UNIVERSIDAD DIEGO PORTALES

FACULTAD DE INGENIERÍA Y CIENCIAS
ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Sistemas Distribuidos
Tarea 2

Profesor:
Nicolás Hidalgo

Ayudantes:
Nicolás Nuñez
Cristian Villavivencio
Joaquin Fernandez

Estudiantes:
Nicolás Moncada
Dayana Ruminot

Santiago, Chile **30 de Octubre del 2022**

Índice

1. Problema y Solución

El gremio de sopaipilleros de Chile, encargado de establecer políticas legales para la venta en carritos de sopaipillas, ha crecido a un ritmo agigantado. Los anticuados métodos de trabajo para dar soporte a las tareas del gremio requiere de una actualización que implica la utilización de plataformas informáticas capaces de gestionar dichos procesos de la manera más eficiente y escalable.

La gestión de los procesos internos del gremio es compleja. Los miembros que participan en este reciben de manera constante peticiones (escritas a mano) con tareas que deben realizarse. Por ejemplo, realizar la inscripción de un nuevo miembro. Estas tareas tradicionalmente son repartidas según su tipo a los diferentes encargados de gestionar y llevar a cabo las mismas.

Uno de estos procesos nombrados, corresponde a la inclusión de nuevos miembros. Este proceso es engorroso y tardío; requiere dejar una petición formal en el gremio, el cual puede ser resulto en cuestión de meses. Esta petición viaja a la dirección encargar de procesar y evaluar nuevos miembros, los cuales se fijan en los antecedentes de los dueños de los carritos postulantes, para as ı aprobar una lista de nuevos miembros. Esta lista es mostrada de manera periódica, una vez al mes. Este proceso se puede acelerar pagando una comisión, llamado Inscripción Premium. Lo que se busca es automatizar este proceso utilizando sistemas informáticos.

Por otro lado, los maestros sopaipilleros, poseen carritos modernos (es una de las condiciones para ser parte del gremio).

Estos carritos poseen sistemas inteligentes con internet y GPS. Además, poseen un sistema capaz de registrar ventas y posteriormente ejecutar código, sin embargo, el gremio no puede aprovechar esta ventaja. Se busca poder aprovechar estos sistemas, utilizando sistemas informáticos automatizados que permitan:

- Calcular estadísticas sobre ventas (cantidad de ventas por día y clientes frecuentes).
- Tener la posición geográfica en tiempo real de cada sopaipillero. Además se busca alertar eventos extraños.
- Preparar la reposición del stock de manera automática, al momento de tener menos de 20 de masas de sopaipillas.

La solución propuesta para este problema es tener un **Servidor** principal que pueda recibir peticiones, a través de un CRUD o algún otro medio que permita recibir la información. Este será el corazón del gremio de Sopaipilleros. Al mismo tiempo, se debe tener **Kafka** Levantar un broker de Kafka y configurar los tópicos que usted crea que sean necesarios. Cada tópicos debe tener al menos 2 particiones. Paralelamente se debe tener **Servicio de Base de Datos** el cual se encargará de almacenar la información. Por otra parte, el servidor debe ser capaz de recibir distintos tipos de peticiones, en especial estas 3:

- Registro de un nuevo miembro para el gremio:
  El servidor debe ser capaz de recibir el registro de un miembro para el gremio de sopaipilleros. El registro debe contener los campos de: Nombre, Apel lido, Rut,Correo del dueño, Patente carrito y Registro premium. Cuando este registro sea "Premium" debe ser enviado en una nueva partición.
- Registro de una venta:
  Un maestro sopaipillero enviará una petición de registro de venta, que será procesada por distintos servicios a posterior. El registro debe contener los campos de Cliente, Cantidad de Sopaipillas, Hora, Stock restante y Ubicación del carrito.
- Aviso de un agente extraño:
  Cualquier persona podrá denunciar a un carrito prófugo. El registro contendrá simplemente coordenadas y las enviará por el tópico encargado de enviar las ubicaciones de los carritos, pero utilizando una partición distinta.

