Trabajo Fin de Grado
Ingeniería de Telecomunicación

Diseño y arquitectura de una plataforma para recolección de datos y control de sistemas integrados

Autor: Diego Fernández Barrera

Tutor: Pablo Nebrera Herrera

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Ingeniería de Telecomunicación

Diseño y arquitectura de una plataforma para recolección de datos y control de sistemas integrados

Autor:

Diego Fernández Barrera

Tutor:

Pablo Nebrera Herrera

Profesor asociado

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado: Diseño y arquitectura de una plataforma para recolección de datos y control de sistemas integrados

Autor: Diego Fernández Barrera

Tutor: Pablo Nebrera Herrera

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Índice

1. Introducción **1**

1.1. Motivación 1

1.2. Escenario 1

1.2.1. Usuario 2

1.2.2. Dispositivos integrados 2

1.2.3. Lógica de negocio 2

2. Arquitectura de la plataforma **3**

2.1. Funcionalidad 3

2.1.1. Bidireccionalidad 3

Bidireccionalidad en la interfaz con los servicios del cliente 3

Bidireccionalidad en la interfaz con los dispositivos 4

2.1.2. Tiempo real 4

2.1.3. Persistencia 4

2.1.4. Tenencia múltiple 5

2.2. Diseño 5

2.3. Componentes 6

2.3.1. API 6

2.3.2. Broker 7

2.3.3. Bases de datos 7

2.3.4. Backend de autorización 7

2.3.5. Procesador de flujos 7

3. Protocolo MQTT, AMQP y STOMP **9**

3.1. MQTT 9

3.1.1. El patrón publicador/suscriptor 9

3.1.2. Escalabilidad 10

3.1.3. Filtrado de mensajes 10

3.1.4. Diferencias con un sitema de colas 10

3.2. STOMP 11

3.3. AMQP 11
3.3.1. Brokers 11
3.3.2. El modelo AQMP 11
3.3.3. *Exchanges* 12
Direct Exchange 12
Fanout Exchange 13
Topic Exchange 14
Header Exchange 14
3.3.4. Colas 14
Colas durables 14
3.3.5. *Bindings* 14
3.3.6. Mensages 14
3.4. STOMP vs. MQTT vs. AMQP 15

4. Broker y autorización 17
4.1. RabbitMQ 17
4.1.1. Configuración 18
4.2. Backend de autorización 18

5. API 21
5.1. Funcionamiento 21
5.2. Registro de usuario 21
5.3. Obtención de Fleet Key 22
5.4. Registro de dispositivos 23
5.5. Desarrollo 24

6. Bases de datos 29
6.1. MongoDB 29
6.2. RethinkDB 29

7. Dispositivos y Módulos 31
7.1. Dispositivos 31
7.1.1. Flotas 31
7.2. Módulos 31
7.2.1. Caso 1 32
7.2.2. Caso 2 32
7.3. Conexión al broker 32

8. Ejemplo de uso 33
8.1. Dispositivos 33
8.2. Servicios 34
Alertas 34
Lecturas 35
8.3. Envío de comandos 37

Apéndice A. Sobre el código **39**

Índice de Figuras 41
Índice de Códigos 43
Bibliografía 45

Introducción

Motivación

Hoy en día es innegable que las tecnologías enfocadas al IoT (Internet de las cosas) están en pleno auge. La tendecia es conectar todo lo que se pueda a Internet para así hacerlo inteligente.

Si un fabricante decide crear un dispositivo, digamos que desarrolla un sensor de temperatura para que un usuario pueda monitorizar la temperatura de una habitación mediante una interfaz web o app móvil. Al poco de plantear alguna solución encontrará que desarrollar toda una arquitectura para poder comunicar su sensor de temperatura con la aplicación web puede ser un proceso complejo. Será necesario obtener datos de miles de sensores, tratar los datos, almacenarlos, poder obtenerlos desde la aplicación web o móvil, etc.

Mientras que una organización de gran tamaño con suficientes recursos puede abordar el problema, para pequeñas compañías o *startups* puede suponer un *handicap*. El desarrollo de un backend sobre el que se apoye su producto puede ser una tarea que termine por hacer que el proyecto sea inviable, ya que no permite al desarrollador centrar todos sus esfuerzos en desarrollar su producto y le obliga a gastar recursos en construir y mantener su backend.

En el panorama actual existe una gran cantidad de alternativas a la hora de elegir las diferentes tecnologías que compondrán el sistema. El mero hecho de realizar un estado del arte ya supone un esfuerzo. Para solucionar cada pequeño problema podemos encontrar una gran variedad de soluciones y a la hora de la integración de las diferentes partes pueden surgir más problemas.

Todo esto no hace más que suponer una barrera para los fabricantes que puede desembocar en que el proyecto nunca sea llevado a cabo.

Si buscamos soluciones ya desarrolladas se pueden encontrar varias alternativas, muchas de ellas realizadas por empresas de la talla de *IBM* o *AT&T* que son maduras y ofrecen todo lo necesario. Sin embargo, todas ellas atan al usuario a usar su infraestructura y depender de ella, si el usuario decide que ya no le conviene seguir usando el servicio puede ser un problema migrar a otro.

Dicho esto, este trabajo abordará el diseño de una alternativa *open source* a estos servicios que cualquier usuario u organización pueda instalar en su propia infraestructura.

Escenario

A continuación se describe la nomenclatura que se usará a lo largo del escrito.

Usuario

Se llamará usuario a la persona, personas u organización que utiliza la plataforma que vamos a diseñar para su beneficio. Se pretende cubrir una necesidad, que en este caso es comunicar dos elementos: los dispositivos integrados del usuario y la lógica de negocio del usuario.

El usuario no tiene por qué saber cómo funciona el sistema de forma interna, sólamente deberá saber cómo interactuar con él.

Dispositivos integrados

En primer lugar se encuentran los dispositivos del usuario, que son los elementos que se pretenden dotar de conectividad para que puedan comunicarse de forma eficiente y robusta. Se debe tener en cuenta que cuando se habla de sistemas integrados no se puede suponer que se cuenta con los mismo recursos que en un equipo estándar. La cantidad de memoria, de almacenamiento, de ancho de banda o incluso de energía pueden ser muy limitadas, por lo que no se pueden aplicar las mismas soluciones que se aplicarían en un entorno donde sobran dichos recursos.

Con esto se quiere decir que protocolos como `HTTP` que funcionan de forma correcta en un equipo actual no tienen por qué funcionar igual de bien en un dispositivo que cuenta con poca memoria. Si en lugar de *WiFi* se usa un protocolo más simple orientado a eficiencia energética, `HTTP` puede ser muy pesado y su implementación puede consumir mucha memoria, energía o ancho de banda.

Habrá que buscar soluciones acordes a esta tecnología.

Lógica de negocio

Es el conjunto de *software* que desea comunicarse con los dispositivos, ya sea para recolectar los datos que unos sensores captan o para accionar unos actuadores. Toda esta lógica pertenece al usuario y puede ser desde un pequeño *script* en *Python* hasta una compleja infraestructura desplegada en *Amazon*. Puesto que tiene otras características diferentes a los dispositivos tendrán otros requisitos diferentes, por lo que la forma en la que interaccionarán con la plataforma no tiene por qué ser la misma que los dispositivos.

Uno de los requisitos para la lógica de negocio es que debe poder funcionar en un navegador web, de forma que se podría crear una aplicación web que se comunique **directamente** con la plataforma.

Arquitectura de la plataforma

Con motivo de no hacer esta memoria demasiado extensa, se omitirán todas las comparativas que se han hecho entre diferentes protocolos y servicios y todos los cambios realizados desde la primera versión hasta la definitiva. De esta forma podremos centrarnos en describir la arquitectura final.

Desde el primer diseño de la arquitectura hasta el actual se ha pasado por un enorme proceso de simplificación en el cual se han eliminado bucles, conexiones redundantes, servicios innecesarios, etc. Se puede decir que la arquitectura actual consta de los componentes mínimos para llevar a cabo su funcionalidad.

Funcionalidad

Todas las decisones realizadas para elegir la tecnología más adecuada a la hora de la implementación del sistema se basan en cuatro premisas:

- *Bidireccionalidad:* Los datos debe poder fluir en ambos sentidos. Desde un extremo de la arquitectura al otro y viceversa.
- *Tiempo real:* Los datos deben pasar de un extremo a otro en un tiempo mínimo. Es decir, debe haber el menor número de paradas posibles.
- *Persistencia:* El sistema debe ser capaz de almacenar los datos obtenidos para que puedan ser consultados en cualquier momento.
- *Tenencia múltiple:* Un despliegue de la plataforma debe permitir ser usada por varios usuarios u organizaciones simultáneamente de forma completamente aislada.

Además el diseño debe ser lo suficientemente genérico para que cualquier usuario, independientemente del tipo de datos que quiera enviar, pueda hacer uso del sistema.

Bidireccionalidad

La plataforma está diseñada con el objetivo de permitir la comunicación de forma bidireccional. Como tiene una interfaz con los dispositivos y otra con la lógica de negocio del usuario, tenemos que garantizar que los protocolos que usemos en ambos casos nos permitan bidireccionalidad.

Bidireccionalidad en la interfaz con los servicios del cliente

En el paradigma clásico de internet (`HTTP`), la comunicación siempre va de cliente a servidor. El servidor nunca envía ningún dato al cliente, sino que el cliente debe obtener los datos como respuesta a una petición

que él mismo realice, pero no puede ser notificado en el momento de que haya un nuevo dato. Esto representa un problema a la hora de conseguir tiempo real.

Una primera solución sería que el cliente constantemente realice peticiones al servidor para ver si hay datos nuevos para él. Esto es un método poco eficiente y sólo permite una "ilusión" de que obtenemos los datos en tiempo real. Además este método no es escalable, pues si tenemos miles de clientes sería muy costoso que estén constantemente haciendo peticiones al servidor ya que cada conexión supone una reserva y posterior liberación de recursos.

Afortunadamente, en la actualidad existen numerosas alternativas a este método que permiten la comunicación bidireccional entre cliente y servidor de forma eficiente.

Se ha elegido el protocolo STOMP, que permite a los servicios del cliente establecer una conexión TCP persistente y recibir datos en tiempo real sin realizar consultas períodicas. La conexión TCP se mantiene y así se evita estar constantemente reservando y liberando recursos. El servidor enviará una notificación a través de la conexión TCP establecida.

Bidireccionalidad en la interfaz con los dispositivos

En cuanto al extremo contrario, es importante que los dispositivos también sean capaces de recibir datos en tiempo real desde la plataforma.

Se podría usar la misma idea que en la otra interfaz, sin embargo, existe un protocolo muy popular conocido como MQTT que es fácil de implementar en muchos dispositivos. A diferencia de HTTP, MQTT sí permite una comunicación bidireccional, por lo que se puede usar dicho protocolo para comunicar también los dispositivos con la plataforma sin mayor problema.

Tiempo real

El diseño actual de la arquitectura se centra en ofrecer una comunicación extremo a extremo sin que, en ningún momento, un elemento tenga que solicitar los datos explícitamente al elemento que le precede, sino que será notificado por éste cuando haya nuevos datos.

