UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo al Egreso de Profesionales en Actividad

TÍTULO: Arquitectura orientada a eventos sobre protocolo MQTT

AUTOR: Luciano Chambers

DIRECTOR ACADÉMICO: Ismael Rodriguez

DIRECTOR PROFESIONAL: Silvina Maneglia

CARRERA: Licenciatura en Sistemas

Resumen

El presente trabajo de investigación abarca el estudio, análisis, desarrollo e implementación de un sistema de monitoreo para la plataforma de ventas online de una empresa de turismo. El tema central sobre el que se basa esta investigación es la comunicación basada en eventos utilizando el protocolo MQTT. Para la justificación de la elección del protocolo mencionado, se realizó un estudio para analizar y comparar la performance entre los protocolo MQTT y HTTP en el contexto de comunicación basada en eventos. El trabajo se estructuró de la siguiente manera: El capítulo 1 hace referencia a la introducción del tema general de esta investigación, sus objetivos, motivación y la metodología de investigación a seguir. En el capítulo 2 se desarrolla el marco teórico donde se expone las características y el funcionamiento de los protocolos HTTP y MQTT, y las características de las arquitecturas de software distribuidas. En el capítulo 3 se expone la plataforma de ventas online, la herramienta de monitoreo desarrollada, y el estudio realizado para analizar y comparar la performance de los protocolos HTTP y MQTT en el contexto de comunicación basado en eventos. En el capítulo 4 se expone las conclusiones obtenidas y líneas de trabajos a futuro. Finalmente, se presenta la bibliografía utilizada en el contexto de la presente investigación y un capítulo anexo que incluye el código del algoritmo desarrollado para realizar las pruebas de performance mencionadas.

Palabras Clave

MQTT.
HTTP.
Bus de eventos.
Broker.
Arquitectura.

Conclusiones

El objetivo planteado en esta tesina se considera satisfecho al lograr comprobar que el mejor protocolo para la comunicación basada en eventos es MQTT, ya que se demostró con el estudio y análisis realizado que presenta una mejor perfomance que HTTP a mayor cantidad de procesamiento de eventos.

Trabajos Realizados

En esta tesina luego de haber utilizado el protocolo MQTT para el desarrollo de una aplicación de monitoreo, se analizo la performance del protocolo MQTT en comparación con el protocolo HTTP a través de pruebas con algoritmos que ejecutan miles de eventos en cada uno de estos protocolos.

Trabajos Futuros

Completar pruebas de performance de MQTT con calidad de servicio Qos1 y Qos2.
Investigación sobre motores de base de datos para lograr soportar mayor cantidad de eventos evitando así pérdida de información.

Fecha de la presentación: Noviembre 2021

Índice general

Capítulo 1

Introducción

A lo largo del tiempo, las arquitecturas de las aplicaciones de software han sido modificadas de acuerdo a las tecnologías disponibles y a los requerimientos de las mismas. En un principio se utilizaban aplicaciones monolíticas, y a medida que los sistemas fueron creciendo fue necesario distribuirlas, aprovechando así también las tecnologías de red que ofrecía el mercado para la comunicación de las mismas. De esta forma van surgiendo arquitecturas de software que están compuestas por un conjunto grande de aplicaciones con responsabilidades específicas que necesitan estar comunicadas entre sí, alejándose así de aplicaciones monolíticas con un único módulo de software que contiene toda la responsabilidad de resolver los requerimientos. La intención de esta tesina es hacer una revisión sobre arquitecturas de software, y mayormente en los protocolos de red utilizados para la comunicación de los distintos módulos de software cuando estos se encuentran en un entorno de servidores distribuidos. Luego de esto se presentará un caso de estudio de la plataforma web de una unidad de negocios de **despegar.com** llamada Agencias Afiliadas, específicamente sobre el monitoreo de esta plataforma, donde se realizó a través de un bus de eventos utilizando el protocolo de comunicación MQTT en lugar de utilizar el protocolo de aplicación de red HTTP. Y finalmente se realizarán pruebas comparativas entre los protocolos previamente mencionados.

1.1. Motivación

1.1.1. Desarrollo de software

Hoy en día los grandes sistemas, en término de su funcionalidades y manejo de datos, son necesarios modularizarlos cada vez más, con el objetivo de no tener una única aplicación monolítica de un gran tamaño que conlleva un costo alto de mantenimiento, incrementa los riesgos de bugs en su modificación y se complejiza la escalabilidad de la misma. Con el fin de simplificar estas cuestiones es necesario dividir las aplicaciones en sistemas más pequeños donde cada aplicación tenga un negocio acotado con funcionalidades solamente necesarias para este negocio. De esta forma se realiza más fácil y menos costoso el mantenimiento del sistema completo con menor riesgo de bugs o errores en cada iteración, y pudiendo así aplicar nuevos requerimientos de negocio de una forma más ágil.

1.1.2. Comunicación entre aplicaciones de software distribuidas

Para que las aplicaciones de estas arquitectura previamente mencionadas funcionen en un ecosistema tecnológico, de distintas aplicaciones y tecnologías, es necesario que se comuniquen entre sí, esta comunicación se realiza mediante de la red ethernet/internet y la forma de comunicarse es a través de la interfaz de servicios que expone cada una. En rasgos generales la comunicación de estas aplicaciones se realizan a través de distintos protocolos de red, tales son conocidos como el modelo de capa de protocolos OSI [1], en esta investigación se realizará el análisis sobre el protocolo de aplicaciones HTTP y sobre el protocolo MQTT.

1.2. Objetivos

El objetivo de la tesis será el de desarrollar una herramienta para el monitoreo de aplicaciones de software, esta herramienta se implementará utilizando un protocolo de red específico de IoT llamado MQTT. Para la demostración de esta herramienta se tomará como caso de estudio una unidad de negocios de una empresa de viajes llamada **despegar.com**. Específicamente se utilizara de esta empresa, como caso de estudio, el canal de negocios Agencias Afiliadas, este canal se basa principalmente en brindar una plataforma web a las agencias de viajes, y así las agencias disponer de un sitio web propio para realizar sus propias ventas. Cada producto de turismo que se vende al público en esta plataforma está desarrollado en una aplicación de software independiente de otros, por ejemplo existe una aplicación para la ventas de vuelos, otra para venta de hoteles y otra para los paquetes, desarrollados y puestas en producción en ambiente distribuidos. Este conjunto de aplicaciones genera un ambiente difícil de monitorear los casos de usos y entonces surge la necesidad de la herramienta que se desarrollará para esto. Una vez implementada y demostrada esta herramienta se realizará una comparativa y análisis del protocolos de HTTP y MQTT con el fin de poder afirmar el acierto de la elección del protocolo MQTT para esta solución.

El objetivo específico será entonces el de desarrollar una herramienta para poder llevar a cabo el monitoreo de las acciones que ejecutará el usuario sobre la plataforma previamente mencionada, generando un almacén de datos con esta información, para luego poder realizar el seguimiento de los casos de uso e identificar posibles errores ocurridos en las aplicaciones.

Capítulo 2

Marco Teórico

En este capítulo se incluye la información recopilada correspondiente al marco teórico, concretamente sobre los protocolos más importantes para este proyecto de investigación. MQTT es el protocolo principal del proyecto, por lo que se describirá en profundidad todas las características de el. Por otro lado también es necesario el conocimiento de HTTP ya que es el protocolo por el cual se va a hacer la comparación con MQTT y el que se utiliza hoy en día en el mercado para la solución planteada. Además se describirá el funcionamiento de las distintas arquitecturas utilizadas en el caso de estudio.

2.1. MQTT

MQTT (Message Queuing Telemetry Transport) es un protocolo de mensajería simple diseñado para dispositivos con ancho de banda limitado y alta latencia. Se basa en el paradigma publicador-suscriptor en el que un broker actúa como intermediario encargándose de direccionar los mensajes con el uso de un topic común [2]. El protocolo trabaja sobre TCP/IP u otros protocolos que permitan conexiones bidireccionales, ordenadas y sin pérdidas. Su principal objetivo es minimizar los requerimientos de ancho de banda de red y los recursos de los dispositivos que lo usan, manteniendo cierto grado de habilidad, lo que hace que su uso en IoT (Internet de las cosas) o conexiones machine-to-machine (M2M) sea muy conveniente [3]. MQTT se inventó en 1999 por el Dr. Andy Stanford-Clark de IBM y Arlen Nipper de Arcom y las versiones 5.0 y 3.1.1 (ratificada por el ISO) son estándares de OASIS [4]. Por otro lado IANA (Internet Assigned Numbers Authority) tiene reservados los puertos TCP/IP 1883 para MQTT y 8883 para MQTT sobre SSL [3].

2.1.1. Modelo

La característica principal en el modelo MQTT es que sigue la arquitectura de comunicación cliente/servidor normalmente con una topología en estrella, con un nodo central que funciona de servidor y con una capacidad de hasta 10.000 clientes [5] A continuación se describen los componentes de software principales que aparecen en un escenario MQTT [2] :

- Broker: este componente actúa como servidor encargándose de la transmisión de mensajes con los clientes y la gestión de la red. También mantiene activo el canal con los clientes respondiendo a los mensajes periódicos que estos envían.
- Cliente: el cliente puede tener funciones de publicador y de suscriptor al mismo tiempo.
  - a) Publicador: actúa como cliente y se encarga de transmitir información al broker sobre un determinado topic.
  - b) Suscriptor: actúa como cliente y se encarga de recibir información del broker sobre un determinado topic.
- Mensaje: unidad de datos o información que se transmite o recibe sobre un topic.
- Topic: el tópico al que los clientes pueden suscribirse para recibir información sobre ese asunto o publicarla. Es similar a una cola de mensajes, pero los mensajes pueden almacenarse hasta que sean consumidos y se pueden distribuir a varios clientes.

Para comenzar una comunicación entre el broker y un cliente se establece una sesión, que se cerrará al concluir, con un intercambio de paquetes como se verá en profundidad en el siguiente apartado, y se puede mantener activa el tiempo que se necesite con mensajes periódicos que el cliente envía y el broker debe responder [6] Al ser el broker el encargado de controlar la red y todos los mensajes, hace que la comunicación entre transmisor y receptor se desacople, lo que supone tres ventajas en comunicaciones de este tipo [7]:

- El publicador únicamente necesita conocer la dirección IP y puerto del broker, siendo irrelevantes en la publicación los destinatarios del mensaje.
- Publicador y suscriptor no tienen que estar conectados a la vez ya que el broker se encargará de almacenar la información.
- Publicador y suscriptor no necesitan sincronizarse.

En la figura 2.1 podemos ver un escenario básico de MQTT con tres clientes, donde el cliente de la izquierda hace la función de publicador y los de la derecha de suscriptores. El mensaje enviado desde el publicador en cierto topic llega hasta el broker, que se encarga de distribuirlo a los dos clientes suscritos en el mismo topic.

2.1.2. Formato de Mensajes

En el protocolo MQTT se intercambian una serie de paquetes de control en un orden determinado para llevar a cabo la comunicación, los cuales se componen de tres partes diferentes dependiendo del tipo de paquete: cabecera fija, cabecera variable y payload.

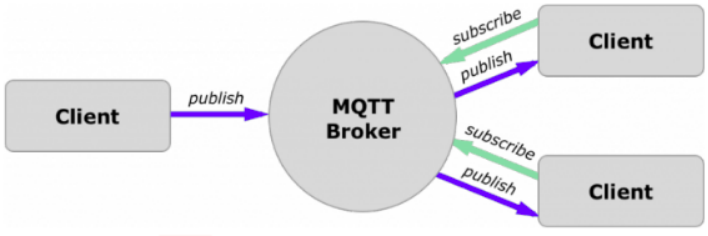


Figura 2.1: Modelo de comunicación MQTT [8]

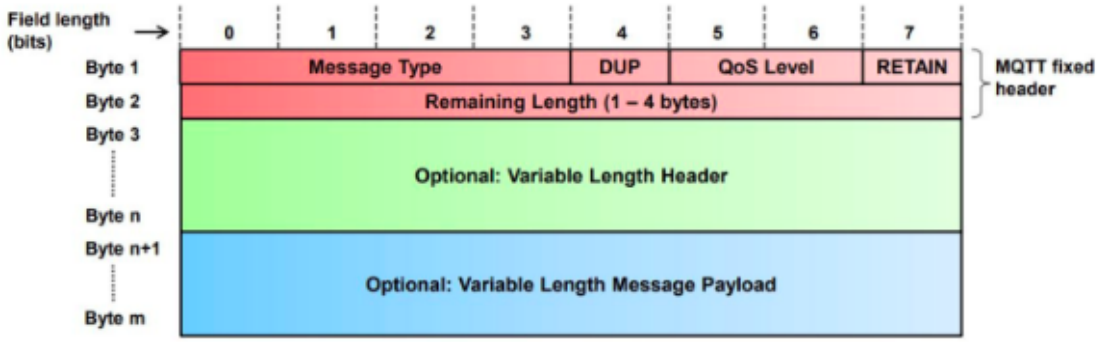


Figura 2.2: Estructura paquete MQTT [6]

Cabecera fija

La cabecera fija o "Fixed Header" es la parte inicial de los paquetes MQTT y es la única que se encuentra presente en todos ellos, esta cabecera se divide a su vez en distintos campos.

- Message Type: corresponde a los cuatro primeros bits del primer byte e indica los posibles tipos de mensajes del protocolo, como se detalla en la figura 2.2.
- Flags: se encuentran en la segunda mitad del primer byte y aunque la mayoría de los mensajes tienen unos valores determinados , otros son variables para indicar ciertas características.
  - DUP: bit de duplicidad que indica si el receptor puede haber recibido ya ese mensaje.
  - QoS: indica la calidad de servicio que se está usando, puede ser de calidad 0, 1 o 2 (Qos0, Qos1, Qos2)
  - Retain: si está activo (1) indica al servidor que retenga el mensaje **publish** para enviarlo a futuras suscripciones.
- Remaining Length : indica el número de bytes restantes del paquete incluyendo la cabecera variable y el payload. La longitud del propio campo también es variable, pudiendo ser desde uno hasta cuatro bytes.

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2...	Remaining Length							

Figura 2.3: Fixed Header paquete MQTT [6]

Cabecera variable

La cabecera variable o Variable Header se encuentra presente en algunos de los paquetes MQTT. Dependiendo del tipo de mensaje o de su nivel de calidad de servicio (QoS) puede contener distintos campos que aporten información específica, como el nombre del topic, pero casi siempre contendrá los bytes de Packet Identier/Message ID, que identifica el paquete con la fórmula del bit más signicativo (MSB) y del menos signicativo (LSB).

Payload

La carga útil o Payload es la parte final del paquete, de longitud variable, que contiene la información que se quiere transmitir y que se encuentra solo en algunos tipos de paquetes como se muestra en el campo Payload de la figura 2.2.

2.1.3. Tipos de Mensajes

CONNECT

Es el primer mensaje que manda el cliente al servidor una vez que la comunicación se ha establecido y solo debe enviarse uno. A continuación, se muestra un mensaje connect y se explican los campos especícos de Variable Header y Payload.

- Protocol name: caracteres con el nombre del protocolo codificado en UTF-8.
- Protocol version: entero con el número de versión.
- User/Password Flag: activo (1) si los campos de usuario/contraseña aparecen en el payload.
- Will Retain: activo si el servidor debe retener el mensaje.
- Will QoS: especica el nivel de QoS.
- Will Flag: indica si aparece un mensaje Will en el payload .
- Clean Session: si está activo (1), el servidor limpiará la información sobre el cliente.
- Keep Alive Timer: intervalo de tiempo máximo entre mensajes del cliente. El servidor comprobará el estado.

Nombre	Valor	Dirección del flujo	Descripción	Payload
Reserved	0	-	Reservado	No hay
connect	1	Cliente → Servidor	Petición de conexión	Obligatorio
connack	2	Cliente ← Servidor	Confirmación conexión	No hay
publish	3	Cliente ↔ Servidor	Mensaje de publicación	Opcional
puback	4	Cliente ↔ Servidor	Confirmación publicación	No hay
pubrec	5	Cliente ↔ Servidor	Recepción de publicación (entrega asegurada I)	No hay
pubrel	6	Cliente ↔ Servidor	Lanzamiento de publicación (entrega asegurada II)	No hay
pubcomp	7	Cliente ↔ Servidor	Publicación completada (entrega asegurada III)	No hay
subscribe	8	Cliente → Servidor	Petición de suscripción	Obligatorio
suback	9	Cliente ← Servidor	Confirmación suscripción	Obligatorio
unsubscribe	10	Cliente → Servidor	Petición de cancelación de suscripción	Obligatorio
unsuback	11	Cliente ← Servidor	Confirmación cancelación de suscripción	No hay
pingreq	12	Cliente → Servidor	Solicitud de PING	No hay
pingresp	13	Cliente ← Servidor	Respuesta de PING	No hay
disconnect	14	Cliente → Servidor	Client desconectado	No hay
Reserved	15	-	Reservado	No hay

Figura 2.4: Descripcion de los mensajes de control MQTT [4]

- Client Identier: identicación del cliente.
- Will Topic: topic en el que se publicará el mensaje.
- Will Message: mensaje que se publicará en el topic .
- Username/Password: usuario y contraseña.

CONNACK

Es el mensaje de respuesta a un connect en el que se acepta (0) o se rechaza la conexión (1-5). La conexión puede rechazarse por incompatibilidad de versión (1), error de identificador (2), de servidor (3), de usuario o contraseña (4) o por no estar autorizada la conexión (5).

PUBLISH

Este mensaje se utiliza para publicar mensajes en un tópico determinado y que los suscriptores reciban la información.

Bit	7	6	5	4	3	2	1	0
byte 1	Packet Identifier MSB							
byte 2	Packet Identifier LSB							

Figura 2.5

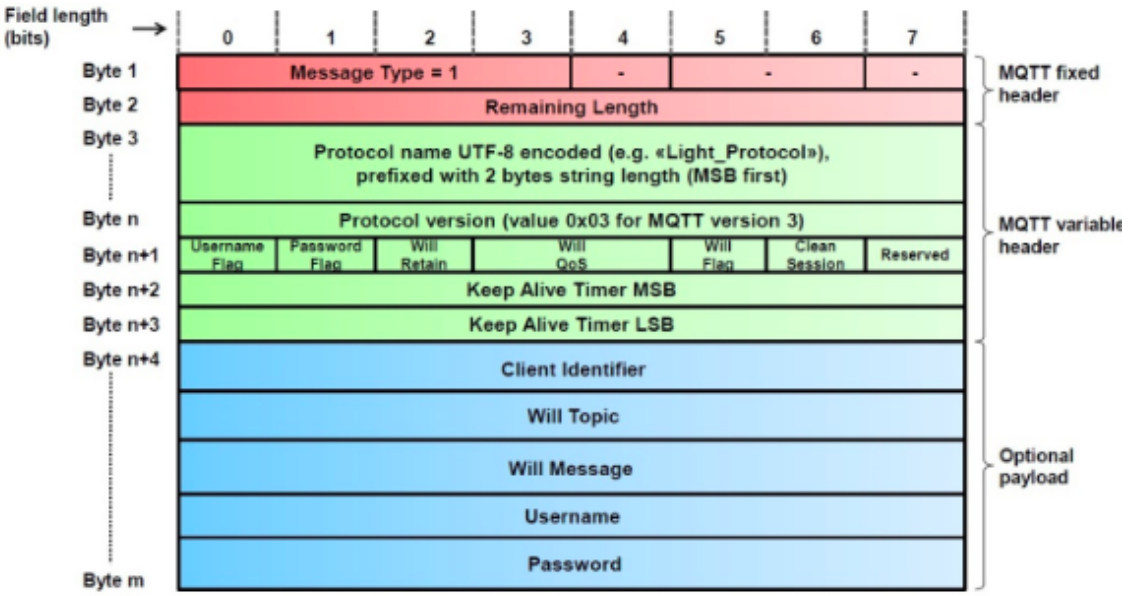


Figura 2.6: Mensaje Connect [6]

- Topic Name String Length/Topic Name: tema donde se publica el mensaje. Los dos primeros bytes indican la longitud del nombre.
- Message ID: solo presente si la QoS (calidad de servicio) es distinta a 0.
- Publish Message: mensaje a publicar.

PUBACK

Este paquete se utiliza como confirmación en caso de que el mensaje publish use QoS de nivel 1.

PUBREC/PUBREL/PUBCOMP

Estos paquetes se utilizan como confirmación en caso de que el mensaje publish use QoS de nivel 2. La respuesta a publish será pubrec, la respuesta a este será pubrel y, por último, se confirmará con pubcomp.

