UNIVERSIDAD DIEGO PORTALES

“Sistemas para la gestión de pedidos de delivery”.

Profesor: Nicolás Hidalgo
Integrantes: Diego Pastrián - Rodrigo Yáñez.
Asignatura: Sistemas Distribuidos.

Índice

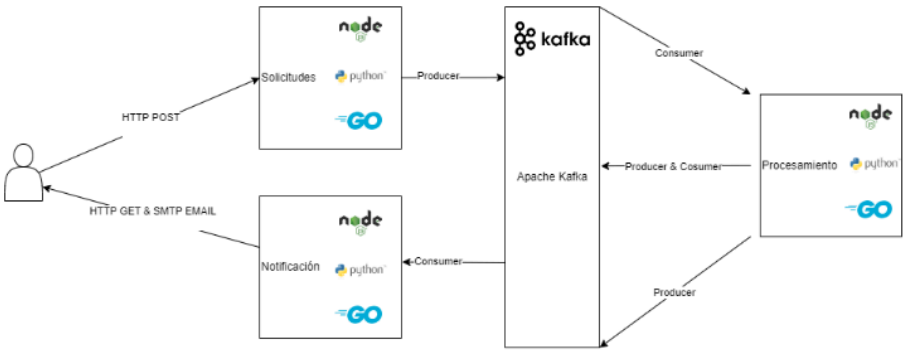
1. Introducción

Los sistemas de procesamiento basados en eventos de tipo stream son usados para procesar datos en la medida en que se generan para así obtener cambios en tales datos. Relacionado a esto, Apache Kafka es una es una plataforma distribuida para la transmisión de datos que permite publicar, almacenar y procesar flujos de registros como también así subscribirse a estos.

En este informe se detalla el desarrollo de un sistema de gestión de pedidos de delivery utilizando apache kafka para la transmisión/procesamiento de eventos en un proyecto nodejs. La principal función de este sistema es la gestión del estado de un pedido, desde su solicitud hasta la entrega al usuario. Como herramientas complementarias se tienen PostgreSQL y Docker.

2. Desarrollo

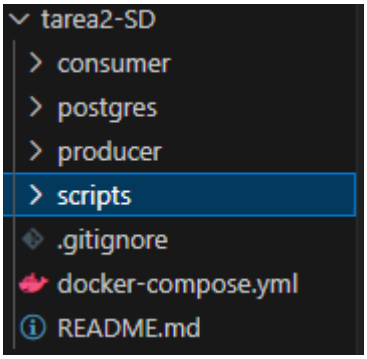
2.1. Planteamiento del sistema



La imagen corresponde a la arquitectura del sistema a implementar.
El problema a desarrollar en este proyecto es el de el procesamiento de datos referentes a un pedido. El usuario va a realizar un pedido el cual va a ser ingresado al sistema y este debe procesarse hasta finalmente estar listo. Referente a lo anterior se deben realizar tres servicios: De solicitud, de procesamiento y de notificación.
Servicio de solicitud: Será el encargado de ingresar un pedido al sistema mediante HTTP POST.
Servicio de procesamiento: Encargado de actualizar el estado del pedido, existiendo 4 estados: “recibido”, “preparando”, “entregado”, “finalizado”.
Servicio de notificación: Se encarga de notificar al usuario por el uso de correos sobre el estado del pedido a medida que este se actualice.
Producer corresponde a quien inyecta datos en el servicio y consumer quien los consume.
Se utilizará una BDD postgres para poder ingresar, consultar y actualizar los pedidos hechos. Además se va a usar un dataset de productos que cuenta con atributo “product_name” y “price”. Este dataset cuenta con 1000 filas.

2.2. Código desarrollado

Los directorios del proyecto se visualizan de la siguiente manera:

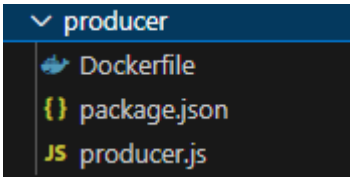


A continuación se realizará un análisis del desarrollo de los directorios importantes del proyecto.
Una de las herramientas principales a utilizar en este proyecto es el uso de Docker, el cual permite ejecutar las distintas aplicaciones asociadas en contenedores. Los contenedores asociados al proyecto están en el archivo docker-compose.yml el cual permite facilitar la

ejecución de múltiples servicios en un solo archivo, donde se organiza la ejecución de los contenedores zookeeper, kafka, postgres, producer y consumer.