Procesamiento, en esta sección cada una de las peticiones deberá ser procesada por distintos servidores. Debido a que Kafka genera flujos de datos encolados, ninguno de los clientes de procesamiento (o consumers) deberá correr necesariamente en paralelo al servidor. Se deben crear los siguiente programas:

- Procesamiento de ventas diarias:
  Se ejecuta una vez al día y calcula, para cada maestro sopaipillero, ventas totales, promedio de ventas a clientes (cuantas les vende en promedio a un cliente) y clientes totales. Los valores los imprime por pantalla o los guarda en un archivo de texto.
- Procesamiento de stock para reposición:
  Nunca se deja de ejecutar y constantemente está leyendo las consultas, las guarda por lotes en un arreglo de tamaño 5 para posteriormente leerlas y preparar las entregas. Entrega un aviso por pantalla.
- Procesamiento de ubicación:
  Constantemente está leyendo las entradas y las va reemplazando en tiempo real para mostrar la posición de cada carrito. Si un carrito no envía su posición luego de 1 minuto, desaparece. Las posiciones son simplemente un aviso por pantalla que organiza y asocia a los carritos con sus coordenadas. También mostrar a los registros de los carritos prófugos.

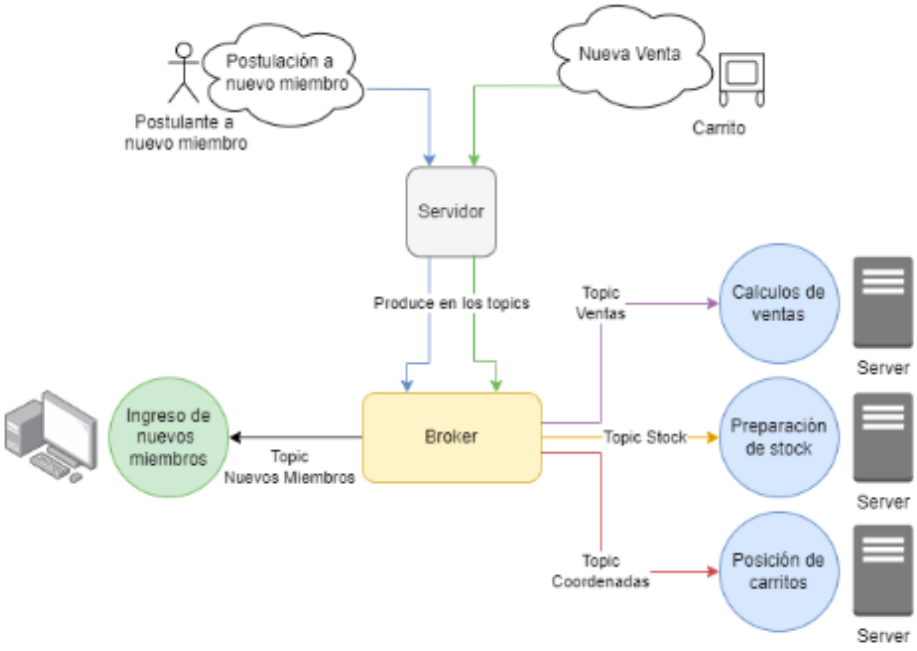

Figura 1: Arquitectura de la solución

2. Códigos

El código se dividirá en las siguientes 4 categorías.

- Docker
- Postgres
- Producer Kafka
- Consumer Kafka

Todos los códigos se pueden encontrar en el repositorio `https://github.com/NicoMonc/Tarea2SD`

2.1. Docker-compose

Descomponiendo parte del docker-compose se puede extraer la siguiente información:

2.1.1. Zookeeper

```yaml
zookeeper:
  image: 'bitnami/zookeeper:3.8.0'
  restart: always
  environment:
    ALLOW_ANONYMOUS_LOGIN: "yes"
  ports:
    - 2181:2181
```

Figura 2: Configuración Zookeeper

Dentro de docker se utiliza zookeeper para poder coordinar los procesos del sistema distribuido, se comunica mediante el puerto 2181 de entrada y 2181 de salida.

2.1.2. Kafka

```yaml
kafka:
  image: 'bitnami/kafka:2.8.1'
  restart: always
  depends_on:
    - zookeeper
  environment:
    ALLOW_PLAINTEXT_LISTENER: "yes"
    KAFKA_CFG_ZOOKEEPER_CONNECT: "zookeeper:2181"
    #KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE: true
    KAFKAJS_NO_PARTITIONER_WARNING: 1
  ports:
    - 9092:9092
```

Figura 3: Configuración Kafka

Kafka utiliza como dependencias a zookeeper por el puerto de salida 2181, siendo este un proceso interno que coordina las actividades de kafka. Por otra parte, los puertos configurados para la entrada y salida son en ambos casos el 9092.