Los datos irán fluyendo por toda la infraestructura hasta llegar al cliente que, por ejemplo, puede mostrarlo en una web en tiempo real, almacenarlo en una base de datos o enviar una notificación push a un dispositivo móvil. En ningún momento el dato se quedará en un elemento intermedio a la espera de que el siguiente componente lo pida explícitamente.

No debe asumirse que este flujo reactivo impida el agrupamiento de varios mensajes en uno (*batching*). Lo que el concepto de flujo reactivo realmente implica es que el elemento que recibe un dato decide cuando enviarlo al siguiente, en lugar de esperar que lo soliciten explícitamente.

Como ya se ha indicado antes, la única espera podría ser para realizar batching y así aumentar la eficiencia del sistema. Una posible mejora sobre la implementación actual sería hacer agrupaciones de mensajes cuyo tamaño sea dinámico y dependa de la carga del sistema. Cuando el sistema está saturado, los mensajes se agrupan en lotes de mensajes de mayor tamaño, mientras que si el sistema está descargado la granularidad de los mensajes es mayor. Otra opción sería realizar batching para los clientes gratuitos mientras que para los clientes de pago se le ofrece granularidad total y, por lo tanto, un tiempo de respuesta menor.

Persistencia

El usuario puede obtener sus datos de dos formas diferente:

- *Flujo:* Los datos son entregados en tiempo real al cliente y éste debe procesarlos conforme van llegando. Los filtros debe aplicarlos el usuario.
- *Consultas:* El cliente puede decidir realizar una consulta a la API cuando él lo decida para obtener los datos. Pueden establecerse ciertos filtros, por ejemplo, puede obtener todos los mensajes de un dispositivo en un rango de tiempo.

Obviamente, para el segundo caso es obligatorio que el sistema sea capaz de persistir los datos. Por ello el sistema dispondrá de diferentes bases de datos para poder llevar a cabo esta tarea.

Tenencia múltiple

La plataforma debe permitir el uso por múltiples organizaciones o usuarios. Cada usuario puede tener una o varias flotas de dispositivos que sean independientes de los otros usuarios. Aunque los datos pasen por la misma plataforma debe existir un aislamiento que impida a un usuario obtener datos de otro.

Otra ventaja de la tenencia múltiple es que una organización puede realizar un despliegue de la plataforma y ofrecer los servicios a otras organizaciones.

Diseño

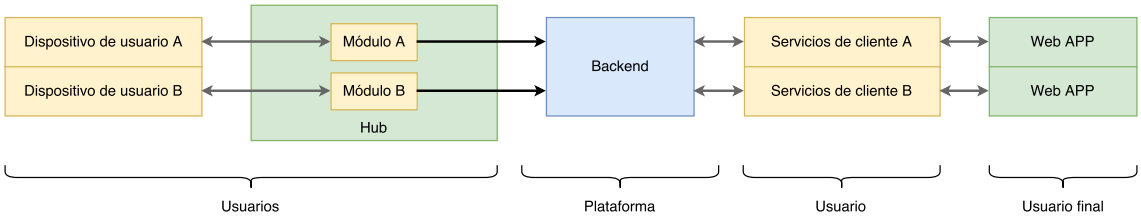

Figura 2.1 Estructura de un aplicación que use la plataforma.

Como sev puede ver en la Figura 3.1, la plataforma se sitúa como elemento intermedio entre los dispositivos y la lógica del usuario. El objetivo es ofrecer de forma transparente y eficiente un canal de comunicación al usuario para que éste pueda centrarse en su negocio, que es ofrecer una serie de servicios al usuario final sin tener que preocuparse por cómo se mueven los datos.

Puesto que la plataforma soporta *multi tennant*, múltiples usuario pueden compartir la plataforma o una organización puede encargarse de mantenerla y ofrecer sus servicios a los usuarios finales.

Sabiendo todo esto, se podría definir la plataforma de la siguiente manera:

> *Es una plataforma que permite a los desarrolladores de dispositivos inteligentes comunicar sus dispositivos con su lógica de negocio de forma transparente, rápida, fiable y segura.*

Para usar la plataforma habrá que preparar a los dispositivos que se distribuyan para que se comuniquen con la API y por otro lado habrá que diseñar los servicios para que obtengan los datos también de la API.

A continuación se puede ver un esquema de la arquitectura final en Figura 3.2.

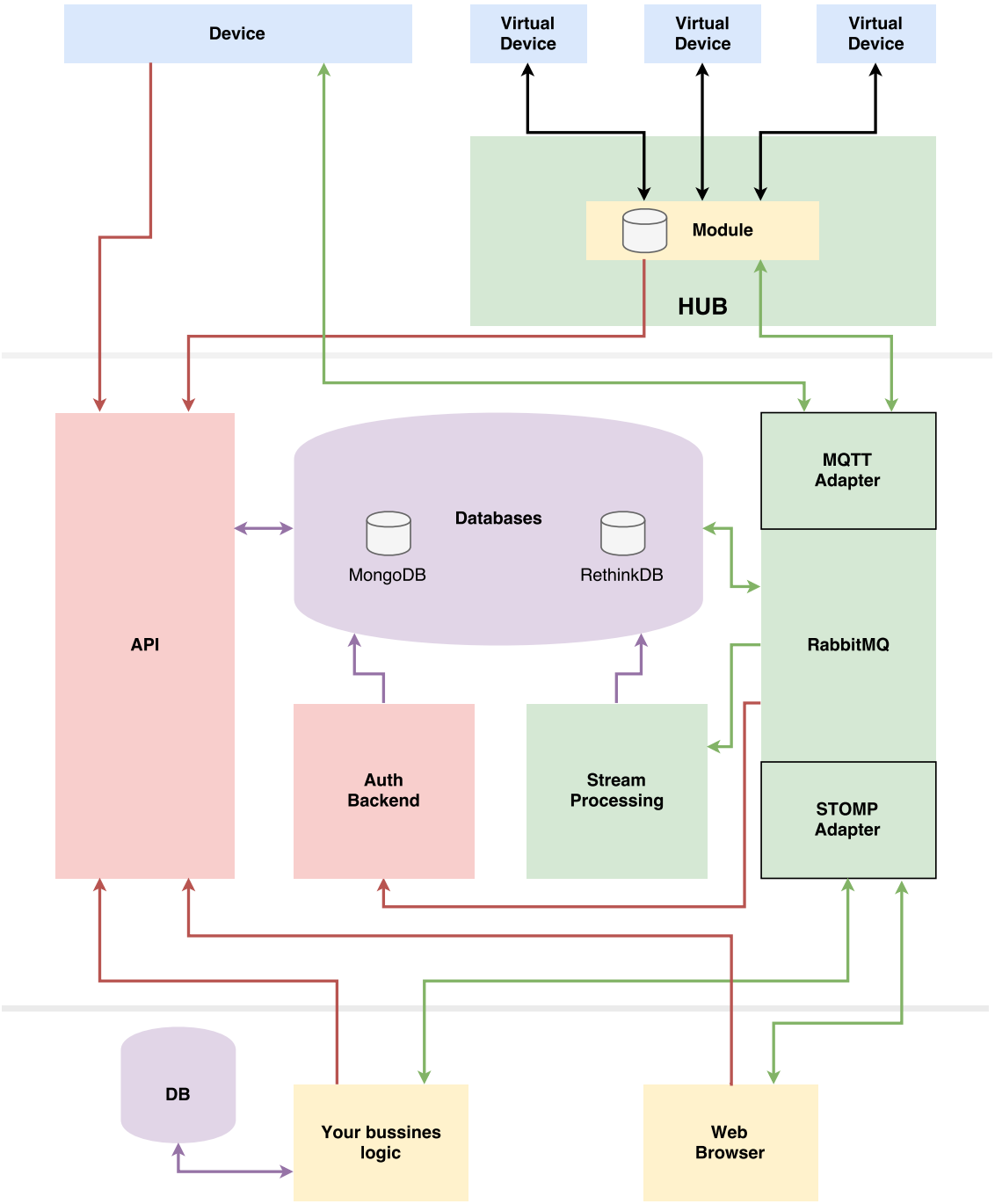

Figura 2.2 Estructura interna de la plataforma.

Componentes

En la Figura 3.2 se pueden ver los elementos de la plataforma, situados entre los dispositivos y la lógica de negocio del usuario. Tenemos los siguientes componentes:

API

Es el componente que permite a los usuarios comunicarse con la plataforma. Tanto los dispositivos como los servicios propios del usuario usarán la API. Es tarea de la API el registro de los dispositivos, el registro de

usuarios y otras tareas de gestión así como permite hacer consultas para obtener datos almacenados en la plataforma.

La API es una aplicación que ha sido desarrollada como parte de este trabajo. Se ha utilizado `Node.js` para su implementación.

Broker

Es el componente central de la arquitectura. Por él pasan los mensajes enviados por los otros elementos. Permite balanceo de carga entre múltiples instancias, persistencia, sincronización, etc.

Como *broker* se ha usado **RabbitMQ** ya que es uno de los más extendidos por su capacidad para escalar, su robustez y su extensibilidad.

Bases de datos

La persistencia del sistema. Múltiples bases de datos se encargan de almacenar los datos recibidos que deben ser persistidos. También será necesaria almacenar información de usarios y dispositivos registrados.

Para almacenamiento de credenciales se usará una base de datos `MongoDB` por ser una de las más populares. Para persitir los datos del usuario que llegan al sistema se usa `RethinkDB`. La característica más importante de esta base de datos es que, a diferencia de las bases de datos convencionales, las aplicaciones no tienen que consultarla de forma periódica para detectar nuevos datos o modificaciones en éstos, sino que es la propia base de datos la que notifica a las aplicaciones.

Backend de autorización

Debido a que es necesario controlar el acceso a la plataforma a los dispositivos y a las aplicaciones, se requiere un control de acceso. En este escenario, el control de acceso debe realizarlo el *broker*, en este caso en particular lo hará `RabbitMQ`. Gracias a un plugin para éste, es posible delegar la autenticación en una aplicación externa, de forma que el *broker* realizará una petición `HTPP` a una aplicación cuando un cliente se intente conectar, siendo esta aplicación quien decida si se permite o no.

El *backend de autorización* es la aplicacion encargada de tomar las decisiones. Tendrá conectividad con la base de datos donde se encuentran las credenciales y verificará que las conexiones son legítimas.

Esta aplicación también ha sido desarrollada como parte del trabajo, una vez más usando `Node.js` como plataforma.

Procesador de flujos

Este componente obtiene los datos en crudo de la cola de mensajes, los trata y los almacena en la base de datos. Puede ser interesante añadir algunos metadatos a la información recibida antes de almacenarse.

Al mismo tiempo, la base de datos notificará a este componente cuando termine de persistir el dato y este componente lo enviará a una cola del *broker* donde posteriormente podrá leerlo una aplicación que se conecte a la plataforma.

También se ha desarrollado como parte del trabajo en `Node.js`.

Protocolo MQTT, AMQP y STOMP

MQTT

MQTT[2] es un protocolo cliente/servidor que permite roles de publicador/suscriptor. Es ligero, abierto, simple y diseñado para ser fácil de implementar en el cliente. Estas características lo hacen ideal para su uso en múltiples situaciones, como por ejemplo, entornos donde la memoria y el ancho de banda son limitados, como la comunicación M2M (*Machine to machine*) o para dispositivos en el Internet de las Cosas.