2.2.1. Producer

Para la carpeta “producer”, se tienen los siguientes archivos:



Dockerfile es utilizado en varias ocasiones en este proyecto para poder crear las instrucciones para construir una imagen Docker en los contenedores a utilizar para cada parte del código. El archivo producer.js es donde se implementa el servicio que actúa como producer en esta arquitectura Kafka a utilizar.

A continuación se desglosa el archivo producer.js:

Primeramente se importan las dependencias a utilizar.

```
1  const express = require('express');
2  const cors = require('cors');
3  const bodyParser = require('body-parser');
4  const { Kafka, logLevel } = require("kafkajs");
5  const pg = require('pg');
```

Se configura la conexión a la BDD en la constante ‘pool’ y también se inicializa la app express, permitiendo solicitudes desde cualquier origen. Además se realiza la configuración del cliente kafka con su id de cliente y brokers a utilizar.

```
const pool = new pg.Pool({
    user: 'user',
    host: 'postgres',
    database: 'tarea2',
    password: 'user',
    port: 5432
});

const app = express();
app.use(cors());
app.use(bodyParser.json());

const kafka = new Kafka({
    clientId: "my-app",
    brokers: ["kafka:9092"],
    logLevel: logLevel.NOTHING
});
```

A continuación se presenta el endpoint para manejar las nuevas solicitudes entrantes, es decir crear una nueva orden. Este se relaciona al servicio de Solicitud.

```
25  app.post('/new_order', async (req, res) => {
26      if (!req.body.name || !req.body.price) {
27          console.log("Body vacio")
28          return res.status(400).json({ message: ".body vacio" })
29      }
30
31      const status = "recibido";
32      const id = (await pool.query(
33          'INSERT INTO products (product_name, price, status) VALUES ($1, $2, $3) returning id',
34          [req.body.name, req.body.price, status]
35      )).rows[0].id;
36
37      const producer = kafka.producer();
38      await producer.connect();
39
40      await producer.send({
41          topic: "delivery",
42          messages: [
43              { value: JSON.stringify({ id, ...req.body, status })}
44          ]
45      });
46
47      await producer.disconnect();
48      res.status(200).json({ message: "Orden enviada" });
49  })
```

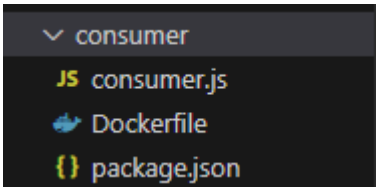
En las líneas 26 a 29 se verifica que el cuerpo de la request JSON contenga precio y nombre. Luego, de las líneas 31 a 36 se inserta el producto en la base de datos con el estado “recibido” obteniendo su id. Después inicializando y conectando el producer Kafka, se envía al tópico delivery la orden con el estado inicial “recibido” y su id designado. Finalmente se desconecta el producer.

```
51  app.get('/orders', async (req, res) => {
52      const response = await pool.query('SELECT * FROM products');
53      console.log(response.rows);
54      res.status(200).json(response.rows);
55  })
56
57  app.listen(3000, () => {
58      console.log("\nServer running on port 3000\n");
59  });
```

El endpoint /orders permite obtener todas las órdenes y se inicia el servidor express en el puerto 3000.

2.2.2. Consumer

Para la parte del consumer, se tienen los siguientes archivos:



Siendo consumer.js donde se implementa el servicio que actúa como consumidor en esta arquitectura Kafka a utilizar y dockerfile el script que contiene lo necesario para construir la imagen consumer para el docker.

```
1   const express = require('express');
2   const cors = require('cors');
3   const bodyParser = require('body-parser');
4   const { Kafka, logLevel } = require("kafkajs");
5   const pg = require('pg');
6   const nodemailer = require('nodemailer');
```