2.1.3. Postgresql

```yaml
database:
  image: postgres:15.0-alpine3.16
  restart : always
  expose:
  - "5432"
  ports:
  - "5432:5432"
  environment:
    - DATABASE_HOST=${DATABASE_HOST}
    - POSTGRES_USER=${POSTGRES_USER}
    - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    - POSTGRES_DB=${POSTGRES_DB}
  volumes:
    - "./sql:/docker-entrypoint-initdb.d"
  command: -p 5432
```

Figura 4: Configuración PostgreSQL

La base de datos utiliza el puerto 5432 de entrada y el puerto 5432 de salida. Además, se le asignan las variables de entorno de host, user, password y DB.

```
DATABASE_HOST=127.0.0.1
POSTGRES_USER=root
POSTGRES_PASSWORD=facil123
POSTGRES_DB=root
```

Figura 5: Archivo .env

2.1.4. Producer Kafka

```
producer:
  build:
    context: ./Producer/
    dockerfile: Dockerfile
  restart: always
  depends_on:
    - kafka
    - database
    - zookeeper
  environment:
    PORT: 3000
    kafkaHost: kafka:9092
  ports:
    - 3000:3000
```

Figura 6: Nodejs Producer

En el producer depende de kafka, database y zookeeper, se utiliza la variable de entorno de kafka. La entrada y salida de datos ocurre en el puerto 3000.

2.1.5. Consumer Kafka

```yaml
consumer:
  build:
    context: ./Consumer/
    dockerfile: Dockerfile
  restart: always
  depends_on:
    - kafka
    - database
    - zookeeper
    - producer
  environment:
    PORT: 3050
    kafkaHost: kafka:9092
  ports:
    - 3050:3000
```

Figura 7: Nodejs Consumer

El consumer tiene asignado el puerto 3050 de salida y 3000 de entrada. Depende de kafka, database, zookeeper y producer, tiene acceso a kafka como variable de entorno en el puerto 3050 de salida y 9092 de entrada.

2.2. PostgreSQL

```sql
DROP DATABASE IF EXISTS sopaipillas;
CREATE DATABASE sopaipillas;
\c sopaipillas;
```

Figura 8: Creación base de datos

```sql
CREATE TABLE miembros (
    rut int NOT NULL,
    nombre varchar NOT NULL,
    apellido varchar NOT NULL,
    email varchar NOT NULL,
    patenteCar varchar NOT NULL,
    premium varchar NOT NULL,
    PRIMARY KEY (rut)
);
```

Figura 9: Tabla miembros

```sql
CREATE TABLE carritos (
    patente varchar NOT NULL,
    ubicacion int NOT NULL,
    stock int NOT NULL,
    autorepo int NOT NULL,
    PRIMARY KEY (patente)
);
```

Figura 10: Tabla carritos

En el enunciado se menciona que los carritos son inteligentes y se auto rellenan cuando el stock es menor a 20, por lo cual se agregó un valor autorepo en la tabla de registro en carritos.

```sql
CREATE TABLE reportes(
    id SERIAL NOT NULL,
    patente varchar NOT NULL,
    ubi int NOT NULL,
    PRIMARY KEY (id)
);
```

Figura 11: Tabla reportes

2.2.1. Nodejs

Primero identificaremos las secciones compartidas entre producer y consumer.

```javascript
//Conexión base de datos
const connectionString = 'postgresql://root:facil123@database:5432/sopaipillas'

const client = new Client({
  connectionString,
})
client.connect()

//"Configuracion librerias"
const app = express()
const port = process.env.PORT
app.use(cors());
app.use(express.json());

//Configuración Kafka
const kafka = new Kafka({
  brokers: [process.env.kafkaHost]
});
```

Figura 12: Inicio de librerias

En esta sección se realiza la conexión con postgres gracias a la librería pg, posterior a eso se configura el puerto de los nodejs y por ultimo se levanta el broker de kafka.

```
FROM node:16.18-alpine3.16

WORKDIR /home/node

COPY . .

RUN npm install

CMD [ "npm", "start" ]
```

Figura 13: Docker file

Docker file encargado de ejecutar la imagen de node - docker y configurar su directorio de trabajo.

2.2.2. Consumer

```
const rule = new schedule.RecurrenceRule();
rule.hour = 23;
rule.minute = 59;

var ventaslistado=[]

const findia = schedule.scheduleJob(rule, async ()=>{
    const ventas = await client.query('SELECT rut.miembros,SUM(cant.ventas) as ventaTotal,'+
    'AVG(cant.ventas) as promedio, count(cant.ventas) as clientes FROM miembros,'+
    'ventas where patenteCar.miembros=patente.ventas')
    ventaslistado=ventas
});
```