Es un protocolo binario ligero que, en comparación con HTTP, tiene una mínima sobrecarga en cuanto a cabeceras para hacer más eficiente el tampaño de los paquetes. MQTT es muy fácil de implementar en el cliente. Esto encaja perfectamente en sistemas integrados con recursos limitados, de hecho, esto es uno de los objetivos que se buscaban cuando MQTT se creó.

El patrón publicador/suscriptor

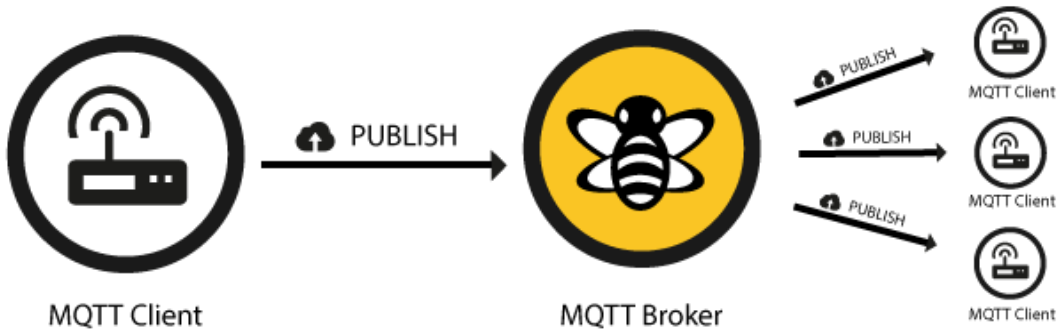

Figura 3.1 Patrón publicador/suscriptor.

El patrón publicador/suscriptor es una alternativa al sistema tradicional de cliente/servidor, donde un cliente se comunica directamente con un destino. Sin embargo, el patrón pub/sub desacopla al cliente que están enviando un mensaje (publicador) de otro cliente o más clientes que reciben mensajes (suscriptor). Esto quiere decir que publicador y suscriptor no son conscientes de la existencia del otro. Existe un tercer componente llamado *broker* que es conocido tanto por el publicador como el suscriptor que es el encargado de filtrar los mensajes que llegan y distribuirlos correctamente.

As already mentioned the main aspect in pub/sub is the decoupling of publisher and receiver, which can be differentiated in more dimensions: Como ya hemos mencionado, el pricipal aspecto del patrón publicador/suscriptor es el desacople en las siguientes dimensiones:

- Espacio desacoplado: Publicador y suscriptor no tiene porqué saber el uno del otro (IP, puerto, etc.).
- Tiempo desacoplado: Publicador y suscriptor no tienen porqué estar en ejecución al mismo tiempo.
- Sincronización desacoplada: Las operaciones de los dos componentes no se detienen durante envío o recepción.

En resumen, el patrón publicador/suscriptor desacopla el publicador o receptor de un mensaje y a través de los filtros para los mensajes se puede elegir qué clientes reciben qué mensajes.

Escalabilidad

El patrón publicador/suscriptor ofrece mejor escalabilidad que el patrón clásico de cliente/servidor. Esto se debe a que las operaciones en el broker pueden ser altamente paralelizadas. También es común el cacheo de mensajes y el enrutado inteligente. Sin embargo, es un reto el escalar a millones de conexiones, esto puede conseguirse mediante el uso de clústeres de *brokers* para distribuir la carga entre múltiples servidores.

Filtrado de mensajes

El filtrado de mensajes es el encargado de que sólamente los mensajes sean recibidos por los clientes que deben. Tenemos varias opciones de filtrado de mensajes:

- Filtrado basado en `topic`: El `topic` puede ser parte de cada mensaje. Cada cliente recibe sólamente mensajes del `topic` en el que está interesado. Los topics generalmente son cadenas de texto organizadas de forma jerárquica, por lo que se puede filtrar en función de una expresión.
- Filtrado basado en contenido: El filtrado se basa en el contenido específico del mensaje. Una gran desventaja de este método es que el contenido del mensaje debe ser conocido por el *broker*, cosa que no siempre ocurre, por ejemplo, cuando el contenido va cifrado.
- Filtrado por tipo: En lenguajes orientados a objetos es típico filtrar en base a un tipo o clase del mensaje o evento. En este caso el suscriptor podría recibir todos los mensajes de un tipo o subtipo.

En el caso de `MQTT` se utiliza el filtrado por `topic` así que cada mensaje contiene un `topic`, el cual es procesado por el `broker` para entregarlo a los clientes que se han suscrito a él.

`MQTT` tiene múltiples niveles de calidad de servicio (QoS). Se puede conseguir fácilmente que un mensaje sea entregado del cliente al *broker* o del *broker* al cliente. Sin embargo, existe la posibilidad de que nadie se suscriba a un `topic` en particular, en este caso depende el *broker* cómo se debe manejar la situación.

Diferencias con un sitema de colas

Existe confusión debido a su nombre en cuanto a sí es un protocolo de colas. Su nombre proviene de *MQseries*, un producto de *IBM* y no de *Message Queue*. Independientemente del nombre, existen diferencias entre `MQTT` y un sistema de colas:

- Un sistema de colas almacena un mensaje hasta que se consume: En un sistema clásico los mensajes se almacenan hasta que son tomados por un cliente (consumidor), esto en `MQTT` no ocurre ya que puede haber cero clientes suscritos a un topic y el mensaje es descartado.

- Un mensaje sólo es consumido por un cliente: Otra gran diferencia es el hecho de que en un sistema tradicional de cola de mensajes, los mensajes son consumidos por un único consumidor así la carga se puede distribuir entre múltiples procesos. En MQTT esto suele ser al contrario, todos los suscriptores reciben el mensaje si se han suscrito al topic.
- Las colas tienen nombres y deben crearse de forma explícita: Una cola es menos flexible que un *topic*, antes de usar una cola debe declarase explícitamente y sólo entonces se pueden consumir los mensajes. En MQTT los *topics* son extremadamente flexibles y son creados en el momento.

STOMP

STOMP (*Simple Text Orientated Messaging Protocol*), a diferencia de MQTT, está orientado a texto, como HTTP. El protocolo permite a clientes conectarse a un *broker* para que se puedan comunicar fácilmente entre diferentes tipos de aplicaciones y plataformas. En STOMP también se sigue un modelo publicador/suscriptor.

Se centra en tener un diseño simple y minimalista y que sea muy fácil de implementar en cualquier lenguaje. Al funcionar en modo texto, hacer un cliente puede ser tan sencillo como abrir una sesión Telnet para loguearse en cualquier *broker* de STOMP e interaccionar con él. Se dice que es posible crear un cliente de STOMP en un par de horas.

AMQP

AMQP[4] (Advanced Message Queuing Protocol) es un protocolo de mensages que permite a aplicaciones de clientes comunicarse con un broker.

Brokers

Los brokers de mensajes reciben los mensajes de los publicadores (aplicaciones que los publican, también conocidos como productores) y los enrutan hacia los consumidores (aplicaciones que los procesan).

Ya que es un protocolo de red, los publicadores, consumidores y el broker pueden residir en diferentes máquinas.

El modelo AQMP

El modelo de AMQP 0-9-1 tiene la siguiente visón de lo que ocurre:

Los mensajes se publican en *exchanges*, que se podrían ver como un buzón o una oficina postal. Los *exchanges*, al recibir mensajes, los distribuyen a las colas (*queues*) siguiendo unas reglas llamadas bindings. A continuación, el *broker* entrega los mensajes a los consumidores que están suscritos a las colas, o los propios consumidores consultan la cola y leen los mensajes.

Cuando se publica un mensaje, se pueden especificar diferentes atributos (*metadata*). Alguno de estos atributos pueden ser usados por el *broker*, sin embargo, el cuerpo del mensajes es completamente opaco para el *broker* y sólamente será usado por la aplicación que recibirá el mensaje.

Las redes pueden tener problemas y las aplicaciones puede fallas al procesar los mensajes, por eso mismo, el modelo AMQP hace uso de ACKs. Cuando un mensaje se entrega a un consumidor, éste debe notificar al broker que ha procesado el mensaje, ya sea de forma automática o cuando el desarrollador de la aplicación lo decida. El *broker* sólamente eliminará el mensaje de la cola cuando éste haya sido confirmado.

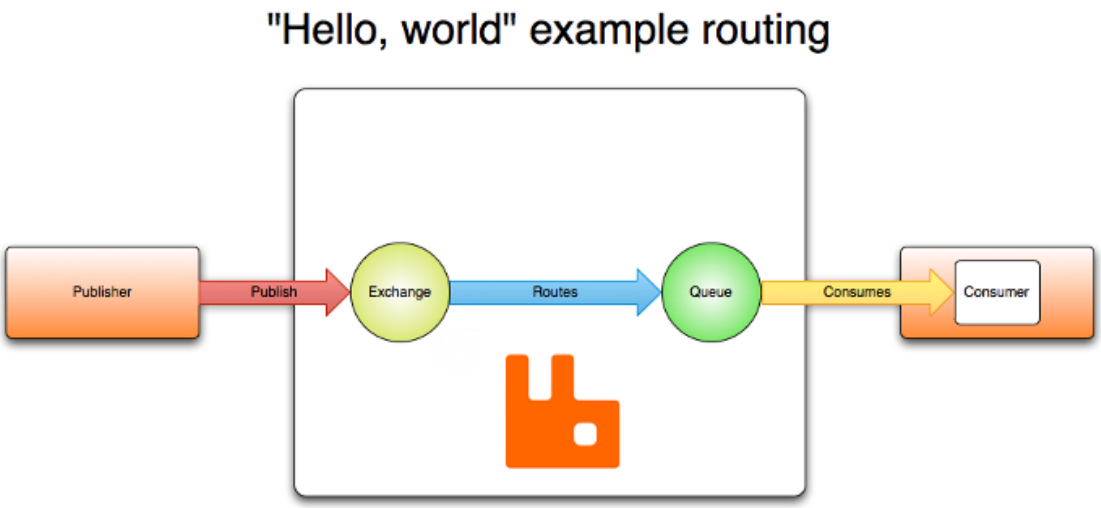

Figura 3.2 Ejemplo de enrutamiento en AMQP.

En algunas situaciones, por ejemplo, cuando un mensaje no puede ser enrutado, el mensaje puede ser devuelto al productor, descartado o, si el broker lo implementa, enviado a una cola especial llamada *dead letter queue*. Los consumidores pueden elegir cómo manejar situaciones como estas usando algunos parámetros a la hora de publicar los mensajes.

Las colas, los *exchanges* y los *bindings* son conocidas como **entidades de AMQP**-

Exchanges

Los *exchanges* son entidades de AMQP a donde llegan los mensajes. Éstos al recibir los mensajes los enrutan hacia cero o más colas. El algoritmo de enrutado depende del tipo de *exchange* y las reglas definidas (*bindings*). Existen cuatro tipos de *exchanges*:

- Direct exchange: `amq.direct`
- Fanout exchange: `amq.fanout`
- Topic exchange: `amq.topic`
- Headers exchange: `amq.match`

Independientemente del tipo de *exchange*, éstos son declarados con ciertos atributos, los más importantes son:

- Name: Nombre del *exchange*.
- Durability: Indica al *broker* que debe sobrevivir a reinicios.
- Auto-delete: Son eliminados sin no hay ninguna cola asociada.
- Arguments: Dependen del *broker*.

