciones diferentes. La creación de grupos se verá en los siguientes ejemplos.

8.1.2 Colección de datos

En este ejemplo se demostrará la colección de datos y el posterior almacenamiento en Kaa. En este caso se definen los esquemas de datos ("log schema"). Para esta aplicación se usa el siguiente esquema:

```
{
        "type" : "record",
        "name" : "DataCollection",
        "namespace" : "org.kaaproject.kaa.schema.sample",
        "fields" : [
                {
                        "name" : "temperature",
                        "type" : "int"
                },
                {
                        "name" : "timeStamp",
                        "type" : "long"
                }
```

```
        ],
}
```

Los valores a almacenar son "temperature" y "timestamp". El esquema de configuración es parecido al del ejemplo anterior. El período de muestreo será usado para subir los datos a Kaa.
Los datos se envían desde el servicio de operaciones a la base de datos. Para el almacenamiento en bases de datos, se necesita configurar log appender. En este ejemplo se usará MongoDB como base de datos, por lo que el esquema tiene que estar basado en MongoDB.
Una vez compilado y ejecutado el SDK en la Raspberry, se envían datos aleatorios simulando un sensor de temperatura. La frecuencia de subida la marca el campo "samplePeriod". Los datos se van almacenando en la base de datos MongoDB, acorde al esquema "log appender".

8.1.3 Perfiles y grupos

En este ejemplo se muestra cómo funciona Kaa con la agrupación de diferentes endpoints en grupos así como los distintos filtros entre los endpoints. 7.5.2
Para este caso he usado Java como plataforma. En este ejemplo hay 3 instancias de un cliente Kaa y cada una de ellas tiene un perfil endpoint (EP). LA estructura de datos de un EP se define mediante un esquema. En este ejemplo se usa el siguiente esquema en el lado del cliente:

```
{
        "type" : "record",
        "name" : "PagerClientProfile",
        "namespace" : "org.kaaproject.examples.pager",
        "fields" : [ {
                "name" : "audioSupport",
                "type" : "boolean"
        }, {
                "name" : "videoSupport",
                "type" : "boolean"
        }, {
                "name" : "vibroSupport",
                "type" : "boolean"
        }]
}
```

Para el lado del servidor se usa el siguiente esquema:

```
{
        "type" : "record",
        "name" : "PagerServerProfile",
        "namespace" : "org.kaaproject.examples.pager",
        "fields" : [ {
                "name" : "audioSubscription",
                "type" : "boolean"
        }, {
                "name" : "videoSubscription",
                "type" : "boolean"
        }, {
                "name" : "vibroSubscription",
                "type" : "boolean"
```

```
        } ]
}
```

Y como esquema de configuración se usa:

```
    {
        "type" : "record",
        "name" : "PagerConfiguration",
        "namespace" : "org.kaaproject.examples.pager",
        "fields" : [ {
            "name" : "audioSubscriptionActive",
            "type" : "boolean",
            "by_default" : false
        }, {
            "name" : "videoSubscriptionActive",
            "type" : "boolean",
            "by_default" : false
        }, {
            "name" : "vibroSubscriptionActive",
            "type" : "boolean",
            "by_default" : true
        } ]
    }
```

Hay 4 grupos de endpoints: **Audio y vibración**, **Solo vibración**, **Todos los servicios soportados**, **Desactivar vibración si hay soporte para audio y vídeo**.
Como se menciona en la sección 7.5.2, para agrupar a los endpoints en grupos se necesita definir filtros para esos grupos. En este caso se han definido filtros asociados con los perfiles de EP en el lado del cliente:

```
% Filtros para el grupo de "Audio y Vibracion"
#cp.audioSupport == true && #cp.videoSupport == false && #cp.vibroSupport ==
    true

// "Solo vibracion"
#cp.audioSupport == false && #cp.videoSupport == false &&
     #cp.vibroSupport == true

// "Todos los servicios soportados"
#cp.audioSupport == true && #cp.videoSupport == true &&
     #cp.vibroSupport == true

// "Desactivar vibracion si hay soporte para audio y video"
#cp.audioSupport == true && #cp.videoSupport == true &&
     #cp.vibroSupport == true
```

Kaa obtiene la lista de los grupos de endpoints de la aplicación y comprueba los filtros de los perfiles de los endpoints tanto del lado del ciente como del lado del servidor. Esta comprobación se hace empezando por el grupo de menor peso (el grupo por defecto 'all') hasta el de mayor peso. Si los perfiles de los EP (definidos mediante los esquemas de client-side y server-side), cumplen las condiciones de los filtros, Kaa asigna el endpoint al grupo. Como resultado, el servidor aplica

la configuración del grupo, presente en los esquemas.
Para este ejemplo, la configuración por defecto en el lado del servidor para cada endpoint es tener audio y video desactivado y vibración activada. Al endpoint 2, en el lado del cliente se le asigna audioSupport=true, videoSupport=true y vibrationSupport=true. Como para este ejemplo, los filtros tan solo actúan en el lado del cliente, el endpoint #2 pertenecerá por defecto al grupo "Todos los servicios habilitados". Por ello, al iniciar la aplicación, se muestra el mensaje de que el endpoint #2 pertenece a dicho grupo.
Cambiando desde la interfaz el filtro del grupo "Desactivar vibración si hay soporte para audio y vídeo", a la misma configuración que el grupo "Todos los servicios soportados", la consola muestra el mensaje de que la configuración ha sido actualizada. El endpoint 2 cambia su configuración al nuevo grupo ya que, aunque tenga la misma configuración, el grupo tiene un mayor peso.

8.1.4 Chat mediante eventos

En este caso se ha usado Android como plataforma. Este ejemplo consiste en emular un chat mediante los eventos de Kaa.
Los eventos se definen mediante los esquemas de clases de eventos, Event Class Schema (EC), en inglés. A su vez, estas clases de eventos se agrupan en familias de clases de eventos, Event Class Family (ECF), en inglés. En ese ejemplo, se han creado dos esquemas de clases de eventos: ChatEvent y Message.
Chat Event, envía un evento cuando el usuario quiere crear o eliminar un chat:

```
{
        "type": "record",
        "name": "ChatEvent",
        "namespace": "org.kaaproject.kaa.examples.event",
        "fields": [
                {
                        "name": "ChatName",
                        "type": {
                        "type": "string",
                        "avro.java.string": "String"
                        }
                },
                {
                        "name": "EventType",
                        "type": {
                        "type": "enum",
                        "name": "ChatEventType",
                                "symbols": [
                                "CREATE",
                                "DELETE"
                        ],
                        "classType": "object"
                        }
                }
        ],
        "description": "",
        "classType": "event"
}
```

Message envía un evento con el nombre del chat y el mensaje.

```
{
        "type": "record",
        "name": "Message",
        "namespace": "org.kaaproject.kaa.examples.event",
        "fields": [
                {
                        "name": "ChatName",
                        "type": {
                        "type": "string",
                        "avro.java.string": "String"
                        }
                },
                {
                        "name": "Message",
                        "type": {
                          "type": "string",
                          "avro.java.string": "String"
                         }
                }
        ],
        "description": "",
        "classType": "event"
}
```

Estos pasos previos se hacen desde la interfaz con el usuario con rol de administrador. Una vez creadas las familias de clases de eventos, se necesita asignarlas a la aplicación. Esto se hace mediante la opción "Event family mapping" de la interfaz (con rol de desarrollador).
Por último, hay que añadir un verificador de usuario para comprobar los credenciales de los usuarios. Los endpoints pueden intercambiar mensajes entre aquellos que pertenezcan al mismo usuario. Por defecto, se usa un verificador de tipo 'Trustful', que acepta cualquier credencial de usuario. Se pueden añadir varios verificadores a la misma aplicación. Cuando se genera el SDK, se elige el verificador requerido.
La app envía un mensaje, como un evento Kaa, a todos los endpoints unidos al mismo chat.

Descubrimiento de recursos en Kaa haciendo uso de los eventos

Haciendo uso del ejemplo anterior, se va a crear una sencilla aplicación para demostrar el requisito de descubrimiento de recursos en Kaa. En este escenario se envía un evento a todos los endpoints cuando cada uno de ellos inicie la aplicación, mostrando así su presencia al resto de endpoints.
Al iniciar la aplicación, se envía un evento a todos los endpoints, mostrándose como una notificación en Android. Este mecanismo puede ser utilizado por los endpoints para enviar su configuración o información sobre qué servicios ofrecen y ser capturado por el resto de endpoints.
Requisitos comprobados: Descubrimiento de recursos.

Detección de la caída de un nodo haciendo uso de los eventos

Kaa puede cumplir con el requisito de control de recursos. Y es que Kaa no implementa un sistema de control de recursos como tal, pero haciendo uso de su sistema de eventos y notificaciones se podría conseguir tal propósito.
El desarrollador implementa toda la funcionalidad en los nodos. En este caso se ha implementado una función 'Ping' en uno de los endpoint y a la que, haciendo uso de los eventos, debe responder el resto de endpoints en un determinado tiempo. Si no lo hacen, se considera como endpoint caído y el nodo podría actuar ante tal propósito, por ejemplo enviando la información a un nodo, conectado a la misma red que el endpoint caído, y, mediante sistemas de control y monitorización, intentar solucionar el problema.
En el escenario que se plantea se han usado dos SDK en Java, instalados en una Raspberry y en un PC, y un tercer SDK en Android, que recibirá la información en caso de fallo en alguno de ellos.

