UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Diseño e implementación de *core* para el *Internet of Things*

Autor: Javier Romera Llave
Tutor: Alejandro Bellogin Kouki

junio 2022

Diseño e implementación de *core* para el *Internet of Things*

AUTOR: Javier Romera Llave
TUTOR: Alejandro Bellogin Kouki

Escuela Politécnica Superior
Universidad Autónoma de Madrid
junio de 2022

Resumen

El Internet de las Cosas lleva conviviendo con nosotros durante mucho tiempo, y últimamente su uso está muy extendido a distintas aplicaciones, desde la industria (como podrían ser plantas de producción robotizadas) hasta la domótica de los hogares para controlar distintos elementos actuadores (como podrían ser sistemas de climatización o iluminación entre otros).

Este Trabajo de Fin de Grado consiste en la elaboración de un pequeño núcleo para el Internet de las Cosas, centrándose en el ámbito de la domótica. Empezaremos analizando algunas de las tecnologías y protocolos que existen actualmente para facilitar la intercomunicación entre las distintas entidades, al mismo tiempo que discutiremos distintos escenarios para la identificación de los diferentes dispositivos. Plantearemos un esquema donde se podrán apreciar las distintas entidades y los múltiples canales de comunicación a través de los cuales interactúan las distintas tecnologías y dispositivos. Se detallará el planteamiento y la implementación de una serie de funcionalidades que permitirán agilizar el proceso de identificación, vinculación y actualización de dispositivos teniendo en cuenta distintos aspectos relacionados con la seguridad informática, así como la comunicación entre los distintos proveedores de servicio y dispositivos.

Finalmente, realizaremos algunas pruebas relacionadas con el rendimiento de algunos de los sistemas que conforman este núcleo, al mismo tiempo que pondremos a prueba distintos casos de uso que podrían suponer un riesgo desde el punto de vista de la seguridad informática.

Palabras clave

Internet de las Cosas (IdC), núcleo IdC, domótica, broker, MQTT, seguridad, Máquina a máquina (M2M)

Abstract

The Internet of Things has been coexisting with us for a long time, lately, its use has been extended to multiple applications, from the industry (with robotized production plants) up to home automation to control different actuators (such as climatization or lightning systems).

This Bachelor Thesis consists in the development of a small core for the Internet of Things, focusing on the field of home automation. We will start analyzing different technologies and protocols that currently exist to facilitate the intercommunication between the different entities while we discuss the different scenarios to identify multiple devices uniquely. We will then present a scheme where the different entities and the multiple communication channels through which different technologies and devices interact could be observed. We will also detail the approach and implementation of a set of functionalities that will speed up the identification, linking, and updating of devices, keeping in mind different aspects related with computer security as well as the communication between devices and service providers.

Finally, we will perform a set of tests related with the performance of some of the systems that compose this core as well as testing different use cases that could entail a risk from the point of view of computer security.

Keywords

Internet of Things (IoT), IoT core, home automation, broker, MQTT, security, Machine to machine (M2M)

Índice

Índice de figuras

Índice de tablas

Índice de código

1

Introducción

1.1 Origen de la idea

Cuando me aventuré en el mundo de los sistemas embebidos utilizando placas micro controladas de desarrollo, empecé por cosas sencillas como encender y apagar un diodo *led*, controlar algunos servomotores, interpretar variables físicas a través de diferentes sensores, al mismo tiempo que quemé mis primeras resistencias y exploté algunos condensadores.

En mi caso me gustaba mucho todo aquello relacionado con la domótica, el poder tener control sobre determinados elementos y poder automatizar determinados procesos rutinarios del día a día, además de tener la posibilidad de controlarlos desde prácticamente cualquier sitio.

Si buscamos información sobre cómo dar vida a este tipo de sistemas, observaremos distintas alternativas, una de ellas es utilizar servicios prefabricados que, usando determinadas herramientas y/o librerías, permiten conectarte con servidores de uso gratuito con numerosas limitaciones. Estas soluciones son demasiado intrusivas, consumen demasiados recursos y tienen un nivel de personalización muy reducido por lo que no resultan muy convenientes. Otra opción consiste en utilizar una mezcla de distintas librerías (no del mismo autor) para conseguir ese nivel de personalización, pero al final se terminan creando demasiadas interfaces intermedias con ligeras penalizaciones de rendimiento y de diseño, pero sin duda alguna es una mejor opción que la anterior. Evidentemente siempre existe la opción de implementar algo desde casi cero, tomando este enfoque podríamos intentar hacer un sistema que cumpliera con todos nuestros requisitos, pero tampoco queremos hacerlo de tal forma que sólo sirva para un caso específico.

A raíz de esto surgió la idea de elaborar un sistema informático que ofreciese la posibilidad de **instanciar un sistema del *Internet of Things***, es decir, la idea es poder ofrecer un pequeño **núcleo** (también llamado *core*) que permita al programador **despreocuparse de ciertos aspectos que resultan iguales en casi cualquier esquema del *Internet of Things***, como puede ser el simple hecho de **identificar los dispositivos**, **generar y gestionar el intercambio de credenciales** o cómo realizar el **proceso de asociación de una persona o entidad con una *cosa*** (cerraduras inteligentes, dispositivos de aire acondicionado, dispositivos de iluminación, etc.), entre otras funcionalidades.

1.2 Qué es y para qué sirve el Internet of Things

Muy resumidamente, el *Internet of Things* (en español: Internet de las Cosas y comúnmente abreviado del inglés como *IoT*) hace referencia a una red colectiva de dispositivos y tecnologías que facilitan la comunicación de diferentes dispositivos con la nube. Esta comunicación por lo general es en tiempo real y facilita la cooperación entre distintos sistemas especializados en realizar tareas de distinta índole.

Las posibles aplicaciones del *IoT* son muy variadas: podemos hablar de estaciones meteorológicas que recogen diferentes datos para realizar los correspondientes análisis de estos y poder así realizar predicciones, hasta conseguir el control remoto de presas, puentes basculantes, centrales eléctricas, etc. Esto quiere decir que no todas las implementaciones del *IoT* tienen por qué ser idénticas, es muy difícil hacer un *core* genérico que funcione para casi cualquier aplicación del *IoT*, es por esto por lo que nosotros nos vamos a centrar en una implementación que resuelva los **problemas más comunes y aislados en el contexto de la domótica**.

1.3 A quién va dirigido

Es importante destacar que este proyecto está **orientado a desarrolladores *hobbyist* que quieran implementar su propio servicio del *IoT* personalizado**, este proyecto no está diseñado para ofrecer soluciones a otras empresas ni tampoco a servicios de índole profesional, es un proyecto diseñado para usuarios que quieren montar su propia instancia del *IoT* y tener un **control total** (así como con capacidad de personalización).

1.4 Principales objetivos

El principal dilema que aparece al hacer un *core* es preguntarse hasta dónde podemos generalizar y abstraer la funcionalidad de éste. Cuantas más funcionalidades traiga de por sí, más restricciones tendrá la aplicación final (será menos flexible), pero nos permitirá reducir el tiempo de desarrollo de la aplicación o servicio; en cambio, si optamos por una solución menos intrusiva (con menos funcionalidades definidas) el núcleo base será mucho más flexible con la pega de que incrementará el tiempo que el programador tiene que invertir para conseguir sus objetivos funcionales. Es por esto, que procuraremos encontrar un término medio que maximice el lado positivo de ambas perspectivas.

Elaboraremos una estructura base sobre la que se podrá crear una instancia del *IoT* al igual que analizaremos distintos aspectos de seguridad que se presentan en un esquema tan amplio, y plantear los distintos tipos de **vulnerabilidades que pudiera haber y cómo mitigarlos**. Profundizaremos en todos los conceptos que rodean a todo este esquema, desde cómo se consigue **identificar a los dispositivos**, cómo se realiza el **intercambio de credenciales**, el proceso de **asociación de una persona con una *cosa***, el proceso por el cual los dispositivos podrán **actualizarse vía OTA**, hasta la **gestión de privilegios**.

Plantearemos distintos protocolos que definan el comportamiento que deben seguir los dispositivos con el servicio remoto, adicionalmente elaboraremos estructuras y esquemas prefabricados de bases de datos. Ofreceremos una *API* como recurso de programación para facilitar el proceso de implementación de los distintos servicios que el programador desee incorporar. Analizaremos también los distintos agentes que interactúan en este esquema, desde servidores *HTTP*, gestores de bases de datos, *brokers* y **servidores de contro**l (o **servicios específicos para determinados dispositivos**).

Aunque no esté destinado para ser utilizado en un sistema sobredimensionado, se diseñará de tal forma que permita en la mayor medida posible paralelizarse y ofrecer una **gran flexibilidad a la hora de escalarse tanto vertical como horizontalmente**.

2

Estado del arte

2.1 Tecnologías actuales

Actualmente existen multitud de tecnologías que nos pueden servir como punto de partida para plantear este proyecto, si investigamos un poco, podemos encontrar distintos protocolos de comunicación como, por ejemplo: *DDS*, *AMQP*, *CoAP* o *MQTT*. Tras debatir qué protocolo se ajusta más a nuestros objetivos, veremos qué implementación de éste nos resulta más flexible y finalmente veremos qué lenguajes utilizamos para el resto de la implementación del *core*, poniendo en perspectiva los principales frentes.

2.1.1 Protocolos de comunicación

Para que cualquier implementación del *IoT* tenga sentido, es necesario que la **comunicación entre los dispositivos esté muy bien definida**, es por esto, que actualmente existen multitud de protocolos de comunicación que nos pueden facilitar la forma en la que se intercambia información entre los distintos dispositivos.

***AMQP* (*Advanced Message Queuing Protocol*):** este protocolo soporta comunicaciones punto a punto y se basa en el concepto de publicación/suscripción. Tiene su origen en el sector de servicios financieros y en el sector de la industria, aunque en este último está bastante limitado. Es especialmente útil de cara a la realización de transacciones y poder garantizarlas de manera completa. Ofrece seguridad a través de autenticación y cifrado mediante SASL o TLS.

DDS (Data Distribution Service): este protocolo también se basa en el concepto de publicación/suscripción y fue diseñado para sistemas de tiempo real. Es un protocolo de estándar abierto y descentralizado por lo que los nodos de DDS tienen la capacidad de comunicarse directamente punto a punto a través de UDP (*multicast*). Este protocolo puede resultar útil para la interacción entre dispositivos que requieren estar intercomunicados para llevar a cabo las tareas de la manera más productiva posible.

CoAP (Constrained Application Protocol): el principal objetivo de este protocolo es proveer la compatibilidad con HTTP con una mínima carga, por lo tanto, estamos ante un protocolo cliente/servidor, el cual es muy similar a HTTP, pero usa UDP en lugar de TCP. Ofrece seguridad a través de DTLS[1].

[1] Datagram Transport Layer Security

MQTT (Message Queuing Telemetry Transport): es un protocolo que ha ido adquiriendo bastante popularidad en los últimos años. Se basa en el concepto de publicación/suscripción y ha sido implementado en multitud de aplicaciones del *IoT*. La comunicación entre los dispositivos consiste en el uso de una estructura jerárquica de *topics* (similar a un sistema de ficheros). Este protocolo está centralizado, por lo que es necesaria la existencia de un nodo central (también denominado *broker*) que se encargue de gestionar todo el tráfico de mensajes. Además, ofrece la posibilidad de identificar a los dispositivos a través de credenciales y, al ser un protocolo que funciona sobre TCP/IP, se puede utilizar TLS para añadir seguridad en la comunicación.

Protocolo	Transporte	Modelo	Conocimiento del contenido	Seguridad	Gestión de prioridad	Centralizado
AMQP	TCP/IP	End to end	Ninguno	TLS	No	Si/No
CoAP	UDP/IP	Req/Res (REST)	Ninguno	DTLS	No	No
DDS	UDP/IP TCP/IP	Pub/Sub Req/Res	Enrutamiento basado en el contenido	TLS, DTLS, DDS	Si	No
MQTT	TCP/IP	Pub/Sub	Ninguno	TLS	No	Sí

Tabla 2.1: Protocolos de comunicación [1].

2.1.2 Discusión sobre los distintos protocolos

El protocolo AMQP tiene multitud de características que permiten garantizar la transaccionalidad de determinadas acciones, el problema que tiene (entre otros) es que no es un protocolo enfocado a conexiones en tiempo real, esto choca bastante con nuestros objetivos, ya que queremos conseguir **comunicaciones reactivas** que nos permitan **transmitir eventos e información en tiempo real** para que tengamos la sensación de que los dispositivos estén "vivos". Por otro lado, el protocolo CoAP trata de reducir el tamaño de la cabecera del protocolo HTTP lo cual puede resultar interesante de cara a la reducción del tamaño de los paquetes que intercambian los dispositivos y además se basa en un protocolo altamente extendido y que está implementado en prácticamente cualquier plataforma, pero de nuevo, el principal problema es que sigue sin ser un protocolo de comunicación en tiempo real y funciona bajo el concepto de solicitud/respuesta. En contraste, el protocolo DDS sí ofrece una comunicación en tiempo real, puede funcionar bajo el concepto de publicación/suscripción o como petición/respuesta, presenta **múltiples formatos de configuración** y en la cabecera de los paquetes se añade información adicional sobre la máquina que envía el mensaje, entre otros datos. Este protocolo resulta especialmente útil si por ejemplo queremos **hacer comunicaciones en redes de acceso local**, por ejemplo, las camillas de monitorización de los hospitales están conectadas con centralitas donde el personal sanitario puede ver el estado de las constantes vitales de los pacientes, al ser un protocolo descentralizado sería una buena elección para esta aplicación, no obstante, resulta ser incompatible con nuestro planteamiento, ya que nuestros dispositivos se encuentran en **redes desconocidas donde la configuración que esto requeriría no estaría a nuestro alcance**. Adicionalmente, MQTT es un protocolo que al igual que DDS permite conexiones en tiempo real, también se basa en el concepto de publicación/suscripción y además presenta el concepto de calidad del servicio (*QoS*) que permite especificar el comportamiento de los mensajes que se transmiten, pero a diferencia del protocolo DDS **proporciona un esquema centralizado** donde existe un nodo principal (comúnmente denominado

como ***broker***) el cual se encarga **de gestionar todo el tráfico de mensajes**. Esto permite despreocuparnos de la red en la que se encuentre nuestro dispositivo y por ende de la configuración (estamos hablando de redes domésticas comunes). Al tener un esquema centralizado, los dispositivos sabrán en todo momento encontrar el servidor de mensajes que hace de puente entre todos los clientes que quieran comunicarse a través de este protocolo. También cabe recordar que nuestro objetivo no consiste en implementar dicho protocolo, sino en utilizar una implementación que nos permita instanciar un *broker* en nuestro sistema. Al ser un protocolo bastante conocido (al menos en el mundo del *IoT*), tenemos la suerte de que existen multitud de implementaciones para distintos entornos de programación. Por tanto, MQTT parece un buen punto de partida para comenzar a diseñar nuestro *core*. Esto implica cierta dependencia con el protocolo de comunicación, por lo que haremos a continuación una serie de valoraciones que nos permitan decidir cuál de las múltiples implementaciones nos resulta más conveniente.

2.1.3 Implementaciones del protocolo MQTT

Existen multitud de implementaciones del protocolo MQTT, pero buscamos una que sea lo suficientemente flexible como para poder configurar diferentes comportamientos que se ajusten a nuestros requerimientos.

Mosquitto: es una implementación de código abierto escrita en C que ofrece soporte para las versiones *5.0, 3.1.1* y *3.1* del protocolo MQTT, esta implementación es muy conocida y sólida, además existe una gran colaboración por parte de la comunidad desarrolladora. Presenta una versión muy liviana que necesita muy pocos recursos para funcionar, adicionalmente, permite la utilización de *plugins* de terceros que suponen un extra en la funcionalidad. Por otro lado, implementa también de manera nativa un sistema de gestión de privilegios muy simple basado en ficheros de configuración.

HiveMQ: esta implementación del protocolo está escrita en Java e implementa (al igual que *Mosquitto*) las versiones *5.0, 3.1.1* y *3.1*, presenta muchas funcionalidades de manera nativa y además también ofrece la posibilidad de extender la funcionalidad gracias al uso de *plugins* de terceros. Permite aplicar distintas técnicas de *clustering* para paralelizar la gestión de los distintos mensajes y por tanto que el *broker* pueda escalarse fácilmente.

Aedes: el lenguaje utilizado en esta implementación es JavaScript (para entornos Node JS). Aedes es la sucesora de la antigua y descontinuada implementación denominada Mosca, y aparee con la finalidad de solucionar diversas limitaciones que presentaba la versión antigua al mismo tiempo que adquiere nuevas cualidades que hacen que esta implementación sea realmente atractiva, como, por ejemplo, mejoras en el rendimiento o la capacidad de paralelizar la gestión de los distintos mensajes al igual que en la implementación de HiveMQ. Soporta las versiones *3.1* y *3.1.1* de MQTT, la versión *5.0* está pendiente de ser implementada.

2.1.4 Comparativa de implementaciones del protocolo MQTT

La implementación de Mosquitto es especialmente eficiente y liviana. Presenta unos requerimientos *hardware* muy poco restrictivos y esto lo hace ideal para ser configurado y ejecutado en máquinas con una capacidad de cómputo relativamente limitada, como podría ser una *Raspberry PI.* Si queremos añadir funcionalidad adicional, tendremos que usar *plugins* de terceros que cumplan con nuestros

requisitos, el problema es que algunos de estos *plugins* están obsoletos y llevan bastante tiempo sin actualizarse. Obviamente, sigue existiendo la posibilidad de crear nuestros propios *plugins*, los cuales evidentemente tendrán que ser implementados en el mismo lenguaje que utiliza la implementación del protocolo, que en este caso es C. Por otro lado, la implementación de HiveMQ presenta muchos parámetros de cara a la configuración del *broker* al mismo tiempo que ofrece múltiples funcionalidades, aun así, esta implementación sigue ofreciendo la posibilidad de extender esta funcionalidad a través de *plugins* de terceros. Adicionalmente, Aedes presenta cualidades muy parecidas a HiveMQ desde el punto de vista de la flexibilidad, ya que ambos se basan en el concepto de *callbacks* para permitir definir la lógica que debe seguirse antes de tomar determinadas decisiones, como podría ser el simple hecho de permitir a un dispositivo suscribirse a un *topic* determinado.

Aunque estemos buscando una implementación del protocolo MQTT no quiere decir que queramos funcionalidades más allá de las definidas por el propio protocolo, preferimos la flexibilidad ante la funcionalidad adicional que se haya añadido sobre la implementación en cuestión, el problema está en que las implementaciones de las que partimos son todas muy similares en este aspecto (ignorando las funcionalidades extra), así que tendremos que cribar a partir del lenguaje que utilizaremos para añadir la lógica que nuestro *core* requiera. Para simplificar el proceso de implementación, optaremos por Aedes como implementación del protocolo MQTT, JavaScript nos permitirá reducir considerablemente el tiempo de desarrollo. El lector podrá pensarse que la elección de Aedes debido al lenguaje que se utiliza podría penalizar en el rendimiento final del *broker*, pero si buscamos por la red los diferentes *benchmarks* realizados en máquinas de prestaciones domésticas con esta implementación, veremos que son cifras perfectamente compatibles con nuestros objetivos, el simple hecho de ofrecer la posibilidad de funcionar con un *cluster* mejora el rendimiento considerablemente.

2.1.5 Lenguajes para la implementación del núcleo

Tenemos dos destacables frentes en este esquema, los dispositivos (también conocidos como clientes en futuras menciones) y el *backend*. La implementación de este *core* requiere tener en mente multitud de entornos. Aunque nuestro principal objetivo sea el *backend* (como un sistema informático), mencionaremos también lenguajes desde el punto de vista de sistemas más específicos que nos ayuden a hacernos una idea de la multitud de entornos que existen.

Al igual que ocurría con los protocolos de comunicación, existen multitud de lenguajes que nos pueden permitir implementar este núcleo, la decisión de usar uno u otro se verá afectada por la comunidad que haya detrás de dicho lenguaje o por el número de paquetes y librerías que lo acompañen, no obstante, es importante analizar las características y diferentes propiedades del lenguaje, sobre todo si estos son interpretados, ya que esto implica, por ejemplo, que la forma de gestionar hilos, procesos o subtareas no tiene por qué ser la misma. Cuanto mayor sea el nivel del lenguaje menor será la dependencia que habrá sobre el *hardware* en el que se ejecutan, pero menor será por tanto el nivel de optimización.

Al estar ante un proyecto con cierta amplitud, donde la convivencia de diferentes tecnologías implica la utilización de distintos lenguajes de programación, no podemos utilizar un único lenguaje como tal, sino que tendremos que adaptarnos a las restricciones que aplique la tecnología en cuestión e incluso los casos en los que no existe ni si quiera la posibilidad de elegir entre un lenguaje u otro.

Lenguajes de menor nivel como C ofrecen un gran nivel de optimización ya que podemos definir con mayor exactitud el comportamiento de las diferentes rutinas. C presenta una gran variedad de recursos y librerías de terceros que pueden facilitar el proceso de implementación, el problema reside en que el tiempo que requeriría implementar este *core* (al menos en la parte del *backend*) usando este lenguaje sería considerablemente elevado y probablemente incompatible con las franjas temporales de las que disponemos.

