Universidad Zaragoza
1542

Trabajo Fin de Máster

Diseño e implementación de un *middleware* CoAP-MQTT-HTTP para la mejora de la interoperabilidad de los protocolos de aplicación en redes IoT

Design and implementation of a CoAP-MQTT-HTTP middleware to improve the interoperability of application protocols in IoT networks

Autor/es

Asier Carbonel Martínez

Director/es

José Ramón Gallego Martínez
Rafael Tolosana Calasanz

Escuela de Ingeniería y Arquitectura (EINA) / Universidad de Zaragoza

2020

Resumen

El rápido incremento de dispositivos IoT *(Internet of Things)* está permitiendo la aparición de nuevas aplicaciones computacionales que pueden tener un sustancial impacto en la sociedad, pero que al mismo tiempo abre un número significativo de posibilidades de negocio para las empresas. No obstante, la rapidez de ese desarrollo en el IoT ha incentivado la aparición de protocolos de comunicación muy diferentes entre sí, como MQTT, MQTT-SN, HTTP REST o CoAP. La diferencia ya no está únicamente en el formato de los mensajes del protocolo (lo que se conoce como protocolo de comunicación), sino en el protocolo de interacción, esto es, en el número de mensajes que los procesos tienen que intercambiarse para realizar la comunicación. Por ejemplo, CoAP y HTTP funcionan bajo el paradigma *REQUEST/RESPONSE* (un mensaje de petición y un mensaje de respuesta); mientras que MQTT y MQTT-SN se comunican a través del paradigma *PUBLISH/SUBSCRIBE*, mucho más sofisticado (un proceso se registra (*subscribe*) para que le lleguen mensajes en cuanto haya una actualización (*publish*)). Todos estos aspectos hacen que la intercomunicación entre protocolos no sea directa y suponen una enorme barrera tecnológica para las pequeñas y medianas empresas europeas.

En este contexto, y dada la escasa oferta de este tipo de soluciones, en este proyecto se estudia, se propone y se implementa un *middleware* que permite comunicar de forma transparente dispositivos IoT basados en protocolos IoT heterogéneos. Aunque la propuesta está centrada en los protocolos más habituales en este contexto, como son MQTT, MQTT-SN, CoAP y HTTP, otros protocolos podrían también integrarse de forma análoga. Dada la vital importancia de garantizar comunicaciones seguras, el *middleware* propuesto permite además la transferencia de información a través de canales con cifrado, mediante mecanismos como DTLS sobre UDP y TLS sobre TCP. Por último, la arquitectura del *middleware* se ha diseñado para que sea escalable con el número de dispositivos IoT conectados. Para ello, múltiples instancias del *middleware* se disponen en máquinas diferentes, y se comunican entre sí directamente, reduciendo la carga de trabajo y permitiendo la interoperabilidad de los datos.

Para validar la aproximación, se realizan diversos análisis de rendimiento del *middleware* en diferentes escenarios propuestos, estudiando su rendimiento en términos de retado, diferenciando entre dispositivos sin limitación de recursos (en el centro de datos) y dispositivos de recursos limitados (fuera del centro de datos, *edge computing*).

Agradecimientos

En primer lugar, agradezco a mi madre el apoyo transmitido en todo momento, así como todo el ánimo recibido durante la realización de este proyecto.

En segundo lugar, dar las gracias a todos los profesores que he tenido durante estos años por su buen hacer docente. Especialmente agradezco el continuo trabajo, esfuerzo y afán de ayudar durante la realización de este proyecto de mis tutores José Ramón Gallego Martínez y Rafael Tolosana Calasanz.

Por último, agradezco a mis amigos haber estado en todo momento a mi lado durante mis años de estudio y sobre todo en esta última etapa como estudiante.

Tabla de contenido

Tabla de figuras

Tabla de tablas

Lista de acrónimos

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
DDS	Data Distribution Service
DTLS	Datagram Transport Layer Security
ETSI	Instituto Europeo de Normas de Telecomunicaciones
E2E	End to End
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
MQTT-SN	Message Queuing Telemetry Transport for Sensor Networks
M2M	Machine to Machine
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RFC	Request for Comments
RTT	Round Trip Time
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WSN	Wireless Sensor Networks
XMPP	Extensible Messaging and Presence Protocol

1. Introducción

Internet de las cosas *(Internet of Things, IoT)* es una tecnología diseñada para comunicar miles de dispositivos a través de internet, permitiendo enviar y recibir información, así como realizar diferentes acciones en función de dicha información. Esta tecnología emergente, cada vez toma más importancia en el campo de la recopilación de datos y toma de decisiones en tiempo real, ejemplos de ello son: la implantación de extensas redes de sensores *(Wireless Sensor Network , WSN)* o las cada vez más comunes comunicaciones máquina a máquina *(Machine to Machine, M2M)* planteadas para el desarrollo de ambiciosos proyectos como ciudades inteligentes *(Smart Cities)*, la evolución de la Industria 4.0, coche autónomo o simplemente la recopilación de grandes volúmenes de datos *(Big Data)*. Todo esto ha impulsado extensos estudios sobre los protocolos de nivel de aplicación propuestos para las comunicaciones IoT, como por ejemplo [11].

Tras años de estudios y de evolución en los protocolos de nivel de aplicación, se conocen las ventajas e inconvenientes de cada una de las alternativas existentes, mostrando cada protocolo fortalezas y debilidades muy diversas en función de diferentes factores como pueden ser recursos *hardware* disponibles, ancho de banda requerido, latencia extremo a extremo, o tolerancia a fallos. A pesar de esto, actualmente no se dispone de un estándar que defina el protocolo a emplear en las comunicaciones IoT, lo que ha generado el despliegue de numerosas infraestructuras IoT trabajando bajo diferentes protocolos de nivel de aplicación, formando así un ecosistema altamente fragmentado y heterogéneo.

El hecho de tener una gran heterogeneidad entre protocolos de aplicación da como resultado la necesidad de disponer de diversas aplicaciones o servicios trabajando bajo diferentes protocolos para poder publicar o recopilar toda la información que se necesita, además de adaptar los diferentes procesos de extracción, recopilación y unificación de datos. Desde un punto de vista económico, este aspecto representa una importante barrera para las pequeñas y medianas empresas que quieren desplegar negocios en este importante sector emergente. Por el contrario, las grandes multinacionales tecnológicas no europeas parten de una importante ventaja, puesto que al contar con gran cantidad de recursos (económicos y humanos), pueden desarrollar soluciones verticales que solucionan estos problemas de heterogeneidad.

En este contexto, durante este proyecto se analizan las diferentes casuísticas a las que se enfrentan los diferentes protocolos de nivel de aplicación, se estudian las relaciones entre ellos y se desarrolla un *middleware* que haga las veces de *broker*[1] de mensajes como pasarela entre clientes/servidores trabajando con diferentes protocolos de nivel de aplicación. Así pues, se implementa una solución *software* que permite intercomunicar de manera transparente los protocolos MQTT *(Message Queuing Telemetry Transport)*[38], CoAP *(Constrained Application Protocol)*[52]y HTTP *(Hypertext Transfer Protocol)*[17]. Se han elegido estos protocolos, puesto que como muestra la

[1] *Broker* de mensajes: agente intermediario de transferencia de mensajes, empleado para intercambiar mensajes entre diferentes aplicaciones emisoras y receptoras.

Figura 1, se tratan de los protocolos de nivel de aplicación más utilizados en aplicaciones IoT en la actualidad.

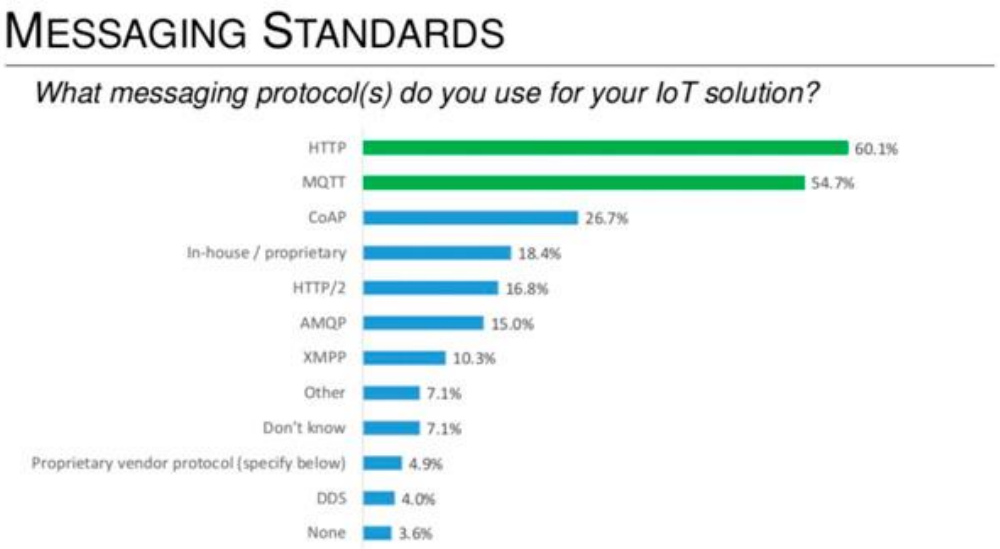


Figura 1. Protocolos IoT más utilizados según [54]

Por último, se analizan las consecuencias del uso de este *middleware* en términos de aumento de retardo, valorando así una solución orientada a la computación en la nube *(cloud computing)* y una solución orientada a la computación de borde *(edge computing)* mediante equipos de recursos limitados.

Este documento está organizado de la siguiente manera. El Capítulo 2 presenta una revisión de los principales protocolos de nivel de aplicación más utilizados actualmente, así como diferentes soluciones de interoperabilidad existentes. En el Capítulo 3 se muestran diversas implementaciones *software* disponibles para utilizar en entornos IoT y se definen las que se emplearan en la solución técnica. En el Capítulo 4 se plantean los casos de uso que debe resolver el *middleware*, así como su arquitectura y funcionamiento. En el Capítulo 5 se definen los escenarios en los que se va a probar el sistema y en el Capítulo 6 se muestran los resultados obtenidos. Por último, en el Capítulo 7 se exponen las conclusiones y las líneas futuras de trabajo.

1.1. Objetivos y metodología

1.1.1. Objetivos

El **objetivo principal** del proyecto es estudiar y analizar la interoperabilidad entre los principales protocolos de nivel de aplicación para IoT, MQTT, CoAP y HTTP. A partir de ahí, se propone un sistema *middleware* que permite una comunicación transparente entre los diferentes protocolos de red.

Tomando como referencia este objetivo general, se han establecido los siguientes **objetivos parciales:**

1. Estudiar el estado del arte de los protocolos de nivel de aplicación.
2. Familiarización con diferentes implementaciones *software.*
3. Desarrollo *software* de la solución técnica junto con continuas verificaciones de su funcionamiento.
4. Despliegue, estudio y análisis de resultados en un escenario IoT basado en *cloud computing.*
5. Despliegue, estudio y análisis de resultados en un escenario IoT basado en *edge computing.*

1.1.2. Metodología

A continuación, se plantea la metodología seguida para la elaboración del trabajo, mostrándose las diferentes etapas por las que ha pasado el proyecto.

1. En primer lugar, se establece la necesidad de estudiar el **estado del arte** relacionado con los principales protocolos de nivel de aplicación HTTP, MQTT, MQTT-SN y CoAP, así como en su interoperabilidad y las principales soluciones *software* existentes en las que se profundizará para desarrollar la solución técnica.

2. Identificadas las características de los diferentes protocolos y las **implementaciones *software* disponibles,** se realiza un estudio sobre las diferentes implementaciones con el fin de familiarizarse con ellas para su posterior uso en el desarrollo del *middleware* propuesto.

3. La siguiente etapa del proyecto, se centra en el **desarrollo técnico del sistema *middleware*** en el que se basa el proyecto. La solución se implementa sobre el lenguaje de programación Java utilizando la herramienta Maven, que permite integrar las diferentes librerías existentes de las que se ha partido y sobre las que se han añadido las funcionalidades necesarias.

4. A partir de la solución técnica, se propone un **primer escenario de aplicación** basado en computación en la nube mediante el cual se estudia el rendimiento del *middleware* en un dispositivo sin limitación de recursos para la aplicación desarrollada. Para ello, se despliegan diversos clientes generando y recibiendo tráfico a través del *middleware.* A partir del tráfico

generado se mide tanto el retardo extremo a extremo como el tiempo de procesado aislado en el *middleware.*

5. Con el fin de contrastar los resultados obtenidos en el primer escenario de uso, se propone un **segundo escenario de aplicación** basado en computación de borde de red, en el que se estudia el rendimiento del *middleware* en un dispositivo de recursos limitados comparándolo con el primer escenario. De forma análoga al proceso seguido en el primer escenario, se despliegan diversos clientes generando y recibiendo tráfico a través del *middleware,* a partir del tráfico generado se mide tanto el retardo extremo a extremo como el tiempo de procesado aislado en el *middleware.*

1.1.3. Herramientas y recursos necesarios

En primer lugar, en cuanto a los recursos *hardware* se dispone de:

- Dispositivo sin limitación de recursos para la aplicación desarrollada: una estación de trabajo Workstation Intel Xeon SkyLake-SP 3106 dual con 480GB de disco SSD, 8TB de disco duro y 128 GB de memoria RAM. En él se implanta el *middleware* en el primer escenario de aplicación.
- Dispositivos de recursos limitados: *Raspberry Pi 3 model B,* cuenta con un procesador *Quad Core* 1.2GHz *y* 1GB de memoria RAM. En él se despliegan las aplicaciones IoT y el *middleware* en el segundo escenario.

En segundo lugar, en cuando a los recursos *software* necesarios:

- Conocimientos sobre Maven [3], herramienta de gestión y construcción de proyectos Java. Es de gran utilidad a la hora de crear y compilar proyectos Java, también permite añadir y gestionar dependencias de forma sencilla.
- Conocimientos en lenguajes de programación como Java o C que nos permiten analizar, emplear y modificar librerías de código abierto existentes para la creación y análisis del *middleware* propuesto.

Por último, la Figura 2, muestra la evolución temporal que se ha seguido en la consecución de los objetivos parciales del proyecto.

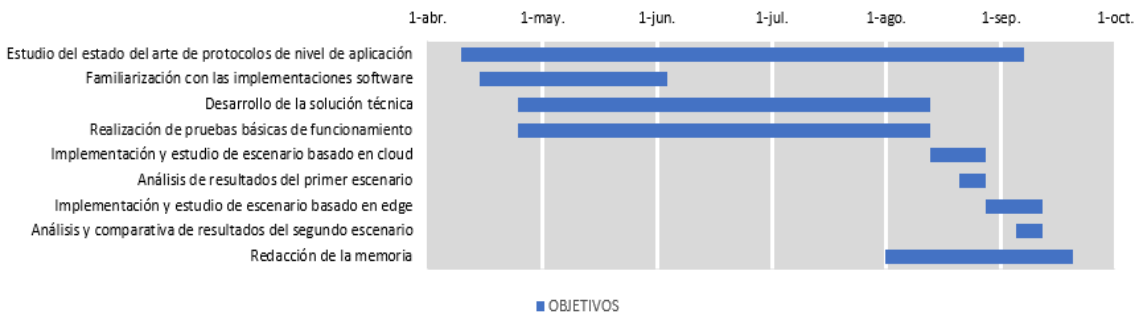


Figura 2. Diagrama de Gantt del proyecto

2. Estado del arte

A lo largo de este capítulo se exponen los fundamentos teóricos de los protocolos de nivel de aplicación empleados en redes IoT, los cuales se utilizarán como parte de la solución final. También se exponen las actuales soluciones que otorgan interoperabilidad en las comunicaciones.

2.1. Paradigmas de comunicación IoT

En este apartado se detallan brevemente los dos paradigmas de transferencia de mensajes en los que se basan los protocolos de nivel de aplicación IoT que se estudian en capítulos posteriores.

2.1.1. Paradigma Publicación / Suscripción

El paradigma publicación/suscripción o más conocido como *publish/subscribe* en inglés, se trata de un modelo de envío de mensajes asíncrono en el cual los mensajes no se envían directamente entre clientes finales sino a través de una infraestructura intermedia comúnmente llamada *broker*.

En la transferencia de información *publish/subscribe* intervienen los siguientes actores:

- Publicador *(Publisher):* cliente encargado de generar la información sobre un tema y publicarla en la infraestructura.
- Suscriptor *(Subscriber):* cliente interesado en recibir información sobre un tema. Este se suscribe a los temas de interés en la infraestructura y esta le notifica cada vez que recibe información sobre dichos temas.
- Infraestructura (*broker)*: situado entre el *publisher* y el *subscriber*. Se encarga de recibir las peticiones de suscripción de los clientes suscriptores, las publicaciones de información de los clientes publicadores y a su vez de retransmitirlas a los clientes suscritos a esos temas.

Se basa en una estrategia de transferencia de información PUSH (o de recepción pasiva) en la que el *broker* notifica de forma proactiva a los clientes suscritos cada vez que recibe información, es por ello por lo que también se conoce como paradigma “uno a muchos” *(one-to-many)* en el cual, los clientes no necesitan conocerse entre sí, únicamente deben comunicarse con el *broker*. En la Figura 3 se muestra un ejemplo de comunicación *publish/subscribe.*

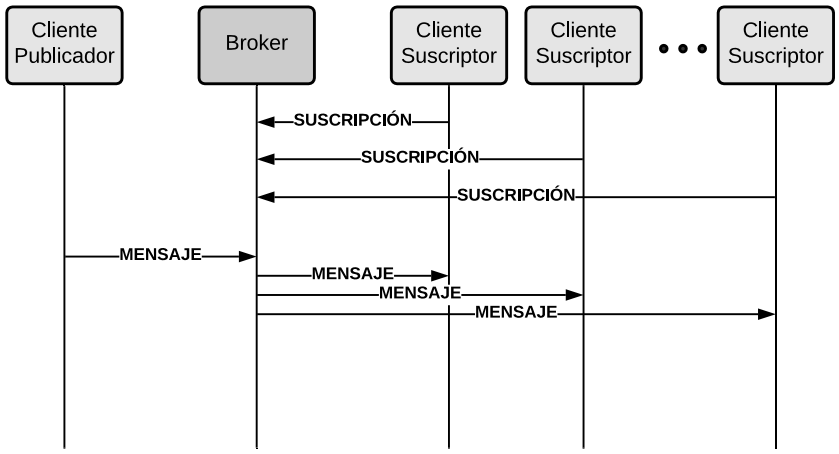


Figura 3. Comunicación modelo Publicación/Subscripción

2.1.2. Paradigma Petición / Respuesta

El paradigma petición/respuesta o también conocido como *request/response* en inglés, se trata de un modelo de mensajes comúnmente utilizado en arquitecturas de red cliente-servidor o *REST (Representational State Transfer)*.

En este modelo de comunicación intervienen los siguientes actores:

- Servidor: aplicación capaz de atender peticiones de diferentes clientes proporcionando como respuesta a dicha petición la información o el servicio requerido.
- Cliente: se trata del consumidor de los datos. Este se encarga de realizar peticiones al servidor con el fin de obtener información o servicios como respuesta.

A diferencia del paradigma *publish/subscribe*, este se basa en una transferencia de información de tipo PULL donde el cliente genera una petición al servidor y este la contesta de forma reactiva. Por esta razón se considera un método "uno a uno". En la Figura 4 se muestra un ejemplo de comunicación petición/respuesta.

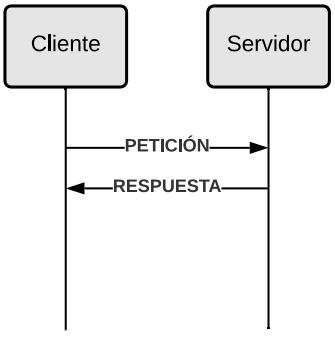


Figura 4. Arquitectura modelo Petición/Respuesta

2.2. Principales protocolos de aplicación para IoT

Cuando se habla de comunicaciones máquina a máquina o de redes de sensores, un aspecto fundamental a tener en cuenta es el protocolo de capa de aplicación a emplear. Las características más importantes a cumplir son principalmente, bajo consumo de ancho de banda, bajo consumo energético, retardo extremo a extremo del orden de milisegundos y bajos requerimientos *hardware*. Actualmente, los principales protocolos cumplen con dichas características de forma diferente lo que genera una gran dificultad a la hora de definir el protocolo de aplicación ideal para este tipo de comunicaciones, por lo que hoy por hoy no existe un único protocolo estandarizado para estas comunicaciones.

En general, los protocolos de comunicación propuestos difieren en el paradigma de interacción, es decir, *request/response* o *publish/subscribe* descritos en la sección 2.1.1. En primer lugar, como ya se ha comentado antes, *request/response* representa un intercambio de mensajes comúnmente conocido en arquitecturas cliente-servidor. Los dos protocolos más utilizados en este tipo de arquitecturas son HTTP y CoAP. Por otro lado, *publish/subscribe* representa un intercambio de mensajes asíncrono empleado en arquitecturas centralizadas mediante un *broker* de mensajes. Los protocolos más empleados en este tipo de arquitecturas son MQTT, MQTT-SN *(MQTT for Sensor Networks)* [53], AMQP *(Advanced Message Queuing Protocol)* [55] y DDS *(Data Distribution Service)* [44].

En este capítulo, se exponen los principales protocolos de capa de aplicación empleados en IoT, centrándonos particularmente en MQTT, MQTT-SN, CoAP y HTTP ya que estos son los más empleados actualmente en este tipo de comunicaciones [54] tal y como muestra la Figura 1 y sobre los que posteriormente se centra el desarrollo de la solución técnica.

2.2.1. MQTT *(Message Queue Telemetry Protocol)*

MQTT se trata de un protocolo de mensajes originalmente desarrollado en 1999 por Andy Standford-Clark (IBM) y Arlen Nipper (Arcom, ahora Cirrus Link) [28]. Diseñado bajo el paradigma *publish/subscribe* con el objetivo de proponer un protocolo de mensajes ligero, de bajo consumo energético y empleando el mínimo ancho de banda. Actualmente adoptado como estándar por OASIS [38] para las comunicaciones IoT.

MQTT funciona sobre TCP *(Transmission Control Protocol)* como protocolo de transporte, el cual está orientado a conexión, garantizando la entrega fiable de paquetes además de otras características como control de flujo y control de congestión. Uno de los aspectos negativos de emplear TCP como protocolo de transporte, es el aumento del retardo experimentado durante el establecimiento de la conexión (envió de mensajes SYN, SYN/ACK y ACK) junto con un aumento del *overhead* debido al tamaño de la cabecera TCP y a la existencia de ACKs. Si bien es cierto, tal y como relatan los autores de [11] en comparación con otros protocolos de nivel de aplicación que emplean TCP, como por ejemplo HTTP, gracias a su liviano tamaño de paquete, se trata de un protocolo

muy bien considerado dentro del contexto de las comunicaciones entre dispositivos de recursos limitados.

MQTT está basado en sesiones, esto quiere decir que, tras establecer la conexión TCP, el proceso completo de comunicación se divide en cuatro etapas, creación de la conexión MQTT, autenticación, comunicación y terminación de la sesión. Para ello se definen los siguientes tipos de mensajes.

- CONNECT: mensaje enviado por el cliente como solicitud de conexión. Contiene información necesaria para el establecimiento de una sesión MQTT.
- CONNACK: mensaje enviado por el *broker* como confirmación del CONNECT, sin este mensaje, el cliente debe cerrar la sesión.
- PUBLISH: mensaje enviado por el cliente para publicar datos sobre un *topic*, contiene principalmente el nombre del tema, los datos y el nivel de QoS requerido.
- PUBACK: mensaje de confirmación enviado como respuesta al PUBLISH, empleado en configuraciones con QoS 1.
- PUBREC: mensaje enviado como respuesta al PUBLISH, empleado en configuraciones con QoS 2.
- PUBREL: mensaje enviado como respuesta al PUBREC, empleado en configuraciones con QoS 2.
- PUBCOMP: mensaje enviado como respuesta al PUBREL, es el cuarto y último paquete empleado en configuraciones con QoS 2.
- SUBSCRIBE: mensaje enviado por el cliente al *broker* para crear una o más suscripciones a los *topic*s de interés. El *broker* envía mensajes PUBLISH a modo de notificación cada vez que recibe datos sobre dichos *topics.* El paquete contiene también el máximo valor de QoS requerido con la que espera recibir los datos mensajes PUBLISH por parte del *broker*.
- SUBACK: mensaje de confirmación enviado como respuesta al SUBSCRIBE. Contiene el valor máximo de QoS otorgado para cada suscripción.
- UNSUSBCRIBE: mensaje enviado por el cliente al *broker* como cancelación de suscripción a un *topic.*
- UNSUBACK: mensaje enviado por el *broker* al cliente como respuesta al UNSUBSCRIBE, confirmando la cancelación de la suscripción.
- PINGREQ: paquete enviado por el cliente, puede usarse para indicar que el cliente sigue activo, para requerir que el *broker* comunique que sigue activo o para indicar que la conexión sigue activa.