- **Configuración en el lado del servidor:** Para este escenario tan solo es necesario configurar los eventos en el lado del servidor. Se crean 2 clases de familias de eventos (ECF), una clase para el envío y recepción del ping y otra para el aviso de fallo en un endpoint. Esto último también se podría haber hecho usando notificaciones. Las clases Ping y Alarm se pueden ver en la figura.../// La clase de eventos Ping tiene un campo String para almacenar el Hash del Endpoint como identificador, un campo Long para almacenar el tiempo de envío y un campo String opcional para enviar si el Ping es de petición o respuesta. Por otra parte, la clase Alarm tan solo tiene un campo String que contendrá el mensaje.
- **Configuración en el lado del cliente:** El endpoint principal, llamémosle A, implementa la función Ping y realiza la comprobación del tiempo enviado por cada endpoint.
  A esta función tienen que responder los endpoints pasándole su hash y el tiempo actual. Si el endpoint A detecta una diferencia de tiempo entre el actual y el enviado, para cada endpoint, mayor a un umbral, enviará un evento notificando del mal funcionamiento al endpoint con el SDK en Android.

```
//response ping
pecf.addListener(new PingEventClassFamily.Listener() {

 @Override
 public void onEvent(Ping event, String source) {
     System.out.println("event ping request: "  + event.getMessage());
     if(event.getMessage().equalsIgnoreCase("request")){
         pecf.sendEventToAll(new Ping(kaaClient.getEndpointKeyHash(), System.currentTimeMillis(), "response"));
     }

 }
});
```

(a) Código de recepción de respuesta

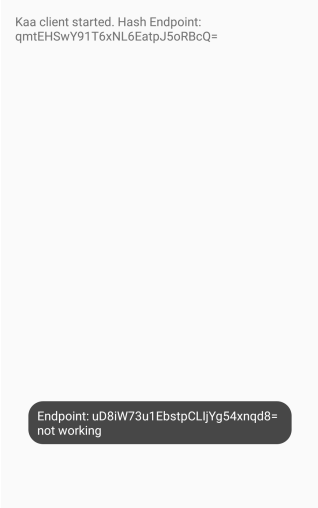
```
scheduledFuture = scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {

        long difference;
        for (HashMap.Entry<String, Long> entry : lists.entrySet()) {
            difference = System.currentTimeMillis() - entry.getValue();
            if (difference > 20000) {
                aecf.sendEvent(new AlarmCTL("Endpoint: " + entry.getKey +
                    " not working"), endpointKeyHash);
            }

        }
    }
}, 0, 30, TimeUnit.SECONDS);
```

(b) Código de evaluación del tiempo para cada endpoint



(c) Recepción de evento en Android

Fig 8.2. Imágenes de configuración y visualización en el escenario planteado

Requisitos comprobados: Control de recursos, interoperabilidad, escalabilidad.

Desplegando Kaa como la arquitectura mostrada en la figura 7.3, se podrían conseguir los requisitos de control y descubrimiento de recursos, desarrollando el gateway para tal propósito. Esto es especialmente útil cuando los endpoints sean dispositivos que no puedan ejecutar el SDK por distintos motivos o cuando hay cientos de miles de dispositivos conectados, ya que los sistemas antes comentados provocarían una mayor latencia en la red.

8.1.5 Activación de actuadores tras recopilación y análisis de datos.

Se plantea el siguiente escenario: una Raspberry recibiendo información de un sensor de temperatura y enviando esos datos a Kaa para su almacenamiento. Una vez enviados esos datos, se usa Apache Zeppelin para analizar los datos y, si supera un umbral, se envía una notificación a la Raspberry haciendo uso de Kaa. Con esta notificación se logrará la activación de un actuador, en este ejemplo representado por un LED. Esta funcionalidad podría emplearse para, por ejemplo,

recoger datos de cientos de sensores en un escenario de Smart Farming y dar órdenes de control del riego o fertilizante en función de las condiciones atmosféricas y otros parámetros.

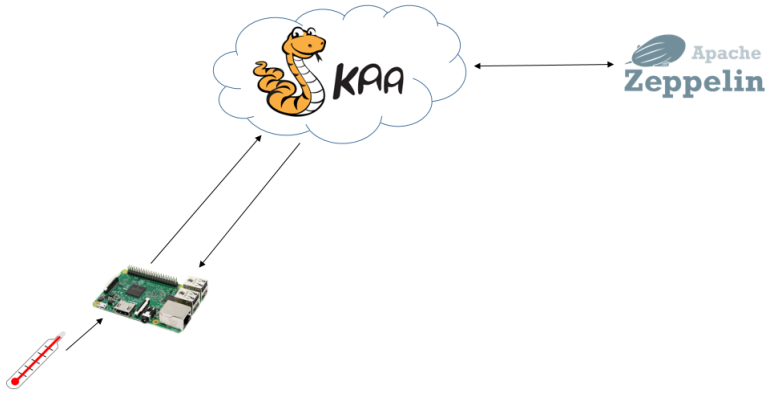


Fig 8.3. Escenario planteado en Kaa

Creación de los esquemas y la aplicación

Todos las pruebas se han hecho en la versión virtual que ya viene preconfigurada.
Para crear una nueva aplicación, el usuario tiene que tener el rol de Tenant Administrador, por ello hay que 'logearse' como admin/admin123. Una vez que se asigna un nombre a la aplicación, hay que logearse de nuevo pero esta vez con el rol de desarrollador (devuser/devuser123).
El funcionamiento de la aplicación será enviar datos a la Raspberry cada cierto tiempo, tiempo definido por el desarrollador en los esquemas de datos. El esquema de configuración queda así:

```
{
      "type": "record",
      "name": "Configuration",
      "namespace": "org.kaaproject.kaa.schema.sample",
      "fields": [
          {
              "name": "samplePeriod",
              "type": "int",
              "by_default": 1
          }
      ]
 }
```

En él se crea el periodo de tiempo en el que se suben los datos, que por defecto es 1. El esquema de datos indica los datos que se envían a Kaa para que sean almacenados en la base de datos:

```
{
      "type": "record",
      "name": "DataCollection",
```

```
    "namespace": "org.kaaproject.kaa.schema.sample",
    "fields": [
        {
            "name": "temperature",
            "type": "int"
        }
    ]
}
```

Se va a usar MongoDB como base de datos. Para el almacenamiento en las bases de datos hay que definir el esquema 'log appender'. Se definen los metadatos que se añadirán al valor de la temperatura. También hay que añadir el valor "keyspace" y el nombre de la tabla que serán usados por cassandra para el almacenamiento. El esquema 'Log Appender' queda como se muestra en la siguiente figura.

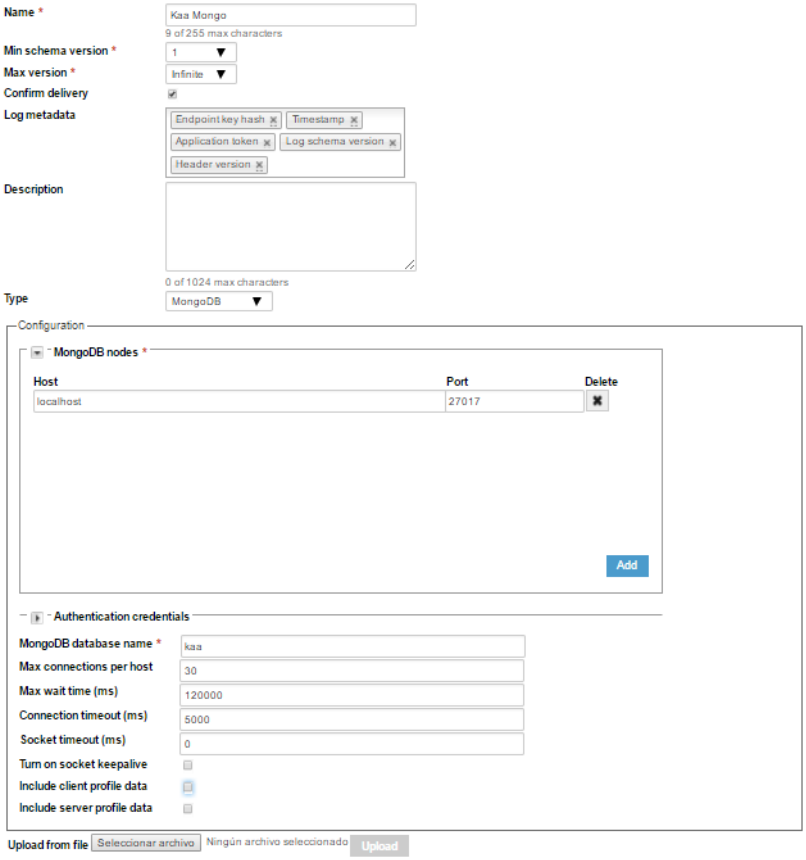


Fig 8.4. Esquema utilizado para el almacenamiento en MongoDB

Por otro lado, para el uso de las notificaciones se tienen que definir un esquema para tal fin. El esquema definido para este ejemplo es el siguiente:

```
{
  "type" : "record",
  "name" : "ExampleNotification",
  "namespace" : "org.kaaproject.kaa.schema.sample.notification",
  "fields" : [ {
    "name" : "message",
    "type" : {
      "type" : "string",
      "avro.java.string" : "String"
    }
  } ],
}
```

Enviar datos desde la Raspberry a Kaa

Una vez configurado el lado del servidor y generado el SDK, se necesita programar el endpoint. El lenguaje del SDK generado será C, un lenguaje compatible con la plataforma (Ver tabla 7.1).
El endpoint se programa para crear el cliente SDK, cargar la configuración definida en el servidor y poder recibir notificaciones. También hay que añadir la parte de código para encender un Led en la Raspberry y recibir información de parte del sensor de temperatura. Esto último se ha realizado generando datos de forma aleatoria, simulando un sensor real.

