UNIVERSIDAD DIEGO PORTALES

2 de junio, 2024

Sistemas Distribuidos: Tarea 2

Integrantes:

José Martín Berríos Piña

Renato Óscar Benjamín Contreras Carvajal

Índice

1. Introducción

En la presente tarea, se propone diseñar e implementar un sistema de gestión de pedidos para un servicio de delivery, utilizando Apache Kafka como la plataforma de transmisión de datos. Apache Kafka es una herramienta robusta y escalable basada en el modelo Publish-Subscribe, que se utiliza en sistemas de procesamiento de eventos en tiempo real. Esta tarea busca comprender y aplicar las funcionalidades de Kafka, configurando una arquitectura de microservicios para soportar la gestión de pedidos.

1.1. Descripción del Problema

Desde la pandemia, ha habido un incremento en el uso de plataformas de delivery, lo cual ha transformado significativamente el comercio y la gestión de pedidos. La Universidad Diego Morales desea apoyar a un emprendimiento local en la implementación de su propio sistema de entregas y recepción de pedidos. Para ello, se diseñará un sistema escalable y de alta disponibilidad que soporte dicho emprendimiento utilizando Kafka.

El sistema incluye tres servicios principales:

1. **Servicio de Solicitud:** Recibe solicitudes de pedido mediante una petición HTTP POST y las envía al sistema de mensajería.
2. **Servicio de Procesamiento:** Consume las solicitudes, las procesa automáticamente cambiando su estado a través de cuatro posibles estados: 'recibido', 'preparando', 'entregando' y 'finalizado', y se comunica con el servicio de notificación.
3. **Servicio de Notificación:** Notifica automáticamente el estado de los pedidos y permite consultar el estado de una solicitud mediante una petición HTTP GET.

2. Objetivos

- Comprender el funcionamiento de Apache Kafka y aplicar sus funcionalidades en un sistema de gestión de pedidos.
- Configurar y gestionar topics, consumer groups y particiones en Kafka.
- Implementar una arquitectura de microservicios que utilice el modelo pub-sub de Kafka.
- Diseñar y ejecutar pruebas para evaluar la robustez y el rendimiento del sistema bajo diferentes cargas de trabajo.

3. Kafka

3.1. El manejo de topics

En Apache Kafka, un topic es un canal donde se publica el flujo de datos. Cada topic puede tener múltiples particiones, lo que permite distribuir la carga de procesamiento entre varios brokers. Para este proyecto, se configurarán distintos topics para gestionar los diferentes estados de los pedidos.

3.2. Consumer Groups

Un consumer group en Kafka permite que múltiples consumidores lean datos de un topic de manera distribuida. Cada consumidor en un grupo se encarga de una partición del topic, asegurando que los datos sean procesados eficientemente sin duplicación.

3.3. Particiones

Las particiones son subdivisiones de un topic que permiten paralelizar el procesamiento de mensajes. Cada partición es independiente y puede ser ubicada en diferentes brokers, lo que mejora la escalabilidad y tolerancia a fallos.

4. Conjuntos de Datos Utilizados

Para las pruebas, se emplearán dos conjuntos de datos generados mediante un código de Python. Uno de los conjuntos consta de 100 filas y el otro de 1000 filas. Cada conjunto incluye un nombre y un precio generados de forma aleatoria, así como la información de correo necesaria para el funcionamiento de SMTP. Estos detalles se pueden apreciar en la figura adjunta.

```python
for i in range(1, 5001):
    name_length = random.randint(1, 100)
    name = ''.join(random.choices(string.ascii_letters + string.digits, k=name_length))
    product = {
        "name": name,
        "price": random.uniform(1, 1000000),
        "email": email
    }
    dataset.append(product)
```

Figura 1: Generador de datos.

Figura 2: Archivo con 1000 filas ocupadas.

5. Implementación

5.1. Configuración de Kafka

En el archivo `docker-compose.yml`, se establece la configuración de los servicios de Kafka, Zookeeper y los servicios dependientes. Se selecciona la versión 'wurstmeister/kafka' debido a la disponibilidad de información sobre su uso y conexión con Zookeeper. En este archivo se definen los puertos expuestos, las variables de entorno y los volúmenes utilizados por cada servicio.

