

Desarrollo de proyectos IoT utilizando Raspberry Pi como plataforma

Miguel Ángel Martínez Sánchez



Desarrollo de proyectos IoT utilizando Raspberry Pi como plataforma

Miguel Ángel Martínez Sánchez

Tutorizada por

Prof. Tutor Primero Apellido Apellido

Prof. Tutor Segundo Apellido Apellido

Prof. Tutor Tercero Apellido Apellido

Índice general

English Abstract	1
1. Los enunciados	5
1.1. Teoremas y demostraciones	5
1.1.1. Otros enunciados	5
Introducción	7
MQTT	9
1.2. MQTT	9
1.3. Protocolo	10
1.4. Requisitos	15
1.4.1. Requisitos funcionales	15
1.4.2. Requisitos no funcionales	17
1.4.3. Requisitos arquitectónicos	20
1.4.4. Documentación	21
1.5. Cloud Computing	21
1.6. Problemas	22

II DESARROLLO DE PROYECTOS IOT UTILIZANDO RASPBERRY PI COMO PLATAFORMA

MQTT-SN	23
1.7. Redes de sensores inalámbricas	23
1.8. MQTT-SN	24
1.9. Arquitectura MQTT-SN	24
1.10. Funcionamiento	25
1.11. MQTT vs MQTT-SN	25
OpenIoT	27
1.12. Introducción	27
1.13. Plataforma	28
1.13.1. Arquitectura	28
1.13.2. Flujo de datos	35
1.13.3. Requisitos	39
1.13.4. Casos de éxito con la plataforma	42
1.14. Problemas	44
KAA	45
1.15. Arquitectura	45
1.16. Funcionamiento	48
1.17. Qué ofrece Kaa	51
1.17.1. Requisitos funcionales	51
1.17.2. Requisitos no funcionales	51
1.17.3. Requisitos arquitectónicos	51
1.18. Uso de la plataforma	57
1.18.1. Instalación	57

ÍNDICE GENERAL III

1.18.2. Partes de la plataforma	58
1.19. Ejemplos prácticos	63
1.19.1. Configuración de datos	64
1.19.2. Colección de datos	65
1.19.3. Perfiles y grupos	66
1.19.4. Chat mediante eventos	68
1.19.5. Encender un LED en la raspberry usando Kaa	72
1.19.6. Edge analytics	76
1.20. Conclusiones	78

Abstract

En este trabajo se han analizado diferentes soluciones de middleware open-source para el Internet de las Cosas (IoT). Se plantea un escenario teleo-reactivo y se pretende dar respuesta a qué middleware open-source es el más idóneo. El análisis se ha realizado comprobando si satisface determinados requisitos y dando respuesta a cómo suplirlos en caso de que no cumpla alguno de ellos.

Keywords: Internet de las cosas (IoT); MQTT; OpenIoT; Kaa

Agradecimientos

1 | Los enunciados

1.1 Teoremas y demostraciones

| **Teorema 1.1 (Euclides).** *Esto es un Teorema. Se numeran a partir del 1 en cada capítulo. Como son importantes, tienen un cuadrado rojo al principio. Llevan letra cursiva.*

Demostración. Esto es la demostración. Al final de la demostración se puede ver un cuadrado rojo similar al de los teoremas. Las demostraciones no llevan letra cursiva.

| **Definición 1.1.** *Esto es una definición. Las definiciones son importantes; también llevan un cuadradito rojo.*

1.1.1 Otros enunciados

Observación 1.1. Esto es una observación, que dice que $e = mc^2$. Como las observaciones no son importantes, no llevan cuadrado rojo, y el tipo de letra no es cursiva.

Demostración. Si la demostración acaba en una fórmula, para poner el cuadrado rojo a la altura de la última formula, hay que usar la orden \qedhere, como en este caso:

$$e = mc^2.$$

Corolario 1.1. Esto es un corolario.

Proposición 1.1. Esto es una proposición.

Lema 1.1 (Gauss). Esto es un lema.

Introducción

Planteamos el siguiente escenario para obtener la automatización de una finca agrícola. Para tal propósito disponemos de Raspberry y Arduino. //////////////Foto////////// El análisis de cada uno de los middleware IoT, se hará siguiendo los requisitos presentes en la siguiente figura. Esta tabla de requisitos está presente en un artículo de 2016 [3] en el que se analizan diferentes middleware para IoT.

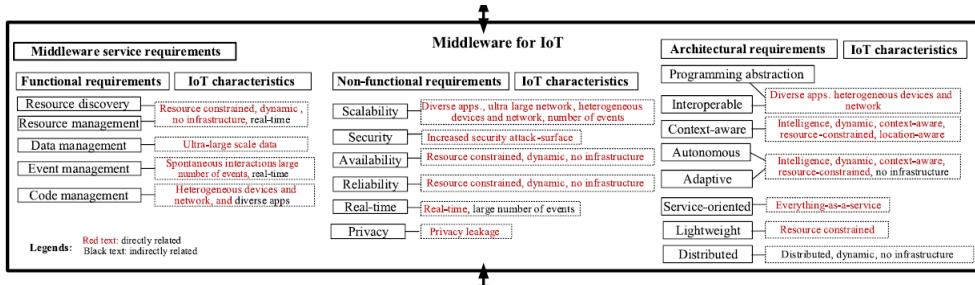


Figura 1.1: Requisitos de un middleware IoT.

MQTT

1.2 MQTT

MQTT(Message Queue Telemetry Transport) 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 "broker" o servidor centralizado para la gestión de los paquetes.

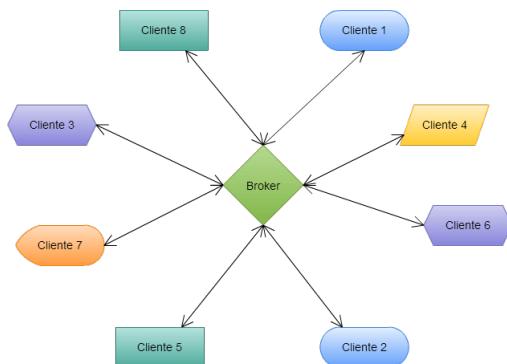


Figura 1.2: Topología MQTT.

10 DESARROLLO DE PROYECTOS IOT UTILIZANDO RASPBERRY PI COMO PLATAFORMA

La comunicación se basa en "topics" (temas) que el cliente que publica el mensaje envía y los nodos que desean recibirlo deben suscribirse 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 1.25

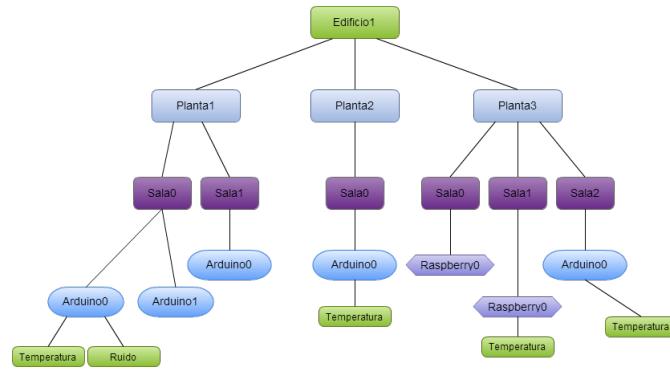


Figura 1.3: Jerarquía Topics.

1.3 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 suscribirse 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 1.1: 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 1.26. El cliente envía un paquete "SUBSCRIBE" con el que el cliente le dice a qué topics quiere suscribirse. 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 1.2. Una vez recibida la confirmación, el cliente puede empezar a enviar mensajes.

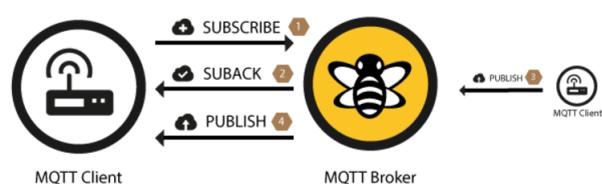


Figura 1.4: Envío de paquetes en MQTT.

12 DESARROLLO DE PROYECTOS IOT UTILIZANDO RASPBERRY PI COMO PLATAFORMA

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

Tabla 1.2: Código de respuesta



Figura 1.5: Paquete SUBSCRIBE.

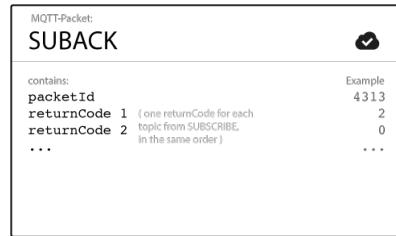


Figura 1.6: Paquete SUBACK.

