



**FACULTAD  
DE INGENIERIA**

---

Universidad de Buenos Aires

Comunicación de Datos (86.12)

Trabajo Práctico 1: Nivel de Aplicaciones y de  
Transporte del Modelo TCP/IP

Domínguez Zandoná, Camila	109679	<a href="mailto:cdominguez@fi.uba.ar">cdominguez@fi.uba.ar</a>
Etcheverry, María Eugenia	109101	<a href="mailto:meetcheverry@fi.uba.ar">meetcheverry@fi.uba.ar</a>

## Objetivos

En este trabajo práctico se utilizará el protocolo MQTT (Message Queuing Telemetry Transport) para intercambiar mensajes en redes de dispositivos que requieren conectividad eficiente y bajo consumo, donde una de las integrantes sea la publicadora (enviadora de mensajes) y la otra la suscriptora (recibidora de mensajes).

Para ello, se utilizaron los códigos en *Python* brindados por la cátedra y se corrieron en la máquina virtual. La que hacía de suscriptora se conectó primera, y luego lo hizo la publicadora. De esta manera, una recibió el mensaje de la otra y se guardó en un archivo de salida, y se verificó que llegó el mensaje completo.

Luego, se modificó el código para simular la pérdida de un paquete intermedio y la que hacía de suscriptora verificó que en el archivo de salida el mensaje estaba truncado.

Finalmente, se implementaron mejoras vinculadas a la segmentación, los *checksum* presente en TCP, que sirve para verificar la integridad de los datos enviados, detectando si los datos han sido alterados o dañados durante su transmisión o almacenamiento

## Actividad 1

1. Investigar el protocolo MQTT en fuentes públicas y escribir una reseña (versiones, niveles de servicio, encriptación, productos comerciales y de código abierto: hay más conceptos que los mencionados arriba).

El protocolo **MQTT** o Message Queuing Telemetry Transport es un estándar de mensajería ligera para la transmisión de datos entre dispositivos, generalmente utilizado en el Internet de las Cosas (IoT), es decir en la interconexión digital de objetos cotidianos a través de internet, permitiendo que se conecten entre sí, y recompilen datos sin intervención humana.

Los dispositivos son intermediados por un servidor denominado broker, en este caso uno gratuito, disponible en Internet, que recibe los mensajes de los publicadores y los distribuye a los suscriptores interesados, gestionando las conexiones y el tráfico.

Los mensajes en MQTT se organizan por *topics*, donde los publicadores envían mensajes a un tema específico y los suscriptores se registran en los temas de su interés. Así, cada suscriptor recibirá solo los mensajes de los temas que le interesan.

El MQT fue diseñado por IBM en 1999, y fue pensado para redes con limitaciones en el ancho de banda y dispositivos con baja capacidad de procesamiento. Esta puede ser analizada a partir de diferentes aspectos, tales como:

- Versiones:
  - MQTT 3.1 Y 3.1.1: son las versiones iniciales y fueron los primeros estándares abiertos del protocolo. Estos se caracterizan por su simplicidad y eficiencia.
  - MQTT 5.0: esta es una versión más actual, la cual agrega nuevas características que mejoran la escalabilidad y la robustez del protocolo, como las propiedades adicionales en los mensajes, soporte para flujos de control más finos y mecanismos de diagnóstico. También introdujo un soporte para una mejor gestión de errores y motivos de desconexión.
- Niveles de Calidad de Servicio (QoS): hay tres niveles de calidad de servicio, lo cual permite garantizar la entrega acorde a las necesidades específicas de cada aplicación:
  - QoS (Al menos una vez): los mensajes se envían sin garantía de entrega. Generalmente se utiliza en situaciones en las que la pérdida ocasional de mensajes es aceptable, siendo que cuenta con una mayor rapidez y eficiencia.
  - QoS 1 (Entrega mínima una vez): el mensaje se envía al menos una vez, pero como el emisor sigue enviando hasta que recibe una confirmación, el mensaje puede llegar duplicado.
  - QoS 2 (Entrega una sola vez): es el nivel más lento y requiere mayor consumo de recursos, pero es el más fiable, ya que garantiza que el mensaje llegue exactamente una vez sin duplicados.
- Encriptación y Seguridad: el MQTT no incluye mecanismos de seguridad robustos por defecto, pero la seguridad se logra típicamente mediante diferentes prácticas:
  - TLS/SSL: se encapsulan las conexiones MQTT para asegurar la transmisión de datos cifrados, protegiendo así contra escuchas y ataque intermedios.
  - Autenticación: MQTT soporta autenticación mediante nombre de usuario y contraseña en la conexión, aunque no está cifrado a menos que se utilice TLS/SSL.
  - Autorización: los sistemas comerciales y de código abierto que implementan este tipo de colas permiten la gestión de permisos, restringiendo el acceso de usuarios a ciertos temas o recursos.

- Productos Comerciales: el MQTT se usa como protocolo de mensajería en distintos productos comerciales en la nube, tales como IBM Watson IoT Platform, Amazon Web Services IoT y Microsoft Azure IoT Hub
- Producto de código abierto: el protocolo MQTT tiene varias implementaciones de código abierto que permiten su uso gratuito y adaptable para diferentes aplicaciones, tales como Eclipse Mosquitto, EMQX o HiveMQ (edición de código abierto).
- Otras características importantes:
  - Retención de Mensajes: MQTT permite que el último mensaje publicado en un tema permanezca almacenado, para que nuevos suscriptores reciban inmediatamente la información más reciente.
  - Last Will and Testament: permite al cliente MQTT definir un mensaje que será enviado por el broker si el cliente se desconecta de manera inesperada.
  - Wildcard Topics: MQTT soporta el uso de comodines en los temas de suscripción para facilitar la suscripción a múltiples flujos de datos de forma eficiente.

## Ejecución de los scripts

2. 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. Detalles técnicos de ejecución de los scripts, en el Apéndice.

Utilizando las plataformas *Virtual Box* y *Wireshark* se realizó la transmisión de datos requerida. Primero, tal como se indicó, se inició la ejecución del *subscriber* y luego del archivo del *publisher*. En las figuras 1 y 2 se muestra el CMD del *subscriber* y *publisher* respectivamente.