Direct Exchange

Los *exchanges* directos entregan los mensajes a las colas basándose en la `routing key`. Son ideales para enrutamiento *unicast*, aunque también se pueden usar para *multicast*. Funcionan de la siguiente manera:

- Una cola se conecta (establece un *binding*) al *exchange* con la `routing key` `K = R`

- El *exchange* directo suele ser usado para distribuir mensajes entre múltiples *workers* en modo *round robin*. Es importante ser conciente de que en `AMQP` se balancea entre consumidores y no entre colas.

Se puede ver de forma grágica en la Figura 3.2.

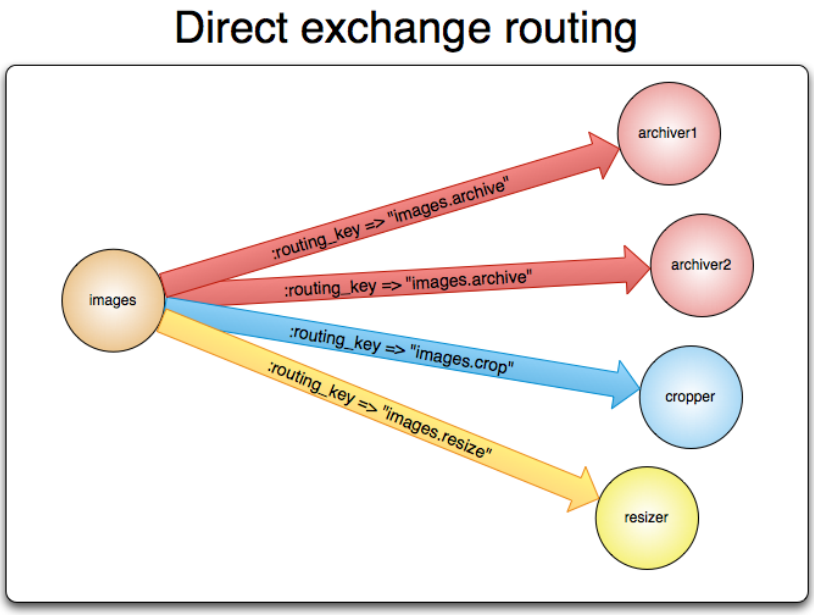

Figura 3.3 *Exchange* directo.

Fanout Exchange

Este tipo de *exchange* enruta todos los mensajes a todas las colas con las que está conectado, es decir, la `routing key` se ignora en este caso. Una copia de cada mensaje es enviado a cada una de las colas. Este tipo de *exchange* es ideal para el tráfico *broadcast*.

Se puede ver de forma grágica en la Figura 3.4.

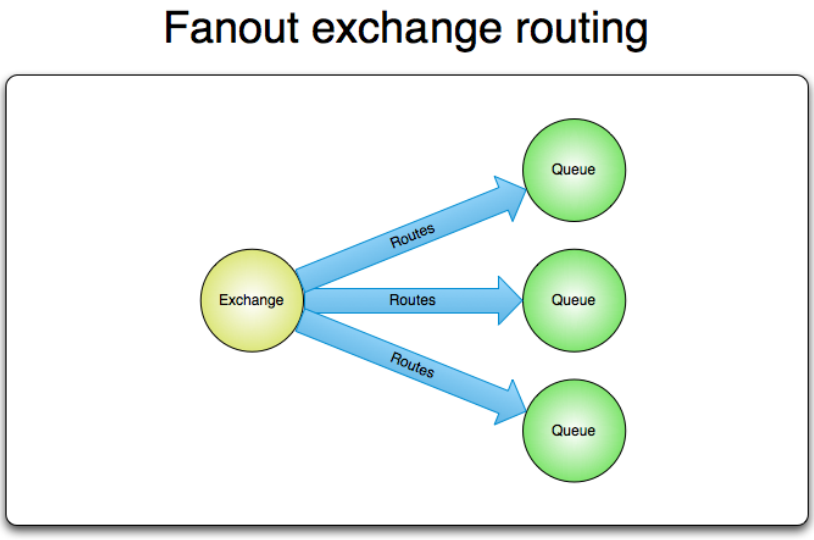

Figura 3.4 *Fanout exchange*.

Topic Exchange

En este caso, los mensajes se envían a una o varias colas, basándose en la `routing key` con la que una cola está conectada a un *exchange*. Este patrón se suele usar en modelos de publicador/suscriptor.

Header Exchange

El *header echange* está diseñado para enrutar mensajes basándose en múltiples atributos que viajan en las cabeceras de los mensajes en lugar de usar la `routing key`.

Colas

Las colas en `AMQP` son muy similares a las colas en otros sistemas de colas. Almacenan mensajes que pueden se consumidos por aplicaciones. Las colas comparten algunas propiedades con los *exchanges*, pero también tienen sus propios atributos:

- Name: Nombre de la cola.
- Durable: Indica al *broker* que deben sobrevivir a reinicios.
- Exclusive: Sólo se permite una conexión a la cola.
- Arguments: Dependen del *broker*.

Las colas deben ser declaradas antes de poder ser usadas. La declaración hace que la cola se cree si no existía anteriormente, si ya estuviese declarada, otra declaración no tendrá ningún efecto. Si se vuelve a declarar con otros atributos diferentes, se producirá una excepción.

Colas durables

Las colas durables se persisten a disco y sobreviven a reinicios del *broker*, en cambio, las colas que no son durables son llamadas transitorias. No todos los escenarios requieren que una cola sea durable.

La durabilidad de una cola no hace que los mensajes que sean enrutados hacia esa cola sean durables. Si el *broker* se reinicia, las colas durables volverán a declararse durante el inicio de forma automática, sin embargo, sólamente los mensajes persistidos podrán recuperarse.

Bindings

Los *bindings* son reglas usadas por los *exchanges* para enrutar los mensajes recibidos hacia las colas. Para que un *exchange* enrute un mensaje a una cola, dicha cola debe enlazarse con el *exchange*. Los *bindings* pueden tener atributos opcionales como las `routing keys`. La finalidad de una clave de enrutado es seleccionar ciertos mensajes publicados en un *exchage* que está enlazado con una cola, es decir, actúan como filtros.

Cuando un mensaje no se puede enrutar hacia una cola se descartará o se devolverá al productor, dependiendo de los atributos que el productor haya ajustado.

Mensages

Los mensajes en `AMQP` tienen atributos. Algunos de ellos son tan común que la especificación los define de forma que los desarrolladores no tienen que preocuparse por ellos.

- Content type
- Content encoding
- Routing key

- Delivery mode
- Message priority
- Message publishing timestamp
- Expiration period
- Publisher application id

Algunos atributos los usa el *broker*, pero la mayoría de ellos son para los consumidores que lo reciben. Algunos atributos son opcionales. Los atributos se fijan cuando el mensaje se publica.

Los mensajes también tienen una carga (los datos) que el *broker* trata como una ristra de *bytes*. El *broker* no inspecciona o modifica los datos. Es bastante común enviar los datos serializados en formatos como `JSON`, `MessagePack`, `Protocol BUffers`, etc. Para comunicar esta información se pueden usar los atributos `content-type` o `content-encoding`.

Los mensajes pueden ser publicados como persistentes, que hace que el *broker* los persista a disco. Si el servidor se reinicia, el sistema se asegura que los mensajes persistentes no se pierdan. Sólo por publicar un mensaje en un *exchange* durable o que la cola a la que el mensaje es enrutado sea durable no hace que el mensaje sea persitido, para ello el mensaje tiene que ser publicado como persistente. Hay que tener en cuenta que publicar mensajes persistentes afecta al rendimiento.

STOMP vs. MQTT vs. AMQP

Como se puede ver, `STOMP` y `MQTT` son protocolos muy semejantes, entonces, ¿Cuándo usaremos cada uno de ellos?. Pues los dispositivos usarán el protocolo `MQTT` ya que ofrece más funcionalidad y es muy fácil encontrar implementaciones para muchos tipos de dispositivos.

Sin embargo, para el lado de las aplicaciones se ha decidido usar `STOMP`. Esto quiere decir, que las aplicaciones del usuario que deseen conectarse al sistema para obtener los datos que los dispositivos envían mediante `MQTT` deben hacerlo usando `STOMP`. Esto es así porque una de las cosas que permite `STOMP` es que puede usar cualquier capa de transporte. Uno de los requisitos que se busca es que un navegador web pueda conectarse directamente a la plataforma y recibir datos. De esta forma no se requeriría que el servidor de la aplicación web obtenga los datos y los envíe al navegador, es decir, se pueden hacer aplicaciones *serverless*.

Desde un navegador web sólo se permite el tráfico `HTTP`, por lo que, en principio, no se puede usar `MQTT` ya que este viaja en `TCP`. Ahí es donde entra en juego `STOMP`, que está diseñado para funcionar independientemente de la capa que haya debajo, lo que lo hace ideal para funcionar sobre `websockets`.

Dicho esto el escenario es el siguiente:

- Los dispositivos se conectan mediante `MQTT` sobre `TCP` a la plataforma. Se suscriben y publican en *topics*.
- Las aplicaciones de la lógica del usuario se conecta a la plataforma usando `STOMP` bien sobre una sesión `TCP` o bien sobre *websockets* en caso de que quien realice la conexión sea un navegador. Se suscriben y publican en *topics*.
- La plataforma, de forma interna, convierte los mensajes `MQTT` y `STOMP` a `AMQP` y trabaja con este sistema de forma interna. Veremos qué es `AMQP` en el siguiente capítulo.

Broker y autorización

RabbitMQ

Como *broker* se usará `RabbitMQ`. El *broker* es una de los componentes más importantes de la plataforma, por no decir el más importante de todos. Cada mensaje que llega al sistema desde un elemento externo, ya sea un dispositivo o una aplicación del usuario debe hacerlo a través del *broker*. Además, los servicios internos de la plataforma también usan el *broker* para la comunicación.

`RabbitMQ` nos ofrece todo el sistema de colas de `AMQP` que ya hemos descrito anteriormente, por lo que todos los datos de la plataforma estarán en colas. Esto permite soportar picos de tráfico, ya que si los servicios o las bases de datos no son capaces de soportar una carga elevada los mensajes simplemente se encolarán y no serán descartados.

Además, gracias a unos plugins oficiales, es posible configurar a `RabbitMQ` para que sea capaz de recibir mensajes `MQTT` y `STOMP`. Es importante aclarar que en realidad no existe un *broker* como tal para estos protocolos, sino que realmente se hace una adaptación a `AMQP`, es decir, que aunque los mensajes lleguen en formato `MQTT`, éstos se guardarán en colas `AMQP` y los leerán servicios mediante este protocolo. Los servicios del núcleo de la plataforma no implementan ni `MQTT` ni `STOMP`.

Un ejemplo de comunicación sería:

```
DISPOSITIVO -> ADAPTADOR MQTT -> AMQP -> STREAM PROCESSOR -> RETHINKDB ->
-> STREAM PROCESSOR -> AMQP -> STOMP -> NAVEGADOR WEB
```

Por supuesto en sentido contrario también funcionaría.

