Universidad Diego Portales

Tarea N2: Kafka:evento

Profesor: Nicolas Hidalgo

Ayudantes:
Joaquín Fernandez
Nicolas Nuñez
Cristian Villavicencio

Integrantes: Abel Baulloza Almeida
Diego Carrillo Ormazabal

FECHA Octubre 2022

Índice

1. Introducción

En el presente informe, se presentara un problema y una solución mediante el uso de sistemas distribuidos. Para este caso se solicita utilizar Kafka, el cual mediante un cliente y un servidor, en conjunto de una base de datos podremos generar un entorno que permita llevar a cabo una solución.

2. Problema y solución

El caso, es un gremio de sopaipilleros que reciben de manera constante peticiones (escritas a mano) con tareas que deben realizarse. Estas tareas son demasiado engorrosas de efectuar mediante el uso de anticuados métodos de trabajo, por lo tanto el gremio debe recurrir a la utilización de plataformas informáticas capaces de gestionar dichos procesos de la manera mas eficiente y escalable.

La estructura que se busca cumplir en este sistema será simulada mediante el siguiente esquema:

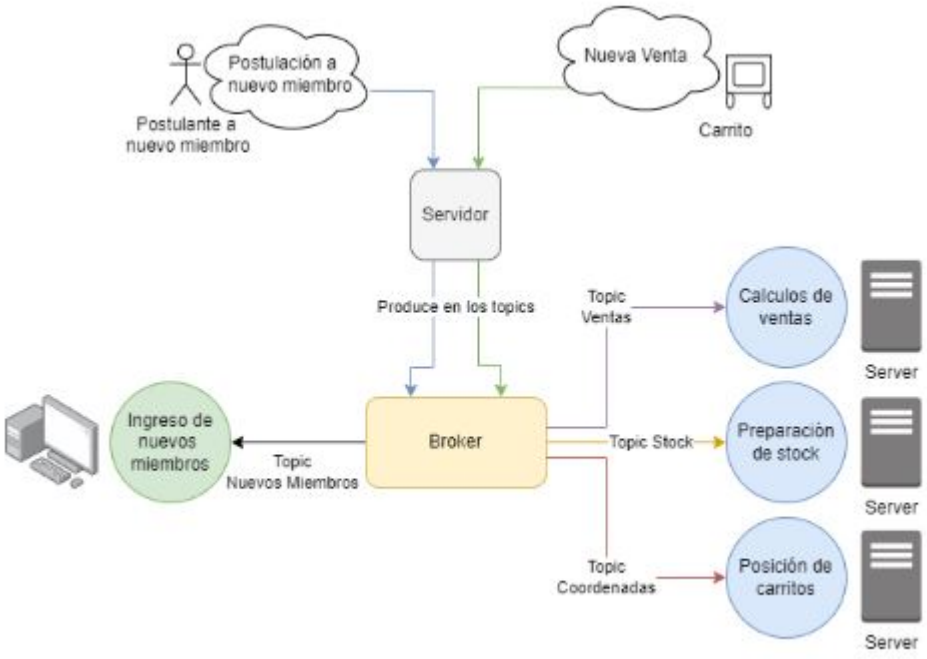

Figura 1: Estructura del Sistema

3. Explicación de módulos de código

A continuación se procede a explicar cada módulo del código desarrollado para dar solución a la problemática planteada. Siendo la base del código, el archivo docker-compose.

3.1. Docker-Compose

Para poder levantar este servidor que funcionaba con node y con Kafka, fue necesario crear el archivo docker-compose, construido con la imagen de bitnami. Dentro de este docker, se generaron varias imágenes, las cuales se distribuyeron de a pares para poder implementar un producer & consumer, que es como trabaja kafka.