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"

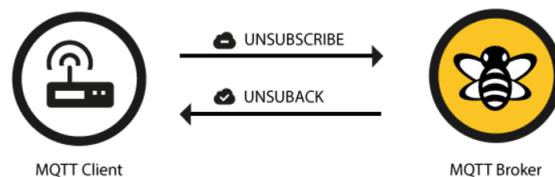


Figura 1.7: Envío de paquetes para desuscribirse de un topic.



Figura 1.8: Paquete UNSUBSCRIBE.

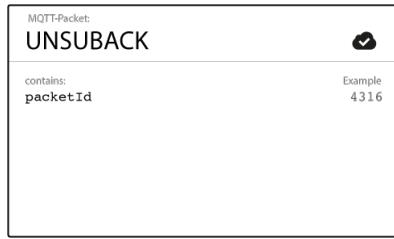


Figura 1.9: Paquete UNSUBACK.

Brokers

Son los encargados de transmitir los mensajes. 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 [1][2] que comercializan un broker MQTT diseñado por ellos mismos. Otro broker comercial es CloudMQTT [14], propiedad de Amazon.

En este trabajo se trabajará con el broker Mosquitto [9], 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.

14 DESARROLLO DE PROYECTOS IOT UTILIZANDO RASPBERRY PI COMO PLATAFORMA

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

Tabla 1.3: Comparación entre brokers MQTT

MQTT Websocket surge para usar MQTT sobre HTTP para 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.

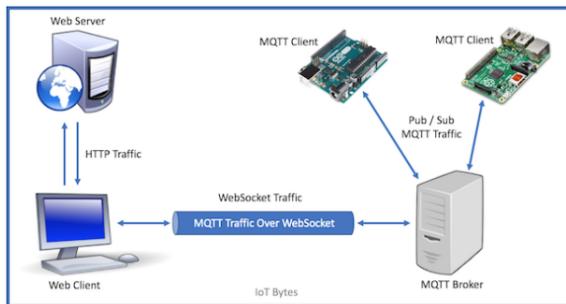


Figura 1.10: MQTT sobre WebSockets

Clients

Se encargan de enviar al broker la información recibida, normalmente por parte de sensores. Los clientes también reciben mensajes de los topics subscriptos. Al igual que los brokers, hay muchos tipos de clientes MQTT. El cliente Open-Source más usado es Eclipse Paho Java Client [13]

1.4 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 pueda ser la seguridad. Aquí se definen algunos de los requisitos.

1.4.1 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, una extensión de MQTT para redes de sensores, 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 por ejemplo MQTTSpy. MQTTSpy se ejecuta sobre Java y proporciona estadísticas . Esta herramienta se conectaría con el broker MQTT ///////////

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, suscrito 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 suscrito a un tópi-



Figura 1.11: Single level

co en el que solamente cambie el nivel indicado por el signo +. En este ejemplo, 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.

- **Multi level:** #

En este caso, llegaría a cualquier cliente que esté suscrito a partir de ese nivel.



Figura 1.12: Multi level

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

Control de eventos

Control de código

1.4.2 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 topic.

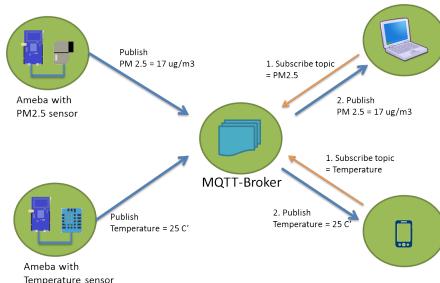


Figura 1.13: Escalabilidad MQTT

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.

Disponibilidad

Debido a su arquitectura ///"brokerizada"///, los sistemas MQTT tienen un único punto de fallo: el broker. La disponibilidad del sistema es baja al basarse únicamente

18 DESARROLLO DE PROYECTOS IOT UTILIZANDO RASPBERRY PI COMO PLATAFORMA

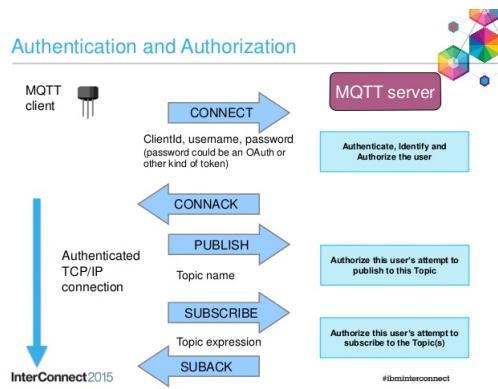


Figura 1.14: Autenticación MQTT

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.

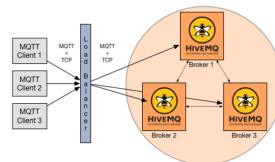


Figura 1.15: MQTT Clustering.

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.

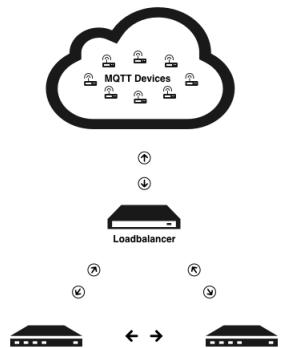


Figura 1.16: MQTT en modo bridge.

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.



Figura 1.17: QoS 0

- **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.

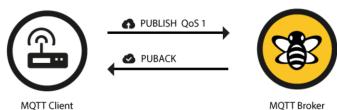


Figura 1.18: QoS 1

- **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.

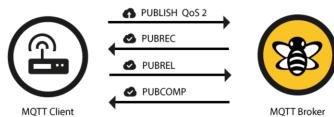


Figura 1.19: QoS 2

Tiempo real

Si tiempo real lo definimos en microsegundos, 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 microsegundos no son necesarios por lo que no es un requisito indispensable.

1.4.3 Requisitos arquitectónicos

Abstracción de la programación

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...

Autónomo

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 bro-

kers como un único broker. El broker Open-Source Mosquitto no cumple con este requisito.

Orientado a servicio

Ligero

El bajo consumo de MQTT es una de las claves de su uso extendido. 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.

1.4.4 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.

1.5 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 son el uso de herramientas como Node-RED[10] junto con MongoDB[11]. 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.

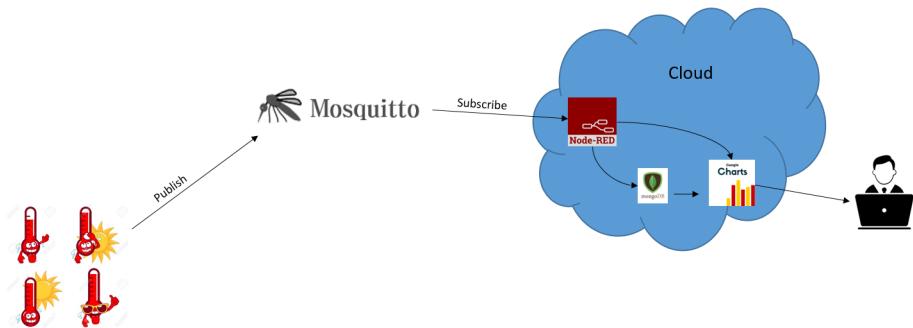


Figura 1.20: Cloud Computing haciendo uso de MQTT

1.6 Problemas

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 más plataformas Open-Source que nos ofrezca una solución más completa. Es decir, un middleware que unifique almacenamiento en la nube, procesamiento y análisis de datos...

MQTT-SN

1.7 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 1.39 se puede ver la estructura de una red de este tipo.

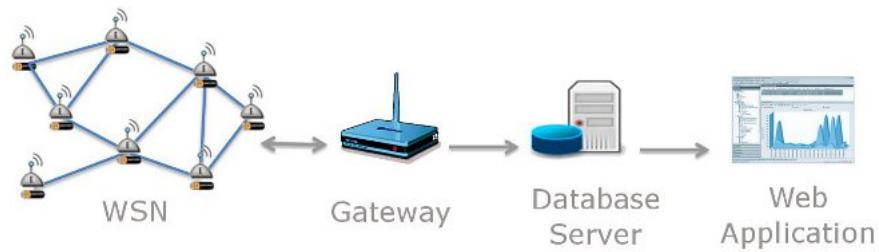


Figura 1.21: Multi level

1.8 MQTT-SN

MQTT es un protocolo usado sobre redes donde el ancho de banda sea 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.

Es por ello donde surge MQTT-SN, un protocolo pub/sub específico para redes de sensores.

1.9 Arquitectura MQTT-SN

La arquitectura MQTT-SN se muestra en la figura 1.22

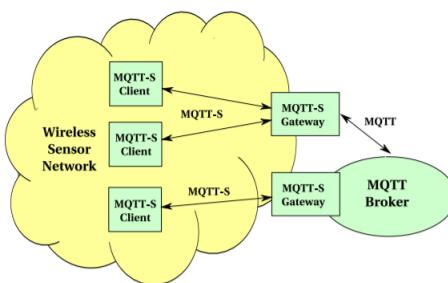


Figura 1.22: Arquitectura MQTT-SN

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.