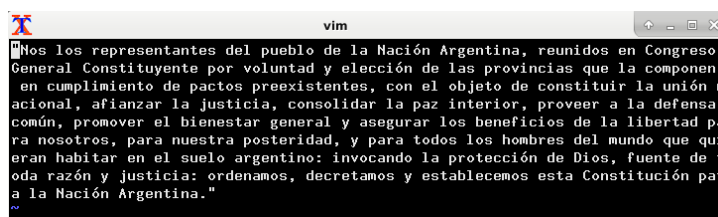
El contenido del archivo fue transmitido en fragmentos de largo variable por el publicador, y el suscriptor los recibió correctamente, reensamblándolos para generar el archivo completo. En ellos se ve que se enviaron y recibieron todos los paquetes. Además, se puede ver en el texto de salida del suscriptor, imagen 3, que el mensaje se recibió completo, y que no hubo pérdida de datos.

```
(tp1) root@comdatos:~/tp1_(Dominguez)_[Etcheverry]/scripts# python subscriber.py
Subscribed: [ReasonCode(Suback, 'Granted QoS 2')] []
Fragmento recibido 0 (size: 69)
Fragmento recibido 1 (size: 52)
Fragmento recibido 2 (size: 55)
Fragmento recibido 3 (size: 68)
Fragmento recibido 4 (size: 68)
Fragmento recibido 5 (size: 50)
Fragmento recibido 6 (size: 61)
Fragmento recibido 7 (size: 62)
Fragmento recibido 8 (size: 50)
Fragmento recibido 9 (size: 68)
Fragmento recibido 10 (size: 60)
File reassembled as output.txt
(tp1) root@comdatos:~/tp1_(Dominguez)_[Etcheverry]/scripts#
```

Figura 1: CMD del *subscriber* al recibir todos los paquetes.

```
(tp1) root@comdatos:~/Downloads/tp1_Dominguez_Etcheverry/scripts# python publisher.py
Fragmento publicado 0 (size: 69)
Fragmento publicado 1 (size: 52)
Fragmento publicado 2 (size: 55)
Fragmento publicado 3 (size: 68)
Fragmento publicado 4 (size: 68)
Fragmento publicado 5 (size: 50)
Fragmento publicado 6 (size: 61)
Fragmento publicado 7 (size: 62)
Fragmento publicado 8 (size: 50)
Fragmento publicado 9 (size: 68)
Fragmento publicado 10 (size: 60)
(tp1) root@comdatos:~/Downloads/tp1_Dominguez_Etcheverry/scripts#
```

Figura 2: CMD del *publisher* al enviar todos los paquetes.



```
vim
"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 n
acional, afianzar la justicia, consolidar la paz interior, proveer a la defensa
común, promover el bienestar general y asegurar los beneficios de la libertad pa
ra nosotros, para nuestra posteridad, y para todos los hombres del mundo que qui
eran habitar en el suelo argentino: invocando la protección de Dios, fuente de t
oda razón y justicia: ordenamos, decretamos y establecemos esta Constitución par
a la Nación Argentina."
```

Figura 3: Archivo de texto que recibe *subscriber* reensamblado.

Por último, se puede analizar las capturas de pantalla del *Wireshark* en este proceso, tanto para el *subscriber* en las figuras 4 y 5, como para el *publisher* en 6. En ellas se puede ver el tráfico MQTT entre las dos direcciones IP entre el suscriptor y el publicador. Ambos están interactuando con el mismo *broker*, (3.66.109.246) y tienen la misma conexión mediante mensajes de "Ping". El patrón de publicación y recepción se sigue correctamente, lo cual confirma lo anterior de que el intercambio de datos MQTT está funcionando como se espera y no se está truncando el mensaje.

No.	Time	Source	Destination	Protocol	Length	Info
18	0.0152337	10.0.2.15	3.66.109.246	MQTT	87	Connect Command
19	0.0152337	3.66.109.246	10.0.2.15	MQTT	87	Subscribe Request (id=1) [tpl.Dominguez_Etcheverry]
20	0.0152337	3.66.109.246	10.0.2.15	MQTT	65	Connect Ack, Subscribe Ack (id=1)
21	0.0152337	3.66.109.246	10.0.2.15	MQTT	50	Ping Request
22	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Ping Response
23	0.0152337	3.66.109.246	10.0.2.15	MQTT	50	Ping Request
24	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Ping Response
25	0.0152337	3.66.109.246	10.0.2.15	MQTT	50	Ping Request
26	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Ping Response
27	0.0152337	3.66.109.246	10.0.2.15	MQTT	50	Ping Request
28	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Ping Response
29	0.0152337	3.66.109.246	10.0.2.15	MQTT	50	Ping Request
30	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Ping Response
31	0.0152337	3.66.109.246	10.0.2.15	MQTT	163	Publish Message (id=41) [tpl.Dominguez_Etcheverry]
32	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=41)
33	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=41)
34	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=41)
35	0.0152337	3.66.109.246	10.0.2.15	MQTT	145	Publish Message (id=101) [tpl.Dominguez_Etcheverry]
36	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=101)
37	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=101)
38	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=101)
39	0.0152337	3.66.109.246	10.0.2.15	MQTT	149	Publish Message (id=151) [tpl.Dominguez_Etcheverry]
40	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=151)
41	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=151)
42	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=151)
43	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=151)
44	0.0152337	3.66.109.246	10.0.2.15	MQTT	267	Publish Message (id=201) [tpl.Dominguez_Etcheverry], Publish Message (id=201)
45	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=201)
46	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=201)
47	0.0152337	3.66.109.246	10.0.2.15	MQTT	64	Publish Release (id=201), Publish Release (id=202)
48	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=201)
49	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=202)
50	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=202)
51	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=202)

Figura 4: Primer captura del Wireshark de MQTT del subscriber.

