INCORPORACIÓN DE RABBITMQ EN ARQUITECTURA DISTRIBUIDA CON TRAEFIK

Partiendo de la arquitectura un proyecto anterior (clientes que envían solicitudes a un servicio de analíticas, y un panel que visualiza los resultados), se desea mejorar la escalabilidad y robustez del sistema incorporando RabbitMQ como sistema de comunicación entre servicios.

1. Revisión conceptual:

1.1 ¿Qué es RabbitMQ y cuál es su función en una arquitectura distribuida?

RabbitMQ es un ***broker de mensajes***; un broker de mensajes (o message broker) es un intermediario que gestiona la transmisión de mensajes entre servicios o aplicaciones. Su propósito principal es ***desacoplar el emisor del receptor***, permitiendo que cada uno funcione de forma ***independiente*** y ***asincrónica.***

En una arquitectura distribuida, donde distintos servicios corren en contenedores separados, RabbitMQ permite que un servicio envíe ***un mensaje (evento)*** y que otro lo reciba más tarde, aunque no estén disponibles al mismo tiempo.

Con RabbitMQ el transporte de datos se hace por medio de ***mensajes*** (JSON, texto plano, binario, etc.)y no a través de solicitudes HTTP (REST, GraphQL, etc.) como lo hace Traefik. Además, RabbitMQ cuenta con persistencia de datos, en contraste con Traefik que solo enruta y no guarda datos.

Los escenarios comunes de RabbitMQ son la ***analítica***, las ***colas de trabajo***, los ***eventos*** y las ***notificaciones***. Las aplicaciones típicas de Traefik son el balanceo de carga, la redirección, la seguridad y la autenticación.

Los protocolos usados por RabbitMQ son: AMQP, MQTT, STOMP

1.2 ¿Qué ventajas ofrece frente a llamadas HTTP directas entre servicios?

- ✓ **Desacoplamiento:** El productor y consumidor no tienen que conocerse, en cambio con llamadas HTTP, las llamadas directas requieren conocer el endpoint.
- ✓ **Tolerancia a fallos:** Los mensajes se pueden encolar si el consumidor cae o no está disponible, en cuanto a las llamadas HTTP, si el servicio destino está caído, el sistema no puede procesar las solicitudes y lanza un error.
- ✓ **Escalabilidad:** Se pueden tener múltiples consumidores fácilmente, mientras que con las llamadas HTTP, escalar requiere balanceadores adicionales.
- ✓ **Entrega asíncrona:** El productor no espera respuesta inmediata, pero con las llamadas HTT, se bloquea hasta recibir respuesta o error.
- ✓ **Flexibilidad:** Se pueden enrutar mensajes a distintas colas, mientras que las llamadas HTTP son punto a punto.

1.3 ¿Qué son las colas, exchanges, publishers y consumers?

- ✓ **Publisher (Sender):** Es el emisor del mensaje. Publica información en un exchange de RabbitMQ para que otros servicios puedan procesarla. Ejemplo: servicio-cliente-uno.

- ✓ **Consumer (Receiver):** Es quien recibe y ***procesa*** el mensaje. Escucha una cola específica y procesa los mensajes que llegan. Ejemplo: servicio-analíticas.

- ✓ **Queue (Cola):** Es donde se almacenan temporalmente los mensajes hasta que un consumer los lea.

- ✓ **Exchange:** Es quien ***recibe los mensajes del publisher y decide a qué cola(s) enviarlos***, según ciertas reglas (binding).

Ejemplo visual simple:

Publisher → [Exchange] → [Queue] → Consumer

2. Análisis del sistema actual: Identificar en la arquitectura del parcial actual:

2.1 ¿Quién produce eventos?

Los servicio-cliente-X (cliente-uno, cliente-dos, etc.) produce los eventos:

- ✓ Estos servicios realizan solicitudes HTTP periódicas a servicio-analíticas.
- ✓ Envían un header personalizado X-Client-Name con su nombre.

2.2 ¿Quién consume estos eventos?

El servicio-analiticas consume los eventos:

- ✓ Recibe las solicitudes HTTP entrantes.
- ✓ Lee el header X-Client-Name y actualiza un contador por cliente.

2.3 ¿Dónde existen acoplamientos directos que podrían desacoplarse?

- ✓ **Acoplamiento directo:** cliente y analíticas: Los clientes dependen de que servicio-analíticas esté disponible en tiempo real. ***Si el servicio-analíticas está caído, los mensajes se pierden o fallan***, es decir, no hay resiliencia ni redundancia. Además, los clientes también deben conocer la URL exacta del backend.