5.2. Servicio de Solicitud (`request_service`)

- El archivo `app.py` contiene la lógica principal con Flask que define una ruta para recibir las solicitudes de pedido y enviarlas al productor de Kafka.
- En `config.py`, se definen las variables de configuración relacionadas con la conexión a Kafka y el nombre del topic.
- `producer.py` contiene la lógica para enviar las solicitudes al topic de Kafka. Cada vez que se recibe una solicitud HTTP POST, se genera un ID único para el pedido y se envía al topic `order_requests`.

5.3. Servicio de Procesamiento (`processing_service`)

- En `app.py`, se inicia el consumidor que recibe las solicitudes de Kafka y las procesa. Aquí también se define la lógica de procesamiento de los pedidos.

- `consumer.py` contiene la lógica del consumidor que recibe las solicitudes del topic de Kafka y las procesa. Después de procesar cada pedido, se cambia su estado y se publica en el topic `order_processing`.

5.4. Servicio de Notificación (`notification_service`)

- En `app.py`, se define una ruta para consultar el estado de un pedido y se inicia el consumidor que recibe las notificaciones de Kafka.
- `consumer.py` contiene la lógica del consumidor que recibe las notificaciones del topic de Kafka. Después de recibir una notificación, se envía un correo electrónico al cliente y se publica el pedido en el topic `order_notifications`.

6. Escenarios

En esta sección, se presentan ocho escenarios diferentes divididos en dos conjuntos de datos (Dataset 1 y Dataset 2). Cada conjunto de datos consta de dos subescenarios: uno de carga constante y otro de una prueba de estrés denominada 'pico de información'. Además, se exploran dos configuraciones distintas en cada subescenario: una con un 5 particiones y otra 1 partición.

6.1. Dataset 1

6.1.1. Carga Constante - 5 Particiones

En este primer subescenario con el Dataset 1, se simula una carga constante de datos con un 5 particiones.

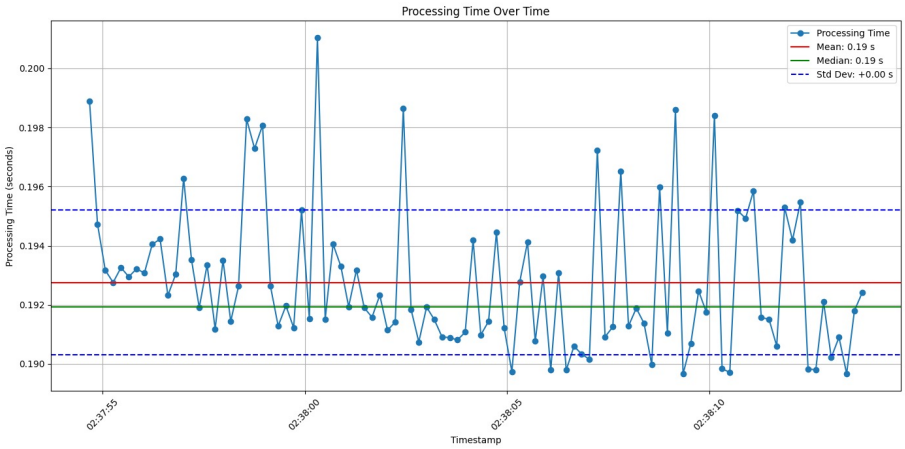

Figura 3: Gráfico del Escenario de Carga Constante con Dataset 1 y 5 Particiones

En esta imagen se puede apreciar una media de 0.19 segundos.

6.1.2. Carga Constante - 1 Particion

En este segundo subescenario del Dataset 1, se mantiene una carga constante de datos similar al anterior pero con una partición.