Es importante hacer notar, que el concepto de cliente es relativamente amplio, ya que un cliente puede consistir en un PC de sobremesa o en un simple módulo *SoC* como el ESP-32 de *Espressif*, esto quiere decir que en algunos casos el lenguaje ideal no tiene por qué ser de bajo nivel, podríamos optar por lenguajes de mayor nivel como Java o Python entre otros, siempre y cuando respetemos la especificación de los distintos protocolos (que veremos en detalle en el capítulo de implementación) que rigen el flujo de interacción entre las distintas entidades de este sistema.

La elección del lenguaje desde el punto de vista del sistema específico dependerá enteramente del entorno o la plataforma que se escoja para desarrollarlo, existiendo así una brecha entre la existencia o no de una API para dicha plataforma, de igual modo, necesitamos encontrar el entorno de programación que más se ajuste a nuestras necesidades para la implementación del *core* en el lado del *backend* que es la que vamos a diseñar y analizar en este documento.

Para decidir el lenguaje de programación que utilizaremos en el *backend,* nos centraremos en múltiples condiciones: disponer de un amplio marco de librerías y componentes, existencia de una gran comunidad de desarrolladores detrás del lenguaje en cuestión, que presente distintas funcionalidades que nos permitan reducir el tiempo de desarrollo, gracias a: gestión de tareas asíncronas, que se pueda integrar fácilmente en diferentes *stacks* y que presente un alto grado de adaptabilidad. A raíz de estas especificaciones, el lenguaje que probablemente más se ajusta a nuestros requisitos es *JavaScript*, más concretamente en el entorno de Node JS, pero en vez de utilizar *vanilla* JavaScript utilizaremos TypeScript que permitirá añadir tipos estáticos y objetos basados en clases. Este lenguaje es especialmente útil para la gestión de diferentes tareas asíncronas, aparte de tener un alto grado de adaptabilidad por la multitud de aplicaciones que puede tener, entre las cuales destaca el *IoT*.

2.2 Soluciones actuales

Actualmente, existen diversas soluciones para gestionar la intercomunicación de distintos dispositivos junto a diversas funcionalidades que enriquecen y simplifican el proceso de implementación. Tenemos multitud de empresas que ofrecen determinados servicios que más o menos se acercan a la idea de este proyecto. Por ejemplo, *Amazon* ofrece diversos servicios relacionados con el *IoT*, uno de ellos es el *AWS*[2] *IoT Core* el cual se asemeja mucho a la idea que se plantea en este proyecto.

Amazon ofrece el punto de interconexión entre los distintos dispositivos (ver Figura 2.1), es decir, su principal objetivo es mitigar el problema de tener que preocuparse por aspectos relacionados con la capacidad de cómputo o del *hardware* como tal, ofrecen también distintos sistemas de persistencia, posibilidad de actualizar los dispositivos vía OTA, entre otras funcionalidades, pero la principal

[2] Amazon Web Services

diferencia es que en nuestro caso el objetivo (como bien hemos mencionado anteriormente) son usuarios que prefieran tener el control total sobre prácticamente todas las entidades de este esquema, teniendo así la oportunidad de conocer en mayor profundidad la funcionalidad más interna de un *core* para el *IoT* y al mismo tiempo dar vida a sus propios servicios de una forma relativamente sencilla. La solución de *Amazon* puede resultar de lo más interesante para empresas que quieran ofrecer soluciones a gran escala, sobre todo a la hora de escalar en función de la demanda que se tenga, pero en casos más aislados donde por ejemplo se pretenda conseguir un servicio más reservado y personalizado puede que el desarrollador que quiera interconectar sus dispositivos se sienta más cómodo configurando sus servicios *On-premise*.

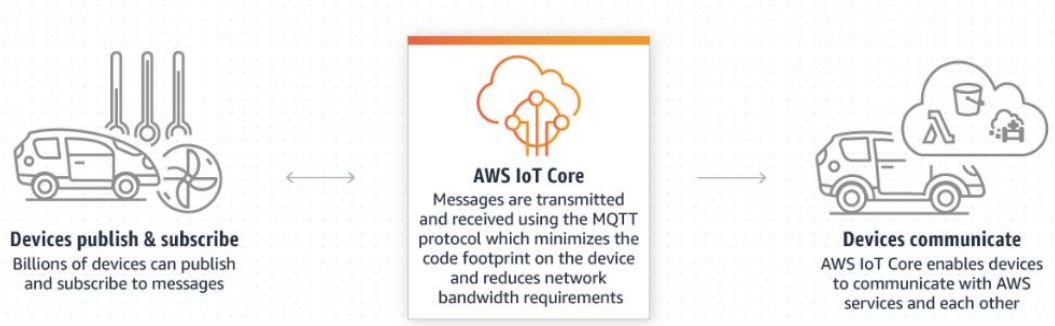

Figura 2.1: AWS IoT Core [2].

2.3 Fundamentos de identificación

Dada la diversidad de dispositivos, hay que tener especial cuidado a la hora de plantear el proceso que se llevará a cabo para su identificación. Recordemos que los dispositivos no tienen por qué tener una funcionalidad en concreto, podemos tener un dispositivo que se encargue de encender o apagar una luz (algún tipo de sistema de iluminación) o si vamos más allá, podemos hablar de un dispositivo que se encargue de controlar la apertura y el cierre de un puente basculante. También hay que recordar que en nuestro caso vamos a utilizar un determinado protocolo de comunicación entre los dispositivos que es MQTT, este protocolo forma parte de un estándar abierto y concreta un montón de procedimientos, pero a pesar de eso no especifica explícitamente cómo se debe interpretar el proceso de identificación de un dispositivo. Es importante fijarse en el término **identificador** que es similar al término **nombre**. Un nombre no debería cambiar en función de la localización de un dispositivo, en contraste con el término **dirección** que se utiliza para referirse a la **localización** de un dispositivo como podría ser una dirección IP. Pensemos por un momento en el identificador de los dispositivos ... ¿debería depender del tipo de servicio que ofrezcan? La respuesta no es sí ni tampoco no, depende mucho de cómo definamos identificación, en este caso, al estar ante un sistema de gran amplitud como lo es en el *IoT* donde se pueden conectar multitud de dispositivos con distintas finalidades, tendremos que lograr garantizar la unicidad entre éstos, aunque también veremos cómo poder adjuntar la información relacionada con el tipo de servicio sin necesidad de añadirla directamente en el identificador.

A priori, nosotros estamos buscando un sistema de identificación que nos permita, al menos, hacer la **asociación de una persona con una *cosa***, por eso quizá en este caso **no sea relevante el incorporar el tipo de servicio que ofrece un dispositivo en el propio identificador**. Esto suele ser especialmente útil en dispositivos que salen de fábrica con un número de serie, ya que este número suele generarse siguiendo un determinado formato que el fabricante define para posteriormente poder obtener información adicional en el identificador.

2.4 Dónde y cuándo generar el identificador

Como hemos dicho antes, no será necesario generar un identificador basándonos en información del tipo de servicio, ni de la localización, ni de la fecha de creación del dispositivo en cuestión, queremos generar un identificador que sea único (al menos único dentro de nuestro sistema). El sistema que estamos creando permitirá conectar dispositivos que sigan y cumplan unos determinados protocolos, es por esto, que en el momento que un dispositivo quiera conectarse a nuestro sistema tendrá que compatibilizarse con el mismo, y para eso, es necesario definir muy bien cuándo y dónde vamos a generar el identificador. A continuación, detallaremos los principales escenarios.

2.4.1 Generar el identificador a *priori*

En un caso práctico estamos hablando de generar el identificador durante el proceso de elaboración del dispositivo, es decir, antes de llegar a conectarse a Internet. Está claro que para llevar un control de los identificadores es necesario tener un registro de los identificadores que ya se han utilizado para evitar colisiones y que un dispositivo no pueda convivir con un mismo identificador que otro dispositivo. Como veremos más adelante, en el caso de que un dispositivo tuviera un identificador duplicado provocaría ciertos problemas de seguridad y se producirían conflictos y choques en la transmisión de la información que estos dispositivos intercambian entre sí, por eso, la existencia de dos identificadores duplicados no es una opción ni tampoco puede ser una posibilidad.

Una planta de producción puede dedicarse a la elaboración de dispositivos con una finalidad específica o que presentan cierta similitud, lo lógico es que cada una de estas plantas de producción se centre y especialice en un campo determinado, esta situación es análoga por ejemplo a la de un dispositivo electrónico y una red eléctrica, existen fabricantes que crean diversos productos electrónicos con miles de formas y aplicaciones distintas, lavadoras, aspiradoras, neveras, etc. Pero todos ellos tienen algo en común, todos se conectan a una toma de corriente (que por lo general suele ser de *230V* o *120V* nominales), en nuestro caso la red eléctrica sería una mezcla entre *Internet* como infraestructura y el *broker*. Esto puede verse como una pequeña desventaja a la hora de generar el identificador del dispositivo, si liberamos al proceso de elaboración de preocuparse de cómo se genera dicho identificador podremos dedicarnos única y exclusivamente a diseñar un sistema específico sin tener en cuenta los distintos aspectos que rodean a dicho identificador.

2.4.2 Generar el identificador a *posteriori*

Ahora bien, ¿qué ocurre en el caso de que generemos el identificador después del proceso de elaboración? En este caso tendremos que conseguir de alguna forma que el dispositivo se comunique de una manera un tanto especial la primera vez que se conecte con el servidor. Imaginemos por un momento que acaba de nacer un ser humano, cuando éste nace no tiene ningún nombre, ningún identificador, nada, en los próximos días al nacimiento los padres ponen un nombre a su hijo, pero ... ¿qué garantías hay de que el nombre sea único? Es altamente probable que exista actualmente otra persona con ese mismo nombre y apellidos, por eso, usar el nombre y los apellidos no sería una buena forma de identificar a una persona (al menos de manera única). En nuestro caso, el nacimiento del niño es el momento en el que se **ha finalizado el proceso de elaboración del dispositivo**, cuando en unos años llega la hora de acudir, por ejemplo, a comisaría para el proceso de identificación, correspondería con el momento en el que nuestro dispositivo se **conecta por primera vez a internet**, pues bien, es

este el momento crítico en el cual el dispositivo solicita al servidor que se **generen unas credenciales para él**.

Llegados a este punto, podemos preguntarnos ¿cómo sabe la persona que la comisaría a la que acude es verdaderamente una comisaría oficial de la policía? Pues bien, ese a priori es un problema de la policía, la policía debe asegurarse de que nadie intenta suplantar su autoridad y evitar la posibilidad de que existan comisarías pirata, pues en nuestro caso, el **problema también es nuestro**, tenemos que conseguir de alguna forma que cuando el dispositivo se conecte con el servidor, éste se esté conectando realmente con nuestro servidor y no con una copia pirata del mismo; para solucionar este problema, podemos ceñirnos al término de **certificado electrónico** dentro de una infraestructura de clave pública o *PKI*. Nuestro sistema presentará una **CA** (Autoridad Certificadora) en la que los dispositivos pueden (y deben) confiar, el servidor (o las distintas instancias que tengamos del servidor) presentarán un certificado que ha sido firmado por dicha autoridad certificadora, cuando el dispositivo se conecte con el servidor, éste enviará su certificado (a priori firmado por nuestra *CA*), ahora bien, ¿cómo consigue el dispositivo asegurarse de que el certificado que le manda el servidor es válido? El **dispositivo necesita el certificado de la autoridad certificadora para validar el certificado del servidor que fue firmado por esa misma autoridad certificadora**. Por esto resulta importantísimo, **que los dispositivos contengan el certificado original de la autoridad certificadora**. Parece que esta opción va a resultar más costosa, pero eso no quiere decir que sea una mala elección, esta opción nos va a permitir tener mayor flexibilidad a la hora de generar identificadores y además nos permitirá incluso cambiar en un futuro la forma en la que estos se generan, pero esta situación debe evitarse a toda costa, e intentar diseñar un sistema de identificación que funcione correctamente durante todo el ciclo de vida.

Antes hemos mencionado el establecimiento de una conexión de bajo nivel TCP/IP con TLS por encima, pero no hemos hablado del protocolo que está por encima de los anteriores (en la pila de red) que es MQTT. Cuando queremos conectarnos con un *broker* MQTT, debemos proporcionar un identificador, pero, antes de nada, hay que dejar claro que este identificador no tiene nada que ver con el que nos hemos estado refiriendo anteriormente, este identificador es **un identificador que utiliza el *broker* para tener un control sobre los dispositivos que hay conectados**.

Recordemos que estamos describiendo (por encima) lo que ocurre la primera vez que el dispositivo se conecta a internet, el dispositivo debe **suministrar siempre un identificador único** a la hora de conectarse con el *broker*, da igual que sea la primera conexión, pero entonces ... ¿qué identificador usa el dispositivo? Todo este proceso está claramente explicado y detallado en el **protocolo de generación e intercambio de credenciales** (ver sección 4.3.1), pero antes de eso veremos por qué es necesario que exista una contraseña (a pesar de la supuesta unicidad y aleatoriedad del identificador de usuario). Si el sistema se basara únicamente en un identificador único universal, no ofrecería ningún tipo de privacidad, ni tampoco existiría la posibilidad de asociar el dispositivo con una persona, **todos los dispositivos serían de alguna manera de todo el mundo**. Con *"de alguna manera"* nos referimos a que un usuario malintencionado podría averiguar con relativa facilidad el identificador del dispositivo en cuestión. En caso de que el dispositivo no posea unas credenciales de acceso (como en el caso de la primera conexión) deberá usar unas establecidas por defecto. Dichas credenciales al igual que otros aspectos como el flujo de generación de credenciales utilizado se especifican en la correspondiente implementación y especificación del protocolo.

3

Diseño

Antes de proceder a detallar todos los aspectos funcionales y estructurales, es importante tener una visión general de todas las entidades que interactuarán en nuestro *core* (como se puede ver en la Figura 3.1).

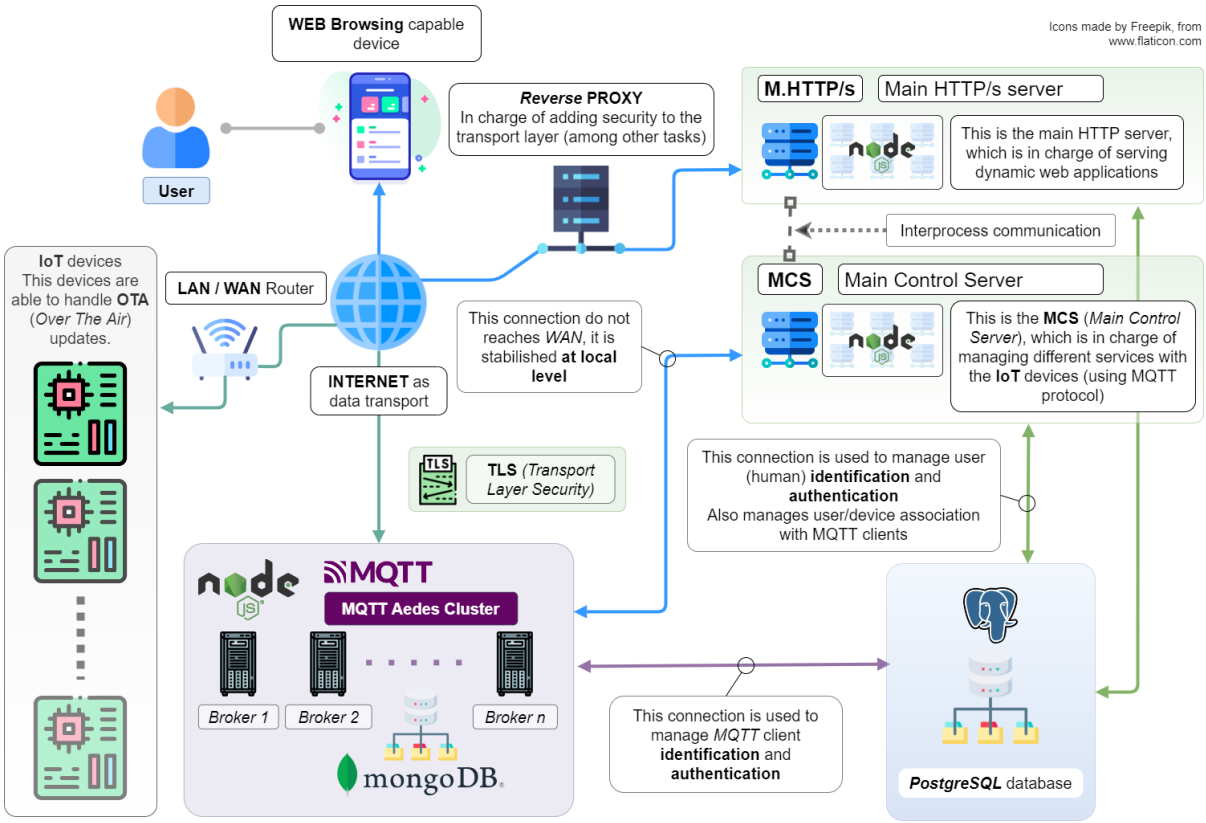

Figura 3.1: Diagrama general de interacción.

Es importante dejar claro que **algunas de las entidades representadas en este diagrama no serán analizadas en este documento**, ya que se **escaparían de nuestro ámbito**, por ejemplo, el dispositivo móvil que se puede ver en la zona superior del diagrama (Figura 3.1) representa una idea de interfaz de usuario que por cuestiones de tiempo y espacio no se planteará ni analizará. Por otro lado, el servidor HTTP sí será analizado, pero como un simple servicio de recursos, que interactúa de alguna forma con los servidores de control que son los encargados de proveer servicios específicos (ya detallaremos cómo). La principal razón por la cual estamos mostrando entidades adicionales es porque se pretende mostrar el potencial que puede llegar a tener este planteamiento gracias al **bajo acoplamiento de los distintos componentes y/o entidades**. Es por esto por lo que el diagrama en cuestión presenta entidades que no tienen por qué ser exactamente las mismas, sino que existe la posibilidad de que sean reemplazadas por otras. Por ejemplo, la base de datos se especifica como una instancia de PostgreSQL, pero podría ser otro gestor (no cualquiera, pero existen ciertas alternativas que veremos en la sección 4.5).

Otro elemento que aparece en el diagrama de la Figura 3.1 y que no es estrictamente obligatorio, pero sí muy recomendable, es el servidor *Proxy* inverso, éste nos permite crear una frontera entre la red local que tenemos configurada y la red externa. Hay que tener especial cuidado a la hora de configurar los servidores Proxy ya que pueden ser los principales cuellos de botella de un sistema informático. En este caso la principal funcionalidad de este *Proxy* consiste en añadir seguridad a la capa de transporte una vez los datos salen de la red local o balancear la carga entre los distintos servidores HTTP (entre otras tareas). Nótese que el *broker* MQTT en el diagrama aparece conectado directamente a Internet al igual que el servidor *Proxy*, pero esto no quiere decir que en un caso práctico tenga que ser así (dando por sentado que el enrutador se encontraría implícito en el *broker* y en el *Proxy*). Dado que nuestros principales objetivos giran alrededor de un entorno doméstico, el enrutador se correspondería con el típico *router* que nos haya proporcionado nuestra operadora (o adquiriendo uno por nuestra cuenta configurándolo con las respectivas credenciales que nos haya proporcionado ésta).

3.1 Análisis de requisitos

A continuación, detallaremos los distintos requisitos que nuestro sistema debe cumplir para satisfacer distintas necesidades funcionales.

3.1.1 Requisitos funcionales

1.-- RF1 **Identificación de dispositivos:** hará posible que los clientes MQTT realicen el proceso de identificación (soliciten credenciales) de manera automática la primera vez que se conectan a Internet (o cada vez que son restablecidos).

2.-- RF2 **Gestión de privilegios:** dado que Aedes (implementación del *broker* MQTT) no presenta ningún tipo de sistema de gestión de privilegios de forma nativa, necesitaremos incorporar uno que nos permita restringir el acceso a determinados *topics*. También resulta necesario que estos permisos sean altamente configurables para los distintos usos que haremos sobre el protocolo MQTT, incluyendo rolesque permitan agrupar distintos privilegios.

3.-- RF3 **Comunicación de dos vías**: debido a que MQTT presenta un sistema de comunicación *One-to-Many*, tendremos que elaborar una determinada nomenclatura y jerarquía de *topics* que nos permitan al mismo tiempo realizar comunicaciones *One-to-One*, esto nos permitirá llevar a cabo tareas como el proceso de asociación de un usuario con un cliente (RF6), al mismo tiempo que facilitará la interacción entre el proveedor de servicios y el cliente (RF5).

4.-- RF4 **Comunicación grupal:** como necesidad del requisito RF7, será necesario permitir de forma nativa a los proveedores el poder enviar mensajes a todos los dispositivos pertenecientes a un mismo servicio.