1.10 Funcionamiento

1.11 MQTT vs MQTT-SN

MQTT-SN se diferencia en MQTT en varios aspectos. A nivel de red, no funciona sobre TCP pero por contra necesita de Gateways para..... ///////////////. 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 anuncio 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.

OpenIoT

1.12 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 para describir las cosas/objetos físicos. No existe un modelo a seguir para integrar todos los servicios, como sí existe en redes IP.

OpenIoT 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 flujo 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.

1.13 Plataforma

1.13.1 Arquitectura

La arquitectura de la plataforma se puede ver en la figura 1.23

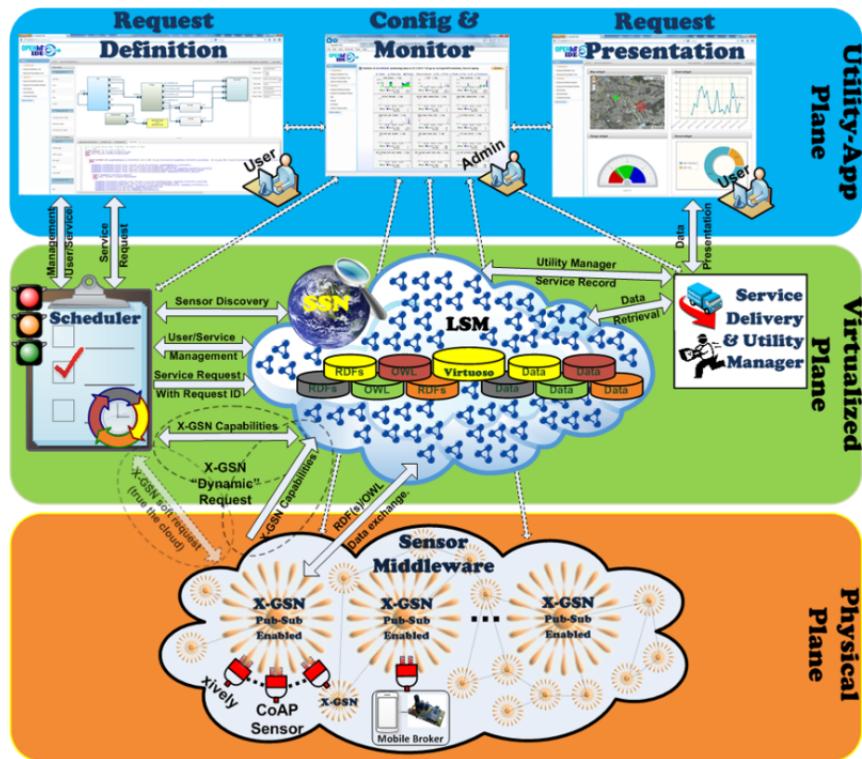


Figura 1.23: Arquitectura OpenIoT

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 colecciónar 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 1.24

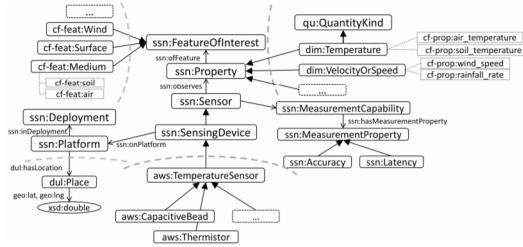


Figura 1.24: Semántica de SSN.

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 1.27 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. Este registro se realiza mediante el envío a la plataforma LSM de un archivo de los metadatos del sensor, en el que se definen las propiedades del mismo. Un ejemplo de este archivo se puede ver en la figura 1.25.

```

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/115080594572850"
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
  
```

Figura 1.25: Metadatos de un sensor.

mediante un archivo XML enviado a la plataforma LSM. En este XML se usa una descripción en formato RDF. Los sensores, situados en el plano físico, envían una instancia de sus datos (situación geográfica, memoria...) a la nube. Estos metadatos se envían en formato RDF, una semántica usada en la Web enlazada ("Linked Web", en inglés) y así pueden conectar con LSM.

En la figura 1.26 se puede ver el proceso para el registro de un sensor en OpenIoT. Inicialmente se crea y se configura el sensor, mediante la descripción XML. Un

ejemplo de esta descripción se puede ver en la figura 1.27. 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, lo que facilita el descubrimiento de recursos.

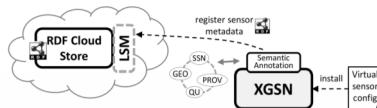


Figura 1.26: Registro de un sensor virtual

```
<?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</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>
```

Figura 1.27: Configuración de un sensor.

OpenIoT ofrece soporte para el descubrimiento y recolección de datos por parte de sensores móviles tales como pulseras, gafas, relojes, 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 suscritos. Como se puede observar en la figura 1.28, 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.

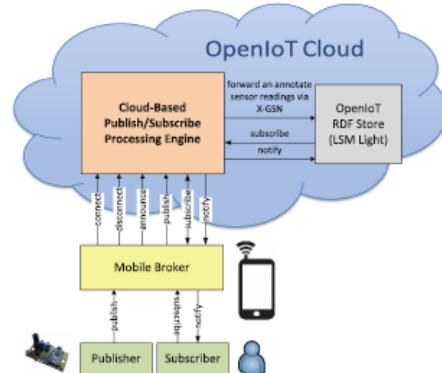


Figura 1.28: Arquitectura publish/subscribe

Plano Virtual

El plano virtual está compuesto por el almacenamiento en la nube (LSM-Light), el Scheduler y el servicio de entrega.

LSM-Light (Linked Sensor Middleware Light) es el componente principal de Ope-

nIoT. 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

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 principales: 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 1.29 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 servicio Request Definition es quien realiza las peticiones al Scheduler para que éste las procese. La forma que tiene de enviar las peticiones es mediante una API, especificada en la documentación.

El SD&UM (Service Delivery & Utility Manager), al igual que el Scheduler, realiza peticiones a la Cloud DB mediante una API.

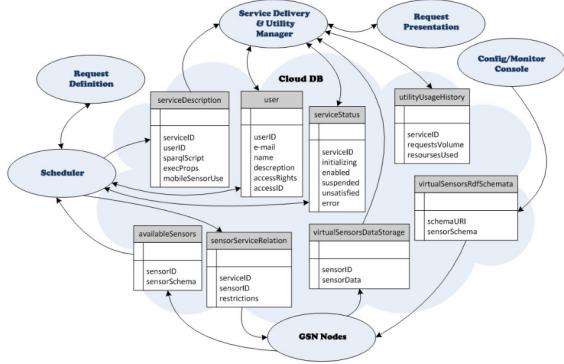


Figura 1.29: Relación de cada servicio con la Cloud DB

Plano de utilidad

En este plano se sitúan las interfaces de usuario. Request Definition es una aplicación que permite a los usuarios visualizar sus servicios de OpenIoT usando una interfaz basada en nodos. Cada modelo 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 almacenan en el Scheduler y se cargan automáticamente cuando el usuario accede a la web. Un ejemplo de esta interfaz se muestra en la figura 1.30