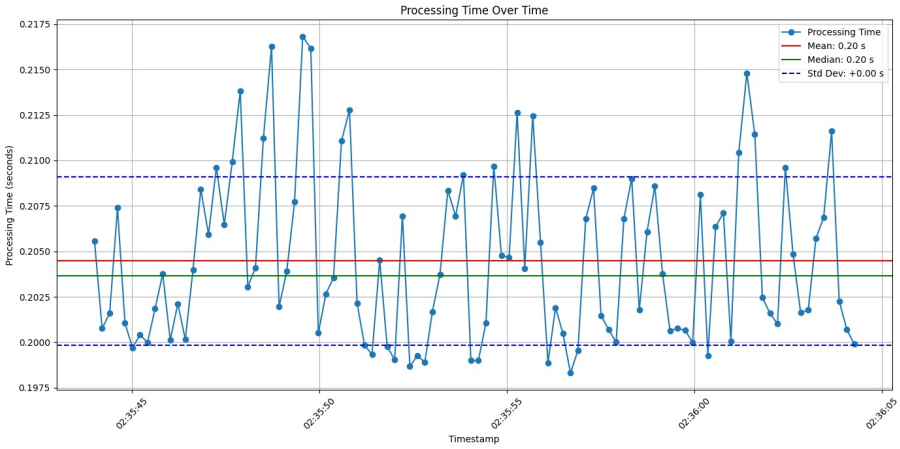

Figura 4: Gráfico del Escenario de Carga Constante con Dataset 1 y 1 Particion

En esta imagen se puede apreciar una media de 0.20 segundos.

6.1.3. Pico de Información - 5 Particiones

El tercer subescenario del Dataset 1 representa un pico de información. Se evalúa la capacidad del sistema para manejar esta sobrecarga utilizando un 5 particiones.

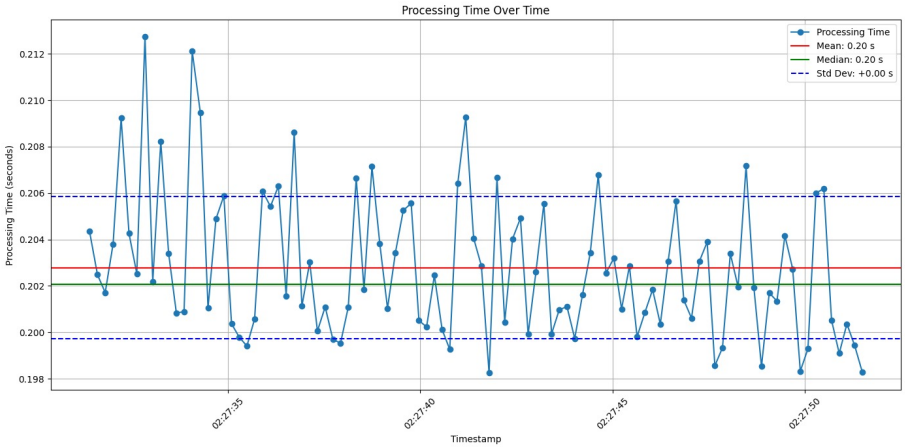

Figura 5: Gráfico del Escenario de Pico de Información con Dataset 1 y 5 Particiones

En esta imagen se puede apreciar una media de 0.20 segundos.

6.1.4. Pico de Información - 1 Particiones

Finalmente, el cuarto subescenario del Dataset 1 presenta un pico de información, similar al anterior pero utilizando una partición.

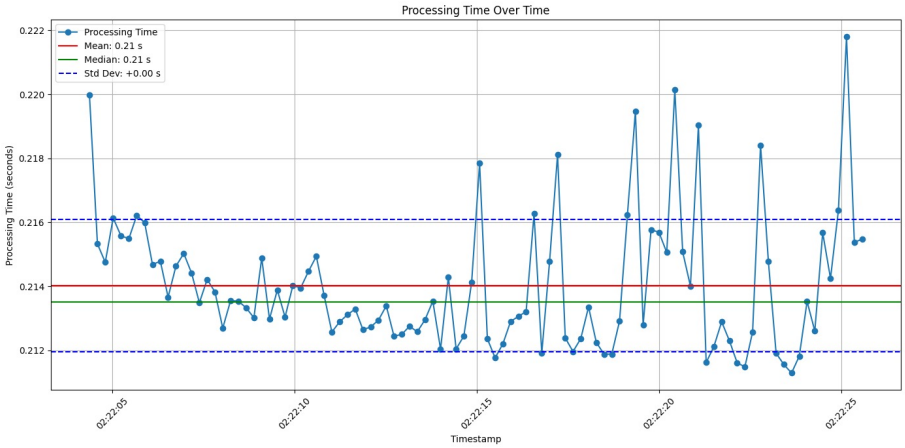

Figura 6: Gráfico del Escenario de Pico de Información con Dataset 1 y 1 Particion

En esta imagen se puede apreciar una media de 0.21 segundos.