Configuración

Código 4.1 Configuración de RabbitMQ.

```
[
  { rabbit, [
      { loopback_users, []},
      { auth_backends, [rabbit_auth_backend_http]}
    ]
  },
  { rabbitmq_mqtt, [
      { subscription_ttl, undefined},
      { tcp_listeners,  [1884]}
    ]
  },
  { rabbitmq_auth_backend_http, [
      { http_method, post},
      { user_path,   "http://auth-backend:3000/auth/user"},
      { vhost_path,  "http://auth-backend:3000/auth/vhost"},
      { resource_path, "http://auth-backend:3000/auth/resource"}
    ]
  }
].
```

Backend de autorización

Puesto que el sistema debe ser seguro y no debe permitir que un usuario pueda leer mensajes de otros debe existir un control de acceso al *broker*. En principio `RabbitMQ` permite autorización y autenticación sin ningún añadido, sin embargo, para el uso que se le va a dar en este escenario es necesario un control con mayor granularidad que el que se puede conseguir por defecto. Otro problema sería sincronizar la base de datos de usuarios con la base de datos de `RabbitMQ`.

Como alternativa se ha decidido usar un plugin que permite delegar la autenticación a un servicio externo, en este caso un servidor web. Cuando un usuario intente realiar una acción en el *broker*, éste realizará una petición al servidor de autorización y será éste último quien decida si se autoriza o se rechaza, simplemente enviado *allow* o *deny* en el cuerpo del mensaje de respuesta.

Para tal caso se ha desarrollado usando `Node.js` un servidor web tiene acceso a la base de datos de usuarios y responde a las peticiones que realiza `RabbitMQ`. Cabe destacar que el plugin implementa una caché para no realizar peticiones cuando el volumen de mensajes sea muy grande. Para cada conexión de un usuario sólo se consulta una única vez y para los mensajes siguientes se usa la caché.

En este caso, por ser una aplicación sencilla sólamente se ha usado el *framework* `Express`[5] para `Node.js`. `Express` es un *framework* minimalista que permite crear aplicaciones web de forma rápida. El funcionamiento

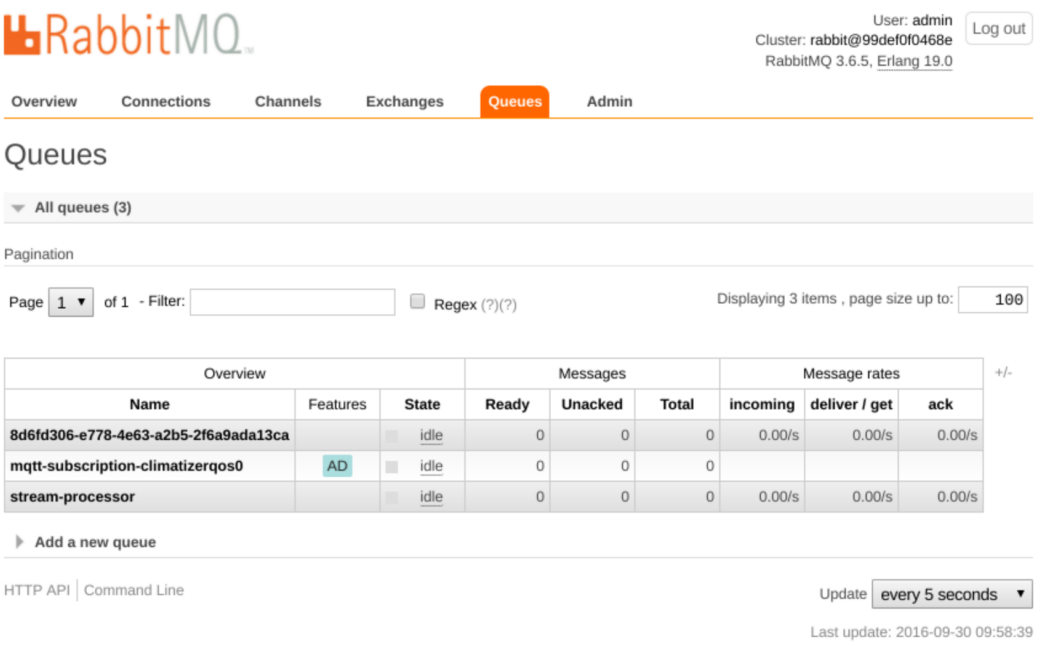

Figura 4.1 RabbitMQ.

de la aplicación es tan sencillo como verificar que en la base de datos exista el usuario que intenta acceder a un mensaje y que tenga permisos para hacerlo.

Código 4.2 Fragmento de código del backend de autorización.

```
// const Fleet = require('./models/fleet');
const Device = require('./models/device');

module.exports = class Router {
  constructor(app, logger) {
    this.app = app;
    this.logger = logger;

    this.app.post('/auth/user', (req, res) => {
      if (req.body.username === 'admin') {
        res.send('allow management');
        return;
      }

      Device.findById(req.body.username, (err, device) => {
        if (err) throw err;
        if (!device) {
          res.send('deny');
          return;
```

```
        }

        if (device.secret === req.body.password) {
          res.send('allow');
        } else {
          res.send('deny');
        }
      });
    });

    this.app.post('/auth/vhost', (req, res) => {
      res.send('allow');
    });

    this.app.post('/auth/resource', (req, res) => {
      if (req.body.resource === 'topic') {
        const fleetId = req.body.name.split('/')[0];

        Device.findById(req.body.username, (err, device) => {
          if (err) throw err;
          if (!device) {
            res.send('deny');
            return;
          }

          if (device.fleetId === fleetId) {
            res.send('allow');
          } else {
            res.send('deny');
          }
        });
      } else {
        res.send('allow');
      }
    });
  }

  listen(port) {
    this.logger.info(`Listening on port: ${port}`);
    this.app.listen(port);
  }
};
```

API

Funcionamiento

Lo primero y más importanta del sistema son los usuarios. Serán quienes exploten las funcionalidades del sistema. Puesto que la finalidad de la plataforma es establecer una comunicación entre los dispositivos de un cliente y su lógica, tenemos dos roles:

- *Dispositivos:* Los componentes que obtienen información o controlan elementos.
- *Lógica:* El software del usuario que procesan la información obtenida y controlan los dispositivos.

Para que el usuario pueda interactuar con la plataforma para conocer el estado de los dispositivos se proveerá de una API RESTful que permita:

- *Registro de usuarios*
- *Administración de flotas*
- *Monitorización de dispositivos*

Registro de usuario

Para crear un usuario se enviará un mensaje HTTP a la API con el contenido del código 5.1:

Código 5.1 Petición para creación de usuario.

```
POST /users

{
  "email": "yo@dominio.com",
  "password": "secreto"
}
```

Una vez registrados, se puede hacer login y realizar peticiones autenticadas a la API. Para realizar un *login* debemos enviar el contenido del código 5.1.

Código 5.2 Petición de login.

```
POST /users/login

{
  "email": "yo@dominio.com",
  "password": "secreto"
}
```

Como respuesta se obtiene el `acess_token` y `user_id`:

Código 5.3 Respuesta a login.

```
{
  "id": "<access_token>",
  "ttl": 1209600,
  "created": "2013-12-20T21:10:20.377Z",
  "userId": "<user_id>"
}
```

A partir de ahora podemos realizar peticiones autorizadas enviando en la cabezera del mensaje `HTTP` el `access_token` el campo `Authorization`.

Obtención de Fleet Key

Para crear una flota es necesario haber realizado el proceso de registro y disponer de un `user_id` y `acess_token`. Ser realizará la petición indicada en el código 5.4.

Código 5.4 Creación de `fleet_key`.

```
POST /users/<user_id>/fleets

Authorization: <access_token>
```

Si todo ha ido correctamente se obtendrá como respuesta el mensaje de respuesta 5.5

Código 5.5 Obtencion de `fleet_key`.

```
{
  "uuid": "<fleet_uuid>"
}
```

Registro de dispositivos

Cuando el dispositivo se inicie por primera vez, deberá realizar un proceso de registro en el que usará la `fleet_key` para poder obtener un `uuid` y un `secret`. Para que un dispositivo pueda registrarse deberá ser capaz de comunicarse mediante `HTTP`, en caso contrario, será necesario el uso de un dispostivo intermedio capaz de realizar esta labor en nombre del dispositivo.

A este dispositivo intermedio lo llamaremos `hub` que puede ser cualquier dispositivo que sea capaz de enviar mensajes `HTTP` y `MQTT` y reenviarlos en el protocolo que sea a los dispositivos reales. Podría ser, por ejemplo, un *router* o una *Raspberry Pi*.

Un caso podría ser un *Arduino* que se comunicase mediante un módulo `NRF24` con un programa que se ejecute en una *Raspberry Pi*. Será dicho programa de la *Raspberry Pi* el responsable de realizar el registro, de forma que el dispositivo no tiene porqué tener conocimiento del proceso ni de la plataforma. Cabe destacar que, en este caso, será la aplicación en cuestión quien debe conocer la `fleet_key`.

Esto puede ser útil en caso de tener dispositivos que no podemos adaptar al sistema, bastaría con crear este intermediario para poder usar el dispositivo con la plataforma.

En cualquier caso, el procedimiento será enviar un mensaje `HTTP` a la API con el contenido de la petición 5.6.

Código 5.6 Registro de dispositivo.

```
POST /fleets/<fleet_key>/registerMote
```

Código 5.7 Respuesta de registro de dispositivo.

```
{
  "uuid":  "<Nuevo UUID de dispositivo>",
  "secret": "<Nuevo secreto>"
}
```

Una vez obtenidas las credenciales para poder enviar datos deberían persistirse para futuros usos.

Desarrollo

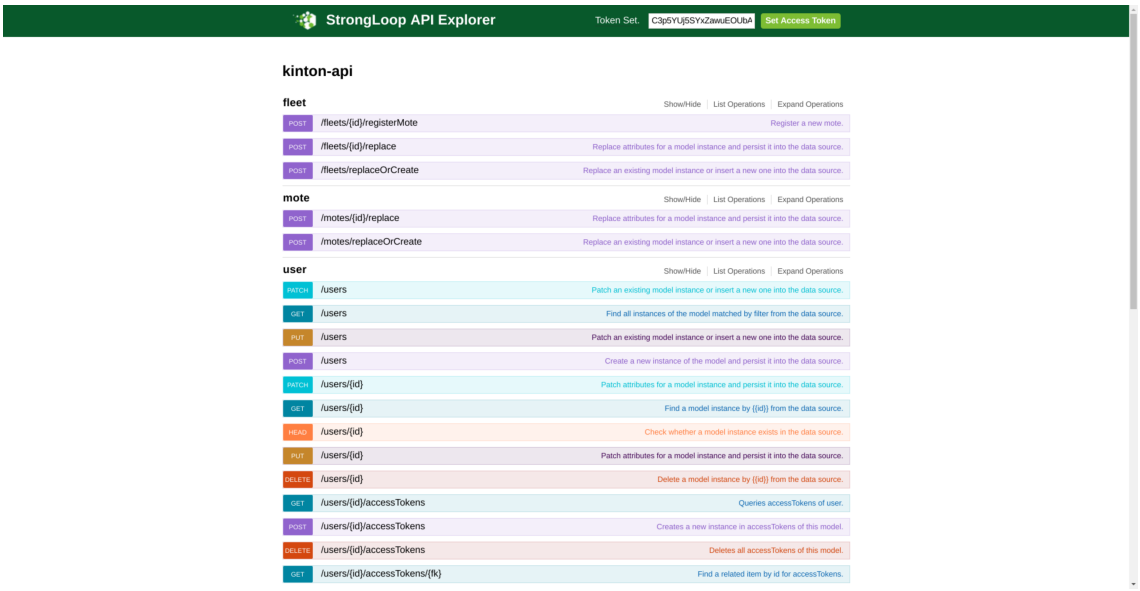

Figura 5.1 Explorador de la API.