Primeramente se realiza la importación de los módulos necesarios a utilizar. Nodemailer se utilizará para el envío de los correos electrónicos.

```
8   const pool = new pg.Pool({
9       user: 'user',
10      host: 'postgres',
11      database: 'tarea2',
12      password: 'user',
13      port: 5432
14  });
15
16  const transporter = nodemailer.createTransport({
17      host: 'smtp.gmail.com',
18      port: 465,
19      secure: true,
20      auth: {
21          user: 'testmailkafka@gmail.com',
22          pass: 'qyixmqzthkghhckq',
23      }
24  });
```

Con pool se configura la conexión a la base de datos y transporter corresponde al transporte para poder enviar los correos electrónicos.

```
26  const app = express();
27  app.use(cors());
28  app.use(bodyParser.json());
29
30  const kafka = new Kafka({
31      clientId: "my-app",
32      brokers: ["kafka:9092"],
33      logLevel: logLevel.NOTHING
34  });
35
```

Se inicializa la app express considerando el análisis de body en formato JSON y se configura el cliente kafka.

Luego se tiene la función update de la línea 36 a la 79 que sirve para actualizar el estado de los pedidos en la BDD y enviar un mensaje al tópico “test” con el nuevo estado del pedido. Esta está relacionada al servicio de procesamiento de mensajes.

```
36    const update = async (message) => {
37        producer = kafka.producer();
38        await producer.connect();
39
40        const id = JSON.parse(message.value.toString()).id;
41        const name = JSON.parse(message.value.toString()).name;
42        const price = JSON.parse(message.value.toString()).price;
43        var status = (await pool.query(
44            'SELECT status FROM products WHERE id = $1', [id])
45        ).rows[0].status;
```

En el fragmento de código se puede observar la conexión de un producer al servidor kafka. Se extraen los datos del mensaje que tiene como parámetro la función update y se consulta por el estado del mensaje en la BDD.

```
47      if (status === "recibido") {
48          status = "preparando";
49          await pool.query('UPDATE products SET status = $1 WHERE id = $2', [status, id]);
50          await producer.send({
51              topic: "test",
52              messages: [
53                  { value: JSON.stringify({ id, name, price, status }) }
54              ]
55          });
56      }
57      else if (status === "preparando") {
58          status = "enviado";
59          await pool.query('UPDATE products SET status = $1 WHERE id = $2', [status, id]);
60          await producer.send({
61              topic: "test",
62              messages: [
63                  { value: JSON.stringify({ id, name, price, status }) }
64              ]
65          });
66      }
67      else if (status === "enviado") {
68          status = "entregado";
69          await pool.query('UPDATE products SET status = $1 WHERE id = $2', [status, id]);
70          await producer.send({
71              topic: "test",
72              messages: [
73                  { value: JSON.stringify({ id, name, price, status }) }
74              ]
75          });
76      }
77
78      producer.disconnect();
79  }
```

Aquí se actualiza el estado del pedido y luego este cambio es guardado en la BDD y enviado al tópico teft en kafka. El producer se desconecta.

A continuación de la línea 81 a la 147 se tiene la función listening que maneja todo lo relacionado al servicio de notificación en esta estructura de comunicación kafka.

```
81  const listening = async () => {
82      const consumer = kafka.consumer({ groupId: "hola" });
83      const consumer_aux = kafka.consumer({ groupId: "updates" });
84      await consumer.connect();
85      await consumer_aux.connect();
86      await consumer.subscribe({ topic: "delivery", fromBeginning: true });
87      await consumer_aux.subscribe({ topic: "test", fromBeginning: true });
```

