Universidad Rey Juan Carlos

Práctica 2: *SOA Patterns* *Service Messaging* y *Event-Driven Messaging*

Diseño y Mantenimiento de Servicios

Pablo González Troyano DNI: [redacted]
Grado en Ciencia, Gestión e Ingeniería de Servicios

Primer Cuatrimestre - Curso 2023-24

Índice

Índice de figuras

Índice de tablas

Este documento no contiene tablas.

Listings

1. Introducción, expectativas y temporización

1.1. Presentación

En el presente trabajo se aplicarán los conocimientos adquiridos hasta el momento en la asignatura de Diseño y Mantenimiento de Servicios, perteneciente al Grado en Ciencia, Gestión e Ingeniería de Servicios por la Universidad Rey Juan Carlos.

Durante todo el desarrollo de la práctica se incluirán capturas y recursos gráficos para demostrar la realización de las pruebas, así como para ejemplificar el proceso y facilitar la comprensión del desarrollo del trabajo.

1.2. *Abstract*

This work will apply the knowledge acquired so far in the subject of *Diseño y Mantenimiento de Servicios*, belonging to the Degree in Science, Management and Engineering of Services (Ciencia, Gestión e Ingeniería de Servicios) by the Universidad Rey Juan Carlos.

Throughout the development of the practice, screenshots and graphic resources will be included to demonstrate the realization of the tests, as well as to illustrate the process and facilitate the understanding of the development of the work.

1.3. Temporización y organización

Tal y como se recoge en la Guía de Estudio de la asignatura, la práctica se engloba dentro del tema 2, *Patrones de Diseño*. Este tema, según la cronología de la citada Guía, se desarrollará entre los días 04/10/2023 y 31/10/2023. (URJC, 2023) Sí cabe mencionar que la fecha límite de entrega de la presente práctica se ha fijado para el día 18/11/2023 en el Aula Virtual.

El trabajo se desarrollará de forma individual.

1.4. Expectativas

Las expectativas de aprendizaje de la presente práctica se basan en la comprensión de los patrones SOA en su conjunto. Con especial hincapié en la familia de patrones y el patrón elegido.

En mi caso, para la realización de la presente práctica he elegido la familia de patrones ***Service Messaging***. Para la explicación detallada se ha elegido el patrón ***Event-Driven Messaging***.

2. Desarrollo de la práctica

2.1. ¿Qué es SOA? ¿Qué son los patrones SOA?

La arquitectura orientada a servicios (del inglés, *service-oriented architecture*, SOA) es un modelo arquitectónico que pretende mejorar la agilidad y la rentabilidad de un negocio, reduciendo al mismo tiempo la carga de las TI sobre el conjunto de la organización. Lo consigue situando los servicios como el medio principal a través del cual se representa la lógica de la solución. La SOA apoya la orientación a servicios en la realización de los objetivos estratégicos asociados a la informática orientada a servicios. (Erl, 2009)

Un servicio es una unidad de lógica del negocio a la que se ha aplicado la orientación a servicios en una medida significativa. Es la aplicación de los principios de diseño de la orientación a servicios lo que distingue a una unidad de lógica como servicio, en comparación con las unidades de lógica que pueden existir únicamente como objetos o componentes. (Erl, 2009)

Cualquier negocio, empresa u organización que se diga *orientada a servicios* debe cumplir con las siguientes cuatro características de SOA: (Erl, 2009)

1. ***Business-Driven***: la arquitectura tecnológica se alinea con la arquitectura empresarial actual. Este contexto se mantiene constantemente para que la arquitectura tecnológica evolucione a la par que la empresa.
2. ***Vendor neutral***: el modelo arquitectónico no se basa únicamente en una plataforma de proveedor patentada, lo que permite combinar o sustituir tecnologías de diferentes proveedores con el tiempo para maximizar el cumplimiento de los requisitos empresariales de forma continua.
3. ***Enterprise-Centric***: el alcance de la arquitectura representa un segmento significativo de la empresa, lo que permite la reutilización y composición de servicios y posibilita que las soluciones orientadas a servicios abarquen los silos de aplicaciones tradicionales.
4. ***Composition-Centric***: la arquitectura admite intrínsecamente la mecánica de agregación repetida de servicios, lo que le permite adaptarse a los cambios constantes mediante el ensamblaje ágil de composiciones de servicios.

2.2. Familia de Patrones SOA: *Service Messaging*

Hoy en día las implementaciones de servicios dependen de la interacción entre muchos actores, tanto internos como internos. De hecho, toda la industria (tanto a nivel de TI, como a nivel de *concepto* de servicios) está migrando a *microservicios*, tal y como atestiguan las estadísticas de Google sobre el interés en la búsqueda: (Trends, 2023)

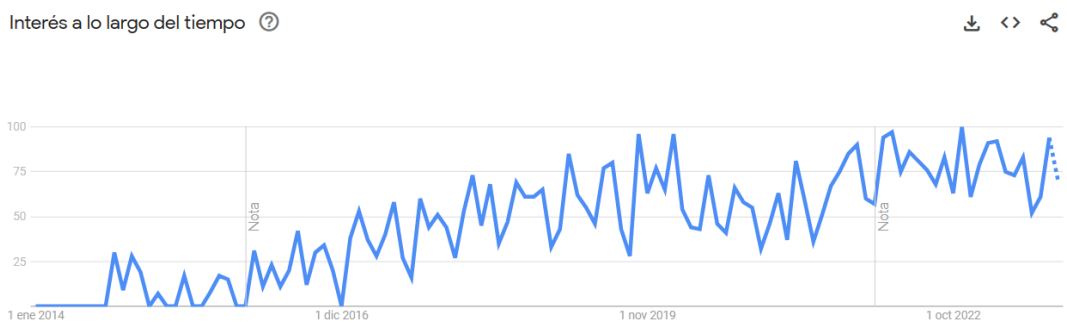


Figura 1: Gráfico de interés en la búsqueda de Google para el término *microservicios*

Todos estos microservicios y actores han de poder comunicarse entre ellos, para alcanzar el objetivo común. **Previamente a la implantación** generalizada de los patrones SOA de esta familia, era necesario mantener canales de comunicación seguros entre todos y cada uno de los actores que intervenían. O, en su defecto, inicializarlas cada vez que se quería entregar un mensaje entre dos entidades. Se solían implementar soluciones basadas en tecnología RPC, protocolos binarios (en muchas ocasiones privativos y de un vendedor concreto, lo que limitaba el margen de innovación, mejora e interoperabilidad).