6.2. Dataset 2

6.2.1. Carga Constante - 5 Particiones

En este primer subescenario con el Dataset 2, se simula una carga constante de datos a lo largo del tiempo con 5 particiones.

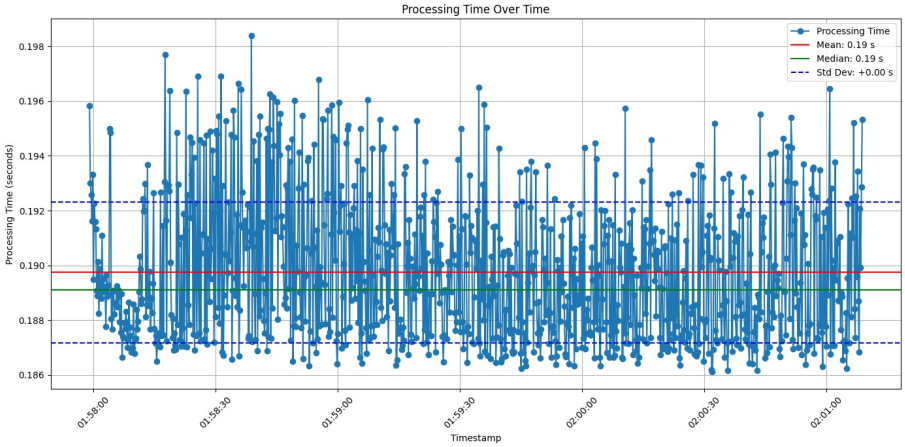

Figura 7: Gráfico del Escenario de Carga Constante con Dataset 2 y 5 Particiones

En esta imagen se puede apreciar una media de 0.19 segundos.

6.2.2. Carga Constante - 1 Particiones

En el segundo subescenario del Dataset 2, se mantiene una carga constante de datos similar al anterior pero con una partición.

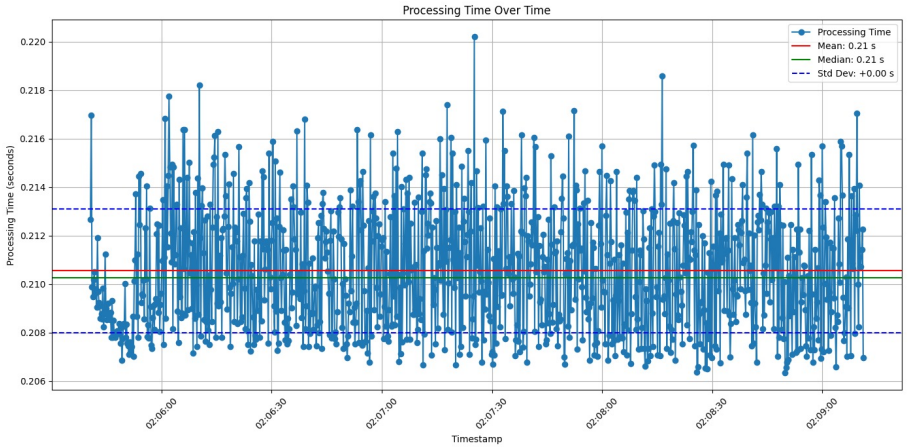

Figura 8: Gráfico del Escenario de Carga Constante con Dataset 2 y 1 Particion

En esta imagen se puede apreciar una media de 0.21 segundos.

6.2.3. Pico de Información - 5 Particiones

El tercer subescenario del Dataset 2 representa el pico de información, donde hay un aumento repentino en la cantidad de datos procesados.