Figura 14: Uso de libreria node-schedule

Gracias a la librería node-schedule podemos programar la ejecución de una sección de código, donde en este caso configuraremos una query para el fin del día, guardando el rut del miembro, sus ventas totales, ventas promedio y cantidad de clientes.
Este proceso de ejecutará diariamente a las 23:59.

```
const consumereportes = kafka.consumer({ groupId: 'reportes', fromBeginning: true });
await consumereportes.connect();
await consumereportes.subscribe({ topic: 'reportes' });
await consumereportes.run({
    eachMessage:async ({topic,partition,message}) =>{
        if(message.value){
            var data= JSON.parse(message.value.toString())
            console.log(data)
            arrreportes.push(data)
        }
    },
})
```

Figura 15: Estructura base Kafka

El código anterior será la base de todos los procesos del consumidor, como primer paso le entregamos un grupo, el cual es recomendable que sea de consumidores para el mismo tópico, dado a que en la mayoría de documentaciones

se mencionaban problemas al registrar consumidores de distintos tópicos en el mismo grupo.
Posterior a esto se hace la conexión y se suscribe a un tópico, donde en nuestro caso existirán los siguientes:

- reportes
- ventas
- miembros

En la sección run se agregará la función eachMessage, donde por cada mensaje entrante ejecutará el código que se encuentra en su interior.

2.2.3. Producer

```
await producer.connect();
    await producer.send({
        topic: 'reportes',
        messages: [{value: JSON.stringify(req.body)}]
    })
    await producer.disconnect().then(
        res.status(200).json({
          "Estado":"Completado"
      })
    )
```

Figura 16: Estructura envío de información a Kafka

El tratado de información por cada caso de la api es distinto pero lo que todos comparten es la forma de enviar información a un tópico de kafka.
Se hará la conexión y se enviará un mensaje a un tópico en especifico. Todos los mensajes fueron enviados gracias a JSON.stringify.

3. Explicación

```
var patente=req.body.patente
var ubicacion= req.body.ubicacion
req.body.situacion= "actualizacion"
const updateCarrito = 'UPDATE carritos SET ubicacion = $1 WHERE patente= $2'
await client.query(updateCarrito, [ubicacion, patente])
```

Figura 17: Implementación ubicación

Para la implementación de nuestro código hemos planteado la ubicación de los carritos como un arreglo unidimensional, o para ser aún mas claros, como puestos de cualquier feria, donde existirán puestos contiguos desde el puesto 1 al infinito (no se hicieron limitaciones en el código para no perder tiempo en aquello y dedicar mas tiempo a kafka).
Lo anterior debido a que en un inicio de planteó cada ubicación con coordenadas para un sistema x-y-z, pero hacía el sistema mucho mas complicado de trabajar, tanto por la parte de código como manejo de la base de datos.

4. Preguntas

1. ¿Cómo Kafka puede escalar tanto vertical como horizontalmente?

   Dado que Kafka se caracteriza por administrar flujos de datos de varias fuentes y distribuirlos a diversos usuarios necesita un procesamiento de alto rendimiento y idealmente una gran cantidad de memoria.

   Gracias a que es un sistema distribuido, permite realizar escalamiento horizontal y absorber picos de carga que se puedan generar en un sistema sin perder el rendimiento, al mismo tiempo consiguiendo tener una baja latencia entre todos sus componentes. Además, los consumidores y productores están completamente desacoplados, no existe relación directa entre ellos y esto nos permite escalar tanto a los productores como los consumidores según se necesite o para el caso de uso que se le quiera dar, permitiendo así un escalamiento vertical.

   Para el escalamiento vertical Kafka se centra en aumentar la capacidad para guardar datos. Por otra parte, para el escalamiento horizontal Kafka se centra en agregar brokers para dividir la carga, esta forma la latencia no aumenta y el rendimiento es parejo desde todas partes.

2. ¿Qué caracteríısticas puede observar de Kafka como sistema distribuido? ¿Cómo se reflejan esas propiedades en la arquitectura de Kafka?

   La principal característica de kafka es que fue creada para el uso de Linkedin, es una plataforma de código abierto, surge como una cola de mensajería distribuida basada en el modelo productor-consumidor, con la finalidad de mejorar la comunicación y la interacción entre sistemas de distintos tipos, lo cual es a lo que apunta un sistema distribuido. La idea principal es dar el confort y confianza a los usuarios para que usen linkedin desde cualquier parte del planeta y que sin importar el lugar el funcionamiento de esta sea el mismo, siempre eficaz, siempre rápido y sin segregación del lugar en el cual se ubica el usuario. Kafka proporciona una plataforma basada en componentes distribuidos que permite publicar información de distintas fuentes y ponerla a disposición de otras aplicaciones que requieran de dicha información, esta manera se consigue tener centralizada la información de distintas fuentes que en un principio podría estar dispersa, y esta información se almacena de una forma tolerante a fallos y con una alta disponibilidad.