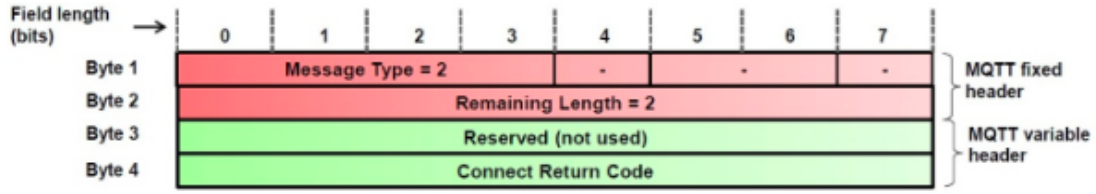


Figura 2.7: Mensaje Conact [6]

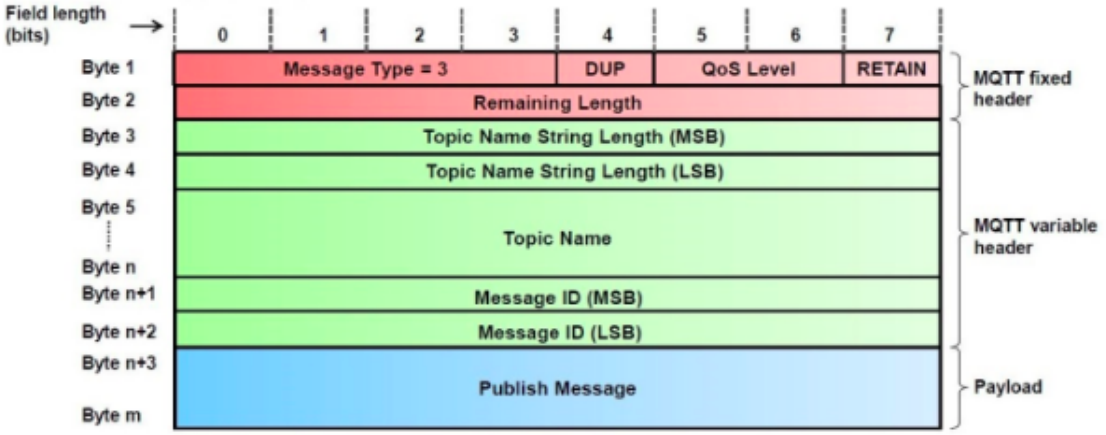


Figura 2.8: Mensaje Publish [6]

SUBSCRIBE

Este paquete se enviará desde el cliente hacia el servidor para suscribirse en uno o más topics y que el cliente reciba todos los mensajes que se publiquen en dicho topic . Deberá incluir siempre el Topic Name y los campos de QoS.

SUBACK

Este paquete se utiliza como confirmación a los paquetes subscribe.

UNSUBSCRIBE

En estos paquetes el cliente notifica al servidor para que no continúe enviando información sobre uno o más topics.

UNSUBACK

Este paquete se utiliza como confirmación a los paquetes unsubscribe.

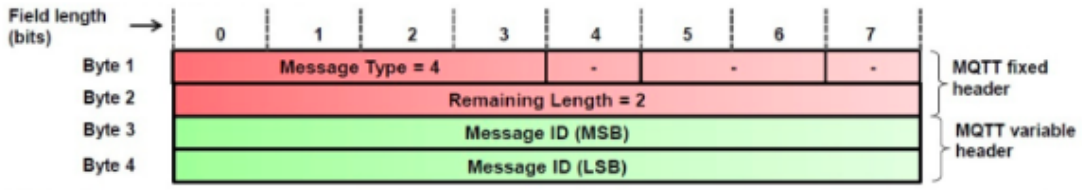


Figura 2.9: Mensaje Puback

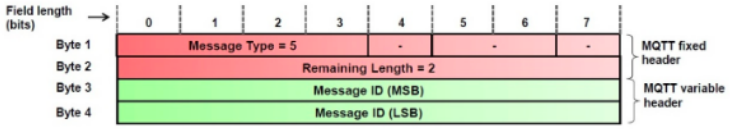


Figura 2.10: Mensaje Pubrec [6]

DISCONNECT/PINGREQ/PINGRESP

Estos mensajes, junto a connect/connack, constituyen los mensajes de sesión del protocolo MQTT. Con el mensaje disconnect se informa que la sesión ha terminado y debe cerrarse, mientras que con pingreq/pingresp la comunicación se mantiene activa para que no se desactive transcurrido un tiempo.

2.1.4. Calidad de Servicio

Como se menciono en apartados anteriores, MQTT posee distintos nivels de validad de servicio o QoS (Quality Of Service), que es el nivel de acuerdo entre emisor y el receptor para garantizar la correcta entrega de un mensaje. Aunque TCP/IP garantiza la entrega de datos, las pérdidas pueden seguir ocurriendo si las conexiones TCP se rompen, por lo que el protocolo MQTT incluye tres niveles distintos de QoS [9]:

- At most once (0): es el nivel mínimo de calidad en el que se utiliza best-eort . La recepción del mensaje no está garantizada ya que los mensajes no se almacenan y,una vez que se envían, el protocolo se olvida de ellos.
- At least once (1): este nivel de calidad garantiza que el mensaje ha llegado al destino al menos una vez. Los mensajes se almacenan hasta que se recibe un puback con el que se confirma la correcta entrega.
- Exactly once (2): es el nivel más alto de calidad de servicio (y el más lento) en el que se asegura que el mensaje ha llegado solo una vez. Esto se consigue con un four-part handshake, es decir, un intercambio de cuatro mensajes para confirmar la llegada.

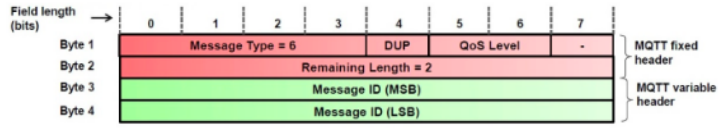


Figura 2.11: Mensaje Pubrel [6]

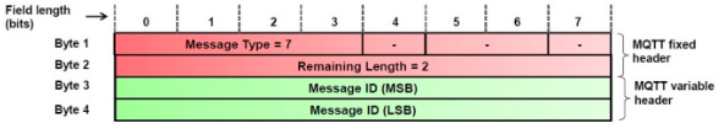


Figura 2.12: Mensaje Pubacomp [6]

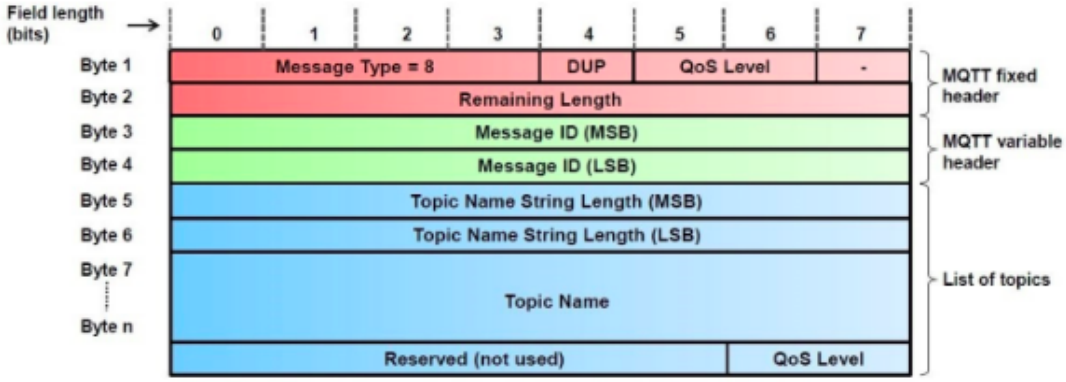


Figura 2.13: Mensaje Subcribe [6]

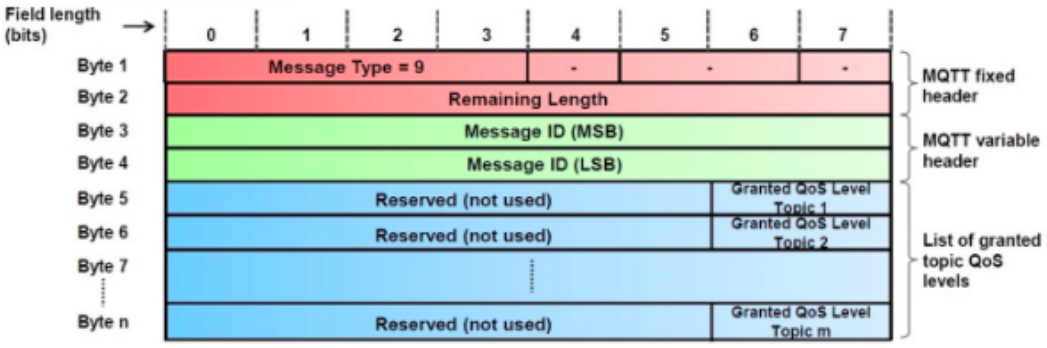


Figura 2.14: Mensaje Suback [6]

2.1.5. Seguridad

El protocolo MQTT ofrece distintas opciones para implementar mecanismos de seguridad y que las comunicaciones no se vean comprometidas. El más recomendado es el uso de MQTT con

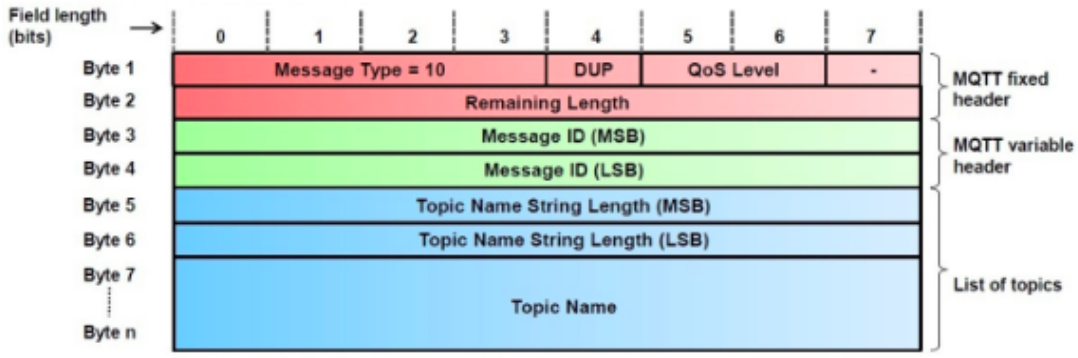


Figura 2.15: Mensaje Unsubscribe [6]

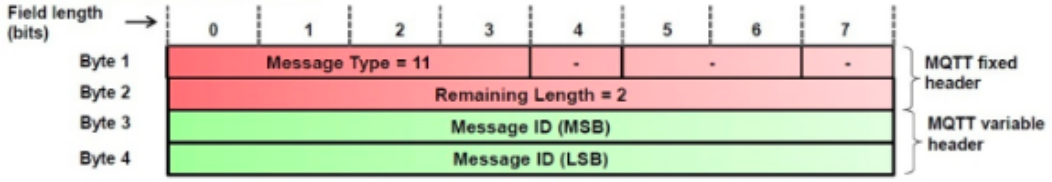


Figura 2.16: Mensaje Unsuback [6]

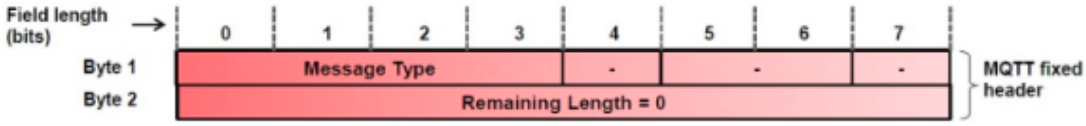


Figura 2.17: Mensaje disconnect/pingreq/pingresp [6]

TLS (secure-mqtt). Se implementa en el servidor y debe usar el puerto TCP 8883, que permite la encriptación en el intercambio de mensajes usando concretamente los algoritmos de cifrado simétrico AES o DES. Además, el uso de TLS provee integridad y privacidad en la comunicación [3]. También se proporciona autenticación con los campos usuario y contraseña de los paquetes y comprobando que los identificadores del paquete o las direcciones corresponden a lo esperado.

2.1.6. Variantes

En este apartado se muestran algunas de las posibles variantes que se utilizan sobre el protocolo MQTT para mejorar algunas de sus características.

- SMQTT (Secure MQTT): esta variante usa TLS con el protocolo por lo que lo hace más seguro, como se ha explicado en el apartado anterior.

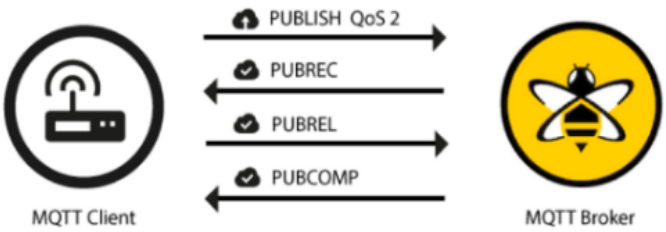


Figura 2.18: Mensaje para Qos nivel 2 [7]

- MQTT-SN (MQTT Sensor Networks): es una variante diseñada específicamente para redes inalámbricas con sensores de poca batería y la mayor parte del tiempo en sleep mode. La diferencia principal es que usa unos paquetes con payload reducido y que utilizan UDP para no necesitar una conexión permanente [10]
- DM-MQTT (Direct Multicast MQTT): es una variación del protocolo para funcionar de forma más eficiente en redes multicast.

2.2. HTTP

2.2.1. Modelo

HTTP es un protocolo de aplicación, es el protocolo de transferencia de hipertexto (HTTP, HyperText Transfer Protocol) es el utilizado en la capa de aplicación de la Web y se encuentra en el corazón de la Web. Está definido en los documentos [RFC 1945 [11]] y [RFC 2616[12]]. El protocolo HTTP se implementa en dos programas: un programa cliente y un programa servidor. El programa cliente y el programa servidor, que se ejecutan en sistemas terminales diferentes, se comunican entre sí intercambiando mensajes HTTP. Ademas define la estructura de estos mensajes y cómo el cliente y el servidor intercambian los mensajes. Antes de continuar con una explicación detallada de HTTP, se va a realizar un breve repaso de la terminología Web.[13]

Una página web (también denominada documento web) consta de objetos. Un objeto es simplemente un archivo (como por ejemplo, un archivo HTML, una imagen JPEG o un video), que puede direccionarse mediante un único URL. La mayoría de las páginas web estan por un archivo base HTML y varios objetos referenciados. Por ejemplo, si una página web contiene texto HTML y cinco imágenes JPEG, entonces la página web contiene seis objetos: el archivo base HTML y las cinco imágenes.[1]

El archivo base HTML hace referencia a los otros objetos contenidos en la página mediante los URL de los objetos. Cada URL tiene dos componentes: el nombre de host del servidor que alberga al objeto y el nombre de la ruta al objeto. Por ejemplo, en el URL:

```
1 http://www.unaEscuela.edu/unDepartmento/imagen.gif
```

"http://www.unaEscuela.edu/" corresponde a un nombre de host y "/unDepartmento/imagen.gif" es el nombre de una ruta. Puesto que los navegadores web (como Internet Explorer y Firefox) implementan el lado del cliente de HTTP, en el contexto de la Web utilizaremos los términos navegador y cliente de forma indistinta. Los servidores web, que implementan el lado del servidor de HTTP, albergan los objetos web, siendo cada uno de ellos direccionable mediante un URL. Entre los servidores web más populares se incluyen Apache y Microsoft Internet Information Server.

HTTP define cómo los clientes web solicitan páginas web a los servidores web y cómo estos servidores transfieren esas páginas web a los clientes. Por otra lado utiliza TCP[1] como su protocolo de transporte subyacente (en lugar de ejecutarse por encima de UDP). El cliente HTTP primero inicia una conexión TCP con el servidor. Una vez que la conexión se ha establecido, los procesos de navegador y de servidor acceden a TCP a través de sus interfaces de socket, en el lado del cliente la interfaz del socket es la puerta entre el proceso cliente y la conexión TCP; en el lado del servidor, es la puerta entre el proceso servidor y la conexión TCP. El cliente envía mensajes de solicitud HTTP a su interfaz de socket y recibe mensajes de respuesta HTTP procedentes de su interfaz de socket. De forma similar, el servidor HTTP recibe mensajes de solicitud de su interfaz de socket y envía mensajes de respuesta a través de la interfaz de su socket. Una vez que el cliente envía un mensaje a su interfaz de socket, el mensaje deja de estar "en las manos" del cliente y pasa "a las manos" de TCP[1]. Este proporciona un servicio de transferencia de datos fiable a HTTP, que implica que cada mensaje de solicitud HTTP enviado por un proceso cliente llegará intacto al servidor; del mismo modo, cada mensaje de respuesta HTTP enviado por el proceso servidor llegará intacto al cliente. Esta es una de las grandes ventajas de una arquitectura en capas: HTTP no tiene que preocuparse por las pérdidas de datos o por los detalles sobre cómo TCP recupera los datos perdidos o los reordena dentro de la Red. Éste es el trabajo de TCP y de los protocolos de las capas inferiores de la pila de protocolos. Es importante observar que el servidor envía los archivos solicitados a los clientes sin almacenar ninguna información acerca del estado del cliente. Si un determinado cliente pide el mismo objeto dos veces en un espacio de tiempo de unos pocos segundos, el servidor no responde diciendo que acaba de servir dicho objeto al cliente; en su lugar, el servidor reenvía el objeto, ya que ha olvidado por completo que ya lo había hecho anteriormente. Dado que un servidor HTTP no mantiene ninguna información acerca de los clientes, se dice que HTTP es un protocolo sin memoria del estado. [1]

2.2.2. HTTP Conexiones Persistentes y no persistentes

En muchas aplicaciones de Internet, el cliente y el servidor están en comunicación durante un periodo de tiempo amplio, haciendo el cliente una serie de solicitudes y el servidor respondiendo a dichas solicitudes. Dependiendo de la aplicación y de cómo se esté empleando la aplicación, las solicitudes pueden, hacerse una tras otra, periódicamente a intervalos regulares o de forma intermitente. Cuando esta interacción cliente-servidor tiene lugar sobre TCP, el desarrollador de la aplicación tiene que tomar una decisión importante: ¿debería cada par solicitud/respuesta enviarse a través de una conexión TCP separada o deberían enviarse todas las solicitudes y sus correspondientes respuestas a través de la misma conexión TCP? Si se utiliza el primer método, se dice que

la aplicación utiliza conexiones no persistentes; si se emplea la segunda opción, entonces se habla de conexiones persistentes.[14]

2.2.3. Mensaje HTTP

Formato de mensaje HTTP

Las especificaciones HTTP [RFC 2616] [12] incluyen las definiciones de los formatos de los mensajes HTTP. A continuación vamos a estudiar los dos tipos de mensajes HTTP existentes: mensajes de solicitud y mensajes de respuesta.

Mensaje de solicitud HTTP

A continuación le proporcionamos un mensaje de solicitud HTTP típico [15]:

```
GET /unadireccion/pagina.html HTTP/1.1
Host: www.unaescuela.edu
Connection: close
User-agent: Mozilla/4.0
Accept-language: fr
```

Podemos aprender muchas cosas si miramos en detalle este sencillo mensaje de solicitud. En primer lugar, podemos comprobar que el mensaje está escrito en texto ASCII normal, por lo que cualquier persona con conocimientos informáticos puede leerlo. En segundo lugar, vemos que el mensaje consta de cinco líneas, cada una de ellas seguida por un retorno de carro y un salto de linea. La última línea va seguida de un retorno de carro y un salto de linea adicionales. Aunque este mensaje en concreto está formado por cinco líneas, un mensaje de solicitud puede constar de muchas más líneas o tener tan pocas como únicamente una. La primera línea de un mensaje de solicitud HTTP se denomina línea de solicitud y las siguientes líneas son las líneas de cabecera. La línea de solicitud consta de tres campos: el campo de método, el campo URL y el campo de la versión HTTP.