Analizar datos con Apache Zeppelin

Una vez enviados los datos a la plataforma, se tienen que analizar de manera externa a ella con herramientas de análisis de datos. Para ello, se ha instalado Apache Zeppelin [34] en el servidor ya preconfigurado. Zeppelin estará escuchando conexiones por uno de los puertos. Zeppelin está formado por diferentes intérpretes, cada uno de los cuales ejecutar el código en el lenguaje de dicho intérprete. MongoDB no es un intérprete por defecto de Zeppelin pero se puede añadir mediante un plugin. Una vez añadido el intérprete, se envía una petición a través de la cual se conecta y obtiene los datos almacenados.

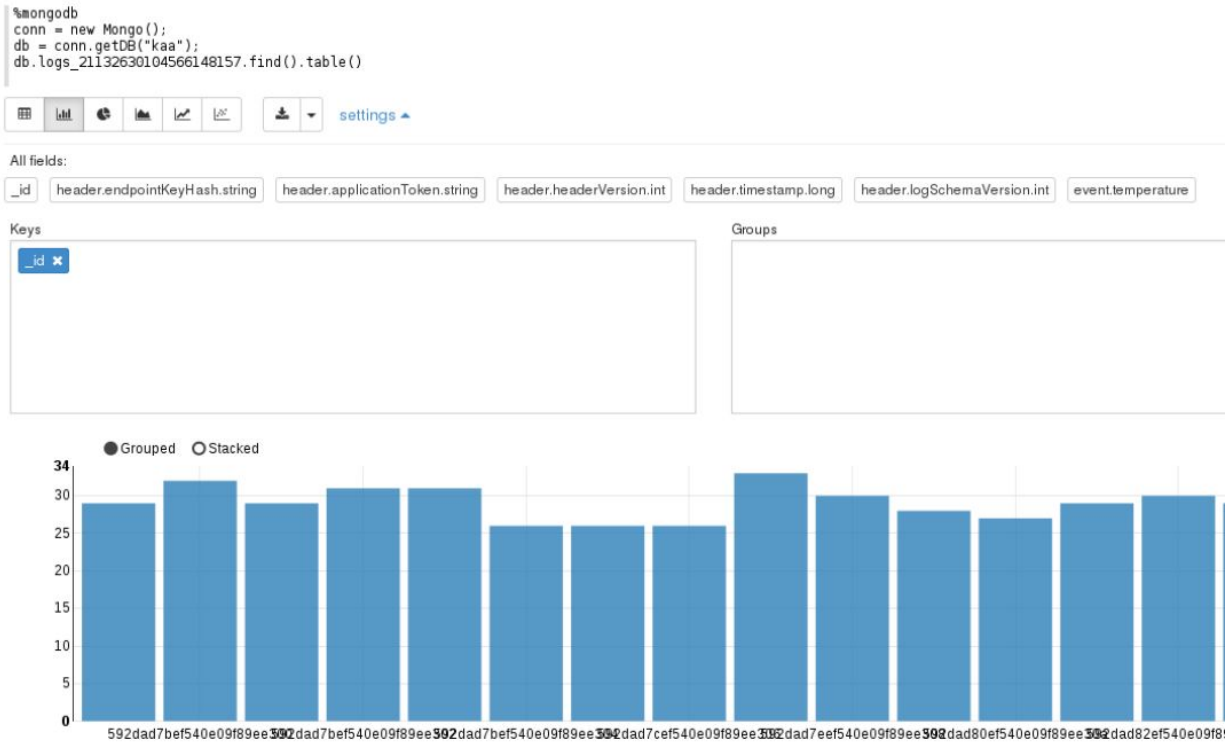


Fig 8.5. Obtención de datos de MongoDB

Envío de notificaciones a la Raspberry para encender un LED

Una vez analizados los datos, se envía una notificación a la Raspberry para encender un LED. Para tal propósito se usan las notificaciones en Kaa. La notificación se envía mediante una llamada a la API y, una vez recibida por Kaa, ésta la envía a la Raspberry, previemante subscrita al tópico, de acuerdo al esquema de notificaciones previamente configurado. La llamada a la API /sendNotification tiene el siguiente formato:

```
curl -v -S -u devuser:devuser123 -F'notification={"applicationId":"32768","
    schemaId":"65536","topicId":"32769","type":"USER"};type=application/json'
    -F file=@notification.json
"http://192.168.0.107:8080/kaaAdmin/rest/api/sendNotification" | python -
    mjson.tool
```

Donde el archivo notification.json tiene los datos a enviar en la notificación.
Una vez recibida en la Raspberry, ésta encenderá el LED. En este sencillo ejemplo se demuestra el uso de las notificaciones en Kaa, así como el almacenamiento y análisis de datos en herramientas externas.

Requisitos comprobados: Control de datos, tiempo real, abstracción de la programación

8.2 Ejemplo de caso de estudio para Smart Farming

A pesar de que la agricultura ha sido siempre un entorno en el que la innovación tecnológica ha tardado en llegar, están apareciendo nuevas soluciones para este entorno. Soluciones como el uso de drones para controlar un cultivo o la automatización de un terreno agrícola que pueda ser controlado desde un Smartphone, son cada vez más comunes. No obstante, en Estados Unidos se estima que entre el 20 y 80 % de la comunidad agrícola utiliza este tipo de soluciones, por el contrario en Europa se estima que las utilizan entre un 0 y un 24 % (Fuente: SmartAKIS [35]).
Smart Farming o la aplicación de las TIC a la agricultura, está cambiando el panorama del sector agrícola a través de soluciones como el IoT. Esto ayuda a una gestión de las explotaciones más eficiente, desde un mejor uso del agua hasta la optimización de los tratamientos fitosanitarios. Este tipo de soluciones recopilan datos por parte de sensores situados en el cultivo para su posterior análisis y toma de decisiones de forma automática.
En la figura 8.6 se puede ver la estimación de la cantidad de datos que se generarán al día (de media) en la agricultura, realizada en 2015 por BI Intelligence.

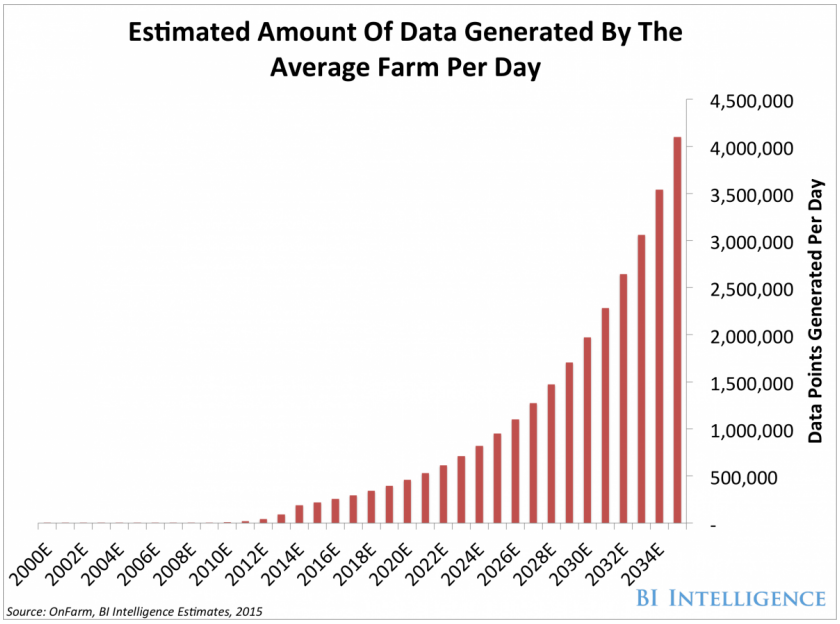


Fig 8.6. Estimación de la cantidad de datos generados en la agricultura.

Como caso de estudio final llevamos a la práctica el uso de Kaa al escenario agrícola planteado al comienzo de este TFG (eliminando, como ya se mencionó en capítulos anteriores, la

arquitectura multijerárquica pensada inicialmente). Se ha intentado realizar un ejemplo lo más completo posible, que incorpore todas las funcionalidades descritas en los ejemplos previos y además se asemeje en la mayor medida posible a una situación realista. Para la monitorización del terreno agrícola habrá sensores de humedad, temperatura, etc. Estos datos se subirán a la plataforma para ser almacenados y analizados; y en función de los resultados se enviarán notificaciones a nodos actuadores para, por ejemplo, abrir o cerrar válvulas.
Para no extender en el tiempo este trabajo fin de grado y hacerlo lo más funcional posible, resumimos el escenario al de la figura 8.7. En él, los diferentes actuadores (como por ejemplo las válvulas de riego) serán ejemplificados mediante LEDs. Se utilizarán sensores de luz, temperatura y humedad que ejemplificarán los que se podrían emplear en un escenario agrícola para monitorizar las condiciones ambientales, del suelo y de los cultivos.