- PINGRESP: paquete enviado por el *broker* en respuesta al PINGREQ, indica que el *broker* sigue activo.

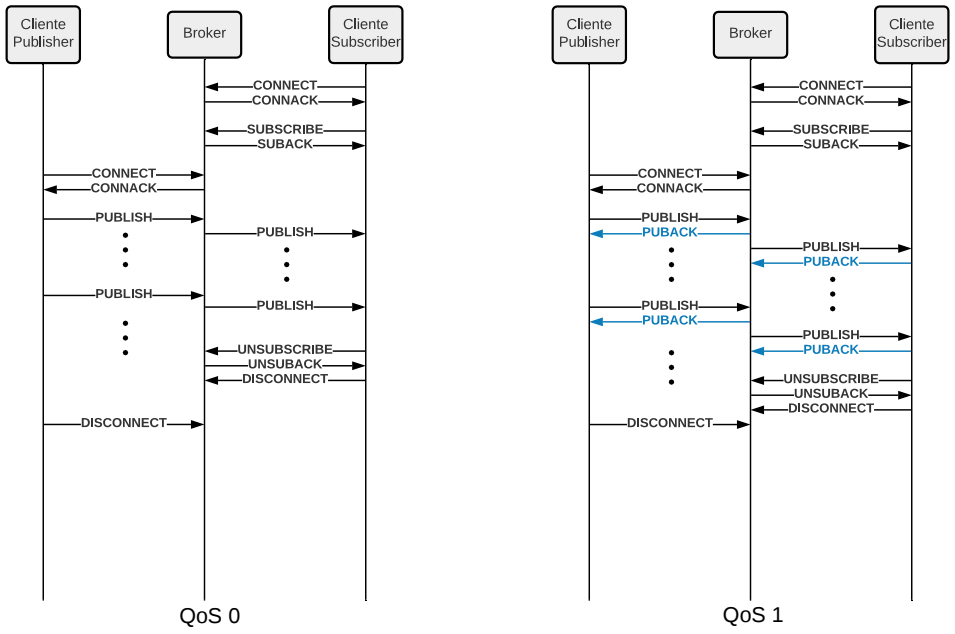
- DISCONNECT: paquete final de la sesión, enviado por el cliente indicando que se ha desconectado limpiamente.

Por último, se definen tres niveles de calidad de servicio, QoS 0, 1 y 2 [43].

- QoS 0: el receptor no envía confirmación sobre la recepción de un mensaje PUBLISH, por lo que el remitente tampoco realiza ningún reenvío, esto hace que no se garantice la recepción del mensaje. Se le conoce bajo el nombre *"at most once delivery"*.

- QoS 1: el receptor confirma la recepción de un mensaje PUBLISH mediante un mensaje PUBACK, garantizando que el paquete se reciba al menos una vez, por lo que se le conoce como "*at least once delivery"*.
- QoS 2: se trata del nivel más alto de calidad de servicio, empleado cuando no se acepta la perdida de mensajes ni su duplicación. Garantiza la entrega del mensaje exactamente una vez sin duplicados, por lo que se le conoce como *"exactly once delivery"*.

A continuación, la Figura 5 muestra un ejemplo de comunicación entre un cliente suscriptor y un cliente publicador a través de un *broker*, ambos clientes con Qos 0, 1 y 2 respectivamente, incluyendo establecimiento y finalización de la comunicación MQTT. Cabe destacar que la configuración QoS por parte de los clientes *publisher* y *subscriber* es totalmente independiente por lo que no necesariamente ambos extremos de la comunicación deben interactuar bajo los mismos requisitos de QoS.



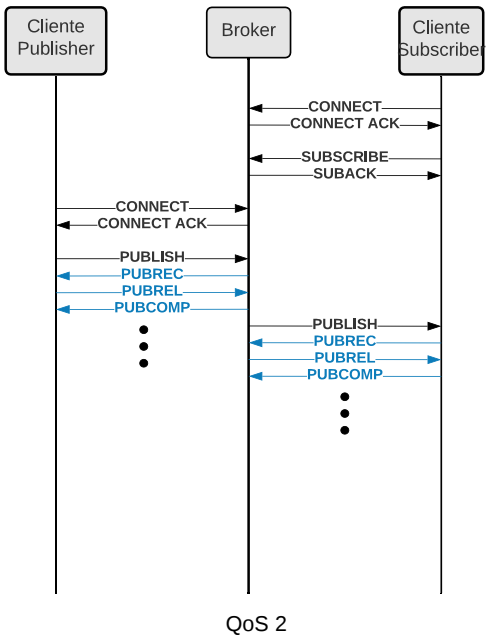


Figura 5. Ejemplo de comunicación MQTT para diferentes niveles de QoS

Por último, resulta interesante destacar que MQTT no define cifrado *per se*, por lo que los datos se transmiten como texto plano. Para garantizar comunicaciones seguras, es necesario implementar una capa de cifrado de forma independiente, comúnmente utilizando TLS a nivel de transporte *(Transport Layer Security)* [9].

2.2.2. MQTT-SN *(Message Queue Telemetry Protocol for Sensor Networks)*

MQTT-SN [53] se trata de una especificación de MQTT diseñada para redes de sensores donde el protocolo TCP resulta demasiado pesado. En redes de sensores donde se prioriza el ahorro energético, la diferencia entre usar TCP o UDP como protocolo de trasporte resulta muy significativa. Se trata de un proyecto promovido por OASIS que a pesar de ser una especificación publica, no está reconocida ni aprobada por el organismo de normalización OASIS, tal y como afirman en [29].

Así, MQTT-SN se basa en el funcionamiento de MQTT, pero emplea UDP como protocolo de transporte, disminuyendo la cantidad de bytes a transmitir, con el fin de reducir el consumo de ancho de banda y a su vez el consumo energético.

MQTT-SN también está diseñado sobre el paradigma *publish/subscribe*, por lo que necesita un *broker* al igual que MQTT mediante el cual publicar y recibir información. Para ello, la especificación define la arquitectura mostrada en la Figura 6.

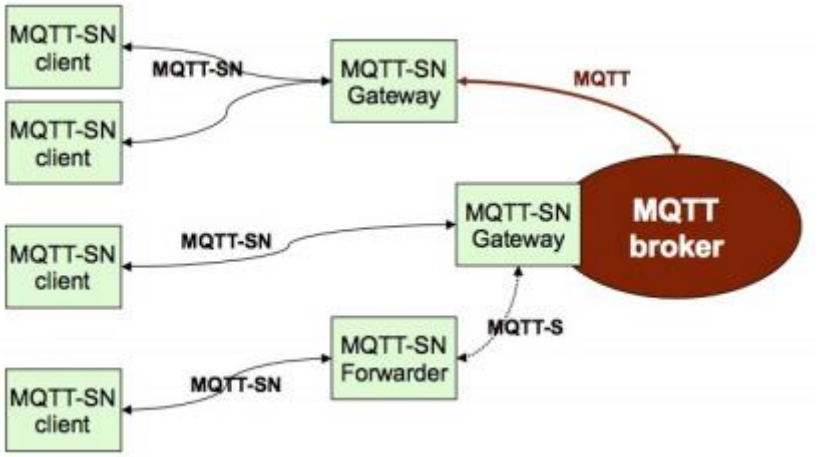


Figura 6. Arquitectura de MQTT-SN, obtenida de [53]

Como se puede ver en la arquitectura, se definen tres tipos de nodos diferentes.

- MQTT-SN Client: clientes MQTT-SN que se conectan al *broker* MQTT a través del MQTT-SN Gateway.
- MQTT-SN Gateway: puede o no estar integrado en el *broker* MQTT. Su función es la traducción entre MQTT-SN y MQTT.
- MQTT-SN Forwarder: sirve como unión entre el cliente MQTT-SN y el MQTT-SN Gateway en caso de que este último no esté en la misma red del cliente.

MQTT-SN, al igual que MQTT establece los niveles de QoS 0, 1 y 2, cuyo funcionamiento es similar al de MQTT. Adicionalmente, se define el nivel de QoS -1 en mensajes de tipo PUBLISH. Este permite enviar publicaciones sin necesidad de crear una conexión con el *broker*, no se transmite el mensaje CONNECT de creación de la conexión, y únicamente se transmite el mensaje PUBLISH sin asegurarse de que la comunicación con el MQTT-SN sea correcta. Esta característica, guarda relación con CoAP puesto que, a nivel de aplicación, este no está orientado a conexión.

2.2.3. CoAP *(Constrained Application Protocol)*

CoAP es un protocolo de transferencia de mensajes estandarizado por el IETF en Junio de 2014 definido en el estándar RFC 7252 [52]. Su uso está destinado a dispositivos con recursos limitados. Diseñado bajo el paradigma petición/respuesta, proporciona un modelo de intercambio de mensajes para transferir datos de sensores como temperatura, humedad, ubicación… en arquitecturas de red tipo REST, permitiendo fácilmente su traducción a HTTP. Esto permite la integración de datos de sensores en servicios basados en *web* aportando gran valor en el ecosistema IoT tal y como lo detalla el instituto ETSI [16].

En su primera versión, RFC 7252 [52], CoAP se define sobre UDP *(User Datagram Protocol)* como protocolo de transporte en lugar de TCP reduciendo el

overhead a costa de eliminar la fiabilidad en la entrega de paquetes que este último garantiza. Debido a la perdida de la fiabilidad que supone el uso de UDP, en Febrero de 2018, ETSI propone el uso de CoAP sobre TCP en el estándar RFC 8323 [5] con el fin de mejorar la fiabilidad en el uso de CoAP, evitando la pérdida de paquetes y garantizando mecanismos de control de flujo y de congestión.

CoAP se divide estructuralmente en dos capas como se muestra en la Figura 7.

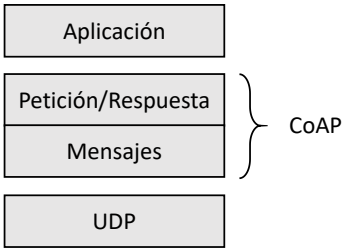


Figura 7. Estructura de capas de CoAP

La primera capa define el paradigma petición/respuesta tradicional, muy similar a HTTP. Esta capa establece los métodos GET, PUT, POST o DELETE que los clientes pueden emplear para generar peticiones sobre diferentes URI *(Uniform Resource Identifier)* dirigidas a un servidor y los diferentes códigos de respuesta. En redes de sensores, por ejemplo, un cliente puede usar el método GET en una petición dirigida a un servidor y como respuesta recibirá un paquete con los datos requeridos. Para llevar a cabo la comunicación, CoAP, en la segunda capa define cuatro tipos de mensajes: *Confirmable* (CON), *Non-Confirmable* (NON), *Acknoledgement* (ACK) y *Reset* (RST). Se pueden diferenciar dos posibles configuraciones, con mensajes confirmables y no confirmables dotando a CoAP de dos niveles de QoS diferentes, ver Figura 8. La confirmación de mensajes permite asegurar una comunicación fiable frente a pérdidas de paquetes a nivel de aplicación, supliendo en parte la carencia de fiabilidad en caso de emplear UDP como protocolo de transporte. Si lo comparamos con el protocolo MQTT descrito en la sección 2.2.1 la configuración de QoS 0 de MQTT es equivalente al uso de mensajes NON y la configuración de QoS 1 de MQTT es equivalente al uso de mensajes CON.

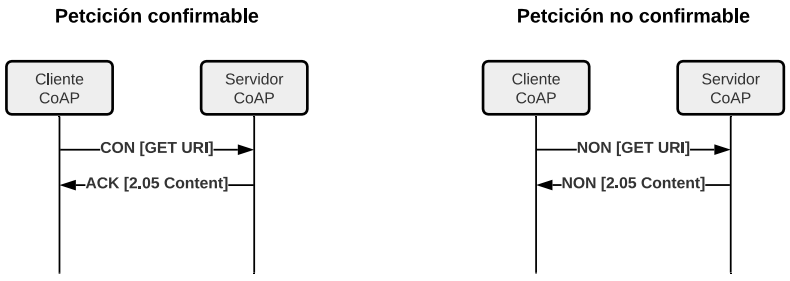


Figura 8. Tipos de mensajes CoAP (confirmables y no confirmables)

Por otro lado, una funcionalidad de suma relevancia que le aporta valor añadido al protocolo CoAP y que va más allá del modelo petición/respuesta, se trata de la opción de OBSERVACIÓN. Se trata de una opción o *flag* adicional a la petición GET que

permite a los clientes mantener la comunicación abierta y recibir notificaciones continuas de forma asíncrona por parte del servidor cada vez que cambia el estado del recurso solicitado, tal y como se muestra en la Figura 9. Esta funcionalidad acerca al protocolo CoAP al modelo *publish/subscribe* aportándole una gran flexibilidad.

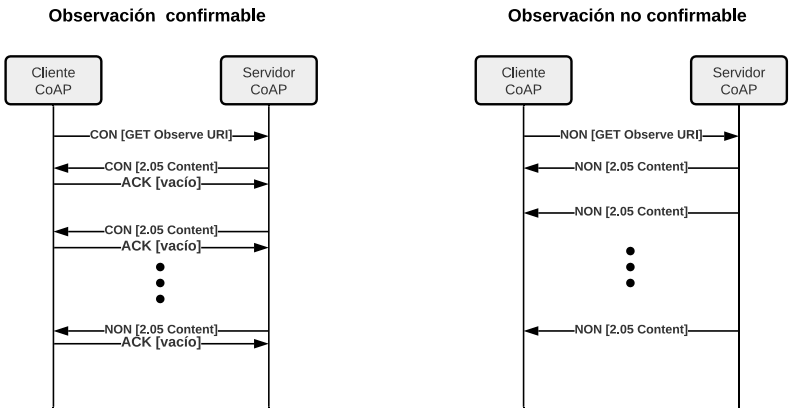


Figura 9. Observación de CoAP

Además, en un esfuerzo de acercar el protocolo CoAP de forma definitiva al paradigma *publish/subscribe*, IETF en Octubre de 2016 publico *draft-ietf-core-coap-pubsub* [32] en el que se propone una arquitectura de publicación/suscripción centralizada en un *broker* para CoAP, siendo su última actualización en Abril de 2020. La Figura 10 muestra la arquitectura propuesta por el IETF.

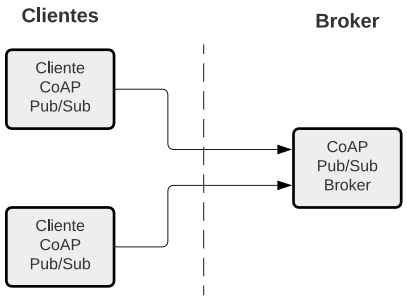


Figura 10. Arquitectura publish/subscribe de CoAP propuesta.

Por último, como capa de seguridad, inicialmente se define DTLS *(Datagram Transport Layer Security)* [47] como mecanismo de seguridad empleado sobre UDP en la versión original de CoAP, RFC 7252 [52]. Con la publicación de la RFC 8323 [5] en la que se incluye el uso de CoAP sobre TCP, se define también TLS [46] como mecanismo de seguridad.

2.2.4. HTTP *(Hypertext Transfer Protocol)*

HTTP es un protocolo de nivel de aplicación basado en una arquitectura cliente-servidor frecuentemente utilizado en servicios *web*. La versión comúnmente utilizada del protocolo es HTTP/1.1, definida en la RFC 2616 [17] en junio de 1999. Diseñado bajo el paradigma petición/respuesta, proporciona un modelo de intercambio de datos entre cliente y servidor basado en peticiones.

Del mismo modo que CoAP, HTTP define los métodos GET, PUT, POST o DELETE mediante los cuales el cliente puede interactuar solicitando datos, actualizarlos o borrándolos respectivamente en un servidor. A pesar de no haber sido diseñado para escenarios IoT, autores como [4] [1] [58] [40] comparan el rendimiento de HTTP con protocolos como MQTT y CoAP. Si bien es cierto que el gasto de recursos es mucho mayor en HTTP frente a otros protocolos, son muchas las aplicaciones IoT basadas en HTTP actualmente.

En relación con el protocolo de transporte utilizado, HTTP usa TCP aportando fiabilidad en la entrega de mensajes además de control de flujo y de congestión. Un detalle importante a tener en cuenta es que HTTP está diseñado para el envío esporádico de información, por lo que la creación de la conexión TCP cada vez que se inicia una comunicación, da como resultado un gran aumento del ancho de banda consumido frente a otros protocolos. Por otro lado, no ofrece opciones de QoS como si ocurre en MQTT y CoAP puesto que TCP garantiza la entrega correcta de los paquetes a nivel de transporte.

Por último, como es bien sabido, HTTP emplea TLS como mecanismo de seguridad habilitando un canal de comunicaciones cifrado, conocido como HTTPS.

2.2.5. Otros protocolos IoT

Como ya se ha comentado, MQTT, CoAP y HTTP son los principales protocolos de nivel de aplicación empleados en la gran mayoría de soluciones IoT, a pesar de esto, existen otros protocolos de menor popularidad como DDS, AMQP y XMPP *(Extensible Messaging and Presence Protocol).*

DDS se trata de un protocolo implementado bajo el paradigma *publish/subscribe* estandarizado por OMG [44] pero a diferencia de otros protocolos similares, DDS define una arquitectura *peer-to-peer* descentralizada en la que no se depende de un *broker.* Una de las ventajas de usar DDS es el amplio abanico de niveles de QoS definidos en el estándar [7] más de veinte niveles. Como protocolo de transporte, se definen tanto TCP como UDP implementando TLS o DTLS respectivamente.

AMQP es otro protocolo *publish/subscribe* definido por OASIS en [55] pensado para transmitir flujos de datos y transacciones comerciales en tiempo real. Evita soluciones propietarias, ofreciendo como potencial la reducción de costes de implementación empresarial. AMQP define dos versiones totalmente diferentes, AMQP 0.9.1 establece una arquitectura centralizada en un *broker* de mensajes mientras que AMQP 1.0 únicamente propone el protocolo sin especificar la arquitectura, pudiéndose

usar en comunicaciónes *peer-to-peer*. Como protocolo de transporte emplea TCP garantizando comunicaciones seguras gracias a TLS y define 3 niveles de QoS diferentes.

Por último, XMPP es un protocolo estandarizado por el IETF en los estándares RFC 6120 [49] y RFC 6121 [50], diseñado originalmente para mensajería de texto instantánea entre aplicaciones basado en XML *(Extensible Markup Language) y* diseñado para soportar interacción cliente-servidor y *publish/subscribe*. Como protocolo de transporte emplea TCP y también incorpora TLS como mecanismo de seguridad. A diferencia de otros protocolos, no define diferenciación en niveles de QoS, al igual que HTTP.

Finalmente, se muestra la Tabla 1 en la que se resumen brevemente los principales fundamentos de los protocolos descritos a lo largo del capítulo.

Protocolo	Paradigma	Prot. Trasporte	QoS	Seguridad
MQTT	Pub-Sub	TCP	3 niveles	TLS/SSL
CoAP	Req-Resp	UDP/TCP	2 niveles	DTLS y TLS
HTTP	Req-Resp	TCP	-	TLS/SSL
DDS	Pub-Sub	TCP/UDP	Mas de 20 niveles	TLS y DTLS
AMQP	Pub-Sub y Req-Resp	TCP	3 niveles	TLS/SSL
XMPP	Pub-Sub y Req-Resp	TCP	-	TLS/SSL

Tabla 1. características principales de los protocolos de aplicación IoT

2.3.Protocolos de transporte para comunicaciones seguras

Las aplicaciones IoT al igual que el resto de los servicios de información basados en redes de comunicaciones están expuestos a entornos de red conflictivos, por lo que resulta impensable desarrollar un sistema de comunicación que no garantice confidencialidad, autenticidad e integridad. Es por ello, por lo que en esta sección se van a introducir brevemente los protocolos de nivel de transporte TLS y DTLS utilizados comúnmente por las aplicaciones IoT y que se emplean en la solución técnica del proyecto.

2.3.1. TLS *(Transport Layer Security)*

TLS es un protocolo diseñado para garantizar privacidad e integridad en las comunicaciones de datos entre clientes o servidores de red. Se pueden diferenciar varias versiones. Las versiones más modernas y recomendadas son TLS 1.2 definida en el estándar RFC 5246 [9] y TLS 1.3 definida en el estándar RFC 8446 [46], ya que versiones anteriores han quedado obsoletas.

Se trata de un protocolo orientado a conexión, antes de empezar la transmisión de datos de aplicación se realiza el conocido TLS *Handshake* en el que se realiza la autenticación del cliente y servidor, la negociación del *cipher suite* con el que se cifran

los datos y el intercambio de claves. La Figura 11 muestra el intercambio de mensajes durante el TLS 1.2 *handshake*.

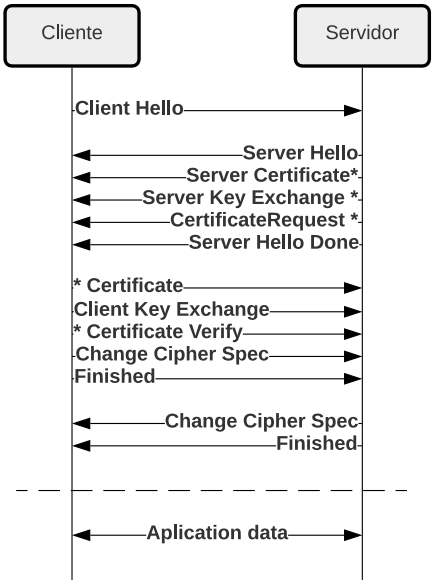


Figura 11. Handshake de TLS 1.2

* Indica mensajes opcionales o dependientes de la situación, no siempre enviados.

Durante el *handshake* se producen tres eventos a destacar:

- Negociación del *Cipher Suite:* cada extremo de la comunicación enumera los algoritmos de cifrado que admite y en esta fase de negociación, se fija el algoritmo de cifrado a emplear.
- Autenticación: ambos extremos de la comunicación se autentican mediante el uso de certificados (el cliente se autentica bajo petición del servidor).
- Intercambio de claves: el cifrado de los datos de aplicación se realiza mediante criptografía de clave simétrica, una misma clave compartida por el cliente y el servidor. Para el establecimiento de dicha clave entre ambos, se emplea criptografía de clave asimétrica (clave pública y privada), el cliente envía una secuencia aleatoria al servidor cifrada mediante la clave pública del servidor. Con esta secuencia, ambos generan la clave simétrica empleada para cifrar las comunicaciones posteriores. El *handshake* concluye con el envío de un mensaje de finalización cifrado con la clave secreta simétrica.

2.3.2. DTLS *(Datagram Transport Layer Security)*

DTLS es un protocolo de nivel de transporte que garantiza seguridad en las comunicaciones basadas en UDP, definido por el ETSI en el estándar RFC 6347 [47]. DTLS se basa en el protocolo TLS utilizado en comunicaciones sobre UDP.

El funcionamiento de DTLS es similar al funcionamiento de TLS, inicialmente se realiza un intercambio de mensajes, *handshake*¸ en el que se establecen la configuración de la sesión y a continuación se transmiten los datos de forma segura y cifrada.

2.4. Interoperabilidad entre protocolos

Como se ha plasmado en la sección 2.3, en torno al ecosistema IoT existe un gran conjunto de protocolos de nivel de aplicación basados en paradigmas y arquitecturas muy diferentes. La falta de un protocolo unificado, estandarizado y normalizado para las comunicaciones IoT provoca que actualmente, cada fabricante cree sus propios protocolos y los diseñadores de servicios basados en IoT como por ejemplo las redes de sensores, *Smart cities, Smart grid,* o la industria 4.0 usen diferentes protocolos de aplicación sin preocuparse siquiera por homogeneizar dichas comunicaciones.

Esto puede dar como resultado escenarios IoT heterogéneos a causa de las diferencias entre protocolos, ya sea debido al paradigma de comunicación (petición/respuesta o publicación/suscripción) o a características intrínsecas de los protocolos. Esta situación deriva en problemas tales como la dependencia de *hardware* propietario para poder utilizar protocolos de comunicación privados o la necesidad de disponer de aplicaciones compatibles con cada protocolo para obtener datos de diferentes proveedores.

Partiendo de la definición de interoperabilidad, la cual se define como “la habilidad de dos o más sistemas o componentes para intercambiar información y usar la información que se ha intercambiado” según IEEE en [31], nos encontramos ante una comunidad que presenta grandes dificultades de interoperabilidad.

Por este motivo, cada vez son más los esfuerzos realizados por buscar puntos de unión entre los diferentes protocolos expuestos con el fin de lograr la mayor interoperabilidad posible entre ellos. Es así como surgen iniciativas de proyectos *software* que permitan unificar las comunicaciones entre protocolos.

Es común encontrar sistemas *proxy* entre CoAP y HTTP debido a la similitud de los protocolos tanto arquitectural como semántica. Ambos protocolos utilizan el protocolo de interacción petición/respuesta, se fundamentan en la arquitectura cliente-servidor y comparten gran parecido en cuanto a los mensajes definidos en los dos estándares (son similares semánticamente). Estos aspectos facilitan en gran medida la traducción entre ambos protocolos, de manera que puede realizarse a través de una plataforma *software* intermedia. De esta manera, si se desea acceder a datos accesibles a través de CoAP, mediante tecnologías tradicionales fundamentadas en http, una

comunicación indirecta, a través de una plataforma que realiza la traducción, puede ser de gran interés. Existen implementaciones como *crosscoap* [24] o la de los autores [35].