El campo que especifica el método puede tomar diferentes valores, entre los que se incluyen GET, POST, HEAD, PUT y DELETE . La inmensa mayoría de los mensajes HTTP utilizan el método GET. Este método se emplea cuando el navegador solicita un objeto, identificando dicho objeto en campo URL. En este ejemplo el navegador está solicitando el objeto /unadireccion/pagina.HTML. El campo correspondiente a la versión se explica por si mismo; en este ejemplo el navegador utiliza la version HTTP/1.1 Analicemos ahora las líneas de cabecera de este ejemplo. La línea de cabecera “Host:www.unaescuela.edu” especifica el host en el que reside el objeto. Podría pensarse que esta línea de cabecera es innecesaria, puesto que ya existe una conexión TCP activa con el host. Pero las cachés proxy web necesitan la información proporcionada por la línea de cabecera del host. Al incluir la línea de cabecera “Connection:close” , el navegador está diciendo al servidor que no desea molestarse en trabajar con conexiones persistentes, sino que desea que el servidor cierre la conexión después de enviar el objeto solicitado. La línea de cabecera User-

agent especifica el agente de usuario, es decir, el tipo de navegador que está haciendo la solicitud al servidor. En este caso, el agente de usuario es Mozilla/4.0, un navegador de Netscape. Esta línea de cabecera resulta útil porque el servidor puede enviar versiones diferentes del mismo objeto a los distintos tipos de agentes de usuario (el mismo URL direcciona a cada una de las versiones). Por último, la línea de cabecera Accept-language indica que el usuario prefiere recibir una versión en francés del objeto, si tal objeto existe en el servidor; en caso contrario, el servidor enviará la versión por defecto. La línea de cabecera Accept-language sólo es una de las muchas cabeceras de negociación del contenido disponibles en HTTP.

Mensaje de respuesta HTTP

A continuación proporcionamos un mensaje de respuesta HTTP típico. Este mensaje de respuesta podría ser la respuesta al mensaje de solicitud ejemplo que acabamos de ver.

```
HTTP/1.1 200 OK
Connection: close
Dates Sat, 07 Jul 2007 12 00 15 GKT
Server: Apache/1.3.0 (Unix)
Last-Modified: Sun, 6 May 2007 09:23:24 GMT
Content-Length: 6821
Content-Type: text/html
( datos datos datos ... )
```

Examinando detenidamente este mensaje de respuesta se pueden observar tres secciones: una línea de estado inicial, seis líneas de cabecera y el cuerpo de entidad. El cuerpo de entidad es la parte más importante del mensaje, ya que contiene el objeto solicitado en sí (representado por la línea datos datos datos datos datos). La línea de estado contiene tres campos: el que especifica la versión del protocolo, el correspondiente al código de estado y el tercero que contiene el mensaje explicativo del estado correspondiente. En este ejemplo, la línea de estado indica que el servidor está utilizando HTTP/1.1 y que todo es correcto (OK); es decir, que el servidor ha encontrado y está enviando el objeto solicitado.

Veamos ahora las líneas de cabecera. El servidor utiliza la línea “Connection: close” para indicar al cliente que va a cerrar la conexión TCP después de enviar el mensaje . La línea de cabecera Date indica la hora y la fecha en la que se creó la respuesta HTTP y fue enviada por el servidor. Se puede observar que no especifica la hora en que el objeto fue creado o modificado por última vez; es la hora en la que el servidor recupera el objeto de su sistema de archivos, inserta el objeto en el mensaje de respuesta y lo envía.

- Linea de cabecera de archivos inserta el objeto en el mensaje de respuesta y lo envía.
- Linea de cabecera Server identifica que el mensaje fue generado por un servidor web Apache; esta es análoga a linea de cabecer User-agent
- Linea de cabecera User-agent del mensaje de solicitud HTTP.

- Linea de Last-Modified especifica la hora y la fecha en que el objeto fue creado o modificado por última vez.
- Linea de cabecera de Last-Modified resulta fundamental para el almacenamiento en caché del objeto, tanto en el cliente local como en los servidores de almacenamiento en caché de la red (también conocidos como servidores proxy).
- Linea Content-Length especifica el número de bytes del objeto que está siendo enviado.
- Linea Content-type indica que el objeto especificado en el cuerpo de entidad es texto HTML. (El tipo de objeto está indicado oficialmente por la línea de cabecera Content-Type y no por la extensión de archivo)

Una vez dicho esto pasaremos a comentar los códigos de estado de una respuesta http:

- 200 OK: La solicitud se ha ejecutado con éxito y se ha devuelto la información en el mensaje de respuesta
- 301 Moved Permanently: El objeto solicitado ha sido movido de forma permanente; el nuevo URL se especifica en la linea de cabecera Location del mensaje de respuesta. El software cliente recuperará automáticamente el nuevo URL
- 400 Bad Request: Se trata de un código de error genérico que indica que la solicitud no ha sido comprendida por el servidor.
- 404 Not Found: El documento solicitado no existe en el servidor.
- 505 HTTP Version Not Supported: La versión de protocolo HTTP solicitada no es soportada por el servidor

Este protocolo lo podemos definir como un modelo cliente-servidor. En este modelo, el servidor tiene los datos y responde a los pedidos del cliente. En una arquitectura cliente-servidor siempre existe un host activo, denominado servidor, que da servicio a las solicitudes de muchos otros hosts, que son los clientes. Los hosts clientes pueden estar activos siempre o de forma intermitente. Un ejemplo clásico es la Web en la que un servidor web siempre activo sirve las solicitudes de los navegadores que se ejecutan en los hosts clientes. Cuando un servidor web recibe una solicitud de un objeto de un host cliente, responde enviándole el objeto solicitado. Por esto que, con la arquitectura cliente-servidor, los clientes no se comunican directamente entre sí; por ejemplo, en la aplicación web, dos navegadores no se comunican entre sí. Normalmente, en una aplicación cliente-servidor un único host servidor es incapaz de responder a todas las solicitudes de sus clientes. Por ejemplo, el sitio de una red social popular puede verse rápidamente desbordado si sólo dispone de un servidor para gestionar todas las solicitudes. Por esta razón, en las arquitecturas cliente-servidor suele utilizarse una agrupación (cluster) de hosts, que a veces se denomina centro de datos, para crear un servidor virtual de gran capacidad. Los servicios de aplicaciones basadas en una arquitectura cliente-servidor a menudo precisan una infraestructura intensiva, ya que requieren que los proveedores de servicios compren, instalen y mantengan granjas de servidores

2.2.4. HTTP sobre TCP

Como se menciono previamente HTTP utiliza TCP como su protocolo de transporte subyacente (en lugar de ejecutarse por encima de UDP). El cliente HTTP primero inicia una conexión TCP con el servidor. Una vez que la conexión se ha establecido, los procesos de navegador y de servidor acceden a TCP a través de sus interfaces de socket

2.2.5. Tipos de mensajes HTTP / REST

Introducción

Tras muchos años de intentar crear servicios web basados en tecnologías RPC, tales como CORBA o SOAP, la industria del desarrollo de software se encontraba en un punto muerto. Se había conseguido el gran logro de que un servicio implementado en .NET consiguiera comunicarse con uno escrito en Java, o incluso con otro hecho a base de COBOL, sin embargo todo no resultaron alcanzar, se había invertido cantidades de recursos en distintas tecnologías, frameworks y herramientas, y las recompensas eran escasas. Como consecuencia de esto las compañías se encontraban con problemas. Por un lado la mantenibilidad de la base de código resultante era bastante baja. Se necesitaban complejos IDEs (aplicaciones o entornos de desarrollo integrado) para generar el código necesarias para interoperar. Dificultando esto a los desarrolladores la posibilidad de que se descubriera algún bug en el desarrollo realizado.

Por otro lado, para depurar cualquier problema de interoperabilidad, había que observar los paquetes de HTTP. Cuando la situación se hizo mas dificultosa, y algunos gigantes de la informática como Amazon, Google o Twitter necesitaron interoperabilidad a escala global y barata, se descubrió el camino al futuro mirando hacia el pasado, y surgió REST.

El caso de uso más sencillo al diseñar servicios REST con HTTP se produce cuando dichos servicios publican operaciones CRUD sobre nuestra capa de acceso a datos. El acrónimo CRUD responde a “Create Read Update Delete” y se usa para referirse a operaciones e mantenimiento de datos, normalmente sobre tablas de un gestor relacional de base de datos. En este estilo de diseño existen dos tipos de recursos: entidades y colecciones

Las colecciones actúan como listas o contenedores de entidades, y en el caso puramente CRUD se suelen corresponder con tablas de base de datos. Normalmente su URL se deriva del nombre de la entidad que contienen. Por ejemplo, http://www.server.com/rest/libro sería una buena URL para obtener la colección de todos los libros dentro de un sistema. Para cada colección se suele usar el siguiente mapeo de métodos HTTP a operaciones:

Método	HTTP Operación
GET	Leer todas las entidades dentro de la colección
PUT	Actualización mútiple y/o masiva
DELETE	Borrar la colección y todas sus entidades
POST	rear una nueva entidad dentro de la colección

Las entidades son ocurrencias o instancias concretas, que viven dentro de una colección. La URI de una entidad se suele modelar concatenado a la URI de la colección correspondiente un

identificador de entidad. Este identificador sólo necesita ser único dentro de dicha colección. Ej. http://www.server.com/rest/libro/2314 sería el libro cuyo identificador es 2314. Normalmente se suele usar la siguiente convención a la hora de mapear métodos HTTP a operaciones cuando se trabaja con entidades.

Método	HTTP Operación
GET	Leer los datos de una entidad en concreto
PUT	Actualizar una entidad existente o crearla si no existe
DELETE	Borrar una entidad en concreto
POST	Añadir información a una entidad ya existenten

A continuación, en las siguientes secciones, veremos más en detalle algunas opciones de diseño para cada operación CRUD.

2.2.6. GET

Todas las operaciones de lectura y consulta deben hacerse con el método GET, ya que según la especificación HTTP, indica la operación de recuperar información del servidor. El caso más sencillo es el de leer la información de una entidad, que se realiza haciendo un GET contra la URI de la entidad. Esto no tiene mucho más misterio, salvo en el caso de que el volumen de datos de la entidad sea muy alto. En estos casos es común que queramos recuperar los datos de la entidad pero sólo para consultar una parte de la información y no toda, con lo que estamos descargando mucha información que no nos es útil [16].

Una posible solución es dejar sólo en esa entidad los datos de uso más común, y el resto dividirlo en varios recursos hijos. De esta manera cuando el cliente lea la entidad, sólo recibirá los datos de uso más común y un conjunto de enlaces a los recursos hijos, que contienen los diferentes detalles asociados a ésta. Cada recurso hijo puede ser a su vez o una entidad o una colección. En general se suele seguir la convención de concatenar el nombre del detalle a la URI de la entidad padre para conseguir la URI de la entidad hija. Por ejemplo, dada una entidad "rest/libro/23424-dsdff", si se le realiza un GET, recibiríamos un documento, con el título, los autores, un resumen, valoración global, una lista de enlaces a los distintos capítulos, otra para los comentarios y valoraciones, etc. Una opción de diseño es hacer que todos los libros tengan una colección de capítulos como recurso hijo. Para acceder al capítulo 3, podríamos modelar los capítulos como una colección y tener la siguiente URL: "/rest/libro/23424-dsdff/capitulo/3".

Con este diseño tenemos a nuestra disposición una colección en "/rest/libro/23424-dsdff/capitulo", con la cual podemos operar de forma estándar, para insertar, actualizar, borrar o consultar capítulos. Este diseño es bastante flexible y potente. Otro diseño, más simple, sería no tener esa colección intermedia y hacer que cada capítulo fuera un recurso que colgara directamente del libro, con lo que la URI del capítulo 3 sería: /rest/libro/23424-dsdff/capitulo3. Este diseño es más simple y directo y no nos ofrece la flexibilidad del anterior.

Volviendo al problema de tener una entidad con un gran volumen de datos, existe otra solución en la que no es necesario descomponerla en varios recursos. Se trata simplemente de hacer un GET

a la URI de la entidad pero añadiendo una query string. Por ejemplo, si queremos ir al capítulo número 3, podemos hacer GET sobre /rest/libro/23424-dsdff?capitulo=3. De esta forma hacemos una lectura parcial de la entidad, donde el servidor devuelve la entidad libro, pero con sólo el campo relativo al capítulo 3. A esta técnica la llamo slicing. El usar slicing nos lleva a olvidarnos de esta separación tan fuerte entre entidad y colección, ya que un recurso sobre el que podemos hacer slicing es, en cierta medida, una entidad y una colección al mismo tiempo.

Como se aprecia REST es bastante flexible y nos ofrece diferentes alternativas de diseño, el usar una u otra depende sólo de lo que pensemos que será más interoperable en cada caso. Un criterio sencillo para decidir si hacer slicing o descomponer la entidad en recursos de detalle, es cuantos niveles de anidamiento vamos a tener. En el caso del libro, ¿se accederá a cada capítulo como un todo o por el contrario el cliente va a necesitar acceder a las secciones de cada capítulo de forma individual? En el primer caso el slicing parece un buen diseño, en el segundo no lo parece tanto. Si hacemos slicing, para acceder a la sección 4 del capítulo 3, tendríamos que hacer: "/rest/libro/23424-dsdff?capitulo=3seccion=4". Este esquema de URI es menos semántico, y además nos crea el problema de que puede confundir al cliente y pensar que puede hacer cosas como esta: "/rest/libro/23424-dsdff?seccion=4" ¿Qué devolvemos? ¿Una lista con todas las secciones 4 de todos los capítulos? ¿Un 404 no encontrado? Sin embargo en el diseño orientado a subrecursos es claro, un GET sobre "/rest/libro/23424-dsdff/capitulo/3/seccion/4" nos devuelve la sección 4 del capítulo 3, y sobre "/rest/libro/23424-dsdff/seccion/4" nos debería devolver 404 no encontrado, ya que un libro no tiene secciones por dentro, sino capítulos. Otra desventaja del slicing es que la URI no es limpia, y el posicionamiento en buscadores de nuestro recurso puede ser afectado negativamente por esto (sí, un recurso REST puede tener SEO, ya lo veremos más adelante). A veces no tenemos claro cual va a ser el uso de nuestra API REST. En estos casos es mejor optar por el modelo más flexible de URIs, de forma que podamos evolucionar el sistema sin tener que romper el esquema de URIs, cosa que rompería a todos los clientes. En este caso el sistema más flexible es descomponer la entidad en recursos de detalle, usando colecciones intermedias si es necesario. Recordad que se tome la decisión que se tome, esta no debe afectar al diseño interno del sistema. Por ejemplo, si decidimos no descomponer la entidad en recursos hijos, eso no significa que no pueda internamente descomponer una supuesta tabla de libros, en varias tablas siguiendo un esquema maestro detalle. Y viceversa, si decido descomponer la entidad en varios subrecursos, podría decidir desnormalizar y tenerlo todo en una tabla, o quizás no usar tablas sino una base de datos documental. Estas decisiones de implementación interna, guiadas por el rendimiento y la mantenibilidad del sistema, deben ser invisibles al consumidor del servicio REST.

2.2.7. PUT

A la hora de actualizar los datos en el servidor podemos usar dos métodos, PUT y POST. Según HTTP, PUT tiene una semántica de UPSERT, es decir, actualizar el contenido de un recurso, y si éste no existe crear un nuevo recurso con dicha información en la URI especificada. POST por el contrario puede usarse para cualquier operación que no sea ni segura ni idempotente, normalmente para añadir un trozo de información a un recurso o bien crear un nuevo recurso. Si queremos actualizar una entidad lo más sencillo es realizar PUT sobre la URI de la entidad, e incluir en el

cuerpo de la petición HTTP los nuevos datos [17]. Por ejemplo:

```
PUT /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json
{
 "author": "Enrique Gomez Salas",
 "title": "Desventuras de un informatico en Paris",
 "genre": "scifi",
 "price": { "currency": "$", "amount": 50}
}
```

Y la respuesta es muy escueta: 1 HTTP/1.1 204 No Content

Esto tiene como consecuencia que el nuevo estado del recurso en el servidor es exactamente el mismo que el que mandamos en el cuerpo de la petición. La respuesta puede ser 204 o 200, en función de si el servidor decide enviarnos como respuesta el nuevo estado del recurso. Generalmente sólo se usa 204 ya que se supone que los datos en el servidor han quedado exactamente igual en el servidor que en el cliente. Con la respuesta 204 se pueden incluir otras cabeceras HTTP con meta-información, tales como ETag o Expires. Sin embargo en algunos casos, en los que el recurso tenga propiedades de sólo lectura que deban ser recalculadas por el servidor, puede ser interesante devolver un 200 con el nuevo estado del recurso completo, incluyendo las propiedades de solo lectura. Es decir, la semántica de PUT es una actualización donde reemplazamos por completo los datos del servidor con los que enviamos en la petición

2.2.8. DELETE

Para borrar una entidad o una colección, simplemente debemos hacer DELETE contra la URI del recurso.

```
DELETE /rest/libro/465 HTTP/1.1
Host: www.server.com
```

Y la respuesta: 1 HTTP/1.1 204 No Content

Normalmente basta con un 204, pero en algunos casos puede ser útil un 200 para devolver algún tipo de información adicional. Hay que tener en cuenta que borrar una entidad, debe involucrar un borrado en cascada en todas las entidades hijas. De la misma forma, si borramos una colección se deben borrar todas las entidades que pertenezcan a ella. Otro uso interesante es usar una query string para hacer un borrado selectivo [18]. Por ejemplo:

```
 DELETE /rest/libro?genero=scifi HTTP/1.1 2
 Host: www.server.com
```

Borraría todos los libros de ciencia ficción. Mediante este método podemos borrar sólo los miembros de la colección que cumplen la query string

2.2.9. POST

Una forma de crear nuevos recursos es mediante POST. Simplemente hacemos POST a una URI que no existe, con los datos iniciales del recurso y el servidor creará dicho recurso en la URI especificada. Por ejemplo, para crear un nuevo libro:

```
PUT /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json
{
"author": "Enrique Gomez Salas",
"title": "Desventuras de un informatico en Paris",
"genre": "scifi",
"price": { "currency": "   ", "amount": 50}
}
```

Notase que la petición es indistinguible de una actualización. El hecho de que se produzca una actualización o se cree un nuevo recurso depende únicamente de si dicho recurso, identificado por la URL, existe ya o no en el servidor [19]. La respuesta:

```
HTTP/1.1 201 Created
Location: http://www.server.com/rest/libro/465
Content-Type: application/json;charset=utf-8
{
"id": "http://www.server.com/rest/libro/465",
"author": "Enrique G mez Salas",
"title": "Desventuras de un inform tico en Paris",
"genre": "scifi",
"price": { "currency": "$", "amount": 50}
}
```

Notase que el código de respuesta no es ni 200 ni 204, sino 201, indicando que el recurso se creó con éxito. Opcionalmente, como en el caso del ejemplo, se suele devolver el contenido completo del recurso recién creado. Es importante fijarse en la cabecera Location que indica, en este caso de forma redundante, la URL donde se ha creado el nuevo recurso.

2.3. Comparación HTTP y MQTT

En esta sección se realizara una breve descripción sobre la comparación de estos protocolos [20] en función de su performance. Se van a observar los gráficos de latencia y trafico de red. La prueba realizadas de tiempos se hace ejecutando 20 comunicaciones que indican el envío de un mensaje en ambos protocolos, y la de tráfico se realiza con 30 comunicaciones en ambos protocolos.

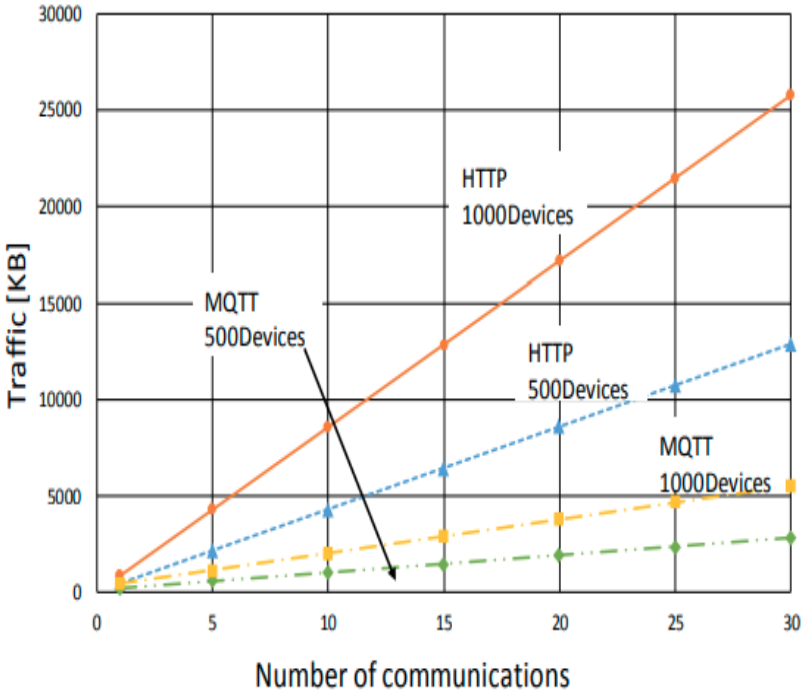


Figura 2.19: Tráfico HTTP y MQTT

Como se puede observar en la figura 2.19 la comparación de tráfico con respecto a la cantidad de comunicaciones es mayormente menor para el protocolo MQTT con las mismas cantidad de dispositivos utilizados que HTTP. Por otro lado, en los gráficos 2.20 y 2.21 se observa como es la latencia para ambos protocolos, claramente se ve que para el protocolo MQTT la latencia es menor, ya que el tiempo promedio se encuentra cerca de 0.00106 segundos, a diferencia de HTTP que se encuentra en 4.9915 segundos[20] .