Esto era, claramente, un problema, pues generar, mantener y terminar dichas conexiones, suponía mayor costo en términos computacionales y de tiempo (por tanto, dinero para la organización). A su vez, añadía puntos de falla y lógicas de negocio adicionales. Por ejemplo, ¿qué hacer si en el momento en el que se genera el mensaje para el servicio B, este no está disponible? ¿Debe descartarse con la consiguiente pérdida de información? ¿Debe encargarse el servicio emisor de reintentar la entrega, con el consiguiente consumo de recursos que esto implica?

Como solución a la casuística descrita, se propone la utilización de *mensajes*. Estos *mensajes* no requieren que existan conexiones persistentes entre los actores implicados. En lugar de eso, los *mensajes* son transmitidos como unidades independientes de información, utilizando una estructura de mensajería especializada, y la propia infraestructura subyacente.

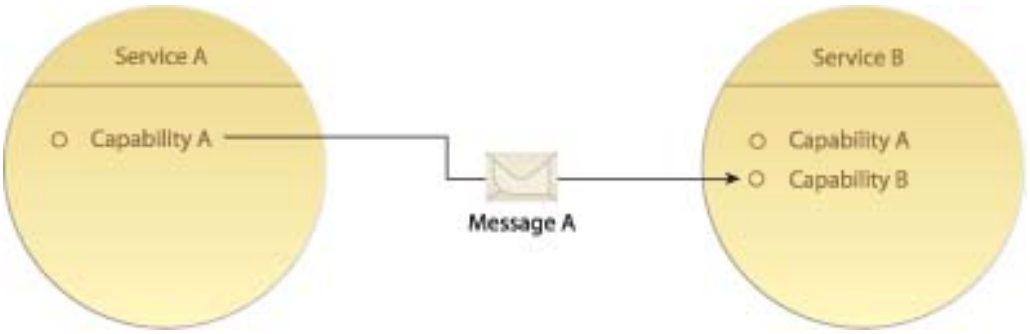


Figura 2: Servicios interactúan a través de mensajes. Extraída del libro *SOA Design Patterns*

Estos mensajes son gestionados por una aplicación especializada (en ocasiones denominada *messages broker*) que se encarga: (IBM, 2023b)

- Garantizar la entrega de los mensajes, así como la notificación en el caso de fallo. El estándar MQTT (AWS, 2023a) define varios niveles de calidad del servicio entre *casi una vez* y *solo una vea*. La elección dependerá de la criticidad del servicio en el que se esté trabajando. (Jackson et al., 2021)
- Securizar el contenido del mensaje durante el transporte del mismo.
- Gestionar estado y contexto del mensaje. (IBM, 2019)
- Transmitir los mensajes de forma eficiente, como parte de las interacciones en tiempo real.

Si bien es *uno de los patrones de diseño más importantes*, (Erl, 2009) tal y como ocurre con el resto de patrones estudiados, suelen (y deben) ser implementados de forma conjunta con otros patrones. De esta forma, Erl destaca los patrones de *Canonical Protocol* y *Canonical Schema* en lo relativo a la interoperabilidad.

La familia de patrones *Service Messaging* establece las guías básicas para muchos y más especializados patrones de mensajes. En la siguiente imagen se puede observar cómo interactúan entre ellas.

Figura 3: Patrones relacionados. Adaptación desde (Erl, 2009)

2.3. Patrón SOA de la familia: *Event-Driven Messaging*

La información aquí descrita tiene como fuente, en su mayoría, el libro de Thomas Erl, *SOA Design Patterns*. El resto de fuentes son citadas como corresponde.

¿Cómo pueden los consumidores de servicios ser notificados de forma automática cuando ocurren eventos de su interés?

2.3.1. Problema

En entornos tradicionales, los consumidores de servicios han de elegir entre comunicación en un solo sentido y pregunta-respuesta, ambos originándose en el consumidor. Con este planteamiento, el consumidor tiene que *consultar* continuamente al tercer servicio para saber si ha ocurrido algún evento de su interés; y, de ocurrir, recuperar los detalles.

Este modelo es ineficiente, puesto que genera numerosas peticiones innecesarias (que implican costes económicos y posibles puntos de fallo). A su vez, pueden producir retrasos debido a que el consumidor solo consulta al servicio generador en intervalos de tiempo fijados.

En el esquema siguiente, cuando el servicio A (actuando como consumidor de servicio, *service consumer*) «consulta» (en inglés, *to poll*) en intervalos de una hora, sobre un evento en el que el Servicio A está interesado. Cada ciclo de consulta implica un intercambio síncrono de pregunta-respuesta. Después de la cuarta hora, el Servicio A recibe la información.