Para el desarrollo de la `API` se ha usado `Node.js` y `loopback.io`[1] como *framework*. `loopback.io` es un *framework* diseñado específicamente para el desarrollo de *APIs REST* de forma muy rápida ya que ahorra considerablemente la lógica a desarrollar.

Las *APIs* se construyen mediante archivos de configuración donde se declara cómo debe comportarse y el *framework* se ocupa de toda la lógica. También abstrae de las bases de datos, mediante el uso de plugins simplemente se configura la conexión y el desarrollador no tiene que preocuparse de nada más. En caso de que se requiera implementar una lógica, el *framework* lo permite, ya que es totalmente extensible.

Este *framework* es ideal para construir *APIs* que usen modelos que se creen, actualicen, eliminen, etc. Permite también establecer una jerarquía entre modelos y también implementa un potente sistema de control de acceso a los recursos expuestos por la *API*.

Código 5.8 Declaración de una flota en Loopback.io.

```
{
  "name": "fleet",
  "plural": "fleets",
  "base": "PersistedModel",
  "idInjection": true,
  "options": {
    "validateUpsert": true
  },
  "mongodb": {
   "collection": "fleets"
  },
  "properties": {
    "uuid": {
      "type": "string",
      "id": true,
      "index": true
    }
  },
  "validations": [],
  "relations": {
    "user": {
      "type": "belongsTo",
      "model": "user",
      "foreignKey": "ownerId"
    },
    "mote": {
      "type": "hasMany",
      "model": "mote",
      "foreignKey": "fleetId"
    }
  },
  "acls": [],
  "methods": {}
}
```

Código 5.9 Declaración de usuario en Loopback.io.

```
{
  "name": "user",
  "plural": "users",
  "base": "User",
  "idInjection": true,
  "options": {
    "validateUpsert": true
  },
  "mongodb": {
   "collection": "users"
  },
  "properties": {},
  "validations": [],
  "relations": {
    "fleet": {
      "type": "hasMany",
      "model": "fleet",
      "foreignKey": "ownerId"
    }
  },
  "acls": [{
    "principalType": "ROLE",
    "principalId": "$owner",
    "permission": "ALLOW",
    "property": "__get__fleet"
  }, {
    "principalType": "ROLE",
    "principalId": "$owner",
    "permission": "ALLOW",
    "property": "__create__fleet"
  }],
  "methods": {}
}
```

Código 5.10 Declaración de dispositivo en Loopback.io.

```
{
  "name": "mote",
  "plural": "motes",
  "base": "PersistedModel",
  "idInjection": true,
  "options": {
    "validateUpsert": true
  },
  "mongodb": {
   "collection": "devices"
  },
  "properties": {
    "secret": {
      "type": "string",
      "required": true,
      "description": "secret of the device"
    },
    "uuid": {
      "type": "string",
      "id": true,
      "generated": false,
      "required": true,
      "index": true,
      "description": "UUID of the device"
    }
  },
  "validations": [],
  "relations": {
    "fleet": {
      "type": "belongsTo",
      "model": "fleet",
      "foreignKey": "fleetId"
    }
  },
  "acls": [],
  "methods": {}
}
```

Bases de datos

En el proyecto se han utilizado dos bases de datos, una de ellas se utilizará para almacenar datos que usará el sistema de forma interna, como usuarios, permisos, *topics*.

Para almacenar los datos que envían los dispositivos y aplicaciones se utilizará otra base de datos diferente. Lo que se guarda en esta base de datos es opaco para el sistema, se almacena como un buffer y en ningún momento es utilizado por la propia plataforma.

MongoDB

Es una base de datos de propósito general, se utilizará para almacenar credenciales de usuarios, ids de los dispositivos y, en general, cualquier tipo de datos que deba consultar el sistema para su correcto funcionamiento.

Se elige `MongoDB` por su popularidad ya que es muy fácil de conectar con frameworks ya existentes, como `loopback.io` y tiene una gran cantidad de librerías para muchos lenguajes.

Figura 6.1 MongoDB.

RethinkDB

`RethinkDB`[6] una base de datos moderna con un enfoque diferente al resto de las bases de datos actuales. En el paradigma clásico de las bases de datos, el cliente se conecta a ellas y hace consultas. Al igual que ocurre con el paradigma clásico de la web, es necesario estar lanzando peticiones constántemente por lo que se tienen instantes de tiempo en los que el sistema no está sincronizado.

Con `RethinkDB` los clientes que se conectan a las bases de datos pueden suscribirse a cambios en los datos almacenados, lo cual nos permite tener un sistema más asíncrono, escalable y más adecuado para aplicaciones en tiempo real. Al no tener que hacer constantes peticiones a la base de datos, se alivia la carga y permite al sistema escalar de forma más fácil.

Como se puede aprecer en la Figura 6.2 los datos de usario son tratados por la base de datos como una ristra de bytes.

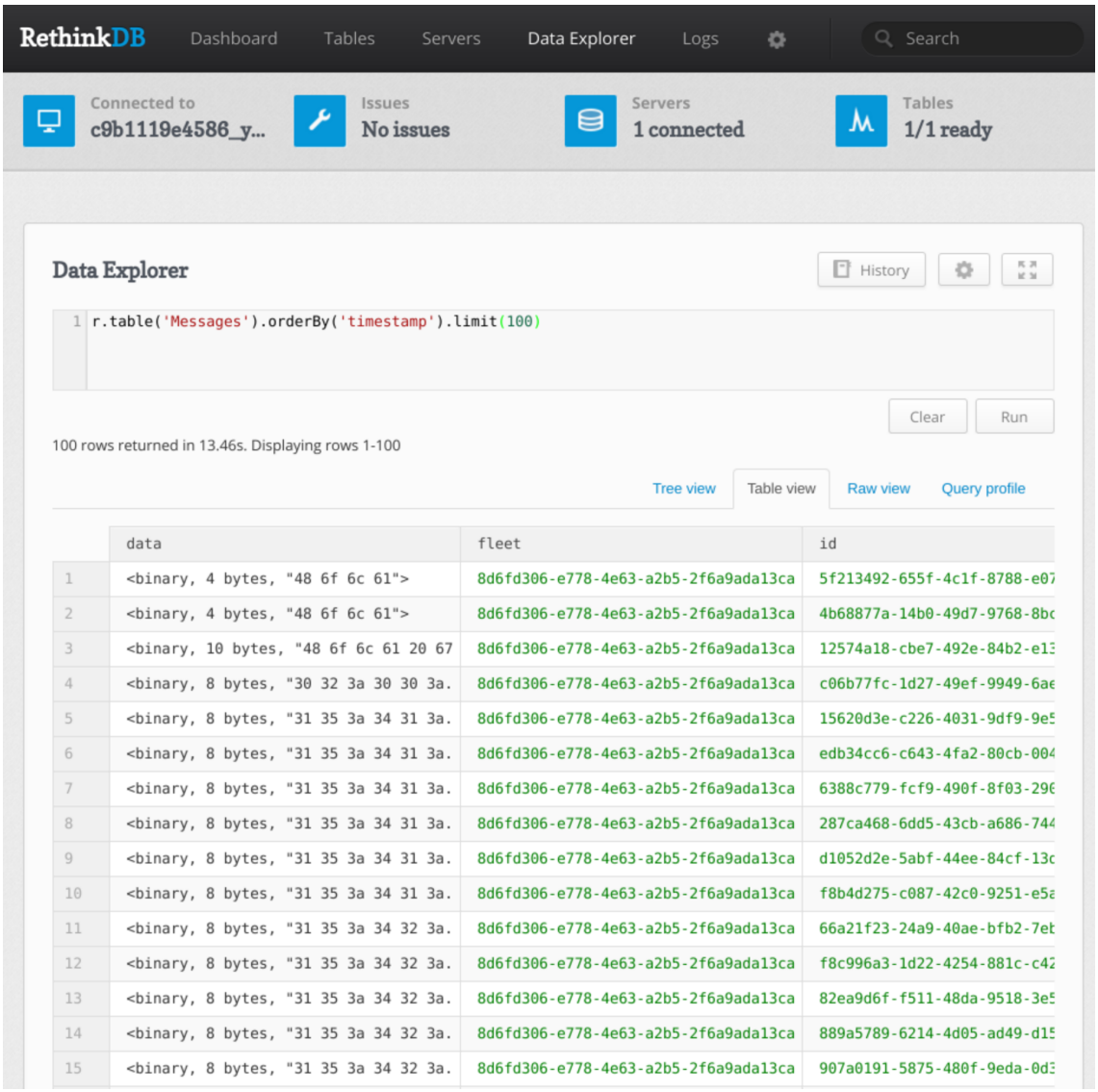

Figura 6.2 Realizando una consulta a RethinkDB.

Dispositivos y Módulos

Dispositivos

Los dispositivos son elementos externos al sistema. El propio sistema está diseñado para proveer servicios a estos dispositivos, por lo que podemos verlos como un especie de usuario de la plataforma.

Cada dispositivo debe estar registrado en el sistema para poder interactuar con él, por lo que deberán implementar un proceso en el cual puedan darse de alta en la plataforma. Cada dispositivo estará asociado a un usuario, esto quiere decir que a la hora de registrarse, el dispositivo debe proveer cierta información para que la plataforma pueda determinar a qué usuario quedará asociado.

Flotas

La forma de asociar un dispositivo a un usuario es mediante las **flotas**. Las flotas son agrupaciones independientes de dispositivos que puede crear el usuario para gestionarlos. Cada usuario puede tener múltiples flotas. Al crear una flota, se generará de forma automática una `fleet_key`, que es una clave única que representa a cada flota. Cuando un dispositivo ejecuta el proceso de registro deberá proveer la clave para que el sistema lo reconozca como perteneciente a su flota correspondiente a la hora de registrarlo en la base de datos.

Dispositivos de diferentes flotas no pueden comunicarse entre ellos de forma directa, es decir, si en una flota un dispositivo publica en el *topic* `temperature` y otro dispositivo de una flota diferente está suscrito a este *topic* no recibirá el mensaje. Sin embargo, siempre será posible que una aplicación o dispositivo pertenezca a ambas flotas y sea capaz de reenviar los mensajes.

Módulos

Los módulos son aplicaciones que simulan ser dispositivos. Su objetivo es actuar como intermerdiario entre los dispositivos reales y la plataforma.

El dispositivo en el que corren los módulos se denominará hub. Una Raspberry Pi puede ser muy adecuado como hub ya que puede implementar los protocolos necesarios y puede comunicarse con los dispostivos reales usando un adaptador Bluetooth o ZigBee.