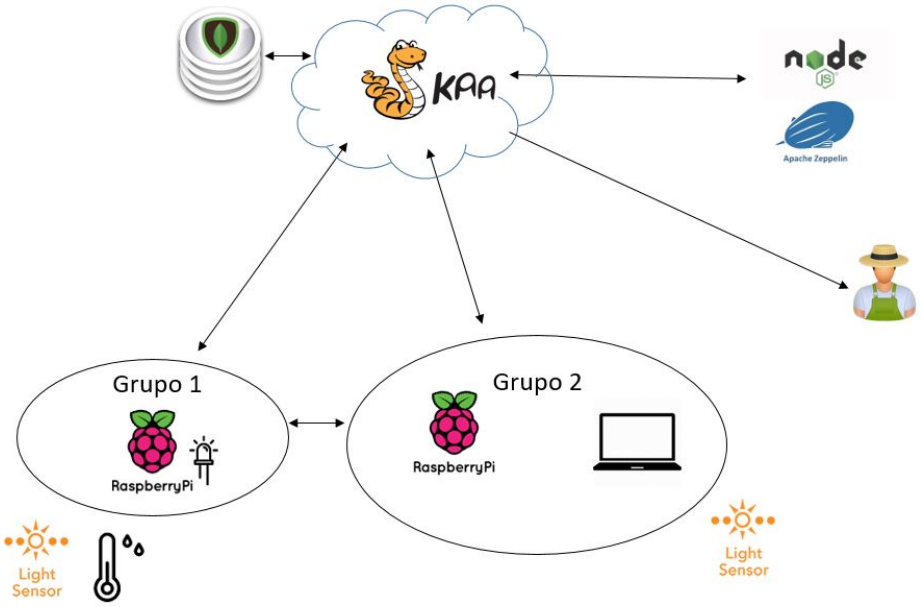


Fig 8.7. Escenario planteado.

A nivel más bajo, se sitúan 2 Raspberry y un PC que actuará como endpoint, y así demostrar la interoperabilidad entre dispositivos. Los endpoints se han agrupado en 2 grupos, uno conteniendo a una de las Raspberry y el otro a los 2 endpoints restantes. La Raspberry del grupo 1 lee datos de los sensores de luz, temperatura y humedad. La Raspberry restante tan solo recopila datos de un sensor de luz. El PC simula la lectura de datos de un sensor de luz.
Los datos del endpoint del grupo 1 se suben a Kaa para ser almacenados, en un intervalo de tiempo diferente a los del grupo 2. Este intervalo se define para cada grupo en el servidor. Los endpoints del grupo 2, al iniciar, envían información sobre ellos y datos recopilados por sus sen-

sores a la Raspberry del grupo 1. Además, esta Raspberry envía cada cierto tiempo un evento a los endpoints restantes para que éstos respondan en un determinado tiempo. Si no lo hacen, se considerarán como endpoints caídos y así poder actuar ante ello. Esta función también se usa como descubrimiento de nuevos dispositivos por parte de la Raspberry del grupo 1.
En el lado del servidor, se definen los esquemas para la configuración, notificaciones, eventos y almacenamiento de datos. La configuración del lado del servidor se puede ver en el anexo A. Como base de datos se ha elegido MongoDB para el almacenamiento de datos. Se usa Apache Zeppelin para la visualización de datos y nodejs para el análisis de datos. En nodejs se ejecuta un script que, cada cierto tiempo, obtiene los datos de MongoDB y analiza si un determinado valor, en este caso, el leído por el sensor de temperatura, supera un umbral. En caso de que lo supere, se envía una notificación vía REST a Kaa para que la reciba la Raspberry del grupo 1. Cuando la reciba, encenderá un LED y enviará un correo al agricultor para avisar del problema. Este correo lo puede recibir el agricultor para que tenga constancia del problema. Debido a la interoperabilidad de Kaa, se podría crear una aplicación móvil que ejecute el SDK en Android, siendo así un endpoint más, y recibir la notificación en la App. Esta App aparte de recibir notificaciones, podría mostrar un mapa del terreno con la situación exacta del endpoint así como recibir datos de los cultivos en tiempo cuasi-real.
Como lenguajes de programación se han usado Java para los endpoints, Python para obtener los datos de los sensores en la Raspberry y nodejs para la conexión con la base de datos y su posterior procesado de los datos. En el anexo 2 se puede ver algunas porciones de código usadas para desplegar toda la arquitectura, tanto en el lado del cliente (SDKs) como en el lado del servidor (NodeJS, Zeppelin).
Este escenario ha sido 100 % funcional en el que se ha demostrado requisitos como descubrimiento de recursos, control de datos y recursos, interoperabilidad o seguridad y privacidad. Esto se puede trasladar a una situación real, en la que existen cientos de nodos en el cultivo recopilando datos de sensores. La escalabilidad de Kaa permitirá añadir cientos de nodos tan solo compilando el SDK. Cada uno de estos nodos tendría embebido un SDK en un determinado lenguaje que le permitirá comunicarse con el servidor Kaa, situado fuera del cultivo (en la nube) y controlado por el desarrollador. Esta automatización del cultivo le permite al agricultor controlar los datos del cultivo o visualizarlos de forma gráfica. Por otra parte, cualquier fallo en uno de los nodos o en la lectura de un cultivo podría ser notificado en tiempo cuasi-real tanto al desarrollador como al agricultor. Para una mayor automatización, se podría desarrollar un nodo ajeno a Kaa que actúe ante la caída de un nodo, por ejemplo: intentando restablecer la conexión ante un fallo en la red. Como se puede ver, es un entorno con muchas y útiles posibilidades en el que su aplicación en un entorno real es 100 % funcional. Muestra de ello son las diferentes soluciones creadas por la empresa KaaIoT.

9 | Conclusiones y trabajos futuros

9.1 Conclusiones

A lo largo de este trabajo se ha realizado una introducción al paradigma del IoT, indicando sus principales características y las barreras que dificultan su desarrollo. A continuación se han introducido algunos protocolos y middleware para IoT, y se ha descrito secuencialmente el proceso seguido hasta dar con la plataforma óptima para desplegar nuestro escenario de Smart Farming: análisis de MQTT, análisis de OpenIoT y análisis de Kaa. Con MQTT y la ayuda de otras herramientas se puede obtener una solución para un entorno IoT pero es el desarrollador quien tiene que crear esa solución mediante la integración de diferentes servicios y no una plataforma que integre esos servicios, con todo lo que ello conlleva; Empresas líderes en el ámbito de las telecomunicaciones están trabajando para desarrollar soluciones completas y siempre será más eficiente (seguridad, facilidad, menor tiempo de desarrollo...) que si es el propio desarrollador quien tiene que crearla.
En este punto, se comenzaron a analizar algunas plataformas-middleware IoT para conocer cuál sería la que más se adapta a nuestro caso. Finalmente, se han desarrollado una serie de ejemplos prácticos para probar las funcionalidades de Kaa y se ha desplegado el escenario originalmente planteado de la forma más fiel que ha sido posible. Se han satisfecho algunos requisitos como control de recursos, seguridad, interoperabilidad, escalabilidad, control de código, descubrimiento de recursos, control de datos o disponibilidad. Por otro lado, no se ha podido implementar la arquitectura multijerárquica, que queda pendiente para futuros trabajos.
En definitiva, se puede concluir que Kaa es una plataforma con una curva de aprendizaje bastante alta pero con características que la hacen una plataforma a tener en cuenta en el IoT. Con Kaa, el programador se olvida de la parte del servidor centrándose solo en la parte del cliente, facilitándole así el proceso. Kaa cuenta con un gran soporte y actualización; a fecha de publicación de este documento hay disponible, para desarrolladores, una versión beta de Kaa (1.0.0) con nuevas características, cuya versión final estará disponible en un futuro próximo.

9.2 Trabajos futuros

Como continuación de este trabajo, y como cualquier otro con una demanda y actualidad como es el caso del IoT, existen varias líneas de desarrollo que quedan abiertas y en las que es posible continuar trabajando. Entre los posibles trabajos futuros se destacan:

- Seguir con el análisis de otras plataformas Open-Source, comprobando los requisitos que cumplen para diferentes escenarios IoT.
- Realizar empleando Kaa una integración con otros servicios, como puedan ser servicios Web (e.g. NodeJS + MongoDB + AngularJS) o una aplicación móvil, de tal modo que sea más sencillo para el usuario analizar el sistema, desde un punto de vista gráfico.
- Desplegar Kaa con gateways, pudiendo así usar protocolos de menor consumo como MQTT u obtener funcionalidades como el descubrimiento de recursos.
- Añadir nuevas funcionalidades al escenario agrícola planteado como por ejemplo, las características de edge y fog computing.
- Analizar las características que ofrece la nueva actualización de Kaa e intentar aplicarlas al escenario para obtener nuevas funcionalidades o mejoras.
- Estudiar algún modo de implementar la arquitectura multijerárquica planteada en el escenario inicial, ya sea con Kaa u otra plataforma.
- Desde un punto de vista práctico, realizar el despliegue de los nodos en un terreno real y analizar su rendimiento y aspectos de mejora.