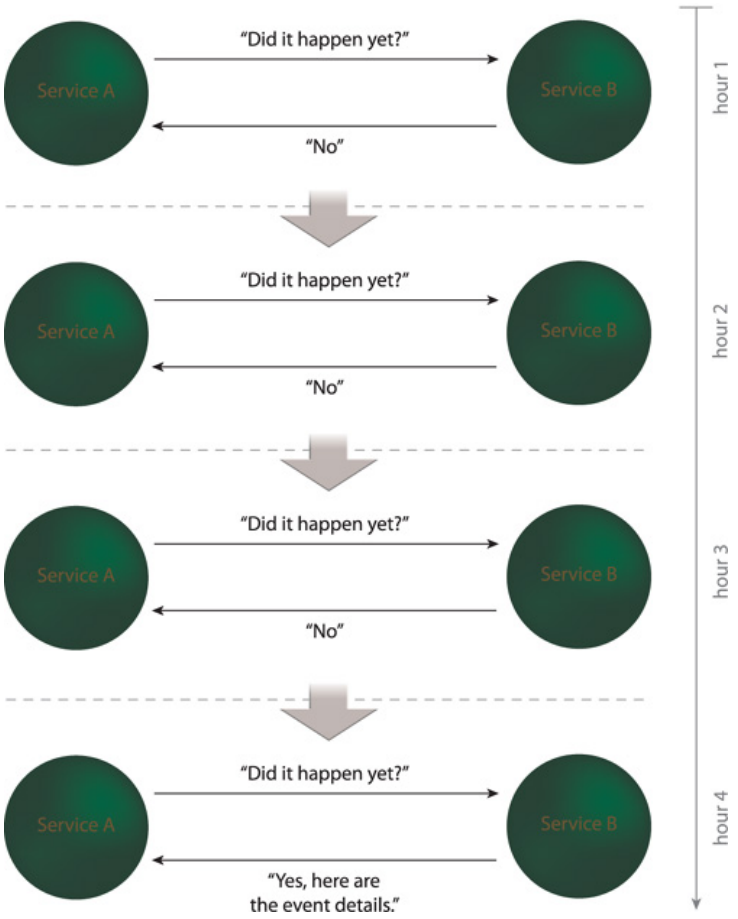


Figura 4: Consulta en intervalos de 1 hora.

2.3.2. Solución

La solución propuesta consiste en la implementación de un programa/sistema gestor de eventos, de esta forma, el **servicio consumidor** se configura como **suscriptor** de eventos asociados con un servicio que **produce eventos** que asume el rol de ***publicador***. (RedHat, 2023) (KIBO, 2019) Pueden existir diferentes tipos de eventos (que el programa/sistema gestor de eventos hace disponibles), y los consumidores pueden elegir a cuáles de esos eventos se suscriben.

Cuando esos eventos ocurren, el servicio (actuando como **publicador**) automáticamente envía los detalles del evento al sistema gestor de eventos, que luego lo retransmite a los suscriptores registrados para ese tipo de evento, tan pronto como sea posible.

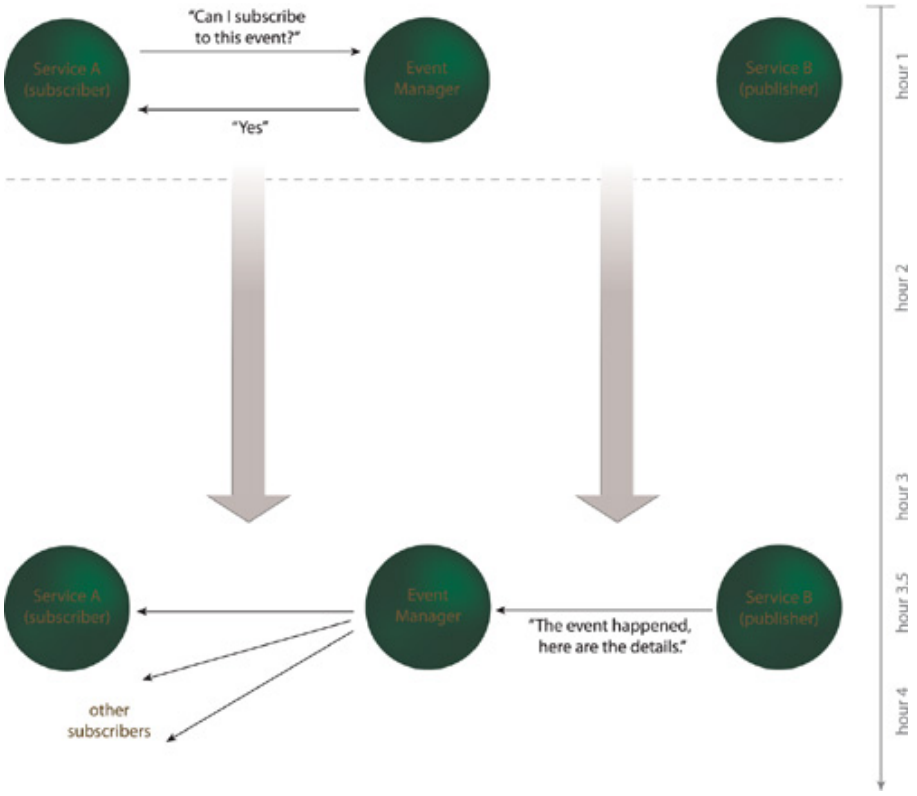


Figura 5: Proceso Pub/Sub. Extraída de (Erl, 2009)

Esta solución propuesta está estrechamente relacionada con el concepto de *Event-Driven Arquitecture*, EDA. (IBM, 2023a) (AWS, 2023b)

2.3.3. Aplicación

Una estructura de mensajes es implementada como una extensión del inventario. Son muchas las plataformas que ofrecen las capacidades necesarias: trazabilidad, procesamiento complejo de eventos, filtrado y correlación. Entre estas aplicaciones está, por ejemplo, RabbitMQ. (RabbitMQ, 2023d)

2.3.4. Impacto

Event-Driven Messaging se basa en intercambios de mensajes de forma asíncrona que ocurren esporádicamente, dependiendo de cuando se produce el evento (en el lado del servicio). Puesto que envíos de eventos no se pueden predecir con facilidad, algún consumidor siempre ha de estar disponible para poder recibir, y procesar, los mensajes. Estos mensajes son enviados típicamente en una comunicación de un

solo sentido y no suelen requerir *ACK* (confirmación de recepción) por parte del destinatario; aunque este requisito podría implementarse.

Estas características hacen que puedan surgir problemas de desempeño y fiabilidad. Para solucionar dichos problemas se suele recurrir, entre otras medidas, a la implementación de medidas como *Asynchronous Queuing* y *Reliable Messaging*. (Erl, 2009)

2.3.5. Relaciones

Como ya se ha visto en la sección anterior, 2.3.4, el patrón *Event-Driven Messaging* se suele implementar de forma conjunta con otros patrones, que extienden y aseguran la funcionalidad. Entre estos patrones relacionados encontramos *Asynchronous Queuing* y *Reliable Messaging*.