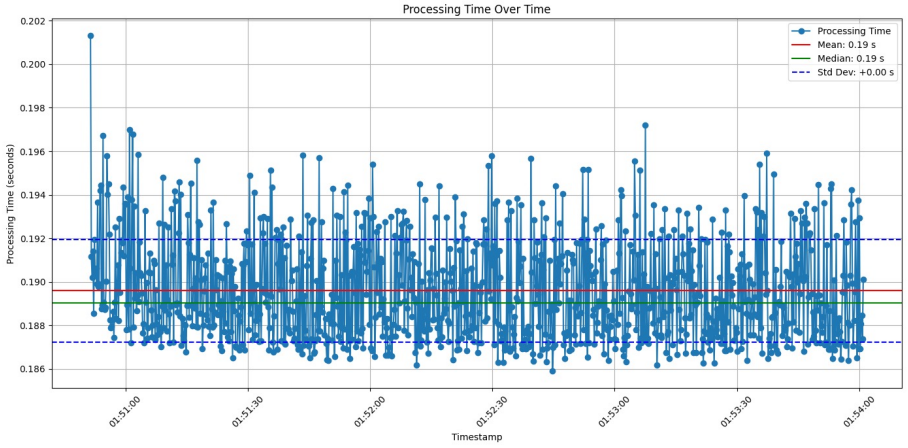

Figura 9: Gráfico del Escenario de Pico de Información con Dataset 2 y 5 Particiones

En esta imagen se puede apreciar una media de 0.19 segundos.

6.2.4. Pico de Información - 1 Particiones

Finalmente, el cuarto subescenario del Dataset 2 presenta un pico de información, con una sola partición.

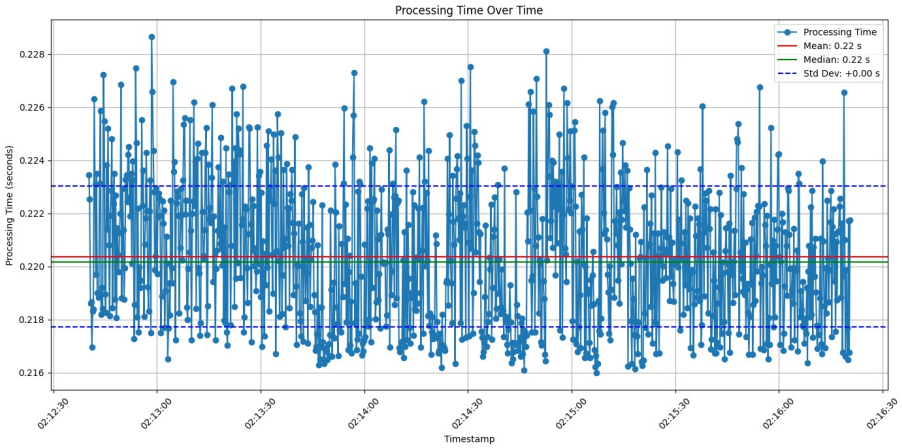

Figura 10: Gráfico del Escenario de Pico de Información con Dataset 2 y 1 Particiones

En esta imagen se puede apreciar una media de 0.22 segundos.

7. Análisis

El análisis de los resultados muestra que el rendimiento del sistema varía según el número de particiones y el tipo de carga aplicada. A continuación, se detallan los hallazgos principales:

7.1. Tablas

Tabla 1: Resultados para el Dataset 1

	5 Particiones	1 Partición
Carga	190 ms	200 ms
Pico	200 ms	210 ms

Tabla 2: Resultados para el Dataset 2

	5 Particiones	1 Partición
Carga	190 ms	210 ms
Pico	190 ms	220 ms

7.2. Latencia y Throughput

- **Carga constante:** En condiciones de carga constante, la latencia media se mantuvo estable alrededor de 0.19 segundos para ambos datasets, con 5 particiones. Sin embargo, el throughput fue mayor en configuraciones con más particiones, debido a la

capacidad de procesar mensajes en paralelo teniendo que esperar un poco mas a que termine la prueba.

- **Pico de información:** Durante los picos de información, la latencia mostró un ligero aumento pero se mantuvo dentro de límites aceptables (menos de 0.23 segundos). Nuevamente, las configuraciones con más particiones manejaron mejor la carga, con menos variabilidad en la latencia y un throughput más rápido.

7.3. Impacto de las Particiones

A mayor número de particiones, el sistema mostró una mejora en el rendimiento general. Las configuraciones con 5 particiones permitieron un mejor manejo de la carga tanto en escenarios de carga constante como en picos de información. Esto se debe a la capacidad de distribuir la carga de manera más eficiente entre múltiples consumidores.