2.4. Arquitectura de aplicaciones distribuidas

A continuación se describirá el marco teórico sobre las arquitecturas de software distribuidas, ya que la aplicación del caso de estudio esta montada sobre una arquitectura de este tipo. Una arquitectura distribuida es una aplicación con distintos componentes que se ejecutan en servidores

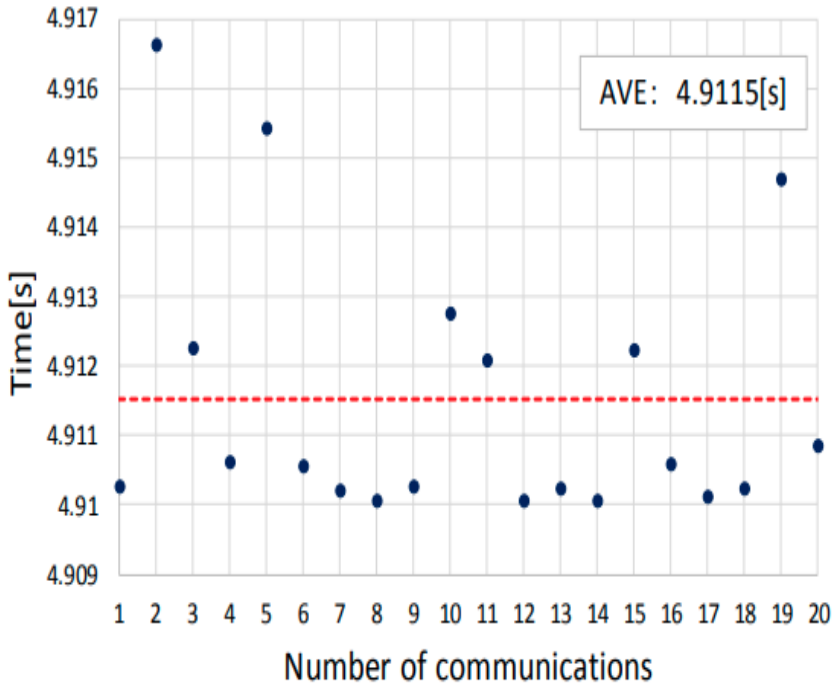


Figura 2.20: Tiempo HTTP

separados, comunicados entre si a través de la red en los protocolos previamente mencionados, normalmente en diferentes plataformas conectadas. Esta distribución se refiere a la construcción de software por partes, a las cuales le son asignadas un conjunto específico de responsabilidades dentro de un sistema. Esta distribución como bien enuncia la definición formal, habla de que las partes o componentes se encuentran en entornos separados, sin embargo, lo que tiene implícito esta definición, es que para realizar esta separación física primero debe tenerse clara la separación lógica de las partes de una aplicación, esto quiere decir que pro-gramáticamente existe una forma de separar o agrupar los componentes. La separación física no es en todas la ocasiones en "máquinas diferentes" de acuerdo a la arquitectura también puede ser la ubicación de un conjunto de funcionalidades en archivos, rutas o montadas sobre tecnologías diferentes dentro de la misma máquina. Cuando hablamos de distribución lógica lo entenderemos como separación por "capas" y cuando hablemos de distribución física usaremos el término separación en "niveles". Una capa puede contener muchos componentes, un mismo componente puede ubicarse en varias capas de acuerdo a su naturaleza y a las consideraciones explícitas de la arquitectura.

Una arquitectura en un ambiente distribuido describe la estructura y la organización de los componentes del software, sus propiedades y la conexión entre ellos para formar el sistema; la cantidad y la granularidad de comunicación que se necesita para la interacción y los protocolos de interfaz usada por la comunicación. En una aplicación distribuida en n-capas los diferentes elementos que integran la aplicación se agrupan de forma lógica según la funcionalidad que reciben

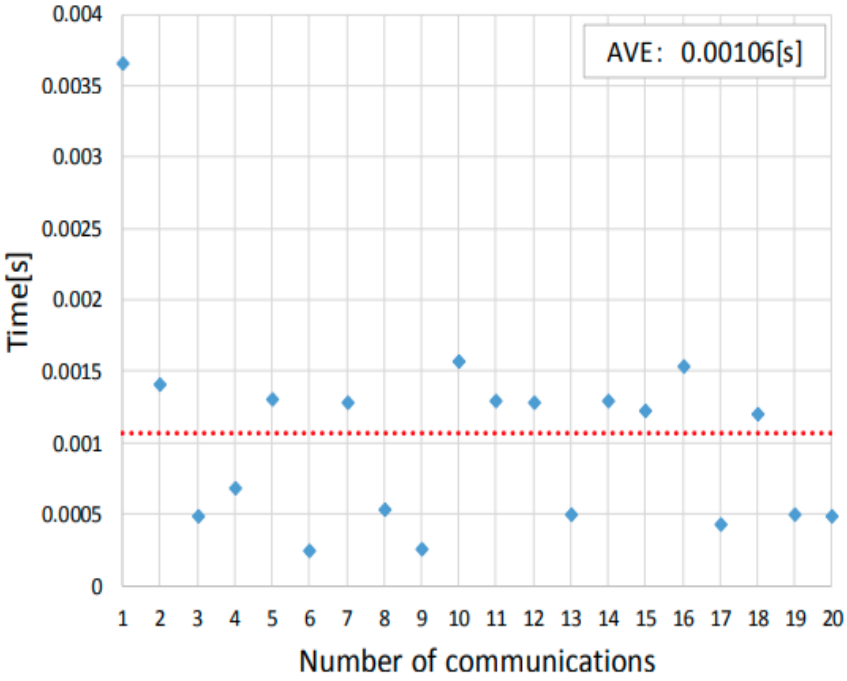


Figura 2.21: Tiempo MQTT

o suministran al o desde el resto de los elementos. Así, algunos elementos se limitarán a recibir peticiones de datos mientras que otros interactuarán con el usuario y su función será principalmente la de solicitar a otros elementos la información que el usuario precisa. Una vez agrupada la funcionalidad en capas lógicas es fácil relacionar unas con otras. El usuario interactúa con la capa de presentación, solicitando datos o desencadenando acciones. Las solicitudes serán atendidas por la capa de negocios, que se encargará de su gestión o de la traducción necesaria para que la capa de servidor realice la tarea solicitada. La capa de servidor debe proporcionar datos los cuales se devolverán a la capa de negocios, la cual los gestionará o transmitirá a la capa de presentación.

Es importante notar que el esquema que mostramos es un esquema lógico, no físico. El modo de distribuir físicamente las capas se corresponderá con el esquema lógico en todo o en parte, pero no necesariamente existirá una correspondencia exacta entre la distribución lógica de los elementos y su distribución física. La capa de negocios podría residir en diferentes máquinas, por ejemplo, o las entidades de negocio y la capa de servidor podrían formar parte de la misma maquina.

2.5. Arquitectura de microservicios

En esta sección se describirá la utilidad de los microservicios, ya que estos se montan sobre una arquitectura distribuida como lo mencionado en la sección anterior. Los microservicios son tanto un estilo de arquitectura como un modo de programar software. Con los microservicios, las

aplicaciones se dividen en sus elementos más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, los microservicios son elementos independientes que funcionan en conjunto para llevar a cabo las mismas tareas. Cada uno de esos elementos o procesos es un microservicio. Este enfoque de desarrollo de software valora el nivel de detalle, la sencillez y la capacidad para compartir un proceso similar en varias aplicaciones. Es un elemento fundamental de la optimización del desarrollo de aplicaciones hacia un modelo nativo de la nube.

¿Pero cuáles son las ventajas de utilizar una infraestructura de microservicios? En pocas palabras, el objetivo es distribuir sistemas de software de calidad con mayor rapidez, lo cual es posible gracias a los microservicios, pero también se deben considerar otros aspectos.

Dividir las aplicaciones en microservicios no es suficiente; es necesario administrarlos, coordinarlos y gestionar los datos que crean y modifican.

Los microservicios son un tipo de arquitectura que sirve para diseñar aplicaciones. Lo que distingue a la arquitectura de microservicios de los enfoques tradicionales y monolíticos es la forma en que desglosa una aplicación en sus funciones principales. Cada función se denomina servicio y se puede diseñar e implementar de forma independiente. Esto permite que funcionen separados (y también, fallen por separado) sin afectar a los demás.

¿Cuáles son los beneficios de una arquitectura de microservicios? A través del desarrollo distribuido, los microservicios potencian a los equipos y las rutinas. También puede desarrollar múltiples microservicios de forma simultánea. Gracias a ello, más desarrolladores pueden trabajar en la misma aplicación simultáneamente para reducir el tiempo invertido en el desarrollo. A continuación se listan otros beneficios de estos tipos de arquitectura:

Aplicaciones listas para comercializarse más rápidamente

Debido a la reducción de los ciclos de desarrollo, una arquitectura de microservicios permite que la implementación y las actualizaciones se realicen más rápidamente.

Gran capacidad de expansión

A medida que crece la demanda de ciertos servicios, es posible realizar implementaciones en distintos servidores e infraestructuras para satisfacer sus necesidades.

Capacidad de recuperación

Si estos servicios independientes están bien diseñados, no pueden afectar a los demás. Esto significa que si una parte falla, no afecta a toda la aplicación, a diferencia del modelo de aplicaciones monolíticas.

Facilidad de implementación

Debido a que las aplicaciones basadas en microservicios son más modulares y más pequeñas que las aplicaciones monolíticas tradicionales, ya no es necesario preocuparse por su implementa-

ción. Se requiere más coordinación, pero las ventajas son enormes.

Accesibilidad

Dado que las publicaciones más grandes se desglosan en piezas más pequeñas, los desarrolladores pueden comprender, actualizar y mejorar más fácilmente esas piezas; de esta manera, se obtienen ciclos de desarrollo más rápidos, especialmente cuando se combinan con metodologías de desarrollo ágiles. Aplicaciones más abiertas debido al uso de API políglotas, los desarrolladores tienen la libertad de elegir los mejores lenguajes y tecnologías para la función que se necesita.

2.6. Arquitectura orientada a eventos

En esta sección se explicará cómo funcionan las arquitecturas orientadas a eventos que cumplen el patrón "Publish / Subscribe Messaging". Esta arquitectura con la combinación de las mencionadas previamente es la utilizada en el caso de estudio que va a ser implementada bajo el protocolo MQTT.

Publish / Subscribe Messaging: Patrón de uso dentro de la tipología de arquitectura de “Cola de mensajes”, utilizado para la comunicación entre aplicaciones. Este se engloba más en el movimiento de información, aunque también puede cumplir aspectos de preparación o modificación. También se denomina “publish/subscribe”, “publicador/suscriptor” o “productor/consumidor”. Para su funcionamiento existe un elemento “publisher” que al generar un dato (message) no lo dirige o referencia específicamente a un “subscriber” en concreto, es decir, no lo envía de forma directa a la dirección del subscriber. Para ello se dispone de listas de temas/topics publicados específicos y un conjunto de suscriptores, el productor trata de clasificar el mensaje en base a una tipología, lo pone en la lista de un tema específico y el receptor se suscribe a la listas para recibir ese tipo de mensajes. Este tipo de comunicación se utiliza sobre todo para la comunicación asíncrona y permite para esto distintos tipos de configuración

- Tradicional: Cada suscriptor está asociado a uno o varios topic en concreto. Existen muchas variaciones:
- Grupos de consumo: Los suscriptores se pueden agrupar por grupo, este rupo está escuchando un topic y sólo un miembro del grupo tendrá la capacidad de atender el mensaje.
- Radio Difusión: Todos los suscriptores que están escuchando el topic reciben el mensaje (cada suscriptor es responsable de interpretar el mensaje de forma independiente).

Por otro lado requiere de otra pieza de software que sirve de intermediario (broker) donde se publican los tópicos. Este es el clasificador de temas y los suscriptores sólo reciben datos relacionados con esos temas, y el contenido de estos es el mensaje que se envía a través del tópico mencionado.

Capítulo 3

Caso de Estudio

3.1. Introducción

En este capítulo se mostrará el desarrollo de la aplicación de monitoreo mencionada en el primer capítulo, en la cual se utilizó el protocolo Iot MQTT para la recepción de información de las aplicaciones a monitorear. Esta aplicación de monitoreo se desarrolló principalmente utilizando una arquitectura orientada a eventos para la recepción de los mismos y se utilizó un bus de eventos implementado con una librería llamada "mosquitto". Luego de esto en la seccion 3.5 se realizara una comparativa del protocolo MQTT sobre HTTP a partir de los atributos mencionados de cada uno en el marco teórico con el fin de determinar cuando es conveniente utilizar un protocolo u otro, para esto cabe destacar que esta plataforma en producción recibe aproximadamente 10000 request por minuto, por lo que es necesario realizar las pruebas con estos request o mas para poder determinar que protocolo tiene mejor performance.

3.2. Modelo de negocios

Antes de abordar sobre el desarrollo de la aplicación, se hará una breve introducción sobre el modelo de negocios de **despegar.com**, más específicamente sobre la unidad de negocios que se utilizó en esta tesina, llamada Agencias Afiliadas. La empresa **despegar.com** es conocida por ser una plataforma web de ventas de productos de turismo al público, esta empresa posee otro canal de ventas además de su plataforma web, llamado programa de Agencias Afiliadas, donde las agencias de turismo pueden disponer de una plataforma web personalizada con su propia marca e imagen para realizar ventas de productos turísticos a sus clientes, estos productos son específicamente Alojamientos, Vuelos y Paquetes turísticos. De esta forma las agencias disponen de su propia web para poder realizar sus propias ventas.

A continuación se mostrará un ejemplo de una plataforma brindada a una Agencia Afiliada por la unidad de negocio de **despegar.com**. Esta agencia de turismo configurada para esta tesina, es llamada "PITUCON", y como se puede apreciar configuró su sitio web de la forma que su marca lo requiere.

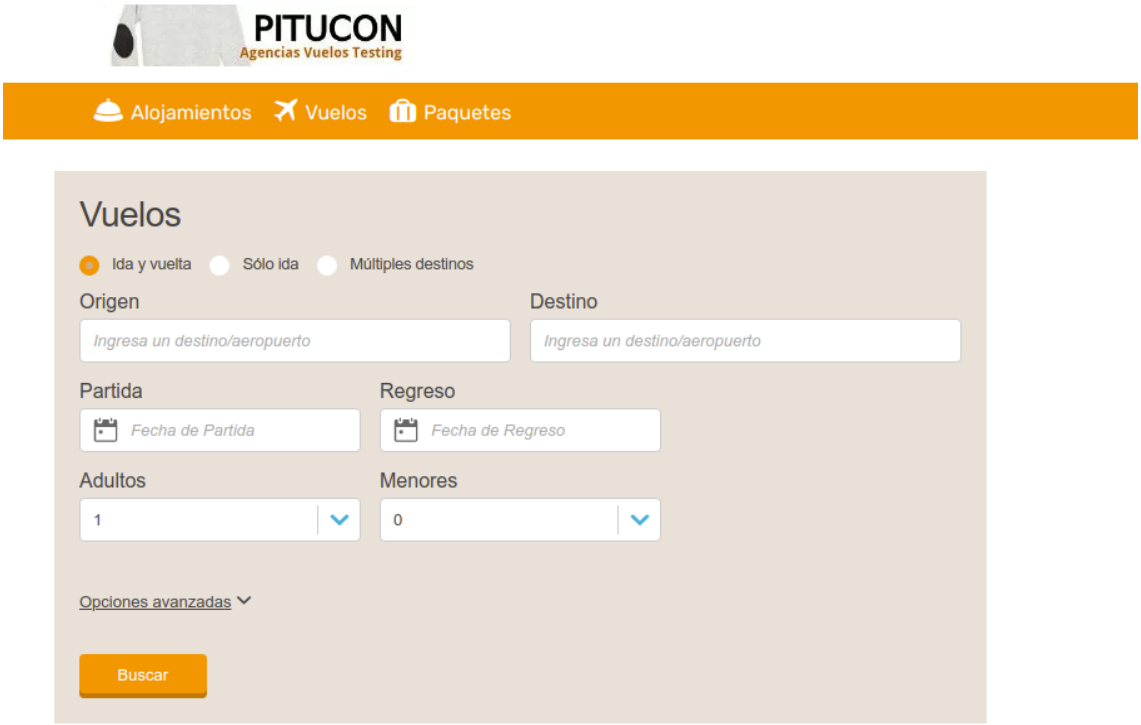


Figura 3.1: Caja de Busqueda

Como se puede observar en la Figura 3.1, el sitio web está formado por tres partes. Un encabezado donde se sitúa el nombre y logo de la agencia, un menú de opciones donde se puede seleccionar el producto a buscar (alojamiento, vuelo o paquetes) y una caja de búsqueda para que el usuario encuentre el destino deseado. De cada producto ofrecido el usuario realizará la búsqueda deseada, donde esta acción lo redireccionará a una página de resultados (Figura 3.2), de acuerdo a este resultado se puede acceder más en detalle al ítem buscado para luego acceder a una página de checkout donde finalmente se efectuará la compra (Figura 3.3). Más adelante se explicará los casos de uso en detalle de cada uno de estos flujos mencionados.

Una vez que el usuario realiza la compra, esta misma se realiza en esta plataforma con el desconocimiento que el usuario la realiza en **despegar.com**. Esta venta pasará a ser parte del sistema de despegar.

3.2.1. Arquitectura de aplicaciones de Agencias Afiliadas

A continuación haremos una descripción de la arquitectura de Aplicaciones de Agencias Afiliadas. La plataforma tecnológica de Agencia está desarrollada internamente por un conjunto de aplicaciones que interactúan entre sí. De esta forma se desarrolló un un par de aplicaciones para la venta de alojamientos, una aplicación para servir la vista al browser que llamaremos frontend y una aplicación que responde a la lógica de negocios y se comunica con un repositorio de datos,

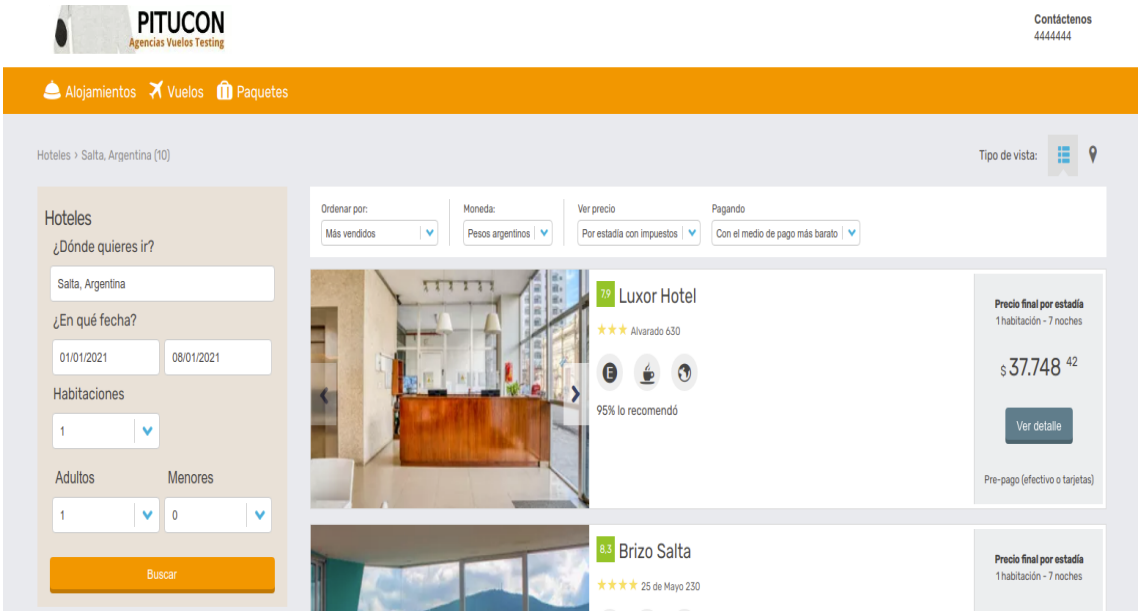


Figura 3.2: Resultado de Búsqueda

esta aplicación la llamaremos backend. Este par de aplicaciones mencionado existe para cada tipo de producto, uno para vuelo, unos para hoteles y uno para para paquetes.