Por otro lado, también se pueden encontrar sistemas que habilitan la comunicación entre HTTP y MQTT. No obstante, en este caso, debido a la diferencia entre los protocolos, la interoperabilidad bidireccional no resulta tan sencilla. Teniendo en cuenta que MQTT está basado en un modelo de interacción *publish/subscribe* mediante una arquitectura centralizada en un *broker*, y HTTP se basa en un modelo petición/respuesta mediante una arquitectura cliente-servidor, estos sistemas se fundamentan en la traducción de los métodos PUT y POST de HTTP, que permiten actualizar los datos en el servidor, en mensajes MQTT PUBLISH, permitiendo generar datos en un *broker* MQTT a través de HTTP. Por otro lado, si lo que se quiere es obtener información publicada mediante MQTT desde peticiones HTTP GET, es habitual encontrarnos con implementaciones que retienen los datos publicados por los clientes MQTT en un servidor HTTP y este responde las peticiones con los datos almacenados. Actualmente existen proyectos como [1] [25] [26] y [30].

Por último, lo que resulta más interesante dadas las tendencias de protocolos predominantes en IoT actualmente, se trata de un sistema de mensajería que permita una interoperabilidad total entre MQTT, CoAP y HTTP con la posibilidad de añadir futuros protocolos. Bajo estas condiciones, el proyecto que cuenta con más solidez y dispone de una implementación de código abierto *plug and play* y funcional se trata del *broker* Ponte, del cual se habla continuación.

2.4.1. Ponte

Ponte [45] [8] es un sistema de mensajería que actúa como pasarela entre los protocolos MQTT, CoAP y HTTP, desarrollado por Eclipse Foundation [13], actualmente se encuentra en fase de incubación tal y como indican en la documentación oficial. Pese a que no se trata de una solución final, es posible encontrar el código de la última versión en Github [23].

Se trata de un proyecto diseñado sobre el entorno de trabajo Node.Js y de rápida instalación, únicamente requiere disponer de la versión node.js 0.10 y la última versión disponible de NPM (sistema de gestión de paquetes por defecto de node.js).

En su documentación oficial se detalla la arquitectura mostrada en la Figura 12.

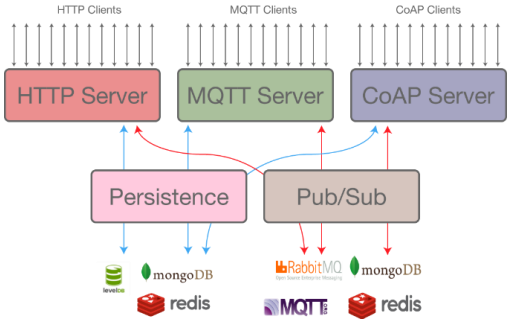


Figura 12. Arquitectura de Ponte, obtenida de [45]

Se diferencian tres interfaces diferentes, un servidor HTTP, un servidor MQTT y un servidor CoAP, además de diferentes herramientas de almacenamiento e indexación de datos.

El servidor MQTT está diseñado para funcionar como *broker*, se corresponde con una implementación de código abierta llamada Mosquito [15] que pertenece a Eclipse Foundation. Este se encarga de procesar las diferentes publicaciones y suscripciones MQTT.

El servidor CoAP implementa una interfaz que permite manejar peticiones con los métodos GET, PUT, POST y DELETE típicos del patrón REST. Permite a un cliente CoAP publicar datos a través de los métodos PUT y POST al igual que obtener información de ellos a través del método GET. Tiene total compatibilidad con la opción de observación por lo que es fácilmente equiparable al paradigma *publish/subscribe* de MQTT.

Por otro lado, el servidor HTTP funciona de forma similar al servidor CoAP, mediante los métodos PUT y POST permite a un cliente HTTP publicar datos en el sistema mientras que con el método GET permite obtener los datos. Es importante destacar que HTTP no dispone de la funcionalidad de suscripción como MQTT o de observación como CoAP, por lo que los datos publicados deben ser retenidos por el *broker* Ponte, para poder indexarlos desde HTTP en caso de recibir una petición GET.

Resulta interesante destacar los aspectos que, de momento no se abordan en el *broker* Ponte y que consideramos que son de utilidad para garantizar la mayor interoperabilidad posible y en las mejores condiciones. En primer lugar, desde el punto de vista de CoAP, Ponte únicamente da servicio a aplicaciones basadas en clientes CoAP que generan mensajes PUT/POST sobre el *broker*. Sin embargo, no se contempla la posibilidad de dar servicio a aplicaciones basadas en servidores CoAP tradicionales definidos en el estándar, que son los más utilizados en despliegues típicos. Por otro lado, Ponte, si bien en su documentación oficial indica que se encuentra en desarrollo, actualmente no incorpora ningún mecanismo de cifrado en las comunicaciones. Por último, dado el auge de la implantación de arquitecturas IoT basadas en *edge computing*, resultaría interesante disponer de una versión de Ponte que permitiera aumentar la escalabilidad del sistema en escenarios con diferentes *broker* interconectados, característica que actualmente no posee.

Todos estos aspectos se abordan en este proyecto mediante el desarrollo de un *middleware* que además de las funcionalidades proporcionadas por Ponte, da servicio a servidores CoAP tradicionales y permite comunicaciones tanto cifradas mediante TLS/DTLS como sin cifrar. Además, se propone una solución que permite tener múltiples instancias del *broker* interconectadas, posibilitando la escalabilidad del sistema.

3. Implementaciones *software* existentes

En este capítulo se muestran las principales librerías *software* existentes y se aborda la elección de las librerías utilizadas durante el desarrollo técnico. A continuación, se muestra una breve revisión de las implementaciones más destacadas de MQTT, MQTT-SN y CoAP.

3.1. Implementaciones CoAP

En la Tabla 2 se muestran las librerías más conocidas y utilizadas por la comunidad CoAP, junto a sus características principales.

Nombre	Lenguaje	RFC implementada	Cliente / Servidor	Características	Licencia	Referencia
Aiocoap	Python 3	7252	Cliente y servidor	Blockwise Transfers y Observe	MIT	[2]
Californium	Java	7252	Cliente y servidor	Observe, Blockwise Transfers y DTLS	EPL + EDL	[12]
CoAPThon	Python	7252	Cliente y servidor	Blockwise Transfers, Observe, DTLS	MIT	[6]
Libcoap	C	7252	Cliente y servidor	Blockwise Transfers, Observe y DTLS	BSD/GPL	[33]
Node-coap	JavaScript	7252	Cliente y servidor	Blockwise Transfers	MIT	[41]

Tabla 2. Implementaciones de CoAP

3.2. Implementaciones MQTT

En la Tabla 3 se muestran las librerías más conocidas y utilizadas por la comunidad MQTT, junto a sus características principales.

Nombre	Lenguaje	Cliente/*Bróker*	Versiones del protocolo	TLS	Licencia	Referencia
Mosquitto	C	Cliente y *broker*	5.0, 3.1.1 y 3.1	Si	EPL/EDL	[15]
Paho MQTT	C, C++, Java, JavaScript, Python, Go	Cliente	3.1.1 y 3.1	Si	EPL	[14]

wolfMQTT	C	Cliente	5.0 y 3.1.1	Si	GPL	[57]
Moquette	Java	*Broker*	3.1.	Si	EPL	[37]
HiveMQ CE	Java	*Broker*	5.0 y 3.x	Si	Apache v2	[27]
HiveMQ Client	Java	Cliente	5.0 y 3.1.1	SI	Apache v2	[27]

Tabla 3. Implementaciones de MQTT

3.3. Implementaciones MQTT-SN

MQTT-SN se trata de un protocolo relativamente nuevo, por lo que no posee la suficiente madurez como MQTT o CoAP. Esto hace que existan escasas implementaciones y no tan contrastadas y utilizadas como ocurre con los otros protocolos. A continuación, la Tabla 4 muestra alguna implementación disponible en Github de MQTT-SN.

Nombre	Lenguaje	Cliente/GW /Forwarder	Versiones del protocolo	Licencia	Referencia
Mqtt-sn-tools	C	Cliente y *forwarder*	1.2	MIT	[22]
Mqtt-sn	C	Cliente y *gateway*	1.2	MPL	[18]
Paho.mqtt-sn.embedded-c	C	Cliente (incompleto) y *gateway*	1.2	EPL y EDL	[20]
Mqtt-sn-gateway	Java	*Gateway*	1.2	EDL y EPL	[21]
Mqttsn	Python	Cliente	1.2	MIT	[39]

Tabla 4. Implementaciones de MQTT-SN

3.4. Implementaciones elegidas

Para el desarrollo técnico del proyecto se ha elegido Java como lenguaje de programación. Entre los motivos por lo que se ha escogido Java se encuentran los siguientes: se trata de un lenguaje multiplataforma que funciona sobre la máquina virtual de Java (JVM) por lo que es fácilmente portable entre distintos sistemas operativos, es un lenguaje orientado a objetos, gracias a esto se definen estructuras de fácil manipulación, se conoce como un buen lenguaje del lado del servidor [10] y permite implementar programación concurrente multihilo de forma sencilla.

Las librerías seleccionadas han sido:

- Moquette MQTT basada en Eclipse Paho para el *broker* MQTT
- Eclipse Californium para la arquitectura CoAP
- Mqtt-sn-tools para los clientes MQTT-SN
- Paho.mqtt-sn.embedded-c para el MQTT-SN Gateway
- Oracle HTTP para el servidor HTTP

Moquette y Californium son librerías desarrolladas y respaldadas por *Eclipse Foundation* en la cual participan empresas como IBM, Oracle, RedHat, Bosch o Huawei, esto garantiza estabilidad y continuas actualizaciones de software. Además, cuentan con una gran comunidad de desarrolladores donde encontrar ayuda y soporte. Por otro lado, Oracle HTTP se encuentra integrada entre los paquetes de Java contando con una extensa documentación en su API. Por último, la elección de las librerías de MQTT-SN se basó en heurística, las librerías existentes cuentan con características similares, por lo que se eligieron las librerías más sencillas de usar y que no mostraban fallos en su ejecución.

4. Solución técnica

En este capítulo se aborda el desarrollo técnico de un *middleware* de mensajería que permite la intercomunicación de clientes IoT utilizando diferentes protocolos de nivel de aplicación, MQTT, CoAP y HTTP. El proyecto *software* completo, se encuentra disponible en el repositorio de Github [19]. Adicionalmente, el Anexo 1 se muestra detalladamente la estructura de clases del proyecto.

4.1. Planteamiento del sistema

Como punto de partida se definen los casos de uso que se quieren tratar y los diferentes tipos de mensajes que se ven implicados.

Recepción de un mensaje PUBLISH MQTT

En caso de recibir un mensaje MQTT de tipo PUBLISH dirigido a un *topic*, se retransmite a todos los clientes MQTT suscritos a dicho *topic* (funcionamiento tradicional de un *broker* MQTT). A su vez, si existe algún cliente CoAP con una relación de observación establecida sobre ese *topic* (GET Observe), internamente el sistema traduce el mensaje MQTT recibido y lo reenvía en forma de notificacion CoAP a los clientes CoAP que observan dicho *topic*. Por último, el contenido del mensaje se almacena en la memoria del servidor CoAP y del servidor HTTP para poder acceder a él a través de consultas HTTP GET y CoAP GET simples (únicamente se guarda el ultimo valor actualizado).

Recepción de un mensaje PUT/POST CoAP

En caso de recibir un mensaje CoAP de tipo PUT/POST dirigido a un *topic*, se retransmite a todos los clientes CoAP que tengan una relación de observación establecida sobre ese *topic* (GET Observe). A su vez, internamente el sistema traduce el mensaje PUT/POST CoAP en un mensaje PUBLISH MQTT, de esta manera el *broker* MQTT notifica a todos los clientes MQTT suscritos a ese *topic*. Del mismo modo que antes, el contenido del mensaje se almacena en la memoria del servidor CoAP y del servidor HTTP para poder acceder a él a través de consultas HTTP GET y CoAP GET simples (únicamente se guarda el ultimo valor actualizado).

Recepción de un mensaje PUT/POST HTTP

En caso de recibir un mensaje HTTP de tipo PUT/POST dirigido a un *topic*, el sistema internamente lo traduce a un mensaje PUT/POST CoAP y lo interpreta como tal, notificando a todos los clientes CoAP que tengan una relación de observación establecida con ese *topic* (GET Observe). También lo traduce a un mensaje PUBLISH MQTT, de esta manera el *broker* MQTT notifica a todos los clientes MQTT suscritos a ese *topic*. Del mismo que en los casos anteriores, el contenido del mensaje se almacena en la memoria del servidor CoAP y del servidor HTTP para poder acceder a él a través de consultas HTTP GET y CoAP GET simples (únicamente se guarda el ultimo valor actualizado).

Con estos tres casos de uso, se tienen bajo control todas las posibles **publicaciones** de información en el *middleware* a través de los distintos protocolos. Por otro lado, un caso de uso concreto e importante es el hecho de darle servicio a servidores CoAP tradicionales. Si se recuerda el diseño arquitectural de CoAP, lejos de tratarse de un modelo *publish/subscribe* se diseñó como modelo *request/response* basándose en una arquitectura cliente-servidor. Con los casos abordados anteriormente, desde el punto de vista de CoAP, el *middleware* únicamente da servicio a aplicaciones basadas en clientes CoAP que generan mensajes PUT/POST de forma proactiva para publicar datos. Sin embargo, **uno de los principales objetivos de este sistema, es permitir integrar todas aquellas aplicaciones basadas en servidores CoAP** que recopilan datos y los comunican de forma reactiva como respuesta a una petición GET, aunando así los dos paradigmas, *publish/subscribe* y *request/response,* esto resulta fundamental puesto que, como se detalla en la sección 2.2.3, el estándar define CoAP como un modelo cliente-servidor y su funcionamiento *publish/subscribe* actualmente es una propuesta sin estandarizar, por lo que de momento existen infinidad de fabricantes y proveedores que basan sus servicios en servidores CoAP. Este funcionamiento, tal y como se ha comentado en el apartado 2.4.1, es una de las carencias del *broker* Ponte que se propone añadir. Para ello se definen los siguientes procedimientos y casos de uso.

Descubrimiento de *topics*

Debido a que un servidor CoAP no transmite datos de forma proactiva sino de forma reactiva tras la recepción de una petición, el *middleware* debe conocer qué *topics* están alojados en los servidores a los que da servicio. Para ello el sistema, debe consultar a todos los servidores CoAP qué *topics* contiene y almacenarse esta información en memoria para poder ser utilizada más tarde. Se emplea la funcionalidad de descubrimiento de recursos disponible en el estándar CoAP, los servidores constan de la URI *.well-known/core*, realizando una petición a dicha URI el servidor devuelve información de todos los *topics* que contiene. Por lo que el *middleware* define un cliente CoAP para cada servidor a los que da servicio y a través de peticiones GET a esta URI descubre todos los *topics* disponibles.

Recepción de un mensaje SUBSCRIBE MQTT

En caso de recibir un mensaje MQTT de tipo SUBSCRIBE sobre un *topic*, el *broker* MQTT añade ese cliente a la lista de suscripciones de ese *topic* y en el momento en que se recibe un mensaje MQTT PUBLISH sobre ese *topic*, el cliente es notificado, Figura 13 (a). Del mismo modo pasa con los mensajes PUT/POST CoAP o PUT/POST HTTP debido a la traducción interna entre protocolos comentada anteriormente, Figura 13 (b) y Figura 13 (c). Adicionalmente, el sistema consulta si ese *topic* corresponde con un *topic* alojado en un servidor CoAP. En caso positivo, crea un cliente CoAP que establece una relación de observación con dicho servidor (envía un GET Observe) obteniendo los datos, cada vez que el servidor los actualice e internamente se traducirá por un PUBLISH MQTT que se reenviara por la interfaz MQTT notificando al cliente suscrito, Figura 13 (d). Cabe destacar, que para mejorar la eficiencia y ahorrar el consumo de ancho de banda, esto último solo se realiza para la primera suscripción, en caso de tener más de un cliente MQTT suscrito al mismo *topic* no se replica el tráfico entre el *middleware* y el servidor CoAP, sino que se utiliza la misma conexión.

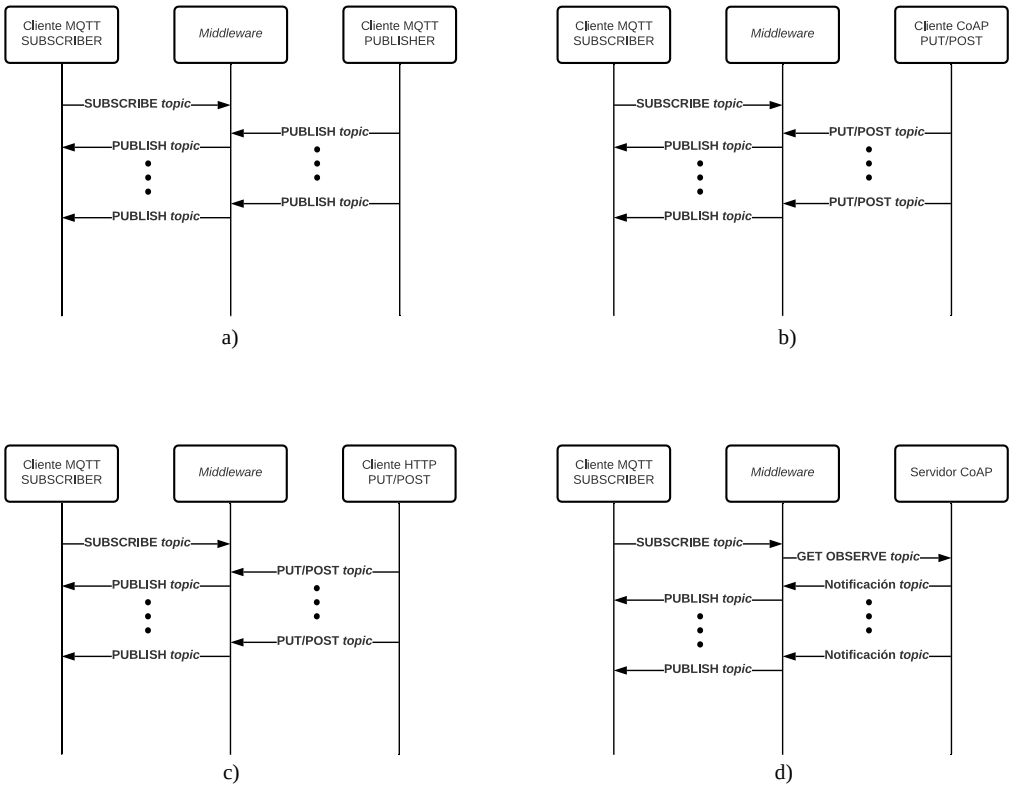


Figura 13. Caso de uso de suscripción MQTT a un topic.

Recepción de un mensaje GET OBSERVE CoAP

En caso de recibir un mensaje CoAP de tipo GET OBSERVE sobre un *topic*, se añade la relación de observación entre ese cliente y el *topic*, de forma que si se recibe un mensaje CoAP de tipo PUT/POST dirigido a ese *topic* se notificara a dicho cliente con los datos nuevos, Figura 14 (b). Del mismo modo pasa con los mensajes recibidos de tipo PUT/POST HTTP o MQTT PUBLISH gracias a la traducción interna entre protocolos que realiza el sistema, Figura 14 (c) Figura 14 (a) respectivamente. Adicionalmente, el sistema consulta si ese *topic* se corresponde con un *topic* alojado en un servidor CoAP. En caso positivo, crea un cliente CoAP que establece una relación de observación con dicho servidor (envía un GET Observe) obteniendo los datos, cada vez que el servidor los actualice e internamente se actualizan los datos en el *topic* del *middleware* notificando al cliente observador, Figura 14 (d). Al igual que antes, esto último solo se realiza para la primera suscripción (independientemente de que sea MQTT o CoAP), en caso de tener más de un cliente suscrito al mismo *topic* no se replica el tráfico entre el *middleware* y el servidor CoAP, sino que se utiliza la misma conexión.

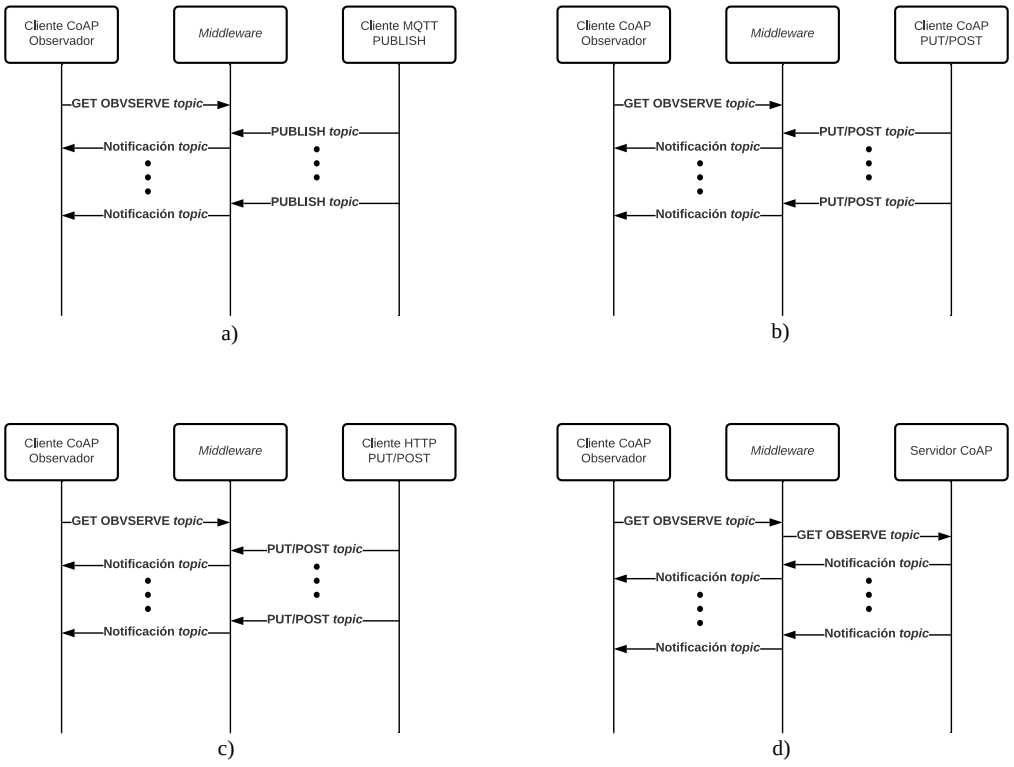


Figura 14. Caso de uso GET OBSERVE CoAP

Con los casos de uso descritos hasta el momento, se cubre la publicación y recepción de datos basados en eventos en tiempo real (un actor genera datos y otro actor en tiempo real es notificado). Sin embargo, CoAP y HTTP permiten obtener datos mediante petición/respuesta a través de los métodos HTTP GET y CoAP GET (no observe). En este caso no se busca obtener datos en tiempo real, sino el ultimo valor almacenado sobre un *topic.* Para ello se definen los siguientes casos de uso.

Recepción de un mensaje CoAP GET (no observe)

En caso de recibir un mensaje CoAP de tipo GET (no observe) sobre un *topic*, primero se consulta si el *topic* se corresponde con un *topic* alojado en un servidor CoAP. En caso positivo, se crea un cliente CoAP que retransmite la petición GET al servidor CoAP obteniendo la última información actualizada y se responde al cliente original con dicha información, Figura 15 (a). En caso de no corresponder con un *topic* alojado en un servidor CoAP, se trata de un recurso publicado mediante un mensaje PUT/POST CoAP, PUT/POST HTTP o mediante un mensaje PUBLISH, por lo que se responde con el ultimo valor publicado en el *middleware,* Figura 15 (b). Por esta razón se almacena el último dato publicado.

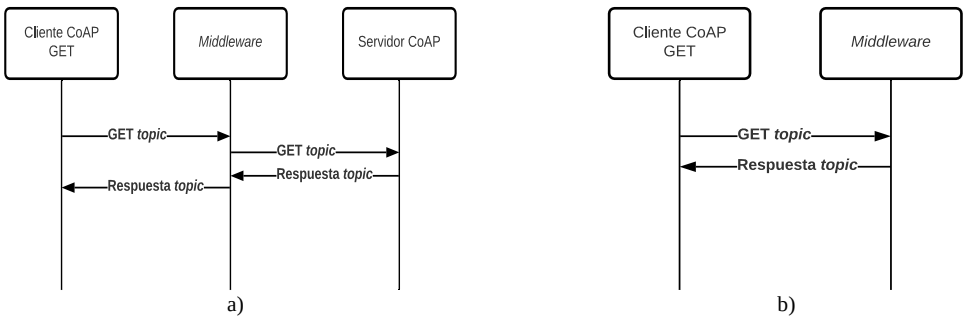


Figura 15. Caso de uso CoAP GET (no observación)

Recepción de un mensaje HTTP GET

En caso de recibir un mensaje HTTP de tipo GET sobre un *topic*, primero, se realiza el mismo proceso que en con la recepción de un mensaje CoAP GET. Si el *topic* está alojado en un servidor CoAP, se crea un cliente CoAP que retransmite la petición GET al servidor CoAP. Con la información obtenida responde a la petición HTTP, Figura 16 (a). En caso de no corresponder con un *topic* alojado en un servidor CoAP, se trata de un recurso publicado mediante un mensaje PUT/POST CoAP, PUT/POST HTTP o mediante un mensaje PUBLISH, por lo que se responderá con el ultimo valor publicado en el *middleware*, Figura 16 (b).