No.	Time	Source	Destination	Protocol	Length	Info
77	0.0152337	10.0.2.15	3.66.109.246	MQTT	60	Publish Complete (id=151)
78	0.0152337	3.66.109.246	10.0.2.15	MQTT	267	Publish Message (id=201) [tpl.Dominguez_Etcheverry], Publish Message (id=201)
79	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=201)
80	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=201)
81	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=201)
82	0.0152337	3.66.109.246	10.0.2.15	MQTT	64	Publish Release (id=201), Publish Release (id=202)
83	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=201)
84	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=202)
85	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=202)
86	0.0152337	3.66.109.246	10.0.2.15	MQTT	144	Publish Message (id=251) [tpl.Dominguez_Etcheverry]
87	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=251)
88	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=251)
89	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=251)
90	0.0152337	3.66.109.246	10.0.2.15	MQTT	154	Publish Message (id=301) [tpl.Dominguez_Etcheverry]
91	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=301)
92	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=301)
93	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=301)
94	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=301)
95	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=301)
96	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=301)
97	0.0152337	3.66.109.246	10.0.2.15	MQTT	155	Publish Message (id=351) [tpl.Dominguez_Etcheverry]
98	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=351)
99	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=351)
100	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=351)
101	0.0152337	3.66.109.246	10.0.2.15	MQTT	143	Publish Message (id=401) [tpl.Dominguez_Etcheverry]
102	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=401)
103	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=401)
104	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=401)
105	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=401)
106	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=401)
107	0.0152337	3.66.109.246	10.0.2.15	MQTT	163	Publish Message (id=451) [tpl.Dominguez_Etcheverry]
108	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=451)
109	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=451)
110	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=451)
111	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=451)
112	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=451)
113	0.0152337	3.66.109.246	10.0.2.15	MQTT	156	Publish Message (id=501) [tpl.Dominguez_Etcheverry]
114	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=501)
115	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=501)
116	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=501)
117	0.0152337	3.66.109.246	10.0.2.15	MQTT	156	Publish Message (id=551) [tpl.Dominguez_Etcheverry]
118	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Received (id=551)
119	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=551)
120	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=551)
121	0.0152337	3.66.109.246	10.0.2.15	MQTT	60	Publish Complete (id=551)
122	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Release (id=551)

Figura 5: Segunda captura del Wireshark de MQTT del subscriber.

No.	Time	Source	Destination	Protocol	Length	Info
18	0.0152337	10.0.2.15	3.66.109.246	MQTT	87	Connect Command
19	0.0152337	3.66.109.246	10.0.2.15	MQTT	163	Publish Message (id=1) [tpl.Dominguez_Etcheverry]
20	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Connect Ack
21	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=1)
22	0.0152337	3.66.109.246	10.0.2.15	MQTT	145	Publish Message (id=2) [tpl.Dominguez_Etcheverry]
23	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=2)
24	0.0152337	3.66.109.246	10.0.2.15	MQTT	149	Publish Message (id=3) [tpl.Dominguez_Etcheverry]
25	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=3)
26	0.0152337	3.66.109.246	10.0.2.15	MQTT	162	Publish Message (id=4) [tpl.Dominguez_Etcheverry]
27	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=4)
28	0.0152337	3.66.109.246	10.0.2.15	MQTT	161	Publish Message (id=5) [tpl.Dominguez_Etcheverry]
29	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=5)
30	0.0152337	3.66.109.246	10.0.2.15	MQTT	144	Publish Message (id=6) [tpl.Dominguez_Etcheverry]
31	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=6)
32	0.0152337	3.66.109.246	10.0.2.15	MQTT	154	Publish Message (id=7) [tpl.Dominguez_Etcheverry]
33	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=7)
34	0.0152337	3.66.109.246	10.0.2.15	MQTT	155	Publish Message (id=8) [tpl.Dominguez_Etcheverry]
35	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=8)
36	0.0152337	3.66.109.246	10.0.2.15	MQTT	143	Publish Message (id=9) [tpl.Dominguez_Etcheverry]
37	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=9)
38	0.0152337	3.66.109.246	10.0.2.15	MQTT	163	Publish Message (id=10) [tpl.Dominguez_Etcheverry]
39	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=10)
40	0.0152337	3.66.109.246	10.0.2.15	MQTT	156	Publish Message (id=11) [tpl.Dominguez_Etcheverry]
41	0.0152337	3.66.109.246	10.0.2.15	MQTT	62	Publish Received (id=11)
42	0.0152337	3.66.109.246	10.0.2.15	MQTT	58	Disconnect Req

Figura 6: Captura del Wireshark de MQTT del publisher.

### 3. Sobre la aplicación, ¿de qué modo consigue implementar la fragmentación?

En el código del *publisher*, la fragmentación de los mensajes se implementa manualmente a nivel de la aplicación. Esto sigue una serie de pasos:

1. Lectura del contenido completo: el publicador lee el contenido completo del archivo de texto (input.txt) y lo guarda en una variable como un solo bloque de texto.
2. División en fragmentos: se divide el contenido en fragmentos de tamaño variable, el cual se determina aleatoriamente dentro de un rango definido (entre 50 y 70 caracteres). Esta fragmentación no está basada en ningún límite impuesto por MQTT, sino que es controlada por la aplicación misma. Esto se puede ver en el fragmento de código:

```
fragment_size = random.randint(min_size, max_size) # Tamaño aleatorio entre 50 y 70
fragment = content[index:index+fragment_size] # Extrae el fragmento del contenido
```

Cada fragmento contiene una porción del archivo original, y estos fragmentos son procesados secuencialmente hasta que todo el archivo ha sido publicado.

3. Metadatos en los mensajes: cada fragmento lleva metadatos en el mensaje publicado, el número de fragmentos, el tamaño del fragmento y el indicador del último fragmento. Estos metadatos se incluyen en el payload del mensaje publicado en formato delimitado por '—':

```
payload = f'{{fragment_number}}|{{fragment_size}}|{{total_length}}|{{is_last}}|{{fragment}}'
```

4. Publicación de los fragmentos: cada fragmento se publica en el broker MQTT utilizando QoS 2 (exactamente una vez). De esta forma, el publicador garantiza que los mensajes lleguen correctamente, aunque puedan llegar en orden diferente o ser reenviados en caso de problemas de red:

```
client.publish(topic, payload, qos=2, retain=False)
```

5. Fragmentación manual: esta técnica de fragmentación es manual, lo que significa que la responsabilidad de dividir los mensajes, enviar metadatos y garantizar la reconstrucción del mensaje completo en el receptor recae sobre la lógica de la aplicación, no sobre el protocolo MQTT en sí.

4. 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?

La fragmentación de los scripts tiene similitud con la fragmentación del protocolo TCP, ya que ambos fragmentan datos en segmentos antes de iniciar la transmisión. En el caso del script, estos segmentos serían los mensajes a mandar. Durante el reensamblado de los fragmentos, el receptor recibe los fragmentos y los ensambla nuevamente para reconstruir el mensaje original, de forma análoga se realiza en el protocolo TCP.

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

Las funcionalidades de TCP ausentes son las siguientes:

- **Control de errores y retransmisión:** TCP incorpora mecanismos donde se verifican los segmentos recibidos mediante sumas de verificación, de esta forma si un segmento se pierda o tiene fallas, TCP lo retransmite.
- **Control de Flujo:** TCP utiliza un sistema de control de flujo que ajusta la cantidad de datos que se pueden enviar antes de recibir la confirmación del suscriptor.
- **Orden de los fragmentos:** TCP garantiza la recepción en el orden correcto incluso si la red los envía de manera desordenada.
- **Control de congestión:** TCP cuenta con algoritmos para evitar congestión de la red.