Se crean dos consumers: consumer y consumer_aux. Estos son conectados a kafka y se suscriben al tópico delivery y test respectivamente.

```
89      await consumer.run({
90          eachMessage: async ({ topic, partition, message }) => {
91              const id = JSON.parse(message.value.toString()).id;
92              const name = JSON.parse(message.value.toString()).name;
93              const price = JSON.parse(message.value.toString()).price;
94              const status = JSON.parse(message.value.toString()).status;
95              console.log("Pedido id: " + id + ", nombre: " + name+ ", precio: " + price + ", estado: " + status);
96              let email = {
97                  from: 'testmailkafka@gmail.com',
98                  to: 'testmailkafka@gmail.com',
99                  subject: 'Pedido ' + id + ' Producto: ' + name,
100                 text: 'Pedido número' + id +
101                     '\nProducto: ' + name +
102                     '\nEstado de tu pedido: ' + status + '.'
103             };
104             transporter.sendMail(email, (error, info) => {
105                 if (error) {
106                     console.log(error);
107                 }
108                 else {
109                     console.log('Email sent: ' + info.response);
110                 }
111             });
112             let count = 0;
113             const intervalId = setInterval(() => {
114               update(message);
115               count++;
116               if (count === 3) {
117                 clearInterval(intervalId);
118               }
119             }, 1000 * 30);
120         }
121     })
```

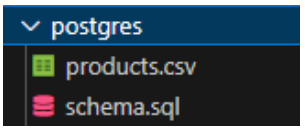
Consumer.run activa al consumidor de kafka para que empiece a recibir y procesar los mensajes. Dentro de esto, se define eachMessage, donde se define cómo van a manejar los mensajes al ser recibidos.

Se extraen las propiedades del mensaje para luego enviar el correo respectivo al pedido y su estado con destinatario al correo de prueba creado para este proyecto. Finalmente se establece la lógica para definir el intervalo de tiempo en el cual se va a actualizar el mensaje, habiendo un update cada 30 segundos hasta 3 veces.

```
150  listening();
151
152  app.listen(3001, () => {
153      console.log("\nServer running on port 3001\n");
154  });
```

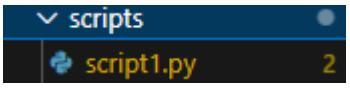
Se inicia la escucha de mensajes en Kafka y se inicia el server express en el puerto 3001.

2.2.3. Postgres



En este directorio solamente se encuentra el dataset a utilizar para la actividad el cual consiste en un dataset de dos atributos: product_name y price, y además se cuenta con el archivo schema.sql que define la estructura de la BDD para la tabla de pedidos, la cual se crea al inicializar el docker postgres.

2.2.4. Scripts



Se cuenta con un script en python que a partir del archivo products.csv en la carpeta postgres inserta peticiones post a la arquitectura kafka para realizar las pruebas del sistema implementado con las filas existentes en el archivo csv.

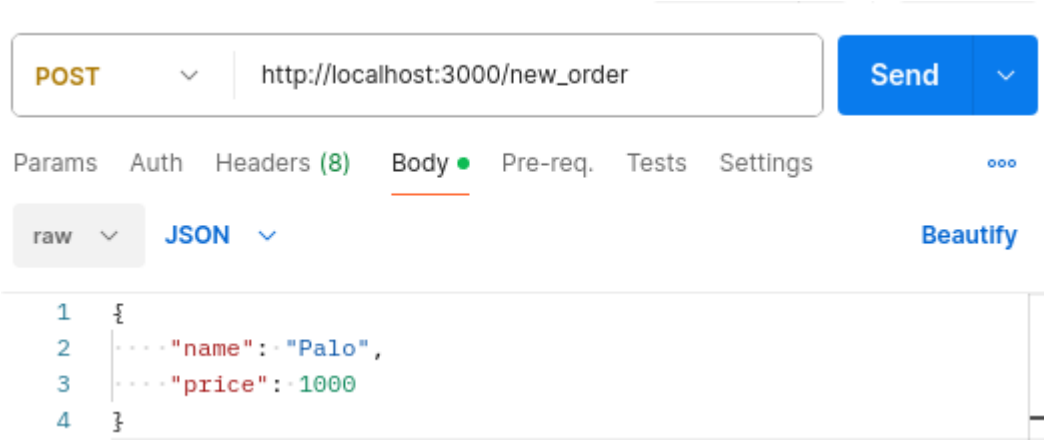
```python
import requests
import pandas as pd

def test_api():
    data = pd.read_csv('../postgres/products.csv')

    for i, row in data.iterrows():
        try:
            response = requests.post('http://localhost:3000/new_order', json={
                'name': row['product_name'],
                'price': row['price']
            })
            print(f'Row {i} - Status: {response.status_code}')
        except Exception as e:
            print(f'Error on row {i}: {str(e)}')

test_api()
```

