Trabajo Práctico: Nivel de Aplicación y Nivel de Transporte del Modelo TCP/IP

Redes de Comunicaciones (TB067) - 2C2024 - FIUBA
Martin Klöckner (123456) - mklockner@fi.uba.ar
Matteo Aguilar Cafferata (123456) - meaguilar@fi.uba.ar

El protocolo MQTT

MQTT (Message Query Telemetry Transport) es un protocolo de red ligero y eficiente, implementado de extremo a extremo, con patrón de diseño publisher-subscriber (publicador-suscriptor) el cual a partir de la versión 3.1 es de código abierto.

Este protocolo está diseñado principalmente para dispositivos en locaciones remotas con recursos y/o ancho de banda limitado, tal como los sistemas embebidos, aunque puede ser utilizado en cualquier tipo de dispositivo.

El protocolo MQTT se ubica en el nivel de aplicación del modelo TCP/IP por lo que requieren de protocolos subyacentes; en particular protocolos ordenados, seguros y sin pérdida de datos, tal como los definidos por el modelo TCP/IP.

Agente (Broker)

El agente es quien actúa como intermediario entre los publicadores (publishers) y los suscriptores (subscribers), recibiendo mensajes de los publicadores y redireccionándolos a los respectivos suscriptores. Cada suscriptor se suscribe a uno o más temas (topics) del agente, de esta manera el agente lleva una cuenta de los suscriptores y les reenvía los mensajes al momento de que los publicadores envían un mensaje con ese tema.

Nivel de Servicio (QoS)

El Protocolo MQTT soporta 3 niveles de servicio (QoS ó Quality of Service): QoS 0, QoS 1 y QoS 2. Cuanto mayor es el nivel del servicio mayor será la demanda de recursos, pero mayor será la fiabilidad del servicio.

QoS 0

El nivel de servicio 0 es el más ligero en cuanto a uso de recursos, en este nivel los mensajes se envían y no se confirma la recepción, por lo que puede que ocurra pérdida de mensajes.

QoS 1

El nivel de servicio 1 es más fiable que el nivel de servicio 0, ya que los mensajes si se confirman y se reenvían en caso de ser necesario. El publicador envía un mensaje al agente y espera confirmación de recepción antes de proceder. Si el agente no responde en un intervalo de tiempo el publicador reenvía el mensaje. En este nivel de servicio, no ocurre pérdida de mensajes, pero puede ocurrir que se dupliquen mensajes

QoS 2

El nivel de servicio 2 es el más fiable de todos los niveles, pero como tal requiere de más recursos. Este nivel garantiza la recepción de mensajes una sola vez sin ser duplicados y sin pérdida de mensajes. En este nivel el publicador y el agente establecen un proceso de confirmación de recepción de 2 pasos, en la cual el agente almacena el mensaje hasta que haya recibido confirmación de recepción por parte del suscriptor.

Seguridad

El protocolo MQTT al priorizar el menor uso posible de recursos no cuenta con un sistema propio de encriptación de mensajes; siendo la única seguridad que este ofrece, el uso de un usuario y contraseña para autenticacion y autorizacion, este usuario y contraseña se envían como texto plano, sin ningún tipo de codificación.

En caso de necesitar algún tipo de seguridad, es recomendable encriptar los usuarios y contraseñas previo a enviarlas en el mensaje de MQTT; también utilizar canales seguros de conexión, por ejemplo mediante SSL/TLS.

Aplicaciones Comerciales

Existen múltiples aplicaciones tanto comerciales como sin fines de lucro del protocolo MQTT, puesto a que es ligero y eficiente energéticamente es ideal para ser usado en la comunicación con dispositivos en locaciones remotas con recursos escasos, como lo son sistemas embebidos o sensores de diversa índole, por dar un par de ejemplos; aunque no se limita a esos casos ya que también puede ser utilizado en centros de control con abundantes recursos que necesiten recibir esa información a tiempo real para la toma de decisiones cruciales y administración de activos.

A partir de la versión 3.1 el protocolo MQTT es de código abierto, por lo que cualquiera puede acceder a la especificación sin necesidad de tener que pagar una licencia, otro beneficio de que sea de código abierto es que permite modificar la implementación, esto resulta útil para extender el protocolo en caso de necesitarlo sin requerir de un proveedor que lo implemente.

Versiones

Las versiones de MQTT son controladas por la organización sin fines de lucro OASIS (Organization for the Advancement of Structured Information Standards). El protocolo es de código abierto a partir de la versión 3.1.