En la figura 3.4 se puede observar lo mencionado anteriormente y la vez cuales son los flujos de comunicación entre ellas, como se puede ver las aplicaciones de un producto no interactúan con las aplicaciones de otro. Cada aplicación frontend se comunica con su backend a través del protocolo de red HTTP (como se describió en la sección 2.4 y 2.5). El modulo de Backend exponen una API Rest que contienen los servicios sobre las distintas entidades que maneja para ser consumidos por el modulo frontend.

3.2.2. Casos de uso

En el siguiente apartado se realizará una descripción de los casos de usos donde el usuario final interactúa con las aplicaciones de Agencias Afiliadas en su conjunto, estos casos de usos más adelante serán los identificados para monitorear por la aplicación de monitoreo. Un usuario de la plataforma web de Agencias Afiliadas la primer acción que realiza es la de seleccionar un producto e ingresar los parámetros de búsqueda, estos parámetros dependen del producto a buscar, en el caso de un alojamiento se ingresa la ciudad destino y las fechas deseadas en las que el usuario se quiera hospedar, y para el caso de un vuelo se ingresa el país o ciudad destino y las fechas de ida y vuelta del mismo. Esta acción mencionada lo llamaremos caso de uso de “Búsqueda de Producto”. Una vez que el usuario ingresó estos parámetros de búsqueda, se dispara el caso de uso mencionado, y el resultado de ejecutar este caso de uso es visualizar en el browser del usuario el listado de todos los posibles destino que cumplen con la condición de la búsqueda. De este listado generado por el caso de uso inicial , el usuario debe seleccionar el item o resultado deseado, con que dispara el

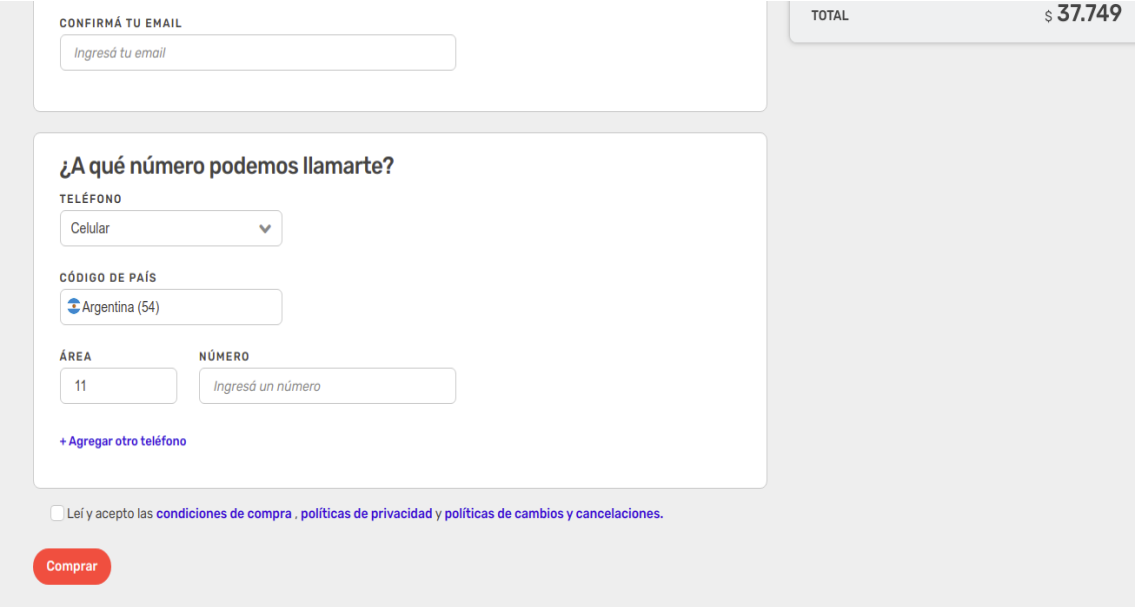


Figura 3.3: Pagina de Checkout

segundo caso de uso que es “Detalle de un Producto”. Una vez que el usuario visualiza el detalle del producto y este lo satisface presiona el botón “comprar” que dispara el siguiente caso de uso que es el de “Página de Checkout”. Luego el usuario ya en la página de checkout intentará realizar la compra ingresando sus datos personales y los datos de pagos solicitados. Una vez ingresada el usuario presiona el botón de “Finalizar Comprar” y aquí se disparará el caso de uso “Intento de Compra de un Producto”, este caso uso realizara las operaciones respectiva para realizar la comprar y el pago de la misma, que puede obtener dos resultados distintos, uno es disparar el caso de uso “Compra Exitosa” y la otra opción es “Error en intento de compra”. Luego de lo mencionado anteriormente se lista los casos de usos identificados:

- Búsqueda de un Producto
- Detalle de un Producto
- Página de Checkout
- Intento de compra de un Producto
- Comprar de forma exitosa
- Error en el intento de la compra

Detectado estos casos de usos o flujo de navegación que realiza un usuario con la aplicación, podemos monitorearlos con el fin de identificar que fue realizando un usuario, como por ejemplo realizar un número de búsquedas, intentar comprar y comprar exitosamente o con error. A partir de

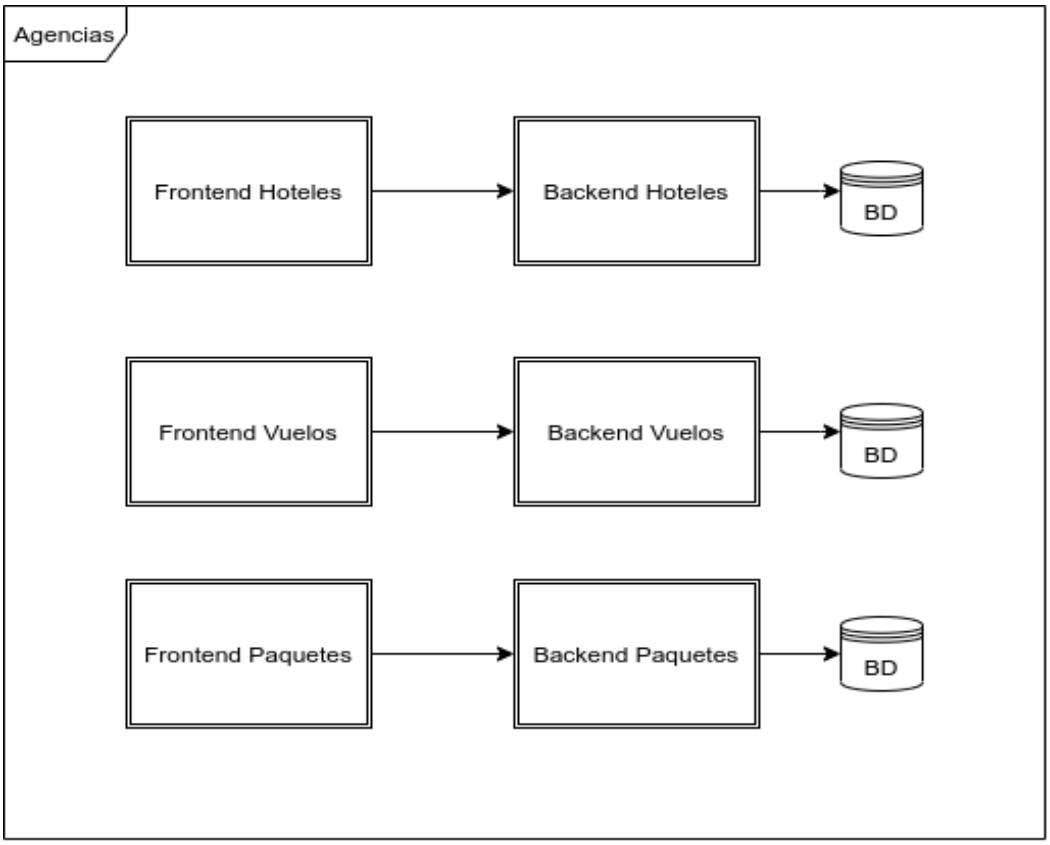


Figura 3.4: Arquitectura

este monitoreo se puede obtener métricas de cómo funciona la aplicación y proponer mejoras en distintos casos de uso. Por otro lado esto es también útil cuando existe algunos reclamos de parte del usuario, con esta herramienta se puede detectar cual pudo haber sido el punto de falla o qué acciones realizó el usuario, así detectar los errores de aplicación de una forma más ágil.

Ejemplo de interacción de usuario

A continuación se va a describir paso por paso como el usuario interactúa con la aplicación y los casos de usos que se ejecutan. En este caso el usuario realiza una búsqueda de un hotel en la ciudad de Salta (Argentina) con una fecha de ingreso de el 1 de enero del 2021 y salida el 6 de enero del 2021, en este caso se ejecutaría el caso de uso de “Búsqueda de un Producto”.

Luego de esta acción del usuario, el sitio el resultado de la búsqueda realizada, ofreciendo al usuario un listado de hoteles que cumplen con el criterio de búsqueda. Así de esta forma se puede ver un listado de los hoteles disponible, con su respectiva información, como una imagen principal, nombre y precio entre otras, mostrado en las figura 3.2 .

Una vez elegido el hotel, el usuario observará el detalle de este hotel de la página de detalle se

y presionado el botón “comprar” ejecutará el caso de uso “Página de Checkout” donde dirige al usuario a página de checkout (Figura 3.3), donde el usuario ya puede realizar la compra, este debe cargar información personal e información de pago para efectuar la compra. Un a vez hecho esto y presionar el botón “Finalizar Comprar”, se ejecuta el caso de uso “Intento de compra” que puede tener dos salidas posibles, una es la ejecución del caso de uso “Comprar de forma exitosa”, o en caso de que por algún motivo no se pueda efectuar la compra, como puede ser por error de medio de pago, se ejecuta el caso de uso “Error en el intento de la compra”.

En cada ejecución de los casos de usos que se mencionaron se disparará un evento MQTT con un tópico específico que será de utilidad para la aplicación de monitoreo SAPO que se explicara en los siguientes apartados.

3.2.3. Arquitectura de eventos

A partir de la arquitectura desarrollada anteriormente, y como se mencionó en el capitulo 1, surge la necesidad de poder realizar un monitoreo en tiempo real de cómo los usuarios finales interactúan con la aplicación web. De esta forma, detectar por ejemplo la cantidad de búsquedas que realiza un mismo usuario, detectar en primer momento los errores que surgen evitando que impacten en una mayor cantidad de usuarios, y a su vez poder medir el funcionamiento correcto de la aplicación luego de un nuevo release en ambientes productivos. Para este monitoreo se decidió utilizar una arquitectura orientada eventos, donde cada aplicación dispara un evento por cada caso de uso que ejecuta el usuario con información específica del mismo.

La decisión de utilizar una comunicación orientada a eventos y no una comunicación sincrónica de servicios web fue dada con el objetivo de reutilizar estos mismo eventos para otros fines. Un escenario de esto es realizar la funcionalidad de un envío de email al equipo de operaciones cada vez que se realiza una venta, de esta forma una aplicación estaría subscrita al tópico de “compra” y enviará un mail cada vez que recibe este evento. Este escenario donde más de una aplicación responda con distintas funcionalidades al mismo evento no podría ser posible a través de una comunicación sincrónica punto a punto entre aplicaciones, en todo caso podría ser posible pero se deberían enviar tantas comunicaciones como aplicaciones quieran conocer la ocurrencia de este evento. A continuación se muestra en la figura 3.5 esta arquitectura, donde se puede observar cómo interactúan los módulos de frontend y backend de las aplicaciones y la interacción con el bus de evento

3.3. Aplicacion SAPO

Sapo es la aplicación responsable de estar suscritos a estos eventos que disparan las aplicaciones de los distintos productos, guardarlos en una base de datos y a brindar al usuario interno de **despegar.com** una interfaz de usuario para poder consultar sobre estos eventos y monitorear los casos de usos explicados anteriormente.

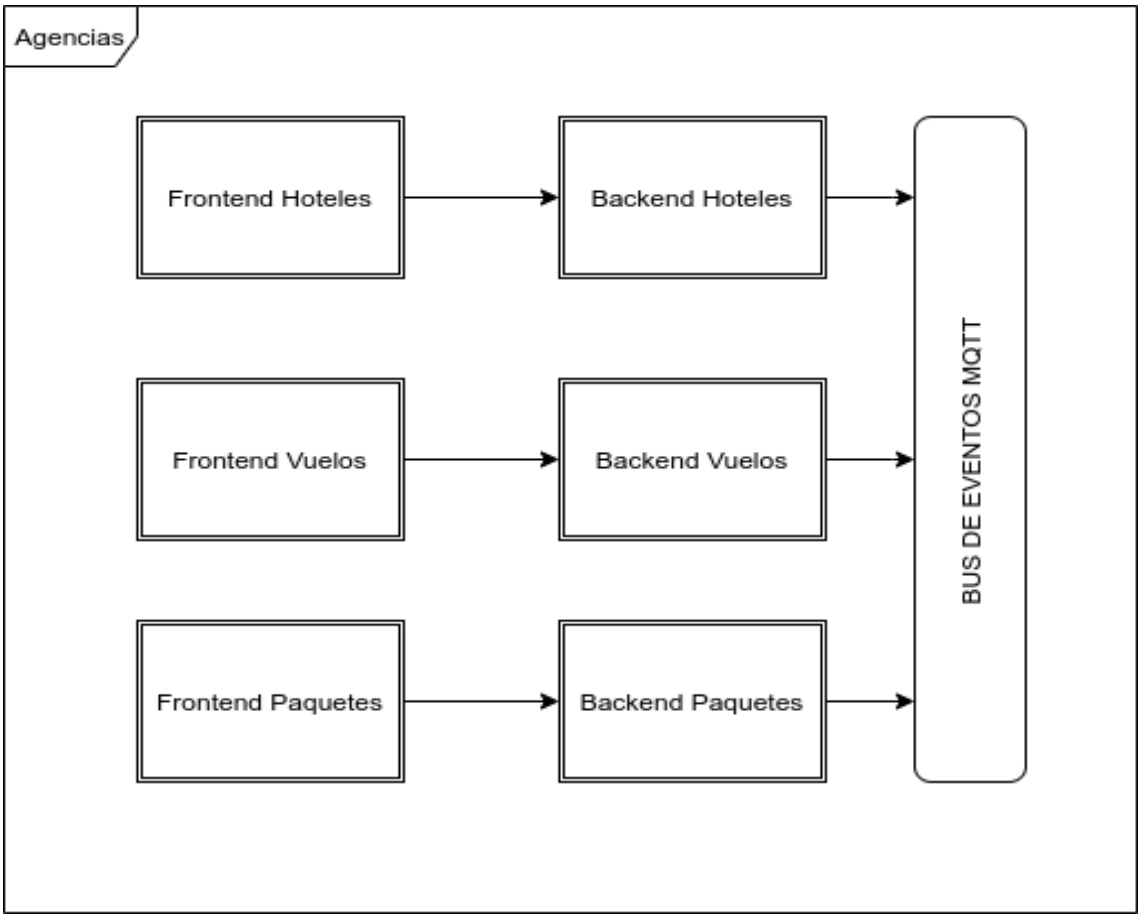


Figura 3.5: Arquitectura

3.3.1. Suscripción a eventos

A partir de la arquitectura y funcionalidad de envío de eventos descrita anteriormente se desarrolló esta aplicación responsable de consumir estos eventos. Esta aplicación se encarga de recibir estos eventos y realizar la persistencia de los mismos en una base de datos específica. La aplicación llamada SAPO es la responsable de escuchar los eventos enviados al broker de eventos MQTT con la implementación de "mosquitto"[21] y guardarlos en una base de datos. Para esto se definen los tópicos cumpliendo con el estándar de MQTT que se indica de la siguiente manera:

nombreDeCategoria1/nombreSubCategoria1/nombreSubCategoria2

Donde cada string o palabra define un nivel o categoría separada por el carácter especial "/". Utilizando este patrón se definieron las siguientes categorías para estas aplicaciones

NombreDeProducto/Ambiente/País/Agencia/UsuarioLogueado(true|false)/CasoDeUso

Por ejemplo un caso de uso de una búsqueda en un ambiente productivo sería el siguiente tópico:

HOTELS/PROD/AR/AG11593/true/search

Por otro lado MQTT provee wildcard para poder escuchar tópicos multiniveles de categoría y poder aplicar una lógica de negocios distinta al tópico escuchado. De este forma el tópico al que se realiza la suscripción desde la aplicación SAPO en un ambiente productivo se realiza de la siguiente forma:

HOTELS/PROD/+/+/+/+

Ejemplo en lenguaje nodejs de la aplicación llamada SAPO:

```
1  client.on('connect', function() {
2          client.subscribe('HOTELS/'+configEnv.ambiente+'/+/+/+/+   );
3  });
```

En este ejemplo se puede observar el uso de la variable configEnv.ambiente que representa una propiedad de la aplicación que varía de acuerdo al ambiente donde se ejecuta. Esto es útil en caso de necesitar un ambiente de desarrollo (DEV) o un ambiente de Beta (BETA).

3.3.2. Interfaz de Usuario

La aplicación Sapo ofrece al usuario una interfaz de usuario a través de una aplicación web, donde es el punto donde el usuario puede interactuar con los datos que se fueron obteniendo en los eventos de las aplicaciones previamente mencionadas. Como se muestra en la figura 3.6 a la aplicación solo pueden acceder usuarios autorizados por lo que el acceso a esta es una página de inicio de sesión para que solo puedan acceder a la información los usuario autorizados, estos usuarios autorizados tienen acceso a todas las funcionalidades que brinda la aplicación. Para la seguridad de la aplicación la información de los usuarios se guarda en una base de datos mongodb.

Una vez que el usuario ingresa a la aplicación se visualiza la página llamada “home” donde el usuario puede visualizar el listado de acciones que puede realizar sobre los datos obtenidos de los eventos de las aplicaciones.

Como se puede observar en la figura 3.7 existen varias opciones donde el usuario puede interactuar, a continuación se va desarrollar la opción de “Track and Trace” (búsqueda de información), que es donde el usuario puede consultar sobre los eventos previamente mencionados.

Track Trace - Busqueda:

A continuación se hace una descripción de las funcionalidades que provee la aplicación con respecto a la página de “Track and Trace”, aquí es donde el usuario puede acceder a los eventos que se fueron “disparando” en las aplicaciones, y puede realizar distintos tipos de búsquedas sobre estos a partir de distintos criterios de acuerdo a lo que necesite. Como se muestra en la Figura 3.8 se puede observar que en el lateral izquierdo de la paǵina se visualizan los atributos por los que el usuario puede realizar las búsquedas.

Los atributos de búsqueda y el significado de cada uno son los siguientes:

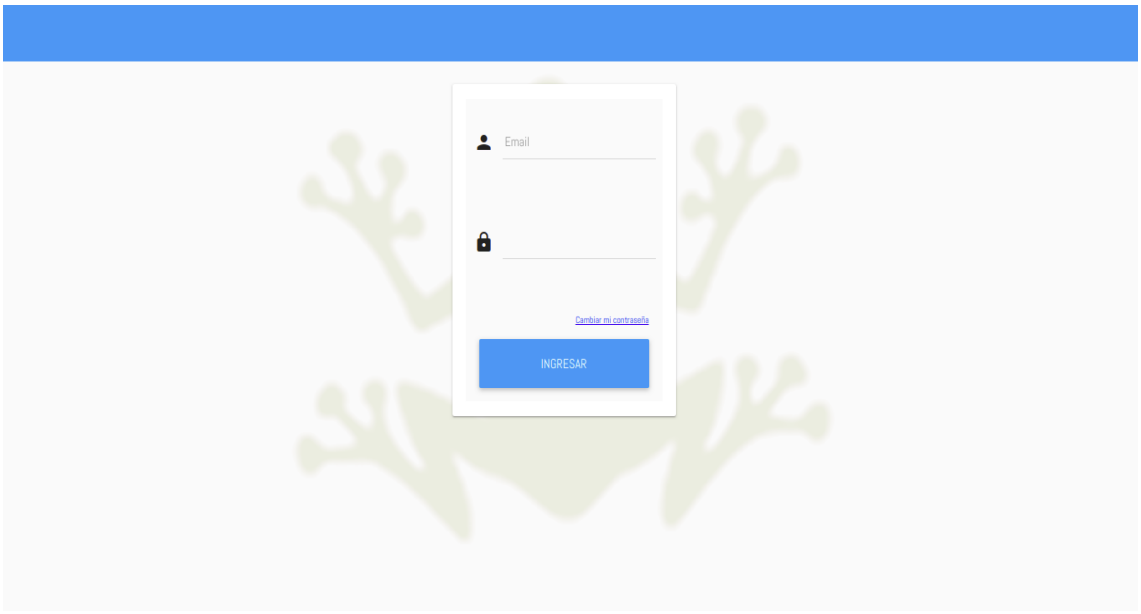


Figura 3.6: Home Sapo

- Fecha Desde: fecha inicial del rango donde ocurre el evento
- Fecha Hasta: fecha final del rango donde ocurre el evento
- Producto: producto en cuestión donde se realizó el evento (Hotel, Vuelo o Paquete)
- Pais: país donde se ejecutó el evento, las Agencias Afiliadas pueden corresponder a distintos paises de latinoamerica, para estos casos se utiliza una sigla correspondiente
  - Argentina: AR
  - Colombia: CO
  - Peru: PE
  - Chile: CL
  - Mexico: MX
  - Brasil: BR
- Nombre Fantasía: el nombre con el que se identifica la agencia afiliada para el usuario final, por ejemplo “PITUCON”, que es la mencionada anteriormente.
- ID Agencia: es un identificador interno en la base de datos, que identifica a la agencia afiliada.
- Reserva: numero de una reserva específica, este dato va a existir solamente para el caso de uso de intento de compra, en flujo de búsqueda o detalle no se registra este número ya que todavía no se realizó o intento hacer la compra.