5.-- RF5 **Envío y recepción de comandos:** esta interfaz de comunicación a través de comandos nos permitirá ejecutar acciones más específicas (parametrizables) dentro del servicio del dispositivo, esta interfaz es de carácter bidireccional, tanto el servidor como el cliente podrán enviar y recibir comandos.

6.-- RF6 **Vinculación de dispositivos:** hará posible que un usuario pueda vincular un dispositivo que haya adquirido por su cuenta, de tal forma que podrá interaccionar con él gracias a las interfaces que consideremos oportunas.

7.-- RF7 **Actualización OTA pasiva:** este requisito es necesario para que los dispositivos puedan ser notificados en el momento que se lanza la actualización (mientras el cliente está conectado).

8.-- RF8 **Actualización OTA activa:** se aplica en la situación en la que un dispositivo se encuentra desconectado del *broker*, de esta forma, al volver a conectarse tendrá la posibilidad de poder solicitar la última versión disponible para comprobar si debe actualizarse o no.

3.1.2 Requisitos no funcionales

1.-- RNF1 **Seguridad en la comunicación:** ya que estamos hablando de comunicaciones que utilizan Internet como transporte, necesitaremos garantizar ciertos aspectos relacionados con la seguridad informática, como la confidencialidad, la integridad y la autenticidad.

2.-- RNF2 **Seguridad en la generación de credenciales:** debido al requisito RF1, tendremos que realizar una serie de consideraciones en el momento de la generación, como garantizar la unicidad de los identificadores y al mismo tiempo generar contraseñas criptográficamente seguras.

3.-- RNF3 **Seguridad en el intercambio de credenciales:** no se puede permitir que durante el proceso de credenciales estas puedan ser comprometidas por clientes malintencionados.

4.-- RNF4 **Control de versiones:** a raíz de la posibilidad de realizar actualizaciones OTA (RF7 y RF8), se realizará un registro de versiones básico sobre el histórico de actualizaciones de un proveedor de servicio específico.

5.-- RNF5 **Comunicación con los proveedores de servicio:** uno de nuestros principales objetivos es independizar los procesos de los distintos servicios, tanto de los que se implementan de forma nativa en el *core* como aquellos que añada el programador para dar servicio a sus propios dispositivos. Al crear este tipo de aislamientos entre los distintos procesos, surge la necesidad de comunicarse con ellos desde otros procesos para notificar sobre eventos o distintas tareas que se deseen realizar. Esto implica la creación de *wrappers* que nos faciliten la comunicación con los distintos servicios.

3.2 Modelado de datos

A continuación, elaboraremos una serie de modelos que nos permitan visualizar el diseño de este sistema, veremos primero una representación en forma de entidad/relación que nos permite **modelar la estructura lógica** de la base de datos y las **relaciones necesarias entre las distintas entidades**. Posteriormente haremos una representación visual de cómo vamos a estructurar el sistema de cara a la implementación y finalmente representaremos distintos diagramas de secuencia que nos permitan visualizar el flujo que se sigue en base a determinados casos de uso.

3.2.1 Diagramas Entidad Relación

Necesitamos elaborar una estructura relacional que nos permita definir las distintas relaciones que existen entre los distintos dispositivos y los usuarios, además necesitamos crear una estructura adicional que nos permita gestionar los distintos privilegios. Por otro lado, incluiremos una pequeña relación para tener un cierto control sobre las diferentes versiones del *firmware* de los distintos dispositivos.

Como se puede ver en la Figura 3.2, existe una relación *Many-to-Many* (*mqtt_client_user*) entre los dispositivos (*mqtt_client*) y los usuarios (*user*), esto es así, ya que puede darse el caso en el cual un dispositivo pertenezca a múltiples usuarios (se da por sentado el caso en el cual un usuario tiene múltiples dispositivos). Por ejemplo, un sistema de iluminación en una vivienda por lo general es compartido entre las distintas personas que conviven en la misma, este tipo de relaciones facilitará la implementación de servicios que contemplen este tipo de situaciones.

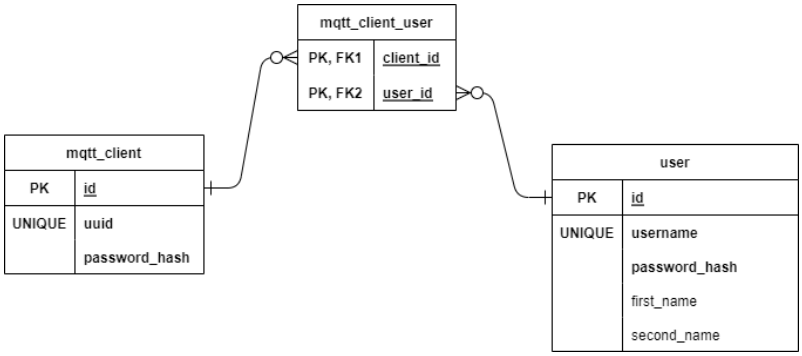

Figura 3.2: Diagrama entidad relación para la vinculación de dispositivos.

El diagrama representado en la Figura 3.3 es una extensión del diagrama mostrado en la Figura 3.2, nótese la abundancia de relaciones *Many-to-Many* debido a la multitud de situaciones en las que nos podríamos encontrar, si el programador pretende crear un servicio donde se puedan comunicar los dispositivos entre sí, será necesario flexibilizar lo máximo posible la configuración de los distintos privilegios, es por esto que se ha incorporado el concepto de roles (*mqtt_role*), el cual consiste básicamente en agrupar en un mismo nombre un conjunto de permisos (RF2), esto resultará especialmente útil de cara a la reutilización de conjuntos de permisos, como podría ser por ejemplo, el rol de un cliente público (el cual tendría accesos muy limitados) pero que resultan comunes a todos los clientes de carácter público.

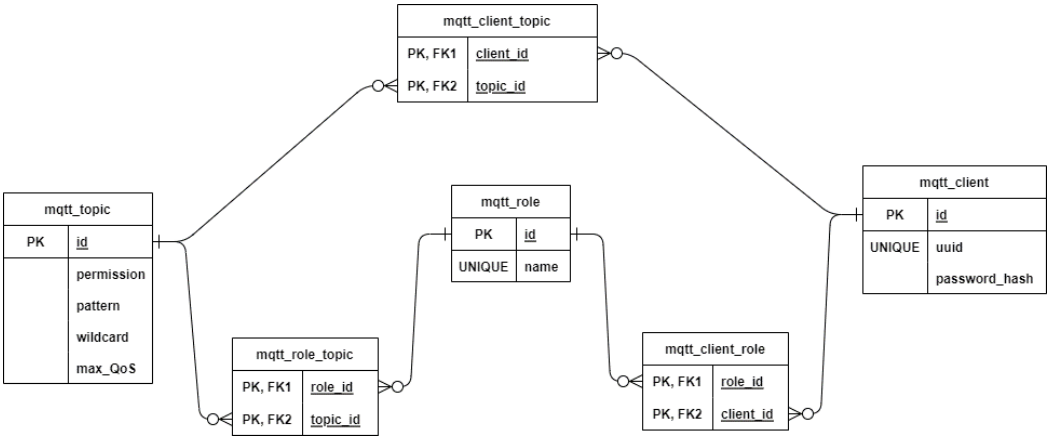

Figura 3.3: Diagrama entidad relación para el control de los distintos privilegios.

Si nos fijamos en la entidad *mqtt_client* (*MC*) de la Figura 3.3, podemos observar que además del identificador universal del dispositivo, tenemos un identificador adicional, la principal diferencia es que el identificador universal *uuid* consiste en un número muy grande (128 *bits*) y el identificador *id* es simplemente un número entero de 32 *bits*, al tener la entidad *mqtt_client_topic* (*MCT*), y la entidad *mqtt_client_role* (*MCR*) que permiten relacionar a los clientes con los diferentes roles y *topics*, estaríamos consumiendo una gran cantidad de espacio físico en la memoria de la máquina innecesariamente, además de que las planificaciones del gestor de la base de datos no serían tan óptimas. Para detallar mejor la efectividad de este planteamiento partiremos de un ejemplo sencillo, imaginemos que tenemos $N = 1000000$ clientes y cada uno de ellos tiene al menos un rol asociado y un *topic*, si partimos del planteamiento propuesto utilizando **un identificador adicional** obtenemos la siguiente estimación de espacio (ignorando el campo *password_hash* de la entidad *mqtt_client*):

$$Size \approx \sum_{1}^{N} \left(MCR_{Tuple} + MCT_{Tuple} + MC_{Tuple}\right) \approx \sum_{1}^{1000000} (8 + 8 + 20) \approx 34.33\ MB$$

En cambio, si estimamos el espacio **sin usar un identificador adicional**, obtenemos $Size \approx 53.41\,MB$. Aunque se siguen representando cifras relativamente pequeñas, no quiere decir que podamos conformarnos con la última opción, ya que está consumiendo 1.56 veces más de espacio y al mismo tiempo penalizando el rendimiento de las búsquedas.

La entidad *mqtt_topic* contiene varios campos los cuales están estrechamente relacionados con el concepto de *topic* en el protocolo MQTT, dado que estos campos pueden resultar poco intuitivos, detallamos un poco por encima cada uno: *pattern* hace referencia al patrón que debe coincidir con el *topic* a comprobar, el campo *permission* se utiliza para indicar qué tipo de permiso se está concediendo (publicación, suscripción o ambos), *wildcard* indica si se permite el uso de comodines sobre el *topic* solicitado, y finalmente el campo *max_QoS* se utiliza para limitar la calidad del servicio máxima en el *topic* en cuestión.

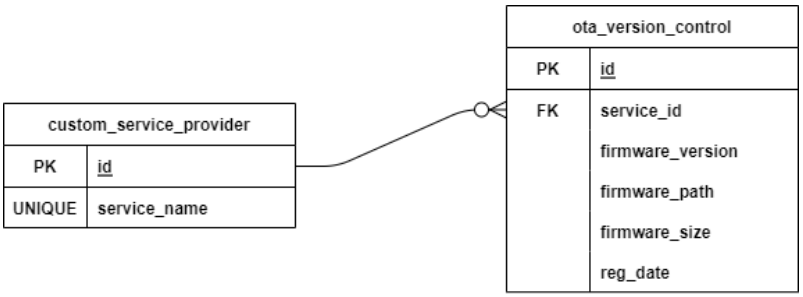

Figura 3.4: Diagrama entidad relación para el control de actualizaciones OTA.

Para tener un registro y control sobre las distintas versiones del *firmware* de los distintos dispositivos (RNF4), se han elaborado dos entidades, la primera de ellas es *custom_service_provider*, que hace referencia a los propios servicios registrados por el programador, y por otro lado, la entidad de control de versiones (*ota_version_control*) cuyo objetivo es registrar las distintas actualizaciones, al mismo tiempo que se registra información adicional relacionada con la actualización en sí, como puede ser la **versión** de la misma, la **fecha de registro** o el **tamaño total de la actualización**. Resulta evidente que el *firmware* debe almacenarse **en sistemas de ficheros independientes** y en la base de datos guardamos **únicamente una referencia al mismo**. Tener este tipo de control de versiones **facilitará el proceso de *downgrading*** en caso de que se detecten distintas anomalías en alguna de las actualizaciones. También se podría pensar en el caso en el cual un dispositivo presenta muchas similitudes con otro como para considerarse de un mismo servicio, en estos casos, a pesar de las **pequeñas diferencias**, la acción recomendada sería **elaborar un nuevo servicio que extienda la funcionalidad del original**.

3.2.2 Diagramas de clases

Los diagramas de clases van a permitir hacernos una idea sobre la estructura principal del *software* en nuestro sistema, analizando las distintas relaciones que hay entre las distintas clases junto a algunas de sus operaciones y atributos más relevantes. Como se puede observar en la Figura 3.5, la clase abstracta *ServiceProvider* permite modelizar un proveedor de servicio, los servicios que se implementan de forma nativa en nuestro *core* heredan directamente de *ServiceProvider*, en cambio todos los servicios que quiera incorporar de manera adicional el programador tendrán que heredar de *CustomServiceProvider*, el cual ofrece métodos genéricos que facilitan la publicación de distintos mensajes siguiendo una estructura de *topics* determinada, al mismo tiempo que este incorpora la lógica necesaria para la gestión automática de actualizaciones OTA, junto a un *manager* de comandos.

Dentro de los propios servicios encontramos el concepto de acción (*Action*), este nombre puede resultar algo ambiguo, pero precisamente por esto tenemos la posibilidad de extenderlo y enfocarlo a las distintas necesidades que tengamos. A continuación, mostraremos el diagrama general de las acciones y los subtipos que hay. Fijándonos en la Figura 3.6, los comandos ofrecen una estructura que facilitará al programador la implementación de cara a intercambiar información con el dispositivo, los comandos tienen como principal objetivo ejecutar rutinas más específicas dentro de los dispositivos, al mismo tiempo que nos permiten parametrizar la ejecución de estas. Al ser de carácter bidireccional, existen dos tipos: los salientes y los entrantes, *OutgoingCommand* e *IncomingCommand* respectivamente, ambos heredan de *Command* que abstrae la estructura básica de un comando.

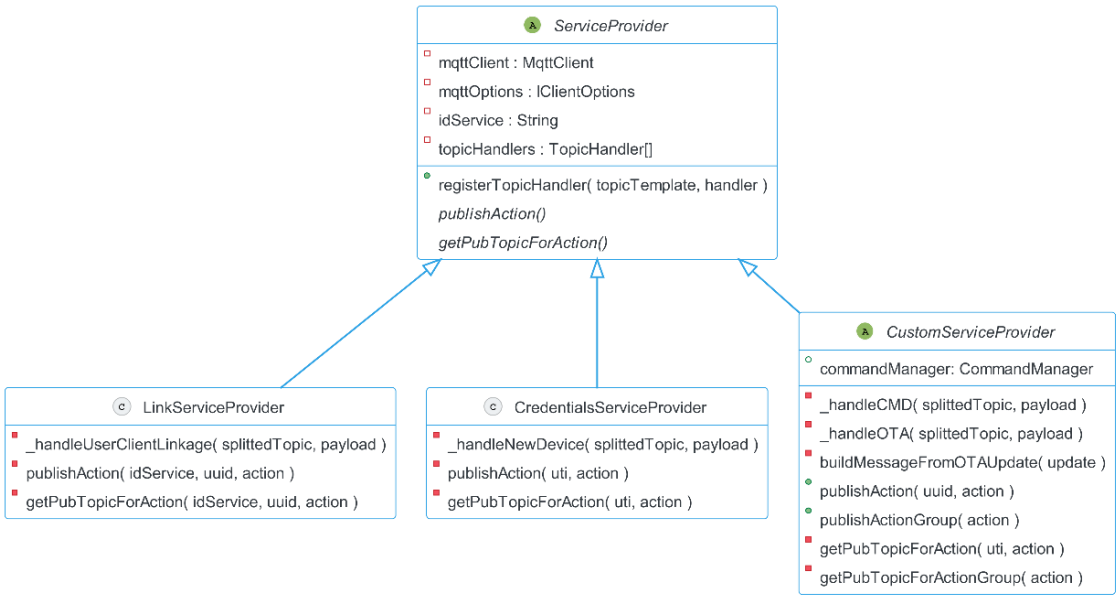

Figura 3.5: Diagrama UML de proveedores de servicio

.

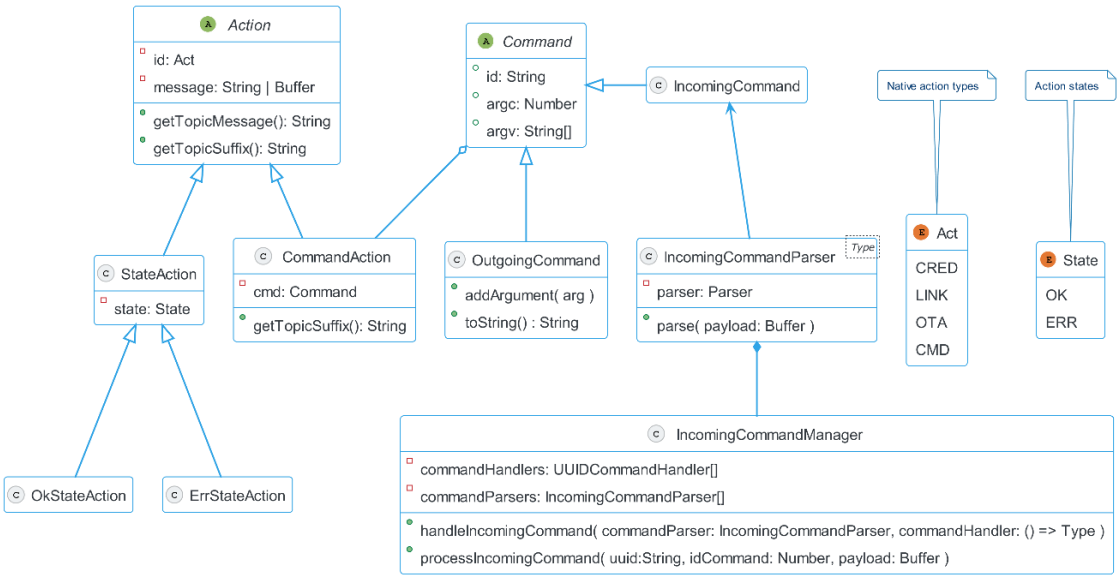

Figura 3.6: Diagrama UML de acciones y comandos.

La clase *IncomingCommandParser* nos permitirá analizar gramaticalmente las cadenas de bytes provenientes de los dispositivos en base a una gramática muy simple acordada (de la que hablaremos en la sección 4.3.3), resultará especialmente útil a la hora de convertir los comandos en estructuras de datos del propio lenguaje.

La enumeración *Act* permite identificar el tipo de acción, esto es necesario para permitir que tanto los dispositivos como los proveedores de servicios puedan interpretar correctamente las acciones que se deban llevar a cabo. Adicionalmente, el programador podrá crear sus propias acciones, pero los comandos ofrecerán una interfaz bastante completa con la que poder trabajar. Las acciones de estado se modelan a través de *StateAction*, este tipo de acciones tienen como objetivo principal transmitir mensajes con un estado determinado, por ejemplo, un intento de vinculación podría resultar exitoso o fallido, el contenido del mensaje correspondería con una descripción del error que se ha producido.

En la Figura 3.7 se muestran una serie de clases e interfaces cuya finalidad es facilitar el registro y la ejecución de los distintos servicios. Nótese el patrón *Singleton* del envoltorio para los proveedores de servicios (*ServiceProviderWrapper*) que es el principal encargado de abstraer toda la complejidad relacionada con la comunicación de los distintos servicios. La interfaz *IMainServiceProvider*, permite abstraer el concepto de proceso independiente de cada uno de los proveedores de servicio. Como se puede observar en las clases de ambos laterales, *MainCredentialsServiceProvider* y *MainLinkServiceProvider* (que son unos de los servicios implementados de forma nativa en este núcleo), contienen a la instancia correspondiente que implementa la lógica del servicio en cuestión, al fin y al cabo, estamos creando una serie de adaptadores que permiten abstraer el funcionamiento de cada uno de los servicios.

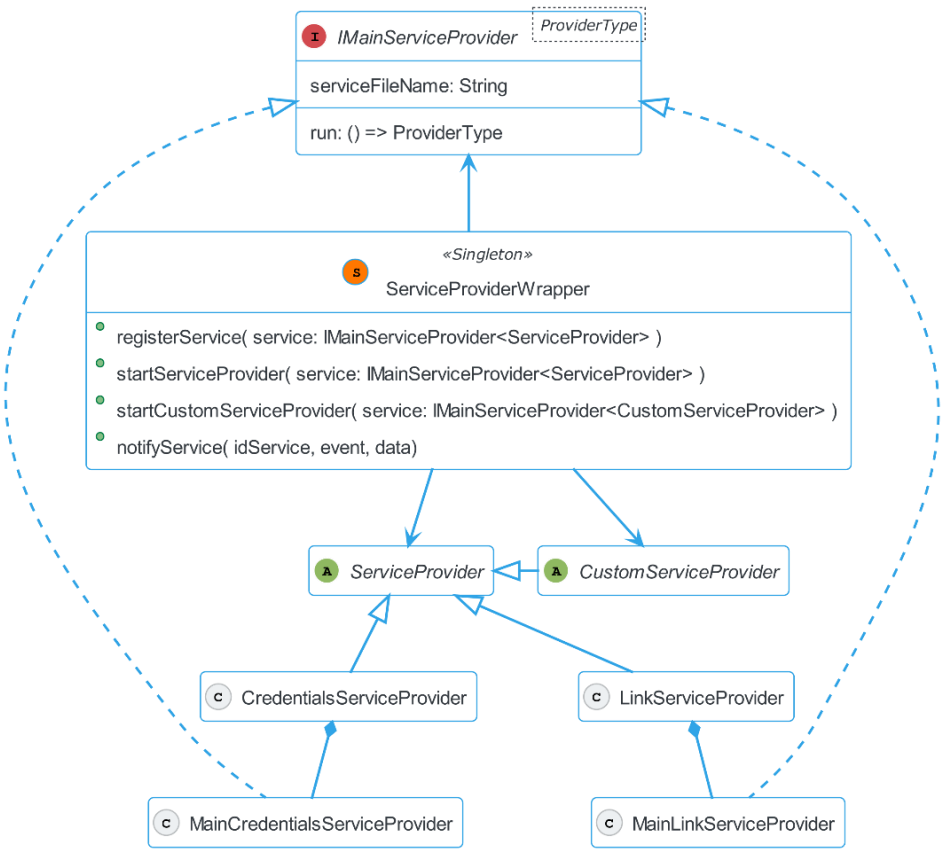

Figura 3.7: Diagrama UML de envoltorios de servicio.