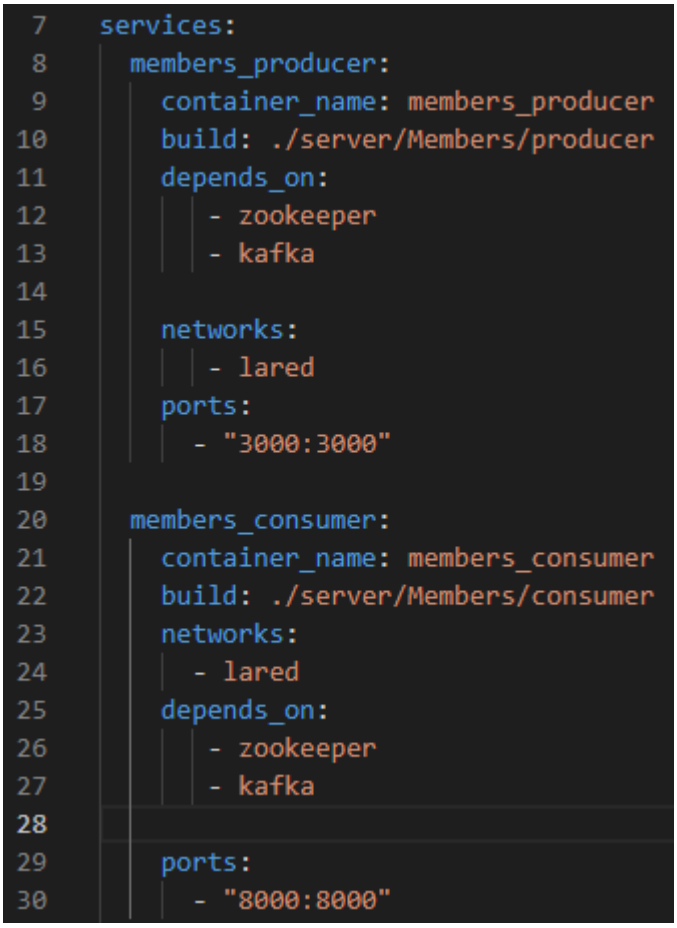

Figura 2: Ejemplo de Docker

Como se muestra en la imagen anterior, este proceso fue creado para el caso de member, asignándole un consumer y un producer, ambos trabajando en puertos distintos. Cada producer se trabajó en los puertos $300X$: 3000, mientras que el puerto correspondiente para producer fue $800X$: 8000; Siendo X variable desde 0 a 3, para evitar colisión para los demás servicios.

Para cada servicio se creó un apartado similar al de la imagen y estos son los servicios creados:

- Miembros
- Ventas
- Stock
- Ubicación

Quedando de la siguiente manera los archivos:

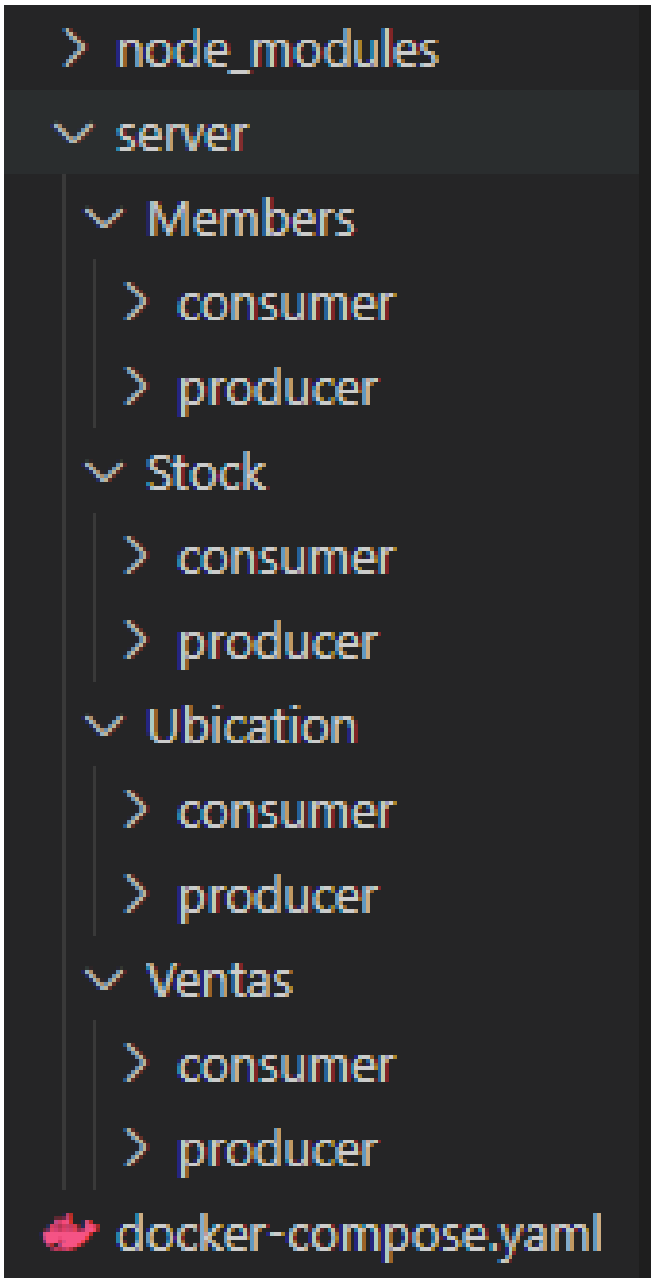

Figura 3: Archivos

Para finalizar todo lo que corresponde a la estructura de este archivo, es necesario mostrar como es que ase

agrega Kafka y zookeeper dentro del compose, ya que, estos dos trabajan en conjunto y zookeeper de por si es una dependencia de kafka.

Primero, está el zookeeper, el cual es sacado de una imagen de bitnami y se verán, a su vez, los puertos que este tiene asociado para su funcionamiento.

```
109   zookeeper:
110     container_name: elzookeeper
111     image: 'bitnami/zookeeper:latest'
112     environment:
113       ALLOW_ANONYMOUS_LOGIN: "yes"
114     networks:
115       - lared
116     ports:
117       - 2181:2181
118       - 2888:2888
119       - 3888:3888
120
```

Figura 4: Docker : zookeeper

Finalizando el docker-compose, esta lo mas importante, Kafka, que es el centro de este trabajo y lo que permitió que se realizaran todas las peticiones y puntos a desarrollar en el presente trabajo.

```
kafka:
  container_name: elkafka
  image: 'bitnami/kafka:latest'
  networks:
   - lared
  depends_on:
    - zookeeper
  environment:
    KAFKA_ADVERTISED_HOST_NAME: "kafka"
    KAFKA_ADVERTISED_PORT: "9092"
    ALLOW_PLAINTEXT_LISTENER: "yes"
    KAFKA_CFG_ZOOKEEPER_CONNECT: "zookeeper:2181"
    KAFKA_CFG_NUM_PARTITIONS: "2"

  ports:
    - 9092:9092
```

Figura 5: Docker : Kafka

Dentro de esto, se puede ver como a diferencia de otros servicios, contiene una gran cantidad de entornos o ambientes, donde se le asignan cosas como:

- Nombres de host
- Puertos para anuncios
- Conexión con zookeeper
- Asignación numero de particiones

3.2. Producer Estructura Básica

Como se trabajó con varios producer, fue posible determinar una estructura "basica" para el uso de estos. Lo cual empieza llamando al servidor express() que hace referencia al uso de node.js, luego se le asignan los puertos que serán utilizados, siendo estos 3000 para los producer.

Por otro lado, se crea una variable asociada kafka que permite establecer la conexión.

Figura 6: Producer

Luego, es necesario realizar la conexión del producer y conectarse.