MQTT 5

Esta versión publicada en 2019 introdujo nuevas utilidades como códigos de motivo que especifican el tipo de error en la comunicación, suscripciones compartidas para equilibrar la carga entre los suscriptores sin saturar al broker, caducidad de mensajes para evitar el envío de información vieja o desactualizada y alias de temas para ahorrar espacio con los nombres de temas muy largos.

MQTT 3.1.1

Esta versión clarifica diversas partes de su antecesora además de terminar se asentar sus cambios e introducir otras mejoras como IDs de mayor extensión, clientes anónimos, flags de sesión activa y más específicos códigos de error, entre otras mejoras menores.

MQTT 3.1

Esta versión fue la primera lanzada como código abierto por IBM en 2010 e introdujo las funcionalidades de usuario y contraseña así como la codificación UTF-8.

MQTT-SN v1.2

Esta versión fue creada como una adaptación de la 5.0 para entornos de transporte por fuera del protocolo TCP/IP

Experimento con los Protocolos MQTT y TCP

Comunicación entre publicador y suscriptor

> Los dos scripts Python 3 proporcionados en este trabajo práctico constan de un publicador y un suscriptor. Adicionalmente, se brinda un archivo de texto. El publicador está diseñado para transmitir el contenido del archivo, en tanto el suscriptor está diseñado para recibirlo. La transmisión se realiza mediante la fragmentación del texto en mensajes de largo variable y la recepción por lo tanto recibe fragmentos y los ensambla para escribir el contenido completo en un archivo.

> El objetivo es verificar que el publicador pueda transmitir al suscriptor el archivo completo, siendo que cada integrante lo hará desde una PC diferente, por ejemplo, cada cual en su domicilio particular. Para ello, el suscriptor debe ejecutarse antes que el publicador.

Teniendo los scripts de python otorgados por la cátedra y todas las dependencias instaladas, se enviará el archivo `input.txt` desde el publicador al suscriptor utilizando el protocolo MQTT, para esto se requiere de un agente, en este caso se utiliza HiveMQ, el cual es gratis y publico; un tema, en este caso será `tp1/aguilar_klockner` y dos participantes o extremos, los cuales deben elegir el rol de suscriptor o publicador, en este caso se eligen los roles aleatoriamente.

Teniendo el agente, el tema y los roles ya definidos el suscriptor debe ejecutar el script `suscriber.py` para iniciar una conexión con el agente y suscribirse al tema, a continuación se ejecuta el script `subscriber.py` desde el extremo del suscriptor y se muestra la salida del comando, el cual queda esperando hasta que el agente envíe un mensaje

```
$ python3 subscriber.py
Subscribed: [ReasonCode(Suback, 'Granted QoS 2')] []
```

Se puede ver que la salida del programa sugiere un nivel de servicio QoS 2, el cual como mencionamos previamente es el más fiable de todos los niveles pero a costa de una mayor utilización de recursos, en este caso como se ejecutan en dos computadoras de escritorio con una buena conexión a internet, los recursos no son escasos, por lo que no es un problema utilizar ese nivel de servicio.

Luego de ejecutar el script y utilizando Wireshark para analizar el tráfico durante la ejecución del mismo (archivo `captura_klockner_suscriber.pcapng`), se puede ver que en el lado del suscriptor se inicia una conexión TCP con el agente (paquetes de número 5, 6 y 7 de la captura de Wireshark), posterior a la resolución de la dirección URL del agente por DNS (paquetes número 1, 2, 3 y 4), luego de iniciada la conexión TCP se envía un comando de conexión MQTT, Connect Ack (paquete 8) y posteriormente un mensaje de petición de suscripción Subscribe Request (paquete numero 10), con lo cual el agente responde con un mensaje Connect Ack y Subscribe Ack (paquetes 12 y 14 respectivamente).

Teniendo el suscriptor con una conexión ya iniciada, el publicador, ejecuta el script de nombre `publisher.py`, como se muestra a continuación

```
$ python3 publisher.py
Fragmento publicado 0 (size: 64)
Fragmento publicado 1 (size: 70)
Fragmento publicado 2 (size: 57)
Fragmento publicado 3 (size: 61)
Fragmento publicado 4 (size: 66)
Fragmento publicado 5 (size: 51)
Fragmento publicado 7 (size: 57)
Fragmento publicado 8 (size: 59)
Fragmento publicado 9 (size: 68)
Fragmento publicado 10 (size: 54)
```

Se observa en la salida del comando que se publicaron 10 fragmentos del archivo `input.txt` de tamaño variable (en algunos casos puede que sean más fragmentos ya que el tamaño de cada fragmento se elige aleatoriamente).

