1 Revisión conceptual:

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

RabbitMQ es un sistema de mensajería de código abierto que actúa como intermediario en arquitecturas distribuidas. Su función principal es facilitar la comunicación asíncrona entre servicios mediante el protocolo AMQP (Advanced Message Queuing Protocol). A diferencia de las llamadas HTTP directas, que requieren que ambos servicios estén disponibles simultáneamente, RabbitMQ permite que los servicios productores (publishers) envíen mensajes a colas, donde permanecerán hasta que los servicios consumidores (consumers) los procesen.

En una arquitectura distribuida, RabbitMQ gestiona el flujo de mensajes entre servicios. Por ejemplo, en el sistema descrito en el taller, los clientes ya no necesitan contactar directamente al servicio de analíticas mediante HTTP. En su lugar, publican mensajes en una cola de RabbitMQ, que luego son consumidos por el servicio de analíticas cuando este está disponible. Esto evita pérdidas de datos durante fallos temporales.

Al estandarizar la comunicación mediante mensajes, servicios escritos en diferentes lenguajes o tecnologías pueden interactuar sin problemas. Por ejemplo, un microservicio en Python puede publicar un mensaje en formato JSON que será consumido por otro servicio en Java. Esta interoperabilidad simplifica la integración en entornos complejos.

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

RabbitMQ ofrece importantes ventajas frente a las llamadas HTTP directas, al ser asíncrona y desacoplada. Mientras que HTTP requiere una conexión directa y síncrona entre servicios - donde el cliente debe esperar activamente una respuesta del servidor -, RabbitMQ actúa como un intermediario que almacena mensajes en colas hasta que el consumidor pueda procesarlos. Esto elimina la dependencia temporal entre servicios, permitiendo que sigan funcionando incluso si alguno de ellos falla temporalmente o está sobrecargado.

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

Las **colas** son como buzones de almacenamiento temporal donde se guardan los mensajes hasta que son procesados. Funcionan bajo el principio FIFO (First In, First Out), aunque pueden configurarse para priorizar ciertos mensajes. Las colas mantienen los mensajes seguros incluso si los servicios consumidores fallan o se reinician, garantizando que nada se pierda. Cada cola tiene un nombre único y puede ser configurada con diferentes propiedades como durabilidad (persistencia en disco) o autoeliminación.

Los **exchanges** son los directores de tráfico del sistema. Reciben los mensajes de los publishers y deciden a qué colas deben enviarlos, aplicando reglas específicas. Existen diferentes tipos de exchanges (direct, fanout, topic y headers) que varían en su lógica de enrutamiento. Por ejemplo, un exchange "fanout" envía copias del mensaje a todas las colas vinculadas, mientras que un exchange "direct" lo envía solo a las colas cuya clave de enlace coincida exactamente con la routing key del mensaje.

Los **publishers** (o productores) son los servicios o aplicaciones que originan los mensajes. Su responsabilidad es crear mensajes con el formato adecuado (como JSON o XML) y enviarlos a un exchange específico, indicando una routing key cuando sea necesario. Los publishers no necesitan saber qué servicios eventualmente procesarán sus mensajes ni cuándo lo harán, lo que permite un completo desacoplamiento. Solo requieren conocer la existencia del exchange al que envían los mensajes.

Los **consumers** (o consumidores) son los servicios que reciben y procesan los mensajes. Se suscriben a colas específicas y RabbitMQ les entrega los mensajes según estén disponibles. Los consumers pueden enviar acuses de recibo (acknowledgements) para confirmar el procesamiento exitoso o indicar fallos. Múltiples consumers pueden trabajar en paralelo consumiendo de la misma cola, permitiendo distribuir la carga de trabajo horizontalmente.

2. Análisis del sistema actual

2.1 Identificar en la arquitectura del parcial actual

2.1.1 ¿Quién produce eventos?

En el sistema actual, los principales productores de eventos son los servicios cliente (service-cliente-X), que generan eventos de registro periódicos. Cada instancia de servicio cliente, identificada por su SERVICE_ID único, produce estos eventos mediante llamadas HTTP POST al endpoint /registro del servicio registro-app. Estos eventos se generan tanto cuando se accede a la ruta raíz del servicio cliente como mediante un hilo en segundo plano que ejecuta registros cada 10 segundos.

2.1.2 ¿Quién consume estos eventos?

En el sistema actual basado en HTTP, el consumidor de los eventos es el servicio registro-app, que recibe y procesa las solicitudes de registro de manera síncrona.

2.1.3 ¿Dónde existen acoplamientos directos que podrían desacoplarse?