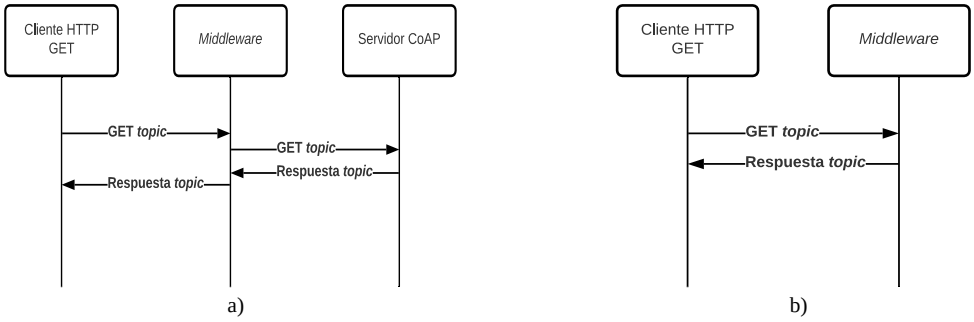


Figura 16. Caso de uso HTTP GET

A fin de evitar conflictos con los nombres de los *topics* en estos dos últimos casos de uso, se define una limitación en la configuración en los nombres de los *topics*: es fundamental que los nombres de los *topics* alojados en los servidores CoAP no sean iguales entre ellos y tampoco iguales a los *topics* publicados mediante mecanismos *publish/subscribe.*

4.2. Arquitectura del sistema

En cuanto a la solución *software*, se define una arquitectura basada en un *broker* MQTT, un *broker* CoAP y un servidor HTTP independientes. Como nexo, se define una arquitectura de clases de java que implementa todos los procedimientos necesarios para realizar la gestión interna de *topics* entre los tres servicios y la correcta traducción de protocolos para cada caso de uso. La arquitectura superficial se muestra en la Figura 17.

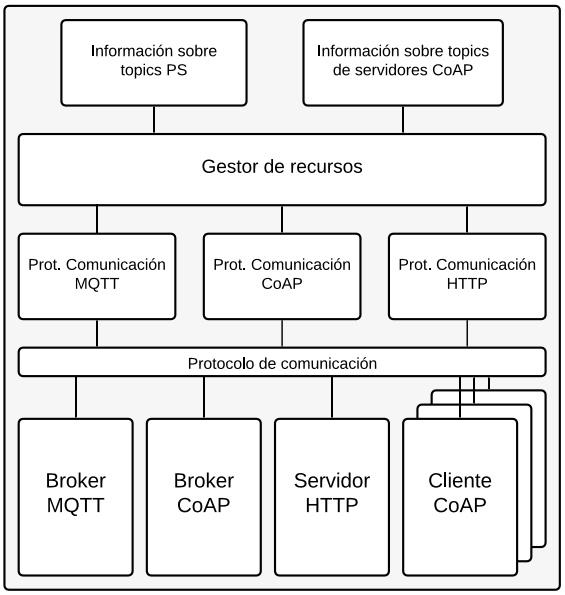


Figura 17. Arquitectura del middleware

El desarrollo de las interfaces MQTT y HTTP resulta inmediata a partir de la documentación oficial de cada librería, permitiendo realizar todo lo anteriormente comentado. Sin embargo, la librería CoAP Californium implementa la RFC 7252, que por sí sola no permite tener una estructura *publish/subscribe* basada en un *broker* CoAP tal y como se define en *draft-ietf-core-coap-pubsub*. Por esto, es necesario modificar la librería para añadir esta funcionalidad, ya que resulta necesaria para este proyecto. Afortunadamente, existe un proyecto desarrollado por Thomas Wiss en la universidad de Suecia disponible en Github [56], entre otras cosas contiene una modificación de la librería CoAP Californium en la que se implementa una API *publish/subscribe* para CoAP. Por lo tanto, en la interfaz CoAP se implementa un *broker* CoAP basado en dicho proyecto.

La principal diferencia a tener en cuenta entre la librería Californium que implementa la especificación RFC 7252 de CoAP y la librería modificada de Californium que implemente el *draft* propuesto para una arquitectura *publish/subscribe* es la siguiente. Del lado del servidor, la librería Californium CoAP está diseñada para poner en marcha un servidor que contiene diferentes *topics*, en este caso bajo diferentes URIs configuradas de forma estática en el código de la aplicación del servidor. Son los gestores de las URIs los que manejan las peticiones entrantes dirigidas a ellas. Los clientes CoAP pueden

generar peticiones con diferentes métodos para obtener datos o modificarlos. Sin embargo, es imposible generar peticiones PUT/POST de publicación sobre URIs no existentes en el servidor, de forma que no resulta viable tener una arquitectura de publicación/suscripción dinámica. Para esto, el *draft* propone la configuración de la URI */ps/* en el servidor, de forma que las publicaciones de tipo PUT/POST vayan dirigidas a dicha URI y los *topics* que se publican, se añaden como *child topic* de */ps* como muestra el ejemplo de la Figura 18. De esta forma, aunque el *topic* no exista dentro del servidor CoAP, el funcionamiento interno de la URI /ps/ es el encargado de crear los *topic* dinámicamente. Este funcionamiento viene añadido en la librería de Californium modificada por Thomas Wiss.

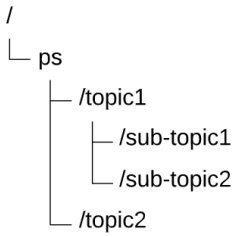


Figura 18. Ejemplo de la estructura de topic propuesta en CoAP publish/subscribe

Una vez explicadas las librerías utilizadas, a continuación, se detalla la arquitectura de *middleware* implementada mediante diferentes clases de Java.

El sistema está basado en tres interfaces independientes, cada una procesa los mensajes recibidos de cada protocolo. El hecho de tener tres procesos independientes genera la necesidad de poder comunicarlos entre ellos, para ello se plantean dos posibilidades, mediante memoria compartida o mediante paso de mensajes. Dadas las grandes diferencias de construcción de las librerías empleadas y del funcionamiento de cada protocolo, se propone la comunicación de los tres procesos mediante paso de mensajes. De esta forma no es necesario modificar en gran medida las librerías seleccionadas.

La comunicación entre los procesos y la traducción de protocolos se realiza en el *Protocolo de Comunicación* (MQTT, CoAP o HTTP), ver Figura 17, teniendo una instancia para cada proceso: *Protocolo de comunicación MQTT, Protocolo de comunicación CoAP y Protocolo de comunicación HTTP*. En ellos, se programa el funcionamiento necesario para cumplir con los casos de uso propuestos anteriormente en la sección 4.1. La sincronización entre procesos se lleva a cabo a través de la instancia *GestorRecursos*, ver Figura 17. Este gestor contiene información actualizada sobre los *topics* publicados, información sobre los *topics* alojados en servidores CoAP y sobre los clientes suscritos a cada *topic*. Por lo tanto, cada *Protocolo de comunicación,* antes de realizar la traducción de mensajes pertinente, consulta la información necesaria en el *GestorRecursos*, consiguiendo así sincronización entre procesos para su posterior comunicación.

Cada *Protocolo de comunicación* tras analizar la información necesaria, realiza la traducción y comunicación mediante paso de mensajes con los otros dos procesos restantes. A continuación, se muestran diferentes diagramas de flujo que representan los procesos de comunicación entre interfaces.

Protocolo de comunicación MQTT

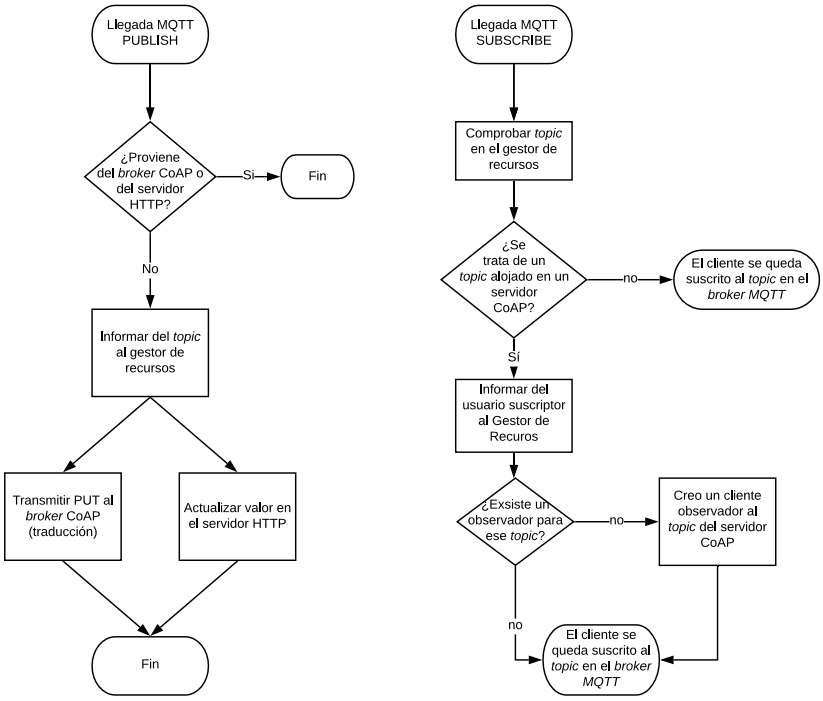


Figura 19. Diagrama de flujo de la interfaz MQTT

En la Figura 19 se observa el diagrama de flujo de la interfaz MQTT para los siguientes casos:

- Recepción de un mensaje MQTT PUBLISH: en este caso, si el mensaje proviene de un cliente MQTT PUBLISH, el *Protocolo de comunicación MQTT* informa al *Gestor de recursos* de la recepción del mensaje dirigido al *topic*. El gestor actualiza la información de control sobre el *topic* y a continuación se envía un mensaje CoAP PUT con los datos del *topic* a la interfaz CoAP. Cuando esta interfaz lo recibe, se encarga de notificar a los clientes CoAP que tienen una relación de observación a ese *topic*. Adicionalmente, se actualiza el servidor HTTP con dichos datos para poder acceder a ellos a través de peticiones HTTP GET. Por otro lado, si el mensaje MQTT PUBLISH ha sido generado por el *broker* CoAP o por el servidor HTTP, no se hace nada puesto que es fruto de la traducción de un mensaje CoAP PUT/POST o HTTP PUT/POST y el *broker* MQTT únicamente notifica a los clientes MQTT SUBSCRIBER.
- Recepción de un mensaje MQTT SUBSCRIBE: en este caso, el *Protocolo de comunicación MQTT* lo primero que hace es consultar la información de control sobre el *topic* al que hace referencia el mensaje a través del *Gestor de recursos*. Si se trata de un *topic* alojado en un servidor CoAP, notifica al *Gestor de recursos* sobre el usuario suscriptor. Esto permite

llevar la cuenta de los usuarios que están suscritos. A continuación, comprueba a través del *Gestor de recursos* si ya existe un *Observador* creado a dicho *topic* alojado en el servidor CoAP. Si ya existe uno, únicamente deja al cliente MQTT suscrito al *topic* en el *broker* MQTT. En caso de que no exista un *Observador*, lo crea y deja al cliente MQTT suscrito al *topic* en el *broker* MQTT. Por otro lado, si no se trata de un *topic* alojado en un servidor CoAP, únicamente deja al cliente MQTT suscrito en el *broker* MQTT

Protocolo de comunicación CoAP

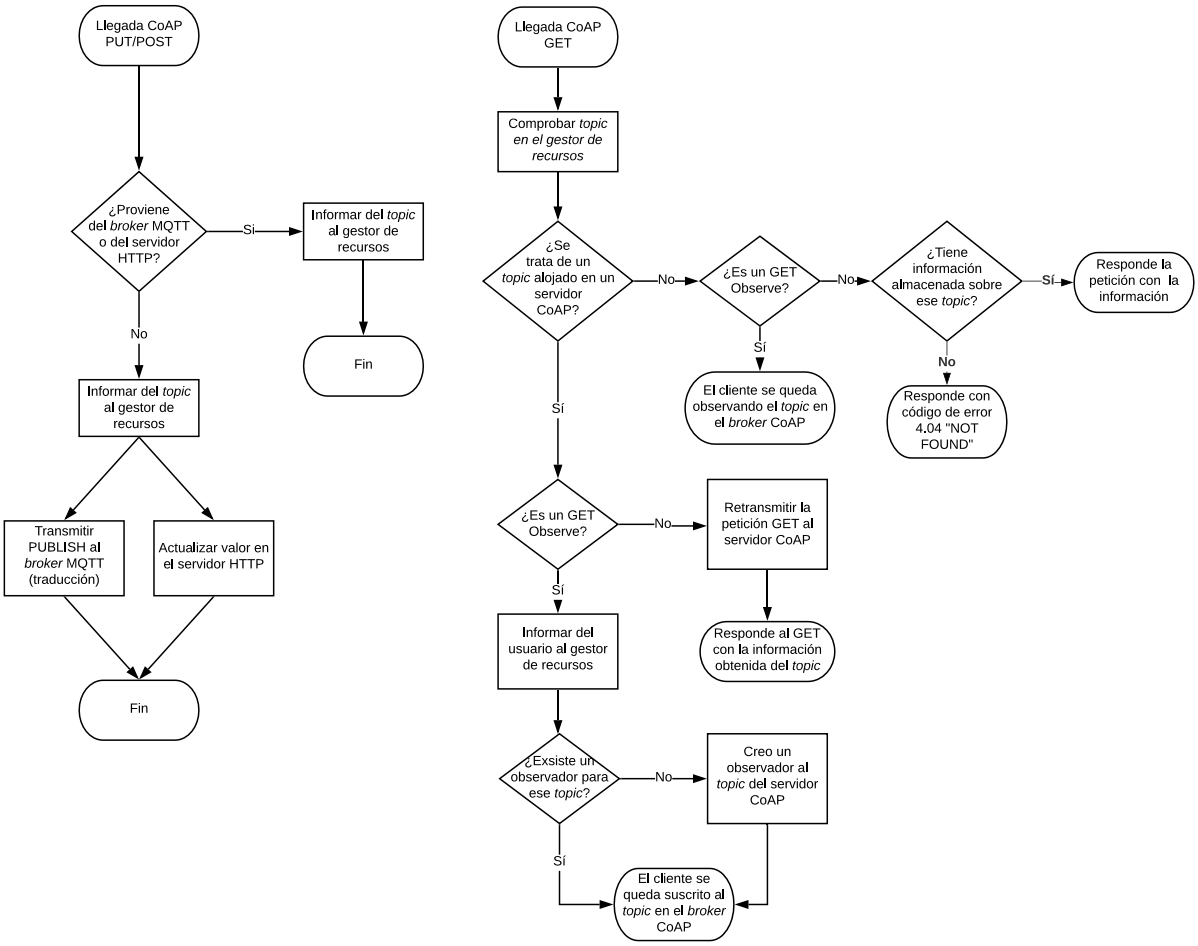


Figura 20. Diagrama de flujo de la interfaz CoAP

En la Figura 20, se observa el diagrama de flujo de la interfaz CoAP para los siguientes casos:

- Recepción de un mensaje CoAP PUT/POST: en este caso, si el mensaje proviene de un cliente CoAP PUT/POST, el *Protocolo de comunicación CoAP* informa al *Gestor de recursos* de la recepción del mensaje dirigido al *topic*. El gestor actualiza la información de control sobre el *topic* y a continuación se envía un mensaje MQTT PUBLISH con los datos del *topic* a la interfaz MQTT. Cuando esta interfaz lo recibe, se encarga de notificar a los clientes MQTT suscritos ese *topic*. Adicionalmente, se actualiza el servidor HTTP con dichos datos para poder acceder a ellos a través de

peticiones HTTP GET. Por otro lado, si el mensaje CoAP PUT/POST ha sido generado por el *broker* MQTT o por el servidor HTTP, no se hace nada, puesto que es fruto de la traducción de un mensaje MQTT PUBLISH o HTTP PUT/POST y el *broker* CoAP únicamente notifica a los clientes CoAP que tienen una relación de observación establecida con ese *topic*.

- Recepción de un mensaje CoAP GET: en primer lugar, se consulta la información sobre ese *topic* en el *Gestor de recursos*. Si se trata de un *topic* alojado en un servidor CoAP existen dos posibilidades: a) si se trata de una petición GET simple, se retransmite la petición al servidor y con la información obtenida se responde a la petición del cliente, o b) si se trata de una petición GET Observe, se notifica al *Gestor de recursos* sobre el usuario observador. Esto permite llevar la cuenta de los usuarios que están observando. A continuación, se comprueba a través del *Gestor de recursos* si ya existe un *Observador* creado a dicho *topic* alojado en el servidor CoAP. Si ya existe uno, el gestor únicamente deja al cliente CoAP observando el *topic* en el *broker* CoAP. En caso de que no exista un *Observador*, lo crea y deja al cliente CoAP suscrito al *topic* en el *broker* CoAP. En caso de que no sea un *topic* alojado en un servidor CoAP existen dos posibilidades: a) si es una petición GET simple se consulta si se dispone de información almacenada (publicada mediante un mensaje MQTT PUBLISH, CoAP PUT/POST o HTTP PUT/POST). En caso afirmativo se contesta con dicha información y en caso negativo se contesta con el código de error 4.04 NOT FOUND, o b) si se trata de una petición GET Observe se deja al cliente observando el *topic* en el *broker* CoAP.

Protocolo de comunicación HTTP

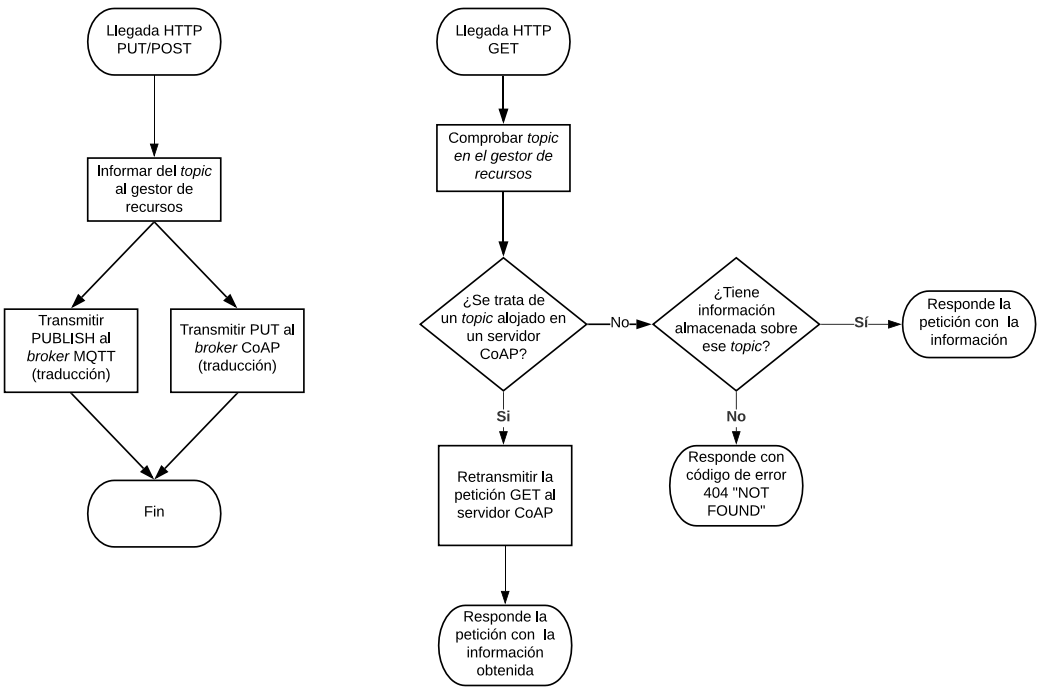


Figura 21. Diagrama de flujo de la interfaz HTTP

En la Figura 21, se observa el diagrama de flujo de la interfaz HTTP para los siguientes casos:

- Recepción de un mensaje HTTP PUT/POST: en este caso, el *Protocolo de comunicación HTTP* informa al *Gestor de recursos* de la recepción del mensaje dirigido al *topic*. El gestor actualiza la información de control sobre el *topic* y a continuación se envía un mensaje MQTT PUBLISH con los datos del *topic* a la interfaz MQTT. Cuando esta interfaz lo recibe, se encarga de notificar a los clientes MQTT suscritos ese *topic* y se envía un mensaje CoAP PUT con los datos del *topic* a la interfaz CoAP, que se encarga de notificar a los clientes que tienen una relación de observación establecida.
- Recepción de un mensaje HTTP GET: en primer lugar, se consulta la información sobre ese *topic* en el *Gestor de recursos*. Si se trata de un *topic* alojado en un servidor CoAP se retransmite la petición GET al servidor CoAP y con la información recibida se responde la petición original. En cambio, si no se trata de un *topic* alojado en un servidor CoAP existen dos opciones: a) si se dispone de información publicada a través de mensajes MQTT PUBLISH, CoAP PUT/POST o HTTP PUT/POST se contesta a la petición con dicha información, o b) si no existe información almacenada, se contesta con el código de error 404 NOT FOUND.

Adicionalmente, se definen dos elementos más: *Gestor de descubridores* y *Descubridor*. El primero de ellos dispone de una lista de servidores CoAP a los que se da servicio a través del *middleware*. Es el encargado de crear un proceso *Descubridor* por cada servidor CoAP. Se encarga de realizar peticiones CoAP GET a la URI */.well-known/core* del servidor. La respuesta que obtiene cada *Descubridor* contiene la lista de *topics* de los que dispone el servidor CoAP al que le consulta. Esta información se le comunica al *Gestor de recursos* para emplearse como información de control.

Por último, resulta interesante destacar los **mecanismos de cifrado introducidos en el *middleware***. A diferencia de Ponte, el sistema desarrollado permite la creación de canales de comunicación seguros a través DTLS y TLS. En este caso, las interfaces CoAP, MQTT y HTTP permiten dos tipos de conexiones diferentes: conexiones sin cifrar y conexiones cifradas. En el caso de las conexiones cifradas, en la interfaz CoAP se ha integrado el uso de DTLS sobre UDP y en el caso de las interfaces MQTT y HTTP se ha integrado el uso de TLS sobre TCP. En los tres casos se utilizan certificados digitales como medio de autenticación del servidor y del cliente, aunque también es posible el uso de una clave secreta preestablecida *(pre-shared key)* como medio de autenticación. La creación de los certificados digitales se detalla en el Anexo 2.

4.3. Solución *middleware* escalable

El gran aumento de las aplicaciones de publicación/recepción de datos y el consecuente crecimiento de las redes IoT, junto con el auge de la migración de los elementos arquitecturales a ubicaciones perimetrales de red, establece la necesidad de plantear soluciones escalables que permitan adaptarse al crecimiento continuo del número de usuarios de manera fluida. Cabe destacar que esta solución ha venido inspirada por el trabajo [34], en el cual proponen una arquitectura distribuida dinámica de *brokers* MQTT como solución a arquitecturas centralizadas en un único *broker*.

La solución inicial implementada en las secciones 4.1 y 4.2, establece un sistema estático que no permite desplegar de forma distribuida el *middleware* desarrollado. Por este motivo, en esta sección se exponen elementos y funcionamientos añadidos sobre la arquitectura base, que permiten desplegar el *middleware* en más de un dispositivo garantizándose la comunicación entre ellos. Como ya se ha comentado antes, esta es una característica que se echa en falta en el *broker* Ponte y que se aborda a continuación.

El objetivo de querer garantizar la escalabilidad del *middleware*, implica directamente la necesidad de intercomunicar múltiples dispositivos en los que se instale dicho *middleware*, permitiendo así una arquitectura distribuida. Para realizar dicha comunicación entre dispositivos se ha elegido el protocolo MQTT, debido a que es un protocolo orientado a entornos *publish/subscribe* cuyo funcionamiento resulta ideal para cumplir con este objetivo. El diseño del *middleware* distribuido se basa en una arquitectura en forma de árbol donde los nodos de la red se configuran de forma estática evitando lazos cerrados entre ellos.