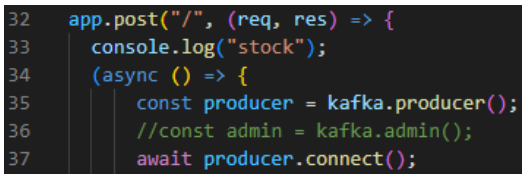

Figura 7: Producer

Posterior a esto, se crea por ejemplo un diccionario que contenga la información a publicar por el producer. Para finalizar la estructura base, se procede a publicar la información estableciendo el topic por el cual se publicará la data. Esto se realiza de la siguiente manera:

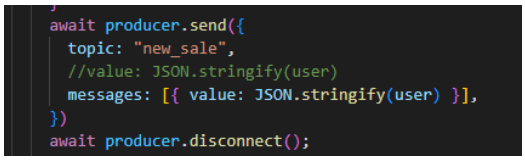

Figura 8: Producer

3.3. Estructura Consumer

Al igual que con los producer, en esta actividad se trabajó con varios consumer los cuales comparten una estructura 'básica´ para su uso.
Esta estructura empieza similar al producer, estableciendo variables globales para el correcto funcionamiento del servidor y el uso de la librería kafkajs.

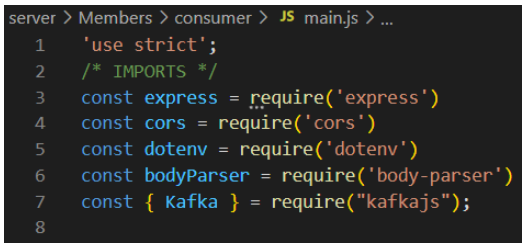

Figura 9: Consumer

Luego, se establece conexión a un broker en específico, llamado kafka:9092, ya establecida la conexión se crea un objeto consumer a través del cliente de kafka conectado al broker con un Consumer Group asociado llamado groupId.

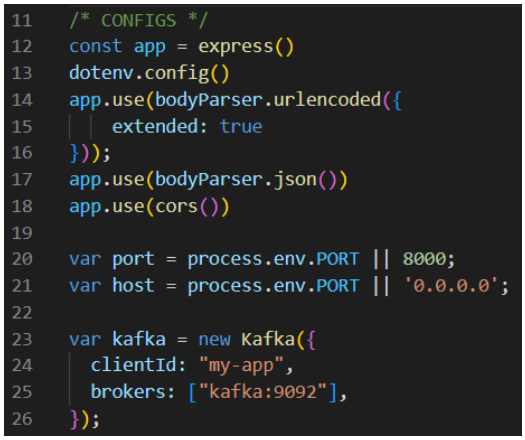

Figura 10: Consumer

Después, el objeto consumer se conecta al broker y se le da la orden que se subscriba a un tópic específico del broker. Finalmente, se necesita del evento run para ejecutar el consumer y poder consumir la información publicada por el producer.

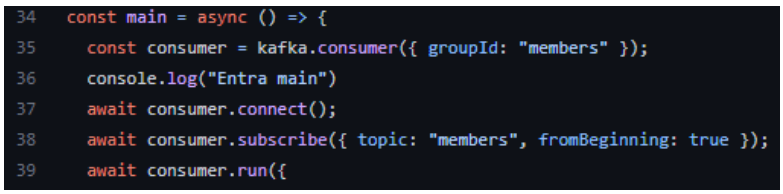

Figura 11: Consumer

4. Peticiones y Procesamientos

En el siguiente apartado, es donde se explicaran las peticiones o puntos solicitados en la tarea. Estos son desarrollados en el producer, donde la información es recolectada. Si la información en este instante esta "mal" recolectada, el desarrollo de toda la actividad se vería afectada. Por eso, esto es la base de todo el trabajo.

Es posible que dentro de este apartado no esté el código completo para cada caso, solo estará lo fundamental. Ya que, la estructura correspondiente al producer esta en otra sección por separado.

4.1. Registro de un nuevo miembro para el gremio

Para la primera petición, la información recibida desde una petición POST "/new_member" se guarda en un diccionario llamado **member**, el cual almacena el "name, lastname, dni, mail, patente, premium", luego se procede mediante la llave premium del diccionario a decidir si esta petición corresponde a un miembro premium o no premium.

```
34  app.post("/new_member", (req, res) => {
35    (async () => {
36        const producer = kafka.producer();
37        //const admin = kafka.admin();
38        await producer.connect();
39        const { name, lastname, dni, mail, patente, premium } = req.body;
40        let member = {
41          name: name,
42          lastname: lastname,
43          dni: dni,
44          mail: mail,
45          patente: patente,
46          premium: premium,
47        }
48        value = JSON.stringify(member);
```

Figura 12: Petición 1: Registro de un nuevo miembro del gremio.