El modelo *Pub/Sub* provee funcionalidades asíncronas avanzadas que aprovechan la lógica y capacidades de enrutado de forma nativa por los ESB. (Jansen & Saladas, 2020)

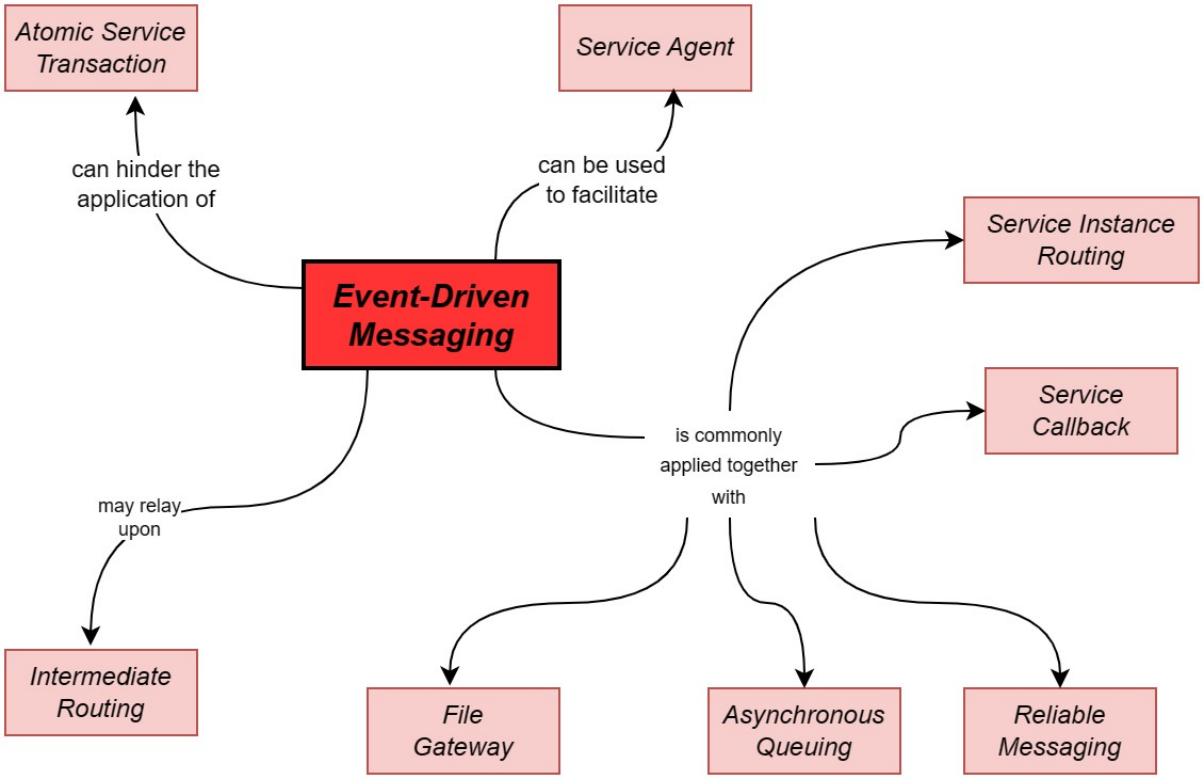


Figura 6: Relaciones entre patrones. Adaptación desde (Erl, 2009)

2.4. Extra: Demostración con RabbitMQ

En esta sección se realizarán pruebas con el software *RabbitMQ*, que es el *software* de mensajería *open source* más desplegado. (RabbitMQ, 2023e)

El contenido de esta sección está extraído parcialmente de: (RabbitMQ, 2023c), (RabbitMQ, 2023b) y (RabbitMQ, 2023f)

2.4.1. Instalación del contenedor

Tal y como se indica en la guía oficial, lo mejor para probar la solución es desplegar el contenedor Docker. (RabbitMQ, 2023a)
Se puede usar el siguiente comando, habiendo instalado previamente el sistema Docker siguiendo las instrucciones de la documentación oficial. (Docs, 2023)

```
1 docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management
```

Listing 1: Ejecutar contenedor Docker de RabbitMQ

Al ejecutar el comando, comenzará la descarga de las *layers*/capas que conforman el contenedor:

```
PS C:\Users\pablo> docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.12-management
Unable to find image 'rabbitmq:3.12-management' locally
3.12-management: Pulling from library/rabbitmq
aece8493d397: Pull complete
95a7e9731f67: Pull complete
3d5bb659a487: Pull complete
bfe39921a0ad: Pull complete
3a222b0c42b1: Pull complete
af78870756d5: Pull complete
7dd1547e5e0a: Pull complete
9e00ac9a1fa5: Pull complete
0fdde25f2038: Pull complete
3f80ae364b06: Pull complete
f8161bf23459: Pull complete
8b4ff40ac6d0: Pull complete
Digest: sha256:5b2785170790631f20ab091b733cfceeb56df765c0be5eb8f87a80d5f03d1afc
Status: Downloaded newer image for rabbitmq:3.12-management
```

Figura 7: Descarga del contenedor

Tras unos segundos, veremos un mensaje similar al siguiente por pantalla, indicándonos que ya se ha descargado el contenedor y que este está ejecutándose:

```
2023-10-24 10:42:51.399894+00:00 [info] <0.601.0> Server startup complete; 4 plugins started.
2023-10-24 10:42:51.399894+00:00 [info] <0.601.0>  * rabbitmq_prometheus
2023-10-24 10:42:51.399894+00:00 [info] <0.601.0>  * rabbitmq_management
2023-10-24 10:42:51.399894+00:00 [info] <0.601.0>  * rabbitmq_management_agent
2023-10-24 10:42:51.399894+00:00 [info] <0.601.0>  * rabbitmq_web_dispatch
2023-10-24 10:42:51.453672+00:00 [info] <0.9.0> Time to start RabbitMQ: 9833944 us
```

Figura 8: *Logs* del contenedor ejecutándose

Si en este momento accedemos a la URL `localhost:15672` veremos el panel de administrador de RabbitMQ.
El usuario y la contraseña por defecto es `guest` en ambos casos.

2.4.2. Paquete Python necesario

Desarrollaremos el ejemplo utilizando Python, por ser un lenguaje de programación sencillo y ampliamente extendido.