- ✓ **Acoplamiento temporal:** El productor (cliente) y el consumidor (analíticas) deben estar activos al mismo tiempo. ***No hay forma de almacenar mensajes*** si analíticas está momentáneamente fuera de servicio.

- ✓ **Falta de escalabilidad y desacoplamiento:** No se puede escalar el servicio-analiticas fácilmente ***si muchos clientes lo llaman al mismo tiempo*** y, no se pueden reenviar o redirigir mensajes fácilmente a otros servicios (por ejemplo, un logger).

3. Propuesta de rediseño:

3.1 Objetivo del rediseño: Desacoplar la comunicación directa entre los clientes (servicio-cliente-X) y el backend (servicio-analiticas) para lograr un sistema más escalable, tolerante a fallos y flexible, utilizando RabbitMQ como intermediario de mensajes y Traefik como reverce-proxy.

3.2 Cambios clave en la arquitectura

- ✓ Los servicio-cliente-X ya no llaman directamente al servicio-analíticas, en su lugar, publican mensajes a un exchange y una cola en RabbitMQ.
- ✓ El servicio-analíticas ahora actúa como consumer, leyendo mensajes desde una cola y procesándolos.
- ✓ El panel-visual continúa consultando el estado actual del servicio de analíticas (sin conectarse a RabbitMQ directamente).

3.3 Flujo de mensaje propuesto

servicio-cliente → [Exchange: eventos_clientes] → [Queue: cola_analiticas] → servicio-analiticas

|

(Almacena mensajes si el servicio está caído)

3.4 Elementos del diseño

- ✓ **Publisher:** Su nombre como servicio será ***cliente_x*** y es responsable de recibir los eventos de usuarios o sistemas y publicarlos en el Exchange de RabbitMQ (llamado eventos_clientes). Estos servicios se comunican indirectamente con el servicio de análisis a través de RabbitMQ.

  Estos servicios ya no se comunican directamente con el servicio de analíticas, en su lugar, publican los eventos recibidos en una cola RabbitMQ (cola_analiticas).
- ✓ **Exchange:** Tendrá como nombre ***eventos_clientes*** y se encarga de recibir mensajes de los clientes y enrutarlos a la cola correspondiente. Tipo direct o fanout. En la versión inicial es: el *Exchange por defecto* de RabbitMQ
- ✓ **Queue**: Se identificará como ***cola_analiticas*** y es donde se encolan los eventos generados por los clientes. En la versión inicial se llama *hola*.
- ✓ **Consumer**: Se le denominará ***analiticas*** y es el servicio que consume los mensajes de cola_analiticas y actualiza contadores por cliente.
- ✓ **Panel-visual (Consultor HTTP):** No se conecta directamente a RabbitMQ. En su lugar, consulta periódicamente el estado del sistema mediante peticiones HTTP al servicio de analíticas.
  De esta manera, puede mostrar en tiempo real las métricas procesadas, sin necesidad de estar suscrito a la cola.
- ✓ **RabbitMQ (Broker)**: Se llamará ***rabbitmq-service*** como servicio. El contenedor de RabbitMQ actúa como intermediario para la comunicación entre los diferentes servicios. Expondrá los puertos necesarios para AMQP y para la interfaz de administración.
- ✓ **Reverse Proxy (Traefik)**: Se llamará ***reverse-proxy*** y este servicio maneja la configuración de Traefik como Reverse Proxy para los otros servicios, exponiendo el panel de administración en el puerto 8081 y redirigiendo las solicitudes HTTP. Se asegura de que los servicios estén expuestos correctamente y maneja el tráfico entre ellos.

- ✓ **Middleware y Enrutamiento con Traefik**: El servicio panel-visual usa un middleware strip-panel para eliminar el prefijo /panel antes de pasar la solicitud al contenedor que maneja Flask.

3.5 Formato del mensaje: Será tipo JSON. Este formato es ligero, legible y fácilmente extensible. Además, permite agregar más campos (como tipo de evento, metadata, etc.) sin cambiar la estructura principal.

3.6 Lógica del consumidor (servicio-analíticas)

I. Se conecta a RabbitMQ y escucha la cola_analiticas.

II. Por cada mensaje recibido:

- ✓ Extrae el nombre del cliente.
- ✓ Incrementa un contador en memoria para ese cliente.

III. Expone /reporte para visualizar el total de eventos recibidos por cada cliente.

3.6 Estructura del proyecto:

```
├── docker-compose.yml
├── .env
├── sender/
│   ├── app.py
│   ├── requirements.txt
│   └── Dockerfile
├── servicio-analiticas/
│   ├── analiticas.py
│   ├── requirements.txt
│   └── Dockerfile
├── panel-visual/
│   ├── panel.py
│   ├── requirements.txt
│   └── Dockerfile
```