34 DESARROLLO DE PROYECTOS IOT UTILIZANDO RASPBERRY PI COMO PLATAFORMA

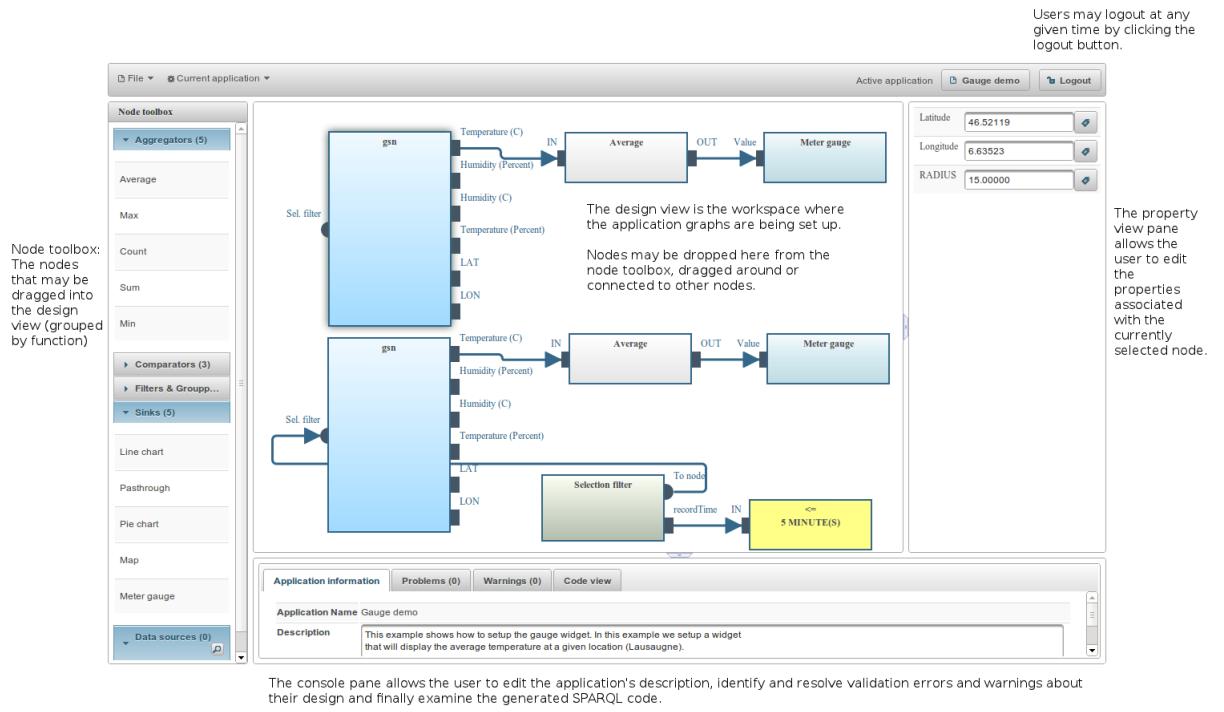


Figura 1.30: Request Definition

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 1.31 se muestra un ejemplo con la definición del servicio en Request Definition, y en la figura 1.32 la interfaz gráfica de los datos en Request Presentation, una vez recopilados.

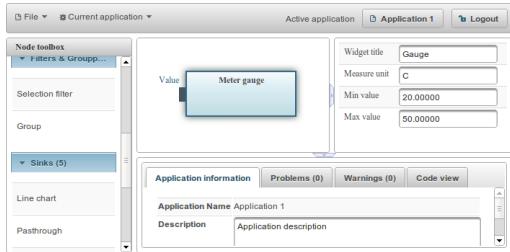


Figura 1.31: Definition

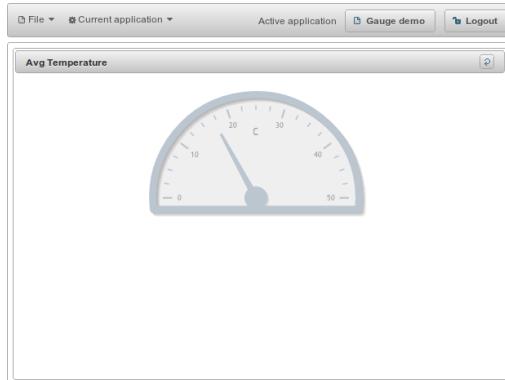


Figura 1.32: Presentation

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.

1.13.2 Flujo de datos

En base a la arquitectura que se muestra en la figura ??, la figura 1.33 representa un ejemplo del flujo que siguen los datos en la plataforma.

X-GSN publica los datos de los sensores virtuales basados en la configuración local de cada nodo (sensor). Paso 0.

Los usuarios realizan peticiones al Scheduler (paso 1) de los sensores disponibles con determinados atributos usando la interfaz Request Definition.

El Scheduler ejecuta (paso 2) estas peticiones (en lenguaje SPARQL) enviadas por los usuarios.

Una vez que tenga la respuesta (los sensores disponibles) se envía de vuelta al Scheduler (paso 3) y éste la reenvía al módulo Request Definition (paso 4), mostrándose la información al usuario.

El usuario, con ayuda de Request Definition, define peticiones para realizar determinadas reglas sobre los sensores analizados. Esta información se guarda en un objeto OSDSpec (Figura 1.37). Este objeto se envía entonces al Scheduler con la ayuda de

'registerService (paso 5).

El Scheduler analiza la información recibida y envía la petición al servicio necesario (paso 6).

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 (pasos 7,8,9 y 10) recupera todos los servicios/aplicaciones registrados de acuerdo a un usuario específico

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') (paso 11). El SD&UM realiza una petición ("getService") (paso 12) para solicitar toda la información relacionada al servicio.

El servicio proporciona la información al SD&UM (paso 13).

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 (paso 14).

El resultado se envía al SD&UM (paso 15) en formato SparqlResultsDoc

El SD&UM lo reenvía al Request Presentation (paso 16) en un objeto que incluye información de cómo esos datos se deben presentar (Figura 1.35).

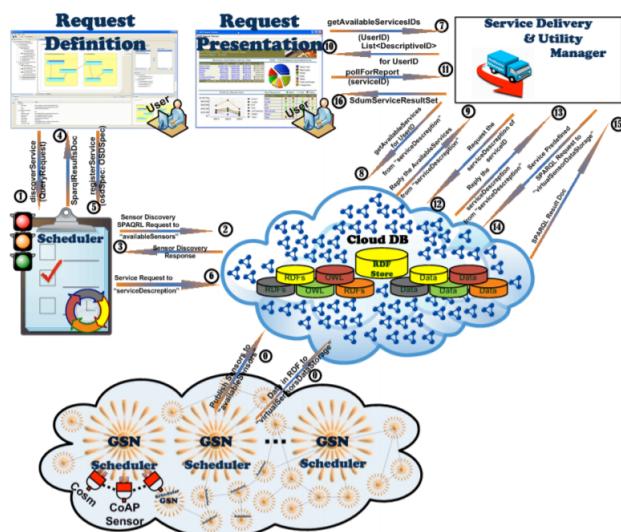


Figura 1.33: Flujo de datos en OpenIoT

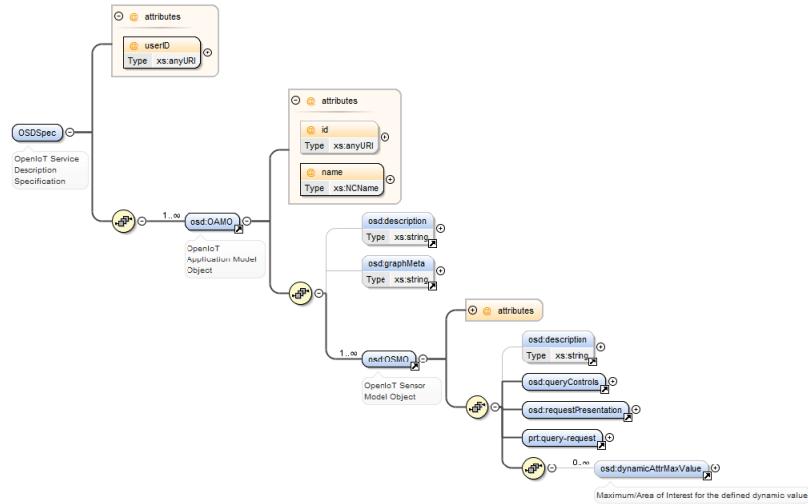


Figura 1.34: Grafo de un objeto OSDSpec.

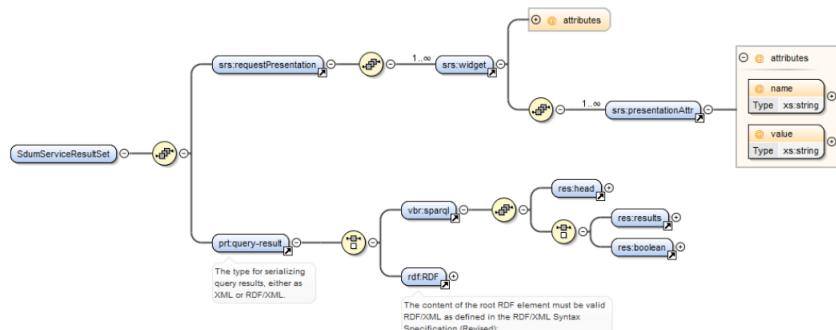


Figura 1.35: Grafo de un objeto SdumServiceResultSet.

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, TinyDB o MQTT, mencionado anteriormente.

A pesar de que con los middleware se facilita el desarrollo de aplicaciones en IoT, cada middleware cumple una serie de requisitos que puede que no sirvan para una determinada aplicación. Por ejemplo, en este caso, MQTT no cumplía alguno de los requisitos exigidos. No hay un middleware que reúna todas necesidades exigidas. Es aquí donde aparece OpenIoT, que permite reunir los 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 (figura 1.37) 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 (SDUM) o definir los servicios IoT (Request Definition). Esto permite el desarrollo de aplicaciones IoT de una manera más rápida y sencilla.