Partiendo de la arquitectura base del *middleware* expuesta en el apartado 4.2, y teniendo en cuenta que el elemento *Gestor de recursos* es el encargado de almacenar la información de control para el correcto funcionamiento de los diferentes *Protocolos de comunicación*, se añaden dos listas nuevas: una lista que contiene la dirección IP y el ID del resto de *middlewares* a los que cada uno se conecta y una lista de *topics* externos en el *Gestor de recursos.* Esta nueva lista de *topics,* almacena la información de los *topic* que son publicados en otros *middlewares* distribuidos que forman la red. Esta es la única información adicional que se precisa para cumplir con el objetivo propuesto y que, a continuación, se muestra cómo se emplea y actualiza de forma dinámica en los diferentes casos de uso implicados.

- Recepción de un mensaje PUBLISH MQTT: en primer lugar, se comprueba el emisor del mensaje mediante el *ClientId* del mensaje MQTT y la lista de dispositivos que almacena el *Gestor de recursos.* Si el mensaje proviene de un dispositivo *middleware* distribuido, el *Protocolo de comunicación MQTT* informa al *Gestor de recursos* de dicho *topic* y de la fuente de origen. Adicionalmente el *Protocolo de comunicación MQTT* propaga el mensaje PUBLISH a los siguientes dispositivos que ejecutan el *middleware*; de esta forma se distribuye la información por toda la arquitectura distribuida. Por otro lado, si el mensaje proviene de un cliente MQTT *publisher* y es la primera vez que se recibe una publicación sobre

dicho *topic*, el *Protocolo de comunicación MQTT* propaga el mensaje PUBLISH a todos los dispositivos distribuidos de la lista; de esta forma se actualiza la información en todos. El funcionamiento comentado se representa en la Figura 22, en azul se resaltan los bloques añadidos frente al diagrama de flujo del sistema no escalable.

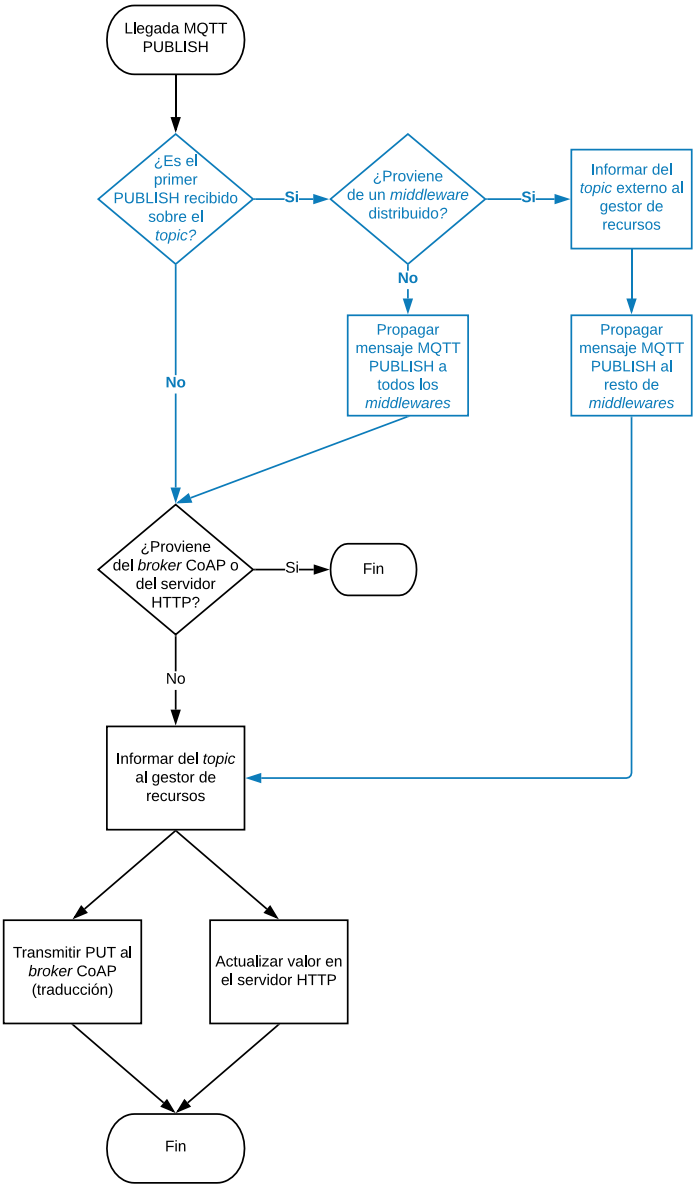


Figura 22. Diagrama de flujo de la recepción de un mensaje MQTT PUBLISH (middleware escalable)

- Recepción de un mensaje PUT/POST CoAP o HTTP: en este caso, el mensaje únicamente puede haber sido enviado por un cliente CoAP o HTTP. El *Protocolo de comunicación CoAP o HTTP*, consulta en el *Gestor de recursos* si es la primera vez que se recibe una publicación sobre el *topic* al que va dirigido el mensaje. En caso afirmativo se envía un mensaje MQTT PUBLISH al resto de dispositivos que ejecutan el

middleware; de esta forma se les informa sobre la existencia dicho *topic* a todos. El funcionamiento comentado se representa en la Figura 23, en azul se resaltan los bloques añadidos frente al diagrama de flujo del sistema no escalable.

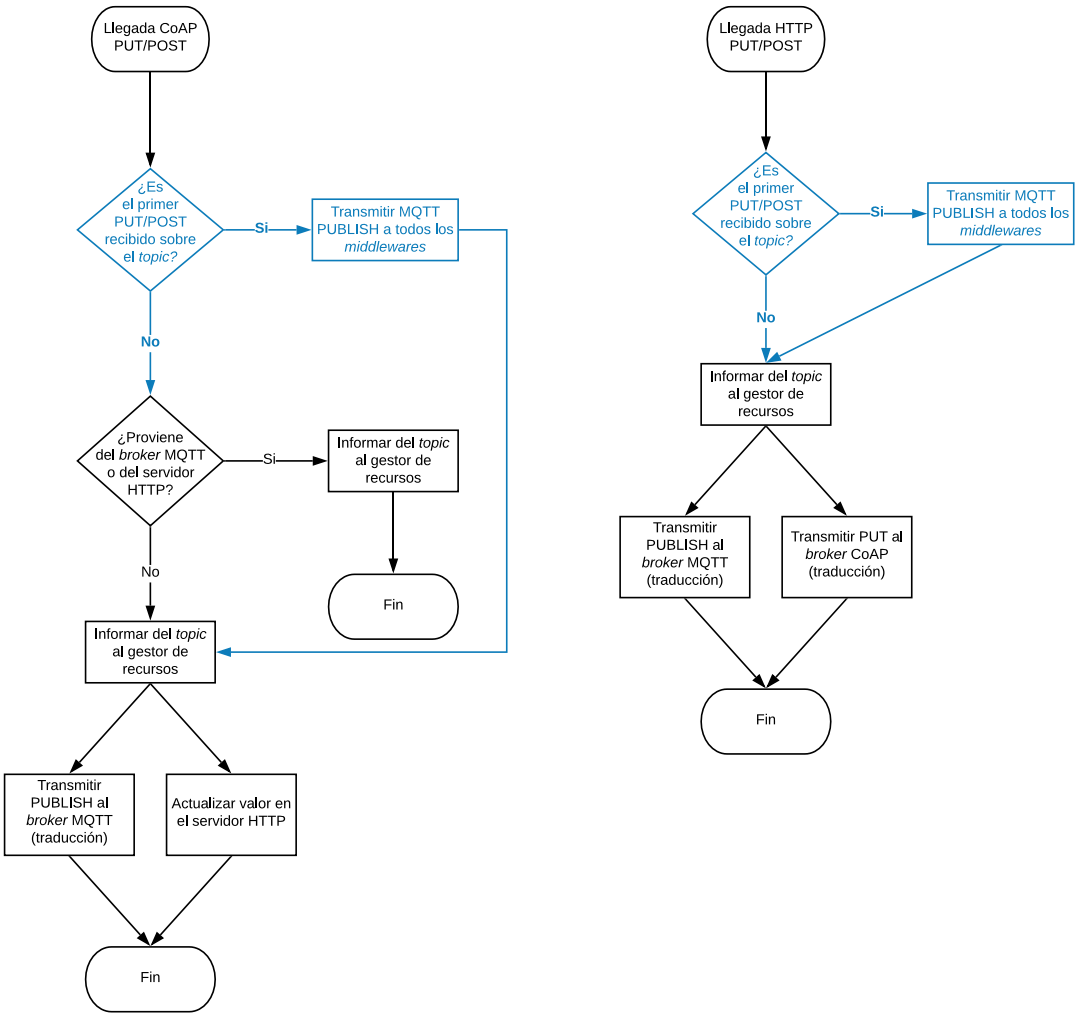


Figura 23. Diagrama de flujo de la recepción de un mensaje PUT/POST CoAP/HTTP (middleware distribuido)

- Recepción de un mensaje SUBSCRIBE MQTT o GET OBSERVE CoAP: en este caso el *Protocolo de comunicación MQTT o CoAP* consulta si ese *topic* se corresponde con un *topic* que ha sido publicado en otro dispositivo distribuido de la red que ejecuta el *middleware*, a través del *Gestor de recursos*. En caso afirmativo, el *Protocolo de comunicación MQTT o CoAP* crea un cliente MQTT SUBSCRIBER que se suscribe a dicho dispositivo sobre ese *topic*. Cada vez que reciba una notificación sobre ese *topic* se notifica a todos los clientes MQTT suscritos y a todos los clientes CoAP observadores. Es importante destacar que esto solo se realiza para la primera suscripción MQTT u observación CoAP que se recibe, lo que permite disminuir el tráfico generado. En caso de que se cancelen todas las suscripciones MQTT y todas las relaciones de observación CoAP sobre

ese *topic*, el *middleware* debe cancelar la suscripción MQTT a los demás dispositivos que ejecutan el *middleware*. En la Figura 24 y Figura 25 se muestra el comportamiento tanto frente a un mensaje MQTT SUBSCRIBE como a un mensaje CoAP GET OBSERVE respectivamente, resaltando en azul los bloques añadidos frente a los diagramas de flujo del sistema no escalable.

- Recepción de un mensaje CoAP GET (no observe) o HTTP GET: en este caso el *Protocolo de comunicación MQTT o CoAP* consulta si ese *topic* se corresponde con un *topic* que ha sido publicado en otro dispositivo distribuido de la red que ejecuta el *middleware*, a través del *Gestor de recursos*. En caso afirmativo, el *Protocolo de comunicación CoAP o HTTP* crea un cliente CoAP o un cliente HTTP que propaga la petición CoAP GET o HTTP GET hacia el dispositivo que ejecuta el *middleware* distribuido correspondiente. Por último, se responde a la petición original con la información obtenida. En la Figura 25 y Figura 26 se muestra el comportamiento frente tanto a un mensaje CoAP GET como a un mensaje HTTP GET respectivamente, resaltando en azul los bloques añadidos frente a los diagramas de flujo del sistema no escalable.

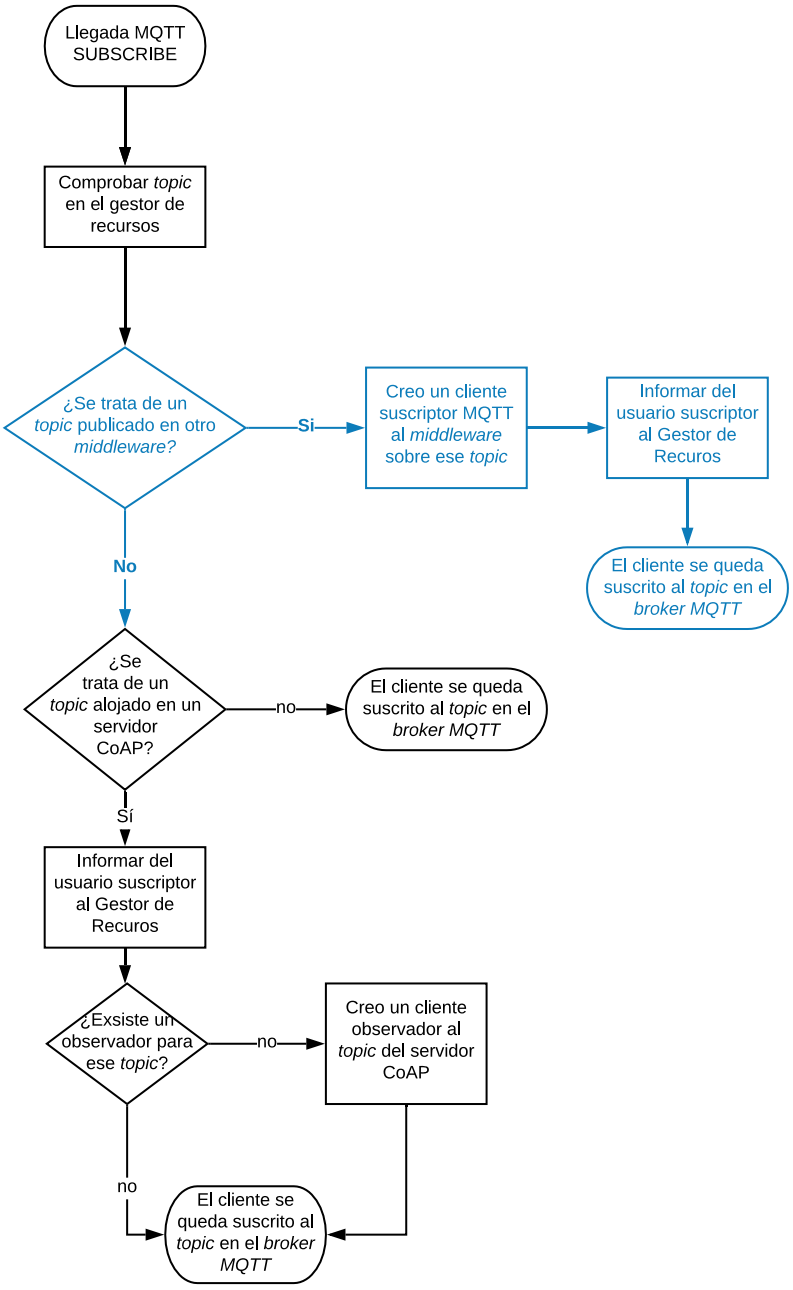


Figura 24. Diagrama de flujo de la recepción de un mensaje MQTT SUBSCRIBE (middleware distribuido)

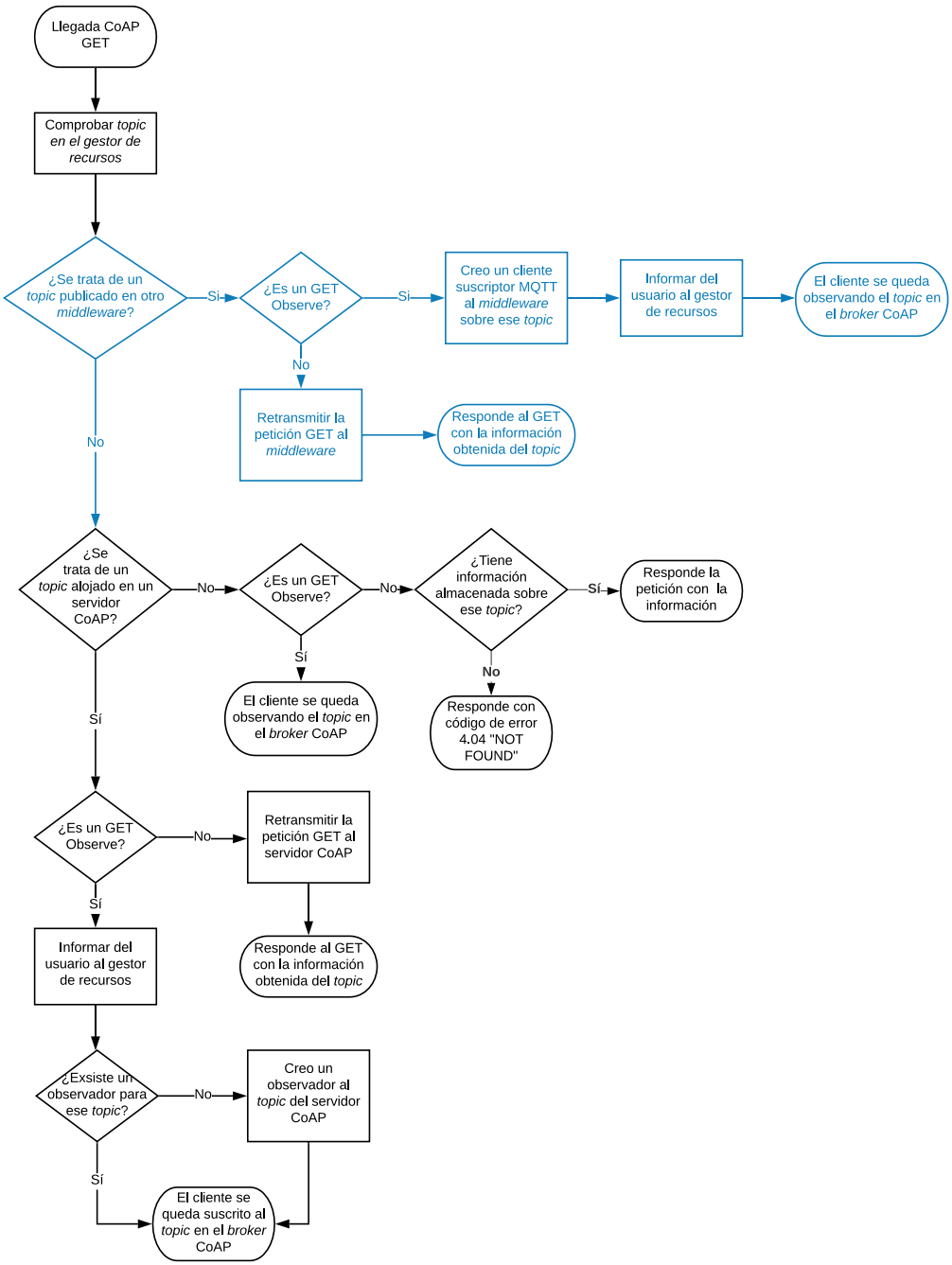


Figura 25. Diagrama de flujo de la recepción de un mensaje CoAP GET (middleware distribuido)

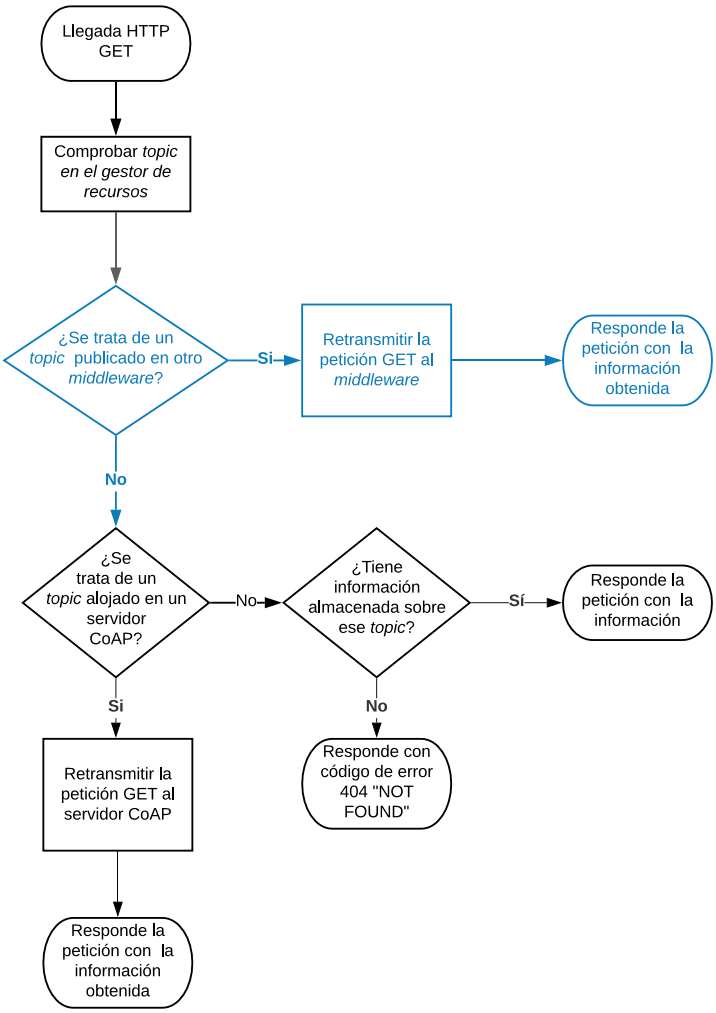


Figura 26. Diagrama de flujo de la recepción de un mensaje HTTP GET (middleware distribuido)

Adicionalmente, el elemento *Descubridor* que se encarga de descubrir los diferentes *topics* alojados en servidores CoAP, también genera un mensaje MQTT PUBLISH a cada dispositivo que ejecuta el *middleware* de forma distribuida. De esta manera también comunica la existencia de este tipo de *topics*.

Del mismo modo que se comentaba en el apartado 4.2, a fin de evitar conflictos con los nombres de los *topics*, se define una limitación en la configuración en los nombres de los *topics*: es necesario que los nombres de los *topics* que se publican en diferentes sistemas *middleware* distribuidos sean diferentes e independientes entre ellos.

Por último, la Figura 27 muestra un pequeño ejemplo de comunicación de clientes IoT ubicados en diferentes redes interconectados a través de la arquitectura de *middlewares* distribuida. El ejemplo está formado por un cliente MQTT que publica de datos sobre un *topic* en el *middleware* ubicado en la red 1. A su vez un cliente MQTT se suscribe a dicho *topic* y un cliente CoAP observa el *topic*, ambos sobre el *middleware* ubicado en la red 2. En la figura se observa por un lado la capacidad de intercomunicar clientes que trabajan con diferentes protocolos y también muestra la escalabilidad del *middleware*.

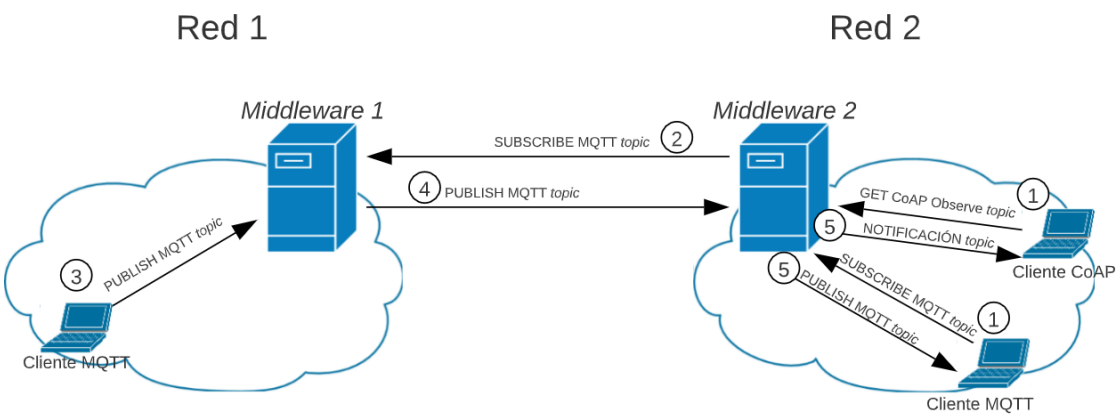


Figura 27. Ejemplo de comunicación distribuida

5. Descripción de los escenarios de aplicación

En este capítulo, se describen los escenarios de pruebas considerados en el trabajo, con los que se pretenden estudiar los diferentes aspectos, tanto positivos como negativos, que surgen al utilizar el sistema *middleware* desarrollado y explicado en el Capítulo 4.

Se proponen dos casos de estudio. El primero de ellos se basa en el análisis de la sobrecarga añadida que aparece al utilizar el *middleware* en caso de ubicarlo en un centro de datos *cloud*, siguiendo el modelo de computación para IoT en el que los datos se transmiten a recursos computacionales de un centro de datos remoto, donde se dispone típicamente de máquinas sin limitaciones. El segundo de ellos, acercándonos a las tendencias actuales, se propone el estudio del *middleware* en caso de alojarlo en el borde ("*edge*" en inglés) de la red IoT, siguiendo el modelo de computación perimetral o e*dge computing* en inglés. En este último caso, los recursos computacionales suelen ubicarse cerca de donde se generan los datos, pero suelen estar limitados en cuanto a la memoria disponible, la capacidad de la CPU o el almacenamiento.

5.1. Escenario 1: *Middleware* ubicado en la nube

Para el primer escenario se propone la topología de red mostrada en la Figura 28. Como se puede observar, la red de sensores y el *middleware* se encuentran ubicados en localizaciones diferentes.