Anexos

A | Configuración en el lado del servidor

Este anexo presenta la configuración llevada a cabo en el lado del servidor para el caso de estudio. Se describirán todos los esquemas así como el proceso de configuración del server.

A.1 Esquema de configuración

Este esquema se usa para determinar el periodo con el que se obtienen y se suben al servidor los datos de los sensores. Cambia dependiendo del grupo de endpoints. Por defecto es 60.

```
{
  "type" : "record",
  "name" : "configuration",
  "namespace" : "org.farming.configuration.sample",
  "fields" : [ {
    "name" : "sample",
    "type" : "int",
    "by_default" : 60
  } ],
  "version" : 1,
  "dependencies" : [ ],
  "displayName" : "SamplePeriod"
}
```

A.2 Esquema de datos

Este esquema se usa para la colección de datos en una estructura definida. En este caso se ha usado un campo de tipo "Union" que contiene el tipo de datos de temperatura, humedad y luz, que será usado por los endpoints para enviar sus datos. Con esta definición, se permite a cada endpoint, elegir qué datos subir. El otro campo es "status", usado por nodejs para el análisis de los datos.

```
{
  "type" : "record",
  "name" : "MeasurementsData",
  "namespace" : "org.farming.sensors.measurements",
  "fields" : [ {
    "name" : "Measurements",
    "type" : [ {
      "type" : "record",
      "name" : "TemperatureData",
      "fields" : [ {
        "name" : "temperature",
        "type" : "long"
      } ],
      "displayName" : "TemperatureData",
      "description" : "Measurements from temperature sensor"
    }, {
      "type" : "record",
      "name" : "HumidityData",
      "fields" : [ {
        "name" : "humidity",
        "type" : "long"
      } ],
      "displayName" : "HumidityData",
      "description" : "Measurements from humidity sensor"
    },{
      "type" : "record",
      "name" : "LightingData",
      "fields" : [ {
        "name" : "sunlight",
        "type" : "long"
      } ],
      "displayName" : "LightlingData",
      "description" : "Measurements from light sensor"
    } ],
    "displayName" : "Measurements"
  }
  ],
  "displayName" : "MeasurementsData",
  "description" : "Measurements from sensors"
}
```

A.3 Esquema de notificaciones

Esquema usado para definir los campos de las notificaciones.

```
{
    "type":"record",
    "name":"NotificationSchema",
    "namespace":"org.farming.notifications",
    "fields":[
        {
            "name":"message",
            "type":"string"
        },
                {
                        "name":"endpointHash",
            "type":"string"
                }
```

```
    ],
        "displayName" : "NotificationsSchema"
}
```

Una vez definido el esquema, se deben crear los tópicos para las notificaciones y así recibir notificaciones de un determinado tópico.

A.4 Esquema del lado del cliente

Este esquema se usa para el agrupamiento de endpoints. Con este esquema se define el perfil a usar en el lado del cliente y, posteriormente, se define cada grupo mediante el filtro de perfil. Los grupos se hacen simulando parcelas de cultivo.

```
{
  "type" : "record",
  "name" : "ClientProfile",
  "namespace" : "org.farming.profile.client",
  "fields" : [ {
    "name" : "plot",
    "type" : {
      "type" : "string"
    },
    "displayName" : "Parcel of land"
  } ]
}
```

A.5 Esquemas de eventos

En este caso se definen 2 esquemas de familias de eventos, uno para una función "ping" a la que tienen que responder el resto de endpoints, y el otro, para enviar información recopilada por los sensores y del sistema, al iniciar el endpoint por primera vez.

Listing A.1. Esquema de eventos de la clase Ping

```
{
  "type" : "record",
  "name" : "PingEventClass",
  "namespace" : "org.farming.event.ping1",
  "fields" : [ {
    "name" : "currentTime",
    "type" : "long"
  }, {
    "name" : "hashEndpoint",
    "type" : {
      "type" : "string"
    }
  }, {
    "name" : "message",
    "type" : [ {
```

```
      "type" : "string"
    } ]
  } ],
  "displayName" : "Ping"
}
```

Listing A.2. Esquema de eventos de la clase SystemInformation

```
{
  "type" : "record",
  "name" : "SystemInformation",
  "namespace" : "org.farming.simulation.event.systemInformation",
  "fields" : [ {
    "name" : "systemInformation",
    "type" : {
      "type" : "string"
    },
    "displayName" : "",
    "displayPrompt" : ""
  }, {
    "name" : "dataSensor",
    "type" : [ {
      "type" : "array",
      "items" : "long"
    }, "null" ]
  } ]
  "displayName" : "System Information Event"
}
```

A.6 Grupos de endpoint

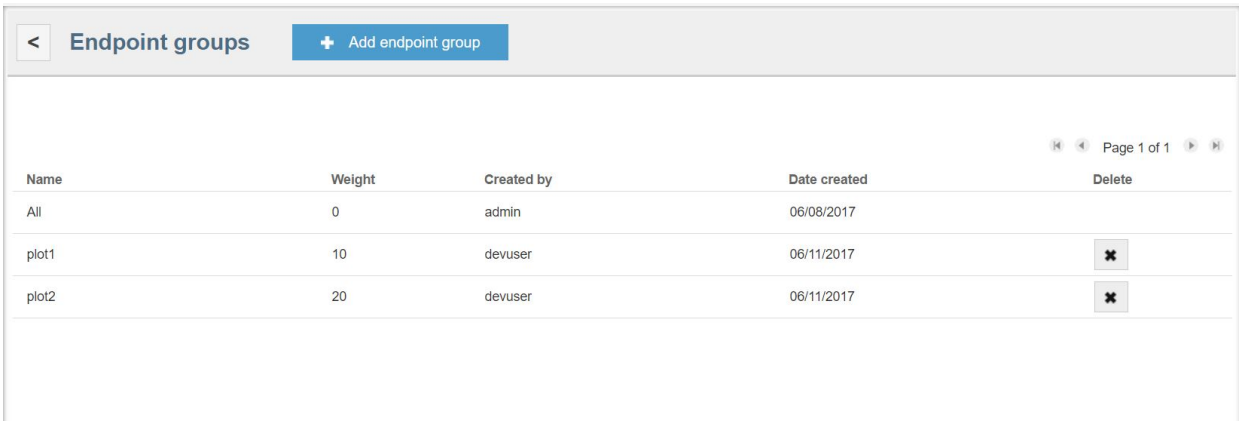


Fig A.1. Grupos de endpoint.

Una vez definido el esquema de perfil de endpoint en el lado del cliente, se pueden aplicar filtros para cada grupo y así asignar el endpoint a un grupo. El filtro, acorde al esquema, para el grupo 1 es: "#cp.plot==’1’" y para el grupo 2 : "#cp.plot==’1’"

En el grupo 1 se establece a 60 el periodo de muestreo y a 30 para el grupo 2. También se añade el tópico de notificación al grupo 1.

A.7 Mapeo de familias de eventos y verificador de usuario

Una vez creadas las familias de clases de eventos, es necesario añadirlas a la aplicación, mediante la opción "Event family mapping". También es necesario añadir un verificador de usuario. En este caso se ha añadido un verificador de tipo "trustful verifier", el cual acepta conexiones de cualquier endpoint.

A.8 NodeJS

En NodeJS se ejecuta un script que, cada cierto tiempo, obtiene los valores de MongoDB y comprueba si el campo de temperatura supera un umbral. Si lo hace, envía una notificación vía REST a Kaa.