6. 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?

En el protocolo MQTT las sesiones se establecen entre los clientes (publicador y suscriptores) y un servidor central denominado *broker*.

Por un lado, tanto el publicador como el suscriptor son considerados clientes en MQTT, y son uno de los extremos de las sesiones, ya que ambos se conectan al *broker*, el otro extremo de las sesiones, para enviar o recibir mensajes, pero no se conectan entre sí. El publicador es el cliente que envía mensajes al *broker*, publicando datos en temas específicos. El suscriptor es el cliente que escucha temas específicos en el *broker* y recibe los mensajes publicados por otros clientes. Por el otro, el *broker* MQTT actúa como el servidor en el modelo de comunicación. Es el intermediario entre publicadores y suscriptores, gestionando el flujo de mensajes y asegurando que los datos lleguen a los suscriptores adecuado.

Uno de los extremos de las sesiones es el cliente publicador abre una sesión con el *broker* cuando se conecta para publicar el mensaje. Cada mensaje que publica se envía al *broker*, que actúa como el otro extremo de la sesión. El otro extremo es el cliente suscripto, que establece otra sesión con el *broker*, que actúa como el servidor también en este caso. El suscriptor recibe mensajes del *broker* que coinciden con los temas a los que se ha suscrito.

A su vez, esto se puede ver en las capturas del *Wireshark*, que se muestran en las figuras 7 y 8.

60	386.186352019	3.66.109.246	10.0.2.15	MQTT	163 Publish Message (id=51) [tpl_Dominguez_Etcheverry]
62	386.190966633	10.0.2.15	3.66.109.246	MQTT	60 Publish Received (id=51)
64	387.044585380	3.66.109.246	10.0.2.15	MQTT	62 Publish Release (id=51)
65	387.049644966	10.0.2.15	3.66.109.246	MQTT	60 Publish Complete (id=51)
67	388.041537164	3.66.109.246	10.0.2.15	MQTT	145 Publish Message (id=101) [tpl_Dominguez_Etcheverry]
68	388.043387456	10.0.2.15	3.66.109.246	MQTT	60 Publish Received (id=101)
70	388.429880749	3.66.109.246	10.0.2.15	MQTT	62 Publish Release (id=101)
71	388.425427645	10.0.2.15	3.66.109.246	MQTT	60 Publish Complete (id=101)
73	389.097631214	3.66.109.246	10.0.2.15	MQTT	149 Publish Message (id=151) [tpl_Dominguez_Etcheverry]

Figura 7: Captura del *Wireshark* del suscriptor para ver extremos.

14	0.635288025	3.66.109.246	10.0.2.15	MQTT	62 Publish Received (id=1)
16	1.365248694	10.0.2.15	3.66.109.246	MQTT	145 Publish Message (id=2) [tpl_Dominguez_Etcheverry]
18	1.630036694	3.66.109.246	10.0.2.15	MQTT	62 Publish Received (id=2)
20	2.386064875	10.0.2.15	3.66.109.246	MQTT	149 Publish Message (id=3) [tpl_Dominguez_Etcheverry]
22	2.664054627	3.66.109.246	10.0.2.15	MQTT	62 Publish Received (id=3)
24	3.390344005	10.0.2.15	3.66.109.246	MQTT	162 Publish Message (id=4) [tpl_Dominguez_Etcheverry]
26	3.717676883	3.66.109.246	10.0.2.15	MQTT	62 Publish Received (id=4)
28	4.408785096	10.0.2.15	3.66.109.246	MQTT	161 Publish Message (id=5) [tpl_Dominguez_Etcheverry]

Figura 8: Captura del *Wireshark* del publicador para ver extremos.

En la primera, el suscriptor actúa como cliente, donde se ve que se publicaron los datos en el *broker* y luego se los envía al suscriptor. En la segunda, se ve como el publicador le envía los datos al *broker*, y este le envía una confirmación de receptor. Es por ello, que tanto el suscriptor como el publicador actúan como clientes y el *broker* como servidor.

7. 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)?

En el protocolo MQTT, las sesiones pueden ser persistentes o no persistentes, dependiendo de cómo se configuren. Esto afecta cómo se manejan las conexiones y los mensajes cuando un cliente (publicador o suscriptor) está desconectado. Por un lado, las sesiones pueden ser persistentes si el cliente lo solicita, permitiendo que las suscripciones y mensajes se almacenen y se entreguen cuando el cliente se vuelve a conectar. Por el otro, al desconectarse el cliente la sesión se cierra.

El protocolo MQTT crea, por defecto, sesiones no persistentes a menos que se indique lo contrario, siendo que en la sección del código:

```
client.connect(broker_address, 1883, 60)
```

no se especifica nada del tipo de conexión. Si se quisiera hacer persistentes, se debería agregar como parámetro de la función:

```
clean_session=False .
```

A su vez, esto también se puede ver en *Wireshark*, en la figura 9, en el flag *Clean Session Flag*, que al estar habilitada indica que la conexión es no persistente.

```
> Frame 95: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
> Linux cooked capture
> Internet Protocol Version 4, Src: 10.0.2.15, Dst: 3.69.200.245
> Transmission Control Protocol, Src Port: 57175, Dst Port: 1883, Seq: 1, Ack: 1, Len: 14
> MQ Telemetry Transport Protocol, Connect Command
  > Header Flags: 0x10, Message Type: Connect Command
    Msg Len: 12
    Protocol Name Length: 4
    Protocol Name: MQTT
    Version: MQTT v3.1.1 (4)
  > Connect Flags: 0x02, QoS Level: At most once delivery (Fire and Forget), Clean Session Flag
    0... .. = User Name Flag: Not set
    .0... .. = Password Flag: Not set
    .0... .. = Will Retain: Not set
    ...0... = QoS Level: At most once delivery (Fire and Forget) (0)
    ....0... = Will Flag: Not set
    ....1... = Clean Session Flag: Set
    ....0... = (Reserved): Not set
  Keep Alive: 60
  Client ID Length: 0
  Client ID:
```

Figura 9: Captura del *Wireshark* para comprobar que la sesión es no persistente.

8. 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?

Para esta parte del trabajo cambiamos el filtro de *Wireshark* para que no solo aparecieran los MQTT, sino también los TCP.