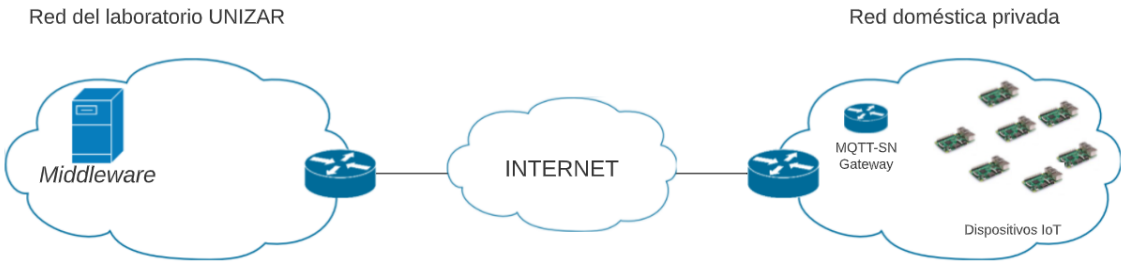


Figura 28. Arquitectura IoT escenario 1

La red de sensores se encuentra en una red local privada doméstica, los dispositivos empleados para los clientes/servidores IoT son *Raspberry Pi 3 model B* que disponen de un procesador *Quad Core* 1.2GHz *y* 1GB de memoria RAM, características más que suficientes para aplicaciones IoT.

Entre los dispositivos IoT, para obtener los resultados se han implementado los siguientes:

- Cliente MQTT PUBLISHER
- Cliente MQTT SUBSCRIBER
- Cliente MQTT-SN PUBLISHER

- Cliente MQTT-SN SUBSCRIBER
- Cliente CoAP PUT
- Cliente CoAP GET y GET Observe
- Servidor CoAP
- Cliente HTTP GET
- Cliente HTTP PUT

Como se ha explicado en la sección 2.2.2, los clientes MQTT-SN se comunican a través del MQTT-SN Gateway, que también se ejecuta sobre una *Raspberry Pi 3 model B.*

El *middleware* se encuentra en un equipo ubicado en la red del laboratorio de investigación de la Universidad de Zaragoza. El dispositivo empleado se trata de una estación de trabajo Workstation Intel Xeon SkyLake-SP 3106 dual con 480GB de disco SSD, 8TB de disco duro y 128 GB de memoria RAM. Con esta configuración se pretende simular un entorno *cloud* sin limitación de recursos en el que el tiempo de procesamiento del *middleware* no se vea afectado negativamente por los recursos del dispositivo.

Respecto al *software* utilizado para los dispositivos IoT, se han empleado las siguientes librerías para desarrollar las aplicaciones de los clientes.

- CoAP PubSub (java) [56]para los clientes CoAP.
- Californium CoAP y CoAP PubSub (java) [12] [56] para el servidor CoAP.
- Moquette (java) [37] para el *broker* MQTT
- Eclipse Paho MQTT (java) [14] para los clientes MQTT.
- Mqtt-sn-tools (C) [22] para los clientes MQTT-SN.
- Eclipse Paho MQTT-SN Embedded (C) [20] para el GW MQTT-SN.
- Oracle HTTP (java) para los clientes y servidor HTTP.

5.2. Escenario 2: *Middleware* ubicado en el borde de la red

Para el segundo escenario, se propone la topología de red mostrada en la Figura 29. Como se puede observar, la red de sensores y el *middleware* está ubicado en la misma ubicación, siendo esta una red privada doméstica.

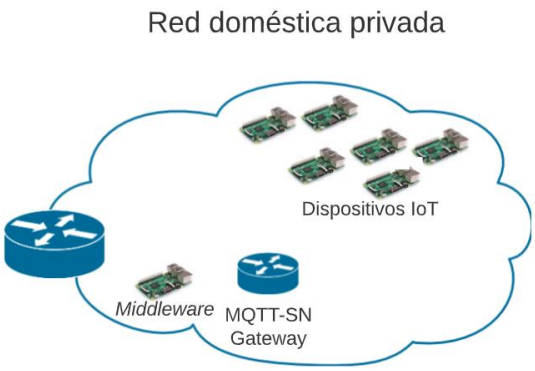


Figura 29. Arquitectura IoT escenario 2

En este caso, los dispositivos empleados son los mismos tanto para clientes/servidores IoT como para el *middleware*, son también *Raspberry Pi 3 model B*. A diferencia del Escenario 1, la limitación en recursos del dispositivo que aloja el *middleware* repercute directamente en la capacidad de cómputo de este y por ende en el retardo de las comunicaciones como se muestra en el Capítulo 6.

El *software* y *hardware* empleado para los dispositivos IoT es el mismo que el mencionado en el Escenario 1.

Con esta configuración se pretende simular un entorno de computación perimetral o *edge computing*, con infraestructura de red basada en dispositivos de recursos limitados. De esta forma se estudian los efectos de esta arquitectura frente a la arquitectura propuesta en el Escenario 1 en términos de retardo.

6. Resultados

El objetivo principal es estudiar la latencia extremo a extremo en un entorno de red real para los escenarios comentados en el Capítulo 5, haciendo hincapié en el retardo introducido por el *middleware* como resultado de la traducción de mensajes entre protocolos. Para ello se plantea la siguiente metodología.

Para poder medir el retardo extremo a extremo en comunicaciones de tipo *publish/subscribe*, es necesario muestrear el instante temporal en el que el cliente que genera datos envía el mensaje y el instante temporal en el que el cliente receptor lo recibe. Para ello, el cliente *publisher* transmite el *timestamp*[2] de *unix* capturado en el momento del envío del mensaje, por otro lado, el cliente *subscriber* captura el *timestamp* de *unix* en el momento de la recepción del mensaje y se calcula la diferencia. Es importante destacar que es fundamental tener los clientes sincronizados[3] temporalmente para que las medidas sean correctas.

Por otro lado, también resulta interesante poder medir el tiempo consumido en comunicaciones de tipo petición/respuesta, como pueden ser la interacción mediante HTTP GET o CoAP GET. En este caso, no se mide el retardo extremo a extremo, sino que se mide el tiempo de ida y vuelta de la combinación petición-respuesta, también conocido como RTT *(Round Trip Time)*. Para ello, el cliente registra el *timestamp* de *unix* en el momento del envío de la petición y se calcula la diferencia con el *timestamp* del momento de la recepción de la respuesta.

La Figura 30 muestra la metodología seguida durante la medición del retardo extremo a extremo.

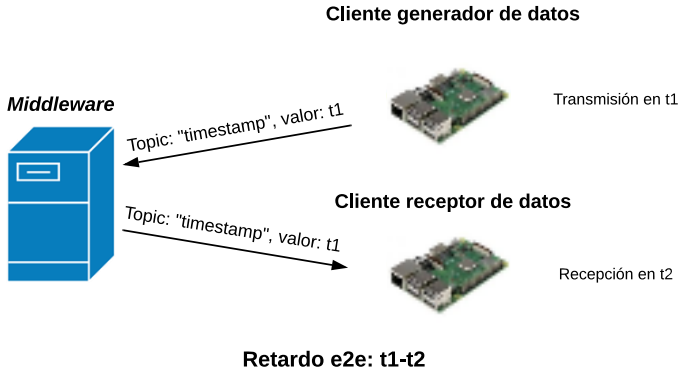


Figura 30. Esquema de medición del retardo extremo-extremo

[2] *Timestamp:* marca temporal de *unix*, se define como la cantidad de segundos transcurridos desde el 1 de enero de 1970.

[3] Para sincronizar diferentes máquinas bajo un mismo reloj se emplea el protocolo NTP *(Network Time protocolo) [36]* o el protocolo PTP *(Precision Time Protocol) [48]* en aplicaciones de mayor precisión.

Adicionalmente, se pretende calcular de forma aislada el tiempo de procesado interno en el *middleware* de cada tipo de comunicación, tanto para comunicaciones directas entre protocolos como para comunicaciones en las que interviene la traducción de protocolos. Para ello se captura en la interfaz de red del dispositivo en el que se ejecuta el *middleware*. Este estudio, se centra comunicaciones *publish /subscribe*, ya que son más habituales en aplicaciones orientadas a comunicaciones en tiempo real. Para ello se toma como referencia el instante temporal en el que se recibe el mensaje del cliente *publisher* y se calcula la diferencia con el instante temporal en el que se captura el envío de la notificación a los clientes *subscriber*.

6.1. Resultados escenario 1

Con el objetivo de verificar el funcionamiento del sistema y de obtener medidas representativas de su utilización, y dada la gran variedad de posibles combinaciones disponibles en las comunicaciones IoT, se han propuesto diferentes casos en función de los diferentes niveles de QoS y para comunicaciones cifradas y no cifradas.

Tiempo medio extremo-extremo y RTT

En primer lugar, **para comunicaciones no cifradas y niveles de QoS 0 y 1** respectivamente, se comparan los distintos retardos extremo a extremo medios en comunicaciones del **tipo publicación/suscripción**. Para ello se definen: un cliente MQTT Suscriptor, un cliente MQTT-SN Suscriptor y un cliente CoAP observador y se procede a publicar datos mediante: un cliente MQTT Publisher, un cliente MQTT-SN Publisher, un cliente CoAP PUT, un servidor CoAP y un cliente HTTP PUT respectivamente. Con el fin de ser equitativo entre protocolos, la configuración de QoS 0 de MQTT/MQTT-SN se compara con CoAP usando mensajes NON y la configuración de QoS 1 de MQTT/MQTT-SN con CoAP usando mensajes CON. Como HTTP no diferencia niveles de QoS, solo se compara con la configuración QoS 0 de MQTT/MQTT-SN y mensajes NON de CoAP. Los resultados se muestran en la Figura 31 y Figura 32.

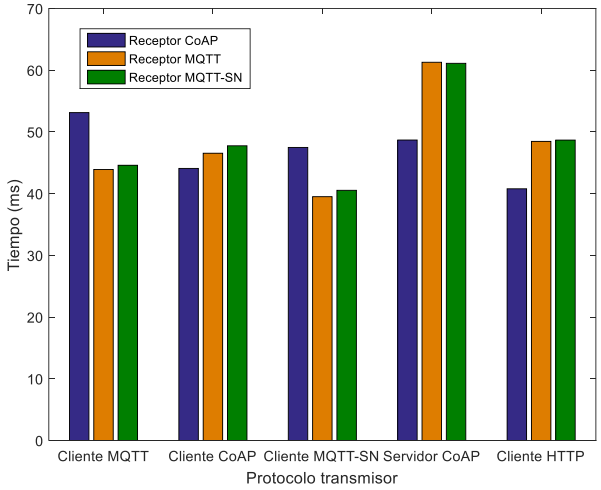


Figura 31. Retardo medio extremo-extremo QoS 0 (MQTT/MQTT-SN) y mensajes NON (CoAP) sin TLS/DTLS, escenario 1

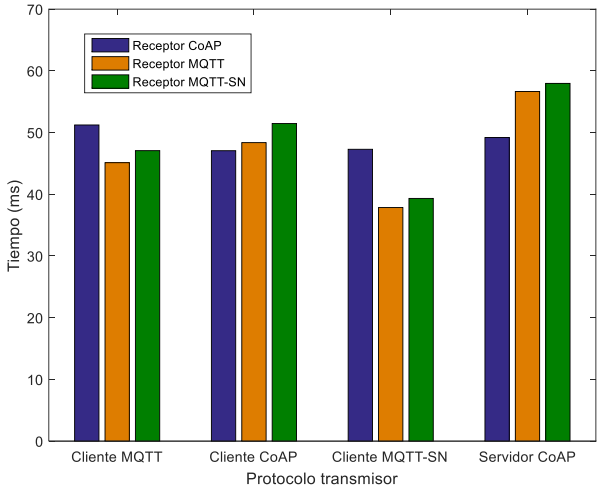


Figura 32. Retardo medio extremo-extremo QoS 1 (MQTT/MQTT-SN) y mensajes CON (CoAP) sin TLS/DTLS, escenario 1

Siguiendo la misma metodología, se realizan las mismas pruebas **para comunicaciones cifradas y niveles de QoS 0 y 1.** En este caso no se incluyen resultados para clientes MQTT-SN puesto que no cuentan con implementación TLS. Los resultados se muestran en la Figura 33 y Figura 34.

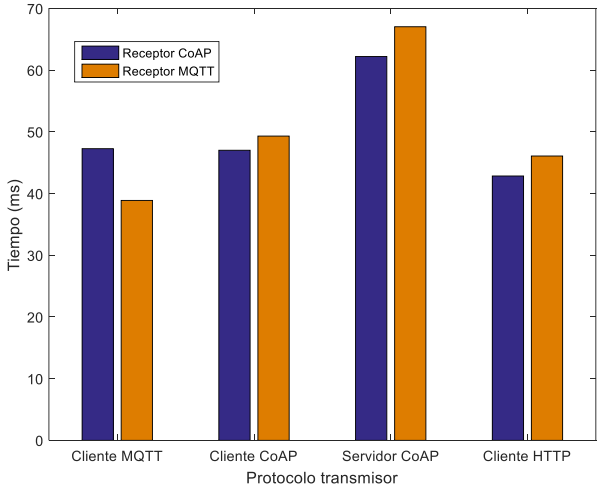


Figura 33. Retardo medio extremo-extremo QoS 0 (MQTT/MQTT-SN) y mensajes NON (CoAP) con TLS/DTLS, escenario 1

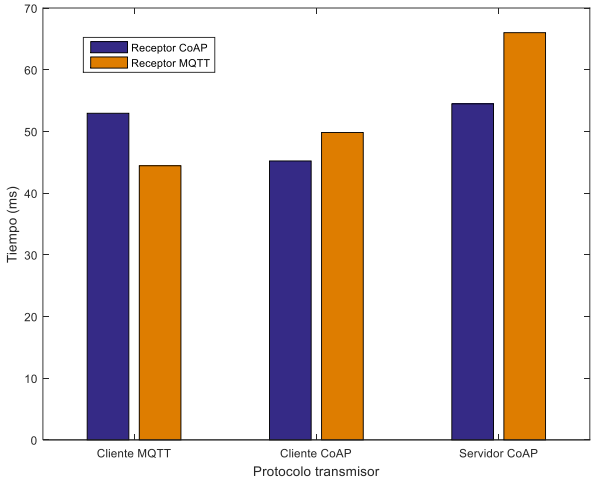


Figura 34. Retardo medio extremo-extremo QoS 1 (MQTT/MQTT-SN) y mensajes CON (CoAP) con TLS/DTLS, escenario 1

De este primer estudio, se puede ver una tendencia en cuanto a la diferencia de tiempos. En primer lugar, en todos los casos se puede observar una diferencia de tiempos clara en las comunicaciones que requieren traducción de protocolos (en torno a 10-12 milisegundos de diferencia) como pueden ser:

- Transmisor Cliente CoAP y receptores MQTT o MQTT-SN
- Transmisor Servidor CoAP y receptores MQTT o MQTT-SN
- Transmisor MQTT y receptor CoAP
- Transmisor MQTT-SN y receptor CoAP
- Transmisor HTTP y receptores CoAP, MQTT y MQTT-SN

Se observa además que el retardo es mayor para el caso en el que se transmite mediante un servidor CoAP. Esto es así porque la observación del *topic* en el servidor

CoAP se realiza mediante un cliente CoAP integrado en el *middleware*. Esto hace que esta comunicación y su posterior traducción interna del *middleware* añadan un retardo adicional. En el resto de los casos, la diferencia entre retardos se debe únicamente al tiempo de procesado empleado por el *middleware* durante la traducción de protocolos.

Un aspecto a destacar es la escasa diferencia en el retardo experimentado por los receptores MQTT y MQTT-SN, siendo de 1 o 2 milisegundos, esto es debido a que la comunicación fuera de la red de sensores entre el *middleware* y el MQTT-SN Gateway se realiza mediante MQTT.

Por último, resulta importante destacar que, en la mayoría de los casos, la utilización del mismo protocolo en transmisión y en la recepción resulta más eficiente en términos de retardo, como era de esperar, siendo la mejor opción en términos de retardo el protocolo MQTT-SN.

Por otro lado, se analizan también las comunicaciones de **tipo petición/respuesta en comunicaciones sin cifrar y cifradas** para los clientes HTTP y CoAP obteniendo información a través de peticiones GET.

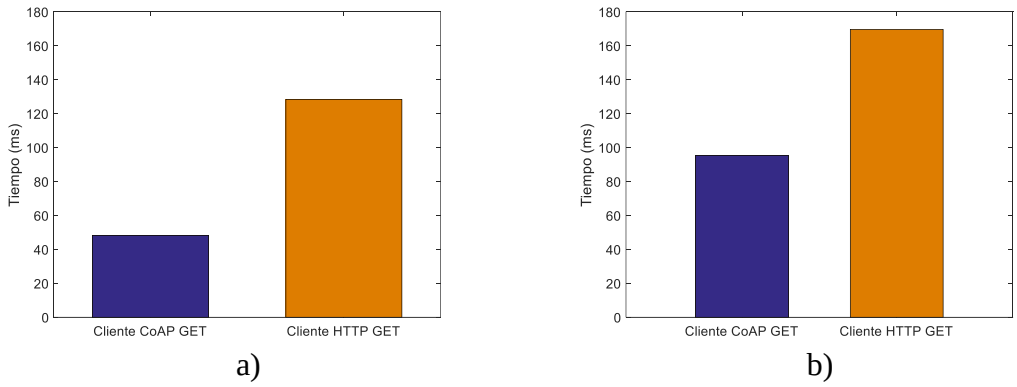


Figura 35. Retardo medio RTT para una petición GET a un topic en memoria (a), y a un recurso alojado en un servidor CoAP (b) sin TLS/DTLS, escenario 1

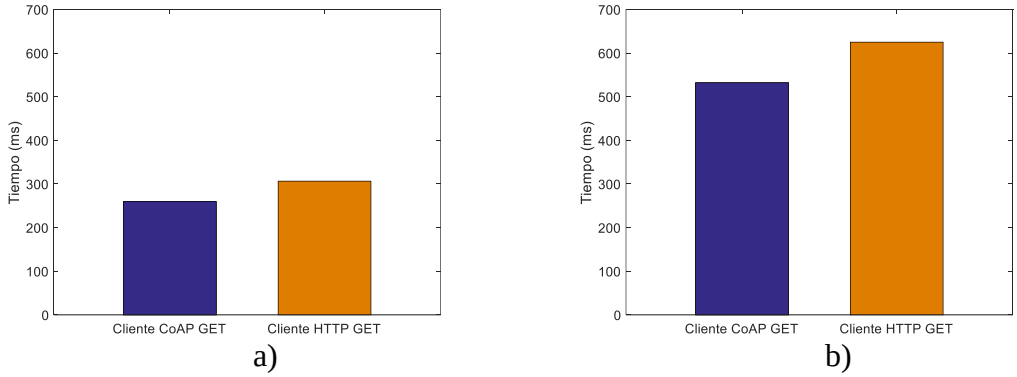


Figura 36. Retardo medio RTT para una petición GET a un topic en memoria (a), y a un recurso alojado en un servidor CoAP (b) con TLS/DTLS, escenario 1

Como se ve en la Figura 35 y la Figura 36, se realizan peticiones GET desde clientes CoAP y HTTP dirigidas al *middleware*. En las imágenes de la izquierda (Figura 35 a) y Figura 36 a)), el recurso se ha publicado desde clientes *publisher* tipo MQTT, MQTT-SN, CoAP PUT/POST o HTTP PUT/POST, por lo que la última información recibida se encuentra almacenada en memoria y el *middleware* contesta directamente con ella. En las imágenes de la derecha (Figura 35 b) y Figura 36 b)), se trata de un recurso alojado en un servidor CoAP, por lo que de forma proactiva el *middleware* retransmite la petición GET al servidor para poder acceder a la información.

Como era de esperar en el segundo caso (Figura 35 b) y Figura 36 b)), el retardo es mayor para ambos protocolos puesto que el *middleware* no contesta con la petición directamente, sino que la retransmite y espera a recibir la información para contestar. En ambos casos, el retardo RTT medio es menor para el cliente CoAP frente al cliente HTTP. Esto no es debido a la traducción de protocolos en sí, sino a la diferencia de funcionamiento entre ambos protocolos. CoAP funciona sobre UDP y no está orientado a conexión frente a HTTP que funciona sobre TCP y en cada petición GET debe establecer la conexión TCP. Esto incrementa el número de paquetes y bytes transmitidos y por ende el retardo total.

En el caso de comunicaciones cifradas, la diferencia entre CoAP y HTTP se reduce drásticamente puesto que al emplear CoAP sobre DTLS, este establece la conexión segura mediante un *handshake* inicial muy similar al de HTTP sobre TLS, por lo que el número de mensajes intercambiados y de bytes transmitidos se asemeja.

Tiempo medio de procesado en el *middleware*

Tanto los retardos medios extremo a extremo como los retardos RTT han sido medidos en un escenario de red real. Esto nos permite tener una idea del comportamiento del sistema, pero no de forma estricta, puesto que el estado de la red puede variar de una realización a otra y las medidas pueden sufrir variaciones. Por esta razón, como medida más precisa se pretende aislar el tiempo de procesado interno del *middleware* a partir de capturas de Tcpdump realizadas en la máquina en la que dicho *middleware* se ejecuta. La Tabla 5 muestra un resultado más preciso del tiempo empleado por el *middleware* en procesar cada tipo de mensaje en las diferentes configuraciones.

		Tiempo (ms) de procesado en comunicaciones *publish/subscribe*			
		Sin cifrado		Con cifrado	
Transmisor	**Receptor**	**QoS 0 y NONs**	**QoS 1 y NONs**	**Qos 0 y NONs**	**QoS 1 y NONs**
CoAP	**CoAP**	2,14	2,58	3,20	3,64
	MQTT	9,27	10,39	11,24	13,12
	MQTT-SN	9,35	10,72	-	-
MQTT	**CoAP**	10,74	11,45	10,05	12,49
	MQTT	1,74	2,13	2,21	3,16
	MQTT-SN	1,77	2,48	-	-
MQTT-SN	**CoAP**	10,15	10,81	-	-
	MQTT	1,64	2,11	-	-
	MQTT-SN	1,53	2,38	-	-
Servidor CoAP	**CoAP**	8,65	11,31	9,10	11,69
	MQTT	17,65	22,94	18,81	22,15
	MQTT-SN	17,94	23,29	-	-
HTTP	**CoAP**	4,34	-	5,07	-
	MQTT	14,95	-	14,42	-
	MQTT-SN	15,61	-	-	-

Tabla 5. Tiempo medio de procesado en comunicaciones publish/subscribe, escenario 1

Tal y como se comentaba anteriormente, las comunicaciones que requieren traducción de protocolos incrementan el tiempo de procesado interno del *middleware* entre 8ms y 22ms, siendo las más elevadas en el caso de recibir información desde un servidor CoAP. Adicionalmente, se ve cómo el uso de un nivel de QoS 1 (MQTT/MQTT-SN) y mensajes CON (CoAP) no introduce grandes efectos negativos frente a emplear un nivel de calidad de servicio inferior, siendo la diferencia del orden de 1ms. Por último, emplear comunicaciones cifradas también introduce en media 1ms o 2ms de retardo extra debido a las operaciones de cifrado y descifrado realizadas.

Ancho de banda medio consumido en la red de sensores

Dada la flexibilidad que aporta el *middleware* a la hora de seleccionar un protocolo de transmisión de datos, para completar el estudio, se analiza el ancho de banda consumido por los nodos transmisores en la red de sensores, en términos de bytes por segundo. Para la realización de este estudio se han empleado en todos los casos clientes que publican datos cada 2 segundos cuya carga útil es de 13 bytes. Las Figura 37,Figura 38,Figura 39 yFigura 40 muestran los resultados para las diferentes configuraciones.