Listing A.3. Script en NodeJS.

```
.function main() {


    var MongoClient = require('mongodb').MongoClient;
    var ObjectId = require('mongodb').ObjectID;
    var http = require("http");
    var time_min = 1000 * 60;
    var records = [];

    var fs = require('fs');
    var fileName = './notification.json';
    var file = require(fileName);

    MongoClient.connect("mongodb://192.168.178.44:27017/kaa_smartFarming"
        , function (err, db) {
        if (err) {
            return console.dir(err);
        } else {
            function getData() {
                db.collection('logs_09692608474505312390').find({
                    "event.Measurements.org. farming. sensors.
                        measurements. LightingData.sunlight": { $gt: 27}
                }).toArray(function (err, doc) {
                    console.log(doc);
```

```
                if (err) {
                    console.log(err);
                }
                if (doc.length > 0) {
                    if (doc[0].event.status.string !== 'in progress')
                        {
                        var n = doc[0].event.Measurements;
                        var keys = Object.keys(n);
                        var data = n[keys].sunlight
                        var endpoint = doc[0].header.endpointKeyHash.
                            string;
                        console.log("endpoint: " + endpoint);

                        var myquery = {
                            "event.Measurements.org. farming. sensors
                                . measurements. LightingData.sunlight"
                                : data
                        };
                        var newvalues = {
                            $set: {
                                "event.status.string": "in progress"
                            }
                        };
                        db.collection("logs_09692608474505312390").
                            update(myquery, newvalues, function (err,
                            res) {
                            if (err) throw err;
                            console.log(res.result.nModified + "
                                record updated");

                        });
                        file.endpointHash = new String(endpoint);;
                        fs.writeFile(fileName, JSON.stringify(file),
                            function (err) {
                            if (err) return console.log(err);
                            console.log(JSON.stringify(file));
                            console.log('writing to ' + fileName);
                            requestGet(function () {});
                        });

                    }
                }
            });
        }

        setInterval(getData,time_min);
    }

});
```

```
}
main(function () {});

function requestGet() {
    var fs = require('fs');
    var request = require('request');
    var crypto = require('crypto');
    var CRLF = "\r\n";
    var md5 = crypto.createHash('md5');

    function multipartRequestBodyBuilder(fields, boundary) {
        var requestBody = '';
        for (var name in fields) {
            var field = fields[name];
            var data = field.data;
            var fileName = field.fileName ? '; filename="' + field.
               fileName + '"' : '';
            var type = field.type ? 'Content-Type:' + field.type + CRLF :
                '';
            requestBody += "--" + boundary + CRLF +
                "Content-Disposition: form-data; name=\"" + name + "\"" +
                    fileName + CRLF +
                type + CRLF +
                data + CRLF;
        }
        requestBody += '--' + boundary + '--' + CRLF
        return requestBody;
    }

    function getBoundary() {
        md5.update(new Date() + getRandomArbitrary(1, 65536));
        return md5.digest('hex');
    }

    function getRandomArbitrary(min, max) {
        return Math.random() * (max - min) + min;
    }

    var notificationValue = {
        "applicationId": "163840", //131072
        "schemaId": "294919", //262146
        "topicId": "131072", //32768
        "type": "USER"
    };
    var postData = {
        notification: {
            data: JSON.stringify(notificationValue),
            type: "application/json"
        },
```

```
        file: {
            data: fs.readFileSync("notification.json"),
            fileName: 'notification.json',
            type: 'application/octet-stream'
        }
    }

    var boundary = getBoundary();

    var opts = {
        url: 'http://192.168.178.44:8080/kaaAdmin/rest/api/
           sendNotification',
        method: 'POST',
        auth: {
            user: 'devuser',
            password: 'devuser123'
        },
        headers: {
            'content-type': 'multipart/form-data; boundary=' + boundary
        },
        body: multipartRequestBodyBuilder(postData, boundary)
    };

    request(opts, function (err, resp, body) {
        if (err) {
            console.log("Error: " + err);
        } else {
            console.log("Satus code: " + resp.statusCode + "\n");
            console.log("Result: " + body);
        }
    });

    function data() {
    }
}
```

Listing A.4. "Archivo Notification.json."

```
{"message":"Temperature exceeded",
"endpointHash":"AQJpeSdcRYPk453ASUs0IvWh858="}
```

B | Configuración en el lado del cliente

En este anexo se presentará la configuración realizada en los SDKs (lado del cliente). Se describirá la configuración para la Raspberry principal del grupo 1 y para los endpoints del grupo 2, cuya configuración es idéntica.

B.1 Endpoints del grupo 1

La Raspberry del grupo 1, se considera como el endpoint principal, el cual recibe información del resto de endpoints y la respuesta de una función ping cada cierto tiempo por parte del resto de endpoints.