Analizando la captura de Wireshark realizada durante la ejecución del script desde el lado del publicador (archivo `captura_aguilar_publisher.pcapng`) podemos ver que de manera análoga al lado del suscriptor, se resuelve la URL de agente por DNS (paquetes 1 a 4), y luego se inicia una conexión TCP con el mismo (paquetes 5, 6 y 7). Luego de iniciada la conexión TCP se envía un mensaje de comando MQTT Connect Command (paquete 8), sin tener en cuenta los paquetes del protocolo de transporte TCP, recibe posteriormente confirmación de conexión Connect Ack (paquete número 12), establecida la conexión con el agente, envía 11 mensajes MQTT de comando Publish Message (paquetes de número 10, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52), cada paquete con su respectivo mensaje de confirmación, debido al nivel de servicio QoS 2 (paquetes de número 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54). Finalmente se envía un mensaje MQTT de tipo Disconnect Req (paquete número 56) el cual finaliza la conexión con el agente

Volviendo al lado del suscriptor se recibe la totalidad de los fragmentos del archivo, y se guardan en un nuevo archivo de nombre `output.txt`, en pantalla se muestra lo

siguiente, recordemos que luego de ejecutar el script este se quedó esperando hasta que el agente envía mensajes con el tema al cual el suscriptor está suscripto

```
$ python3 subscriber.py
Subscribed: [ReasonCode(Suback, 'Granted QoS 2')] []
Fragmento recibido 0 (size: 64)
Fragmento recibido 1 (size: 70)
Fragmento recibido 2 (size: 57)
Fragmento recibido 3 (size: 61)
Fragmento recibido 4 (size: 66)
Fragmento recibido 5 (size: 51)
Fragmento recibido 7 (size: 57)
Fragmento recibido 8 (size: 59)
Fragmento recibido 9 (size: 68)
Fragmento recibido 10 (size: 54)
File reassembled as output.txt
```

Volviendo a la captura de Wireshark realizada desde el lado del suscriptor (archivo `captura_klockner_subscriber.pcapng`) luego de recibir la confirmacion de suscripcion del agente, y al mismo tiempo que el publicador comienza la ejecución del script, el suscriptor, comienza a recibir mensajes de comando MQTT de tipo Publish Message por parte del agente (paquetes[1] de número 16, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83), seguido de mensajes de comando de tipo Publish Release (paquetes de número 20, 26, 32, 38, 44, 50, 56, 62, 68, 74, 80, 86), cada uno con su respectiva confirmación del suscriptor al agente, Publish Receive y Publish Complete para los mensajes de tipo Publish Message y Publish Release respectivamente (debido al nivel de servicio QoS 2). Al finalizar la recepción de la totalidad del mensaje el script `suscriber.py` termina la conexión TCP, enviando un segmento con el flag FIN activo (paquete número 87). Cabe destacar que es el script quien termina la conexión, ya que podría ser que el suscriptor siga conectado al agente de manera de seguir recibiendo mensajes que sean enviados con el respectivo tema

Fragmentación

> Sobre la aplicación, ¿de qué modo consigue implementar la fragmentación?

La aplicación implementa la fragmentación de manera manual, a través del script `publisher.py`, donde usa un bucle para dividir el contenido en partes según un tamaño aleatorio dentro de un rango arbitrario definido por el usuario para luego enviar cada parte en un mensaje MQTT diferente.

[1] Aquí el número de paquetes recibido por el suscriptor es mayor a los enviados por el publicador, esto se debe a que las capturas no corresponden al mismo experimento, pero la secuencia y procedimiento de envío de paquetes es la misma, solo difieren en tamaño y que se envía un fragmento más

Similitudes con fragmentación de TCP

> En comparación con la fragmentación que implementa el protocolo TCP, ¿hay funcionalidades o características de la fragmentación en estos scripts que sean similares a la segmentación TCP? ¿Cuáles?

Es similar en el sentido de que en ambas la partición y el envío de datos se realizan de forma simultánea, cada segmento fragmentado se envía antes de tener listo el siguiente. Luego al igual que en la fragmentación de TCP son ensamblados cuando se recibe la totalidad de los fragmentos.

> En comparación con la segmentación que implementa el protocolo TCP, ¿hay funcionalidades o características de la segmentación TCP que en estos scripts están ausentes? ¿Cuáles?