Para poder trabajar con RabbitMQ desde Python es necesario instalar el paquete `pika`. La instalación se puede hacer ejecutando el siguiente comando, teniendo el gestor de paquetes `pip` instalado:

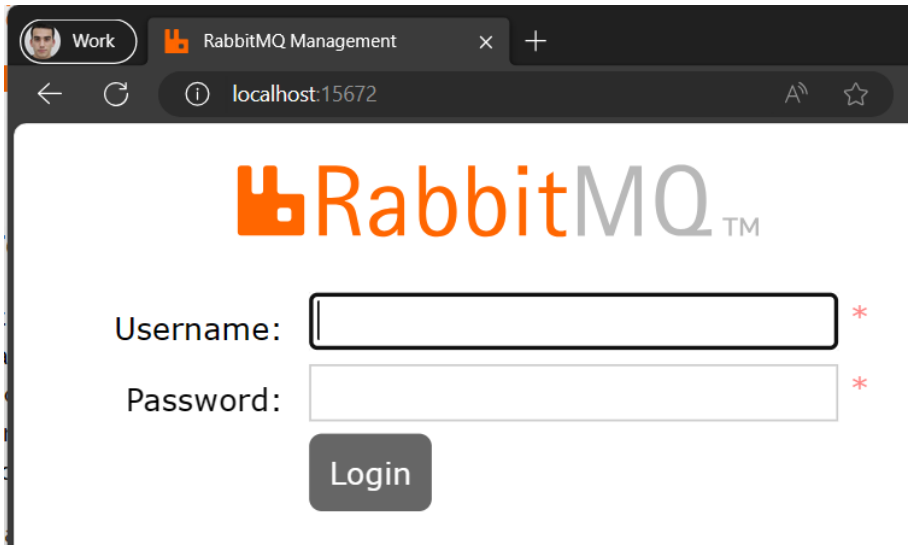


Figura 9: Panel de administración web de RabbitMQ

```
python -m pip install pika --upgrade
```

Listing 2: Instalación de *pika* con *pip*

2.4.3. Enviar mensaje a la cola

Con el siguiente código se puede mandar un mensaje a la cola. En los comentarios se explica qué hace cada instrucción.

```
#!/usr/bin/env python

# Importar el paquete pika, necesario para trabajar con el protocolo de RabbitMQ
import pika

# Conectarse a la instancia de RabbitMQ, corriendo en nuestro equipo (localhost)
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Antes de enviar un mensaje tenemos que asegurarnos de que la cola exista.

# Creamos una cola:
channel.queue_declare(queue='prueba')

# Para enviar un mensaje con el contenido "¡Hola Mundo!" a la cola "prueba" ejecutamos:
channel.basic_publish(exchange='',
                      routing_key='prueba',
                      body='¡Hola Mundo!')

# Para que aparezca la confirmación por pantalla:
print(" [x] Sent 'Hello World!'")

# Cerramos la conexión
connection.close()
```

Listing 3: send.py - Código para enviar un mensaje a la cola

Tal y como se esperaba, la cola (*queue*) *prueba* se ha creado correctamente. En este momento hay un mensaje encolado:

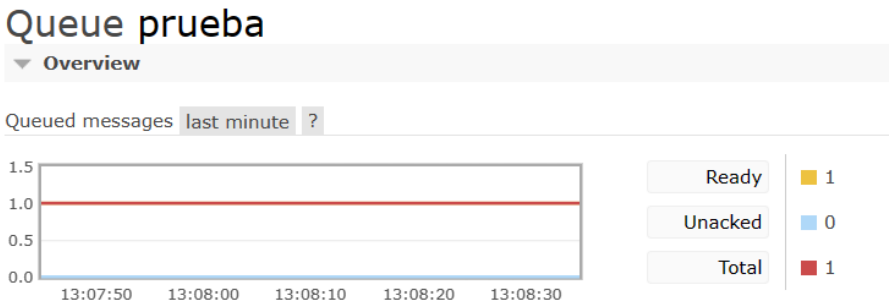


Figura 10: Captura de pantalla con un mensaje encolado

Y, efectivamente, vemos pendiente el mensaje que hemos enviado desde el *script* en Python anterior:

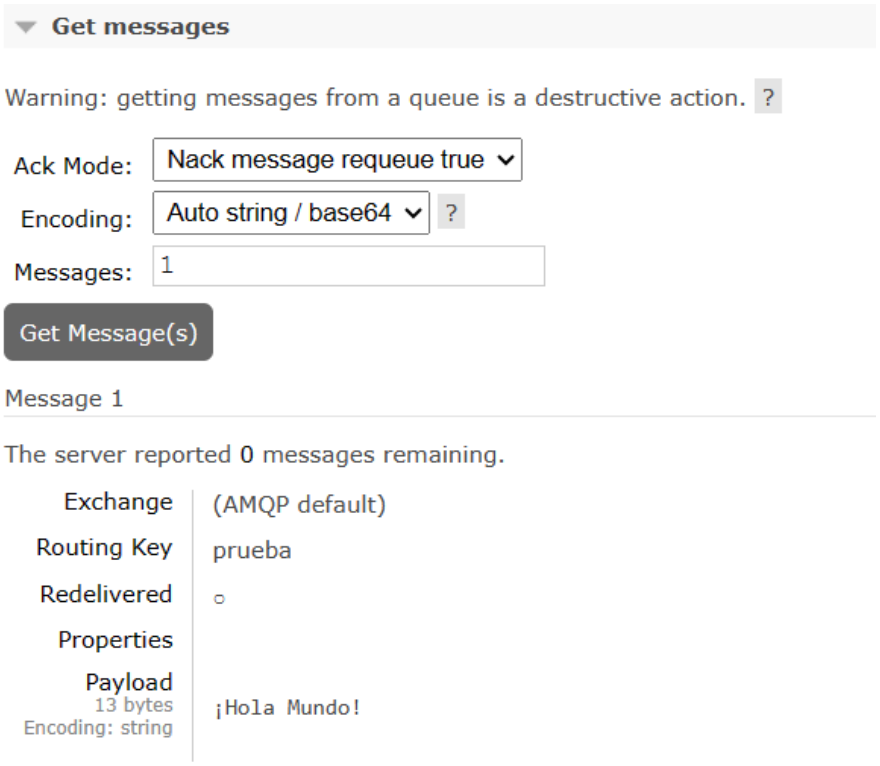


Figura 11: Captura de pantalla con los detalles del mensaje encolado

2.4.4. Consumir mensajes

Con el siguiente código se pueden consumir los mensajes de la cola. En los comentarios se explica qué hace cada instrucción.

```
#!/usr/bin/env python

# Importar el paquete pika, necesario para trabajar con el protocolo de RabbitMQ
import pika

# Conectarse a la instancia de RabbitMQ, corriendo en nuestro equipo (localhost)
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