- a. Tenemos dos números de secuencia del primer segmento TCP de perición de conexión, el normal y el relativo, como se puede ver respectivamente en las figuras 10 y 11. Mientras que en el relatio vemos que empieza en 0, el otro tiene un número aleatorio, lo cual ayuda con la seguridad del protocolo.

```
Transmission Control Protocol, Src Port: 46359, Dst Port: 1883, Seq: 326736124, Len: 0
Source Port: 46359
Destination Port: 1883
[Stream index: 0]
[TCP Segment Len: 0]
Sequence number: 326736124
[Next sequence number: 326736124]
Acknowledgment number: 0
1010 ..... = Header Length: 40 bytes (10)
Flags: 0x002 (SYN)
```

Figura 10: Captura del *Wireshark* del *publisher* para ver el número de secuencia.

```
Transmission Control Protocol, Src Port: 46359, Dst Port: 1883, Seq: 0, Len: 0
Source Port: 46359
Destination Port: 1883
[Stream index: 0]
[TCP Segment Len: 0]
Sequence number: 0 (relative sequence number)
[Next sequence number: 0 (relative sequence number)]
Acknowledgment number: 0
1010 ..... = Header Length: 40 bytes (10)
Flags: 0x002 (SYN)
```

Figura 11: Captura del *Wireshark* del *publisher* para ver el número de secuencia relativo.

- b. Como se ve en la imagen 12, hay varias opciones implementadas, las cuales constante de *maximum segment size*, que indica el tamaño máximo de un segmento de datos que puede ser enviado en un solo paquete de TCP, *SACK permitted*, el cual permite a un receptor informar al emisor sobre los segmentos de datos que han sido recibidos correctamente, así como aquellos que faltan, *time stamp ption*, se utiliza para mejorar la gestión de la sincronización y el rendimiento de las conexiones TCP, *No-operation*, se utiliza principalmente como un relleno.<sup>en</sup> las cabeceras TCP y *Window Scale*, que se utiliza para permitir un tamaño de ventana de recepción más grande en conexiones TCP.

```
Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
  TCP Option - Maximum segment size: 1460 bytes
    Kind: Maximum Segment Size (2)
    Length: 4
    MSS Value: 1460
  TCP Option - SACK permitted
    Kind: SACK Permitted (4)
    Length: 2
  TCP Option - Timestamps: Tsval 31569709, TSecr 0
    Kind: Time Stamp Option (8)
    Length: 10
    Timestamp value: 31569709
    Timestamp echo reply: 0
  TCP Option - No-Operation (NOP)
    Kind: No-Operation (1)
  TCP Option - Window scale: 7 (multiply by 128)
    Kind: Window Scale (3)
    Length: 3
    Shift count: 7
    [Multiplier: 128]
```

Figura 12: Captura del *Wireshark* para ver las opciones implementadas.

- c. Para ver la cantidad de bytes que tiene el buffer de recepción según se informa al inicio de puede ver en la imagen 13

```
Window size value: 65535
[Calculated window size: 65535]
Checksum: 0xbd3f [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
```

Figura 13: Captura del *Wireshark* para ver la cantidad de bytes en la secuencia, en *window size value*.

- d. Para ver la hora a la que se envió el primer segmento se puede mirar la figura 14, mientras que la hora de recepción se puede ver en la figura 15.

```
Arrival Time: Oct 9, 2024 11:14:57.966146214 -03
[Time shift for this packet: 0.000000000 seconds]
Epoch Time: 1728483297.966146214 seconds
[Time delta from previous captured frame: 0.000910755 seconds]
[Time delta from previous displayed frame: 0.000910755 seconds]
[Time since reference or first frame: 41.994786192 seconds]
```

Figura 14: Captura del *Wireshark* del publicador para ver la hora de envío del primer segmento.



```
Arrival Time: Oct 9, 2024 11:14:59.343516063 -03
[Time shift for this packet: 0.000000000 seconds]
Epoch Time: 1728483299.343516063 seconds
[Time delta from previous captured frame: 0.000806804 seconds]
[Time delta from previous displayed frame: 0.000806804 seconds]
```

Figura 15: Captura del *Wireshark* del suscriptor para ver la hora de recibo del primer segmento.

Finalmente, para ver cuál es la RTT (Round Trip Time) hay que restarle al horario de recepción el horario de envío del primer segmento de datos. Teniendo que el envío fue a las 11 : 14 : 57,966146214, y el recepción a las 11 : 14 : 59,343516063, tenemos un RTT = 1,377369849 segundos.

- e. Para ver cuál es la longitud de cada uno de los primeros cuatro segmentos TCP que transportan datos se pueden ver en la figura 16, específicamente en la columna de longitud en la fila de recepción, es decir las que dicen *Publish Message* (*id = n*). Vemos que los primeros cuatro tienen una longitud de 158, 161, 155 y 153 bytes.

11	155.101261432	10.0.2.15	3.69.200.245	MQTT	158 Publish Message (id=1) [tpl.Dominguez.Etcheverry]
12	155.101880710	3.69.200.245	10.0.2.15	TCP	62 1883 -> 41139 [ACK] Seq=1 Ack=117 Win=65535 Len=0
13	155.364557935	3.69.200.245	10.0.2.15	MQTT	62 Connect Ack
14	155.364600026	10.0.2.15	3.69.200.245	TCP	56 41139 -> 1883 [ACK] Seq=117 Ack=5 Win=64236 Len=0
15	155.468712464	3.69.200.245	10.0.2.15	MQTT	62 Publish Received (id=1)
16	155.468753822	10.0.2.15	3.69.200.245	TCP	56 41139 -> 1883 [ACK] Seq=117 Ack=9 Win=64232 Len=0
17	155.107629568	10.0.2.15	3.69.200.245	MQTT	153 Publish Message (id=2) [tpl.Dominguez.Etcheverry]
18	156.108697562	3.69.200.245	10.0.2.15	TCP	62 1883 -> 41139 [ACK] Seq=9 Ack=222 Win=65535 Len=0
19	156.487992868	3.69.200.245	10.0.2.15	MQTT	62 Publish Received (id=2)
20	156.488039038	10.0.2.15	3.69.200.245	TCP	56 41139 -> 1883 [ACK] Seq=222 Ack=13 Win=64228 Len=0
21	157.112876990	10.0.2.15	3.69.200.245	MQTT	155 Publish Message (id=3) [tpl.Dominguez.Etcheverry]
22	157.113584029	3.69.200.245	10.0.2.15	TCP	62 1883 -> 41139 [ACK] Seq=13 Ack=321 Win=65535 Len=0
23	157.430826205	3.69.200.245	10.0.2.15	MQTT	62 Publish Received (id=3)
24	157.430864993	10.0.2.15	3.69.200.245	TCP	56 41139 -> 1883 [ACK] Seq=321 Ack=17 Win=64224 Len=0
25	158.116987883	10.0.2.15	3.69.200.245	MQTT	153 Publish Message (id=4) [tpl.Dominguez.Etcheverry]
26	158.117683070	3.69.200.245	10.0.2.15	TCP	62 1883 -> 41139 [ACK] Seq=17 Ack=418 Win=65535 Len=0
27	158.377542047	3.69.200.245	10.0.2.15	MQTT	62 Publish Received (id=4)
28	158.377591049	10.0.2.15	3.69.200.245	TCP	56 41139 -> 1883 [ACK] Seq=418 Ack=21 Win=64220 Len=0