Existen varios escenarios donde puede ser interesante usar módulos:

Caso 1

Los dispositivos no tienen capacidad de enviar mensajes `HTTP` o `MQTT`, que son los protocolos usados por la plataforma. En tal caso, se puede crear un módulo que corra en otro dispositivo capaz de implementar estos protocolos y, al mismo tiempo, sea capaz de comunicarse con los dispositivos reales. Estos módulos tienen la responsabilidad de registrarse como si fuesen dispositivos reales, obtener datos y reenviarlos a los dispositivos correspondientes. Útiles para dispositivos que usan `Bluetooth` o `ZigBee` como protocolo de comunicación.

Caso 2

Los dispositivos ya existen y no es posible modificar el software para que se comunique con la plataforma directamente. Por ejemplo porque los dispositivos los fabrica un tercero. En este caso, el módulo tiene una función de "pasarela" entre los dispositivos reales y la plataforma.

Conexión al broker

Para enviar datos a la plataforma será necesario disponer de los siguientes datos:

- `host`: El nombre o dirección IP del equipo donde está desplegada la plataforma.
- `puerto`: El puerto donde se reciben los mensajes de los dispositivos mediante `MQTT`.
- `user`: El `uuid` del dispositivo, generado en el proceso de registro.
- `password`: La contraseña del dispositivo, generada en el proceso de registro.
- `topics`: `<fleet_key>/#`. Donde "#" es el nombre original del topic.

A la hora de publicar mensajes en un topic, el dispositivo debe preceder el topic con el `fleet_key`, por ejemplo, si se quiere publicar al topic "temperature", deberá enviarse el mensaje al topic `<fleet_key>/temperature`.

Para suscribirse a un topic se aplicará la misma regla, precediendo el nombre del topic original por el Fleet Key.

Se debe usar usar TLS ya que sino las credenciales viajarán en texto plano y pueden ser capturadas por un tercero.

Ejemplo de uso

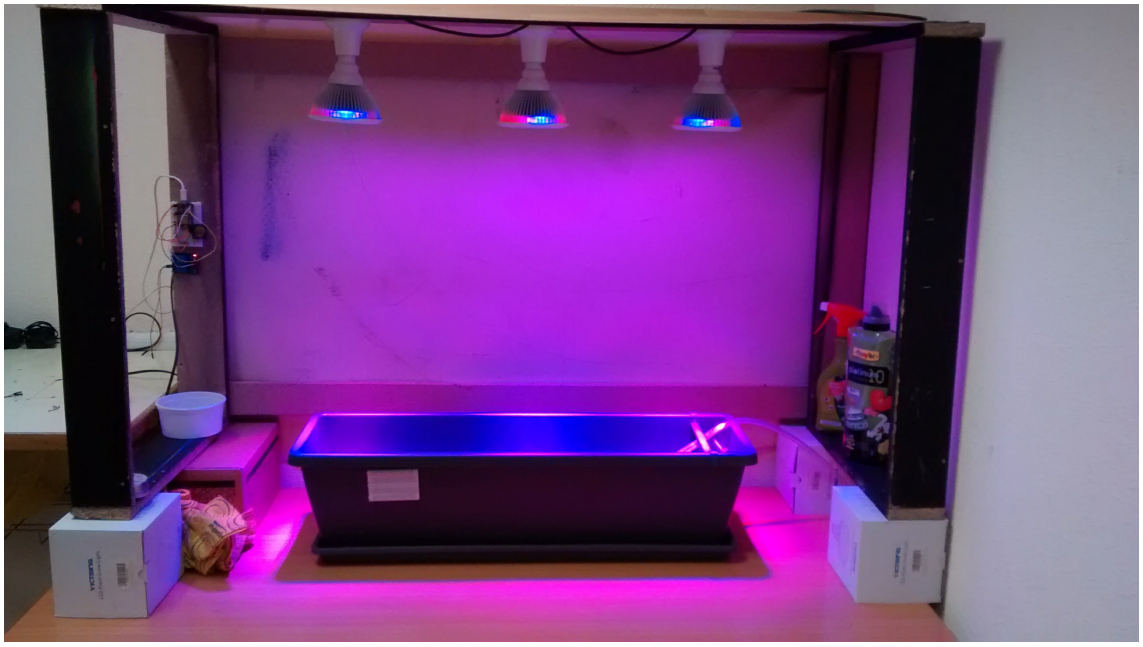

Figura 8.1 Sistema de riego inteligente y control de luz.

Una vez explicado el funcionamiento del sistema es conveniente realizar un ejemplo real en el que pueda apreciarse el sistema en completo funcionamiento. Para tal efecto se ha decidido diseñar un sistema de riego inteligente que sea capaz de monitorizar unas platas y actuar sobre ellas. Dado que el objetivo de este trabajo no es construir este sistema no se entrará en detalles de implementación, sino que se centrará en describir cómo el sistema aprovecha las funcionalidades de la plataforma desarrollada para realizar su cometido.

Dispositivos

El sistema estará compuesto por múltiples dispositivos que se encuentran físicamente en el lugar de las plantas que se quieren monitorizar. Estos dispositivos dispondrán de múltiples sensores que captarán la humedad del suelo de la planta, así como la luminosidad y la temperatura. Estos dispositivos contarán, además, con diferentes módulos de comunicación que les permitirán registrarse y conectarse a la plataforma diseñada, en

algunos casos de forma directa y en otros a través de una pasarela que lleve instalada un **módulo**.

A continuación se listan los dispositivos que participan en el sistema:

- **Sensor de humedad**, temperatura o luminosidad: Reporta lecturas de sensores que se encuentran en una zona circundante a una planta.
- **Controlador de luz**: Mediante un relé es capaz de controlar la luz que reciben las plantas. Usa un temporizador para encender y apagar las luces a una hora configurada. Tiene tres modos de funcionamiento: "encendido", "apagado", "temporizador".
- **Control de riego**: Usando una bomba de agua permite regar las plantas. Usa también un reloj para regar a unas horas configuradas o puede accionarse de forma manual.

Los dispositivos pueden funcionar de forma autónoma, es decir, no necesitan recibir órdenes para realizar su cometido, de esta forma se consigue que no dependan de conexión a internet constante. La conexión con la nube se usará entonces para poder ser configurados en remoto y/o accionados de forma manual.

Servicios

Usando `Node.js` se ha desarrollado un servicio que recibe la información emitida por los dispositivos. Cada vez que cambia el estado de un elemento, por ejemplo, la luz se enciende o se comienza a regar; el dispositivo emite un mensaje que describe el evento.

Cuando se reciben eventos, la aplicación genera alertas de forma que se puede saber en tiempo real que se ha regado o se ha encendido la luz. Esta aplicación realiza dos funcionalidades:

- Si recibe una lectura de un sensor la almacena en una base de datos.
- Si recibe la ocurrencia de un evento envía una alerta

Alertas

Para el envío de alertas se usa `IFTTT`[3]. Se trata de un servicio que permite conectar unos servicios con otros por medio de recetas. Uno de los servicios disponibles es `Pushbullet` que permite recibir notificaciones `PUSH` en los *smartphones*. Mediante una receta que conecta la aplicación desarrollada con `Pushbullet` podemos recibir los eventos directamente en los dispositivos móviles.

En la Figura 8.3 se puede ver mensajes recibidos en `Pushbullet` de `IFTTT` anunciando eventos.

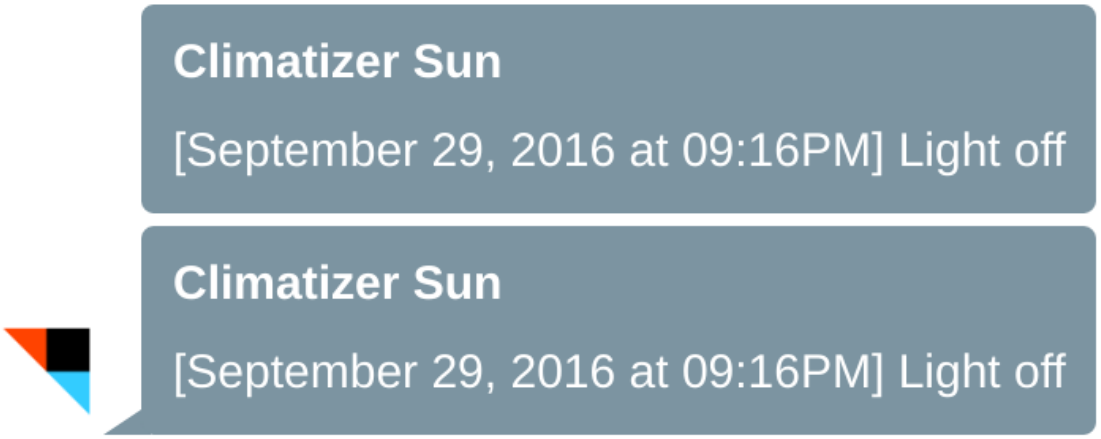

Figura 8.3 Recibiendo mensajes de IFTTT en Pushbullet.

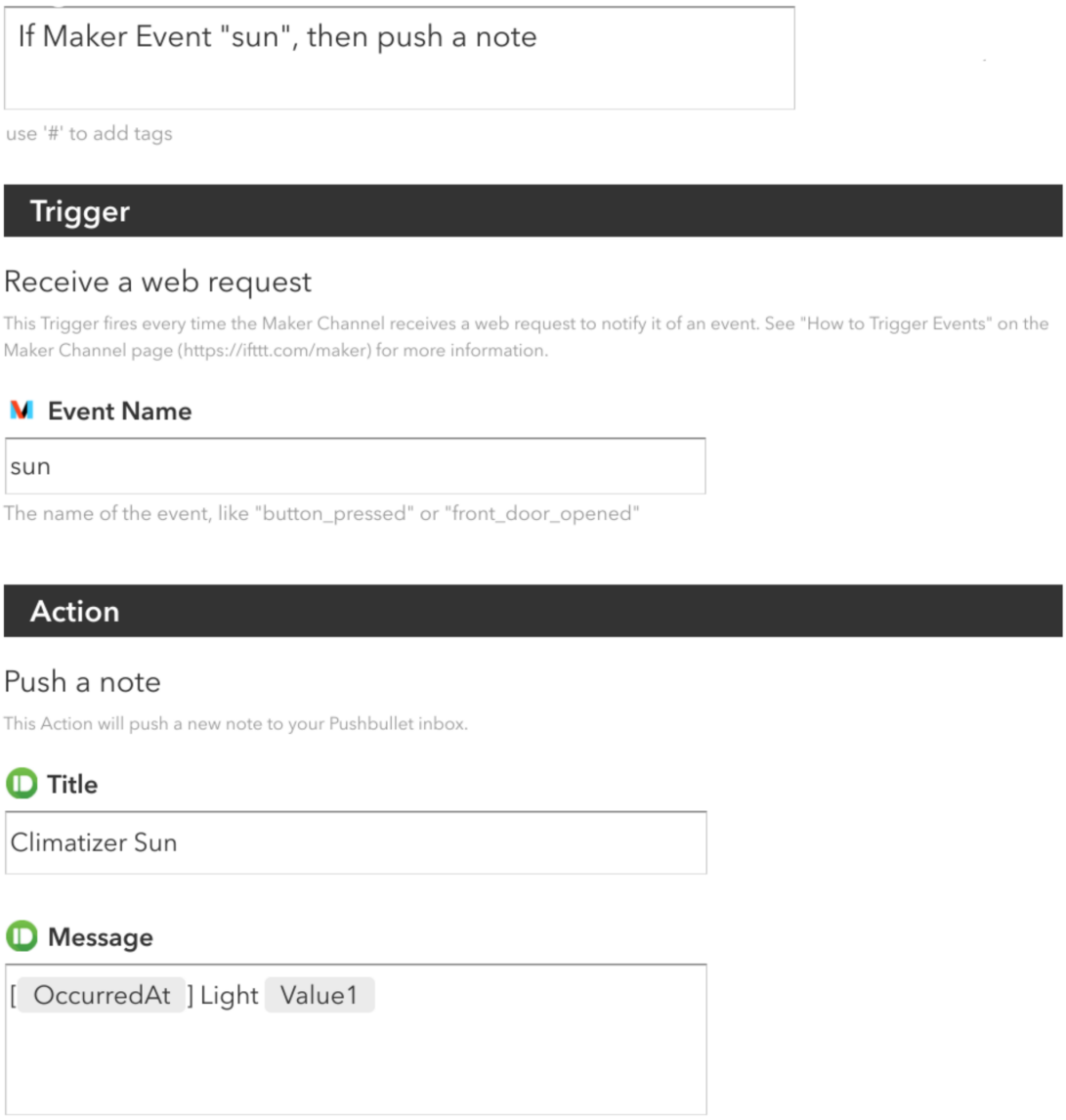

Figura 8.2 Receta de IFTTT para recibir alertas en Pushbullet.