Es distinto ya que en TCP el emisor le pregunta al receptor cuantos datos puede recibir por segmento y la cantidad de segmentos simultáneos es dinámica, mientras que aquí el mínimo y máximo de cada fragmento está fijado por valores completamente arbitrarios.

Modificando el script `publisher.py` de manera tal que los segmentos sean mayores, por ejemplo 100 bytes, se puede ver que se logra enviar menos mensajes MQTT; a continuación se muestra la salida de la ejecución del script `subscriber.py` del lado del suscriptor cuando el archivo a enviar se divide en fragmentos de 100 bytes

```
$ python3 suscriber.py
Subscribed: [ReasonCode(Suback, 'Granted QoS 2')] []
Fragmento recibido 0 (size: 84)
Fragmento recibido 1 (size: 76)
Fragmento recibido 2 (size: 84)
Fragmento recibido 3 (size: 77)
Fragmento recibido 4 (size: 92)
Fragmento recibido 5 (size: 80)
Fragmento recibido 6 (size: 97)
Fragmento recibido 7 (size: 93)
File reassembled as output.txt
```

Probando efectivamente que la fragmentación del archivo a enviar es completamente arbitraria, a diferencia de la fragmentación que realiza TCP.

Sobre las Sesiones

> Acerca de las sesiones del publicador y del suscriptor, ¿cuáles son los extremos de las sesiones? ¿quién toma el rol de cliente y quién de servidor?

Los extremos son el publicador y el suscriptor los cuales actúan como clientes solicitando el servicio de enviar o recibir mensajes y el servidor sería el agente que permanece activo

esperando a que le soliciten una tarea, tal como conectarse, desconectarse, suscribirse o publicar un mensaje en un tema.

> Sobre la extensión de las sesiones, ¿son persistentes, cómo se sostienen cuando no hay tráfico? ¿no son persistentes? ¿cómo se cierran (en qué momento y quién lo inicia)?

Las sesiones no son persistentes ya que existen únicamente para realizar la tarea. La sesión del publicador se abre cuando este solicita enviar información y se cierra tras terminar este trabajo. La sesión del suscriptor dura desde que solicita suscribirse hasta que recibe toda la información que el agente posea sobre su tema y después se cierra, en caso de no recibir nada queda en esperar recibiendo pings del agente para mantener la sesión activa hasta que se agote el tiempo establecido para mantenerla viva.

> En detalle sobre las sesiones: a. ¿Cuál es el número de secuencia del primer segmento TCP de petición de conexión? b. ¿Cuáles son las opciones implementadas, si las hay? c. ¿Cuántos bytes tiene el buffer de recepción según se informa al inicio? d. ¿A qué hora se envió el primer segmento (el que contiene datos de la aplicación)? ¿A qué hora se recibió el ACK de este primer segmento que contiene datos? ¿Cuál es su RTT? e. ¿Cuál es la longitud (encabezado más carga útil) de cada uno de los primeros cuatro segmentos TCP que transportan datos?

Considerando el lado del suscriptor, debido a que este es quien primero inicia la conexión con el agente, y analizando la captura de Wireshark realizada durante la ejecucion del script `suscriber.py` (archivo `captura_klockner_suscriber.pcapng`) el número de secuencia del primer segmento TCP (número de paquete 6) de petición de conexión es `0` en términos relativos (relativos al primer segmento TCP, en este caso es el mismo paquete) y `3241009901` en términos absolutos.

Las opciones activas en el primer segmento que envía el suscriptor son Maximum Segment Size, el cual es `1460` bytes; SACK y Escalado de Ventana.

El suscriptor en este primer paquete (número de paquete 6), anuncia una ventana de recepción de `64240` bytes.

El primer paquete TCP que envía datos de la aplicación, se origina en el publicador y va hacia el agente, en la respectiva captura (archivo `captura_aguilar_publisher.pcapng`) se puede ver que este primer segmento (número de paquete 10) que envía datos tiene la hora en términos de tiempo Unix de `1727972104.779687350` mientras que el ACK de este primer segmento (número de paquete 11) tiene una hora en tiempo Unix de `1727972104.783581297` haciendo la diferencia resulta en `0.0038938522338867188` lo que es aproximadamente `3.89` ms