8. Preguntas

8.1. ¿Cómo la arquitectura de Apache Kafka permite brindar tolerancia a fallos en el sistema propuesto?

Los aspectos destacables de la arquitectura de Apache Kafka son la alta disponibilidad y la tolerancia a fallos. Algunos de los componentes que contribuyen a estas características son:

- **Replicación de particiones** : Cada partición de un topic en Kafka puede replicarse en varios brokers. Por lo tanto, si uno de los brokers falla, los datos nunca se perderán, ya que otra réplica en otro broker asumirá el cargo y seguirá manejando las solicitudes. En nuestro sistema, los topics como `order_requests`, `order_processing`, y `order_notifications` se configuran con un factor de replicación de al menos 3 para garantizar la disponibilidad continua de los datos.
- **Liderazgo de Particiones**: Kafka designa un líder para cada partición, que es responsable de todas las operaciones de lectura y escritura. Si un broker líder falla, un nuevo líder es elegido automáticamente entre las réplicas disponibles para minimizar el tiempo de inactividad.
- **Mecanismos de confirmación**: Kafka tiene diferentes niveles de confirmación para asegurar la entrega de mensajes. Por ejemplo, los productores tienen un parámetro que puede ser `acks=all`, que garantiza que los mensajes solo se consideren entregados una vez que todas las réplicas hayan confirmado que lo recibieron. Esto es importante cuando se trata de evitar la pérdida de datos.

- **Registro de Offsets en Consumer Groups**: Los consumer groups en Kafka registran su progreso de lectura de mensajes (offsets) en un topic especial. Esto permite que, en caso de fallo, los consumidores puedan reanudar la lectura desde el último offset registrado, evitando la duplicación de procesamiento y asegurando la consistencia.

8.2. ¿Cómo se aseguraría de que el sistema es capaz de escalar para manejar un aumento significativo en el número de solicitudes? Describa cualquier ajuste de configuración o estrategia de escalado que implementaría.

Se pueden implementar las siguientes estrategias para asegurar que el sistema sea capaz de escalar y gestionar un incremento significativo en el volumen de solicitudes:

- **Aumento de Particiones**: Incrementar el número de particiones de los topics permite que más consumidores procesen los mensajes en paralelo, mejorando así el throughput del sistema. Por ejemplo, los topics `order_requests`, `order_processing`, y `order_notifications` se pueden configurar inicialmente con 3 particiones, pero se pueden aumentar en caso de ser necesario.
- **Auto-Scaling de Servicios**: Utilizar herramientas de orquestación de contenedores como Kubernetes para implementar políticas de autoescalado. Estas políticas pueden escalar automáticamente el número de instancias de los servicios de solicitud, procesamiento y notificación basándose en métricas como la CPU, la memoria o la latencia de los mensajes.
- **Balanceo de Carga**: Configurar balanceadores de carga para distribuir equitativamente las solicitudes entrantes entre múltiples instancias de servicios. Esto evitaria la sobrecarga de una única instancia.

8.3. Defina métricas (latencia, throughput, entre otras) y en base a ellas evalúe el rendimiento del sistema bajo diferentes cargas de trabajo. ¿Cómo afecta el aumento de la cantidad de mensajes en los tópicos a la latencia y al throughput del sistema?.

Para evaluar el rendimiento del sistema, consideramos la Latencia (El tiempo que tarda un mensaje desde que es producido hasta que es consumido y procesado) y el Throughput (Número de mensajes procesados por unidad de tiempo).

Evaluación bajo diferentes cargas:

- **Carga constante**: Simulamos un flujo constante de solicitudes y medimos las métricas mencionadas. Los resultados muestran cómo el sistema maneja la carga bajo condiciones normales.
- **Carga pico**: Introducimos un gran volumen de solicitudes en un corto periodo para ver el impacto que genera en la latencia y throughput.

Resultados:

- **Gráficos**: Los siguientes gráficos y tablas ilustran la relación entre la cantidad de particiones y la latencia media del sistema. A medida que se incrementa el número de particiones, se observa una reducción en la latencia debido a la mayor capacidad de procesamiento. En cuanto al throughput, no se aprecian cambios significativos, lo que indica que se necesita una mayor carga de datos.