En la Figura 3.8 se muestra una visión general de las dependencias y relaciones entre las distintas clases.

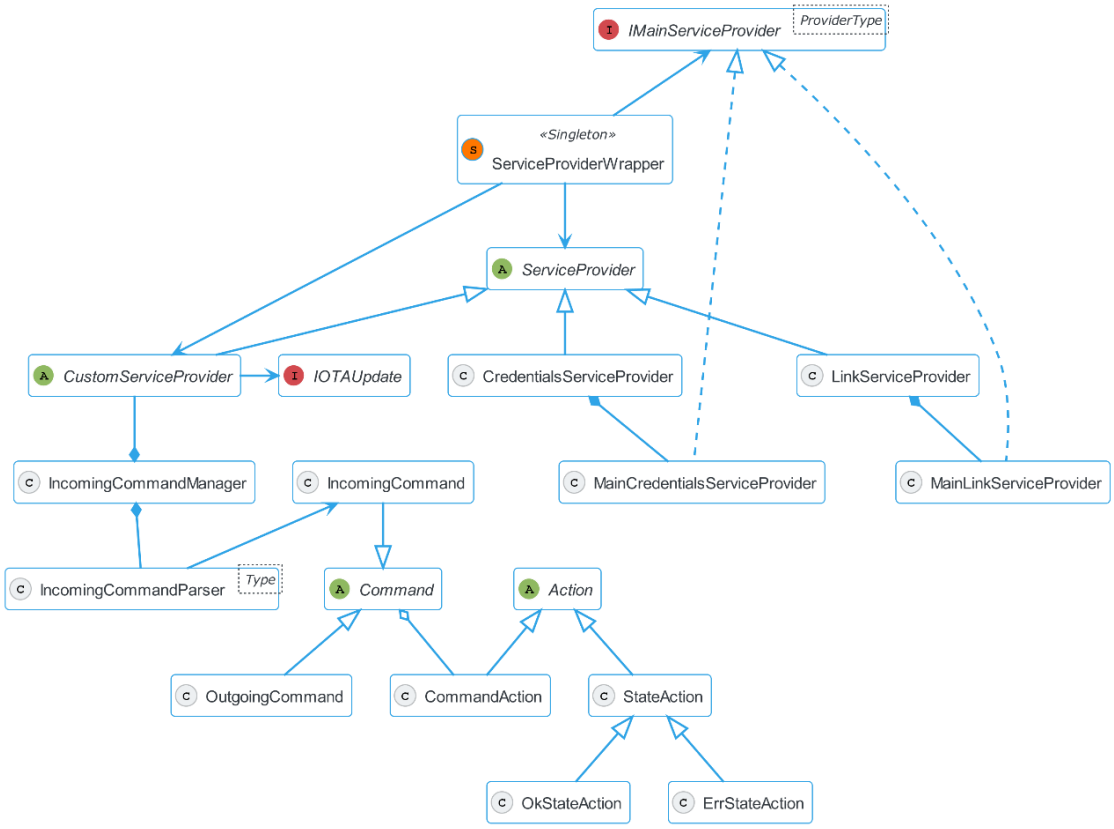

Figura 3.8. Diagrama UML general de proveedores de servicio.

3.2.3 Diagramas de secuencia

En esta sección realizaremos una serie de descripciones de los diagramas de secuencia elaborados, los cuales modelan determinados casos de uso donde se podrá apreciar la interacción entre las distintas entidades.

El diagrama mostrado en la Figura 3.9, muestra el proceso por el cual un dispositivo realiza la solicitud de registro en el sistema donde el servicio correspondiente responde con las nuevas credenciales de acceso. Por otro lado, tenemos la secuencia de la Figura 3.10 donde se puede observar el proceso de vinculación de los dispositivos, este proceso implica la existencia de un usuario previamente registrado. Los diagramas mostrados en la Figura 3.11 y en la Figura 3.12, muestran los dos principales escenarios a la hora de actualizar los dispositivos vía OTA, en el primer caso el dispositivo se encuentra conectado en el momento del lanzamiento y en el otro caso el dispositivo está desconectado. Adicionalmente se han elaborado otros dos diagramas de secuencia adjuntos en el anexo A, donde la Figura A.1 representa el proceso de identificación de los dispositivos cada vez que estos solicitan conectarse con el *broker* y finalmente en la Figura A.2 se muestra el proceso de validación de *topics* que es muy similar al proceso de identificación pero con peculiaridades que detallaremos en la sección 4.2.

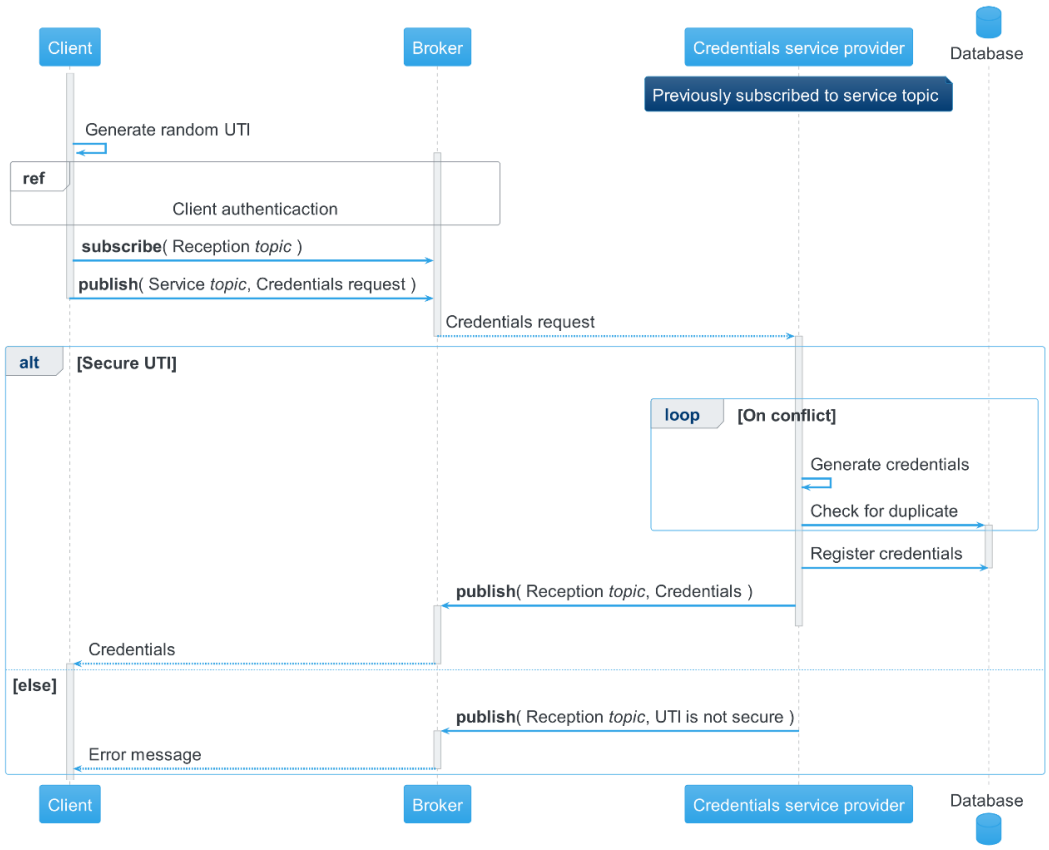

Figura 3.9: Secuencia del proceso de generación e intercambio de credenciales.

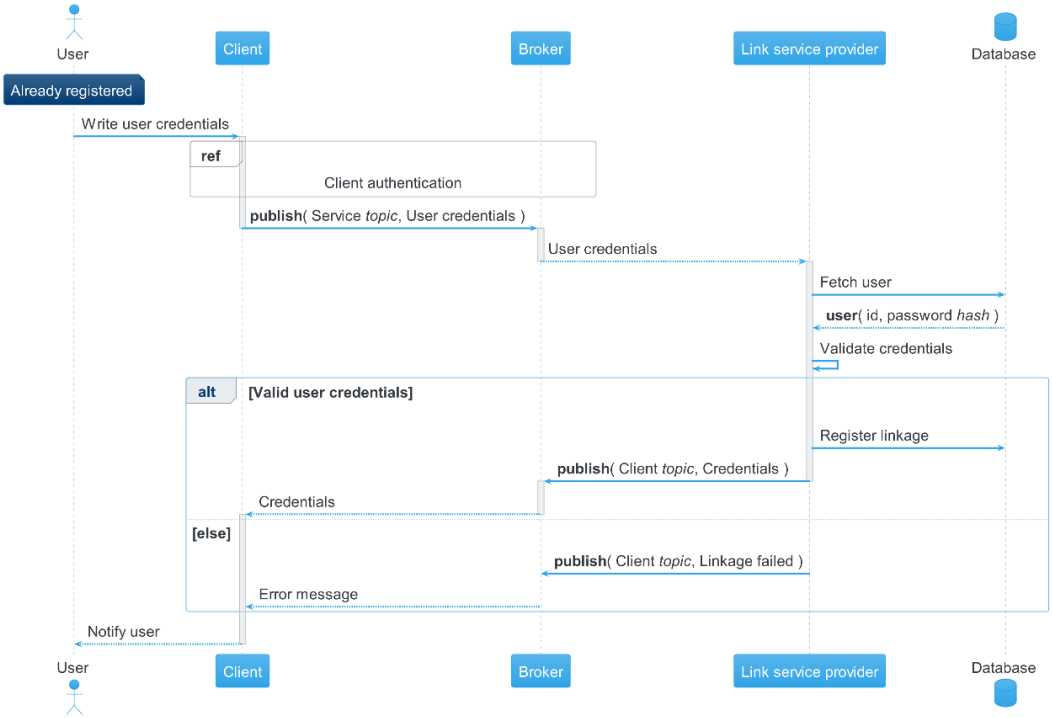

Figura 3.10: Secuencia del proceso de vinculación de dispositivos.

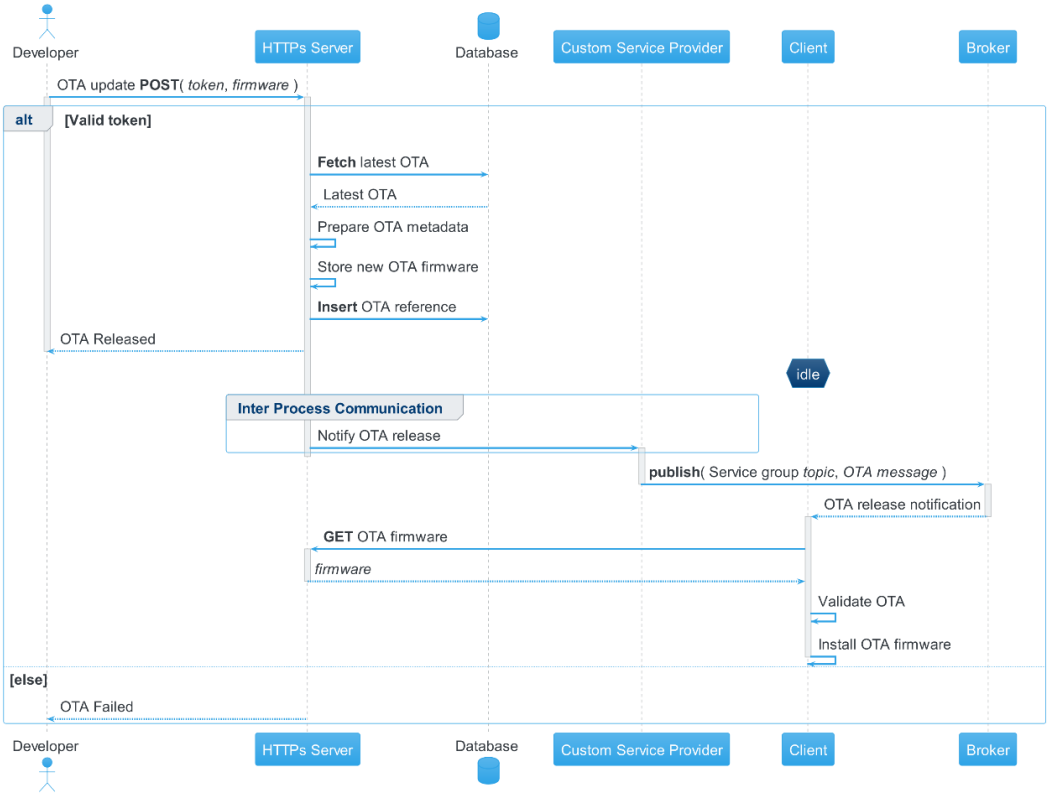

Figura 3.11: Secuencia de actualización OTA pasiva.

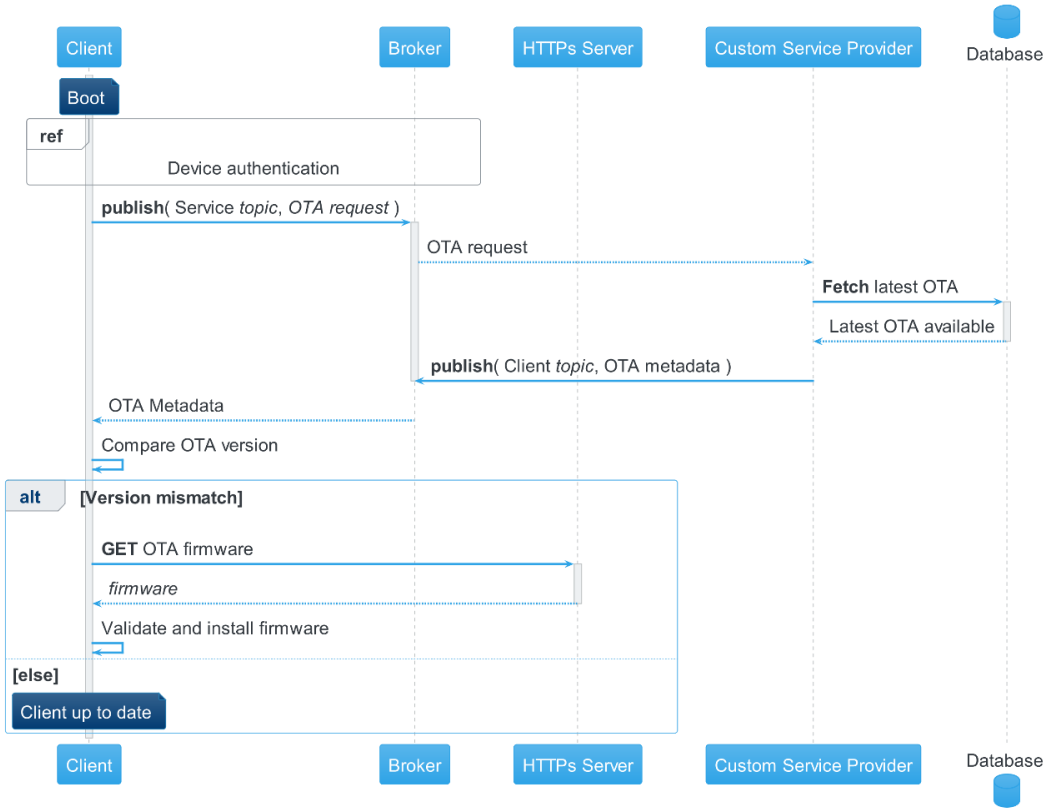

Figura 3.12: Secuencia de actualización OTA activa.

4

Implementación

En base a los requisitos especificados y el diseño planteado, empezaremos detallando distintos aspectos de seguridad de cara a la implementación, posteriormente hablaremos sobre el sistema de gestión de privilegios sobre MQTT, definiremos una serie de protocolos que deben respetarse para garantizar a su vez ciertos aspectos de seguridad en determinados casos de uso y finalmente terminaremos hablando sobre la comunicación entre procesos y los distintos módulos de terceros utilizados.

4.1 Seguridad en la comunicación

Uno de nuestros principales requisitos es garantizar la seguridad a la hora de realizar las comunicaciones entre los distintos dispositivos. Recordemos que estamos utilizando Internet como transporte y, debido a esto, los datos a priori son accesibles por cualquiera que tenga acceso a la red, de tal modo que tendremos que hacer una serie de consideraciones a la hora de implementar nuestro sistema para garantizar cierta seguridad en la comunicación de los dispositivos y los distintos proveedores de servicio. Recordemos que MQTT funciona sobre TCP/IP, por lo que podemos añadir una capa de seguridad denominada TLS, esta capa nos va a permitir garantizar ciertos aspectos de seguridad, como, por ejemplo, que los dispositivos se conectarán siempre con las máquinas originales y no con copias pirata, entre otras cosas. Al tener un sistema de identificación paralelo basado en credenciales, implica que los clientes presentarán en **sus memorias únicamente el certificado original de la autoridad certificadora**, por esta misma razón, el servidor no tendrá a priori garantías de la autenticidad de dicho dispositivo, pero sí podemos garantizar que la información intercambiada no será comprometida ni tampoco alterada, es decir, estamos garantizando integridad y confidencialidad, **pero no autenticidad.**

Es importante hacer notar que la autenticidad sí está garantizada desde el punto de vista del cliente que se conecta al servidor, es decir, un **cliente sí puede ser consciente de que el servidor es quien dice ser**, en cambio, es el **servidor quien desconoce la identidad del cliente** (a nivel de TLS), es un caso análogo al de un usuario que navega por la red usando un navegador: para que un servicio web pueda identificar al usuario se piden unas credenciales de acceso, esto implica que los certificados que tenga el navegador de las autoridades certificadoras tienen que ser originales, ocurre exactamente lo mismo en nuestro caso. En la Figura 4.1 puede observar este planteamiento.

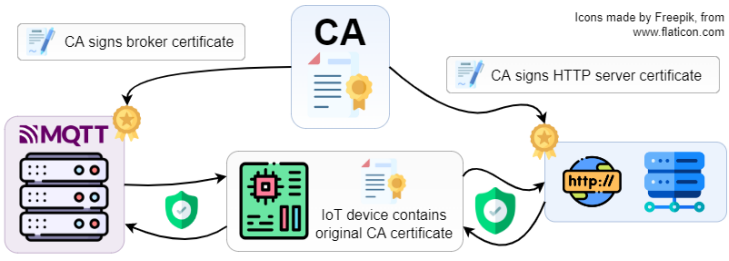

Figura 4.1: Seguridad en la comunicación.

El dispositivo podrá confiar en los servidores de nuestro sistema gracias al certificado de la CA, es importante hacer notar que si quisiéramos implementar un sistema que permitiera distribuir el certificado original de nuestra autoridad certificadora **necesitaríamos una autoridad certificadora superior**, en la que otros **servicios de terceros confíen también**, como podrían ser clientes de navegador y derivados. En el esquema que nosotros estamos planteando actualmente se da por sentado que el **proceso de grabado del certificado de la CA en la memoria de los dispositivos es intrínsecamente seguro,** es decir, que el propio programador (como propietario de su sistema) sería responsable de instalar en los dispositivos el certificado original de la autoridad certificadora.

4.2 Gestión de privilegios sobre MQTT

En base a la necesidad de tener un sistema de registro abierto (RF1), cualquier dispositivo puede conectarse con el *broker* con unas credenciales públicas por defecto (como veremos en la sección 4.3.2) y solicitar unas credenciales de acceso, de tal modo que cualquier dispositivo que conozca la especificación de nuestros protocolos tendría acceso a cualquier *topic*, da igual que estemos utilizando TLS como capa de seguridad si al fin y al cabo el *broker* permite que cualquier dispositivo pueda registrarse y leer la información que intercambian los dispositivos (sea o no de carácter confidencial), es por esto, que tenemos que implementar un sistema de gestión de privilegios (RF2) que permita el acceso a determinados *topics*, y además, tener un control sobre qué dispositivos pueden leer (suscribirse) o enviar (publicar) información en estos.

Comodín	Descripción	Utilizable en
+	Comodín de un solo nivel, puede utilizarse en cualquier parte del *topic*	Suscripción
#	Comodín multinivel, puede utilizarse únicamente al final del *topic*	Suscripción

Tabla 4.1: Uso de *wildcards* en MQTT.

Como se puede ver en la Tabla 4.1, se permite el uso de caracteres comodín en los *topics*, estos comodines son especialmente útiles a la hora de querer consultar información de múltiples *topics* simultáneamente, por ejemplo, un cliente que se suscribe al *topic room/sensor/humidity* puede leer la información de un sensor de humedad, pero también podría suscribirse al *topic room/sensor/+*, de tal modo que puede leer la información de todos los sensores que haya en esa sala, de un modo similar, podemos utilizar el comodín almohadilla para realizar una suscripción multinivel, por ejemplo, una suscripción a *room/#*, suscribiría al cliente de manera implícita a todos aquellos *topic* que se encuentren por debajo de *room*. A raíz de estas características del protocolo MQTT se ha definido una lógica muy similar para el sistema de privilegios, junto a un comodín adicional (ver Tabla 4.2).