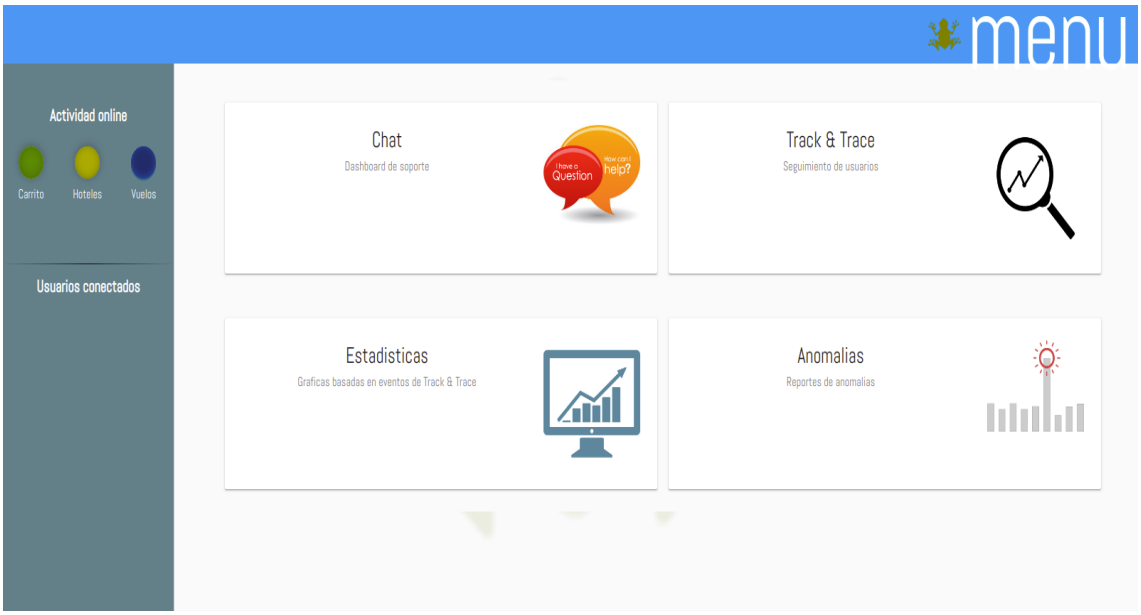


Figura 3.7: Dashboard

- Flujo: flujo o caso de Uso que realizó el usuario, ejemplo de estos son los mencionados anteriormente, búsqueda, detalle, checkout, intento de compra, compra exitosa y error en la compra.
- Email: en el caso de uso de "Intento de compra de un Producto" es el email que el usuario ingresó en la página de checkout.
- UOW: Este atributo hace referencia a un encabezado HTTP que se utiliza internamente dentro del los ambientes de desarrollo de **despegar.com**, es utilizado para identificar la trazabilidad de un request HTTP, un request sobre un servicio puede disparar otros N request, en cada uno de estos request se va a recibir el mismo encabezado HTTP bajo el nombre de UOW.
- Logueado : Determina si el usuario cuando realizó la acción estaba logueado o no al sistema, estas plataformas pueden permitir que exista un usuario registrado para realizar una compra.

Resultados de búsqueda:

A continuación se se muestra una imagen de lo que visualiza el usuario luego de realizar un búsqueda en la aplicación:

En la Figura 3.9 se puede observar que se visualiza cada evento que eventos ocurrido que condice con los parámetros de búsqueda, en el encabezado del evento se indica el "nombre de fantasía" de la agencia que disparó el evento y fecha en la que ocurrió el evento. Luego a partir de este encabezado se puede seleccionar un eventos específico mostrado en pantalla, y a partir de este se observa en detalle cual es la información que el evento incluye (Figura 3.13)

Figura 3.8: Busqueda

Como se observa en la figura 3.10 este detalle está dividido en dos partes, los titulados “Datos Técnicos” y los titulados “Descripción”.

Los **Datos Técnicos** son los datos en formato JSON, es el cuerpo del evento tal cual fue recibido o disparado sin procesar su contenido La **Descripción** es la información del evento pero de una forma más legible para un usuario sin conocimientos técnico. Esta herramienta es utilizada tanto por perfiles de IT como por perfiles administrativos, en este caso el evento ocurrido fue una búsqueda de un hotel, y la información que se muestra en la descripción es la siguiente:

- La Fecha de búsqueda del 22/05/2020 al 22/07/2020
- Se Hotel con 1 habitación/es:
    - 2 adultos
- El navegador que se usó para acceder fue chrome desde la IP 192.169.2.3

3.3.3. Arquitectura de Aplicación SAPO

En el siguiente apartado se realizará una descripción de cómo está compuesta la arquitectura tecnológica de la aplicación SAPO. En la figura 3.11 se observa los componentes que contiene esta arquitectura y la relación que existe entre ellas. Se puede observar la existencia del bus de eventos previamente mencionado, y como las aplicaciones de la plataforma de Agencias Afiliadas se comunican con este bus de eventos.

Como se puede observar en la figura 3.14 la aplicación llamada SAPO es la responsable de escuchar/obtener los eventos MQTT y realizar el guardado de estos en una base de datos. La

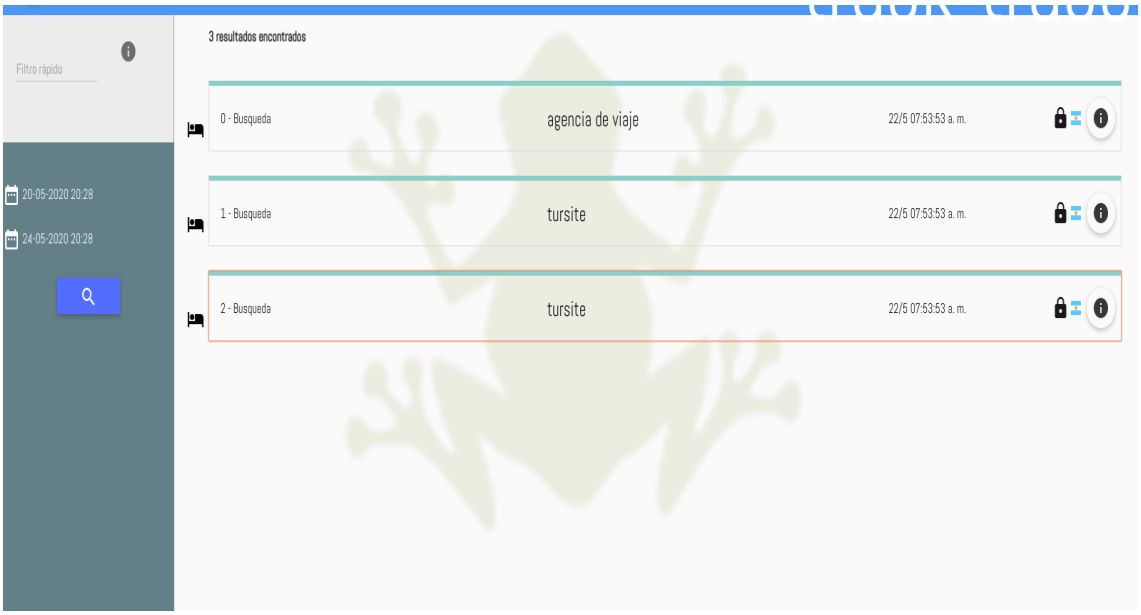


Figura 3.9: Resultado De Busqueda

base de datos que se utilizó es una no relacional, orientada a documentos llamada MongoDB. Sapo fue desarrollada en lenguaje JavaScript que se ejecuta en Node.js,un entorno de ejecución para JavaScript construido con el motor de JavaScript V8. Las aplicaciones de Paquetes, Vuelos y Hoteles están desarrolladas en lenguaje Java, como se mencionó anteriormente estas constan de un modelo de software llamado backend que dispone de una Api de servicios Rest HTTP, donde los principales servicios que responden a los casos de usos mencionados al ser invocados producen un evento MQTT.

Dentro de Sapo

La aplicación llamada Sapo fue desarrollada en dos módulos de software, un módulo llamado frontend , que es la que provee la aplicación web donde el usuario puede interactuar para monitorear los eventos o realizar consultas de eventos pasados, y un módulo de software backend que es el que provee una API rest para que la aplicación frontend se comunique, este módulo backend es el responsable de realizar las consultas a la base de datos donde se encuentran los eventos.

Frontend

El Frontend fue desarrollado , como se mencionó previamente, en tecnología JavaScript con el framework Angular 6. Este es una plataforma y framework utilizado para escribir aplicaciones web en HTML, cuenta con diferentes librerías; muchas son parte del Core y son necesarias para el funcionamiento correcto de nuestras aplicaciones, y otras pueden ser opcionales. Para crear aplicaciones con Angular, se generan templates con HTML y se controlan estos mismos templates con lógica creada en nuestros componentes, que serán exportados como clases. Así mismo,

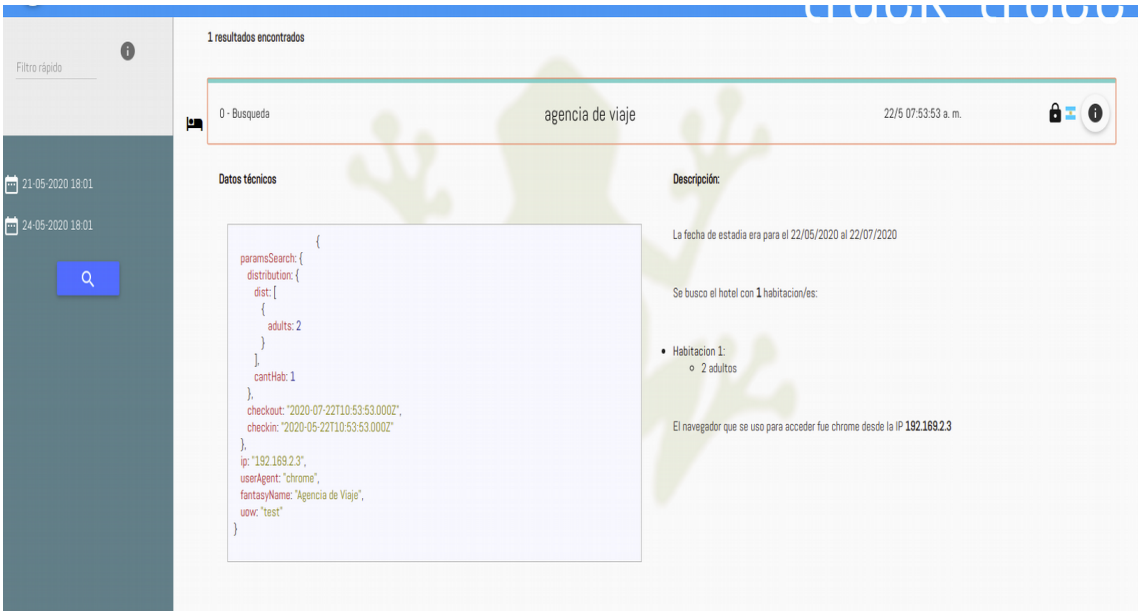


Figura 3.10: Resultado De Búsqueda

se agrega lógica en nuestros servicios para manejar los datos la aplicación tendrá y finalmente se "encapsulan" estos componentes y servicios en módulos. Después se inicia la aplicación mediante el bootstrapping donde Angular toma el control y muestra contenido en el explorador web, reaccionando a la interacción de los usuarios que utilicen la aplicación de acuerdo a las instrucciones que se dieron en lógica de los servicios.

Base de datos

La aplicación Sapo utiliza una base de datos no relacional (NOSQL) mongodb , estás base de datos guarda colecciones en formato JSON. Se utilizó este tipo de base de datos, ya que no fue necesario tener datos estrictamente relacionados como podía ser una base de datos relacional como mySql. Cada mensaje se guarda en la base de datos como un elemento único de una colección sin relacionarlo con otro, por lo que se hace un guardado del mensaje sin manipular su contenido. Al ser aplicaciones de un mismo equipo se acordó que el formato de los mensajes sea JSON sin hacer falta alguna validación técnica sobre el formato de estos mensajes.

Backend

El backend de la aplicación Sapo también está desarrollado en lenguaje JavaScript, ejecutándose en un servidor web nodejs. Este módulo es el encargado de exponer una API REST y tiene la funcionalidad de realizar como ejecución de métodos de esta API las consultas a la base de datos.

A continuación se desarrollará un ejemplo utilizando la liberaría mongoose de cómo el módulo backend realiza la conexión con nodejs a mongodb y así como se realizan las consultas a la misma base de datos. Conexion a mongo:

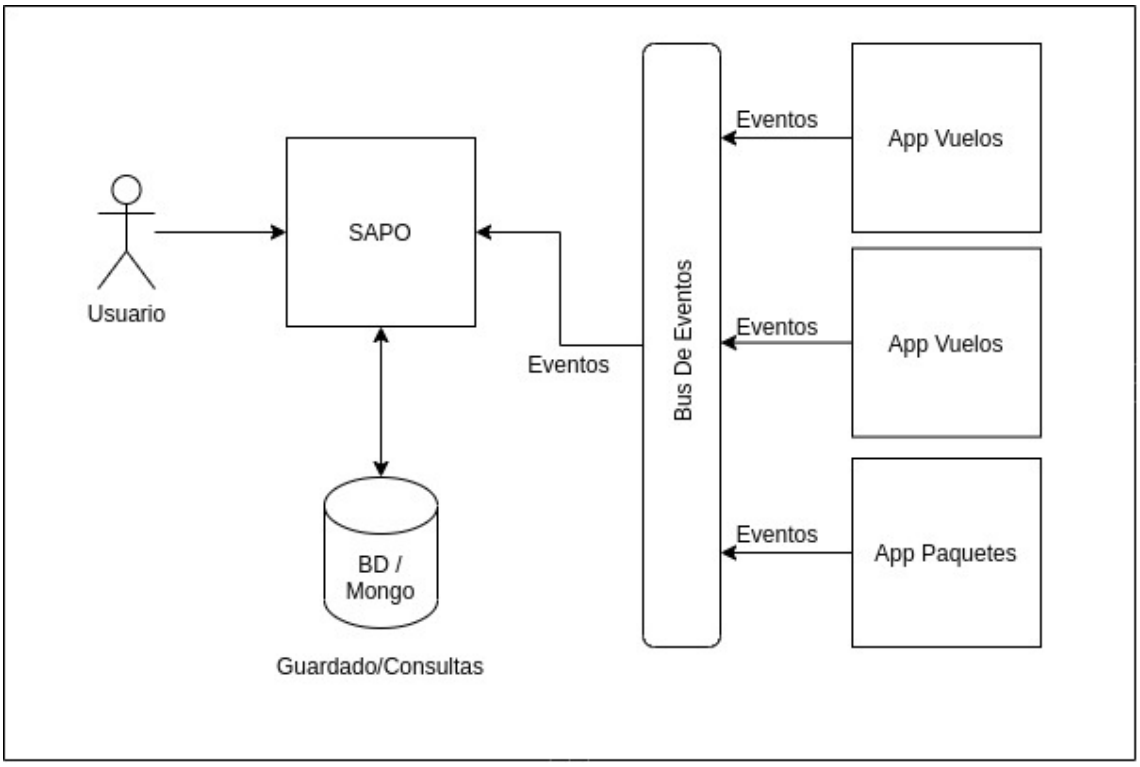


Figura 3.11: Arquitectura Sapo

```
await mongoose.connect('mongodb://localhost/my_database', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});
```

Luego de esta conexión, la librería mongoose para interactuar con las colecciones de la base de datos necesita que se defina un modelo:

```
const Schema = mongoose.Schema;
const ObjectId = Schema.ObjectId;
const BlogPost = new Schema({
  author: ObjectId,
  title: String,
  body: String,
  date: Date
});
```

A partir de esta definición se pueden realizar las operaciones que sean necesarias por la aplicación Guardado de una colección en la BD:

```
const instance = new MyModel();
instance.my.key = 'hello';
instance.save(function (err) {
  //
});
```

Búsqueda de colecciones:

```
MyModel.find({}, function (err, docs) {
  // docs.forEach
});
```

Búsqueda de una colección específica:

```
const instance = await MyModel.findOne({ ... });
console.log(instance.my.key);  // 'hello'
```

3.4. Bus de eventos - Broker MQTT Mosquitto

En esta sección se realizará una descripción de como es el funcionamiento e implementación del Bus de Eventos, que como se puede observar en la figura 3.11 es el intermediario en la comunicación entre las aplicaciones de la plataforma web y la aplicación SAPO, que como se dijo es la aplicación responsable de visualizar los eventos para el monitoreo de los casos de usos de los usuarios.

El bus mencionado es un broker de eventos MQTT OpenSource originalmente diseñado por IBM, ampliamente utilizado debido a su ligereza, lo que nos permite fácilmente emplearlo en gran número de ambientes e incluso si éstos son de pocos recursos. Muy útil para usar con Arduino y dispositivos IoT, como sensores, sistemas de monitorización y diagnóstico, etc. También lo podemos usar como servidor de notificaciones push. Tiene clientes para los principales lenguajes actuales, y el protocolo MQTT es un OASIS standard con lo que tiene soporte y continuidad de la comunidad. Para este bus de eventos particularmente se utilizó una implementación llamada mosquitto [21]

3.4.1. Envíos de mensaje MQTT

A continuación se mostrarán ejemplos del funcionamiento de mensajes MQTT con la implementación de mosquitto, estos mismo fueron realizados a través de una consola shell de linux, instanciando el servidor en la misma computadora.

Envío de mensaje

```
mosquitto_pub -h 192.168.0.12 -t HOTEL/BETA/AR/AG11593/true/ithentthank
    -m "{ \"uow\": \"test\" }" -i testclient -d
Client testclient sending CONNECT
Client testclient received CONNACK
Client testclient sending PUBLISH (d0, q0, r0, m1, 'HOTEL/BETA/AR/
   AG11593/true/ithentthank', ... (17 bytes))
Client testclient sending DISCONNECT
```

Ejemplo de suscripción

```
mosquitto_sub -h 192.168.0.16 -t hotels/# -C 3
```

Ejemplo desde una aplicacion Java

A continuación se muestra un ejemplo en código java, similar a la forma que se utilizó en las aplicaciones de Agencias Afiliadas:

```
String broker = "tcp://localhost:1883";
String topicName = "test/topic";
int qos = 1;

MqttClient mqttClient = new MqttClient(broker,String.valueOf(System.
   nanoTime()));

MqttConnectOptions connOpts = new MqttConnectOptions();

connOpts.setCleanSession(true);
connOpts.setKeepAliveInterval(1000);

MqttMessage message = new MqttMessage("Ed Sheeran".getBytes());

message.setQos(qos);
message.setRetained(true);

MqttTopic topic2 = mqttClient.getTopic(topicName);

mqttClient.connect(connOpts);
topic2.publish(message);
```

Ejemplo de suscripción

```
1
2  MqttClient client = new MqttClient("tcp://localhost:1883","clientid");
3  client.setCallback(this);
4  MqttConnectOptions mqOptions=new MqttConnectOptions();
5  mqOptions.setCleanSession(true);
6  client.connect(mqOptions);
7  client.subscribe("test/topic");
8
9  @Override
10 public void messageArrived(String topic, MqttMessage message) throws
      Exception {
11         System.out.println("message is : "+message);
12 }
```

Los ejemplos mencionados anteriormente son porciones de códigos obtenidas de las aplicaciones de la plataforma Agencias Afiliadas de **despegar.com** que producen los eventos, y así también ejemplos de la aplicación SAPO previamente comentada.