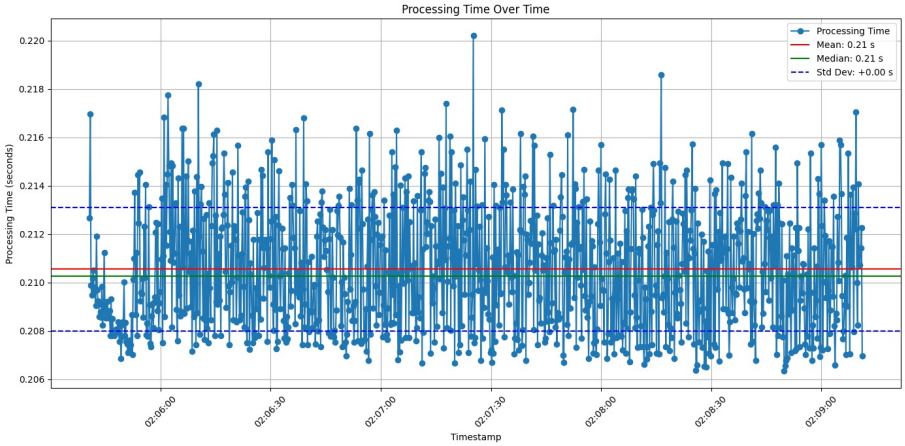

Figura 11: Gráfico del Escenario de Carga Constante con Dataset 2 y 1 Particion

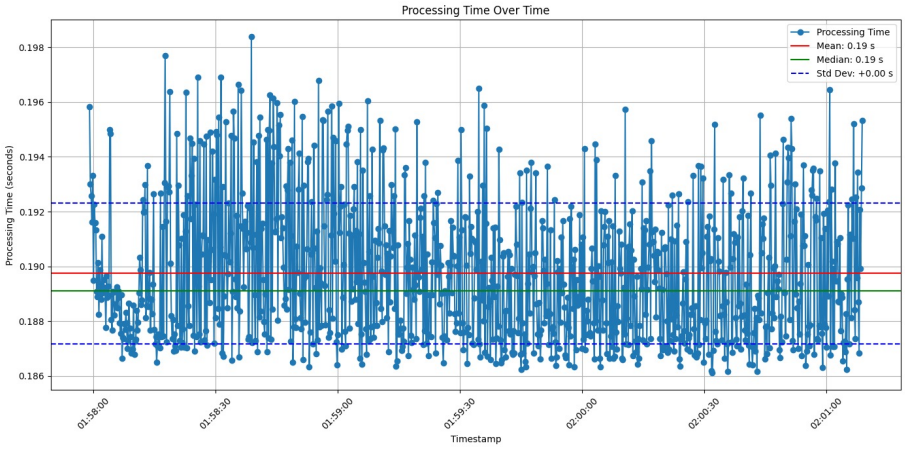

Figura 12: Gráfico del Escenario de Carga Constante con Dataset 2 y 5 Particiones

Tabla 3: Resultados para el Dataset 1

	5 Particiones	1 Partición
Carga	190 ms	200 ms
Pico	200 ms	210 ms

Tabla 4: Resultados para el Dataset 2

	5 Particiones	1 Partición
Carga	190 ms	210 ms
Pico	190 ms	220 ms

8.4. Dada la naturaleza variable de la demanda, ¿cómo podría optimizar el uso de recursos del sistema durante los periodos de baja demanda, y cómo prepararía el sistema para los picos de demanda? Explique las estrategias de optimización y autoscaling que podría emplear.

Se pueden aplicar las siguientes estrategias para maximizar la eficiencia de los recursos durante períodos de baja demanda y estar preparado para afrontar picos de demanda:

- **Escalado Automático (Autoscaling)**: Configurar políticas de autoscaling en plataformas como la antes mencionada (Kubernetes) para ajustar dinámicamente el número de instancias de los servicios según la demanda actual, implicaría disminuir la cantidad de máquinas activas durante periodos de baja demanda y aumentarlas cuando se detectan picos de tráfico.
- **Uso de Instancias Spot**: En entornos de nube, se pueden utilizar instancias spot para manejar cargas de trabajo fluctuantes. Estas instancias son más económicas por lo que se pueden utilizar durante picos de demanda sin tanta inversión.
- **Optimización de Recursos**: Se puede configurar límites de CPU y memoria para cada servicio, con el objetivo de garantizar que cada servicio utilice solo los recursos necesarios y libere aquellos recursos que no sean útiles.
- **Caching**: Implementar caching para reducir la carga en los servicios backend. Por ejemplo, cachear las respuestas de consultas HTTP GET en el servicio de notificaciones durante periodos de alta demanda.