Comodín	Descripción	Utilizable en
+	Comodín de un solo nivel, no permite el uso explícito del comodín +	Sub/Pub
*	Comodín de un solo nivel, pero permite el uso explícito del comodín +	Sub/Pub
#	Comodín multinivel, puede utilizarse únicamente al final del *topic*	Sub/Pub

Tabla 4.2: Uso de *wildcards* en el sistema de privilegios.

Es muy importante no confundir la Tabla 4.1 con la Tabla 4.2, la primera de ellas especifica **qué comodines pueden ser utilizados a la hora de suscribirse al *broker***, y la segunda de ellas se utiliza para indicar **el uso de comodines sobre plantillas de *topics***. Para facilitar la comprensión, utilizaremos un sencillo ejemplo, un cliente solicita suscribirse al *topic room/sensor/+* y al mismo tiempo existe el siguiente permiso registrado en la base de datos: *room/sensor/+*, en base a nuestro criterio, la solicitud de **suscripción del dispositivo será rechazada**, en cambio si el registro en la base de datos fuese *room/sensor/**, la suscripción **sí sería aceptada.** Por defecto, los clientes no tienen ningún permiso, salvo las excepciones que se dan para los clientes de carácter público (estos detalles pueden verse en la sección 4.3.2). Si recordamos la Figura 3.3, los clientes tienen la posibilidad de tener *topics* asignados de manera directa o a través de diferentes roles que a su vez consisten en un grupo de plantillas de *topics*, por tanto, necesitamos una consulta que nos permita aplicar dicha lógica conforme a unos criterios de búsqueda, dichos criterios se basan principalmente en el identificador del cliente, en el tipo de permiso y en la calidad del servicio.

En la consulta que se puede ver en el Código B.1, el parámetro *$1* correspondería con el identificador único universal del cliente, el parámetro *$2* con el tipo de permiso y el parámetro *$3* con la calidad del servicio. Como se puede observar en la Tabla 4.3, los permisos se guardan en la tabla como cadenas de caracteres, pero de una longitud muy reducida, si un dispositivo solicita suscribirse a un determinado *topic* el parámetro *$2* sería sustituido por la letra *r* (*read*), en cambio si fuese una solicitud para enviar un mensaje el parámetro *$2* sería sustituido por la letra *w* (*write*).

topic	***permission***	***wildcard***	***max_QoS***
Plantilla del *topic*	Tipo de permiso que tiene un cliente sobre un determinado *topic*, los cuales pueden ser: *r*, *w* o *rw*	Es un campo booleano que restringe el uso de comodines **de manera estricta**	Máxima calidad del servicio permitida: *0*, *1* o *2*

Tabla 4.3: Descripción de las columnas de la tabla *mqtt_topic*.

La consulta a la base de datos nos hace una buena parte del trabajo, pero no en su totalidad, ya que tras finalizar la consulta lo único que tendríamos sería una lista de plantillas de *topics* y nos faltaría validar si realmente el cliente tiene permiso o no al *topic* en cuestión. El procedimiento que vamos a describir a continuación **resulta crítico para que el sistema de privilegios realice las validaciones correctamente**, partimos de dos principales parámetros de entrada, el *topic* en el cual el cliente solicita recibir o publicar mensajes, y por otro lado la lista extraída de la tabla *mqtt_topic* de la base de datos.

Si nos fijamos en la Figura 4.2, este procedimiento se **descompone en dos fases**, la primera de ellas trabaja con el parámetro del *topic* solicitado fragmentándolo en múltiples niveles y al mismo tiempo analiza cada uno de estos creando una pequeña estructura de datos denominada *SplittedTopic* (ver fragmento de Código B.2), esto nos permitirá agilizar el procesamiento en la siguiente fase cuando comparemos cada una de las plantillas en la lista obtenida anteriormente. La segunda fase consiste en **recorrer cada una de las plantillas de la lista hasta encontrar una coincidencia**, lo bueno es, que al tener una estructura con la información sobre el *topic* solicitado, nos ahorramos el tener que volver a procesar dicho *topic* por cada una de las plantillas de la lista, esto resulta efectivo precisamente porque al tratarse de cadenas de caracteres que presentan comodines, el simple hecho de descartar un *topic* por una condición inicial nos permite ahorrar bastante tiempo al no tener que seguir comprobando el resto de caracteres.

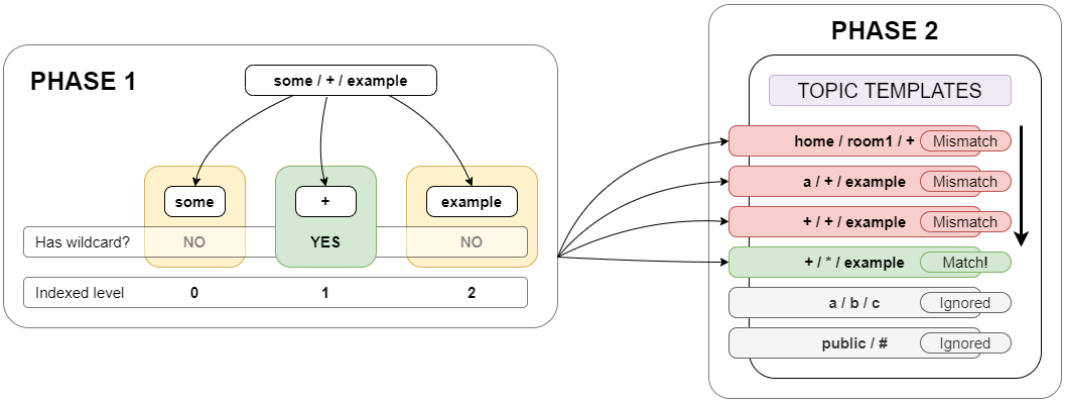

Figura 4.2: Fases en la validación de *topics*.

Como bien hemos dicho antes, el flujo de mensajes entre los distintos clientes puede ser muy continuado y denso, cada vez que un cliente se suscribe a un *topic* o publica un mensaje se realiza una consulta a la base de datos**,** esto **repercutirá negativamente a nuestro gestor** de la base de datos ya que lo estaremos sobrecargando en exceso, esto no resulta ser un problema muy grave si pensamos en las suscripciones, ya que por regla general estas se suelen realizar al inicio de la conexión, pero las publicaciones de mensajes se hacen con muchísima más frecuencia, de tal modo que estas sí resultan ser un problema de cara a nuestro planteamiento, es por esto que adicionalmente se ha implementado un pequeño sistema de cachés.

El **uso de cachés puede aliviar considerablemente el estrés del gestor de la base de datos**, a costa de consumir más recursos que en este caso afectarían directamente a los sistemas de memoria de la máquina, por eso, es posible que el programador configure el sistema de cachés a su gusto, limitando, por ejemplo, el número máximo de entradas que puede tener la caché. Cuando hablamos de cachés debemos tener también muy en cuenta el concepto de invalidación de estas, es decir, debemos tener algún criterio de cara a decidir cuándo deben caducar dichas cachés y así conseguir un ciclo de cacheo satisfactorio. El sistema de *caching* que se puede ver en la Figura 4.3, es relativamente complejo ya que incorpora dos niveles, el primero de ellos consiste en una tabla hash junto a una cola circular que permite encontrar coincidencias exactas del *topic* solicitado, en el caso de que este *topic* no se encuentre en dicha tabla se pasa al segundo nivel donde se realiza la segunda fase del procedimiento mostrado anteriormente, si en este segundo nivel se encuentra una coincidencia sabemos muy bien qué hacer, pero ¿qué ocurre si no la hay? En este caso no nos queda más remedio que consultar de nuevo la base de datos, ya que no siempre podemos permitirnos tener absolutamente todas las plantillas de los *topics* en memoria.

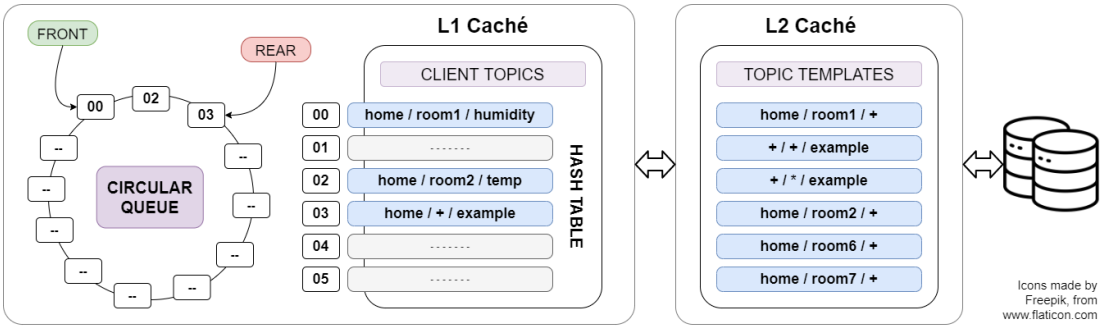

Figura 4.3: Diagrama de cachés de privilegios.

Como se puede apreciar en la Figura 4.3, la caché de **primer nivel contiene los *topics* tal cual han sido enviados por el cliente única y exclusivamente si éste tiene acceso a los mismos**, la caché de segundo nivel presenta las plantillas de los *topics* **tal cual están registradas en la base de datos**. Por ejemplo, si suponemos que las cachés están vacías y el dispositivo solicita suscribirse al *topic home/room/+*, se volcará de la base de datos todos los *topics* de ese cliente con permiso de suscripción, este listado se guardará en la caché de segundo nivel (tenga o no permiso finalmente) y se realizará la comprobación sobre las plantillas descargadas, en caso de que sí tenga permiso, se registrará en la tabla de la caché de primer nivel el *topic* consultado por el cliente. Y en el caso de que la caché de primer nivel esté llena y el *topic* solicitado no se encuentre cacheado, se busca en la caché de segundo nivel si se tiene permiso, y si sigue sin encontrarse la plantilla correspondiente, se hace una consulta a la base de datos, se añaden los nuevos *topics* devueltos por ésta, se busca en este conjunto (no en toda la caché de segundo nivel) si el dispositivo tiene permiso, en caso afirmativo se elimina de la cola circular y de la tabla *hash* el *topic* más antiguo y finalmente se añade el nuevo.

La invalidación de las cachés se realiza cada vez que el dispositivo se desconecta, es decir, la persistencia de la información prevalecerá única y exclusivamente mientras el dispositivo se encuentre conectado. Como se puede observar en la Tabla 4.4, existen una serie de roles por defecto altamente reutilizables que se insertan en la base de datos (si no existen) en la fase de arranque del sistema.

role	***pattern***	***permission***	***wildcard***	***max_QoS***
public	api/+	w	false	0
public	api/+/#	r	true	0
serviceGroup	+/group/#	r	true	2

Tabla 4.4: Roles predefinidos en el sistema de privilegios.

Toda esta lógica ha sido implementada en las funciones de control correspondientes que ofrece la implementación de Aedes [4] para definir el comportamiento que debe seguirse a la hora de aceptar o rechazar a un cliente el poder suscribirse o publicar mensajes en un *topic* determinado.

4.3 Implementación y especificación de protocolos

Dado que el núcleo está compuesto por multitud de tecnologías será necesario definir y especificar el funcionamiento exacto que debe seguirse a la hora de realizar determinadas acciones, como podría ser la generación y el intercambio de credenciales (RF1), el proceso de vinculación para que los dispositivos puedan pertenecer a un usuario (RF6), el protocolo de comunicación de dos vías sobre MQTT (RF3), hasta el protocolo de actualización remota (RF7 y RF8). También hay que resaltar que los protocolos que vamos a especificar no se basan en ningún estándar como tal, estos protocolos han sido elaborados única y exclusivamente para este planteamiento, pero son importantes de cara a la implementación de futuras *APIs* para otros entornos, de esta forma logramos que haya una serie de acuerdos que permitan cierta armonía en la comunicación de las distintas tecnologías y dispositivos. Nótese, que la especificación de estos protocolos se basa en su mayoría en una motivación junto a los distintos aspectos para tener en cuenta antes de especificar exactamente cómo hay que llevar a cabo los distintos procedimientos.

4.3.1 Generación de credenciales

A raíz del requisito RF1 y la secuencia mostrada en la Figura 3.9, necesitamos definir el funcionamiento exacto del protocolo que nos va a permitir realizar el primer proceso de acreditación de los dispositivos que se conectan por primera vez a Internet y por tanto a nuestro servicio. Hay que dejar claro que la generación de las credenciales se realiza en el correspondiente servicio de gestión de credenciales el cual se encuentra implementado de forma nativa en nuestro núcleo (conocido en diseño como *CredentialsServiceProvider*).

A la hora de generar las credenciales debemos tener especial cuidado al especificar el formato y las distintas restricciones (RNF2). Partimos con la ventaja de que estamos generando identificadores **para máquinas y no para humanos**, de tal modo que no tenemos que preocuparnos por el hecho de "*acordarse de una contraseña"*, debido a esto, las credenciales se componen por la combinación de un identificador y una contraseña donde ambos son generados de manera "*completamente aleatoria"*. Con esto nos referimos a que el procedimiento que realizaremos para garantizar esa aleatoriedad será basándonos en la **entropía de la propia máquina**, esta entropía permite reducir la posibilidad de encontrar ciertos patrones en los identificadores y/o contraseñas, recordemos que en un principio cualquier dispositivo puede solicitar identificación y por ende puede generar los identificadores que considere oportunos, esto también implica un riesgo de seguridad que es análogo al caso de registros en otros sistemas donde existe el típico CAPTCHA[3] (además que aplicar este método resultaría incompatible), pero esto está más relacionado con ataques de fuerza bruta, y lo que nos **preocupa en este protocolo es la información que podamos dar a un posible atacante que pretenda buscar patrones en el proceso de generación de credenciales**. Los identificadores son generados utilizando la **cuarta versión del generador de identificadores únicos universales** (*UUID v4*) y las contraseñas se generan cumpliendo con las siguientes restricciones:

- Se crea en base a la **entropía de la máquina** generadora (por ejemplo, utilizando el ruido proveniente de circuitos).
- La longitud de la contraseña es **variable** cumpliendo: $l = 32 + rand(0, 16)$
- Debemos contar con un conjunto de al menos 32 caracteres diferentes, entre los cuales **no se puede encontrar el carácter ASCII decimal 32** (espacio).

El identificador consiste en un **identificador único** (único al menos dentro de nuestro sistema) y, por ende, debemos llevar a cabo un **seguimiento de los identificadores que se hayan generado anteriormente**, aunque la posibilidad de colisión sea prácticamente nula, ya que para conseguir que una de cada dos veces se produzca una colisión ¡sería necesario estar generando miles de millones por segundo durante 100 años! (con la consecuente memoria que esto requeriría). En caso de que ocurriera la remota posibilidad de conflicto no podemos permitir que se guarden dos identificadores idénticos, de modo que, en caso de que se produzca este conflicto se reintenta una vez más para que el conflicto se pueda resolver de forma transparente, si sigue existiendo el conflicto el cliente repetirá la solicitud en el intervalo que considere oportuno.

[3] Completely Automated Public Turing test to tell Computers and Humans Apart

4.3.2 Intercambio de credenciales

Este apartado recoge toda la especificación que describe cómo funciona el protocolo de intercambio de credenciales, es decir, cómo se logra ese intercambio entre el cliente y el proveedor de credenciales, además de garantizar seguridad durante el intercambio (RNF3). En el establecimiento de la primera conexión del cliente se **deberán utilizar unas credenciales de acceso por defecto** que por naturaleza **son públicas** y por tanto no existe ningún peligro si estas se ven comprometidas. Dichas credenciales por defecto son las que se pueden ver en la Tabla 4.5, estas son las credenciales que los dispositivos deberían utilizar en su primera conexión con el *broker* (o siempre y cuando no tengan unas credenciales propias), pero hay que recordar que estamos utilizado el protocolo MQTT para comunicarnos, y nos hace falta suministrar un campo adicional, el cual es el identificador que utiliza el *broker* para administrar las conexiones, de modo que los campos de intento de la primera conexión se verán como en la tabla Tabla 4.5.

Campo	Valor
UTI	Identificador aleatorio generado por el propio cliente MQTT
UUID	00000000-0000-0000-0000-000000000000
Password	0

Tabla 4.5: Paquete de conexión MQTT con identificador único temporal.

Hablemos primero del nuevo concepto de identificador UTI (*Unique Temporary Identifier*) que hemos introducido en la Tabla 4.5, este campo es generado por el propio cliente (el dispositivo que solicita las credenciales), este identificador, como su propio nombre indica, es de carácter temporal y nos va a permitir diferenciar a los dispositivos pese a estar usando las mismas credenciales públicas de acceso. Este **identificador único temporal** está sujeto a las siguientes restricciones:

- Se genera en base a la **entropía de la máquina** generadora.
- La longitud debe ser de **32 caracteres**.
- **No** se pueden utilizar caracteres cualesquiera, es necesario que **sean aceptados por la siguiente expresión regular** `^[a-zA-Z0-9]+$`

El proveedor del servicio encargado de la generación de las credenciales (*CredentialsServiceProvider*) comprobará que **todas las restricciones anteriormente mencionadas sean cumplidas**, en cualquier otro caso la generación de credenciales **no se producirá y se rechazará el intercambio**. Una vez se hayan cumplido los requisitos anteriormente mencionados, se deben hacer una serie de consideraciones previas al intercambio, en primer lugar, todos los dispositivos que se identifiquen con las credenciales por defecto anteriormente mencionadas tienen asociado el rol de *public* (ver Tabla 4.4), el cual tiene única y exclusivamente los siguientes privilegios:

- Permiso para publicar en el *topic* `api/+`
- Permiso para suscribirse en el *topic* `api/+/#`
- En ninguno de los casos anteriores está permitido el uso de calidades de servicio superiores a 0 al igual que el uso de *wildcards*, esto quiere decir que un dispositivo **no podrá suscribirse**, por ejemplo, al *topic* `api/#`, **tendrá que especificar la ruta completa de manera explícita** (ver Tabla 4.4).

Cuando el dispositivo haya generado el UTI, el *broker* debería aceptar la conexión, es ahora cuando el dispositivo y el proveedor deben llevar a cabo los siguientes pasos:

1.-- El dispositivo se suscribirá al *topic* `api/<UTI>/<CRED>` donde `<UTI>` es el identificador temporal que ha generado el dispositivo y `<CRED>` es el identificador de la acción en cuestión el cual es -1.

2.-- El dispositivo publicará un mensaje en el *topic* `api/new` incluyendo en el cuerpo del mensaje el UTI que haya generado anteriormente.

3.-- El proveedor al recibir dicho mensaje generará las credenciales de usuario como se especificó en la sección 4.3.1.

4.-- Una vez se hayan generado las credenciales (y no existan conflictos) el proveedor **otorgará permisos de suscripción y publicación a dicho dispositivo** en el *topic:* `+/<UUID>/#` . También otorgará permisos **de suscripción** en el *topic*: `+/group/#` asignando el rol *serviceGroup* (ver Tabla 4.4)

5.-- Finalmente, si todo ha salido correctamente, el servidor publicará en el *topic* `api/<UTI>/<RET>` un mensaje, donde `<RET>` representa el código del resultado de la operación. En caso de que el resultado sea exitoso, el servidor **incluirá en el cuerpo del mensaje de la respuesta las nuevas credenciales**, de la siguiente forma: `<UUID> <PSW>` **nótese el espacio** entre el identificador y la contraseña.

6.-- Una vez el cliente haya recibido sus nuevas credenciales, **éste deberá registrarlas en su memoria interna**.

7.-- En caso de que se produzca cualquier fallo o desconexión durante este proceso, el cliente solicitante deberá repetir los pasos anteriormente mencionados **generando un nuevo UTI por cada intento**.

Dado que estamos realizando múltiples consultas de manera secuencial en la base de datos, se ha encapsulado todo este procedimiento en una **transacción** para evitar dejar los datos en un estado inconsistente en caso de que se produzca un error en alguna de las consultas intermedias. Para ayudarnos en la elaboración de transacciones y construcción de consultas, nos hemos apoyado en un *query builder* denominado *Knex* [3].

4.3.3 Comunicación de dos vías sobre MQTT

Este proceso está estrictamente relacionado con las futuras conexiones que los clientes harán tras haber sido identificados y surge a raíz del requisito RF3, el cual nos permitirá hacer comunicaciones bidireccionales entre el cliente y el proveedor de servicios, es decir, crearemos un canal de envío de mensajes desde el cliente al proveedor de servicios y otro canal para el envío de mensajes desde el proveedor a un cliente en específico.

A la hora de implementar este protocolo, debemos tener en cuenta que los dispositivos no están solos y pueden ser interrumpidos por otros clientes malintencionados que pretendan corromper el servicio. En el intercambio de credenciales dejamos claro cómo generar el identificador que se utiliza para identificar a los dispositivos dentro del *broker* (UTI), pero esto era necesario porque los dispositivos tenían el mismo identificador de usuario, pero en este caso las credenciales ya no son públicas y son únicas en el sistema, es por esto por lo que tenemos que definir un nuevo concepto de identificador de conexión.