Figura 16: Captura del *Wireshark* del *publisher* para ver la longitud de los primeros cuatro segmentos que transportan datos.

## Implementación

9. 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.

En el código del *publisher* brindado por la cátedra se utiliza QoS 2 para garantizar la entrega de los mensajes, siendo que lee el archivo, lo divide en diferentes fragmentos y publica los fragmentos con metadatos. El código del *subscriber* recibe los fragmentos y recupera el archivo original uniéndolos. Sin embargo, en ningún momento se comunica la cantidad de datos que se van a transmitir, y por lo tanto ante la pérdida de paquetes, el suscriptor no se entera de dicha pérdida.

Por lo tanto, se realizaron cambios en el código de ambos clientes con el fin de conseguir una pérdida de paquetes y tener una verificación por parte del suscriptor para validar la cantidad de datos recibidos. Para ello, en el nuevo código del publicador incluye la longitud total del archivo en cada mensaje, y el suscriptor compara la cantidad de datos recibidos con el total esperado, y muestra un error si falta algún fragmento. Además, se guarda en un nuevo archivo de texto el mensaje truncado, tal como se puede ver en la imagen 17.

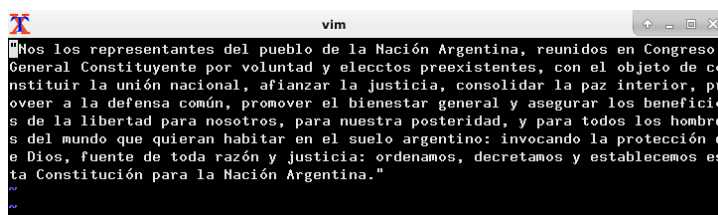


Figura 17: Archivo de texto que recibe el *subscriber* truncado.

Por otro lado, en las figuras 18 y 19 se puede ver, como se mencionó anteriormente, la recepción y el envío de paquetes respectivamente. En ellos se ve como se omite el paquete número 2 para simular la pérdida en el CMD del *publisher* y como en el CMD del *subscriber* no se recibe dicho paquete y anuncia el error diciendo que el archivo se encuentra truncado.



```
(tp1) root@comdatos:~/tp1_Dominguez_Etcheverry/scripts# python subscriber_truncated_modified.py
Subscribed: [ReasonCode(Suback, 'Granted QoS 2')] []
Fragmento recibido 0 (size: 56)
Fragmento recibido 1 (size: 65)
Fragmento recibido 3 (size: 59)
Fragmento recibido 4 (size: 59)
Fragmento recibido 5 (size: 56)
Fragmento recibido 6 (size: 54)
Fragmento recibido 7 (size: 67)
Fragmento recibido 8 (size: 61)
Fragmento recibido 9 (size: 52)
Fragmento recibido 10 (size: 56)
Fragmento recibido 11 (size: 50)
Error: Fragmentos incompletos, el archivo está truncado.
Archivo guardado como output2.txt con contenido truncado.
(tp1) root@comdatos:~/tp1_Dominguez_Etcheverry/scripts#
```

Figura 18: CMD del *subscriber* al recibir los datos truncados.

```
(tp1) root@comdatos:~/tp1_Dominguez_Etcheverry/scripts# python publisher_modified.py
Fragmento publicado 0 (size: 56)
Fragmento publicado 1 (size: 65)
Fragmento 2 omitido para simular pérdida
Fragmento publicado 3 (size: 59)
Fragmento publicado 4 (size: 59)
Fragmento publicado 5 (size: 56)
Fragmento publicado 6 (size: 54)
Fragmento publicado 7 (size: 67)
Fragmento publicado 8 (size: 61)
Fragmento publicado 9 (size: 52)
Fragmento publicado 10 (size: 56)
Fragmento publicado 11 (size: 50)
(tp1) root@comdatos:~/tp1_Dominguez_Etcheverry/scripts#
```

Figura 19: CMD del *publisher* al enviar los datos truncados.

Ahora bien, también se pueden analizar las capturas del *Wireshark* tanto del suscriptor como del publicador. Para el primero vemos las imágenes

No.	Time	Source	Destination	Protocol	Length	Info
8	0.279319124	10.0.2.15	3.77.181.188	MQTT	70	Connect Command
10	0.279319124	10.0.2.15	3.77.181.188	MQTT	87	Subscribe Request (id=1) [tp1_Dominguez_Etcheverry]
12	0.533817901	3.77.181.188	10.0.2.15	MQTT	62	Connect Ack
14	0.541372443	3.77.181.188	10.0.2.15	MQTT	62	Subscribe Ack (id=1)
16	5.584658755	3.77.181.188	10.0.2.15	MQTT	154	Publish Message (id=51) [tp1_Dominguez_Etcheverry]
18	5.586449668	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=51)
20	5.991371567	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=51)
21	5.992686664	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=51)
23	6.687276623	3.77.181.188	10.0.2.15	MQTT	162	Publish Message (id=101) [tp1_Dominguez_Etcheverry]
24	6.687699646	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=101)
26	6.938280563	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=101)
27	6.939436409	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=101)
29	9.339954555	3.77.181.188	10.0.2.15	MQTT	157	Publish Message (id=151) [tp1_Dominguez_Etcheverry]
30	9.340485329	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=151)
32	9.709778431	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=151)
33	9.710999092	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=151)
35	9.965619428	3.77.181.188	10.0.2.15	MQTT	156	Publish Message (id=201) [tp1_Dominguez_Etcheverry]
36	9.966089517	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=201)
38	10.223582173	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=201)
39	10.224689332	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=201)
41	10.599453024	3.77.181.188	10.0.2.15	MQTT	154	Publish Message (id=251) [tp1_Dominguez_Etcheverry]
42	10.600328394	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=251)
44	10.853178345	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=251)
45	10.854385334	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=251)
47	11.622084442	3.77.181.188	10.0.2.15	MQTT	151	Publish Message (id=301) [tp1_Dominguez_Etcheverry]
48	11.622480567	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=301)
50	11.878791402	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=301)
51	11.880153569	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=301)
53	12.607434879	3.77.181.188	10.0.2.15	MQTT	164	Publish Message (id=351) [tp1_Dominguez_Etcheverry]
54	12.607861959	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=351)
56	12.865276535	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=351)
57	12.866587312	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=351)
59	13.643352650	3.77.181.188	10.0.2.15	MQTT	159	Publish Message (id=401) [tp1_Dominguez_Etcheverry]