2.3. Ejecución del sistema

Se realiza una prueba unitaria para mostrar el comportamiento del sistema a partir de una petición http:post hecha en postman. Primeramente se ejecuta el docker-compose para inicializar los contenedores.

```
diego@diegopc:~/Escritorio/tareas/tarea2-SD$ docker-compose up
Creating network "tarea2-sd_default" with the default driver
Creating postgres  ... done
Creating zookeeper ... done
Creating kafka     ... done
Creating producer  ... done
Creating consumer  ... done
Attaching to zookeeper, postgres, kafka, producer, consumer
```

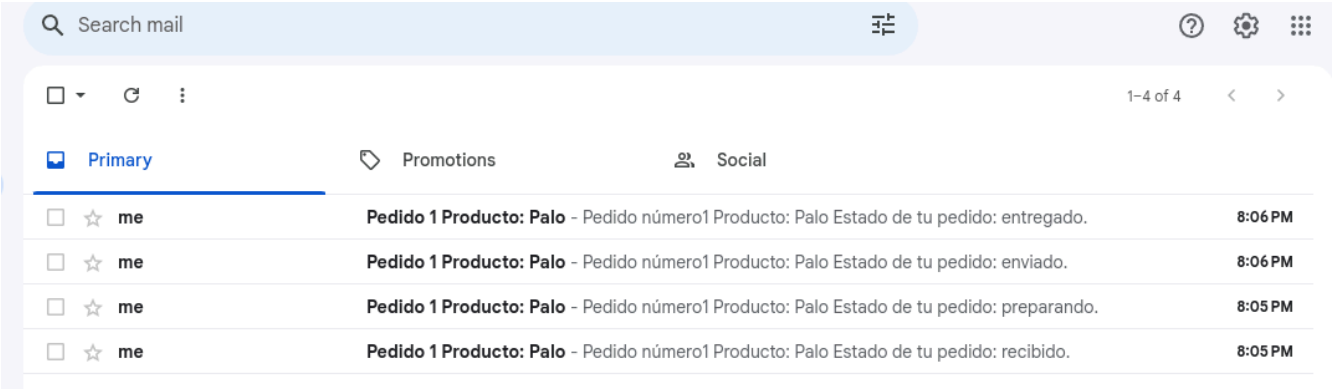
La petición postman se realiza de la siguiente forma con un body JSON:



En la consola se registra el proceso gracias a los console.log() especificados en el código:

```
consumer     | Pedido id: 1, nombre: Palo, precio: 1000, estado: recibido
consumer     | Email sent: 250 2.0.0 OK  1717373109 d2e1a72fcca58-7025ee4fa
8esm1850967b3a.149 - gsmtp
consumer     | Pedido id: 1, nombre: Palo, precio: 1000, estado: preparando
consumer     | Email enviado: 250 2.0.0 OK  1717373138 d9443c01a7336-1f6323
fódadsm51869655ad.221 - gsmtp
consumer     | Pedido id: 1, nombre: Palo, precio: 1000, estado: enviado
consumer     | Email enviado: 250 2.0.0 OK  1717373169 41be03b00d2f7-6c3598
48a32sm4300374a12.69 - gsmtp
consumer     | Pedido id: 1, nombre: Palo, precio: 1000, estado: entregado
consumer     | Email enviado: 250 2.0.0 OK  1717373199 d2e1a72fcca58-702425
df2a4sm4418255b3a.54 - gsmtp
```

Finalmente, en el correo se pueden observar las notificaciones:



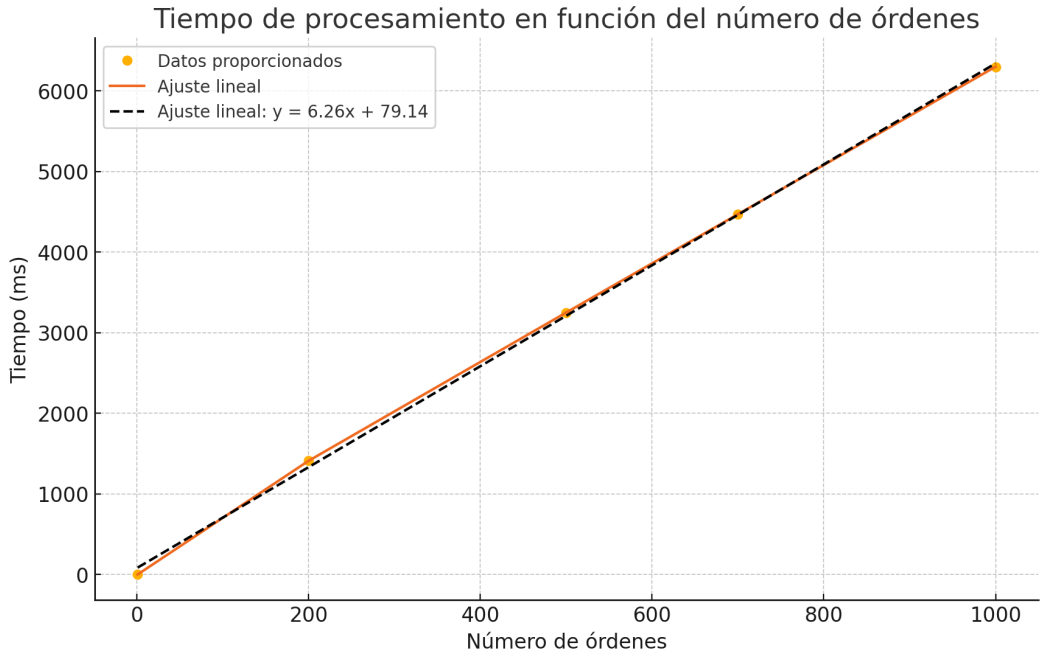
La ejecución de este sistema se puede apreciar en el video explicativo referenciado en este informe.

2.4. Pruebas externas

Se realizaron diferentes pruebas con scripts hechos en python en base al tiempo de procesamiento con distinto número de consumers y particiones.
Prueba para 1 consumidor y 1 partición:

Número de órdenes	Tiempo (ms)
1	0.024
200	1407
500	3250
700	4473
1000	6305

En base a los puntos anteriores [(1,0.025) , (200, 1407),(500, 3250),(700, 4473),(1000,6305)] se genera una gráfica en python, donde el eje X corresponde al número de órdenes y el eje Y corresponde al tiempo medido en milisegundos.

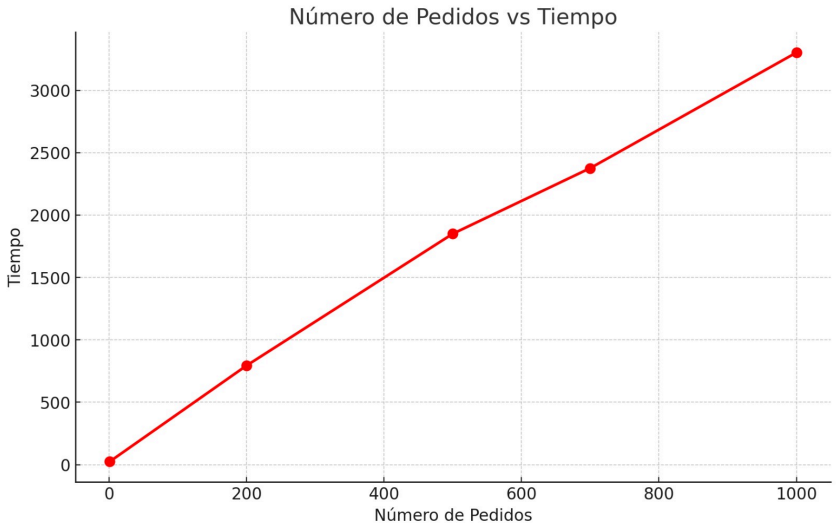


Se puede ver que la cantidad de órdenes a enviar es proporcional al tiempo que el sistema va a tomar en resolverlas todas.