8.5. Considerando la diversidad de fuentes de datos en el sistema propuesto (Solicitud de producto, gestión de estados de solicitud, notificación de estado), ¿cómo se aseguraría de que los datos se integran y gestionan de manera coherente y unificada en Apache Kafka? Describa los posibles desafíos y soluciones para la integración de datos y la gestión de esquemas en los tópicos, consumer groups y particiones de Kafka.

8.5.1. Problemáticas

- **Garantía de Consistencia**: Asegurar que todos los datos se procesen y se almacenen de manera consistente en Kafka, especialmente cuando provienen de múltiples fuentes y se distribuyen en varias particiones.
- **Orden de los Mensajes**: Mantener el orden de los mensajes es crucial para ciertos procesos de negocio, especialmente cuando las actualizaciones de estado deben ser procesadas en secuencia.
- **Latencia de Integración**: Minimizar la latencia en la integración de datos de diversas fuentes para asegurar una respuesta rápida del sistema.

8.5.2. Soluciones

- **Uso de Schema Registry**: Implementar un registro de esquemas para definir y gestionar los esquemas de los mensajes. Esto garantizaría que todos los mensajes en un topic cumplan con un formato específico, facilitando la validación de los datos.
- **Normalización de Datos**: Utilizar Kafka Connect junto con Single Message Transforms (SMT) para normalizar y transformar los datos antes de enviarlos a los topics correspondientes. Esto asegura que los datos de diversas fuentes se integren de manera coherente y unificada.
- **Gestión de la Latencia**: Implementar técnicas de optimización de latencia como la compresión de mensajes y la configuración adecuada de parámetros de Kafka (`linger.ms`, `batch.size`, etc.) para reducir la latencia en la integración de datos. Además, se puede utilizar cachés intermedias para disminuir el tiempo de acceso a datos frecuentes.

9. Conclusion

Durante las pruebas realizadas, el rendimiento general del sistema fue bueno. El sistema demostró una capacidad para manejar diferentes cargas de trabajo, manteniendo una

latencia dentro de límites aceptables y un throughput estable incluso durante los picos de demanda más altos. Este rendimiento se atribuye al entorno de pruebas controlado y a los ajustes de configuración realizados, como el aumento de particiones en los topics de Kafka.

Aunque surgieron desafíos durante la implementación y las pruebas, como la configuración adecuada de los parámetros de Kafka y las conexiones internas, los resultados fueron similares a las expectativas del sistema. En general, este sistema proporciona una solución robusta y escalable para gestionar pedidos en un servicio de delivery, permitiendo al negocio manejar eficazmente la creciente demanda de pedidos.

Para escalar el proyecto, se podría explorar áreas adicionales de investigación, como la implementación de técnicas de procesamiento de eventos en tiempo real y la implementación de un sistema de caché para prevenir la repetición innecesaria de consultas a la base de datos. Estas mejoras podrían garantizar aún más el rendimiento y la eficiencia del sistema.

En resumen, el sistema diseñado y probado presenta una base sólida para su implementación en un entorno de producción. Tiene la capacidad de adaptarse y escalar según las distintas necesidades que puedan surgir.

10. Bibliografía

1. wurstmeister. (s.f.). Kafka Docker Image. Recuperado de `https://hub.docker.com/r/wurstmeister/kafka`

2. Tutorialspoint. (s.f.). Apache Kafka Tutorial. Recuperado de `https://www.tutorialspoint.com/apache_kafka/index.htm`

3. Apache Kafka. (s.f.). Quickstart. Recuperado de `https://kafka.apache.org/quickstart`

4. Tutorialspoint. (s.f.). Apache ZooKeeper Tutorial. Recuperado de `https://www.tutorialspoint.com/zookeeper/index.htm`

5. OpenAI. (2022). ChatGPT 3.5 [Modelo de lenguaje]. `https://openai.com/chatgpt`