   Apache Kafka tiene una arquitectura distribuida capaz de manejar mensajes entrantes con mayor volumen y velocidad. Como resultado, Kafka es altamente escalable sin ningún impacto en el tiempo de inactividad.

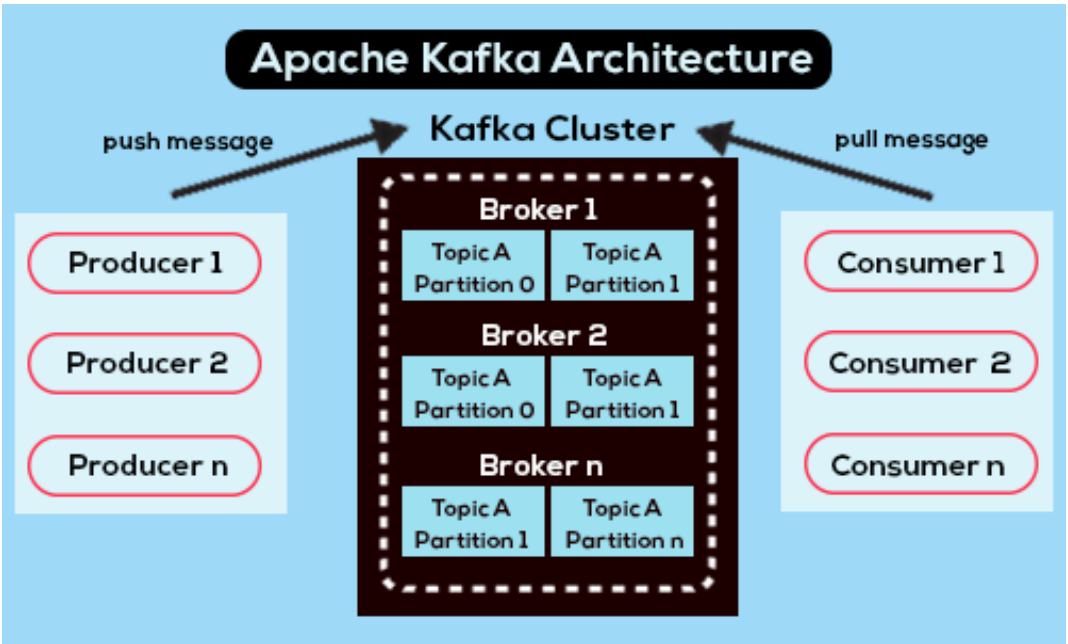

Figura 18: Arquitectura de Kafka

De la imagen anterior se puede describir el funcionamiento de la arquitectura, y el flujo de datos en Kafka.

En primera instacia se tienen los Producer quienes son los que envían el mensaje para que luego el intermediario lidie con el. Los mensajes en Kafka se envían en forma de lotes, conocidos como lotes de registro. Los productores acumulan mensajes en la memoria y los envían en lotes después de acumular una cantidad fija de mensajes o antes de que haya transcurrido un período de tiempo limitado de latencia fija.
Los nodos en un clúster deben enviar mensajes llamados mensajes Heartbeat a ZooKeeper para mantener informado a ZooKeeper de que están activos. En Kafka, el consumidor debe enviar solicitudes a los intermediarios indicando las particiones que desea consumir. Se requiere que el consumidor especifique su compensación en la solicitud y recibe una parte del registro que comienza desde la posición de compensación del intermediario. Los consumidores deben enviar solicitudes a los corredores para indicar que están listos para consumir los datos. Un sistema basado en extracción garantiza que el consumidor no se sienta abrumado con los mensajes y pueda retrasarse y ponerse al día cuando pueda. Un sistema basado en extracción también puede permitir un procesamiento por lotes agresivo de los datos enviados al consumidor, ya que el consumidor extraerá todos los mensajes disponibles después de su posición actual en el registro. De esta manera, el procesamiento por lotes se realiza sin ninguna latencia innecesaria. En el caso de la imagen anterior se puede señalar que los Breokers hacen el trabajo de ser el intermediario entre el productor y el consumidor.

5. Bibliografía

- **Vídeo del funcionamiento** https://youtu.be/xeLzVBfff-U
- **Repositorio** https://github.com/NicoMonc/Tarea2SD
- **¿Qué es apache Kafka?** https://www.tibco.com/es/reference-center/what-is-apache-kafka
- **¿Qué es y cómo funciona Apache Kafka?** https://www.youtube.com/watch?v=UNMML-YLAxs
- **Orquestación del flujo de trabajo basado en eventos de Apache Kafka** https://www.projectpro.io/article/apache-kafka-architecture-/442