Si aumentamos el número de consumidores (*consumers*) al doble, obtenemos lo siguiente:

Número de órdenes	Tiempo (ms)
1	0.024
200	794
500	1850
700	2376
1000	3302

Esto graficado con python se ve de la siguiente manera:

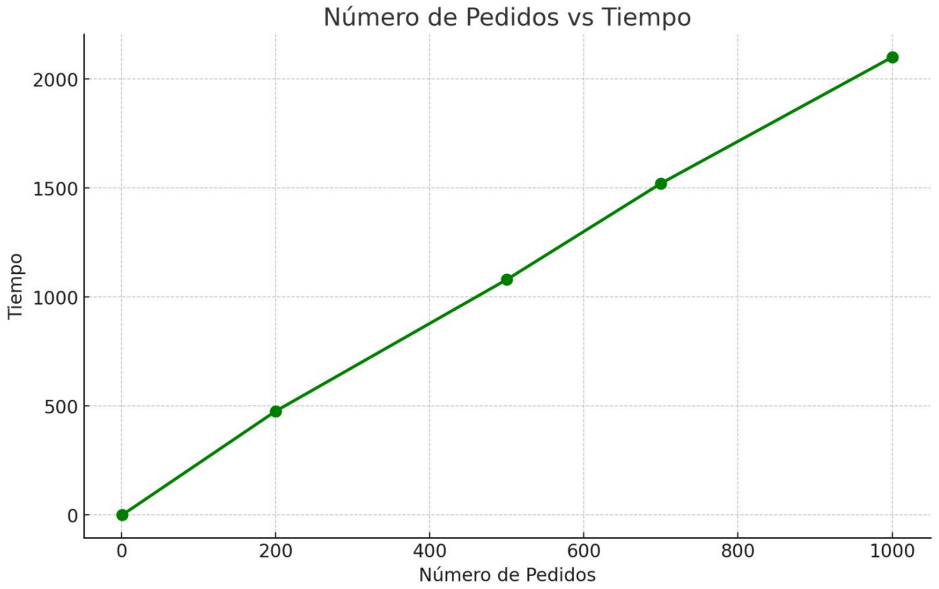


Como se puede observar los tiempos de procesamiento son menores, esto se debe a que más consumidores permiten un mayor paralelismo en el procesamiento de los mensajes.

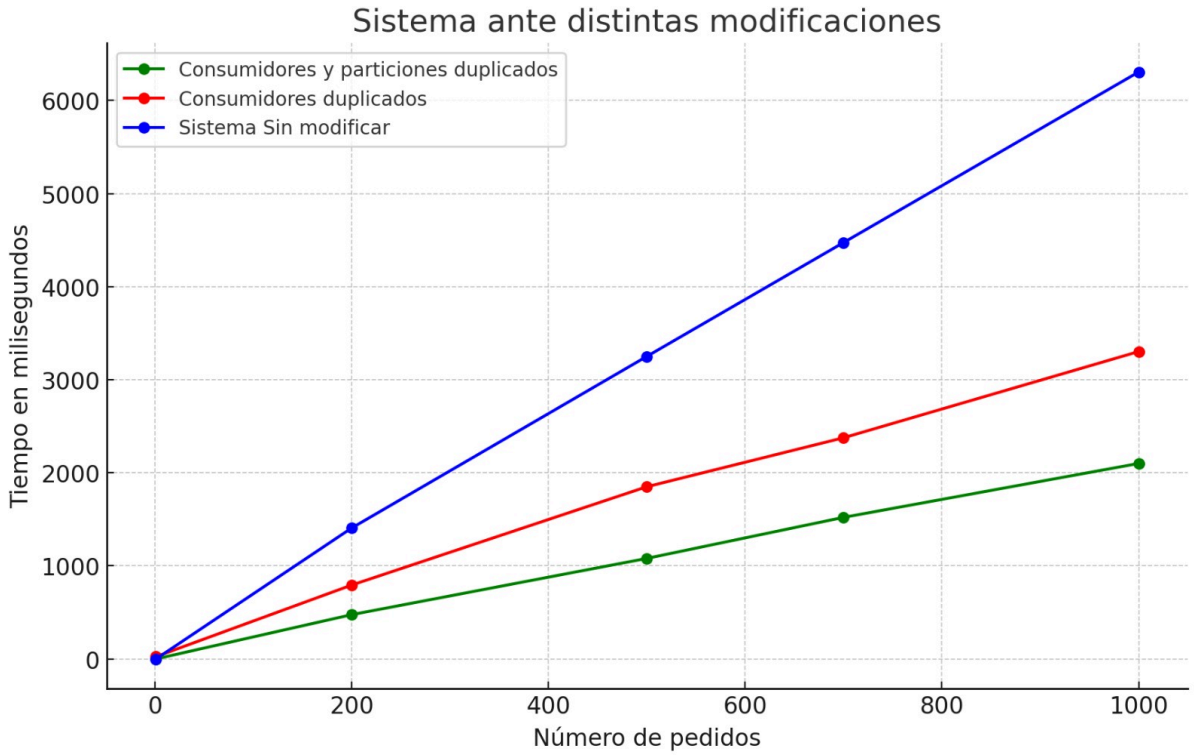
Ahora si mantenemos el doble de consumidores, y además aumentamos el número de particiones de los tópicos se tiene:

Número de órdenes	Tiempo (ms)
1	0.024
200	476
500	1080
700	1521
1000	2100

Esto visto de una manera gráfica:



Como era de esperarse, los tiempos bajaron cerca del 55%, no bajo exactamente a la mitad, debido a que existen distintos factores, como pueden ser la latencia, que afectan directamente a estos resultados.



Al comparar los 3 graficos, se da cuenta que aumentar tanto los consumidores como las particiones resulta en una disminución considerable de tiempo, específicamente casi a la mitad (debido a que aumentamos en doble las particiones y consumidores).

Apache Kafka permite la tolerancia a fallos debido a la existencia de la replicación de las particiones. Cada partición puede tener distintas réplicas, por ende estas permiten tolerancia a fallos en el sistema.
Para manejar la escalabilidad del sistema basado en Apache Kafka, la existencia de los brokers y las particiones permiten una alta escalabilidad en este sistema. Mientras más brokers existen, más se puede distribuir la carga de trabajo existente en el sistema. De la misma forma funciona con las particiones, estas también contribuyen a la distribución de la carga.
A lo largo de la actividad, el uso de la BDD es fundamental para que los datos se gestionen correctamente al pasar por los tres distintos servicios implementados (de procesamiento, notificación e ingreso) ya que así se comparte la información correctamente entre los distintos servicios y también al almacenarse en kafka.

3. ***Conclusión***

En conclusión, se pudo observar la funcionalidad y la arquitectura de un sistema de eventos en tiempo real utilizando Apache Kafka. Se observa la eficiencia y escalabilidad de este ante gran volumen de datos comparándolo a su performance con pocos datos. Se analiza que la inyección de una masividad de datos en el sistema por parte del producer pueden perjudicar la performance del mismo, además de que la cantidad de particiones y brokers puede mejorar su funcionamiento pero al mismo tiempo aumentar los costos.
Siendo más especificos, si tienes más consumidores que particiones, algunos consumidores estarán ociosos. Pero si tienes más particiones que consumidores, pueden procesar varios mensajes en paralelismo. Por lo tanto, si se aumenta el número de consumidores hasta el número de particiones, deberías ver una mejora en el rendimiento. Sin embargo, si ya se tiene un consumidor por partición, añadir más consumidores no mejorará el rendimiento y podría incluso disminuirlo debido a la sobrecarga adicional de coordinar más consumidores. Además, si se aumenta demasiado el número de consumidores, podrías encontrarte con otros problemas, como un aumento en el tiempo de rebalanceo del grupo de consumidores. Por lo tanto, aunque aumentar el número de consumidores puede mejorar el rendimiento, también es importante tener en cuenta estos factores y probar cuidadosamente para encontrar el equilibrio adecuado para el producto final.
También a lo largo del proyecto se aprende a realizar un sistema de notificaciones al correo electrónico como la implementación y conexión de múltiples servicios entre sí. Se valida el funcionamiento del sistema a través de pruebas midiendo así sus métricas.

4. ***Referencias***

Documentacion: https://kafka.js.org/, https://kafka.apache.org/

Repositorio GitHub: https://github.com/RodriYG/tarea2-SD

Video explicativo: https://youtu.be/_n99GCxr6hc