Luego de que el publicador resuelva la URL del agente (paquetes 1 a 4 del archivo `captura_aguilar_publisher.pcapng`) este inicia una sesión TCP mediante un acuerdo de tres pasos (paquetes de número 5, 6 y 7) en este acuerdo de tres pasos no se intercambian datos de la aplicación sino que se sincronizan los números de secuencia y de recibo de las sesiones TCP del agente y publicador, el siguiente paquete que envía el publicador al agente envía un mensaje de comando MQTT de peticion de conexion (Connect Command, paquete numero 8) el cual tiene una carga útil de `14` bytes, sumado a los `20` bytes que ocupa el encabezado TCP resultan en un paquete de `34` bytes, los siguientes paquetes que envían datos son también comandos MQTT Publish Message, Connect Ack y Publish Received (números de paquete 10, 12 y 14), los cuales tienen un

largo (teniendo en cuenta encabezado TCP y carga útil utilizada por el mensaje MQTT) de `97` bytes, `4` bytes y `4` bytes respectivamente.

Mejoras al Protocolo MQTT

> HTTP indica la cantidad de datos a transmitir mediante un encabezado. Esta adecuación de MQTT para transmitir fragmentos no comunica la cantidad de datos que va a transmitir. Si algún paquete se perdiera, el suscriptor no tendría forma de detectar la falta. Modificar el código para simular la pérdida de un paquete intermedio y verificar en el suscriptor que el texto quede truncado. Luego modificar ambos scripts para que se comunique el largo de los datos a transmitir y que el suscriptor pueda validar la cantidad de datos recibidos versus los esperados.

Comenzamos simulando la pérdida de un paquete, para eso modificamos el codigo del archivo `publisher.py` de manera tal que si el fragmento del archivo de entrada tiene un número específico no lo envié al agente, el resultado es el script cuyo nombre es `publisher-missing-package.py` ejecutando este script se obtiene la siguiente salida

```
$ python3 publisher-data-lost.py
Fragmento publicado 0 (size: 59)
Fragmento publicado 1 (size: 56)
Fragmento publicado 2 (size: 69)
Fragmento publicado 3 (size: 51)
Fragmento publicado 4 (size: 60)
Fragmento publicado 5 (size: 56)
Fragmento publicado 7 (size: 55)
Fragmento publicado 8 (size: 58)
Fragmento publicado 9 (size: 60)
Fragmento publicado 10 (size: 54)
Fragmento publicado 11 (size: 58)
```

Puede verse en la salida del mismo que el paquete número 6 nunca se publica, simulando así la pérdida del mismo. Verificando el archivo recibido puede verse que efectivamente falta una parte

```
$ cat output.txt
"Nos los representantes del pueblo de la Nación Argentina, reunidos en
Congreso General Constituyente por voluntad y elección de las
provincias que la componen, en cumplimiento de pactos preexistentes,
con el objeto de constituir la unión nacional, afianzar la justicia,
consolidar la paz interior, proveer a la defensa común, promover el
bienestar general y asegurdad, y para todos los hombres del mundo que
quieran habitar en el suelo argentino: invocando la protección de Dios,
fuente de toda razón y justicia: ordenamos, decretamos y establecemos
esta Constitución para la Nación Argentina."
```

Para identificar y corregir la pérdida de un paquete en la implementación realizada en `publisher-data-lost.py` y `subscriber-data-lost.py` el publicador mide el tamaño del archivo a enviar y lo guarda en la variable `lenght` para luego envíar este dato en el primer fragmento de datos. Mientras tanto el suscriptor recibe este fragmento inicial y no lo incluye en la reconstrucción sino que guarda su información como `control_size` y cuenta el tamaño de cada uno de los demás fragmentos recibidos, sumando y guardandolos como `total_size`. Al final compara los valores y si `total_size` es menor a `control_size` confirma que se perdió un paquete; de lo contrario confirma la recepción correcta.

Además en el archivo `publisher-data-lost.py` se simula la pérdida de un paquete de manera de identificar en el suscriptor si funciona la identificación de pérdida de datos.

Ejecutando el script `publisher-data-lost.py` del lado del publisher se obtiene lo siguiente:

```
$ python3 publisher-data-lost.py
Fragmento publicado 0 (size: 4)
Fragmento publicado 1 (size: 50)
Fragmento publicado 2 (size: 70)
Fragmento publicado 3 (size: 64)
Fragmento publicado 4 (size: 63)
Fragmento publicado 5 (size: 67)
Fragmento publicado 7 (size: 65)
Fragmento publicado 8 (size: 60)
Fragmento publicado 9 (size: 64)
Fragmento publicado 10 (size: 59
Fragmento publicado 11 (size: 60
```