3.5. Evaluacion de performance de MQTT y HTTP

3.5.1. Introducción

Basado en el marco teórico, en esta sección se mostrará un conjunto de resultados de pruebas sobre el protocolo MQTT y el protocolo HTTP. Estas se realizaron con el fin de obtener medidas de latencia y utilización de bytes. Las pruebas son acordes a la necesidad de la plataforma descripta anteriormente, que como se mencionó recibe un tráfico promedio de 10000 request por minuto, y como consecuencia de cada request se envía un evento MQTT. Por esto se realizaron pruebas que simulan miles de request por minuto, llegando a probar 30000 request por minuto, y realizando mediciones de tiempo y recursos sobre la ejecución de estos miles de request. Estas pruebas se realizan con el objetivo de obtener medidas con respecto a la cantidad de eventos, esto se debe a que hoy en día la plataforma recibe 10000 request por minuto, pero a medida que crezca va a obtener mas eventos que estos, es muy probable que ante cualquier campaña de marketing los usuario accedan mas a la plataforma. Por lo que se tiene que saber como se comporta la performance de MQTT y HTTP a mayor cantidad de eventos.

3.5.2. Modelo de Referencia

Para las pruebas mencionada en la sección anterior, se utilizó la arquitectura de aplicaciones graficadas en la figura 3.12. Donde se puede observar que se desarrolló un aplicación Java en un servidor Jetty que realiza la simulación de los **N** eventos necesarios, tanto por HTTP como por

MQTT. Como se puede observar en el Apéndice A, para lograr la simulación de recepción de request de manera concurrente se implementó dentro de esa aplicación un pool de hilos con 150 hilos que representa cada maquina que envía request, así de esta forma se puede simular de manera concurrente los miles de request por minuto que recibe la plataforma. Para el caso de MQTT se utilizó un Broker de eventos Mosquitto[21] que recibe los eventos y los envía a una terminal linux suscrita al tópico en cuestión. Por otro lado para las pruebas HTTP se utilizó un servidor Jetty con una aplicación java que recibe los eventos representados a través de servicios POST.

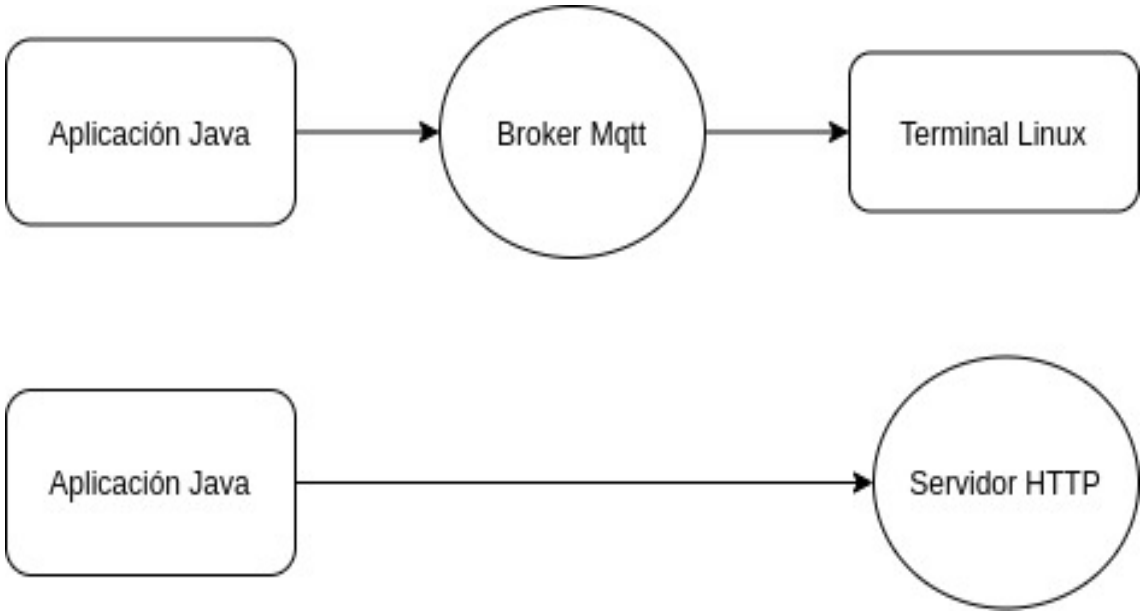


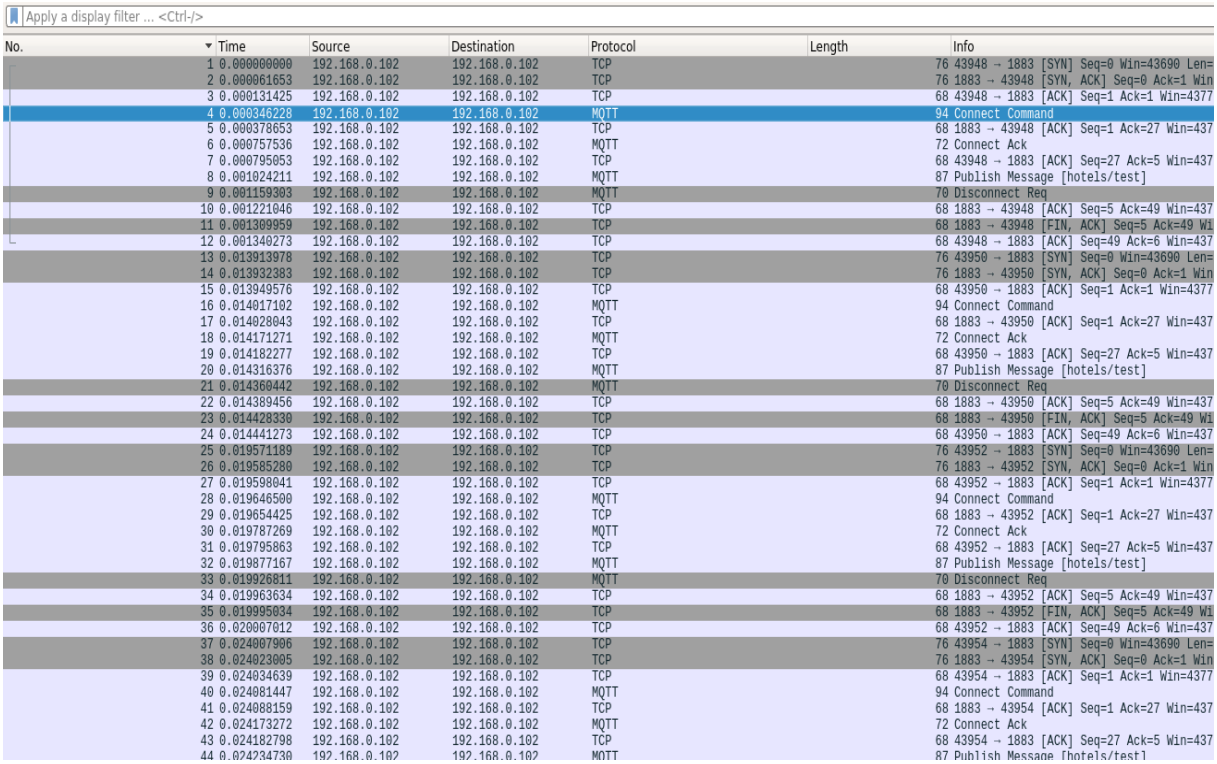
Figura 3.12: Arquitectura de Pruebas

A continuación se listan los dos servicios que expone esta aplicación Java para ejecutar los **N** eventos correspondientes, existe un servicio específico para lanzar eventos MQTT y otro para los eventos HTTP, cada uno esta parametrizado por la cantidad de eventos deseados.

```
1  POST http://localhost:8080/runhttp
2     Parametro: {Cantidad de Eventos}
3  POST http://localhost:8080/runmqtt
4     Parametro: {Cantidad de Eventos}
```

Para las pruebas en cuestión se utilizaron los siguientes valores de N: N = 5000, N = 10000, N = 15000, N = 20000, N = 25000, N = 30000. Se debe tener en cuenta que estas pruebas se realizaron en la mismo servidor por lo que se puede observar que los tiempos de retardo de red van a ser menores que lo mencionado en el capitulo del marco teórico, pero se obtuvo una relación similar a esas pruebas.

Para poder obtener la información de los tiempos consumidos y recursos de red de cada ejecución se utilizó una aplicación de escritorio llamada wireshark [22], esta es una aplicación de escritorio que realiza el monitoreo de paquetes de red que ocurren en el sistema operativo, como se


Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.0.102	192.168.0.102	TCP		76 43948 → 1883 [SYN] Seq=0 Win=43690 Len=
2	0.000061653	192.168.0.102	192.168.0.102	TCP		76 1883 → 43948 [SYN, ACK] Seq=0 Ack=1 Win
3	0.000131425	192.168.0.102	192.168.0.102	TCP		68 43948 → 1883 [ACK] Seq=1 Ack=1 Win=4377
4	0.000346228	192.168.0.102	192.168.0.102	MQTT		94 Connect Command
5	0.000378653	192.168.0.102	192.168.0.102	TCP		68 1883 → 43948 [ACK] Seq=1 Ack=27 Win=437
6	0.000757536	192.168.0.102	192.168.0.102	MQTT		72 Connect Ack
7	0.000795053	192.168.0.102	192.168.0.102	TCP		68 43948 → 1883 [ACK] Seq=27 Ack=5 Win=437
8	0.001024211	192.168.0.102	192.168.0.102	MQTT		87 Publish Message [hotels/test]
9	0.001159303	192.168.0.102	192.168.0.102	MQTT		70 Disconnect Req
10	0.001221046	192.168.0.102	192.168.0.102	TCP		68 1883 → 43948 [ACK] Seq=5 Ack=49 Win=437
11	0.001309959	192.168.0.102	192.168.0.102	TCP		68 1883 → 43948 [FIN, ACK] Seq=5 Ack=49 Wi
12	0.001340273	192.168.0.102	192.168.0.102	TCP		68 43948 → 1883 [ACK] Seq=49 Ack=6 Win=437
13	0.013913978	192.168.0.102	192.168.0.102	TCP		76 43950 → 1883 [SYN] Seq=0 Win=43690 Len=
14	0.013932383	192.168.0.102	192.168.0.102	TCP		76 1883 → 43950 [SYN, ACK] Seq=0 Ack=1 Win
15	0.013949576	192.168.0.102	192.168.0.102	TCP		68 43950 → 1883 [ACK] Seq=1 Ack=1 Win=4377
16	0.014017102	192.168.0.102	192.168.0.102	MQTT		94 Connect Command
17	0.014028043	192.168.0.102	192.168.0.102	TCP		68 1883 → 43950 [ACK] Seq=1 Ack=27 Win=437
18	0.014171271	192.168.0.102	192.168.0.102	MQTT		72 Connect Ack
19	0.014182277	192.168.0.102	192.168.0.102	TCP		68 43950 → 1883 [ACK] Seq=27 Ack=5 Win=437
20	0.014316376	192.168.0.102	192.168.0.102	MQTT		87 Publish Message [hotels/test]
21	0.014360442	192.168.0.102	192.168.0.102	MQTT		70 Disconnect Req
22	0.014389456	192.168.0.102	192.168.0.102	TCP		68 1883 → 43950 [ACK] Seq=5 Ack=49 Win=437
23	0.014428330	192.168.0.102	192.168.0.102	TCP		68 1883 → 43950 [FIN, ACK] Seq=5 Ack=49 Wi
24	0.014441273	192.168.0.102	192.168.0.102	TCP		68 43950 → 1883 [ACK] Seq=49 Ack=6 Win=437
25	0.019571189	192.168.0.102	192.168.0.102	TCP		76 43952 → 1883 [SYN] Seq=0 Win=43690 Len=
26	0.019585280	192.168.0.102	192.168.0.102	TCP		76 1883 → 43952 [SYN, ACK] Seq=0 Ack=1 Win
27	0.019598041	192.168.0.102	192.168.0.102	TCP		68 43952 → 1883 [ACK] Seq=1 Ack=1 Win=4377
28	0.019646500	192.168.0.102	192.168.0.102	MQTT		94 Connect Command
29	0.019654425	192.168.0.102	192.168.0.102	TCP		68 1883 → 43952 [ACK] Seq=1 Ack=27 Win=437
30	0.019787269	192.168.0.102	192.168.0.102	MQTT		72 Connect Ack
31	0.019795863	192.168.0.102	192.168.0.102	TCP		68 43952 → 1883 [ACK] Seq=27 Ack=5 Win=437
32	0.019877167	192.168.0.102	192.168.0.102	MQTT		87 Publish Message [hotels/test]
33	0.019926811	192.168.0.102	192.168.0.102	MQTT		70 Disconnect Req
34	0.019963634	192.168.0.102	192.168.0.102	TCP		68 1883 → 43952 [ACK] Seq=5 Ack=49 Win=437
35	0.019995034	192.168.0.102	192.168.0.102	TCP		68 1883 → 43952 [FIN, ACK] Seq=5 Ack=49 Wi
36	0.020007012	192.168.0.102	192.168.0.102	TCP		68 43952 → 1883 [ACK] Seq=49 Ack=6 Win=437
37	0.024007906	192.168.0.102	192.168.0.102	TCP		76 43954 → 1883 [SYN] Seq=0 Win=43690 Len=
38	0.024023005	192.168.0.102	192.168.0.102	TCP		76 1883 → 43954 [SYN, ACK] Seq=0 Ack=1 Win
39	0.024034639	192.168.0.102	192.168.0.102	TCP		68 43954 → 1883 [ACK] Seq=1 Ack=1 Win=4377
40	0.024081447	192.168.0.102	192.168.0.102	MQTT		94 Connect Command
41	0.024088159	192.168.0.102	192.168.0.102	TCP		68 1883 → 43954 [ACK] Seq=1 Ack=27 Win=437
42	0.024173272	192.168.0.102	192.168.0.102	MQTT		72 Connect Ack
43	0.024182798	192.168.0.102	192.168.0.102	TCP		68 43954 → 1883 [ACK] Seq=27 Ack=5 Win=437
44	0.024234730	192.168.0.102	192.168.0.102	MQTT		87 Publish Message [hotels/test]

Figura 3.13: Captura Wireshark

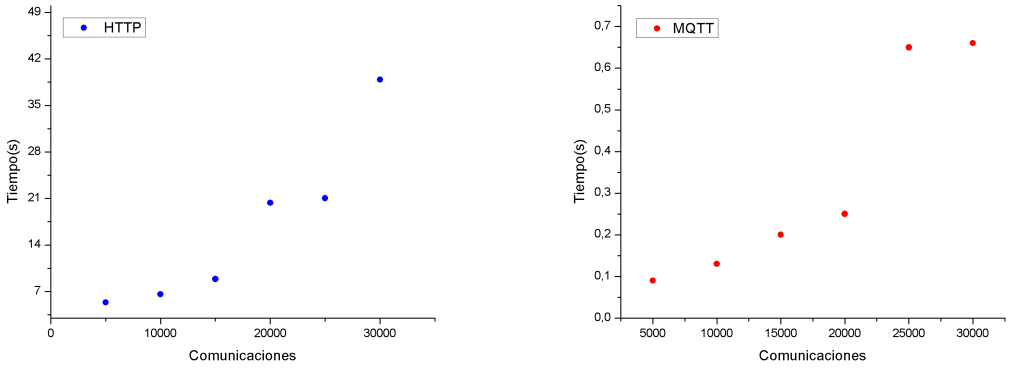
mencionó previamente, fue utilizada para medir la performance de los protocolos HTTP y MQTT. Con Wireshark por otro lado se puede analizar los protocolos de red y ver lo que está sucediendo en su red a un nivel microscópico [22], permite ver todo el tráfico que pasa a través de una red (usualmente una red Ethernet, aunque también es compatible con otras redes) estableciendo la configuración en modo “promiscuo” que ofrece.

En la figura 3.13 se puede observar una captura de pantalla de un ejemplo de monitoreo de paquetes de Wireshark, con el fin de demostrar de donde se obtuvo la información de medición. Wireshark, en su monitoreo dentro de toda la información ofrece la información del tiempo y de longitud de bytes utilizado en cada paquete, se tomaron estos dos valores para realizar las mediciones y las comparaciones en las pruebas.

3.5.3. Comparación de latencia de red

En esta sección se mostrarán los reportes de latencia generados, basados en los datos obtenidos en wireshark en la ejecución de cada prueba. En la figura 3.14 se puede observar el resultado de la prueba de latencia, en el 3.14(a) los tiempos del protocolo HTTP, y de la misma manera en la figura 3.14(b) los tiempos del protocolo MQTT. En ambas mediciones lo que se observa es el tiempo que se tarda en recibir los N eventos que se envían, con el fin de obtener que protocolo utiliza menor cantidad de tiempo a mayor cantidad de eventos. Ya que si demora demasiado en procesar los

eventos estos se puede perder o genera un cuello de botella es la aplicaciones receptoras o en el repositorio de datos. En la figura se puede ver que HTTP cuando procesa los 30000 eventos demora en procesarlos aproximadamente 42 segundos y MQTT demora solo 0.7 segundos. En la figura 3.15 se puede observar esta comparación, demostrando claramente que MQTT esta muy por debajo del promedio en el procesamiento de estos tiempos. Por lo que a mayor cantidad de eventos MQTT tiene mejor performance



(a) Tiempo HTTP (b) Tiempo MQTT (Qos 0)

Figura 3.14: Tiempo Http y Mqtt

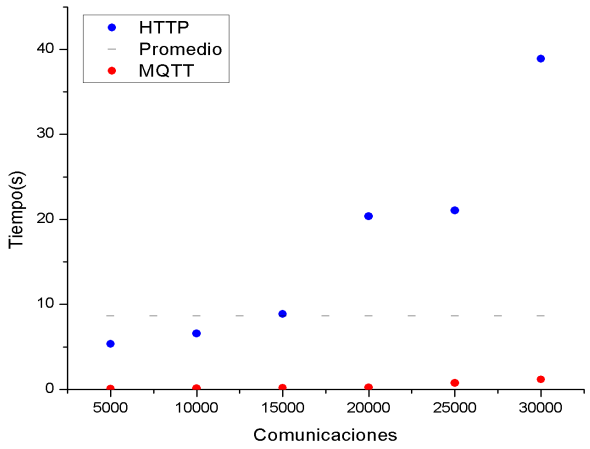
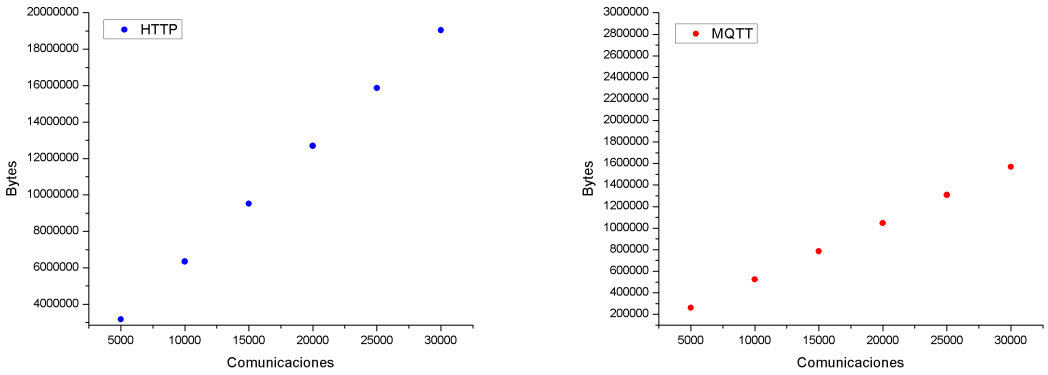


Figura 3.15: Comparación de Latencia

3.5.4. Comparación de recursos de red

En esta sección se mostrarán los reportes de cantidad de bytes utilizados, basados en los datos obtenidos en wireshark en la ejecución de cada prueba. En la figura 3.16 se puede observar el resultado de la prueba de cantidad de bytes, en la figura 3.16(a) el protocolo HTTP, y en 3.16(b) las pruebas del protocolo MQTT, como se puede observar ambas crecen de forma lineal de acuerdo a la cantidad de comunicaciones o eventos. Como se puede observar en la figura 3.17, donde se ve la comparación de los dos gráficos anteriores, la cantidad de recursos utilizados de red es mucho menor al promedio en el protocolo MQTT, por lo que a mayor cantidad de eventos mejor es la perfomance en MQTT.