Figura 1.36: OpenIoT IDE

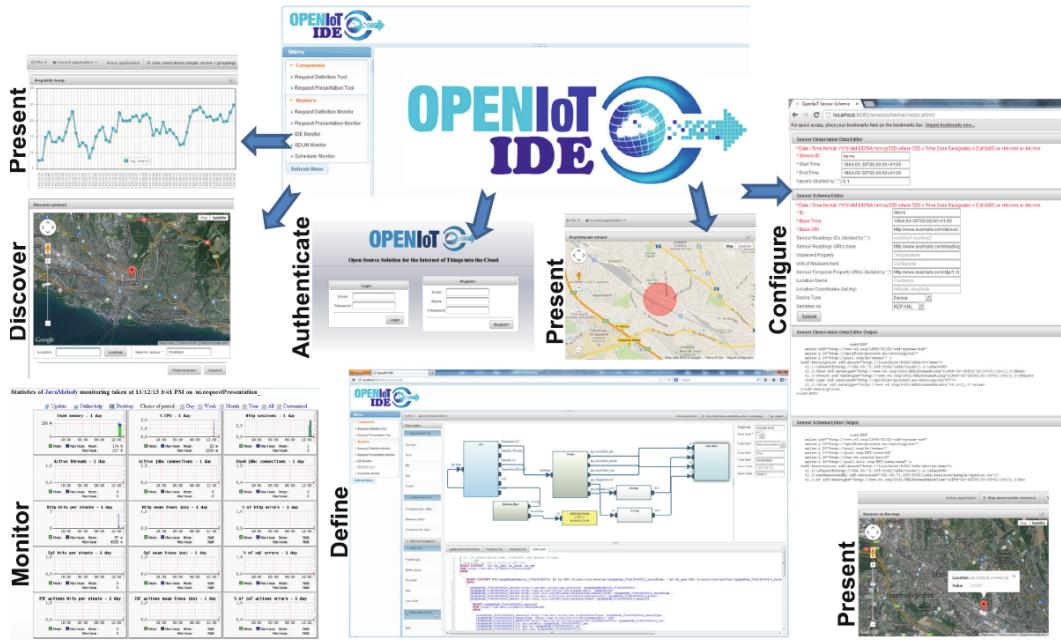


Figura 1.37: Servicios integrados en IDE

1.13.3 Requisitos

Las aplicaciones IoT deben cumplir una serie de requisitos, algunos de ellos necesarios en un entorno IoT. Algunos de estos requisitos son:

Escalabilidad

OpenIoT es una plataforma totalmente escalable debido a la característica de descubrimiento de recursos que posee. 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.

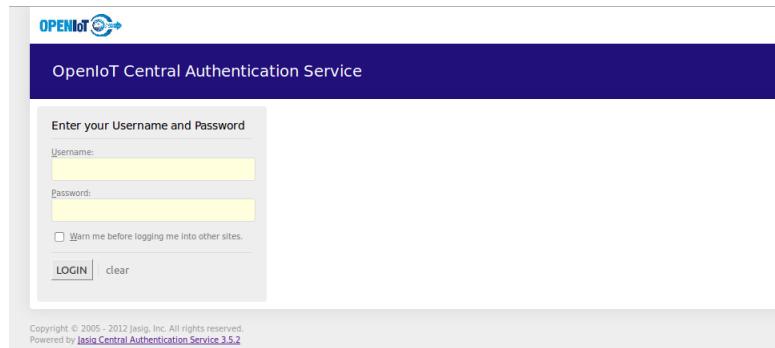


Figura 1.38: Login en CAS

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 módulo proporciona control de acceso y autenticación. Se puede usar en aplicaciones web que interactúan con los usuarios.
- **Security Management:** Se trata de un módulo que se usa para llevar un control sobre servicios, usuarios y permisos.

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 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 de los recursos.

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.

Interoperabilidad

La interoperabilidad de los servicios IoT es la idea fundamental por la que surgió OpenIoT. Se buscaba unificar distintos servicios en una única plataforma. En OpenIoT se unifica todo lo relacionado con redes de sensores (Middleware...) y lo relacionado con el Cloud Computing (base de datos, herramientas de procesamiento...).

Distribuido



1.13.4 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 (www.sensap.eu) 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, 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 SENSAF, para colecciónar 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, 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 1.39

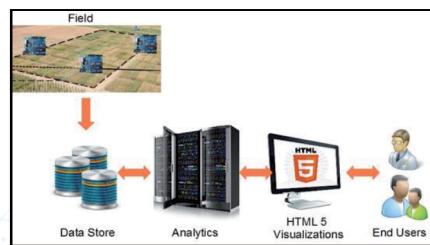


Figura 1.39: Arquitectura de Phenonet

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 través de una API mediante HTML. Phenonet está basado en los siguientes módulos de OpenIoT:

- X-GSN: Se encarga de los datos 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.

1.14 Problemas

Tras un tiempo intentando desplegar sin éxito la plataforma, se ha optado por descartarla. OpenIoT es una plataforma reciente y con un potencial alto pero inexplicablemente no tiene 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 hacerla funcionar correctamente y, navegando por la red, nos damos cuenta que son problemas comunes que no tienen solución. Esto nos ha llevado a descartar la plataforma y buscar otra solución para el problema que planteamos.

KAA

1.15 Arquitectura

Kaa [5][6][7][8] es un middleware de código abierto que, al igual que OpenIoT, busca un control e integración de las aplicaciones IoT. Kaa tiene un poderoso backend 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 'empotrada' 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 y fácil de usar.

En la figura 1.41 se puede ver su arquitectura.

Un cluster Kaa representa un número de nodos (Kaa server) interconectados. La figura 1.44 muestra un flujo de datos común, donde se puede ver un cluster Kaa, formado por un kaa server.

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

//////////Palabras clave: CLUSTER,SDK,NODE,SCHEMA,LOG/////////

Hay empresas como [Kaaiot](#) que se dedican a crear soluciones haciendo uso de Kaa. Todas ellas son soluciones muy completas y con un grado de dificultad bastante alto.

46 DESARROLLO DE PROYECTOS IOT UTILIZANDO RASPBERRY PI COMO PLATAFORMA



Figura 1.40: Arquitectura Kaa

Un conjunto de nodos interconectados representa un Cluster Kaa. Un Cluster requiere bases de datos SQL y NoSQL para almacenar los datos y metadatos de los endpoints. El servidor Kaa usa Apache Zookeeper para coordinar los servicios.

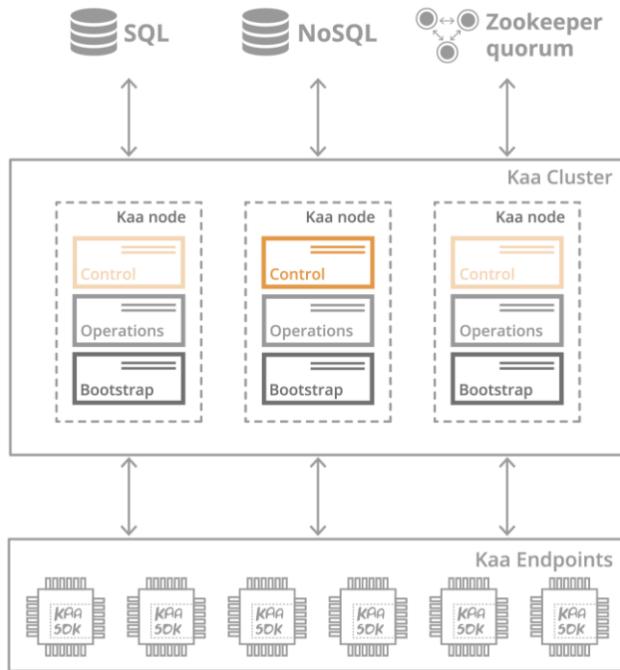


Figura 1.41: Cluster Kaa

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...

1.16 Funcionamiento

La figura 1.44 muestra un flujo de datos en Kaa. Los sensores 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, previamente definido en el Servidor mediante los "Log Schemas". 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. 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.

La plataforma se organiza en cluster conectados entre sí, siendo cada cluster un nodo o servidor. Por defecto, está formada por un único cluster.