Como puede verse en la Figura 4.4, existen diversos escenarios, el primero de ellos supone que el identificador de conexión es igual al identificador de usuario, en el segundo escenario se fuerza a que el identificador de usuario sea igual al identificador de la conexión, y en el tercer y último escenario consiste en seguir manteniendo el UTI como identificador único temporal de la conexión.

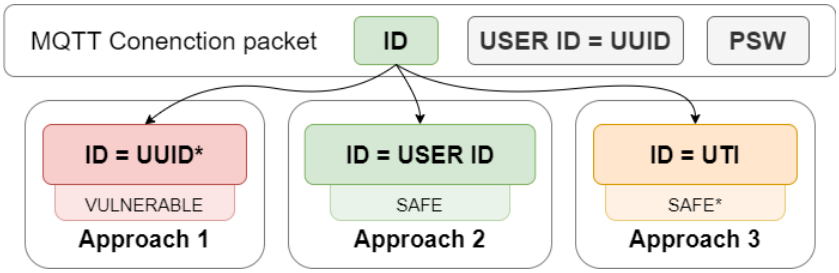

Figura 4.4: Escenarios en el paquete te conexión MQTT.

El primer escenario resulta ser vulnerable por el simple hecho de estar dando por sentado que el dispositivo cliente va a iniciar la conexión con el mismo identificador que el identificador de usuario, podría pensarse que es irrelevante, pero si nos ceñimos a la especificación del protocolo MQTT [5] podemos observar que, si se da el caso en el cual un cliente se encuentra conectado con un determinado identificador de conexión y momentáneamente se conecta otro cliente con ese mismo identificador, el cliente que se encuentra actualmente conectado **será desconectado de manera forzosa**, y se cederá la conexión al nuevo cliente. Debido a esto estaríamos arriesgándonos a un **ataque de interrupción permanente por suplantación de identificador de conexión**, es decir, un cliente malintencionado podría estar interrumpiendo la conexión de manera indefinida a un cliente legítimo por el simple hecho de conocer su identificador. En el segundo escenario este problema queda resuelto completamente ya que añadiendo esta restricción obligamos al cliente a identificarse correctamente (protegiendo así a los dispositivos legítimos), y por último, el tercer escenario resulta seguro por el simple hecho de ser un identificador temporal (se genera un identificador aleatorio por cada conexión), pero la razón por la cual no la hemos elegido, es por el simple hecho de la **existencia de sesiones en el protocolo MQTT**, el estar generando un identificador por cada conexión rompería con el concepto de sesión, y perderíamos la posibilidad de que nuestros dispositivos pudiesen recuperar mensajes usando calidades de servicio superiores a 0.

Para acordar una nomenclatura de **comunicación bidireccional asíncrona** con los **proveedores de servicio** a través del *broker*, el dispositivo cliente deberá suscribirse al *topic* `<SERVICE_ID>/<UUID>/s/#`. En el caso de que un dispositivo soportase un servicio denominado *SKYL1*, el *topic* correspondiente sería: `SKYL1/<UUID>/s/#`. Dado que los proveedores de servicio tienen la capacidad de gestionar múltiples clientes simultáneamente, la suscripción será ligeramente diferente a la de un dispositivo, el proveedor del servicio `<SERVICE_ID>` deberá suscribirse al *topic* `<SERVICE_ID>/+/m/#`. En el caso de que el proveedor desee enviar un mensaje en concreto a un cliente en específico con identificador `<UUID>` que soporta el servicio `<SERVICE_ID>`, deberá publicar un mensaje en el *topic* `<SERVICE_ID>/<UUID>/s/#`. En la figura Figura 4.5 es posible observar este comportamiento al igual que puede verse cómo el proveedor de servicio actúa como un cliente (para el *broker*), pero los protocolos de generación e intercambio de credenciales anteriormente mencionados no se aplican para el establecimiento de la conexión de los proveedores de servicio, esto es debido a que los proveedores se encuentran generalmente en la misma máquina o una red local más restringida de tal modo que la conexión de estos se trata de manera distinta.

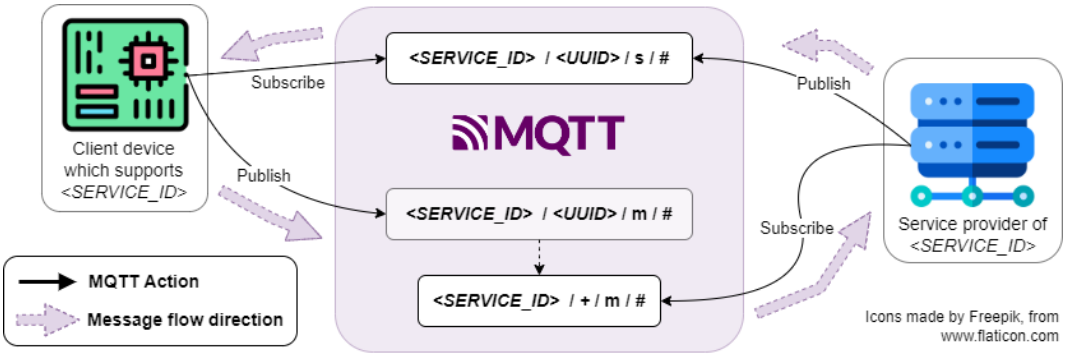

Figura 4.5: Comunicación de dos vías sobre MQTT.

Como bien puede verse en la Figura 4.6, los proveedores de servicio siguen un acuerdo con los dispositivos, en el cual se permite enviar un mensaje a todos los dispositivos de manera simultánea (RF4), esto nos resultará especialmente útil a la hora de realizar las actualizaciones OTA de forma pasiva (ver sección 4.3.5).

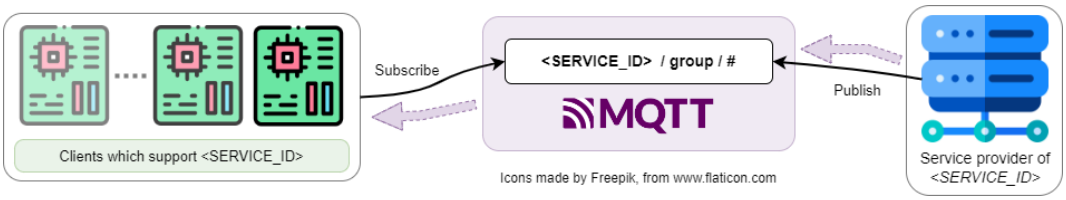

Figura 4.6: Comunicación grupal.

La Figura 4.7 representa la estructura genérica de *topics*, ésta viene a raíz del requisito (RF5), la cual puede verse como una solución al intercambio de los distintos comandos. En diseño se mencionó el concepto de *Action*, estas acciones se pueden entender como funcionalidades más específicas dentro del propio servicio, es por esto que el programador puede extender la funcionalidad añadiendo sus propios conceptos de *Action.* Nótese también la intención de representar los *topics* utilizando el menor número de caracteres posible ya que no tenemos la necesidad de que los *topics* sean interpretados por humanos, de tal modo que podamos ahorrar la mayor cantidad de memoria posible.

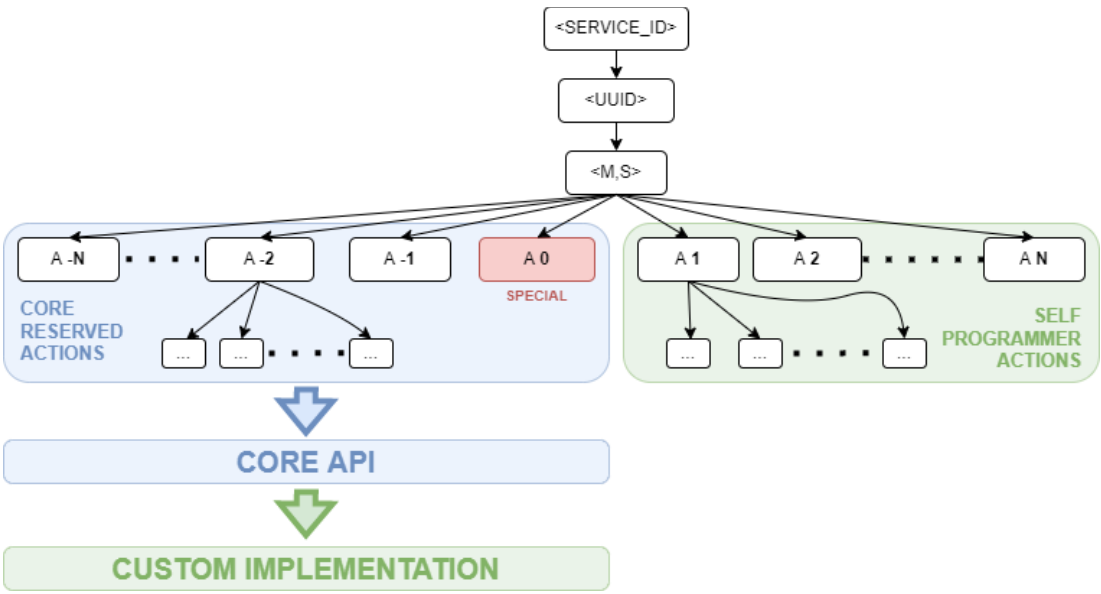

Figura 4.7. Estructura jerárquica de *topics.*

La interfaz de comandos implementada se basa también en el concepto de acción (*Action*) que facilita al programador aún más la estructuración del contenido de los mensajes, pero siguiendo una gramática determinada muy sencilla, la cual permite estructurar el contenido de un mensaje con el uso de espacios. Los comandos ofrecen una estructura simple que es fácilmente interpretable y no requiere mucha lógica, estos heredan de *Action* por lo que también son considerados como acciones dentro de un servicio, pero ofreciendo una funcionalidad más concreta.

Es por esto por lo que la clase correspondiente al proveedor de servicio personalizado del programador (*CustomServiceProvider*) incluye un gestor de comandos denominado *CommandManager*, este gestor procesa el mensaje y busca los comandos registrados previamente por el programador, permitiendo así ejecutar *callbacks* en función del comando que se haya recibido, estos *callbacks* reciben un argumento que contiene la información en una estructura del propio lenguaje que el programador haya definido.

La Figura 4.8 se corresponde con una extensión de las hojas mostradas en el árbol de la Figura 4.7, donde se puede observar un ejemplo del envío de un comando a través del canal maestro *m* (el dispositivo envía un mensaje al proveedor de servicio), se aprovecha el mismo *topic* para la identificación del comando en cuestión y en el cuerpo del mensaje se incluyen los parámetros o la información que se considere oportuna, en la parte derecha de la Figura 4.8 podemos observar dos ejemplos distintos de la estructura de un comando que consiste en una cadena de caracteres separada por espacios para la diferenciación de los distintos parámetros.

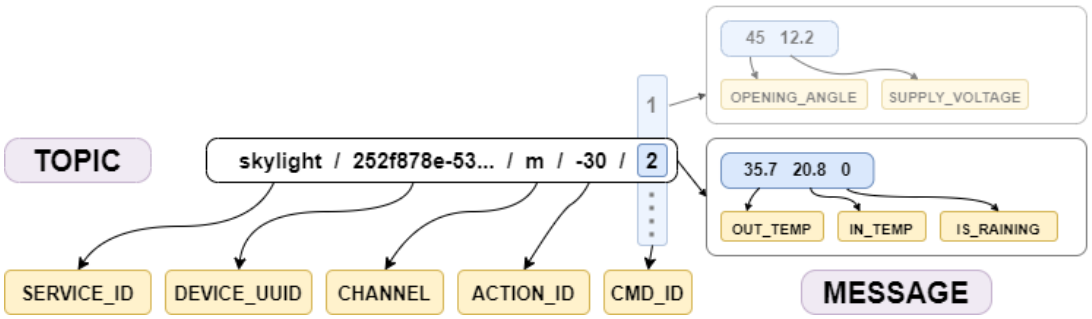

Figura 4.8: Ejemplo de envío de un comando.

4.3.4 Vinculación de dispositivos

El protocolo de vinculación surge a raíz del requisito RF6 y define el proceso por el cual es posible crear un vínculo entre un usuario y un dispositivo, para ello podemos apoyarnos en la secuencia mostrada en la Figura 3.10. Este servicio se encuentra implementado de forma nativa en el núcleo (conocido en diseño como *LinkServiceProvider*). Es necesario que el usuario intervenga para realizar el proceso de vinculación a diferencia del proceso de generación e intercambio de credenciales que se realizan de manera automática, debido a la multitud de dispositivos que puedan existir, la forma en la que el usuario interactúa con el mismo se escapa de nuestro ámbito, pero la forma en la que el dispositivo se comunica con el proveedor sí debemos contemplarla. El proceso de vinculación se ha implementado como una acción (*Action*) nativa de este núcleo, más concretamente como una acción de estado (*StateAction*), que permite informar al dispositivo sobre el resultado de la vinculación, de forma que los pasos que se llevan a cabo para la vinculación son los mostrados a continuación:

1.-- El usuario escribirá sus credenciales en la memoria del dispositivo a través de distintas interfaces (servidor web local, conexión Bluetooth, USB …).

2.-- El *topic* `<LINK_SERVICE_ID>/<UUID>/m/-10` será utilizado por el dispositivo para enviar un mensaje con las credenciales del usuario de la siguiente forma: `<USERNAME> <PASSWORD>` (nótese el espacio entre ambos valores y la necesidad de que el nombre de usuario y la contraseña no puedan contener espacios).

3.-- El servicio de vinculación consultará en la base de datos la existencia de dicho usuario, y en caso afirmativo se calcula el *hash* de la contraseña suministrada por el usuario y se compara con el registro de la base de datos.

4.-- Una vez realizada la comprobación de las credenciales del usuario, el proveedor publicará un mensaje en el *topic* `<LINK_SERVICE_ID>/<UUID>/s/-10/<STATE>`, donde `<STATE>` representa el código de estado, en caso de que el resultado resulte exitoso el cuerpo del mensaje irá vacío, en cualquier otro caso el mensaje incluirá una razón del error.

4.3.5 Actualización remota de dispositivos OTA

Para lograr que los dispositivos puedan ser actualizados de manera remota debemos considerar dos principales escenarios, en el primer escenario (RF7) el programador lanza una actualización OTA mientras el dispositivo se encuentra conectado (ver diagrama de secuencia en la Figura 3.11), es decir, el dispositivo es notificado sobre dicho evento. En el caso opuesto (RF8), si el dispositivo se encuentra desconectado (consultar diagrama de secuencia en la Figura 3.12) éste deberá consultar por su cuenta la última versión disponible, de este modo podemos lograr que los dispositivos siempre tengan la oportunidad de actualizarse con o sin la garantía de que estén conectados.

Debemos recordar que, de cara a una actualización OTA, el simple hecho de tener que transferir un fichero (generalmente binario) de tamaño considerable implica ciertas necesidades sobre el protocolo que utilizamos para facilitar el envío de dichos ficheros a los diferentes dispositivos, el problema radica en que MQTT no es un protocolo diseñado para intercambiar archivos como tal, no obstante, es posible implementar este servicio de actualizaciones sobre MQTT pero estaríamos **provocando un estrés innecesario sobre el *broker***, ya que en la mayoría de los casos tendríamos que fragmentar los ficheros para conseguir enviarlos a trozos, y eso supondría un intercambio excesivo, aumentando así innecesariamente el tráfico de mensajes, pero como bien hemos podido ver en el diseño, disponemos de un servidor HTTP que nos resuelve este problema, de esta forma, utilizaremos MQTT para que los dispositivos intercambien los mensajes relacionados con los metadatos asociados a la actualización en sí, y el servidor HTTP permitirá distribuir dichos ficheros entre los distintos dispositivos. A continuación, detallamos el proceso que se lleva a cabo para que los dispositivos realicen la actualización de forma activa:

1.-- El dispositivo usa el *topic* `<SERVICE_ID>/<UUID>/m/-20` y envía un mensaje con el cuerpo vacío.

2.-- El proveedor de servicio usará el *topic* `<SERVICE_ID>/<UUID>/s/-20` y publicará un mensaje con el siguiente formato: `<VERSION> <SIZE> <PATH>` (nótese los espacios entre los datos asociados al *firmware*).

3.-- Cuando el dispositivo reciba el mensaje en el *topic* correspondiente realizará la siguiente comprobación `<OTA_VERSION> != <CURRENT_VERSION>`

4.-- En caso de que la versión resulte distinta el dispositivo procederá a realizar la actualización haciendo una petición por HTTP al recurso indicado en el `<PATH>`.

Por otro lado, si el dispositivo se encuentra actualmente conectado y de forma pasiva recibe la notificación de que hay disponible una actualización, los pasos resultan ser idénticos pero ahorrándonos el primer paso y con la peculiaridad de que cuando el dispositivo solicita de manera activa la última versión, el proveedor de servicios conoce su identificador para responderle, el problema es, que en este escenario el dispositivo no realiza la solicitud, sino que el proveedor de servicios envía el mensaje directamente al *topic* `<SERVICE_ID>/group/-20`, a partir de aquí el dispositivo continuaría con los pasos 3 y 4.

Cada vez que se lanza una actualización, el sistema guarda una copia del *firmware* suministrado en el sistema de ficheros basándose en la ruta configurada por el programador para llevar a cabo el correspondiente control de versiones (RNF4). Nótese que el proceso por el cual el firmware llega al servidor no se contempla en esta implementación, depende enteramente del programador y cómo desee gestionar esa subida. Este *core* lo que ofrece son los *wrappers* que permiten notificar al servicio correspondiente sobre dicha actualización (consultar sección 4.4), para esto existe el método *publishOTA(idService, update)*, el argumento *idService* representa el identificador del servicio para el cual se lanza dicha actualización, y el parámetro *update* contiene todos los datos asociados a dicha actualización (tamaño, versión, *path* del recurso).

4.4 Comunicación con los proveedores de servicio

Sea cual sea la finalidad de un servicio, todos y cada uno de ellos se encapsulan en procesos independientes para poder distribuir mejor la carga. Gracias a la estructura de *topics* mostrada en la sección 4.3.3, tenemos la garantía de que un proveedor recibirá los mensajes única y exclusivamente relacionados con su propio servicio, de esta forma facilitamos la convivencia entre estos evitando conflictos, esto implica que si queremos comunicarnos con dichos servicios (a nivel de proceso) resulta necesario implementar un adaptador que facilite la comunicación entre estos de forma genérica (RNF5), es por esto que existe un *wrapper* (conocido en diseño como *ServiceProviderWrapper*) que abstrae esta comunicación. Para facilitar la implementación nos hemos apoyado en un paquete de NodeJS denominado *node-ipc* [6], el cual facilita el proceso de comunicación entre los distintos procesos, además, si estos servicios se ejecutan en una misma máquina, podemos utilizar los famosos *Unix Domain Sockets* que nos permiten evitar la interfaz de red y reducir tiempos en términos de latencia.

La creación de estas comunicaciones entre procesos se realiza de forma perezosa, es decir, el *wrapper* se encarga de gestionar si es la primera vez que se abre una comunicación con ese proceso y en consecuencia crea una nueva instancia de la comunicación de forma totalmente trasparente al programador, a partir de aquí el módulo *node-ipc* gestiona el resto del ciclo de vida de la comunicación con el proceso. Para poder notificar sobre distintos eventos a los procesos, se ha implementado una función denominada *notifyService(idService, event, data)*, el primer argumento *idService* identifica al proveedor de servicio que debe ser notificado, *event* hace referencia al nombre del evento y *data* representa el objeto que se envía, como, por ejemplo, el objeto de una actualización OTA, este objeto se convierte a un objeto JSON que permite ser reconstruido una vez es recibido por el proceso destino.

En la Figura 4.8, se muestra un ejemplo de interacción entre el servidor HTTP y los distintos servicios en procesos independientes, nótese que el *ServiceWrapper* no actúa como un cuello de botella, lo que se pretende representar es que cada uno de los procesos utiliza un envoltorio con una serie de utilidades que le permiten recibir y enviar eventos, pero el encargado de la comunicación al fin y al cabo es el propio sistema operativo.

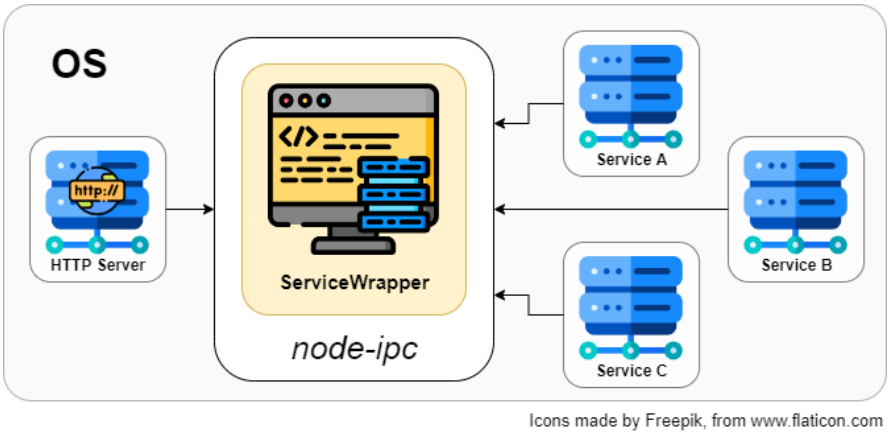

Figura 4.9: Esquema de comunicación entre procesos.