El principal acoplamiento temporal ocurre porque los clientes realizan llamadas HTTP síncronas al servicio de registro, requiriendo que ambos sistemas estén disponibles simultáneamente.

3. Propuesta de rediseño

3.1 Los servicio-cliente-X ya no llamen directamente al servicio-analiticas, sino publiquen mensajes a una cola RabbitMQ.

```
# Conexión a RabbitMQ (reemplaza HTTP)
connection =
pika.BlockingConnection(pika.ConnectionParameters('rabbitmq'))  #
← Broker RabbitMQ
channel = connection.channel()
    channel.queue_declare(queue=f'client_{SERVICE_ID}')  # ←
Declara cola única por cliente

def registrar_servicio():

    message = {
        "service_id": SERVICE_ID,
        "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
    }
    # Publica mensaje en la cola (en lugar de HTTP POST)
    channel.basic_publish(
        exchange='',  # ← Exchange por defecto
        routing_key=f'client_{SERVICE_ID}',  # ← Nombre de la
cola como routing key
        body=json.dumps(message)  # ← Contenido del mensaje
    )
    channel.close()
```

3.2 El servicio-analiticas actúa como consumer de esa cola, procesando los eventos recibidos.

```
# Configuración inicial del consumer
connection =
pika.BlockingConnection(pika.ConnectionParameters('rabbitmq'))
channel = connection.channel()

# Declara las colas
```

```
channel.queue_declare(queue='client_app1')
channel.queue_declare(queue='client_app2')
channel.queue_declare(queue='client_app3')

# Lógica para consumir mensajes
def calculateMessages():
    # Procesa mensajes de client_app1
    while True:
        method_frame, _, body =
channel.basic_get(queue='client_app1', auto_ack=True)
        if method_frame:
            registro[0][1] += 1  # Actualiza contador
            print(f"Consumido de client_app1: {body}")
        else:
            break

    # Repite para client_app2 y client_app3...
    # (Código similar para las otras colas)
```

Preguntas:

¿Qué beneficio aporta RabbitMQ en comparación con el modelo de solicitud directa HTTP?

Mientras que HTTP opera bajo un esquema de solicitud-respuesta sincrónica que requiere disponibilidad inmediata de ambos extremos, RabbitMQ actúa como un buffer que almacena mensajes hasta que puedan ser procesados. Este enfoque elimina la dependencia temporal entre servicios, permitiendo que los productores sigan funcionando aunque los consumidores estén temporalmente inactivos o sobrecargados.

¿Qué problemas podrían surgir si se caen algunos servicios?

En el diseño actual, basado en llamadas HTTP síncronas, la caída de cualquier servicio tiene un impacto inmediato y directo. Por ejemplo, si un servicio cliente se cae, deja de generar eventos, lo cual implica una pérdida de datos en tiempo real, aunque el resto del sistema continúe funcionando. Sin embargo, si el servicio de registro (registro-app) se cae, los servicios clientes no podrán completar sus solicitudes HTTP y los eventos no serán registrados.

En esta nueva arquitectura, si un servicio cliente falla, simplemente se detiene la generación de eventos, pero no afecta a los demás componentes del sistema.

¿Cómo ayuda RabbitMQ a mejorar la resiliencia del sistema?

RabbitMQ mejora la resiliencia del sistema al introducir una capa de intermediación entre los productores y consumidores de mensajes, lo que permite desacoplar los componentes del sistema. Esta desacoplación significa que los productores pueden seguir funcionando y generando eventos, incluso si los consumidores no están disponibles en ese momento, ya que los mensajes se almacenan de forma transitoria en las colas de RabbitMQ. De esta forma, se evita la pérdida de datos y se gana flexibilidad operativa frente a fallos parciales.

¿Cómo cambiaría la lógica de escalabilidad con esta nueva arquitectura?

La cola de RabbitMQ actúa como un buffer temporal que desacopla el ritmo de producción y consumo. Si en algún momento los clientes envían más eventos de los que pueden procesarse inmediatamente, los mensajes se almacenan en la cola y se procesan tan pronto como haya capacidad disponible. Esto evita la sobrecarga de servicios consumidores y permite escalar reactivamente: se pueden lanzar más consumidores solo cuando sea necesario.

¿Qué formato de mensaje es más conveniente y por qué (JSON, texto plano, etc.)?

JSON permite representar datos estructurados de forma clara y jerárquica. A diferencia del texto plano, JSON puede incluir múltiples campos clave-valor, listas, objetos anidados y tipos de datos básicos (números, cadenas, booleanos), lo que facilita el modelado de eventos complejos.

Diagrama de arquitectura con RabbitMQ incluido