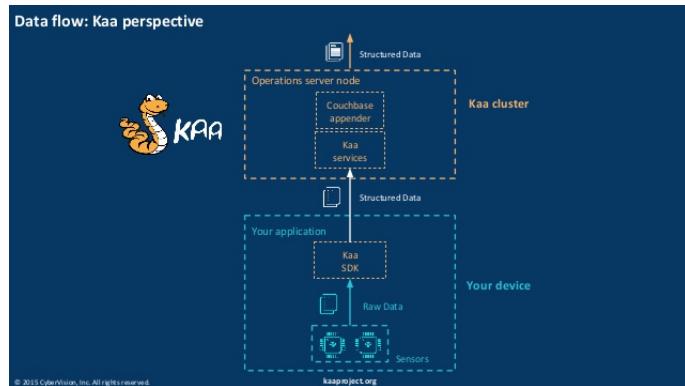


Figura 1.42: Flujo de datos en Kaa

La forma más común de despliegue de la plataforma se muestra en la siguiente figura:

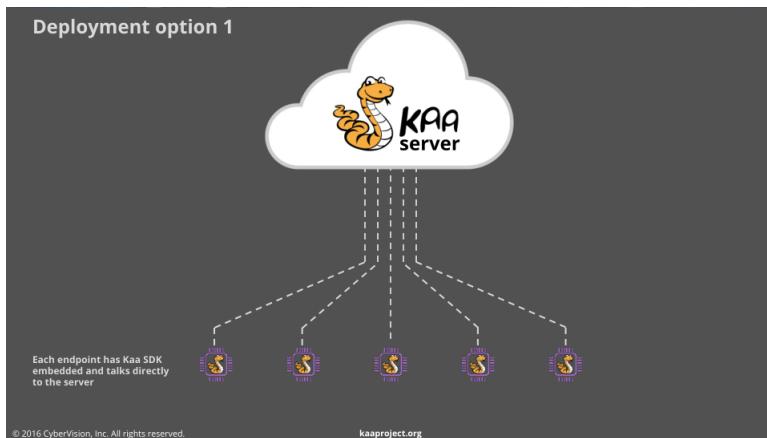


Figura 1.43: Despliegue típico de Kaa

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. Sin embargo, existe otro despliegue posible con la arquitectura y es el siguiente:

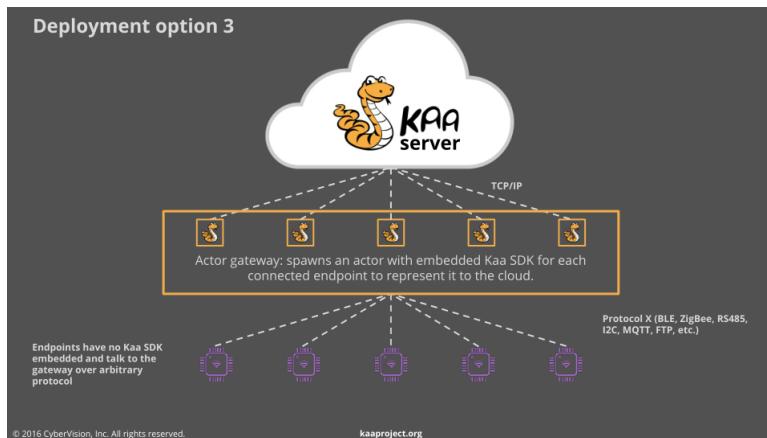


Figura 1.44: Otro despliegue con Kaa

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 el SDK 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. Kaa no ofrece implementación para los gateways, como sí la ofrece para los endpoints, por tanto tiene que ser el desarrollador quien la implemente. Para versiones posteriores se plantea la inclusión de algún <http://jira.kaaproject.org/browse/KAA-752> para simplificar la integración con Kaa en dispositivos en los que no se pueda ejecutar el SDK o bien se busque este tipo de arquitectura ya que tiene algunos beneficios como se verá más adelante.

Esquemas de datos

Los datos se recolectan desde los endpoints y se envían al servidor en el formato definido por el 'log schema', previamente creado por el desarrollador para la aplicación en concreto. Log Schema usa un formato compatible con Apache Avro. En la figura 1.45 se muestra un ejemplo de un esquema de datos. El desarrollador define cómo quiere que se envíe y almacenen los datos mediante estos esquemas, lo que hace a la plataforma muy flexible, facilitando la integración entre los diferentes servicios.

```
{
  "type": "record",
  "name": "LogData",
  "namespace": "org.kaaproject.kaa.schema.sample.logging",
  "fields": [
    {
      "name": "level",
      "type": {
        "type": "enum",
        "name": "Level",
        "symbols": [
          "DEBUG",
          "ERROR",
          "FATAL",
          "INFO",
          "TRACE",
          "WARN"
        ]
      }
    },
    {
      "name": "tag",
      "type": "string"
    },
    {
      "name": "message",
      "type": "string"
    }
  ]
}
```

Figura 1.45: Ejemplo de Log Schema

1.17 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 1.1

1.17.1 Requisitos funcionales

1.17.2 Requisitos no funcionales

1.17.3 Requisitos arquitectónicos

Descubrimiento de recursos

Kaa desplegada como la arquitectura mostrada en la figura ////figura del despliegue 2/// permite un descubrimiento ¿automático? de nuevos dispositivos a través del Gateway. El Gateway es un dispositivo actuando 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 ////Kaa ofrece un sistema de eventos que permite enviarlos a los diferentes endpoints.//// Estos eventos se generan en los endpoints y, al igual que el almacenamiento de datos, los eventos se definen mediante esquemas. La siguiente figura 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).

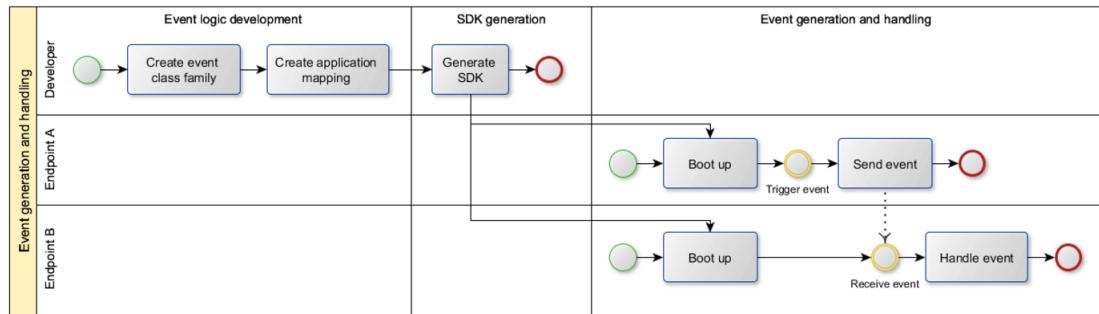


Figura 1.46: Diagrama de eventos en Kaa

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 provenientes 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 Cassandra 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 1.47 se puede ver un esquema de uso con Apache Storm.

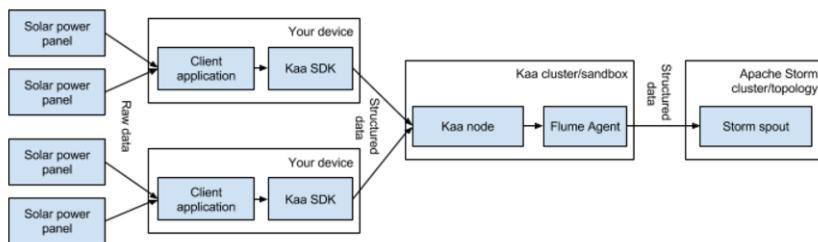


Figura 1.47: Esquema de uso para la monitorización de datos con Apache Storm

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).

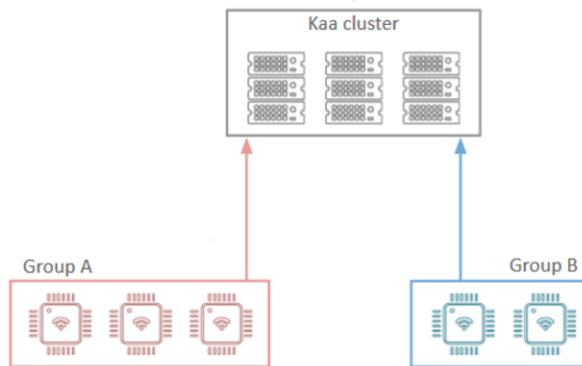


Figura 1.48: Grupos de Endpoints

Tiempo real

Kaa es un sistema en tiempo real, entendiendo tiempo real como unos ms de retraso. Ya que Kaa 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.

Disponibilidad

Kaa es un sistema con una alta disponibilidad. Está formado por clusters, donde cada cluster representa un servidor. Estos clusters están interconectados mejorando así la disponibilidad en caso de fallo de alguno de ellos. Para la coordinación entre los diferentes nodos, Kaa usa Apache Zookeeper. Además, los datos de los endpoints se almacenan en bases de datos tolerantes a fallos.

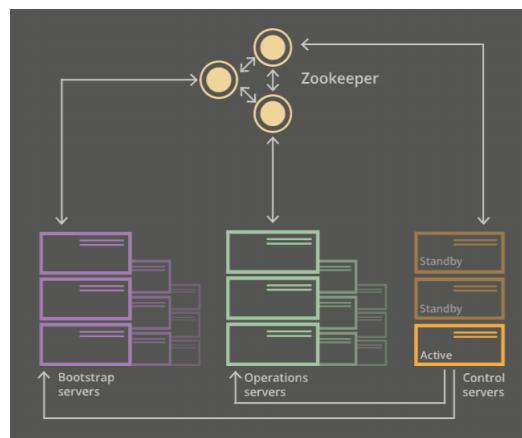


Figura 1.49: Esquema Zookeeper en Kaa

Seguridad y privacidad

Por defecto, en la comunicación entre Kaa y el SDK se usa una encriptación con RSA y AES. Kaa también asegura un almacenamiento de datos seguro en la base de datos así como la autenticación en la plataforma. Kaa usa el hash SHA-1 de la clave pública como identificador del endpoint en el sistema.