Figura 20: Primera captura del *Wireshark* del suscriptor con los datos truncados.

60	13.643773443	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=401)
62	13.896668000	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=401)
63	13.898471724	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=401)
65	14.619685728	3.77.181.188	10.0.2.15	MQTT	150	Publish Message (id=451) [tp1_Dominguez_Etcheverry]
66	14.620116661	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=451)
68	14.882911121	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=451)
69	14.884033442	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=451)
71	15.618205245	3.77.181.188	10.0.2.15	MQTT	155	Publish Message (id=501) [tp1_Dominguez_Etcheverry]
72	15.618572135	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=501)
74	15.872525444	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=501)
75	15.874281879	10.0.2.15	3.77.181.188	MQTT	60	Publish Complete (id=501)
77	16.687884099	3.77.181.188	10.0.2.15	MQTT	116	Publish Message (id=551) [tp1_Dominguez_Etcheverry]
78	16.688318395	10.0.2.15	3.77.181.188	MQTT	60	Publish Received (id=551)
80	16.944358884	3.77.181.188	10.0.2.15	MQTT	62	Publish Release (id=551)

Figura 21: Segunda captura del *Wireshark* del suscriptor con los datos truncados.

10. 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.

Se decidió por agregar la funcionalidad *Checksum*. Esta funcionalidad se utiliza para verificar la integridad de los datos transmitidos o almacenados. Los pasos a seguir para aplicar esta funcionalidad son los siguientes:

- Se genera un fragmento de datos, se calcula su checksum usando una función de hash. Este valor representa el contenido del fragmento.
- Transmisión.
- Verificación: Al recibir el fragmento, el receptor calcula el checksum del mismo fragmento.
- Comparación:
  - Si coinciden: Los datos no han sido alterados mediante la transmisión y se recibieron correctamente.
  - Si no coincide: Los datos transmitidos se han corrompido, esto puede generar que el mismo sea rechazado o se vuelva a pedir.

La funcionalidad *checksum* asegura la integridad de los datos, seguridad y la detección de errores.

Para implementar esta funcionalidad en los *scripts*, se utilizó la librería *zlib* de *Python*. En el caso del *publisher* se implementó la función *zlib.crc32()* la cual realiza el cálculo del *checksum* del fragmento.

```
import zlib
def calculate_checksum(data):
    return zlib.crc32(data.encode())
```

Luego, en la transmisión de datos se asegura que cada fragmento de información enviado pueda ser verificado a través de un *checksum*.

Del mismo modo, para el *subscriber* se verifica el *checksum* al recibir los datos. En las próximas figuras se puede observar las capturas de *cmd* y *Wireshark* del *subscriber* y *publisher* respectivamente.

```
(tp1) root@comdatos:~/tp1_Dominguez_Etcheverry/scripts# python subscriber_1.py
Subscribed: [ReasonCode(Suback, 'Granted QoS 2')] []
Fragmento 0 recibido correctamente con checksum válido.
Fragmento 1 recibido correctamente con checksum válido.
Fragmento 3 recibido correctamente con checksum válido.
Fragmento 4 recibido correctamente con checksum válido.
Fragmento 5 recibido correctamente con checksum válido.
Fragmento 6 recibido correctamente con checksum válido.
Fragmento 7 recibido correctamente con checksum válido.
Fragmento 8 recibido correctamente con checksum válido.
Fragmento 9 recibido correctamente con checksum válido.
Fragmento 10 recibido correctamente con checksum válido.
Fragmento 11 recibido correctamente con checksum válido.
Error: Fragmentos incompletos, el archivo está truncado.
```

Figura 22: Captura de CMD del suscriptor con la funcionalidad *Checksum*.

```
(tp1) root@comdatos:~/tp1_Dominguez_Etcheverry/scripts# python publisher_with_checksum.py
Fragmento publicado 0 (size: 54)
Fragmento publicado 1 (size: 54)
Fragmento 2 omitido para simular pérdida
Fragmento publicado 3 (size: 62)
Fragmento publicado 4 (size: 58)
Fragmento publicado 5 (size: 65)
Fragmento publicado 6 (size: 56)
Fragmento publicado 7 (size: 54)
Fragmento publicado 8 (size: 64)
Fragmento publicado 9 (size: 52)
Fragmento publicado 10 (size: 52)
Fragmento publicado 11 (size: 62)
(tp1) root@comdatos:~/tp1_Dominguez_Etcheverry/scripts#
```

Figura 23: Captura CMD del publicador con la funcionalidad *checksum*.