Lecturas

Cada vez que se recibe una lectura de un sensor nuevo se almacena el valor en una base de datos `InfluxDB`, que está optimizada para almacenar series de valores. Posteriormente se puede utilizar un visualizador para representar estos valores en gráficas. Para ello se puede utilizar cualquier software que sea capaz de leer los datos almacenados en `InfluxDB`, en este caso se ha usado `Grafana` como se puede observar en la Figura 8.4.

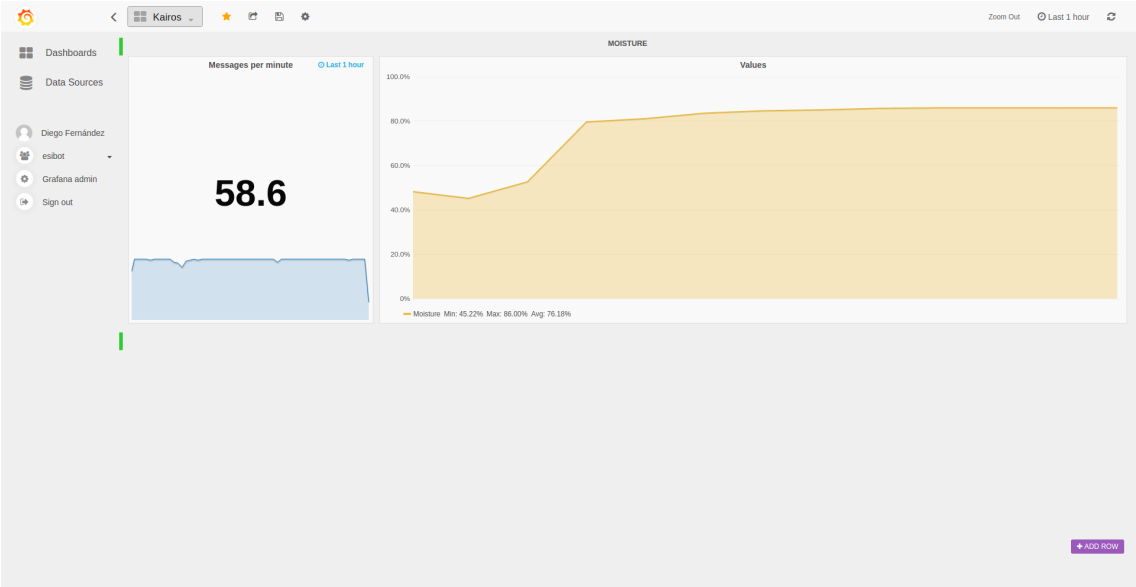

Figura 8.4 Grafana.

Código 8.1 Fragmento de código del backend de autorización.

```
const stompit = require('stompit');
const winston = require('winston');

const KINTON_HOST = process.env.KINTON_HOST || 'stream.kinton.xyz';
const KINTON_PORT = process.env.PORT || 61613;
const FLEET_KEY = process.env.FLEET_KEY || 'b193020b-8cb0-46ae-a91e-53f33ac07
    afc';
// const APP_KEY = '';
// const APP_SECRET = '';

const logger = new(winston.Logger)({
  transports: [new(winston.transports.Console)({
    level: 'debug',
    colorize: true,
    prettyPrint: true,
  })],
});

// STOMP Connection options
const connectOptions = {
  host: KINTON_HOST,
  port: KINTON_PORT,
  connectHeaders: {
    host: '/',
```

```
    login: 'admin', // TODO Use APP_KEY
    passcode: '', // TODO Use APP_SECRET
  },
};

// Connect to Kinton via STOMP
stompit.connect(connectOptions, (connectionError, stomp) => {
  if (connectionError) throw connectionError;

  // Subscribe to our fleet
  stomp.subscribe({
    destination: `/amq/queue/${FLEET_KEY}`, // Queue to connect
    ack: 'client-individual',           // Manual ACK per message
  }, (subscriptionError, message) => {
    if (subscriptionError) throw subscriptionError;
    stomp.ack(message);

    message.readString('utf-8', (readStringError, data) => {
      if (readStringError) throw readStringError;

      const date = new Date(message.headers.timestamp * 1000);
      logger.info('------------------------------------------------------------')
          ;
      logger.info(`MESSAGE UUID: ${message.headers['amqp-message-id']}`);
      logger.info(`DATA: ${data}`);
      logger.info(`TOPIC: ${message.headers.topic}`);
      logger.info(`TIMESTAMP: ${date.toLocaleString()}`);
    });
  });
});
```

Envío de comandos

Además de poder recibir datos de los diferentes dispositivos también se ha visto que se posible enviarle mensajes a dichos dispositivos. En el ejemplo que estamos tratando se puede enviar un mensaje para que el dispositivo comience a regar las plantas. El proceso sería similar al visto en el código 8.1.

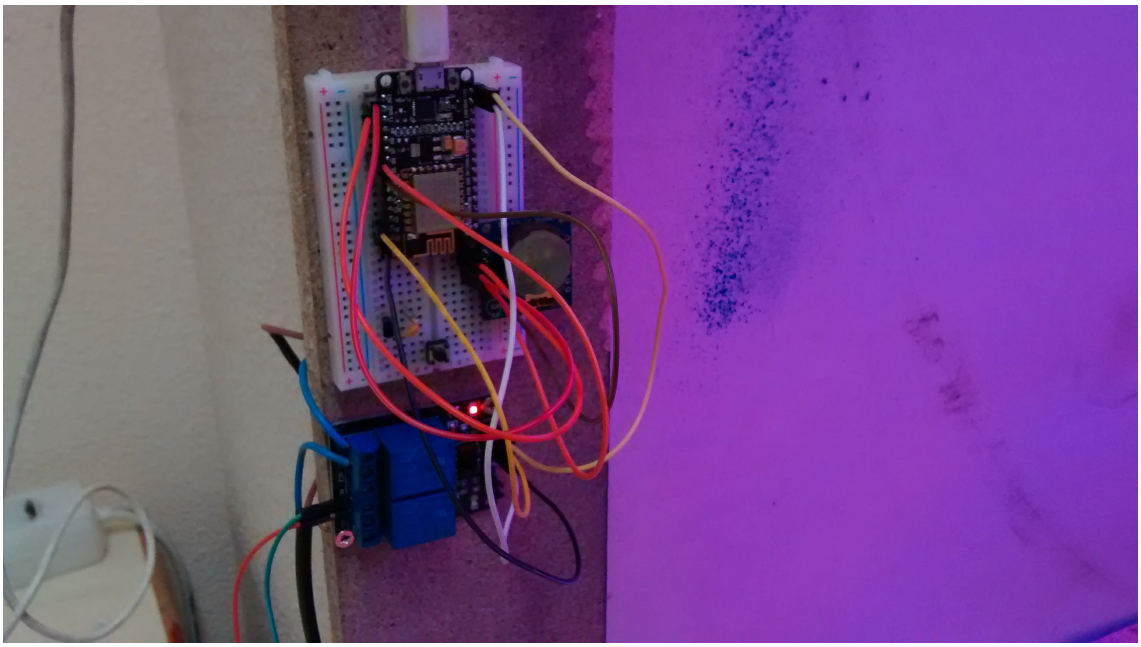

Figura 8.5 Un dispositivo conectado a un Relé que controla las luces.

Apéndice A

Sobre el código

Por motivos de mantener la memoria poco sobrecargada se ha decidido no incluir el código desarrollado como anexo. Ya que el código impreso se queda obsoleto pronto y no resulta cómodo de leer, se deja a continuación el enlace a todo el código desarrollado en GitHub:

https://github.com/kintonxyz

Índice de Figuras

2.1. Estructura de un aplicación que use la plataforma 5
2.2. Estructura interna de la plataforma 6

3.1. Patrón publicador/suscriptor 9
3.2. Ejemplo de enrutamiento en AMQP 12
3.3. *Exchange* directo 13
3.4. *Fanout exchange* 13

4.1. RabbitMQ 19

5.1. Explorador de la API 24

6.1. MongoDB 29
6.2. Realizando una consulta a RethinkDB 30

8.1. Sistema de riego inteligente y control de luz 33
8.3. Recibiendo mensajes de IFTTT en Pushbullet 34
8.2. Receta de IFTTT para recibir alertas en Pushbullet 35
8.4. Grafana 36
8.5. Un dispositivo conectado a un Relé que controla las luces 38

Índice de Códigos

4.1. Configuración de RabbitMQ 18
4.2. Fragmento de código del backend de autorización 19

5.1. Petición para creación de usuario 21
5.2. Petición de login 22
5.3. Respuesta a login 22
5.4. Creación de `fleet_key` 22
5.5. Obtencion de `fleet_key` 22
5.6. Registro de dispositivo 23
5.7. Respuesta de registro de dispositivo 23
5.8. Declaración de una flota en Loopback.io 25
5.9. Declaración de usuario en Loopback.io 26
5.10. Declaración de dispositivo en Loopback.io 27

8.1. Fragmento de código del backend de autorización 36

Bibliografía

[1] StrongLoop an IBM company, *Loopback.io*, *https://loopback.io/*, 2016.

[2] dc-square GmbH, *Everything you need to know about mqtt*, *http://www.hivemq.com/mqtt-essentials/*, 2016.

[3] ifttt, *Ifttt*, *https://ifttt.com/wtf*, 2016.

[4] Present Pivotal Software Inc, *Amqp 0-9-1 model explained*, *http://www.rabbitmq.com/tutorials/amqp-concepts.html*, 2007.

[5] Fundación Node.js, *Express*, *http://expressjs.com/es/*, 2016.

[6] RethinkDB, *Rethinkdb*, *https://www.rethinkdb.com/faq/*, 2016.