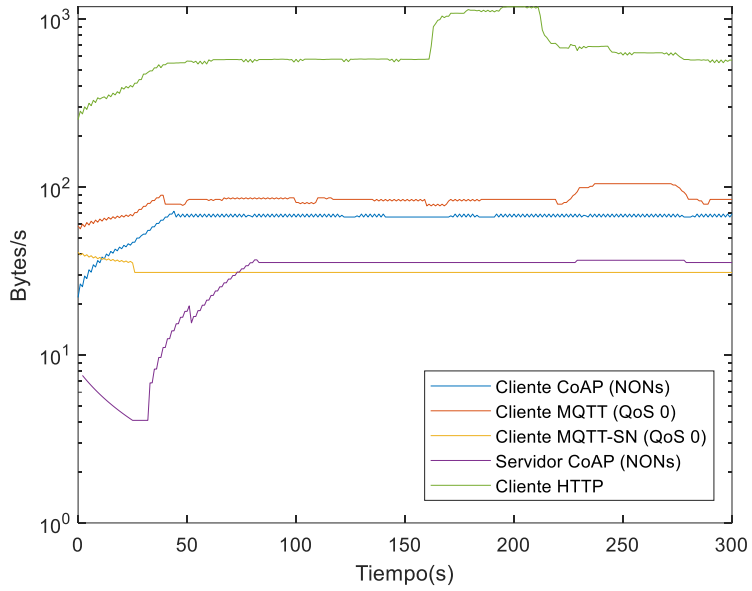


Figura 37. Ancho de banda medio consumido por los diferentes clientes con QoS 0 (MQTT/MQTT-SN) y mensajes NON (CoAP) y HTTP sin TLS/DTLS

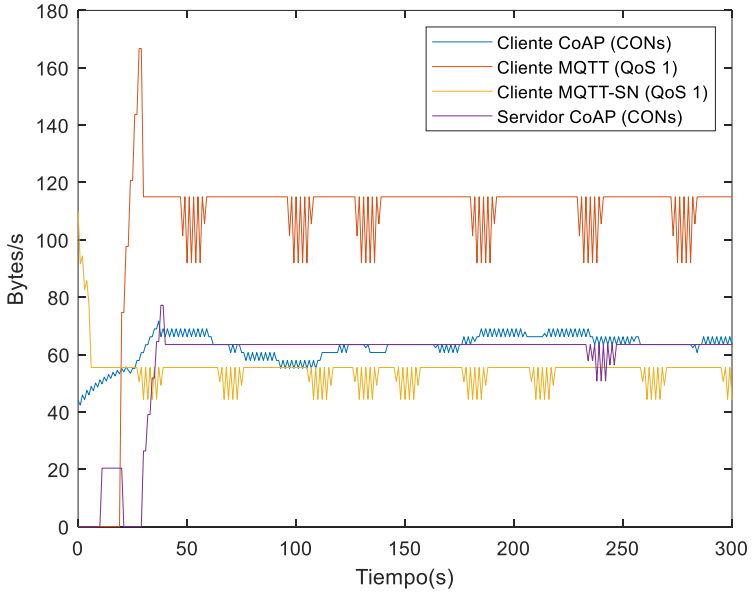


Figura 38. Ancho de banda medio consumido por los diferentes clientes con QoS 1 (MQTT/MQTT-SN) y mensajes CON (CoAP) sin TLS/DTLS

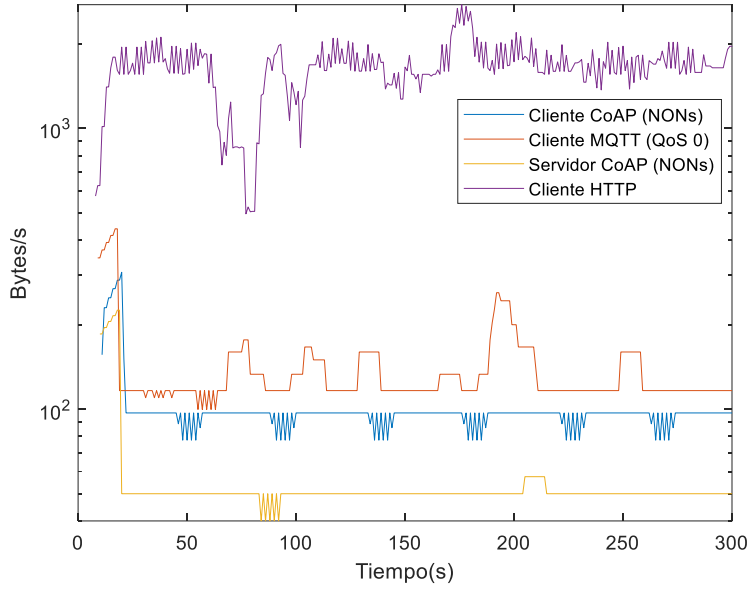


Figura 39. Ancho de banda medio consumido por los diferentes clientes con QoS 0 (MQTT) y mensajes NON (CoAP) y HTTP con TLS/DTLS

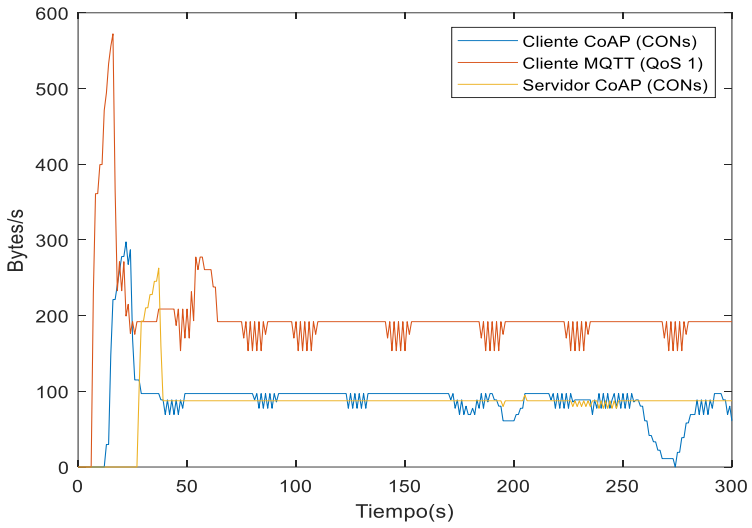


Figura 40. Ancho de banda medio consumido por los diferentes clientes con QoS 1 (MQTT) y mensajes CON (CoAP) con TLS/DTLS

En primer lugar, como era de esperar, el protocolo HTTP se trata el protocolo que más ancho de banda consume, siendo en media aproximadamente 10 veces mayor que el ancho de banda consumido por el resto de los protocolos. No se trata de un comportamiento anómalo puesto que HTTP no fue diseñado para aplicaciones IoT, sino para aplicaciones de transferencia de información más complejas y sin restricciones ni energéticas ni de ancho de banda.

En segundo lugar, el protocolo que más recursos consume se trata de MQTT tanto en configuración de QoS 0 o 1. Tampoco es de extrañar, ya que emplea como protocolo de transporte TCP, lo que hace que el tamaño de la cabecera de nivel de transporte sea

mayor a los protocolos que emplean UDP y además incluye la confirmación de paquetes mediante mensajes ACK.

Por último, existe gran similitud en el ancho de banda consumido por el cliente MQTT-SN, cliente CoAP y servidor CoAP puesto que los tres funcionan sobre UDP como protocolo de transporte y el número y tamaño de los paquetes es similar tanto para configuraciones de QoS 0 / mensajes NON como para QoS 1 / mensajes CON, siendo el cliente MQTT-SN en media el que menos ancho de banda consume en comunicaciones no cifradas.

6.2. Resultados escenario 2

Mediante la realización de este escenario, se pretenden analizar el efecto en términos de retardo que supondría utilizar el *middleware* desarrollado en un dispositivo de bajos recursos como es una *Raspberry Pi 3*. Esto es de gran interés debido a la tendencia actual de acercar los nodos de cómputo al borde de la red *(Edge computing)* disminuyendo así el tiempo de transmisión en escenarios basados en computación en la nube *(cloud computing).*

Para poder realizar un análisis correcto, se han seleccionado los casos de uso más restrictivos, en este caso las **configuraciones con QoS 1 (MQTT/MQTT-SN) y mensajes CON (CoAP) con y sin cifrado**. En este caso, únicamente se han estudiado las comunicaciones de tipo *publish/subscribe* que, en principio, están destinadas a comunicaciones en tiempo real.

Tiempo medio extremo-extremo y RTT

Del mismo modo que en el Escenario 1, en la Figura 41 y Figura 42 se representan los retardos extremo-extremo experimentado en las comunicaciones. A primera vista se visualiza una diferencia de tiempos mucho mayor entre protocolos heterogéneos, siendo esta de entre 40 y 60 ms.

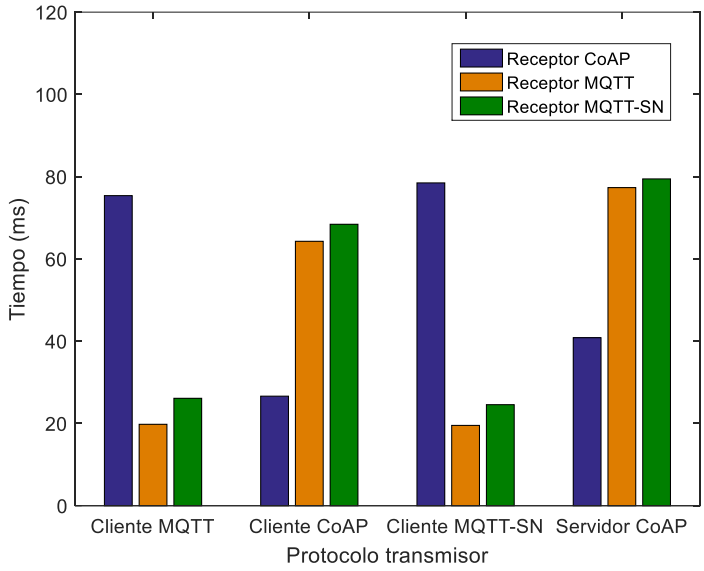


Figura 41. Retardo medio extremo-extremo QoS 1 (MQTT/MQTT-SN) y mensajes CON (CoAP) sin TLS/DTLS, escenario 2

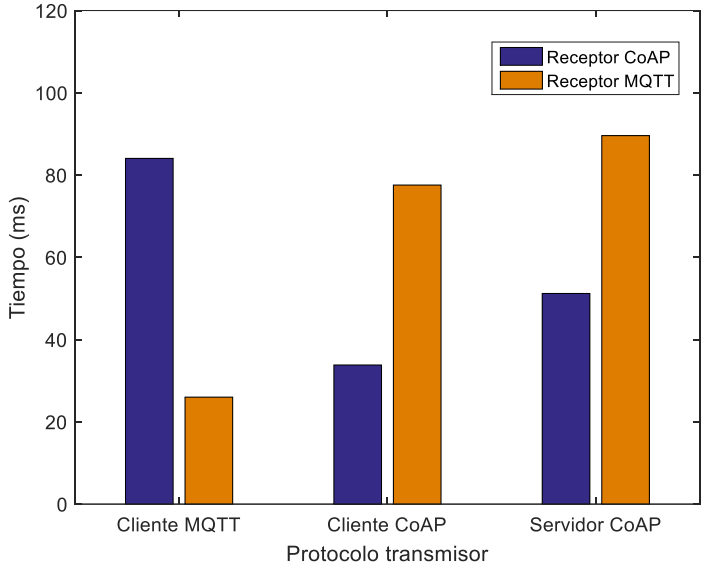


Figura 42. Retardo medio extremo-extremo QoS 1 (MQTT) y mensajes CON (CoAP) con TLS/DTLS, escenario 2

A continuación, la Figura 43 y Figura 44 muestran una comparativa detallada entre los resultados obtenidos en el Escenario 1 con los obtenidos en este escenario en lo referente a cada protocolo de forma independiente.

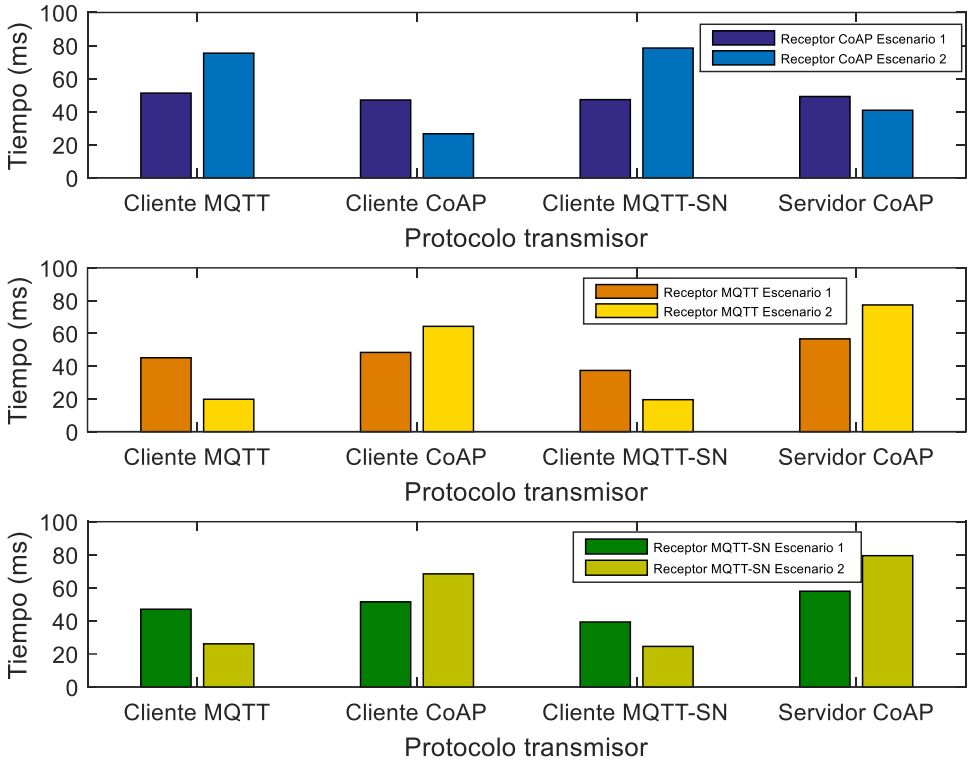


Figura 43. Comparativa de retado medio extremo-extremo entre el escenario 1 y escenario 2, sin TLS/DTLS

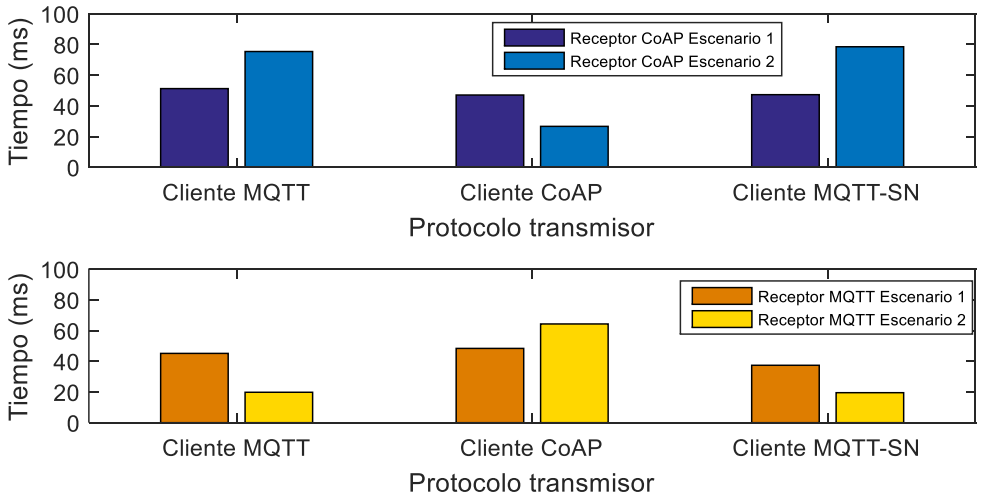


Figura 44. Comparativa de retado medio extremo-extremo entre el escenario 1 y escenario 2, con TLS/DTLS

Se observa una mejora en el retardo extremo a extremo en las comunicaciones en las que los extremos (transmisor y receptor) emplean el mismo protocolo, como, por ejemplo:

- Transmisor CoAP - Receptor CoAP
- Transmisor MQTT/MQTT-SN - Receptor MQTT/MQTT-SN

Esto resulta coherente puesto que el hecho de ubicar el *middleware* en el borde de la red de sensores hace que disminuya el retardo producido por el tiempo de transmisión de los datos hasta el *middleware* en caso de estar ubicado en la nube.

Por otro lado, el retardo extremo a extremo empeora en las comunicaciones en la que los extremos (transmisor y receptor) emplean protocolos diferentes, como, por ejemplo:

- Transmisor CoAP - Receptor MQTT/MQTT-SN
- Transmisor MQTT/MQTT-SN - Receptor CoAP

A primera vista, los resultados no resultan coherentes puesto que el objetivo de ubicar el *middleware* en el borde de la red de sensores es disminuir el retardo producido por el tiempo de transmisión de los datos hasta el *middleware* y con ello el retardo extremo a extremo en la comunicación. A pesar de esto, emplear un dispositivo de recursos limitados influye en el tiempo que emplea el *middleware* en realizar la traducción de protocolos en comunicaciones heterogéneas, pudiendo llegar a ser más significativo que el tiempo de transmisión de los datos.

Tiempo medio de procesado en el *middleware*

Siguiendo la metodología realizada en el Escenario 1, el estudio del retardo extremo a extremo otorga una medida orientativa sobre el comportamiento del sistema, pero no se puede tomar como una medida representativa. Por ello, a continuación, se adjunta la Tabla 6 en la que se muestra el tiempo empleado por el *middleware* en tratar las comunicaciones, obtenido a partir de capturas de Tcpdump.

		Tiempo (ms) de procesado en el *middleware*	
Transmisor	**Receptor**	**QoS 1 y CONs Sin cifrado**	**QoS1 y CONs Con cifrado**
CoAP	**CoAP**	13,60	14,58
	MQTT	55,41	59,88
	MQTT-SN	58,61	-
MQTT	**CoAP**	62,22	72,60
	MQTT	7,69	12,05
	MQTT-SN	10,86	-
MQTT-SN	**CoAP**	61,47	-
	MQTT	8,26	-
	MQTT-SN	10,79	-
Servidor CoAP	**CoAP**	23,72	27,81
	MQTT	61,46	66,96
	MQTT-SN	63,84	-

Tabla 6. Tiempo medio de procesado en comunicaciones publish/subscribe, escenario 2

A partir de los resultados representados en la Tabla 6, se puede afirmar que, en este caso, en lo referente a comunicaciones heterogéneas, implantar el *middleware* en el borde de la red de sensores en un dispositivo de bajos recursos *hardware* puede resultar perjudicial en términos de retardo en las comunicaciones puesto que el tiempo de procesado predomina frente al tiempo de transmisión.

Por último, en la Tabla 7 se muestra de manera cuantitativa la diferencia en tiempo de procesado entre el Escenario 1 y el Escenario 2.

		QoS 1 y CONs Sin cifrado			QoS 1 y CONs Con cifrado		
		Tiempo (ms)		Factor de proporción	Tiempo (ms)		Factor de proporción
Transmisor	**Receptor**	**Escenario 1**	**Escenario 2**		**Escenario 1**	**Escenario 2**	
CoAP	**CoAP**	2,58	13,60	x 5,3	3,64	14,58	x 4,0
	MQTT	10,39	55,41	x 5,3	13,12	59,88	x 4,6
	MQTT-SN	10,72	58,61	x 5,5	-	-	-
MQTT	**CoAP**	11,45	62,22	x 5,4	12,49	72,60	x 5,8
	MQTT	2,13	7,69	x 3,6	3,16	12,05	x 3,8
	MQTT-SN	2,48	10,86	x 4,4	-	-	-
MQTT-SN	**CoAP**	10,81	61,47	x 5,7	-	-	-
	MQTT	2,11	8,26	x 3,9	-	-	-
	MQTT-SN	2,38	10,79	x 4,5	-	-	-
Servidor CoAP	**CoAP**	11,31	23,72	x 2,1	11,69	27,81	x 2,4
	MQTT	22,94	61,46	x 2,7	22,15	66,96	x 3,0
	MQTT-SN	23,29	63,84	x 2,7	-	-	-

Tabla 7. Comparativa entre tiempos de procesado por el middleware entre el escenario1 y escenario 2

Como se venía comentando, se ve claramente como en el Escenario 2 aumenta el tiempo empleado por el *middleware* para tratar los mensajes viéndose multiplicado en un factor 2 en el mejor de los casos o en un factor casi de 6 en el peor de los casos. Comentar que este comportamiento no es único de las comunicaciones en las que existe traducción de protocolos, sino que ocurre de manera casi homogénea en todas las comunicaciones para el mismo protocolo transmisor.

7. Conclusiones y líneas futuras

Tras el desarrollo del *middleware* y su respectivo análisis de características y prestaciones, se presentan las conclusiones obtenidas a partir del trabajo realizado. A continuación, se plantean las conclusiones y las posibles líneas de trabajos futuros con el fin de profundizar en estas tecnologías.

7.1. Conclusiones

Internet of Things ofrece la posibilidad de mantener comunicado cualquier dispositivo electrónico, pudiendo recibir y procesar información en tiempo real. Este nuevo paradigma plantea nuevos modos de toma de decisiones basados en esta disponibilidad de información y nuevas oportunidades para las empresas.

El informe *IoT 2020 Bussines Report* [51] publicado por la firma Schneider Electric, muestra como dos de cada tres empresas planeaban implementar soluciones IoT vía aplicaciones móviles ya en 2016, bajo la creencia de que esta tecnología aportara valor a sus negocios en términos de: creación de nuevas oportunidades de negocio, mejora de la eficiencia en su actividad y un incremento de los beneficios a largo plazo. Esto nos indica claramente quienes son los principales agentes impulsores de las tecnologías IoT, las pequeñas y medianas empresas (PYME), quienes a su vez también sufren las principales dificultades como la gran heterogeneidad en los protocolos de comunicación IoT.

Las PYME conforman la columna vertebral de la economía europea, sin embargo, no tienen la experiencia ni las habilidades suficientes en tecnologías heterogéneas tan utilizadas en IoT como son *cloud, edge o* HPC *(High Performance Computing)* o los diferentes protocolos de comunicaciones empleados (MQTT, CoAP, HTTP). Por esta razón, se considera que el sistema *middleware* propuesto a lo largo de este proyecto puede tener un gran impacto en este tipo de compañías, acercando y facilitando todo tipo de aplicaciones IoT a entornos en los que se cuenta con un menor potencial de recursos.

Bajo esta situación y como se ha visto a lo largo del documento, en este trabajo de fin de máster, se plantea la necesidad de soluciones *software* que permitan la comunicación heterogénea entre diferentes tecnologías IoT, y se ha implementado un sistema *middleware* que cubre dichas necesidades.

Para ello, se proponen varios objetivos parciales: familiarizarse con los principales protocolos de aplicación IoT y sus implementaciones *software*, el desarrollo de un *middleware* que permite la interconexión transparente entre ellos a partir de dichas herramientas *software*, y su posterior evaluación de prestaciones en escenarios basados en *cloud computing* y en *edge computing.*

Gracias a las diferentes librerías de código libre, se ha podido implementar el sistema *middleware* propuesto, viendo cómo es posible desplegar servicios de red que permiten homogeneizar las comunicaciones IoT en un ecosistema tan heterogéneo como es el actual. Esto otorga una gran flexibilidad a la hora de desplegar nuevos servicios IoT o unificar y reutilizar las aplicaciones y servicios IoT existentes.

Por otro lado, se han estudiado diferentes alternativas en cuanto a la ubicación del *middleware*: en la nube o en el borde de la red. Se ha visto como la gran disponibilidad de recursos *hardware* que se pueden emplear en escenarios basados en la nube afecta positivamente al tiempo de cómputo empleado en la traducción de protocolos llevada a cabo en el *middleware,* frente a escenarios basados en el borde de la red. En contraposición a esto, también se ha verificado como, el hecho de ubicar el *middleware* en la nube afecta negativamente al retardo de transmisión que sufren las comunicaciones frente a ubicaciones en el borde de red. Por este motivo resulta fundamental tener en cuenta el compromiso entre tiempo de procesado y tiempo de transmisión a la hora de decidir qué estrategia tomar o incluso diseñar alternativas dinámicas que permitan conmutar de una a otra.

Por último, a pesar de que se ha trabajado con protocolos totalmente diferentes y, por ende, las librerías empleadas están desarrolladas bajo estructuras muy diferentes, los objetivos propuestos al inicio del proyecto han sido completados satisfactoriamente.

7.2. Trabajos futuros

Como se ha plasmado durante el desarrollo del proyecto, *Internet o Things*, *cloud computing* y *edge/fog computing* son tecnologías emergentes con un gran futuro por delante tanto tecnológico como económico, por lo que resultaría interesante llevar a cabo los siguientes posibles trabajos futuros:

- **Estudio e integración del *middleware* desarrollado en entornos reales:** son muchas las plataformas que ofrecen servicios de computación en la nube como por ejemplo: *Amazon Web Services, Google Cloud, Azure.* Resulta interesante integrar este tipo de *software* en plataformas profesionales como las mencionadas puesto que además de ofrecer una gran potencia computacional, disponen de innumerables servicios orientados a la manipulación de grandes volúmenes de datos *(Big Data).* El *middleware* desarrollado junto con este tipo de soluciones, dotarían de gran flexibilidad a pequeñas y medianas empresas que buscan la transformación digital de su actividad.

- **Conmutación dinámica entre *cloud* y *edge:*** tal y como se ha plasmado en este proyecto, a la hora de desplegar el *middleware* en un entorno cloud o en un entorno perimetral, es importante tener en cuenta el compromiso entre el tiempo de transmisión de la información y el retardo de procesado introducido por el *middleware.* Por esta razón, siguiendo el paradigma *cloud computing continuum,* resultaría interesante diseñar alternativas que permitan conmutar entre una arquitectura y otra en función de las características de la red en cada momento.

- **Escalabilidad dinámica del *middleware:*** el sistema diseñado en este proyecto permite desplegar el *middleware* de forma distribuida en diferentes equipos. La solución implementada es estática, los equipos distribuidos se deben configurar de forma estática construyendo un árbol. Con el afán de aportar un mayor dinamismo, resultaría interesante valorar

alternativas similares al *Spaning Tree* tradicional en el cual los dispositivos que forman la arquitectura distribuida se autoconfiguran dinámicamente formando un árbol lógico. Esto aporta una gran flexibilidad en caso de perdidas o de congestión en los enlaces.

Bibliografía

[1] A. A. da Cruz, M., J. P. C. Rodriges, J., Lorenz, P., Solic, P., Al-Muhtadi, J., & C. Albuquerque, V. (2018). A proposal for bridging application layer protocols to HTTP on IoT solutions. *Future Generation Computer Systems, vol. 97, pp. 145-152, Agosto 2019*.