Listing B.1. Código en Java del endpoint 1.

```
public class SmartFarming {
    private static final long DEFAULT_START_DELAY = 1000L;
    private static final Logger LOG = LoggerFactory.getLogger(
       SmartFarming_raspi3.class);
    private PingCF pcf;
    private SystemInformationEventClassFamily siecf;
    private HashMap<String, Long> check_time = new HashMap<>()
       ;
    private static KaaClient kaaClient;
    private String hashMainEndpoint = "";
    private String userId = "farming";
    private String userToken = "farming";
    private BasicNotificationListener_raspi3 specificListener;
    private static ScheduledFuture<?> scheduledFuture;
    private static ScheduledExecutorService
       scheduledExecutorService;
    private static ClientProfile cp = new ClientProfile("1");
```

```
public void init() {
    LOG.info(SmartFarming.class.getSimpleName() + " app
       starting!");

    scheduledExecutorService = Executors.
       newScheduledThreadPool(3);

    // Create the Kaa desktop context for the application.
    DesktopKaaPlatformContext desktopKaaPlatformContext =
       new DesktopKaaPlatformContext();

    kaaClient = Kaa.newClient(desktopKaaPlatformContext,
       new FirstKaaClientStateListener(), true);

    RecordCountLogUploadStrategy strategy = new
       RecordCountLogUploadStrategy(1);
    strategy.setMaxParallelUploads(1);
    kaaClient.setLogUploadStrategy(strategy);

    kaaClient
            .setConfigurationStorage(new
               SimpleConfigurationStorage(
               desktopKaaPlatformContext, "saved_config.cfg
               "));
    kaaClient.setProfileContainer(new ProfileContainer() {

        @Override
        public ClientProfile getProfile() {
            // TODO Auto-generated method stub
            return cp;

        }
    });
    kaaClient.addConfigurationListener(new
       ConfigurationListener() {
        @Override
        public void onConfigurationUpdate(configuration
           configuration) {
```

```
            LOG.info("Received configuration data:",
                configuration.getSample());
            onChangedConfiguration(TimeUnit.MINUTES.
                toMillis(configuration.getSample()));
        }
});

specificListener = new
    BasicNotificationListener_raspi3();
kaaClient.addNotificationListener(specificListener);

// Start the Kaa client and connect it to the Kaa
    server.
kaaClient.start();

Long id = (long)131072;
try {
        kaaClient.subscribeToTopic(id, true);
} catch (UnavailableTopicException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
}

EventFamilyFactory eventFamilyFactory = kaaClient.
    getEventFamilyFactory();

pcf = eventFamilyFactory.getPingCF();
siecf = eventFamilyFactory.
    getSystemInformationEventClassFamily();
kaaClient.attachUser(userId, userToken, new
    UserAttachCallback() {
        @Override
        public void onAttachResult(UserAttachResponse
            response) {
            System.out.println("Attach to user result: " +
                response.getResult());
        }
});

pcf.addListener(new PingCF.Listener() {
```

```
    @Override
    public void onEvent(PingEventClass event, String
       source) {
        System.out.println("event ping response: " +
           event.getMessage());
        if (event.getMessage().equalsIgnoreCase("
           response")) {
            check_time.put(event.getHashEndpoint(),
               event.getCurrentTime());

        }

    }
});

siecf.addListener(new
   SystemInformationEventClassFamily.Listener() {

    @Override
    public void onEvent(SystemInformation event,
       String source) {
        System.out.println("source: " + source);
        System.out.println("System Information: " +
           event.getSystemInformation());
        List<Long> dataSensors = event.getDataSensor()
           ;
        for (Long lg : dataSensors) {
            System.out.println("data: " + lg);
        }
    }
});

List<String> FQNs = new LinkedList<>();
FQNs.add(SystemInformation.class.getName());
FQNs.add(PingEventClass.class.getName());
kaaClient.findEventListeners(FQNs, new
   FindEventListenersCallback() {
    @Override
    public void onEventListenersReceived(List<String>
       eventListeners) {
```

```
                for (String s : eventListeners) {
                    System.out.println("endpoint: " + s);
                }
            }

            @Override
            public void onRequestFailed() {
                // Some code
                System.out.println("searching error");
                ;
            }
        });

        kaaClient.addNotificationListener(specificListener);

        List<Topic> topics = kaaClient.getTopics();
        for (Topic topic : topics) {
            System.out.println("Id:" + topic.getId() + " name:
                 " + topic.getName());
        }
        System.out.println("topics size: " + topics.size());

        LOG.info("--= Press any key to exit =--");
        try {
            System.in.read();
        } catch (IOException e) {
            LOG.error("IOException has occurred: {}", e.
               getMessage());
        }
        LOG.info("Stopping...");
        scheduledExecutorService.shutdown();
        kaaClient.stop();
    }

    private void onKaaStarted(long time) {
        System.out.println("time: " + time);
        if (time <= 0) {
            LOG.error("Wrong time is used. Please, check your
               configuration!");
            kaaClient.stop();
```

```
    System.exit(0);
}

scheduledFuture = scheduledExecutorService.
   scheduleAtFixedRate(new Runnable() {
    @Override
    public void run() {

         long data [] = getTemperatureAndHumidity();
         long temperature = data [0];
         long humidity = data [1];
        long lighting = getLighting();
         LOG.info("Sampled temperature: {}",
            temperature);
         LOG.info("Sampled humidity: {}", humidity);
        LOG.info("sampled lighting: " + lighting);
         TemperatureData td = new TemperatureData(
            temperature);
         HumidityData hd = new HumidityData(humidity);
        LightingData ld = new LightingData(lighting);
        kaaClient.addLogRecord(new MeasurementsData(td
           ,""));
        kaaClient.addLogRecord(new MeasurementsData(hd
           ,""));
        kaaClient.addLogRecord(new MeasurementsData(ld
           , ""));

    }
}, 0, time, TimeUnit.MILLISECONDS);

////// ping request
scheduledFuture = scheduledExecutorService.
   scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
        pcf.sendEventToAll(
                new PingEventClass(System.
                   currentTimeMillis(),
```

```
                kaaClient.getEndpointKeyHash(), "
                    request"));

    }
}, 0, 600, TimeUnit.SECONDS); //10min

//// check time//////
scheduledFuture = scheduledExecutorService.
   scheduleAtFixedRate(new Runnable() {
    @Override
    public void run() {
        double difference;
        int min = 1000*60;
        for (HashMap.Entry<String, Long> entry :
           check_time.entrySet()) {
            difference = System.currentTimeMillis() -
               entry.getValue();
            if (difference > min*30) {//30min
                final String smtpServer = "smtp.gmail.
                   com";
                final String userAccount
                  = "account@gmail.com"; // Sender
                     Account.
                final String password =
                "password"; // Password -> Application
                    Specific Password. final
                  String SOCKET_FACTORY = "javax.net.
                     ssl.SSLSocketFactory"; final
                  String smtpPort = "587"; final
                     String PORT = "465"; final
                     Properties
                  props = new Properties(); props.put(
                     "mail.smtp.host", smtpServer);
                  props.put("mail.smtp.user",
                     userAccount);
                  props.put("mail.smtp.password",
                     password);
                  props.put("mail.smtp.port", smtpPort
                     ); props.put("mail.smtp.auth",
```

```
true); props.put("mail.smtp.starttls
   .enable", "true");
props.put("mail.smtp.debug", "false"
   );
props.put("mail.smtp.socketFactory.
   port", PORT);
props.put("mail.smtp.socketFactory.
   class", SOCKET_FACTORY);
props.put("mail.smtp.socketFactory.
   fallback", "false");

Session session = Session.
   getInstance(props, new
javax.mail.Authenticator() {
   protected PasswordAuthentication
getPasswordAuthentication() { return
    new
PasswordAuthentication(userAccount,
   password); } }); MimeMessage
mimeMessage = new MimeMessage(
   session);

try{
    Address toAddress = new
       InternetAddress("account@gmail
       .com");
    Address fromAddress =  new
       InternetAddress(userAccount);
    mimeMessage.setContent("Endpoint
        "+ entry.getKey() + " not
       working",
    "text/html; charset=UTF-8");
       mimeMessage.setFrom(
       fromAddress);
    mimeMessage.setRecipient(javax.
       mail.Message.RecipientType.TO,
    toAddress); mimeMessage.
       setSubject("Alarm...");
    Transport transport =
    session.getTransport("smtp");
```

```
                                transport.connect(smtpServer,
                                   userAccount, password);
                                transport.sendMessage(
                                   mimeMessage, mimeMessage.
                                   getAllRecipients());
                                transport.close();
                            }
                            catch(Exception e){
                                e.printStackTrace();
                            }
                        }

                    }
                }
            }, 0, 900, TimeUnit.SECONDS); //15min
        }

        private void onChangedConfiguration(long time) {
            if (time == 0) {
                time = DEFAULT_START_DELAY;
            }
            scheduledFuture.cancel(false);
            scheduledFuture = scheduledExecutorService.
               scheduleAtFixedRate(new Runnable() {
                @Override
                public void run() {
                     long data [] = getTemperatureAndHumidity();
                     long temperature = data [0];
                     long humidity = data [1];
                    long lighting = getLighting();
                     LOG.info("Sampled temperature: {}",
                        temperature);
                     LOG.info("Sampled humidity: {}", humidity);
                    LOG.info("sampled lighting: " + lighting);
                     TemperatureData td = new TemperatureData(
                        temperature);
                     HumidityData hd = new HumidityData(humidity);
                    LightingData ld = new LightingData(lighting);
                     kaaClient.addLogRecord(new MeasurementsData(
                        td,""));
```

```
                kaaClient.addLogRecord(new MeasurementsData(
                    hd,""));
                kaaClient.addLogRecord(new MeasurementsData(ld
                    , ""));
            }
        }, 0, time, TimeUnit.MILLISECONDS);
    }

    public class BasicNotificationListener implements
        NotificationListener {

        @Override
        public void onNotification(long topicId,
            NotificationSchema notification) {

            System.out.println(
                    "notification received. topicID: " +
                        topicId + " endpoint: " + notification.
                        getEndpointHash());
            if (topicId == 131072) {
                turnOnLed();
            }
        }

    }

    private long[] getTemperatureAndHumidity() {
        String line;
        String[] data;
        long humidity = 0;
        long temperature = 0;
        Runtime rt = Runtime.getRuntime();
        Process p = null;
        try {
            p = rt.exec("python /home/pi/Desktop/DHT11_Python/
                dht.py");
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    BufferedReader bri = new BufferedReader(new
       InputStreamReader(p.getInputStream()));
    try {
        while ((line = bri.readLine()) != null) {
            data = line.split(",");

            if (data[0].contains("humidity")) {

                humidity = Long.parseLong(data[1].
                   substring(1, 3));
            }
            if (data[0].contains("temperature")) {
                temperature = Long.parseLong(data[1].
                   substring(1, 3));
            }

        }
    } catch (NumberFormatException | IOException e) {
        e.printStackTrace();
    }

    try {
        bri.close();
    } catch (IOException e) {
        e.printStackTrace();
    }

    long[] array = new long[2];
    array[0] = temperature;
    array[1] = humidity;
    System.out.println("Temperature is : " + temperature +
        " C and Humidity is :" + humidity + " %RH");

    return array;
}

private void turnOnLed() {
    Runtime rt= Runtime.getRuntime();
      Process p = null;
      try {
```

```
            p = rt.exec("python /home/pi/pruebasSmartFarming
                /turnonLed.py");
        }
        catch(IOException e) {
        e.printStackTrace(); }

}

private long getLighting() {


        String line; long lighting = 0;; Runtime rt= Runtime
            .getRuntime();
        Process p = null;
        try {
            p = rt.exec("python /home/pi/Desktop/lightling2.
                py");
        }
        catch(IOException e) {
            // TODO Auto-generated catch block
        e.printStackTrace(); }
        BufferedReader bri = new BufferedReader(new
        InputStreamReader(p.getInputStream()));
        try {
            while((line =bri.readLine()) != null){
                lighting = Long.parseLong(line);

            }
        } catch (NumberFormatException | IOException e) {
        // TODO
       // Auto-generated catch block e.printStackTrace(); }

        try { bri.close(); }

        catch (IOException ex) {
            // TODO Auto-generated
            ex.printStackTrace();
        }
        }
        return lighting;
```

```
    }

    private class FirstKaaClientStateListener extends
       SimpleKaaClientStateListener {

        @Override
        public void onStarted() {
            super.onStarted();
            LOG.info("Kaa client started");
            configuration configuration = kaaClient.
               getConfiguration();
            LOG.info("Default sample period: {}",
               configuration.getSample());
            LOG.info("profile: ", cp.getPlot());

            onKaaStarted(TimeUnit.MINUTES.toMillis(
               configuration.getSample()));
        }

        @Override
        public void onStopped() {
            super.onStopped();
            LOG.info("Kaa client stopped");
        }
    }

    public static void main(String[] args) {

        SmartFarming sf = new SmartFarming();
        sf.init();

    }
}
```

B.2 Endpoints del grupo 2

Tanto la Raspberry como el PC pertenecen a este grupo y tienen la misma configuración.