Luego, mediante una condición if se pregunta, si la llave premium del diccionario contiene un valor 1 entonces es un posible miembro premium y se procede a establecer mediante un diccionario llamado **topicMessages** el topic y partición en donde publicara la información el producer. Siendo el topic " members" y en la partición 1 para miembros premium y si la llave premium del diccionario contiene un valor 0 entonces mediante el mismo diccionario topicMessages se establece el topic members y partición 0 para miembros no premium. Para concertar y realizar la publicación de la información por parte del producer se ejecuta el evento **sendBatch** y finalmente se pide desconectar el producer con el evento **disconnect**.

```
49        if(member["premium"] == 1){
50          cpremium.push(value);
51          const topicMessages = [
52            {
53              topic: 'members',
54              partition : 1,
55              messages: [{value: JSON.stringify(member), partition: 1}]
56            },
57          ]
58            await producer.sendBatch({ topicMessages })
59          }else{
60            cpnopremium.push(value);
61            const topicMessages = [
62              {
63                topic: 'members',
64                messages: [{value: JSON.stringify(member), partition: 0}]
65              },
66            ]
67            await producer.sendBatch({ topicMessages })
68          }
69
70          await producer.disconnect();
```

Figura 13: Petición 1: Registro de un nuevo miembro del gremio.

4.1.1. Procesamiento para un nuevo miembro para el gremio

Ahora, se procede a validar en el lado del consumer si el nuevo miembro premium o no premium sera aceptado en el gremio. Esto se realiza primero que todo conectando y subscribiendo al consumer en el topic members para poder consumir la información publicada por el producer.
El consumer al ya estar subscrito y dependiendo de si la petición del nuevo miembro corresponde a un miembro premium o no ingresa a la partición correspondiente. Al momento de entrar en la partición 1 por ejemplo, se realiza una validación muy simple, mediante una variable random llamada **entra**, la cual por cada petición se le atribuirá un numero entero entre 0 y 6, si el valor es menor a 4 entonces el miembro premium sera aceptado y agregado a un array llamado **miembros_premium**, si es mayor a 4 entonces sera rechazado.

```
29  var value = null
30  var members = [];
31  var entra = 0;
32  var miembros_premium = []
33  var miembros_nopremium = []
34  const main = async () => {
35    const consumer = kafka.consumer({ groupId: "members" });
36    console.log("Entra main")
37    await consumer.connect();
38    await consumer.subscribe({ topic: "members", fromBeginning: true });
39    await consumer.run({
40      eachMessage: async ({ topic, partition, message }) => {
41        entra = Math.floor(Math.random()*7)
42        console.log(entra)
43        console.log(partition)
44        var miembro = JSON.parse(message.value.toString());
45        if(partition == 1)
46        {
47          if(entra < 4){
48            console.log('Miembro premium validado')
49            console.log(miembro)
50            miembros_premium.push(miembro)
51          }else{
52            console.log('Miembro premium no validado')
53            console.log(miembro)
54          }
55        }
```

Figura 14: Procesamiento de nuevos miembros para el gremio: Validación para aceptar a un nuevo miembro premium o no premium en el gremio

Lo mismo ocurre si la petición corresponde a un miembro no premium, ingresando a la partición 0 y pregun-

tando si la variable entrar es menor a 5, si lo es entonces el miembro es aceptado y agregado a un array llamado **miembros_nopremium**, si no es simplemente rechazado.