```

```
10
11 # Definimos la función que se ejecutará con cada mensaje:
12 def callback(ch, method, properties, body):
13     print(f" [x] Received {body}")
14
15
16 # Indicamos que queremos CONSUMIR la cola "prueba", y para cada mensaje se ejecutará la
       función "callback", definida anteriormente.
17 channel.basic_consume(queue='prueba',
18                       auto_ack=True,
19                       on_message_callback=callback)
20
21 # Printar mensaje informativo por pantalla.
22 print(' [*] Waiting for messages. To exit press CTRL+C')
23
24 # Empezar a consumir la cola.
25 channel.start_consuming()
```

Listing 4: receive.py - Código para consumir mensajes de la cola

Al ejecutarlo, puesto que hemos enviado previamente a la cola un mensaje, lo vemos en la terminal:

```
C:\Python312\python.exe C:\Users\pablo\PycharmProjects\rabbitmq\receive.py
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received b'\xc2\xa1Hola Mundo!'
```

Figura 12: Recepción de un mensaje de la cola

2.4.5. Prueba: envío y consumo *masivo*

Modificando el código del archivo "send.py" de la siguiente manera, podemos enviar más de un mensaje:

```
1                       for i in range(1,101,1):
2                           cadena = "Hola Mundo! - " + str(i)
3                           channel.basic_publish(exchange='',
4                                                 routing_key='prueba',
5                                                 body=str(cadena))
```

Listing 5: Enviar varios mensajes a la cola con un bucle for.

En la terminal de recepción aparecerán los 100 mensajes enviados:

```
receive ×                                   send ×
[x] Received b'Hola Mundo! - 91'            C:\Python312\python.exe C:\Users\pablo\PycharmProjects\rabbitmq\send.py
[x] Received b'Hola Mundo! - 92'             [x] Sent 'Hello World!'
[x] Received b'Hola Mundo! - 93'
[x] Received b'Hola Mundo! - 94'            Process finished with exit code 0
[x] Received b'Hola Mundo! - 95'
[x] Received b'Hola Mundo! - 96'
[x] Received b'Hola Mundo! - 97'
[x] Received b'Hola Mundo! - 98'
[x] Received b'Hola Mundo! - 99'
[x] Received b'Hola Mundo! - 100'
```

Figura 13: Terminal de recepción con los 100 mensajes recibidos

Si paramos el consumidor de RabbitMQ, empezarán a acumularse los mensajes encolados en RabbitMQ. Esto es una ventaja, pues en caso de que no haya ningún consumidor disponible (por cualquier problema que se haya producido en la infraestructura), cabe la posibilidad de que los mensajes sean consumidos en cuanto haya uno disponible:

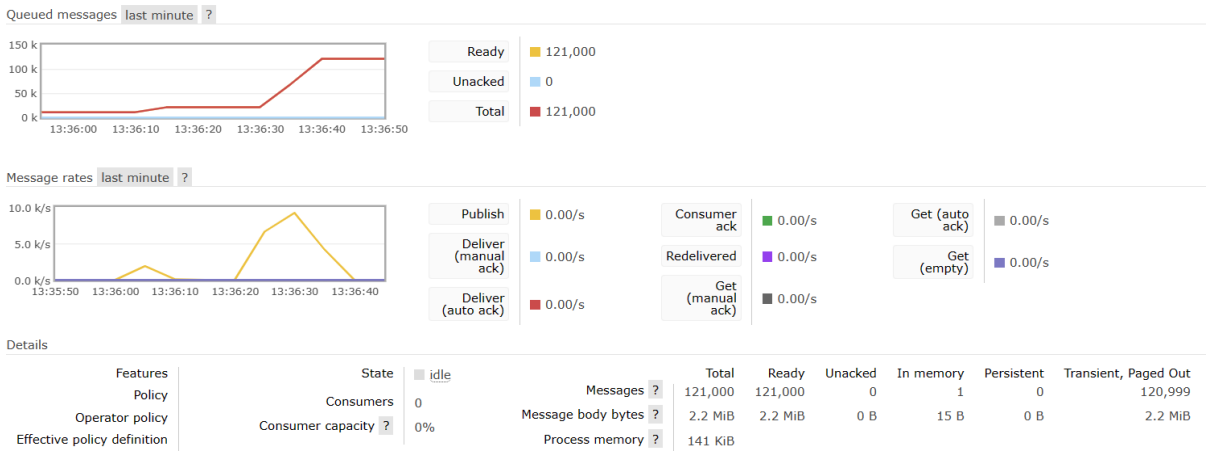


Figura 14: Mensajes acumulados en RabbitMQ. 121.000 mensajes.

Al iniciar de nuevo el consumidor, este se conectará a la cola y comenzará a consumir los 121.000 mensajes pendientes. En cuestión de unos pocos segundos ha consumido todos los mensajes.

2.4.6. Prueba: Añadir tiempo de procesamiento

Sin embargo, la prueba anterior no era realista, pues normalmente cada mensaje requiere un procesamiento. Podemos *pausar* el script para cada mensaje para ver cómo se van acumulando, modificando la función que lee los mensajes:

```
def callback(ch, method, properties, body):
    print("Iniciando procesamiento")
    time.sleep(0.5) # AQUÍ PAUSAMOS
    print(f" [x] Received {body}")
```

Listing 6: Añadir pausa entre lectura de mensajes.

2.4.7. Ejemplo: Colas de trabajo

En este ejemplo se generará una cola que será consumida por varios "*workers*", realizando una serie de tareas sobre los mensajes que reciban. El total de tareas serán distribuidas entre todos los nodos *workers*.

Tal y como se puede ver en la siguiente pantalla:

- Los *nodos worker* se encuentran en las dos terminales de la izquierda, señalados en amarillo.
- Las tareas son enviadas desde la terminal de la derecha, señalada en rojo.