Listing B.2. Código en Java del endpoint 2.

```
public class SmartFarming {
    private static final long DEFAULT_START_DELAY = 1000L;
    private static final Logger LOG = LoggerFactory.getLogger(
       SmartFarming.class);
    private PingCF pcf;
    private SystemInformationEventClassFamily siecf;
    private static KaaClient kaaClient;
    private String hashMainEndpoint = "
       xkGxXOGlvTWLOeuNKQWjaMz8ECg=";
    private String userId = "farming";
    private String userToken = "farming";
    private static ScheduledFuture<?> scheduledFuture;
    private static ScheduledExecutorService
       scheduledExecutorService;
    private static ClientProfile cp = new ClientProfile("2");

    public void init() {
        LOG.info(SmartFarming.class.getSimpleName() + " app
           starting!");
        scheduledExecutorService = Executors.
           newScheduledThreadPool(1);

        DesktopKaaPlatformContext desktopKaaPlatformContext =
           new DesktopKaaPlatformContext();


        kaaClient = Kaa.newClient(desktopKaaPlatformContext,
           new FirstKaaClientStateListener(), true);

        kaaClient
                .setConfigurationStorage(new
                   SimpleConfigurationStorage(
                   desktopKaaPlatformContext, "saved_config.cfg
                   "));
        kaaClient.setProfileContainer(new ProfileContainer() {
```

```
    @Override
    public ClientProfile getProfile() {
        // TODO Auto-generated method stub
        return cp;

    }
});
kaaClient.addConfigurationListener(new
   ConfigurationListener() {
    @Override
    public void onConfigurationUpdate(configuration
       configuration) {

        LOG.info("Received configuration data:",
           configuration.getSample());
           onChangedConfiguration(TimeUnit.MINUTES.
           toMillis(configuration.getSample()));
    }
});
// Start the Kaa client and connect it to the Kaa
   server.
kaaClient.start();
EventFamilyFactory eventFamilyFactory = kaaClient.
   getEventFamilyFactory();

pcf = eventFamilyFactory.getPingCF();
siecf = eventFamilyFactory.
   getSystemInformationEventClassFamily();
kaaClient.attachUser(userId, userToken, new
   UserAttachCallback() {

    @Override
    public void onAttachResult(UserAttachResponse
       response) {
        // TODO Auto-generated method stub
        System.out.println("Attach to user result: " +
            response.getResult());
    }
});
```

```
pcf.addListener(new PingCF.Listener() {
    @Override
    public void onEvent(PingEventClass event, String
       source) {
        System.out.println("event ping request: " +
           event.getMessage());
        if (event.getMessage().equalsIgnoreCase("
           request")) {
            pcf.sendEvent(new PingEventClass( System.
               currentTimeMillis(), kaaClient.
               getEndpointKeyHash(),
                    "response"), source);
        }

    }
});

List<String> FQNs = new LinkedList<>();
FQNs.add(SystemInformation.class.getName());
FQNs.add(PingEventClass.class.getName());
kaaClient.findEventListeners(FQNs, new
   FindEventListenersCallback() {
    @Override
    public void onEventListenersReceived(List<String>
       eventListeners) {
        for (String s : eventListeners) {
            System.out.println("endpint: " + s);
        }
    }

    @Override
    public void onRequestFailed() {
        // Some code
        System.out.println("searching error");

    }
});
```

```
        LOG.info("--= Press any key to exit =--");
        try {
            System.in.read();
        } catch (IOException e) {
            LOG.error("IOException has occurred: {}", e.
               getMessage());
        }
        LOG.info("Stopping...");
        scheduledExecutorService.shutdown();
        kaaClient.stop();
    }

    private void onKaaStarted(long time) {
        System.out.println("time: " + time);
        if (time <= 0) {
            LOG.error("Wrong time is used. Please, check your
               configuration!");
            kaaClient.stop();
            System.exit(0);
        }

        scheduledFuture = scheduledExecutorService.
           scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                long lighting = getLighting();
                LOG.info("Sampled Lighting: {}", lighting);
                LightingData td = new LightingData(lighting);
                kaaClient.addLogRecord(new MeasurementsData(td
                   , ""));

            }
        }, 0, time, TimeUnit.MILLISECONDS);
    }
    private void onChangedConfiguration(long time) {
        if (time == 0) {
            time = DEFAULT_START_DELAY;
        }
        scheduledFuture.cancel(false);
```

```
    scheduledFuture = scheduledExecutorService.
       scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            long lighting = getLighting();
            LOG.info("Sampled Lighting: {}", lighting);
            LightingData td = new LightingData(lighting);
            kaaClient.addLogRecord(new MeasurementsData(td
               , ""));
        }
    }, 0, time, TimeUnit.MILLISECONDS);
}
private long getLighting() {
    ///PC
    long lowerLimit = 123456712L;
    long upperLimit = 234567892L;
    Random r = new Random();
    long number = lowerLimit + ((long) (r.nextDouble() * (
       upperLimit - lowerLimit)));
    return number;
}

private class FirstKaaClientStateListener extends
   SimpleKaaClientStateListener {

    @Override
    public void onStarted() {
        super.onStarted();
        LOG.info("Kaa client started");
        configuration configuration = kaaClient.
           getConfiguration();
        LOG.info("Default sample period: {}",
           configuration.getSample());
        LOG.info("sending system information....");
        ArrayList<Long> dataSensor = new ArrayList<>();
        dataSensor.add(getLighting());
        siecf.sendEvent(new SystemInformation("PC-
           Raspberry2", dataSensor), hashMainEndpoint);
                   onKaaStarted(TimeUnit.MINUTES.toMillis(
           configuration.getSample()));
```

```
        }

        @Override
        public void onStopped() {
            super.onStopped();
            LOG.info("Kaa client stopped");
        }
    }

    public static void main(String[] args) {
        SmartFarming sf = new SmartFarming();
        sf.init();
    }
}
```

B.3 Librerías

Se ha usado DHT11 [39] como librería de python para la recopilación de datos por parte del sensor de temperatura y humedad.
También se ha usado la libería java mail [40] para el envío de correos en java.

Referencias

[1] Fernández D., Sánchez P., Álvarez B., López J.A., Iborra A. (2017) TRIoT: A Proposal for Deploying Teleo-Reactive Nodes for IoT Systems. In: Demazeau Y., Davidsson P., Bajo J., Vale Z. (eds) Advances in Practical Applications of Cyber-Physical Multi-Agent Systems: The PAAMS Collection. PAAMS 2017.

[2] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, Siobhán Clarke. Middleware for Internet of Things, VOL.3, No.1, February 2016

[3] Seguridad en el Internet de las Cosas. UPM. Noviembre 2014

[4] Patrick Guillemin, Friedbert Berens, Marco Carugi, Henri Barthel, Alain Dechamps, Richard Rees, Carol Cosgrove-Sacks, Jamie Clark, Marilyn Arndt, Latif Ladid, George Percivall, Bart De Lathouwer, Steve Liang, Ovidiu Vermesan, Peter Friess. Internet of Things – From Research and Innovation to Market Deployment. June 2014

[5] Pallavi Sethi, Smruti R. Sarangi. Internet of Things: Architectures, Protocols, and Applications. 26 January 2017

[6] http://www.gartner.com/newsroom/id/3598917

[7] Jayavardhana Gubbia, Rajkumar Buyyab, Slaven Marusic, Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions.

[8] Markel Iglesias-Urkia, Adrian Orive, Aitor Urbieta. Analysis of CoAP Implementations for Industrial Internet of Things: A Survey. 2017

[9] Vagan Terziyan, Olena Kaykova, Dmytro Zhovtobryukh. UbiRoad: Semantic Middleware for ContextAware Smart Road Environments.

[10] Chien-Liang Fok, Gruia-Catalin Roman, Chenyang Lug. "Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks"

[11] Edgardo Avilés-López, J. Antonio García-Macías. TinySOA: A service-oriented architecture for wireless sensor networks.

[12] https://www.ibm.com/developerworks/community/blogs/5things/entry/5_things_to_know_about_mqtt_the_protocol_for_internet_of_things?lang=en. Accessed 7 Jul 2017

[13] HiveMQ website. http://www.hivemq.com/. Accessed 28 Jun 2017

[14] HiveMQ. MQTT doc. http://www.hivemq.com/mqtt/. Accessed 28 Jun 2017

[15] CloudMQTT website. https://www.cloudmqtt.com/. Accessed 29 Jun 2017

[16] Mosquitto website. https://mosquitto.org/. Accessed 20 Jun 2017

[17] Paho Client website. https://eclipse.org/paho/clients/java/. Accessed 25 Jun 2017

[18] NodeRed website. https://nodered.org/. Accessed 12 May 2017

[19] MongoDB website. https://www.mongodb.com. Accessed 10 May 2017

[20] Julien Mineraud, Oleksiy Mazhelis, Xiang Su, Sasu Tarkoma. A gap analysis of Internet-of-Things platforms. March 2016

[21] OpenIoT website. http://www.openiot.eu/. Accessed 20 Jun 2017

[22] GIT OpenIoT

[23] D4.3.1 Core OpenIoT Middleware Platform http://cordis.europa.eu/docs/projects/cnect/5/287305/080/deliverables/001-OpenIoTD431Draft.pdf

[24] D6.3.1 Proof-of-Concept Validating Applications a http://cordis.europa.eu/docs/projects/cnect/5/287305/080/deliverables/001-OpenIoTD631131224DraftAresreg.pdf

[25] OpenIoT: Open Source Internet-of-Things in the Cloud, March 2015.

[26] Kaa Official Website. https://www.kaaproject.org. Accessed 5 Jul 2017

[27] KaaIoT Website. https://www.kaaiot.io. Accessed 26 Jun 2017

[28] Source code Kaa. https://github.com/kaaproject. Accessed 1 Jul 2017

[29] Kaa documentation. http://kaaproject.github.io/kaa/docs/v0.10.0/Welcome/. Accessed 4 Jul 2017

[30] Thrift website. https://thrift.apache.org/. Accessed 10 Jun 2017

[31] https://zookeeper.apache.org/. Accessed 8 Jun 2017

[32] http://avro.apache.org/. Accessed 15 May 2017

[33] http://docs.spring.io/spring/docs/3.0.x/reference/expressions.html. Accessed 12 May 2017

[34] https://zeppelin.apache.org/. Accessed 26 Jun 2017

[35] https://www.smart-akis.com/index.php/network/what-is-smart-farming/. Accessed 10 Jul 2017

[36] Forget 'the Cloud'; 'the Fog' Is Tech's Future, Christopher Mims, 2014.

[37] https://www.kaaproject.org/edge-analytics-kaa/. Accessed 18 May 2017

[38] https://www.kaaproject.org/platform/#gate. Accessed 25 May 2017

[39] https://github.com/szazo/DHT11_Python. Accessed 29 Jun 2017

[40] https://mvnrepository.com/artifact/javax.mail/mail/1.4.7. Accessed 26 Jun 2017