4.5 Paquetes y módulos de terceros

Para enriquecer el proceso de implementación se han utilizado distintos paquetes de terceros para determinadas aplicaciones, por ejemplo, como mencionábamos en la introducción del diseño, para ofrecer la posibilidad de que el programador pueda utilizar un gestor de base de datos diferente, como por ejemplo *MySQL*, *MariaDB* u *Oracle* hemos utilizado un constructor de consultas denominado *Knex*, se puede ver un ejemplo de uso en el fragmento de Código B.5. Este constructor de consultas permite también el uso de transacciones de tal forma que hemos podido encapsular las tareas que así lo necesiten, como, por ejemplo, cuando se crea el registro de un dispositivo en el sistema (como se especifica en la sección 4.3.2), además, *Knex* trae consigo una interfaz de comandos que permite gestionar el proceso de migración y sembrado de las bases de datos, lo cual resulta especialmente útil. Para la generación de los identificadores únicos universales se ha utilizado un paquete denominado *uuid* [7], este paquete ofrece múltiples protocolos de generación y sigue los requerimientos formales especificados en los estándares correspondientes. Por otro lado, para la generación de las credenciales, se ha utilizado un paquete denominado *crypto-random-string* [8] que nos permite generar cadenas de caracteres basadas en la entropía de la propia máquina, cumpliendo así con la especificación de la sección 4.3.1.

5

Pruebas y resultados

A raíz de los distintos escenarios de seguridad que hemos tenido que considerar de cara a la implementación, nos centraremos en una de las pruebas más críticas de todo este sistema, la cual está estrechamente relacionada con el proceso de intercambio de credenciales; adicionalmente probaremos la efectividad del sistema de cachés descrito en la sección 4.2.

5.1 Ataque por suplantación del identificador único temporal

El diagrama de secuencias mostrado en la Figura 5.1, representa el escenario en el cual un cliente malintencionado pretende comprometer las credenciales de un cliente legítimo. Evidentemente, es un caso muy especial ya que es altamente improbable que se generen dos identificadores temporales idénticos siguiendo las pautas mostradas en la sección 4.3.2, pero plantearnos esta situación nos permite ver que a pesar de encontrarnos en un caso extremo el sistema sigue siendo seguro, ya que no pasa nada si el intruso consigue robar las credenciales del dispositivo legítimo, lo importante es que **no haya dos dispositivos con las mismas credenciales**. Para poner en práctica este acontecimiento, se ha elaborado un pequeño *script* (ver fragmento de Código B.4) que pretende simular la secuencia mostrada en la Figura 5.1.

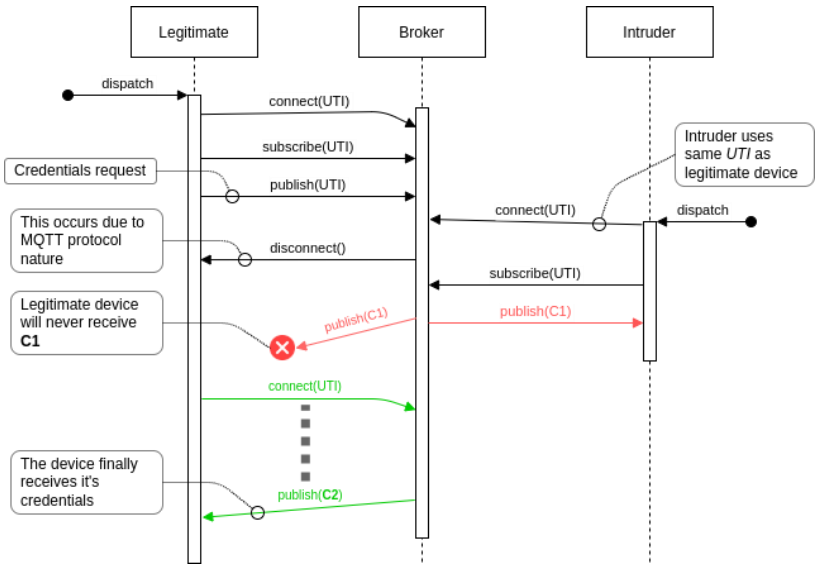

Figura 5.1: Secuencia del ataque por suplantación del UTI.

- *connect(id)* - Representa el inicio de la conexión utilizando el parámetro *id* como identificador de conexión, en el diagrama se utiliza el parámetro *UTI*.
- *subscribe(id)* - Representa la acción de suscribirse al *topic* utilizando el parámetro *id*.
- *publish(id)* - Representa la acción de publicar en el *topic* utilizando el parámetro *id*.
- *disconnect()* - Evento de desconexión.

Esta prueba consiste en iniciar dos conexiones, una conexión para el intruso y otra para el cliente legítimo, cada uno de estos clientes presenta dos conjuntos independientes de credenciales inicialmente vacíos (C_x), el sistema es seguro si tras finalizar los sucesivos intentos de robo se cumple:

$$C_{Legitimate} \cap C_{Intruder} = \emptyset$$

Lo más interesante de esta prueba es que podemos observar (en la Figura 5.1) cómo el intruso realmente roba las credenciales (C1 en el diagrama de la Figura 5.1) del dispositivo legítimo, a pesar de que este último es el que realiza la solicitud de credenciales. El intruso trata de conectarse en el momento más crítico para intentar interceptar las credenciales, el problema que tiene éste es que la especificación del protocolo MQTT (como habíamos mencionado en la sección 4.3.3) no acepta la conexión de dos dispositivos con el mismo identificador de conexión, por eso, cuando el intruso se conecta, antes de que pueda llegar a suscribirse, el dispositivo legítimo es desconectado de manera forzosa. Es por esto por lo que al finalizar las pruebas lo más normal es que veamos credenciales en ambos conjuntos, pero nunca habrá una copia idéntica de las credenciales en ambos. Si se da el extraño caso en el cual un intruso consigue suplantar el identificador, el dispositivo legítimo tras ser desconectado **reintentará la conexión con un nuevo UTI** (como señalábamos en la sección 4.3.2) de tal forma que evitamos que el interceptor pueda interrumpir de manera indefinida al dispositivo legítimo.

5.2 Rendimiento del sistema de cachés

Como ya habíamos mencionado en la sección 4.2, la implementación del sistema de cachés teóricamente debería reducir el estrés del gestor de la base de datos y además acelerar los tiempos de respuesta por parte del *broker*. Es por esto por lo que se han diseñado una serie de pruebas que se encargan de comprobar si realmente resultan tan efectivas como se piensa. Estas pruebas consisten básicamente en incrementar de manera progresiva el número de clientes conectados al *broker*, por cada uno de estos incrementos cada uno de ellos se suscribe en el *topic* `cacheTest/<UUID>/s/rx` y publica un mensaje cualquiera en ese mismo *topic* un número determinado de veces, y se estima el tiempo que se ha tardado en recibir ese mismo mensaje de vuelta. Las pruebas que se muestran a continuación han sido realizadas con los parámetros mostrados en la Tabla 5.1.

Intervalo de clientes	Incremento	Mensajes por cliente
[100, 2000]	100	50

Tabla 5.1: Parámetros para las pruebas del sistema de cachés.

Por cada iteración se vuelca de la base de datos las credenciales de los clientes que corresponda, acto seguido se conectan dichos clientes (como excepción debido a las pruebas, las contraseñas de los clientes se guardaron previamente en texto plano), y una vez conectados, cada uno realiza 50 intercambios de mensajes de manera secuencial, es decir, el siguiente mensaje no se manda hasta que se haya recibido el anterior, antes de enviar de nuevo el mensaje se realiza una espera aleatoria comprendida entre 25 y 250 milisegundos, esto nos permite tener una distribución de mensajes más realista a lo largo de la línea de tiempo. Una vez se haya finalizado la iteración se desconectan todos los clientes y se vuelve a empezar este mismo procedimiento, pero con un mayor número de clientes (registrando los resultados correspondientes).

Como se puede observar en la Figura 5.2, el crecimiento en ambas situaciones es aproximadamente lineal, sin embargo, el aumento del tiempo de respuesta respecto al número de clientes con el uso de caches es mucho menor, cabe destacar que estos tiempos pueden parecer algo elevados a pesar de la poca cantidad de clientes que hay conectados, el problema está en que cada uno de estos clientes está intercambiando 50 mensajes, de modo que en el caso de 2000 clientes tendremos un intercambio total de 100000 mensajes, otras de las penalizaciones en estos tiempos también radica en el simple hecho de que las pruebas se han realizado en la misma máquina, es decir, los dispositivos y el *broker* se han ejecutado localmente por lo que el estrés es mucho mayor, en situaciones reales estos tiempos serían menores pero quizá con un mínimo más elevado debido a la latencia de la red en la que se pudieran encontrar.

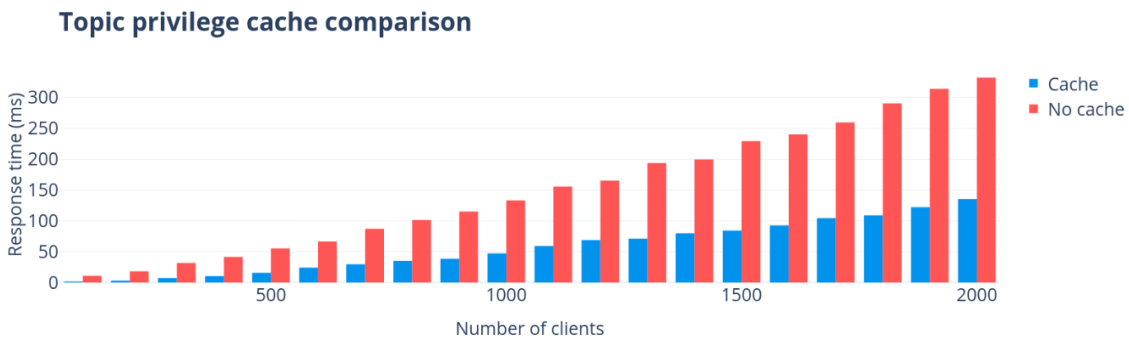

Figura 5.2: Sistema de privilegios con y sin el uso de cachés.

Si nos fijamos en las gráficas mostradas en la Figura 5.3 y Figura 5.4, se puede observar que con el uso de cachés los máximos y los mínimos están más acotados, lo que implica que hay mayor estabilidad y se traducirá en tiempos de respuesta más uniformes durante el intercambio de mensajes.

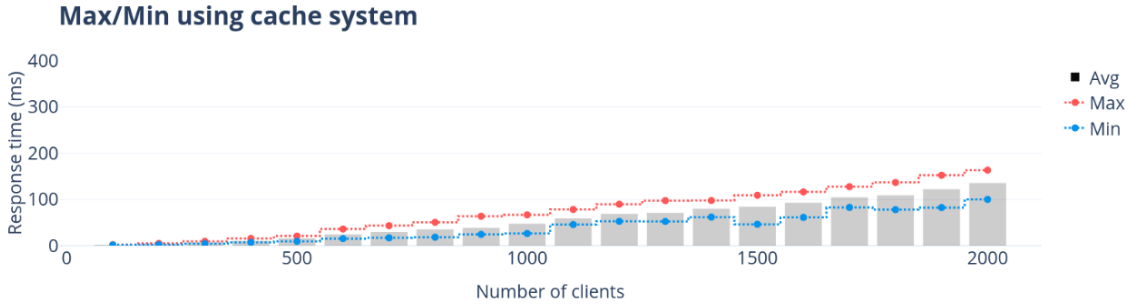

Figura 5.3: Tiempo de respuesta máximo y mínimo con cachés.

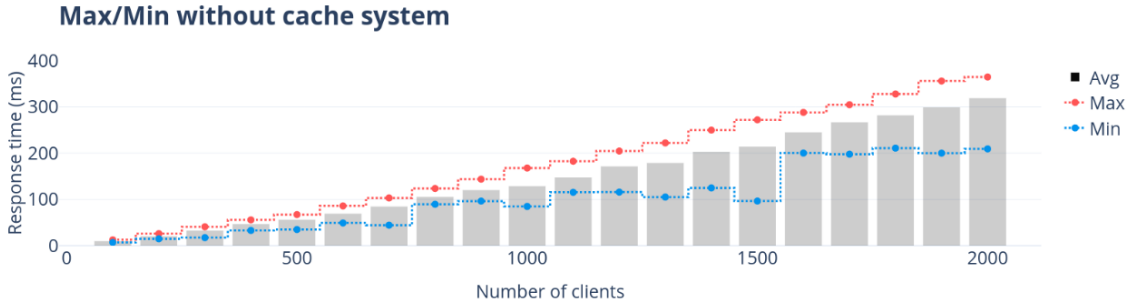

Figura 5.4: Tiempos de respuesta máximo y mínimo sin cachés.

La gráfica mostrada en la Figura 5.5 resulta del cálculo de la media de la desviación típica del tiempo de respuesta de cada uno de los clientes, el uso de cachés disminuye considerablemente dicha desviación, afianzando los resultados de las gráficas mostradas en la Figura 5.3 y Figura 5.4.

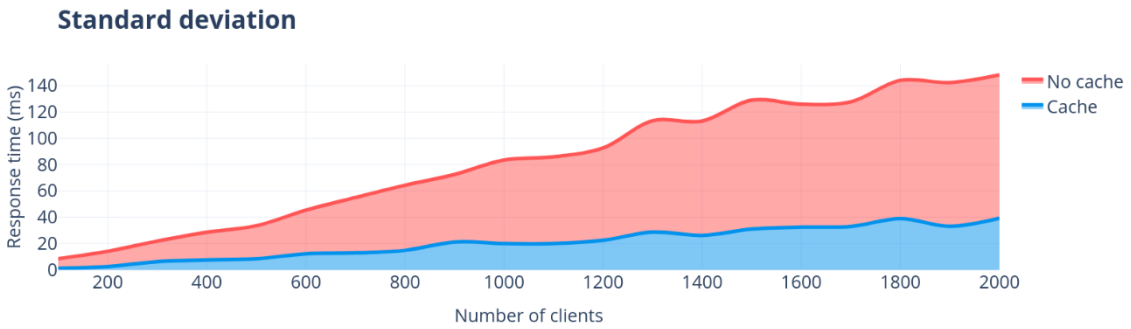

Figura 5.5: Desviación típica del tiempo de respuesta con y sin el uso de cachés.

Dado que nuestro principal objetivo es disminuir el estrés del gestor de la base de datos, en las gráficas mostradas en la Figura 5.6 y Figura 5.7 (ambas extraídas del gestor de la base de datos durante la realización de las pruebas) se puede observar la diferencia en el número de transacciones por segundo. La gráfica que muestra el estrés utilizando las cachés (ver Figura 5.6) muestra una serie de picos que se corresponden con el momento del inicio de cada iteración (cuando se realiza el primer *fetch* para rellenar las cachés). En contraste, si no hacemos uso de las cachés (ver Figura 5.7), la fase de llenado de estas es la menos estresante, ya que el resto del tiempo se está haciendo una consulta a la base de datos por cada mensaje que se envía.

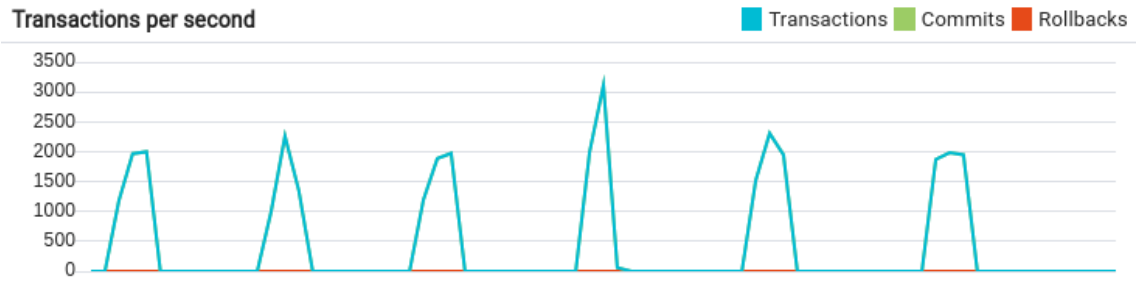

Figura 5.6: Número de transacciones por segundo con cachés.

Figura 5.7: Número de transacciones por segundo sin cachés.

6

Conclusiones y trabajo futuro

6.1 Conclusiones

Hemos comenzado analizando una serie de protocolos para la comunicación entre los distintos clientes al mismo tiempo que hemos debatido cuál era el mejor para cumplir con nuestros objetivos, concluyendo que MQTT resulta ser un protocolo muy flexible e ideal para nuestro ámbito. A raíz de la elección del protocolo, hemos planteado distintos escenarios de identificación, donde hemos podido observar que la mejor opción en base a nuestras condiciones era generar los identificadores después del proceso de elaboración de los dispositivos, lo que ha supuesto desarrollar distintos sistemas como la gestión de privilegios para restringir el acceso a determinados *topics*. A partir de este sistema de privilegios, hemos visto cómo el uso de sistemas de cachés permite reducir el estrés general del sistema, así como descongestionar los canales de comunicación. La elección de un sistema de registro abierto para máquinas ha supuesto tener en cuenta distintos aspectos de seguridad como la posibilidad de que las credenciales sean interceptadas, aunque tampoco hemos podido analizar absolutamente todos debido a las limitaciones de espacio y tiempo (como posibles ataques de fuerza bruta).

Adicionalmente hemos planteado y especificado una serie de protocolos que indican exactamente cómo se deben llevar a cabo determinados procedimientos, desde actualizar los distintos dispositivos de manera remota, hasta el proceso de vinculación que permite crear la relación entre un usuario y un dispositivo, facilitando así el proceso de implementación de una API para sistemas más específicos, como por ejemplo en entornos como *ESP-IDF*[4]. De hecho, para la implementación de este sistema se ha tenido muy en cuenta la documentación *API HAL*[5] [9] proporcionada por el fabricante de *Espressif* de estos SoCs y así poder garantizar cierta adaptabilidad con nuestro planteamiento.

Por otro lado, también hemos visto cómo se produce ese intercambio de mensajes entre los dispositivos y los proveedores de servicio a través de una determinada estructura jerárquica de *topics* acordada entre ambos frentes, que permite realizar una comunicación de dos vías sobre un protocolo de comunicación *One-to-Many* como es MQTT. También hemos elaborado una serie de envoltorios que permiten abstraer la comunicación entre los distintos procesos, lo cual permite facilitar el proceso de notificación de eventos a los distintos servicios, como, por ejemplo, a la hora de lanzar una actualización OTA.

El código implementado actualmente no se encuentra accesible públicamente, pero es probable que en un futuro no muy lejano lo esté, no obstante, estoy abierto a colaboraciones, en cuyo caso el siguiente enlace https://gitlab.lromeraj.net/iot-core contiene los distintos repositorios con la información al respecto.

[4] Official development framework for the ESP32, ESP32-S and ESP32-C Series SoCs

[5] Hardware Abstraction Layer - Application Programming Interface

6.2 Trabajo futuro

Sigue existiendo la posibilidad de continuar trabajando en este núcleo, y no solo añadiendo mejoras y nuevas funcionalidades, como por ejemplo podría ser la inclusión de un sistema de persistencia que permitiese gestionar distintos parámetros de configuración de los dispositivos utilizando otros gestores de bases de datos no relacionales como *Mongo DB*, o incluso podríamos mejorar la distribución de la carga entre procesos dando la posibilidad de que se pudiesen crear múltiples instancias de un proveedor de servicio en específico jugando con una estructura de *topics* adicional que permitiera redirigir mensajes a otros niveles para evitar los correspondientes problemas de concurrencia que surgieran.

Tampoco nos podemos olvidar de un aspecto muy importante que está relacionado con la seguridad, el cual consiste en los distintos ataques de fuerza bruta a los que nos podemos enfrentar, pudiendo añadir sistemas de control adicionales que sean capaces de detectar este tipo de ataques y evitar, por ejemplo, la generación arbitraria de credenciales.

A pesar de las mejoras que podamos añadir sobre este pequeño núcleo, no se nos debe olvidar que falta por ver la parte más interesante a nivel práctico que consiste en conectar los distintos dispositivos y comenzar a trabajar con la información que estos pudieran obtener y al mismo tiempo hacer los respectivos análisis y tratamientos sobre los datos generados, como podría ser, por ejemplo, una pequeña red de estaciones meteorológicas distribuidas geográficamente.

En el ámbito del *IoT* tratamos con la cooperación y comunicación de diferentes tecnologías y dispositivos que implican la adquisición de conocimientos varios sobre distintos entornos de desarrollo, pasando del abstracto mundo del *software* al mundo del *hardware*, de tal modo que resulta necesaria la cooperación de diferentes ingenierías que perfeccionen el sistema en los campos correspondientes, por lo que como ya he dicho, este proyecto está abierto a todo tipo de colaboraciones, y poder así construir un núcleo más robusto y completo (fomentando la mentalidad *DIY* [6] de *"Hazlo tú mismo"*).