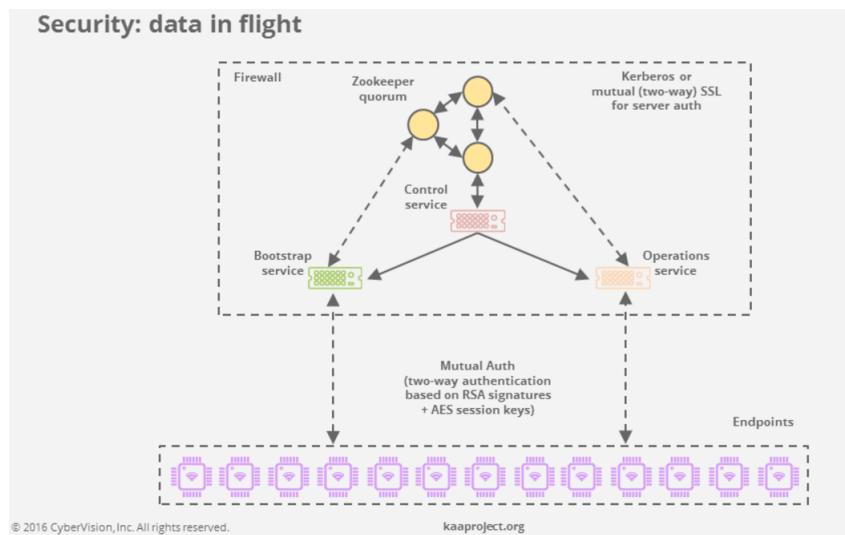


Figura 1.50: Seguridad en Kaa

Popularidad

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

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.

Interoperabilidad

La interoperabilidad de 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. 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.

Tabla 1.4: 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, pueden ser los endpoints quienes realicen una determinada tarea.

Basado en servicio

--

Adaptativo

-- Kaa soporta virtualmente cualquier protocolo de red, endpoint o almacenamiento de datos, lo que hace a la plataforma tolerante a cambios. --

Control de eventos

Control de código

Una de las principales características de Kaa es que es Open Source con todo lo que ello implica. El código es totalmente transparente al usuario.

1.18 Uso de la plataforma

1.18.1 Instalación

Para instalar la plataforma hay 2 formas, o bien usar la maquina virtual que ya viene preconfigurada y con unos ejemplos de uso (SandBox) o bien, construir un servidor

Kaa propio mediante el código fuente. Todo esto está perfectamente documentado.

1.18.2 Partes de la plataforma

Las características de Kaa son:

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 ?? 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 /////////////////explicar 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.

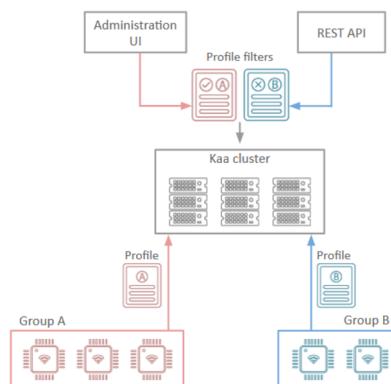


Figura 1.51: Uso de perfiles de grupos de endpoints en Kaa

Los PF están basados en [Spring Language](#). 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 suscribirse a uno o varios tópicos. También, se puede asignar un tópico a todo un grupo de endpoints. El envío de notificaciones se puede hacer mediante la interfaz de usuario o mediante una llamada a la API.

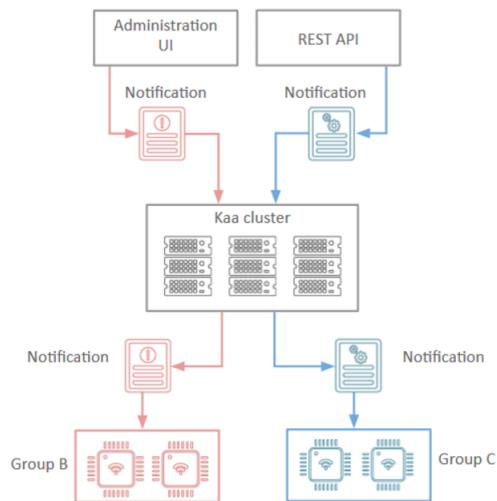


Figura 1.52: Notificaciones en Kaa

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.

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 notificaciones mediante SMS y la configuración y datos mediante TCP. //////////////explicar virtualmente e incluirlo si consigo usar MQTT como protocolo de transporte//////////

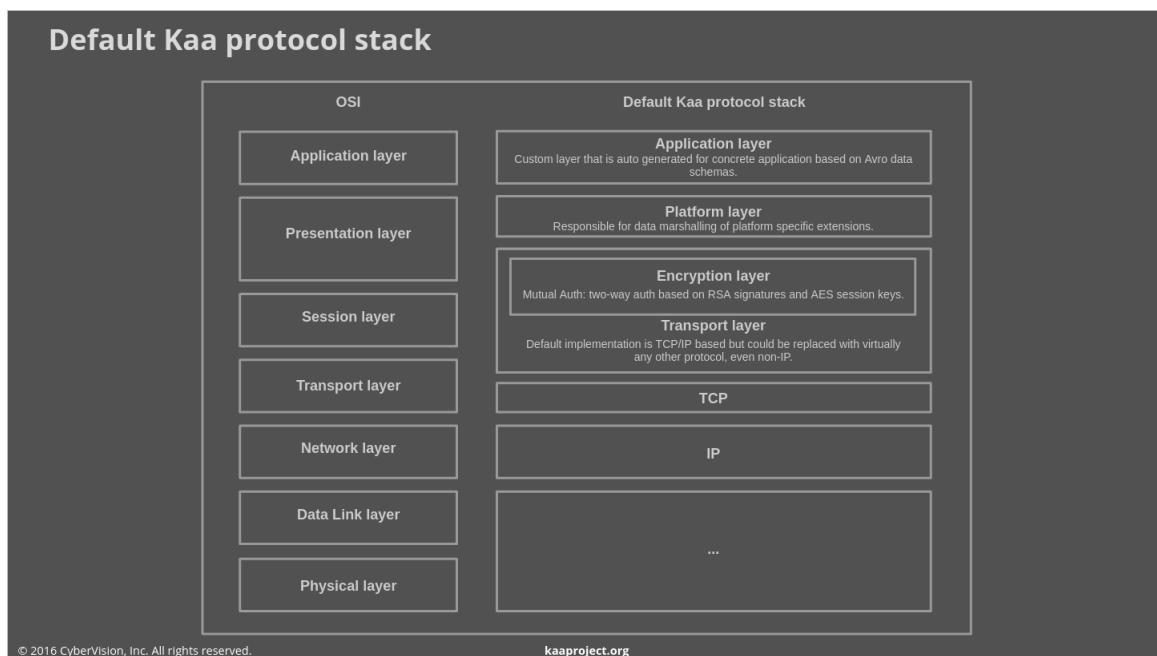


Figura 1.53: Pila de protocolos de Kaa

1.19 Ejemplos prácticos

Además de diferentes escenarios planteados, se van a probar los ejemplos presentes en la versión de VirtualBox preconfigurada, así como los presentes en el [GIT](#) de Kaa. Estos ejemplos me han ayudado a comprender el funcionamiento de Kaa en mayor profundidad, además de comprobar varios de los requisitos que se plantean en este documento para un middleware IoT.

1.19.1 Configuración de datos

En este ejemplo se muestra el funcionamiento de la 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.