No.	Time	Source	Destination	Protocol	Length	Info
246	29.435255183	10.0.2.15	3.69.104.152	MQTT	70	Connect Command
248	29.436181881	10.0.2.15	3.69.104.152	MQTT	87	Subscribe Request (id=1) [tp1_Dominguez_Etcheverry]
250	29.702364774	3.69.104.152	10.0.2.15	MQTT	62	Connect Ack
252	29.709821765	3.69.104.152	10.0.2.15	MQTT	62	Subscribe Ack (id=1)
344	41.274744285	3.69.104.152	10.0.2.15	MQTT	163	Publish Message (id=51) [tp1_Dominguez_Etcheverry]
346	41.275501020	10.0.2.15	3.69.104.152	MQTT	60	Publish Received (id=51)
348	41.533260871	3.69.104.152	10.0.2.15	MQTT	60	Publish Release (id=51)
349	41.535379910	10.0.2.15	3.69.104.152	MQTT	60	Publish Complete (id=51)
351	42.306785327	3.69.104.152	10.0.2.15	MQTT	162	Publish Message (id=101) [tp1_Dominguez_Etcheverry]
352	42.307089539	10.0.2.15	3.69.104.152	MQTT	60	Publish Received (id=101)
354	42.598678806	3.69.104.152	10.0.2.15	MQTT	62	Publish Release (id=101)
355	42.600308102	10.0.2.15	3.69.104.152	MQTT	60	Publish Complete (id=101)
359	44.276835126	3.69.104.152	10.0.2.15	MQTT	169	Publish Message (id=151) [tp1_Dominguez_Etcheverry]
360	44.277161664	10.0.2.15	3.69.104.152	MQTT	60	Publish Received (id=151)
362	44.538894979	3.69.104.152	10.0.2.15	MQTT	62	Publish Release (id=151)
363	44.540684733	10.0.2.15	3.69.104.152	MQTT	60	Publish Complete (id=151)
365	45.283943176	3.69.104.152	10.0.2.15	MQTT	167	Publish Message (id=201) [tp1_Dominguez_Etcheverry]
366	45.284352165	10.0.2.15	3.69.104.152	MQTT	60	Publish Received (id=201)
368	45.553304064	3.69.104.152	10.0.2.15	MQTT	62	Publish Release (id=201)
369	45.555083344	10.0.2.15	3.69.104.152	MQTT	60	Publish Complete (id=201)
371	46.318255902	3.69.104.152	10.0.2.15	MQTT	174	Publish Message (id=251) [tp1_Dominguez_Etcheverry]
372	46.318583175	10.0.2.15	3.69.104.152	MQTT	60	Publish Received (id=251)
384	46.581402138	3.69.104.152	10.0.2.15	MQTT	62	Publish Release (id=251)
385	46.582415544	10.0.2.15	3.69.104.152	MQTT	60	Publish Complete (id=251)
392	47.320540653	3.69.104.152	10.0.2.15	MQTT	164	Publish Message (id=301) [tp1_Dominguez_Etcheverry]
393	47.320953280	10.0.2.15	3.69.104.152	MQTT	60	Publish Received (id=301)
395	47.575538669	3.69.104.152	10.0.2.15	MQTT	62	Publish Release (id=301)
396	47.577634977	10.0.2.15	3.69.104.152	MQTT	60	Publish Complete (id=301)
398	48.329395530	3.69.104.152	10.0.2.15	MQTT	162	Publish Message (id=351) [tp1_Dominguez_Etcheverry]
399	48.329788835	10.0.2.15	3.69.104.152	MQTT	60	Publish Received (id=351)
401	48.616957561	3.69.104.152	10.0.2.15	MQTT	62	Publish Release (id=351)
402	48.618675993	10.0.2.15	3.69.104.152	MQTT	60	Publish Complete (id=351)

Figura 24: Captura de *Wireshark* del suscriptor con la funcionalidad *Checksum*.

No.	Time	Source	Destination	Protocol	Length	Info
38	6.440253816	10.0.2.15	3.69.104.152	MQTT	163	Publish Message (id=1) [tp1_Dominguez_Etcheverry]
40	6.443182614	10.0.2.15	3.69.104.152	MQTT	62	Connect Ack
42	6.712238259	3.69.104.152	10.0.2.15	MQTT	62	Publish Received (id=1)
44	6.755858786	3.69.104.152	10.0.2.15	MQTT	162	Publish Message (id=2) [tp1_Dominguez_Etcheverry]
46	7.453260822	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=2)
48	7.820859532	3.69.104.152	10.0.2.15	MQTT	169	Publish Message (id=3) [tp1_Dominguez_Etcheverry]
53	9.468343848	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=3)
55	9.760752304	3.69.104.152	10.0.2.15	MQTT	167	Publish Message (id=4) [tp1_Dominguez_Etcheverry]
57	10.483306422	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=4)
59	10.783477153	3.69.104.152	10.0.2.15	MQTT	174	Publish Message (id=5) [tp1_Dominguez_Etcheverry]
61	11.495148317	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=5)
63	11.809194332	3.69.104.152	10.0.2.15	MQTT	164	Publish Message (id=6) [tp1_Dominguez_Etcheverry]
65	12.511576530	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=6)
67	12.834121321	3.69.104.152	10.0.2.15	MQTT	162	Publish Message (id=7) [tp1_Dominguez_Etcheverry]
69	13.527725455	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=7)
73	13.965336654	3.69.104.152	10.0.2.15	MQTT	169	Publish Message (id=8) [tp1_Dominguez_Etcheverry]
75	14.559865030	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=8)
77	14.917774813	3.69.104.152	10.0.2.15	MQTT	162	Publish Message (id=9) [tp1_Dominguez_Etcheverry]
98	15.577784962	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=9)
103	15.917309803	3.69.104.152	10.0.2.15	MQTT	161	Publish Message (id=10) [tp1_Dominguez_Etcheverry]
107	16.689919788	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=10)
109	17.036590820	3.69.104.152	10.0.2.15	MQTT	141	Publish Message (id=11) [tp1_Dominguez_Etcheverry]
111	17.700438042	10.0.2.15	3.69.104.152	MQTT	62	Publish Received (id=11)
113	18.072439393	3.69.104.152	10.0.2.15	MQTT	58	Disconnect Req
123	18.726811658	10.0.2.15	3.69.104.152	MQTT		

Figura 25: Captura *Wireshark* del publicador con la funcionalidad *checksum*.

## Conclusiones

El trabajo realizado permitió explorar y aplicar el protocolo MQTT en la transmisión de datos entre un publicador y un suscriptor, haciendo uso de un broker externo. A través de la implementación de scripts en Python, provistos por la cátedra, se logró fragmentar y enviar mensajes de manera eficiente, garantizando la recepción y ensamblaje correcto de los mismos. Además, se estudiaron las características de la fragmentación en MQTT y su comparación con la segmentación en TCP, destacando similitudes y diferencias clave.

La incorporación de un sistema de verificación de integridad basado en el largo total de los datos permitió abordar la problemática de la pérdida de fragmentos, simulada para verificar el correcto funcionamiento del sistema. Aunque MQTT es un protocolo diseñado para mensajes breves, se logró implementar una funcionalidad que asegura la transmisión completa, incluso en escenarios de pérdida de paquetes, lo que mejora la confiabilidad del sistema.

El uso de MQTT en este proyecto, junto con el checksum, demostró ser una herramienta poderosa y eficiente para la comunicación entre dispositivos, con margen para mejorar la robustez mediante técnicas avanzadas de segmentación y validación de datos.