(a) Bytes HTTP (b) Bytes MQTT (Qos 0)

Figura 3.16: Bytes

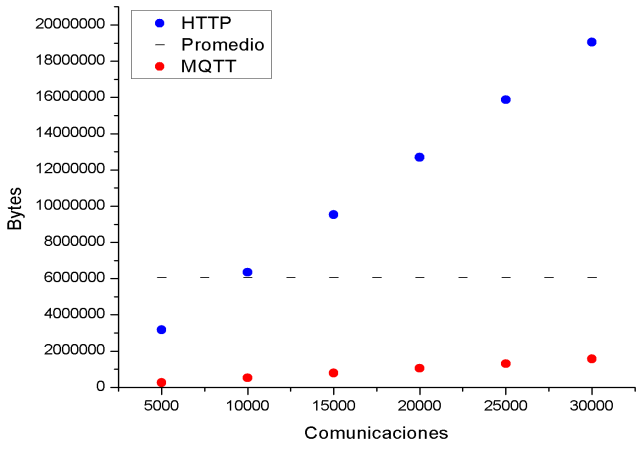


Figura 3.17: Comparación de utilización de Bytes

Capítulo 4

Conclusiones y Lineas de Investigación Futuras

4.1. Introducción

En este capítulo se presentará la conclusión final del presente trabajo de investigación, que se obtuvo basándose en el marco teórico y en las pruebas que se realizaron sobre los protocolos MQTT y HTTP, y por último se mencionarán distintas lineas de investigación que surgieron del caso de estudio.

4.2. Conclusión

Las arquitecturas de software orientadas a eventos solucionan problemas de escalabilidad en sistemas de gran tamaño que necesitan que su subsistemas estén intercomunicados entre si, estas arquitecturas evitan que todos los subsistemas estén comunicados entre todos, evitando así el difícil mantenimiento y escalabilidad de las mismas. Esto le logra a través de un bus de eventos donde se conectan el resto de las aplicaciones del sistema.

A partir de la problemática, mencionada a lo largo de la investigación, donde dentro de la unidad de negocio de Agencias Afiliadas de **despegar.com** se necesitó lograr monitorear todos los eventos ocurridos por cada caso de uso del usuario sobre la plataforma de ventas, el Proyect Leader del equipo de desarrollo optó por elegir una arquitectura de eventos a través del protocolo MQTT. Luego de la implementación de esto y en el transcurso de esta investigación se realizaron pruebas a traves del desarrollo de un algoritmo específico que ejecuta prueba de stress de cada protocolo, donde se obtuvieron resultado de tiempos y uso de recursos, para luego realizar una comparativa en términos de estas dos variables mencionadas.

Como resultado de las pruebas mencionadas anteriormente se obtuvo que para la cantidad de request con valores N = 5000 , N = 10000 , N = 15000 , N = 20000 , N = 25000 y N = 30000, la latencia de MQTT esta muy por debajo del promedio obtenido entre los protocolos MQTT y HTTP. Siendo este promedio de 9 segundos, MQTT tiene una latencia de 0.7 segundos cuando N

se aproxima a 30000, a diferencia de HTTP que es aproximadamente de 38 segundos.

Por otro lado para los mismo valores de N previamente mencionados, el protocolo MQTT esta muy por debajo del promedio en cuanto a la utilización de bytes, siendo el promedio entre ambos protocolos de 6000000 de bytes, MQTT utiliza aproximadamente 200000 cuando N se aproxima a 30000, a diferencia de HTTP que utiliza 19000000 aproximadamente.

Entonces ante la problemática mencionada de poder monitorear los eventos por cada caso de uso de la aplicación y lograr un almacén de datos con esta información, en el marco de este proyecto de investigación donde se realizaron las pruebas necesarias, se puede afirmar que el protocolo MQTT performa mejor que HTTP para lo N eventos cercanos a 30000 por minuto. Por lo que se puede decir que la utilización de MQTT fue una decisión acertada por el Proyect Leader.

Como conclusión final se logró afirmar que para el contexto de la unidad de negocios de Agencias Afiliadas de **despegar.com**, dada la arquitectura de un bus de eventos implementado con MQTT y dado las pruebas mencionadas y demostradas en las secciones anteriores, que el protocolo MQTT presenta un mejor perfomance que HTTP a mayor cantidad de procesamiento de eventos.

4.3. Lineas de Investigación Futuras

Durante la investigación e implementación del proyecto de bus de eventos bajo el protocolo MQTT surgió una problemática relacionado al almacenamiento de datos, mas específicamente ocurre que al tener un crecimiento lineal de cantidad de eventos, ocurre como consecuencia un futuro cuello de botella en la limitante del almacenamiento. Por otro lado también surge el interrogante de la distintas calidad de servicio que ofrece MQTT, en este caso se optó utilizar Qos0 solo porque dado del contexto se podía permitir un pequeño margen de error de perdidas de eventos.

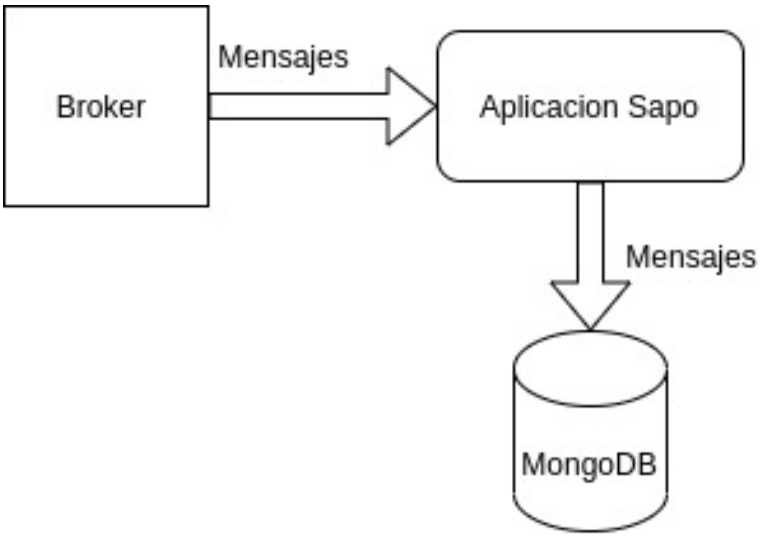


Figura 4.1: Almacenamiento de Eventos

4.3.1. Almacenamiento

Tanto en la sección de Objetivos como en el Caso de estudio, se describió que la recepción de todos los eventos MQTT son almacenados en una base de datos llamada Mongo. Este es un motor de base de datos no relacional, orientado a documentos de código abierto, en lugar de que la representación de los datos este conformada por tablas, como ocurre en una base de datos relaciones, en mongodb se representan los datos a través de documentos.

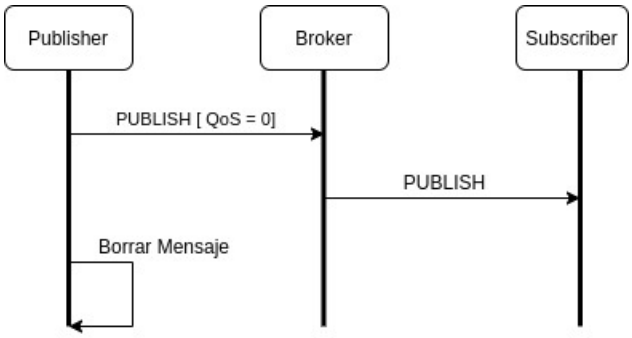
En la figura 4.1 se puede observar como es este flujo de datos de recepción de eventos y almacenamiento en el repositorio mongodb, uno de los inconvenientes que surgió con la solución propuesta es que la cantidad de eventos que se almacenan crecen de manera lineal de acuerdo a la cantidad de request recibidos, por lo que generó un cuello de botella en el motor de base de datos. Como consecuencia de esto se tuvo que configurar un tiempo de vida (TTL) a cada evento guardado para que de esta forma sea eliminado pasado este tiempo. Como futura linea de investigación se sugiere investigar sobre el motor de base de datos mongodb y sus distintas formas de configuración para que logren soportar mayor cantidad de eventos sin necesidad de ser eliminados y como consecuencia de esto evitar pérdida de información.

4.3.2. Calidad de servicio Qos1 y Qos2

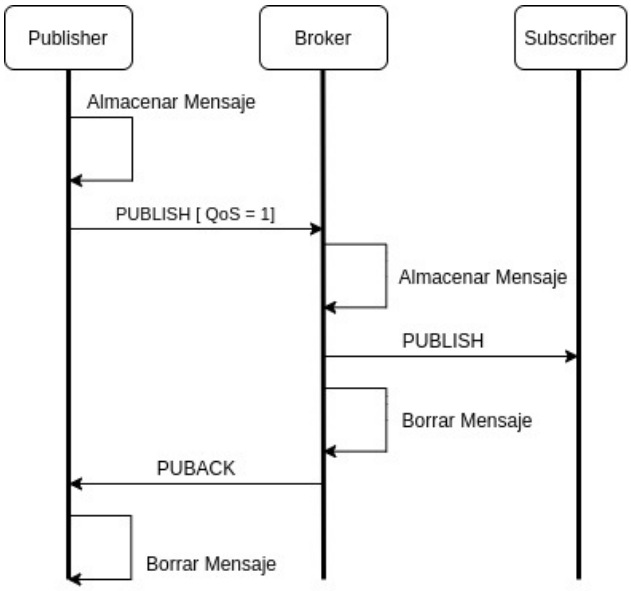
Como se mencionó en el capitulo de Marco Teórico, MQTT implementa tres niveles diferentes de calidad de servicio. En la figura 4.2 se puede visualizar cual es el flujo de comunicación entre las entidades Publisher, Broker y Subscriber para los distintos tipos de calidad de servicio, se puede observar para los niveles de calidad de Qos1 y Qos2 que el trafico de red se incrementa por cada mensaje, y por consecuencia se supone que también se incrementaría los tiempos de recepción de mensajes. La descripción de cada nivel de calidad de servicio es la siguiente[9]:

- QoS 0 – At most once, solo realiza una entrega “best effort”, sin garantías frente a fallos, esta es la opción que se utiliza por defecto.
- QoS 1 – At least once, el mensaje será entregado como mínimo una vez y puede proteger frente a pérdidas de conexión (aunque pueden llegar duplicados). En este caso se utilizan mensajes ACKs para confirmacion de entrega.
- QoS 2 – Exactly once, garantiza que cada mensaje se reciba por la parte receptora sin duplicidad.

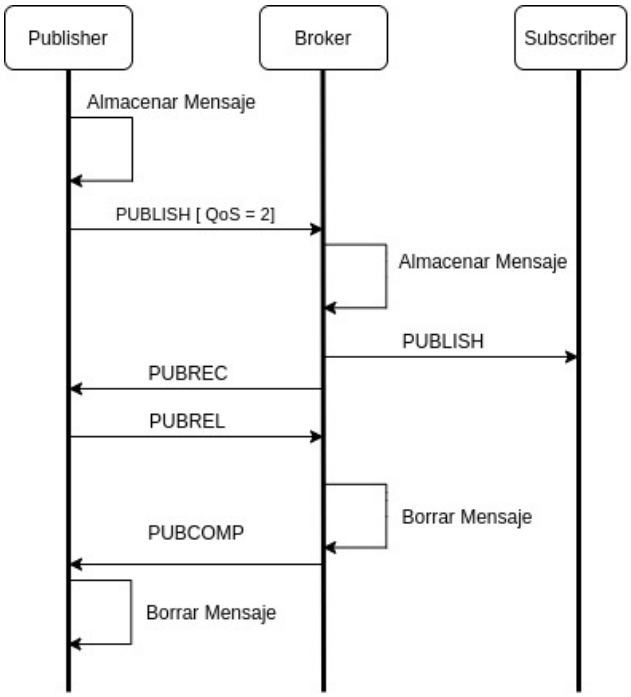
Ante estos niveles de calidad de servicio que ofrece MQTT se sugiere como siguiente linea de investigación realizar medidas y pruebas similares a las que se hicieron en la sección 3.5, pero con la calidad de servicio de MQTT Qos1 y Qos2.



(a) Qos0



(b) Qos1



(c) Qos2

Figura 4.2: Calidad de Servicio MQTT

Bibliografía

[1] Keith W. Ross James F. Kurose. *Redes y Comunicaciones - Segunda Edición*. Pagina 48. 2003.

[2] Alessandro Redondi Edoardo Longo y Pietro Manzoni. *MQTT ST a Spanning Tree Protocol for Distributed MQTT Brokers*. DISCA, Universitat Poltcnica de Valncia, Spain.

[3] Página oficial MQTT. Página oficial MQTT. 2021. URL: https://bit.ly/2Oejbn9.

[4] Oasis. Documentación oficial sobre MQTT. Página oficial MQTT. 2021. URL: https://bit.ly/1UVi0ur.

[5] Geeky Theory. *¿Qué es MQTT?* Página oficial MQTT. 2021. URL: https://bit.ly/2Xjzibc.

[6] Peter R. Egli. *MQTT*. 2021. URL: https://bit.ly/2lJglgZ.

[7] HiveMQ. *MQTT Essentials*. 2021. URL: https://bit.ly/2ZztWq7.

[8] Sheeld. *MQTT Protocol*. 2021. URL: https://bit.ly/2lJglgZ.

[9] Lars Michael Kristensen Alejandro Rodriguez y Adrian Rutle. *On Modelling and Validation of the MQTT IoT Protocol for M2M Communication*. Western Norway University of Applied Sciences, Bergen.

[10] Steve I.G. *Understanding the MQTT Protocol Packet Structure*. 2021. URL: https://bit.ly/2kduLpx.

[11] RFC 1945. *RFC 1945*. 2021. URL: https://www.rfc-editor.org/info/rfc1945.

[12] RFC 2616. *RFC 2616*. 2021. URL: https://www.rfc-editor.org/info/rfc2616.

[13] Keith W. Ross James F. Kurose. *Redes y Comunicaciones - Segunda Edición*. Pagina 88. 2003.

[14] Keith W. Ross James F. Kurose. *Redes y Comunicaciones - Segunda Edición*. Pagina 90. 2003.

[15] Keith W. Ross James F. Kurose. *Redes y Comunicaciones - Segunda Edición*. Pagina 95. 2003.

[16] Enrique Amodeo. *Principios de diseño de APIs REST*. Pagina 7. 2013.

[17] Enrique Amodeo. *Principios de diseño de APIs REST*. Pagina 9. 2013.

[18] Enrique Amodeo. *Principios de diseño de APIs REST*. Pagina 12. 2013.

[19] Enrique Amodeo. *Principios de diseño de APIs REST*. Pagina 14. 2013.

[20] Tetsuya Yokotani Yuya Sasaki. *Performance Evaluation of MQTT as a Communication Protocol for IoT and Prototyping*. 2021. URL: https://core.ac.uk/reader/201652239.

[21] Documentacion de mosquitto. eclipse. 2021. URL: https://mosquitto.org/.

[22] Aplicacion Wireshark. *Sitio Oficial de Aplicacion Wireshark*. 2021. URL: https://www.wireshark.org/.

Apéndice A

Aplicación de pruebas Java

A continuación se encuentra la aplicación Java que se desarrollo para realizar las pruebas de perfomance de los protocolos MQTT y HTTP

```
1  package com.core;
2
3  import ...
4
5  /**
6   *  Configuracion de los hilos que se ejecutan de forma concurrente
       para http y mqtt
7   *  150 hilos para mqtt
8   *  150 hilos para http
9   */
10 @EnableAsync
11 @Configuration
12 public class ConfigurationApplication {
13
14         /**
15          * Creacion del objeto ejecutor de hilos http
16          */
17         @Bean(name = "executorhttp")
18         public Executor taskExecutor() {
19                 ThreadPoolTaskExecutor executor = new
                      ThreadPoolTaskExecutor();
20
21                 executor.setCorePoolSize(100);
22                 executor.setMaxPoolSize(150);
23                 executor.setQueueCapacity(30000);
24                 executor.initialize();
25                 return executor;
26
```

```
27          }
28
29          /**
30           * Creacion del objeto ejecutor de hilos mqtt
31           */
32          @Bean(name = "executormqtt")
33          public Executor taskExecutormqtt() {
34                  ThreadPoolTaskExecutor executor = new
                       ThreadPoolTaskExecutor();
35                  executor.setCorePoolSize(100);
36                  executor.setMaxPoolSize(150);
37                  executor.setQueueCapacity(30000);
38                  executor.initialize();
39                  return executor;
40
41          }
42
43
44  }
```

```
1   package com.core.controller;
2
3   import ...
4
5   /**
6    * API que responde a los servicios :
7    *    /runhttp
8    *    /runmqtt
9    */
10  @Controller
11  public class MessageController {
12
13          @Autowired
14          private MqttService mqttService;
15
16          @Autowired
17          private HttpService httpService;
18
19
20          @CrossOrigin(origins = "*")
21          @RequestMapping(value = "/runhttp", method = RequestMethod.POST
                   )
22          @ResponseBody
23          public String runhttp(@RequestBody String cantidad) {
```

```
24
25                  this.httpService.run(Integer.valueOf(cantidad));
26                  return "OK";
27          }
28
29          @CrossOrigin(origins = "*")
30          @RequestMapping(value = "/runmqtt", method = RequestMethod.POST
               )
31          @ResponseBody
32          public String runmqtt(@RequestBody String cantidad) {
33
34                  this.mqttService.run(Integer.valueOf(cantidad));
35                  return "OK";
36          }
37  }
```

```
1   package com.core.services;
2
3   import ...
4
5   /**
6    * Ejecucion de los mensajes mqtt
7    */
8   @Service
9   public class MqttService {
10
11          @Autowired
12          MqttMessage messageService;
13
14          public void run(int cantidadDeRequest) {
15                  for ( int i = 0 ;  i < cantidadDeRequest; i++ ) {
16                          this.messageService.run();
17                  }
18          }
19  }
```

```
1   package com.core.services;
2
3   import ...
4
5   /**
6    * Ejecucion de los mensajes http
7    */
```

```
@Service
public class HttpService {

        @Autowired
        HttpMessage messageService;

        public void run(int cantidadDeRequest) {
                for ( int i = 0 ;  i < cantidadDeRequest; i++ ) {
                        this.messageService.run();
                }
        }

}
```

```
package com.core.services;

import ...

/**
 * Envio de mensajes http  utilizando el pool de hilos de 150
 */
@Service
public class HttpMessage {


        @Async("executorhttp")
        public void run()  {

          HttpPost post = new HttpPost("http://localhost:8081/message")
             ;

      List<NameValuePair> urlParameters = new ArrayList<>();
      urlParameters.add(new BasicNameValuePair("message", "hola"));
      try {
            post.setEntity(new UrlEncodedFormEntity(urlParameters));
          } catch (UnsupportedEncodingException e) {
           e.printStackTrace();
          }

      try (CloseableHttpClient httpClient = HttpClients.createDefault()
         ;
        CloseableHttpResponse response = httpClient.execute(post)) {

        System.out.println(EntityUtils.toString(response.getEntity()));
```

```
29        } catch (IOException e) {
30                e.printStackTrace();
31          }
32
33         }
34  }
```

```
1   package com.core.services;
2
3   import org.eclipse.paho.client.mqttv3.*;
4
5   /**
6    * Envio de mensajes mqtt  utilizando el pool de hilos de 150
7    */
8   @Service
9   public class MqttMessage {
10
11          private IMqttClient publisher;
12
13          @PostConstruct
14          public void init() {
15                  try {
16                   MqttConnectOptions options = new MqttConnectOptions();
17                   options.setConnectionTimeout(10);
18                   this.publisher = new MqttClient("tcp://192.168.0.20",
                        MqttAsyncClient.generateClientId());
19                   this.publisher.connect(options);
20                  }
21                  catch (MqttException e) {
22                   e.printStackTrace();
23                  }
24          }
25
26          @Async("executormqtt")
27          public void run() {
28                  try {
29                   String msgStr = "Hola";
30                   MqttMessage msg = new MqttMessage(msgStr.getBytes());
31                   msg.setQos(0);
32                   msg.setRetained(true);
33                   this.publisher.publish("hotels/thank", msg);
34                  } catch (MqttException e) {
35                    e.printStackTrace();
```

```
36                                  }
37                          }
38
39      }
```