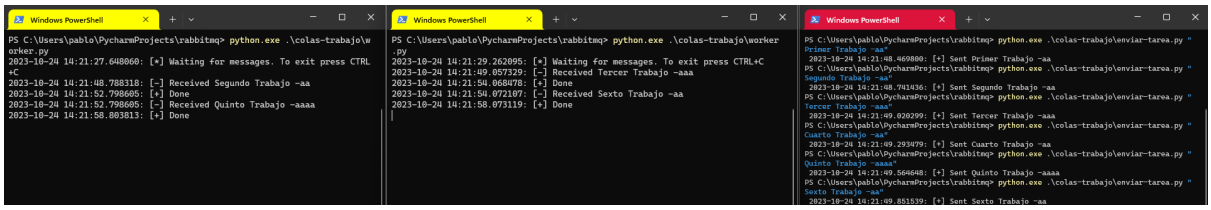


Figura 15: Los nodos 'worker' se han distribuido las tareas en Round-Robin

El código usado para este ejemplo:

```python
#!/usr/bin/env python

# Importar el paquete pika, necesario para trabajar con el protocolo de RabbitMQ
import pika

# También se usarán los paquetes 'sys' y 'datetime', para  diversas funciones que se explican en el código.
import sys
import datetime


# Conectarse a la instancia de RabbitMQ, corriendo en nuestro equipo (localhost)
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

# Declarar que se usará como canal la cola "prueba2"
channel.queue_declare(queue='prueba2', durable=True)

# El mensaje que se enviará a la cola será el primer valor pasado por parámetros al llamar al script, o, en caso de que no se haya pasado ningún valor, "Hola Mundo!"
message = ' '.join(sys.argv[1:]) or "Hola Mundo!"

channel.basic_publish(
    exchange='',

    # Se usa la cola "prueba2"
    routing_key='prueba2',

    # Se envía el valor de la variable "message" como cuerpo del mensaje.
    body=message,

    # Activar el modo persistente de entrega, para que RabbitMQ espere a la confirmación del worker para marcar como entregado el mensaje.
    properties=pika.BasicProperties(
        delivery_mode=pika.spec.PERSISTENT_DELIVERY_MODE
    ))

```

```
# Registrar la hora y la confirmación de que se ha enviado el mensaje.
print(f" {datetime.datetime.now()}: [+] Sent {message}")

# Cerrar la conexión.
connection.close()
```

Listing 7: enviar-tarea.py - Código para enviar mensajes/tareas a la cola

```
import pika
import time
import datetime

connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='prueba2', durable=True)
print(f'{datetime.datetime.now()}: [*] Waiting for messages. To exit press CTRL+C')


def callback(ch, method, properties, body):

    # Printar por pantalla el contenido del mensaje que ha recibido. TODAVÍA NO se ha marcado como entregado.
    print(f"{datetime.datetime.now()}: [-] Received {body.decode()}")

    # Dormirá el número de veces que el caracter "b" aparezca en la cadena. Es un ejemplo algo "inútil", pero sirve como diferenciador de dificultad.
    time.sleep(body.count(b'a'))

    # Printar por pantalla que ya se ha realizado el trabajo, una vez ha pasado el tiempo "dormido" correspondiente.
    print(f"{datetime.datetime.now()}: [+] Done")

    # Por si falla, no decimos a RabbitMQ que marque como entregado el mensaje hasta que no lo hemos terminado de procesar. De esta forma, si el 'worker' tuviera algún problema y fallara, RabbitMQ volvería a entregar el mensaje/tarea a otro worker que sí estuviera disponible.
    ch.basic_ack(delivery_tag=method.delivery_tag)


# Coger mensajes de uno en uno
channel.basic_qos(prefetch_count=1)

# Empezar a consumir la cola "prueba2"
channel.basic_consume(queue='prueba2', on_message_callback=callback)

channel.start_consuming()
```

Listing 8: worker.py - Código para procesar los mensajes recibidos en Round-Robin

3. Conclusiones

Para concluir el presente trabajo podemos afirmar que la implementación de un buen sistema de mensajes es clave para el correcto funcionamiento del servicio ofrecido. Tanto desde el punto de vista tecnológico, para poder adecuar los recursos de procesamiento a la carga de trabajo real, como al más importante de todos: la percepción de la calidad del servicio por parte de los clientes finales.

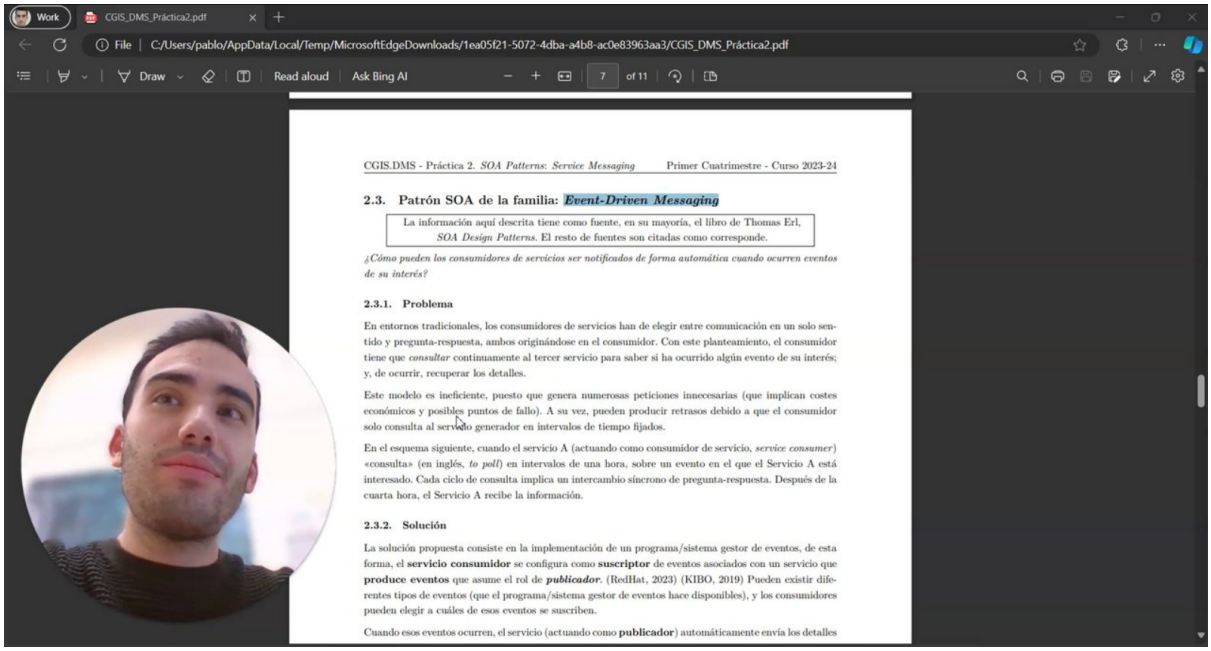
Esta calidad percibida está directamente relacionada con el funcionamiento del sistema en sí. Una comunicación ágil entre actores del sistema permite reducir ineficacias, así como adaptarse a cambios en la carga de trabajo.