[6] Do It Yourself

Referencias

[1] Porro Saez, 2021. IoT: protocolos de comunicación, ataques y recomendaciones. [online] INCIBE-CERT. Available at: <https://www.incibe-cert.es/blog/iot-protocolos-comunicacion-ataques-y-recomendaciones> [Accessed 17 April 2022]

[2] Amazon Web Services, Inc. AWS IoT Core. [online] Available at: <https://aws.amazon.com/es/iot-core/> [Accessed 12 February 2022]

[3] Knexjs.org. 2022. SQL Query Builder for Javascript | Knex.js. [online] Available at: <http://knexjs.org/> [Accessed 26 May 2022]

[4] GitHub. 2022. GitHub - moscajs/aedes: Barebone MQTT broker that can run on any stream server, the node way. [online] Available at: <https://github.com/moscajs/aedes> [Accessed 14 May 2022]

[5] Docs.oasis-open.org. 2015. MQTT Version 3.1.1. [online] Available at: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html> [Accessed 17 April 2022]

[6] GitHub. 2022. GitHub - RIAEvangelist/node-ipc: Inter Process Communication Module for node supporting Unix sockets, TCP, TLS, and UDP. Neural Networking in Node.JS. [online] Available at: <https://github.com/RIAEvangelist/node-ipc> [Accessed 5 May 2022]

[7] GitHub. 2022. GitHub - uuidjs/uuid: Generate RFC-compliant UUIDs in JavaScript. [online] Available at: <https://github.com/uuidjs/uuid> [Accessed 18 March 2022]

[8] GitHub. 2022. GitHub - sindresorhus/crypto-random-string: Generate a cryptographically strong random string. [online] Available at: <https://github.com/sindresorhus/crypto-random-string> [Accessed 26 February 2022]

[9] Docs.espressif.com. 2022. Hardware Abstraction - ESP32 — ESP-IDF Programming Guide latest documentation. [online] Available at: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/hardware-abstraction.html> [Accessed 2 June 2022]

Glosario

API	Application Programming Interface
Broker	Representa al servidor que gestiona el flujo de mensajes publicados por clientes y los difunde entre los distintos suscriptores
On-premise	Hace referencia al software que se instala y se ejecuta en las instalaciones *hardware* de la persona u organización que usa el software
OTA	Over The Air – Hace referencia al proceso por el cual se pueden realizar modificaciones en el firmware de un determinado sistema de manera remota a través de una red de comunicación como puede ser Internet
Servomotor	Es un dispositivo similar a un motor de corriente continua que tiene la capacidad de ubicarse en cualquier posición dentro de su rango de operación, y mantenerse estable en dicha posición
SoC	System on Chip – Es un circuito integrado que incorpora en un mismo chip distintos módulos adicionales como sistemas de memoria, interfaces de entrada y salida (GPIO), entre otros.
Topic	Secuencia de caracteres que permite identificar (en el protocolo MQTT) una dirección a través de la cual los clientes pueden intercambiar mensajes
TLS	Transport Layer Security – Protocolo criptográfico para proporcionar seguridad en la comunicación a través de una red, que, por lo general, suele ser Internet

ANEXOS

A

Anexo A. Diagramas de secuencia

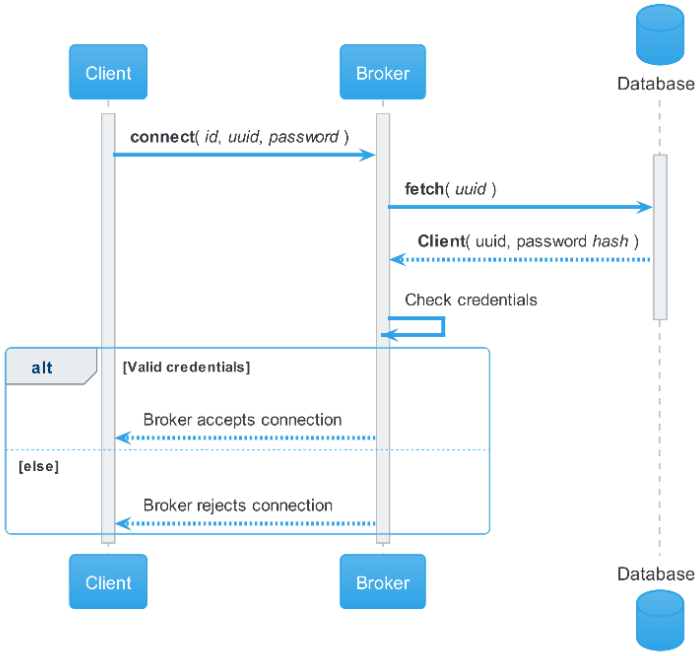

Figura A.1. Secuencia de identificación de dispositivos

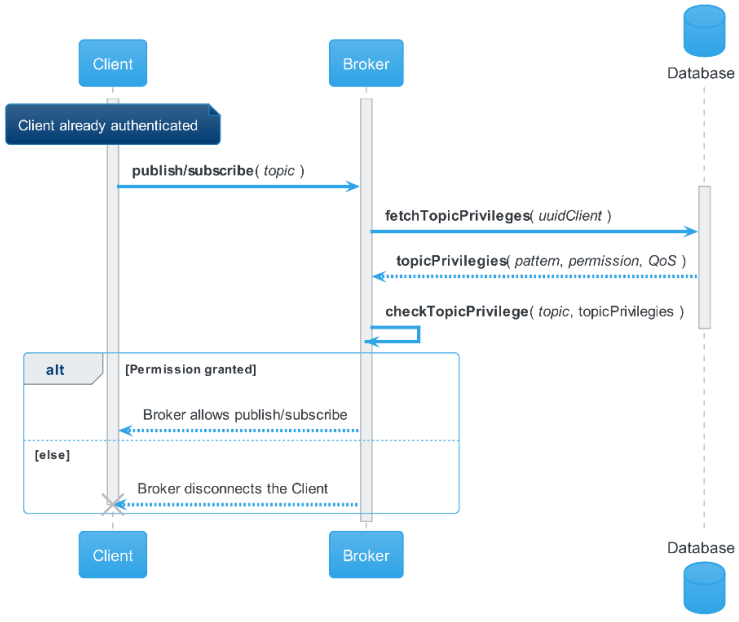

Figura A.2: Secuencia de autorización de *topics*

B

Anexo B. Fragmentos de código

```
WITH client
     AS (SELECT id
         FROM   mqtt_client
         WHERE  uuid = $1),
     topics
     AS (SELECT t.*
         FROM   client
                JOIN mqtt_client_topic AS ct ON ct.client_id = client.id
                JOIN mqtt_topic AS t ON t.id = ct.topic_id
         UNION
         SELECT t.*
         FROM   client
                JOIN mqtt_client_role AS cr ON cr.client_id = client.id
                JOIN mqtt_role_topic AS rt ON rt.role_id = cr.role_id
                JOIN mqtt_topic AS t ON t.id = rt.topic_id)
SELECT pattern,
       wildcard
FROM   topics   WHERE strpos( permission, $2 ) > 0 AND $3 <= max_qos
```

Código B.1: Consulta SQL de privilegios.

```
export class SplittedTopic extends Array<string> {
  splits = 0;
  usesWildcards = false;
  private levelWildcards:boolean[] = [];
  constructor( topic: string ) {
    super();
    let usingWildcard = false;
    const _appendLevel = (starti:number, endi:number) => {
      this[ this.splits ] = topic.substring( starti, endi );
      this.levelWildcards[ this.splits ] = usingWildcard;
    }
    let i = 0, oldi = 0;
    for ( ; i < topic.length; i++ ) {
      let c = topic[ i ];
      if ( c === '/' )  {
        _appendLevel( oldi, i );
        oldi = i+1;
        this.splits++;
        usingWildcard = false;
      } else {
        if ( c === '#' || c === '+' ) {
          usingWildcard = true;
          this.usesWildcards = true;
        }
      }
    }
    _appendLevel( oldi, i );
  }
  levelUsingWildcard( levelIndex:number ) {
    return (this.levelWildcards[ levelIndex ] || false); }
}
```

Código B.2: Clase *SplittedTopic*.

```
export function splittedTopicPatternTest(
  splittedTopic:SplittedTopic, topicPattern:string, allowWildcards:boolean=true
): boolean {
  let split = 0;
  let wordIndex = 0;
  let level:string|null = splittedTopic[ 0 ];
  if ( !allowWildcards && splittedTopic.usesWildcards ) return false;
  for (let i=0; i < topicPattern.length; i++ ) {
    let pc = topicPattern[ i ];
    if ( pc === '/' ) {
      split++;
      if ( split > splittedTopic.splits ) return false;
      wordIndex = 0;
      level = splittedTopic[ split ];
    } else if ( pc === '#' && i === topicPattern.length-1 ) {
      return true;
    } else if ( pc === '*' ) {
      level = null;
    } else if ( pc === '+' ) {
      if ( splittedTopic.levelUsingWildcard( split ) ) return false;
      level = null;
    } else if ( level !== null && pc !== level[ wordIndex++ ] ) {
      return false;
    }
  }
  if ( level !== null && level.length !== wordIndex ) return false;
  return (splittedTopic.splits === split);
}
```

Código B.3: Analizador de plantillas de *topics*.

```
function UTIRevoke( done:jest.DoneCallback ) {
  const commonOpts:mqtt.IClientOptions = {
    resubscribe: false,
    reconnectPeriod: 0,
    clientId: `${
      cryptoRandomString({length: 32})
    }`
  };

  legitOpts = { ...MQTTOptions, ...commonOpts };
  intruderOpts = { ...MQTTOptions, ...commonOpts };

  let intruder = mqtt.connect( intruderOpts ).end( true );
  let legitimate = mqtt.connect( legitOpts );

  let _done = ( msg?:string ) => {
    if ( msg ) done( msg ); else done();
  }

  let reconnectAttemps = 1000;
  let intruderTimesConnected = 0;
  let legitimateTimesConnected = 0;
  let intruderCred:string[] = [];
  let legitimateCred:string[] = [];

  const _checkCredentials = () => {
    intruder.removeAllListeners();
    legitimate.removeAllListeners();
    intruder.end( true );
    legitimate.end( true );
    legitimateCred = Array.from(new Set( legitimateCred ));
    intruderCred = Array.from(new Set( intruderCred ));

    let credentials = [ ...legitimateCred, ...intruderCred ].sort();
```

```
    for (let i=0; i < credentials.length-1; i++ ) {
      if ( credentials[ i ] === credentials[ i+1 ] )  {
        return done("Credentials have been compromised");
      }
    }
    done()
  };

  const _fetchFinished = ():boolean => {
    return (legitimateTimesConnected >= reconnectAttemps );
  }

  legitimate.on("connect", () => {
    legitimateTimesConnected++;

    setTimeout( () => {
      intruderOpts.clientId = legitOpts.clientId;
      intruder.reconnect();
    }, Math.round( Math.random() *100 ) );

    legitimate.subscribe(
      `api/${legitOpts.clientId}/${ServiceProvider.Act.CRED}/#`);

    legitimate.publish(`api/new`, legitOpts.clientId!, () => { });

  })

  legitimate.on("close", () => {
    legitOpts.clientId = cryptoRandomString({length: 32});
  })

  legitimate.on("message", (topic, payload) => {
    legitimateCred.push( payload.toString() );
  });

  intruder.on("connect", () => {
    intruderTimesConnected++;

    intruder.subscribe(
      `api/${intruderOpts.clientId}/${ServiceProvider.Act.CRED}/#`);

    if ( _fetchFinished() ) {
      _checkCredentials();
    } else {
      setTimeout(() => {
        intruder.end( true );
        legitimate.reconnect();
      }, 100 );
    }

  });

  intruder.on("message", (topic, payload) => {
    intruderCred.push( payload.toString() );
  })

}
```

Código B.4: Script malicioso de suplantación de UTI.

```
export function checkMQTTClientTopicPrivilege(
  uuidClient:string,
  topic:string,
  qos:number,
  privilege:MQTTTopicPrivilege
): Promise<boolean> {
  return new Promise( ( resolve, reject ) => {
    qb.with( 'client', sqb => {
      sqb.select('id').from('mqtt_client').where('uuid','=', uuidClient)
    }).with( 'topics', sqb => {
      sqb.union([
        qb.select('t.*').from('client')
          .join('mqtt_client_topic as ct', 'ct.client_id', 'client.id')
          .join('mqtt_topic as t', 't.id', 'ct.topic_id')
        ,
        qb.select('t.*').from('client')
          .join('mqtt_client_role as cr', 'cr.client_id', 'client.id')
          .join('mqtt_role_topic as rt', 'rt.role_id', 'cr.role_id')
          .join('mqtt_topic as t', 't.id', 'rt.topic_id')
      ])
    })
    .select(['pattern','wildcard'])
    .from('topics')
    .whereRaw('strpos( permission, ? ) > 0', [privilege])
    .andWhere(qos, '<=', 'max_qos').then( data => {
      let splittedTopic = new SplittedTopic( topic );
      let grant = data.some( it =>
       splittedTopicPatternTest( splittedTopic, it.pattern, it.wildcard ) );
      resolve( grant );
    }).catch( reject )
  })
}
```

Código B.5: Consulta de privilegios con *Knex*.

C

Anexo C. Manual de configuración

A continuación, se adjunta un pequeño manual que indica cómo configurar inicialmente las distintas bases de datos, también se detallarán los distintos parámetros para la configuración del entorno. Esta guía da por sentado que a la hora de instalar los distintos paquetes y dependencias se está utilizando un sistema operativo basado en *Debian*.

C.1 Dependencias

Es necesario tener instaladas las siguientes dependencias para poder configurar el sistema correctamente:

- *Node JS* junto al gestor de paquetes *npm*. Consulte el siguiente enlace (https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-20-04-es) para instalar NodeJS, asegúrese de instalar una versión mayor igual a *v14.x.x*
- *Mongo DB* consulte el siguiente enlace (https://www.mongodb.com/docs/manual/installation/) para su instalación.
- *PostgreSQL ≥ 11* puede obtener la última versión disposible ejecutando:
  ```
  sudo apt install postgresql postgresql-contrib
  ```

C.2 Configuración de los gestores de las bases de datos

En un principio no es necesario ningún requisito especial a la hora de configurar las bases de datos, se puede configurar como uno considere oportuno, pero se recomienda crear un rol específico para este sistema en cada uno de los gestores. Existe un fichero específico de variables de entorno que permitirá posteriormente especificar las credenciales de acceso a las bases de datos y otro tipo de parámetros, en la sección C.3 se detalla todo lo relacionado con este fichero.

C.2.1 Configuración de PostgreSQL

1.-- Para configurar inicialmente este gestor, debemos entrar en la consola en modo administrador usando el comando: `sudo -u postgres psql`

2.-- A continuación, se deben ejecutar una serie de *queries* para crear el usuario que tendrá permiso en las distintas bases de datos.

3.-- Creamos la base de datos:
```
create database "iot_dev";
```

4.-- Creamos el usuario que nos permitirá tener control sobre dicha base de datos:
```
create user "iotman" with encrypted password '<password>';
```

5.-- Añadiremos los privilegios correspondientes al usuario creado anteriormente:
```
grant all privileges on database "iot_dev" to "iotman";
```

6.-- Conectarse de nuevo utilizando el nuevo usuario: `psql -U iotman -d iot_dev -W`

7.-- Es posible que tras intentar iniciar sesión nos encontremos con el siguiente error:
`psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL:  Peer authentication failed for user "iotman"`, en cuyo caso debemos modificar el fichero `/etc/postgresql/<version>/main/pg_hba.conf` y añadir la siguiente línea:

```
local   iot_dev             iotman                                  md5
```

8.-- Reiniciar el servicio del gestor usando: `sudo service postgresql restart`

9.-- Ahora solo hay que probar que el acceso a la base de datos funciona correctamente ejecutando el comando mencionado en el paso 6.

C.2.2 Configuración de MongoDB

La implementación del protocolo MQTT que hemos elegido (*Aedes*) permite ejecutarse en modo cluster, y para esto es necesario configurar inicialmente una base de datos de Mongo como se indica a continuación, estos pasos son necesarios si es la primera vez que configura el gestor de Mongo:

1.-- Entrar en la terminal de Mongo ejecutando: `mongosh`

2.-- Seleccionamos la base de datos de administración: `use admin`

3.-- Creamos el rol de administrador con el siguiente fragmento:

```
db.createUser(
  {
    user: "root",
    pwd: passwordPrompt(),
    roles: [
      { role: "userAdminAnyDatabase", db: "admin" },
      { role: "readWriteAnyDatabase", db: "admin" }
    ]
  }
)
```

Ahora ya es posible configurar el usuario de acceso para la base de datos que utiliza Aedes para gestionar el tráfico de mensajes:

1.-- Nos identificamos con las credenciales anteriormente utilizadas:
`mongosh --authenticationDatabase "admin" -u "root" -p`

2.-- Ejecutamos el siguiente comando en el terminal de Mongo: `use aedes-cluster`

3.-- Finalmente creamos el usuario que permitirá conectarse con la base de datos del *broker*:

```
db.createUser(
  {
    user: "aedes",
    pwd: passwordPrompt(),
    roles: [
      { role: "dbOwner", db: "aedes-cluster" },
    ]
  }
)
```

A partir de ahora, ya podemos centrarnos en la configuración de las distintas variables de entorno, las cuales especificaremos en la sección C.3.

C.3 Configuración del fichero de variables de entorno

Detallaremos a continuación los distintos parámetros de configuración que permiten al programador personalizar el entorno de ejecución de este núcleo. Por ahora existen dos posibles entornos, el de desarrollo y el de producción, para ello, deben existir dos ficheros de variables de entorno, un fichero denominado `.env` para el entorno de producción y otro fichero `.envdev` para el entorno de desarrollo. Para intercambiar entre un entorno u otro es tan sencillo como crear un fichero vacío o eliminarlo del directorio raíz, este fichero debe llamarse: `.dev` .

Campo	**Descripción**	**Predeterminado**
HTTP_PORT	Puerto de escucha del servidor HTTP	8080

Tabla C.1: Parámetros de configuración para el servidor HTTP.

Campo	**Descripción**	**Predeterminado**
MQTT_PORT	Puerto de escucha del *broker* MQTT para clientes	8883
MQTT_CONTROL_PORT	Puerto de escucha del *broker* MQTT para los proveedores de servicio	8884
MQTT_CONTROL_SSL_PORT	Puerto de escucha (seguro) del *broker* MQTT para los proveedores de servicio. Útil cuando el proveedor se encuentra fuera de la LAN del *broker*. Si el valor del puerto es nulo (0) no se escuchará en el puerto	0
MQTT_HOST_NAME	Nombre de dominio del *broker* MQTT para los servidores de control	localhost
AEDES_DB_HOST	Nombre del dominio del gestor de la base de datos	localhost
AEDES_DB_PORT	Puerto de acceso al gestor de la base de datos	27017
AEDES_DB_USER	Nombre del usuario de acceso	-
AEDES_DB_PSW	Contraseña del usuario de acceso	-
AEDES_DB_NAME	Nombre de la base de datos para la administración del *broker* MQTT	-
AEDES_BROKERS	Número de instancias paralelas que se lanzan del *broker*, usar el valor (0) para que se realice un ajuste automático	0
L1_CACHE_ENTRIES	Número máximo de entradas en la caché de primer nivel	10
L2_CACHE_ENTRIES	Número máximo de entradas en la caché de segundo nivel	20

Tabla C.2: Parámetros de configuración del *broker* MQTT.

Campo	Descripción	Predeterminado
DB_MANAGER	Nombre del gestor de la base de datos	pg
DB_USER	Nombre de usuario de acceso al gestor de la base de datos	-
DB_PSW	Contraseña del usuario de acceso al gestor	-
DB_HOST	Nombre de dominio del gestor	localhost
DB_PORT	Puerto de acceso al gestor	5432
DB_NAME	Nombre de la base de datos	-

Tabla C.3: Parámetros de configuración del gestor de la base de datos.

Campo	Descripción	Predeterminado
BCRYPT_ROUNDS	Número de rondas utilizadas en la generación del hash de contraseña	10

Tabla C.4: Parámetros de configuración generales.

Campo	Descripción	Predeterminado
CA_CERT	Ruta al fichero que contiene el certificado de la autoridad certificadora	./certs/CA/ca.pem
CTRL_SERVER_CERT	Ruta al fichero que contiene el certificado del servidor firmado por la autoridad certificadora	./certs/controlServer.pem
CTRL_SERVER_KEY	Ruta al fichero que contiene la clave privada del servidor	./certs/controlServer.key
MQTT_BROKER_CERT	Ruta al fichero que contiene el certificado del *broker* firmado por la autoridad certificadora	./certs/mqttBroker.pem
MQTT_BROKER_KEY	Ruta al fichero que contiene la clave privada del *broker*	./certs/mqttBroker.key

Tabla C.5: Configuración de certificados.