Mientras que ejecutando el script del lado del suscriptor se obtiene la siguiente salida:

```
$ python3 subscriber-data-lost.py
Subscribed: [ReasonCode(Suback, 'Granted QoS 2')] []
Fragmento recibido 0 (size: 4)
Fragmento recibido 1 (size: 50)
Fragmento recibido 2 (size: 70)
Fragmento recibido 3 (size: 64)
Fragmento recibido 4 (size: 63)
Fragmento recibido 5 (size: 67)
Fragmento recibido 7 (size: 65)
Fragmento recibido 8 (size: 60)
Fragmento recibido 9 (size: 64)
Fragmento recibido 10 (size: 59)
Fragmento recibido 11 (size: 60)
File reassembled as output.txt
Error en la transmisión: Perdida de datos detectada
```

Se puede ver que efectivamente el suscriptor detecta la pérdida de datos ya que no concuerda el largo anunciado a enviar con el que en realidad recibe.

> De las funcionalidades o características vinculadas a segmentación, presentes en TCP pero no cubiertas por estos scripts, implementar una versión mejorada, que incorpore al menos una mejora o una funcionalidad. Se puede usar inteligencia artificial y por qué no, inteligencia natural.

Realizamos una implementación en `publisher-secure.py` y `subscriber-secure.py` que busca agregar un nivel de garantía de entrega de datos por retransmisión como en TCP. En esta usamos un valor pseudo-aleatorio para provocar una probabilidad del 10% de que un fragmento no se envíe; el publicador detecta si no son enviados todos los datos gracias a la comparación entre `control_size` y `total_size` mencionada en el inciso anterior, detectada esta pérdida reinicia la secuencia de envio de informacion. Tanto en caso de reinicio como de salida exitosa le envía un paquete adicional al suscriptor para indicarle la situación.

El suscriptor recibe los datos y dependiendo del paquete adicional del publicador indicando el resultado de la transmisión procede a, en caso de error eliminar los elementos recibidos y continuar recibiendo, o en caso de éxito procede a iniciar la reconstrucción de la salida sin incluir el fragmento indicador de situación.

Debido a un error desconocido de pérdida de paquetes del suscriptor tras 3 iteraciones (15 fragmentos) decidimos aumentar el tamaño de los fragmentos para así reducir su cantidad y evitar errores provocados por el agente, el cual no garantiza el servicio en su totalidad.

Ejecutando el script `subscriber-secure.py` del lado del suscriptor se obtiene la siguiente salida

```
$ python3 subscriber-secure.py
Fragmento recibido 0 (size: 108)
Fragmento recibido 1 (size: 131)
Fragmento recibido 3 (size: 121)
Fragmento recibido 4 (size: 111)
Fragmento recibido 5 (size: 138)
Error en la recepcion. Reiniciando secuencia
Fragmento recibido 0 (size: 118)
Fragmento recibido 1 (size: 118)
Fragmento recibido 2 (size: 136)
Fragmento recibido 3 (size: 113)
Fragmento recibido 4 (size: 112)
Fragmento recibido 5 (size: 129)
Exito en la recepcion. Finalizando secuencia
File reassembled as output.txt
```

Se puede ver que en la primera recepción se detecta que hay una pérdida de datos ya que el tamaño indicado en el primer fragmento no concuerda con el largo total y por lo tanto se reinicia la secuencia de envío de datos.

Del lado del publicador, la ejecucion del script `publisher-secure.py` muestra la siguiente salida:

```
$ python3 publisher-secure.py
Fragmento publicado 0 (size: 108)
Fragmento publicado 1 (size: 131)
Fragmento publicado 2 (size: 140)
Fragmento publicado 3 (size: 121)
Fragmento publicado 4 (size: 111)
Fragmento publicado 5 (size: 138)
Fragmento publicado -1 (size: 4)
Error de transmision. Reiniciando secuencia
Fragmento publicado 0 (size: 118)
Fragmento publicado 1 (size: 118)
Fragmento publicado 2 (size: 136)
Fragmento publicado 3 (size: 113)
Fragmento publicado 4 (size: 112)
Fragmento publicado 5 (size: 129)
Fragmento publicado 0 (size: 4)
Transmision exitosa. Finalizando secuencia
```

Se puede ver al igual que del lado del suscriptor que al detectar la pérdida de datos se reinicia la secuencia de envío

Referencias y Links Consultados

1. MQTT: The Standard for IoT Messaging
2. ¿Qué es MQTT?
3. Definición y detalles.
4. MQTT-SN
5. The Origin of MQTT
6. 3.1.1 upgrades
7. https://mqtt.org/use-cases/