En concreto, el patrón *Event-Driven Messaging* permite que el sistema responda a eventos de forma sencilla, reduciendo peticiones innecesarias (con el consiguiente coste y posibilidad de falla), retrasos y el mantenimiento de conexiones persistentes que consumen recursos de los sistemas. Además, facilita la escalabilidad y la adaptabilidad al cambio.

En tanto al posible margen de mejora, siempre es posible aumentar la calidad de una entrega. Por ejemplo, se podrían haber buscado ejemplos de casos de uso de empresas y organizaciones que ya estuvieran implementando esta clase de patrones.

4. Acceso al vídeo en YouTube. Recursos disponibles

El vídeo está disponible en el siguiente enlace: **https://www.youtube.com/watch?v=cIR1W32H6No**4



A su vez, está disponible:

- Para el ejemplo básico de RabbitMQ:
  - El script en Python para enviar mensajes, `send.py`: en la sección 2.4.3 y en GitHub[1].
  - El script en Python para procesar/recibir los mensajes, `receive.py`: en la sección 2.4.4 y en GitHub[2].
- Para el ejemplo avanzado con colas de trabajo:
  - El script para añadir mensajes/tareas a la cola: en la sección 2.4.7 y en GitHub[3].
  - El script «worker» para procesar las tareas: en la sección 2.4.7 y en GitHub[4].

[1] https://github.com/gonzaleztroyano/CGIS.DMS/blob/main/práctica2-rabbitmq/send.py
[2] https://github.com/gonzaleztroyano/CGIS.DMS/blob/main/práctica2-rabbitmq/receive.py
[3] https://github.com/gonzaleztroyano/CGIS.DMS/blob/main/práctica2-rabbitmq/colas-trabajo/enviar-tarea.py
[4] https://github.com/gonzaleztroyano/CGIS.DMS/blob/main/práctica2-rabbitmq/colas-trabajo/worker.py

5. Bibliografía

AWS. (2023a). ¿Qué es el MQTT? - Explicación del protocolo MQTT - AWS. Consultado el 22 de octubre de 2023, desde https://aws.amazon.com/es/what-is/mqtt/

AWS. (2023b). ¿Qué es la EDA? - Explicación sobre la arquitectura basada en eventos - AWS. Consultado el 19 de octubre de 2023, desde https://aws.amazon.com/es/what-is/eda/

Docs, D. (2023). Installing Docker on Windows. Consultado el 24 de octubre de 2023, desde https://docs.docker.com/desktop/install/windows-install/

Erl, T. (2009). *SOA design patterns* (1st ed) [OCLC: ocn156832597]. Prentice Hall.
Includes indexes.

IBM. (2019, agosto). What is Messaging? Consultado el 19 de octubre de 2023, desde https://developer.ibm.com/articles/what-is-messaging/

IBM. (2023a, julio). Architectural considerations for event-driven microservices-based systems. Consultado el 19 de octubre de 2023, desde https://developer.ibm.com/articles/eda-and-microservices-architecture-best-practices/

IBM. (2023b). What are Message Brokers? | IBM. Consultado el 19 de octubre de 2023, desde https://www.ibm.com/topics/message-brokers

Jackson, C., Stanley, K., & Lane, D. (2021, abril). Why enterprise messaging and event streaming are different. Consultado el 19 de octubre de 2023, desde https://developer.ibm.com/articles/difference-between-events-and-messages/

Jansen, G., & Saladas, J. (2020, julio). Advantages of the event-driven architecture pattern. Consultado el 19 de octubre de 2023, desde https://developer.ibm.com/articles/advantages-of-an-event-driven-architecture/

KIBO. (2019, diciembre). ¿Qué es una arquitectura event-driven? [Section: Arquitecturas Avanzadas]. Consultado el 19 de octubre de 2023, desde https://www.bbvanexttechnologies.com/blogs/que-es-una-arquitectura-event-driven/

RabbitMQ. (2023a). Downloading and Installing RabbitMQ — RabbitMQ. Consultado el 21 de octubre de 2023, desde https://www.rabbitmq.com/download.html

RabbitMQ. (2023b). RabbitMQ tutorial - "Hello world!"— RabbitMQ. Consultado el 21 de octubre de 2023, desde https://www.rabbitmq.com/tutorials/tutorial-one-python.html

RabbitMQ. (2023c). RabbitMQ Tutorials — RabbitMQ. Consultado el 21 de octubre de 2023, desde https://www.rabbitmq.com/getstarted.html

RabbitMQ. (2023d). RabbitMQ: easy to use, flexible messaging and streaming — RabbitMQ. Consultado el 21 de octubre de 2023, desde https://www.rabbitmq.com/

RabbitMQ. (2023e). RabbitMQ: Funcionalidades. Consultado el 24 de octubre de 2023, desde https://www.rabbitmq.com/#features

RabbitMQ. (2023f). Round-Robin on RabbitMQ. Consultado el 24 de octubre de 2023, desde https://www.rabbitmq.com/tutorials/tutorial-two-python.html

RedHat. (2023, agosto). ¿Qué es la automatización basada en eventos? Consultado el 19 de octubre de 2023, desde https://www.redhat.com/es/topics/automation/what-is-event-driven-automation

Trends, G. (2023). Google Trends - Estadísticas para 'microservicios'. https://trends.google.com/trends/explore?date=2014-01-01%202023-10-22&geo=ES&q=microservicios&hl=es

URJC. (2023). Guía de Estudio de la asignatura Diseño y Mantenimiento de Servicios.