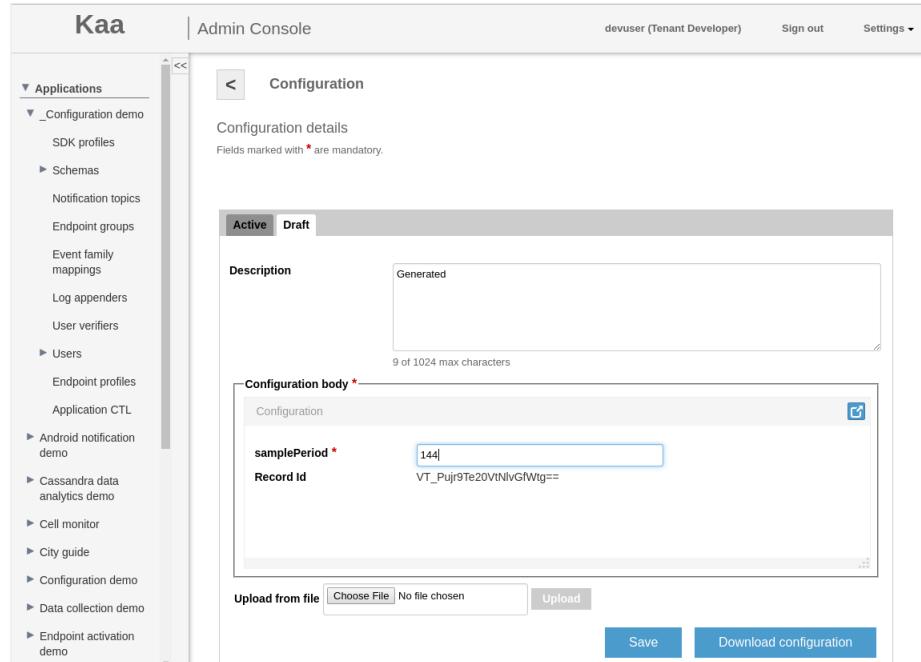


Figura 1.54: Cambio de configuración por defecto

```
[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.

1.19.2 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". Para comprobar su correcto funcionamiento, nos conectamos a MongoDB y obtenemos los datos de la tabla definida en "Log appender":

[captura]

1.19.3 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. [1.18.2](#)

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.page",
    "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.page",
    "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 1.18.2, 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 cliente 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 la siguiente consola:

(captura)

En la que se ve como el endpoint 2 (que en realidad es una instancia de un cliente Kaa) pertenece al grupo "Todos los servicios habilitados".

Cambiando desde la interfaz el filtro en el 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:

(captura)

El endpoint 2 cambia su configuración al nuevo grupo ya que, aunque tenga la misma configuración, el grupo tiene un mayor peso.

1.19.4 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 endpoints en Java, en una Raspberry y en un PC, y Android en un tercer endpoint, 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.curre
        }
    }
});

scheduledFuture = scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
    @Override
    public void run() {
        Long dif;
        for (Map.Entry<String, Long> entry : lists.entrySet()) {
            dif = System.currentTimeMillis() - entry.getValue();
            if(dif > 2000){
                aecr.sendEvent(new AlarmCTL("Endpoint: " + entry.getKey() + " not working", "qmIEHsWY91T6xNL6CatpJ5oRBCQ"));
            }
        }
    }
}, 0, 15, TimeUnit.SECONDS);
}
```

(a) Código de recepción de respuesta

(b) Código de evaluación del tiempo para cada endpoint



(c) Recepción de evento en Android

Figura 1.55: 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 ?? se podrían conse-

uir 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.

1.19.5 Encender un LED en la raspberry usando Kaa

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.

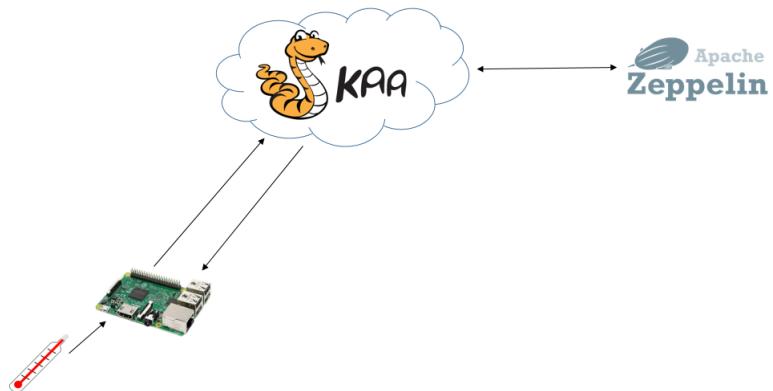


Figura 1.56: Escenario planteado en Kaa

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 tiem-

po, 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 Cassandra 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.

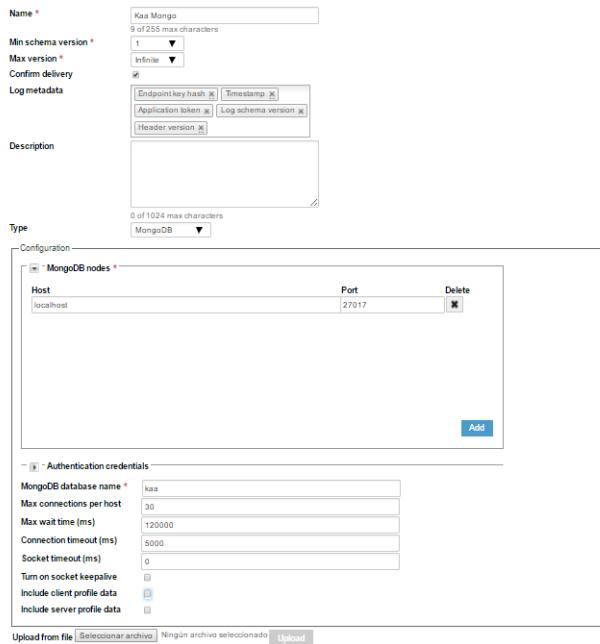


Figura 1.57: Esquema utilizado para el almacenamiento en MongoDB

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, al ser un lenguaje compatible con la plataforma (Ver tabla 1.4).

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[12] 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.

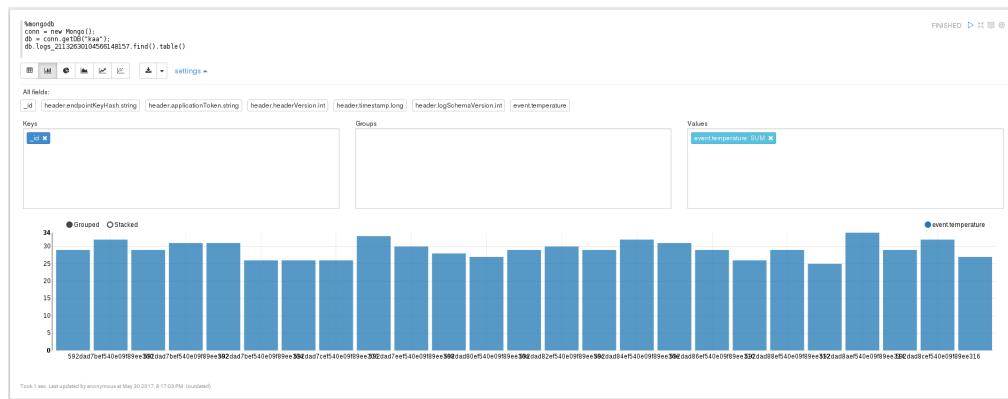


Figura 1.58: Obtención de datos de MongoDB

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, previamente suscrita 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

1.19.6 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 Endpoint, 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[4]. 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.

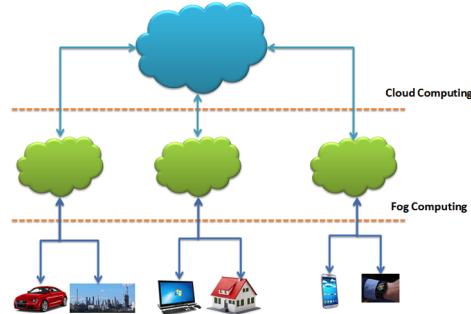


Figura 1.59: Fog Computing

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 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 ([vídeo](#)). Desplegando Kaa como la arquitectura mostrada en la figura ///, 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, 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 ([enlace](#)) 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.

Tareas a realizar

- Hacer funcionar eventos y documentarlo - Comprobar el control de recursos. Crear una función Ping de forma que cada endpoint tenga que responder de forma periodica (Crear primero en Java) y si no responden, enviar notificación. - Almacenar tan solo alarmas o datos analizados para demostrar el uso de Edge Analytics

—> El HASH de la clave pública de un endpoint identifica al endpoint en el cluster Kaa. Por defecto Kaa genera una clave pública en cada endpoint registrado. Están presentes en el archivo public.key

1.20 Conclusiones

Kaa es una plataforma con una curva de aprendizaje bastante alta Con Kaa el programador se olvida de la parte del servidor

Bibliografía

- [1] <http://www.hivemq.com/>
- [2] HiveMQ. MQTT doc:<http://www.hivemq.com/mqtt/>
- [3] Middleware for Internet of Things, VOL.3, No.1, February 2016.
- [4] Forget 'the Cloud'; 'the Fog' Is Tech's Future, Christopher Mims, 2014.
- [5] Kaa Official Website: <https://www.kaaproject.org>
- [6] Kaaiot Website: <https://www.kaaiot.io>
- [7] Source code: <https://github.com/kaaproject>
- [8] Kaa documentation: <http://docs.spring.io/spring/docs/3.0.x/reference/expressions.html>
- [9] <https://mosquitto.org/>
- [10] <https://nodered.org/>
- [11] <https://www.mongodb.com>
- [12] <https://zeppelin.apache.org/>
- [13] <https://eclipse.org/paho/clients/java/>
- [14] <https://www.cloudmqtt.com/>
- [15] <https://www.kaaproject.org/platform/#gate>