[2] *Aiocoap 0.3*. (s.f.). Obtenido de https://pypi.org/project/aiocoap/

[3] *Apache Maven Project*. (s.f.). Obtenido de https://maven.apache.org/

[4] B. Babovic, Z., Protic, J., & Milutinovic, V. (s.f.). Web Performance Evaluation for Internet of Things Applications. *IEEE Access, vol. 4, pp. 6974-6992, 2016*.

[5] Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., & Raymor, B. (Febrero de 2018). *RFC 8323. CoAP (Constrained Application Protocol) over TCP, TLS and WebSockets.* Obtenido de https://tools.ietf.org/html/rfc8323

[6] *CoAPthon 4.0.2*. (s.f.). Obtenido de https://pypi.org/project/CoAPthon/

[7] *Data Distribution Service (DDS) Version 1.4.* (Abril de 2015). Obtenido de https://www.omg.org/spec/DDS/1.4/PDF

[8] Dave, M., Patel, M., Doshi, J., & Arolkar, H. (s.f.). Ponte Message Broker Bridge Configuration Using MQTT and CoAP Protocol for Interoperability of Iot. *COMS2: International Conference on Computing Science, Communication and Security, Julio 2020*.

[9] Dierks, T., & Rescorla, E. (Agosto de 2008). *RFC 5246. The Transport Layer Security (TLS) Protocol Version 1.2.* Obtenido de https://tools.ietf.org/html/rfc5246

[10] *Digital Guide IONOS.* (Enero de 2020). Obtenido de https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/lenguajes-del-lado-servidor-o-del-cliente-diferencias/

[11] Dizdarevic, J., Carpio, F., Jukan, A., & Masip-Bruin, X. (s.f.). A survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration. *ACM Computing Surveys, vol. 51, no. 6, Enero 2019*.

[12] *Eclipse Californium*. (s.f.). Obtenido de https://www.eclipse.org/californium/

[13] *Eclipse Foundation*. (s.f.). Obtenido de https://www.eclipse.org/org/foundation/

[14] *Eclipse Foundation. Paho*. (s.f.). Obtenido de https://www.eclipse.org/paho/

[15] *Eclipse Mosquitto. An open source MQTT broker*. (s.f.). Obtenido de https://mosquitto.org/

[16] *ETSI. Successful first Internet of Things (IoT) CoAP Plugtests.* (17 de Abril de 2012). Obtenido de https://www.etsi.org/newsroom/news/390-news-release-17-april-2012

[17] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (Junio de 1999). *RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1*. Obtenido de https://tools.ietf.org/html/rfc2616

[18] *Github arobenko/mqtt-sn*. (s.f.). Obtenido de https://github.com/arobenko/mqtt-sn

[19] *Github AsierCM/Proyecto-middleware-MQTT-HTTP-CoAP*. (s.f.). Obtenido de https://github.com/AsierCM/Proyecto-middleware-MQTT-HTTP-CoAP

[20] *Github eclipse/paho.mqtt-sn.embedded-c*. (s.f.). Obtenido de https://github.com/eclipse/paho.mqtt-sn.embedded-c

[21] *Github jsaak/mqtt-sn-gateway*. (s.f.). Obtenido de https://github.com/jsaak/mqtt-sn-gateway

[22] *Github njh/mqtt-sn-tools*. (s.f.). Obtenido de https://github.com/njh/mqtt-sn-tools

[23] *Github. Eclipse/ponte*. (s.f.). Obtenido de https://github.com/eclipse/ponte

[24] *Github. Ibm-security-innovation/crosscoap*. (s.f.). Obtenido de https://github.com/ibm-security-innovation/crosscoap

[25] *Github. njh/mqtt-http-bridge.* (s.f.). Obtenido de https://github.com/njh/mqtt-http-bridge

[26] *Github. petkov/http_to_mqtt.* (s.f.). Obtenido de https://github.com/petkov/http_to_mqtt

[27] *HiveMQ Open Source*. (s.f.). Obtenido de https://www.hivemq.com/developers/community/

[28] *HiveMQ. MQTT Essentials*. (Enero de 2015). Obtenido de https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt/

[29] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt-sn. (s.f.). *OASIS. MQTT SN Subcommitee.*

[30] *HTTPtoMQTT.* (s.f.). Obtenido de http://httptomqtt.ineptum.dk/#

[31] *IEEE Standard Computer Dictionary.* (s.f.).

[32] Koster, M., Keranen, A., & Jimenez, J. (Septiembre de 2019). *Publish-Subscribe Broker for the Constrained Application Protocol (CoAP).* Obtenido de https://datatracker.ietf.org/doc/draft-ietf-core-coap-pubsub/

[33] *Libcoap. C-Implementation of CoAP*. (s.f.). Obtenido de https://libcoap.net/

[34] Longo, E., E. C. Redondi, A., Cesana, M., Arcia-Moret, A., & Manzoni, P. (s.f.). MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers. *ICC 2020 - 2020 IEEE International Conference on Communications (ICC), Dublin, Irlanda, 2020, pp. 1-6.* .

[35] Ludovici, A., & Calveras, A. (s.f.). A Proxy Design to Leverage the Interconnection of CoAP Wireless Sensor Networks with Web Applications. *Sensors, vol. 15, no. 1, pp. 1217-1244, Enero 2015.*

[36] Mills, D., Delaware, U., Martin, J., Burbank, J., & Kasch, W. (s.f.). *RFC 5905. Network Time Protocol Version 4: Protocol and Algorithms Specification.* Obtenido de https://tools.ietf.org/html/rfc5905

[37] *Moquette MQTT broker*. (s.f.). Obtenido de https://moquette-io.github.io/moquette/

[38] *MQTT. The Standard for IoT Messaging*. (s.f.). Obtenido de https://mqtt.org/

[39] *mqttsn 0.0.4*. (s.f.). Obtenido de https://pypi.org/project/mqttsn/

[40] Naik, N. (s.f.). Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, 2017, pp. 1-7.*

[41] *NPM. Node-coap*. (s.f.). Obtenido de https://www.npmjs.com/package/coap

[42] *OASIS. Message Queuing Telemetry Transport (MQTT)*. (s.f.). Obtenido de https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt

[43] *OASIS. MQTT Version 3.1.1*. (29 de Octubre de 2014). Obtenido de http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[44] *OMG. Data Distribution Service (DDS)*. (s.f.). Obtenido de https://www.omg.org/omg-dds-portal/

[45] *Ponte. Connecting Things to Developers.* (s.f.). Obtenido de https://www.eclipse.org/ponte/#

[46] Rescorla, E. (Agosto de 2018). *RFC 8446. The Transport Layer Security (TLS) Protocol Version 1.3.* Obtenido de https://tools.ietf.org/html/rfc8446

[47] Rescorla, E., & Modadugu, N. (Junio de 2012). *RFC 6347. Datagram Transport Layer Security Version 1.2.* Obtenido de https://tools.ietf.org/html/rfc6347

[48] *RFC 8173. Precision Time Protocol Version 2 (PTPv2).* (2017). Obtenido de https://tools.ietf.org/html/rfc8173

[49] Saint-Andre, P. (Marzo de 2011). *RFC 6120. Extensible Messaging and Presence Protocol (XMPP): Core.* Obtenido de https://tools.ietf.org/html/rfc6120

[50] Saint-Andre, P. (Marzo de 2011). *RFC 6121. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence.* Obtenido de https://tools.ietf.org/html/rfc6121

[51] *Schneider Electric. IoT 2020 Bussiness Report.* (2020). Obtenido de https://download.schneider-electric.com/files?&p_enDocType=Brochure&p_File_Name=998-19699217_GMA-US_IoT_Report_CS6_v3.pdf&p_Doc_Ref=98-19699217_IoT_Report_2016_v2.pdf

[52] Shelby, Z., Hartke, K., & Bormann, C. (2014). *RFC 7252. The Constrained Application Protocol (CoAP).* Obtenido de https://tools.ietf.org/html/rfc7252

[53] Standford-Clark, A., & Linh Truong, H. (14 de Noviembre de 2013). *MQTT For Sensor Networks (MQTT-SN) Protocol Specification. Version1.2*. Obtenido de https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf

[54] *Tecnología para los negocios*. (s.f.). Obtenido de https://ticnegocios.camaravalencia.com/servicios/tendencias/caminar-con-exito-hacia-la-industria-4-0-capitulo-12-infraestructuras-ii-protocolos/

[55] Vasters, C., & Godfrey, R. (2014). *OASIS. Advanced Message Queuing Protocol (AMQP) TC.*

[56] Wiss, T. (s.f.). *Github tbwiss/CoAP_PubSub*. Obtenido de https://github.com/tbwiss/CoAP_PubSub

[57] *WolfSSL. WolfMQTT Client Library*. (s.f.). Obtenido de https://www.wolfssl.com/products/wolfmqtt/

[58] Yokotani, T., & Sasaki, Y. (s.f.). Comparison with HTTP and MQTT on required network resources for IoT. *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC), Bandung, 2016, pp. 1-6*.

* Todos los enlaces *web* y las referencias bibliográficas han sido revisadas y verificadas a día 22 de Septiembre de 2020.

Anexos

Anexo 1: Diagrama de clases del *middleware*

A continuación, en la Figura 45 se muestra el diagrama de clases del *middleware.* Es necesario aclarar que las librerías empleadas para las interfaces MQTT, CoAP y HTTP: Moquette, Californium CoAP PubSub y HTTP Oracle son sumamente extensas y en este diagrama, únicamente se muestran las clases empleadas para el desarrollo del proyecto.

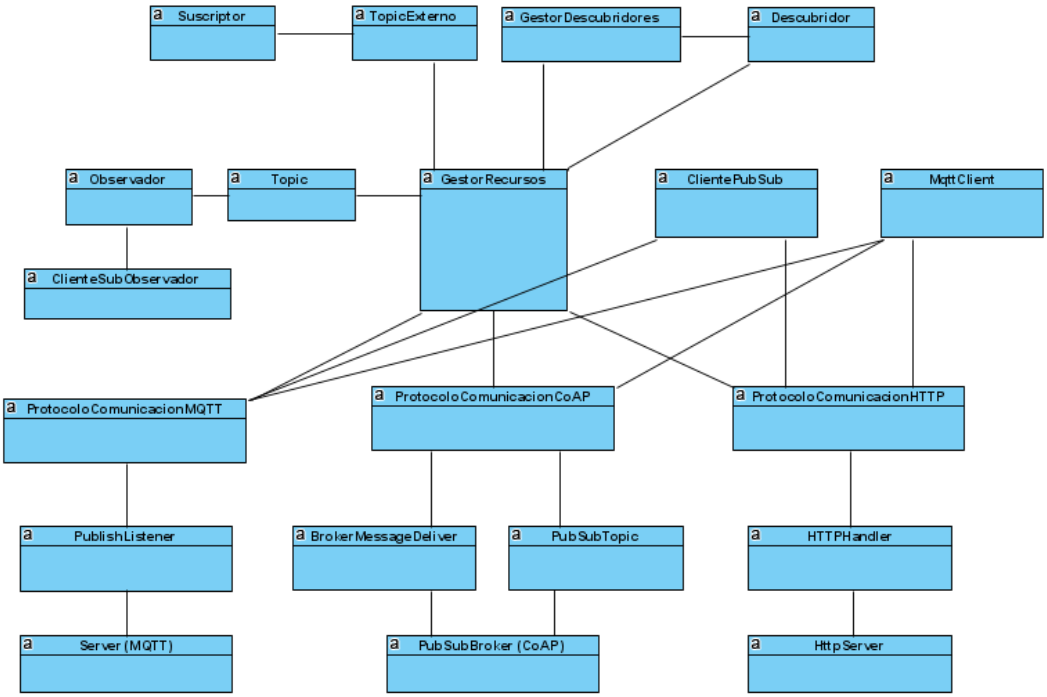


Figura 45. Diagrama de clases del midddleware

En primer lugar, se emplean las clases: ***Server (MQTT), PubSubBroker (CoAP)*** **y** ***HttpServer,*** que pertenecen a las librerías mencionadas. Mediante estas clases se crean las instancias de los tres servidores que reciben los mensajes de los tres protocolos. En estas clases se configuran los parámetros de los servidores.

A continuación, se tienen las clases: ***PublishListener, BrokerMessageDeliver, PubsubTopic y HTTPHandler.*** Estas clases son las encargadas de manejar los mensajes que se reciben en los tres servidores. Dichos mensajes son los que se analizan posteriormente en las clases de niveles superiores para llevar a cabo el funcionamiento del *middleware.* A continuación, se explica brevemente el funcionamiento de cada clase.

- ***PublishListener*:** esta clase permite analizar los mensajes MQTT entrantes a través de los métodos *onPublish(…), onSubscribe(…),* entre otros. Desde estos métodos se invoca el método *iniciarComunicación(…)* de la clase *ProtocoloComunicaciónMQTT* para iniciar la conversión a los protocolos CoAP y HTTP en caso de que sea necesario.

- ***BrokerMessageDeliver:*** esta clase permite analizar los mensajes CoAP entrantes mediante el método *deliverRequest(…).* Esta clase se emplea para analizar si el mensaje va dirigido a un recurso existente o no. En caso de que exista el recurso en el *broker* CoAP, se pasa la petición al objeto correspondiente de la clase ***PubsubTopic*** y este se encarga de gestionar la petición. Por esta razón, en la clase ***BrokerMessageDeliver*** se realiza una primera comprobación de si la petición va dirigida a un *topic* alojado en un servidor CoAP. En caso positivo se invoca el método *iniciarComunicación(…)* en el *ProtocoloComunicaciónCoAP*. Si no se realizara esta acción aquí, el metodo *deliverRequest(…)* responde a la petición mediante el código de error 4.04 NOT FOUND puesto que no se trata de un recurso publicado en el *broker.*
- ***PubSubTopic:*** cada instancia de esta clase, que contiene los métodos *handleGet(…), handlePut(…), handlePost(…)* entre otros, hace referencia a un *topic.* En caso de que el *topic* haya sido publicado, existirá una instancia de esta clase que gestione las peticiones entrantes a este *topic.* En los métodos nombrados, es donde se invoca el método *iniciarComunicación(…)* en el *ProtocoloComunicaciónCoAP* para iniciar la conversión a los protocolos MQTT y HTTP en caso de que sea necesario.
- ***HTTPHandler:*** esta clase permite analizar los mensajes HTTP recibidos a través del método *handle(…).* En este se verifica si la petición recibida contiene un mensaje GET, PUT/POST entre otros. A partir de aquí se invoca el método *iniciarComunicación(…)* en la clase *ProtocoloComunicaciónHTTP* para iniciar la conversión a los protocolos MQTT y HTTP en caso de que sea necesario.

Cabe destacar que la clase ***PublishListener*** extiende la clase ***AbstractInterceptHandler*** de la librería Moquette, las clases ***BrokerMessageDeliver*** y ***PubSubTopic*** son clases del proyecto CoAP_PubSub y la clase ***HTTPHandler*** extiende la clase ***HttpHandler*** de la librería de Oracle HTTP. El resto de clase explicadas a continuación se han desarrollado durante el proyecto.

Las siguientes clases en las que se basa la arquitectura del *middleware* son: ***ProtocoloComunicaciónMQTT, ProtocoloComunicaciónCoAP*** **y** ***ProtocoloComunicaciónHTTP.*** Estas son las encargadas de realizar las acciones necesarias para permitir la intercomunicación entre los protocolos MQTT, CoAP y HTTP. Para la traducción de los mensajes entre protocolos, estas tres clases consultan y actualizan de forma dinámica la información de control sobre los *topics* almacenada en el objeto de la clase ***GestorRecursos.*** Las tres clases cuentan con los siguientes métodos:

- *iniciarComunicación(…):* cuando se recibe un mensaje, este método es el encargado de gestionar su posible traducción a otro protocolo. En caso de tratarse de un PUBLISH MQTT se traduce a un PUT/POST COAP o HTTP y viceversa. Si lo que se recibe es un mensaje de tipo MQTT SUBSCRIBE o un mensaje de tipo GET CoAP o HTTP se comprueba si es un *topic* alojado en un servidor CoAP o en un *middleware* distribuido y se actúa en consecuencia, tal y como se detalla en la memoria del proyecto.

- *cancelarCounicación(…):* a través de este método se gestiona la desconexión de los diferentes usuarios que se encuentran suscritos u observando un *topic.* A través de él se informa al ***GestorRecursos*** para tener control sobre ello.

Los tres *ProtocolosComunicacion(MQTT, CoAP y HTTP)* emplean instancias de las clases ***MQTTClient*** y ***ClientPubSub***. Estas clases implementan clientes MQTT y CoAP que se encargan de generar los mensajes MQTT PUBLISH y CoAP PUT/POST fruto de la traducción de los mensajes recibidos.

Una vez visto donde se reciben y se manejan los mensajes entrantes da cada protocolo, la siguiente clase importante es la clase ***GestorRecursos***. En ella se guarda la información de control necesaria para la comunicación y sincronización entre los diferentes *ProtocolosComunicacion(MQTT, CoAP, HTTP)***.** En ella, existen tres atributos importantes: *topicList, topicListTotal* y *topicListExternos*. En estas listas se almacena la información necesaria sobre *topics* alojados en servidores CoAP, *topics* publicados mediante publicaciones MQTT-CoAP-HTTP y *topics* recibidos desde otros *middlewares* distribuidos respectivamente. En relación con estas listas, existen métodos como *informarTopic(..), informarTopicTotal(…) o informarTopicExterno(..),* mediante los cuales se actualiza la información de control necesaria sobre esos *topics.* También existen los métodos *comprobarTopic(…), comprobarTopicTotal(…)* y *comprobarTopicExterno(..),* mediante los cuales se consulta la existencia de un *topic* en las distintas listas. Además, en relación con los usuarios suscritos a los diferentes *topics*, los métodos *informarUsuario(…) e informarUsuarioTopicExterno(…)* se emplean para añadir o borrar información sobre los usuarios activos. Por último, en esta clase, se almacena la lista de los identificadores y la dirección IP de los diferentes *middlewares* distribuidos que forman la red.

A su vez, la clase ***GestorRecursos*** utiliza instancias de la clase ***Topic***. Esta clase representa un *topic* que ha sido publicado desde un cliente MQTT, CoAP o HTTP o que se encuentra en un servidor CoAP y se almacena en la lista *topicList* y *topicListTotal* dentro de ***GestorRecursos.*** Contiene información sobre el nombre, el protocolo desde el cual ha sido publicado, el contenido, la lista de usuarios que están suscritos y un objeto de la clase ***Observador***. En el caso de que se realice una petición GET CoAP o HTTP o un mensaje MQTT SUBSCRIBE hacia un *topic* alojado en un servidor CoAP, es necesario lanzar un cliente CoAP creando una relación de observación en el servidor. Para ello se genera la instancia de la clase ***Observador*** que también se almacena en la clase ***Topic.*** Este es un proceso que controla una instancia de la clase ***ClienteSubObservador,*** la cual contiene el cliente CoAP con el que se establece la comunicación con el servidor. Los datos recibidos en el objeto ***Observador*** son traducidos y transmitidos internamente mediante el ***GestorRecursos***.

Adicionalmente, la clase ***GestorRecursos*** utiliza instancias de la clase ***TopicExterno***. Cada instancia de esta clase representa un *topic* que ha sido notificado desde otro *middleware* distribuido y las instancias de esta clase se almacenan en la lista *topicListExternos* dentro del ***GestorRecursos.*** Contiene información sobre el nombre del *topic*, la identidad del *middleware* desde el que ha sido notificado, una lista de usuarios suscritos y un objeto de la clase ***Suscriptor*.** Este último contiene un cliente MQTT mediante el cual se realiza una suscripción MQTT a un *middleware* distribuido en el caso

de recibir una petición GET CoAP o HTTP o un mensaje MQTT PUBLISH hacia un *topic* controlado por otro *middleware* de la red.

Por último, se define la clase ***GestorDescubridores***, que contiene principalmente una lista con las direcciones IP de todos los servidores CoAP a los que el *middleware* da servicio. Este se encarga de generar una instancia de la clase ***Descubridor*** para cada servidor CoAP. Cada objeto ***Descubridor*** contiene un cliente CoAP que se encarga de realizar una petición GET a la URI *.well-known/core* del servidor CoAP correspondiente. La respuesta obtenida contiene todos los *topics* alojados en este servidor. A continuación, esta información se le comunica al objeto ***GestorRecursos.***

Anexo 2: Creación de certificados digitales

Para la utilización de canales cifrados mediante los protocolos TLS y DTLS es necesario la creación de certificados digitales tanto para el servidor como para los diferentes clientes. Para ello se emplea la herramienta **Keytool** disponible en Linux. Esta herramienta permite crear almacenes de claves e importar y exportar certificados de los almacenes. A continuación, se muestra un pequeño de código *Shell* a modo de ejemplo, obtenido de la página oficial de Eclipse Californium Scandium.

```
#!/bin/bash

KEY_STORE=keyStore.jks
KEY_STORE_PWD=endPass
TRUST_STORE=trustStore.jks
TRUST_STORE_PWD=rootPass
VALIDITY=365


#creating root key and certificate
echo "creating root key and certificate..."
keytool -genkeypair -alias root -keyalg EC -dname 'C=CA,L=Ottawa,O=Eclipse
IoT,OU=Californium,CN=cf-root' \
        -ext BC=ca:true -validity $VALIDITY -keypass $TRUST_STORE_PWD -keystore
$TRUST_STORE -storepass $TRUST_STORE_PWD



#creating CA key and certificate
echo "creating CA key and certificate..."
keytool -genkeypair -alias ca -keyalg EC -dname 'C=CA,L=Ottawa,O=Eclipse
IoT,OU=Californium,CN=cf-ca' \
        -ext BC=ca:true -validity $VALIDITY -keypass $TRUST_STORE_PWD -keystore
$TRUST_STORE -storepass $TRUST_STORE_PWD

keytool -keystore $TRUST_STORE -storepass $TRUST_STORE_PWD -certreq -alias ca | \
  keytool -keystore $TRUST_STORE -storepass $TRUST_STORE_PWD -alias root -gencert -
validity $VALIDITY -ext BC=0 -rfc | \
  keytool -alias ca -importcert -keystore $TRUST_STORE -storepass $TRUST_STORE_PWD



#creating server key and certificate
echo "creating server key and certificate..."
keytool -genkeypair -alias server -keyalg EC -dname 'C=CA,L=Ottawa,O=Eclipse
IoT,OU=Californium,CN=cf-server' \
        -validity $VALIDITY -keypass $KEY_STORE_PWD -keystore $KEY_STORE -storepass
$KEY_STORE_PWD

keytool -keystore $KEY_STORE -storepass $KEY_STORE_PWD -certreq -alias server | \
  keytool -keystore $TRUST_STORE -storepass $TRUST_STORE_PWD -alias ca -gencert -
validity $VALIDITY -rfc > server.pem

keytool -alias server -importcert -keystore $KEY_STORE -storepass $KEY_STORE_PWD -
trustcacerts -file server.pem



#creating client key and certificate
echo "creating client key and certificate..."
keytool -genkeypair -alias client -keyalg EC -dname 'C=CA,L=Ottawa,O=Eclipse
IoT,OU=Californium,CN=cf-client' \
        -validity $VALIDITY -keypass $KEY_STORE_PWD -keystore $KEY_STORE -storepass
$KEY_STORE_PWD

keytool -keystore $KEY_STORE -storepass $KEY_STORE_PWD -certreq -alias client | \
  keytool -keystore $TRUST_STORE -storepass $TRUST_STORE_PWD -alias ca -gencert -
validity $VALIDITY -rfc > client.pem

keytool -alias client -importcert -keystore $KEY_STORE -storepass $KEY_STORE_PWD -
trustcacerts -file client.pem
```

Para la creación de los certificados se usa una cadena de confianza de varios niveles:

1. Par de claves privada/publica junto con un certificado autofirmado que en conjunto representan la identidad de la CA raíz *(root).*
2. Par de claves privada/publica junto con un certificado firmado con la clave de la CA raíz que en conjunto representan la identidad de la CA intermediaria.
3. Par de claves privada/publica junto con un certificado firmado con la clave de la CA intermediaria que en conjunto representan la identidad de un servidor.
4. Par de claves privada/publica junto con un certificado firmado con la clave de la CA intermediaria que en conjunto representan la identidad de un cliente.

Las claves y certificados se almacenan en dos almacenes de claves: *keyStore.jks* y *trustStore.jks,* explicados a continuación.

1. *keyStore.jks*: contiene las claves y cadenas de certificados para el cliente y el servidor,
2. *trustStore.jks*: contiene el certificado autofirmado de la CA raíz, así como la cadena de certificados de la CA intermedia. Estos certificados se usan como certificados de confianza para verificar la identidad de cliente y servidor.

Se recomienda el uso de cadenas multinivel para que, en caso de que la clave privada de una CA intermediaria se vea comprometida, la CA raíz pueda revocar su certificado y no comprometer la seguridad total de la cadena. Para ello se recomienda encarecidamente mantener almacenada la clave privada de la CA raíz en *hardware* no accesible desde la red y correctamente protegida.