Por comodidad, al momento de realizar una petición GET '/listmemberpre" se imprimir a por pantalla todos los miembros premium validados, de igual forma para los miembros no premium pero con una petición get '/listmembernopre ".

```
56        else if(partition == 0)
57        {
58          if(entra < 5){
59            console.log('Miembro no premium validado')
60            console.log(miembro)
61            console.log("PARTICION:", partition)
62            miembros_nopremium.push(miembro)
63          }else{
64            console.log('Miembro no premium no validado')
65            console.log(miembro)
66          }
67        }
68      },
69    })
70  }
71
72  app.get('/listmemberpre', (req,res) => {
73    console.log(miembros_premium)
74    res.json(miembros_premium)
75  })
76
77  app.get('/listmembernopre', (req,res) => {
78    console.log(miembros_nopremium)
79    res.json(miembros_nopremium)
80  })
```

Figura 15: Procesamiento de nuevos miembros para el gremio: Validación para aceptar a un nuevo miembro premium o no premium en el gremio

4.2. Registro de una venta

Para este caso, se pide al igual que el caso anterior realizar una petición post en la que se registren ventas, la actividad pide que los parámetros sean:

- Cliente
- Stock Restante
- Hora
- Stock Restante
- Ubicación del carrito
- patente_carro

Estos son enviados a través del body de la petición post mediante el software de insomnia. Por comodidad se agrega el parámetro 'patente_carro´ para el procesamiento. Al igual que para el registro de un nuevo miembro, se crea una variable llamado **topicMessages** que contiene el topic donde se publicará, la data y la partición específica del topic, luego de esto se realiza la acción de publicar gracias al evento **sendBatch**.

```
30  app.post("/sales", (req, res) => {
31    console.log("sales");
32    (async () => {
33        const producer = kafka.producer();
34        await producer.connect();
35        const { client, count_sopaipillas, hora, stock, ubicacion, patente_carro } = req.body;
36
37        let sale = {
38          client: client,
39          count_sopaipillas: count_sopaipillas,
40          hora: hora,
41          stock: stock,
42          ubicacion: ubicacion,
43          patente_carro:patente_carro
44        }
```

Figura 16: Petición 2: Registro de una venta realizada por un carrito.

En este registro ,como se explicó, el producer publica información en dos topic, uno es 'sales´ el cual es topic donde se realizará el procesamiento de ventas diarias y el topic 'stock´ es donde se realizará el procesamiento de stock para reposición, ambos procesamientos serán explicados mas adelante.

```
48          const topicMessages = [
49            {
50              // Stock debe estar leyendo constantes consultas
51              topic: 'sales',
52              messages: [{value: JSON.stringify(sale)}]
53            },
54            {
55              // Stock debe estar leyendo constantes consultas
56              topic: 'stock',
57              messages: [{value: JSON.stringify(sale)}]
58            }
59          ]
60          await producer.sendBatch({topicMessages})
61          console.log("Envie", JSON.stringify(sale))
62
63          await producer.disconnect();
64          res.json(sale);
65          console.log('Venta registrada')
66      })();
67    });
```

Figura 17: Petición 2: Registro de una venta realizada por un carrito.

4.2.1. Procesamiento 2: Stock para reposición

Primero se explicara el procesamiento realizado a través de la información en el topic stock. En este caso, se solicitaba poder registrar ventas de un carrito, en donde existían ciertas condiciones. Publicada la data perteneciente al topic 'stock´, se procede a consumir la data a través del consumer, este se subscribe al topic 'stock´ y mediante el evento eachMessage de manera asíncrona se consume la data, se trabaja de tal forma que la información sea accesible de manera fácil, esto se realiza aplicando la función **value.toString()** y luego la función **parse()**. Ahora, con la información accesible se empieza el procesamiento de la información. Primero existe el caso en donde se realiza una compra pero para una cantidad de productos menor a 20, lo que significará que necesitará realizar una reposición de stock más adelante. Es por eso, que cuando se cumpla esta condición, la información debe de guardarse hasta una máximo de 5 compras. Lo anterior, con el fin de dejar un supuesto registro de reposición de stock. Por otro lado, el usuario podría realizar compras de más de 20 productos, pero para este caso, no se guardarán para realizar reposición, pero si se registrará la información de venta para realizar el procesamiento de ventas totales, clientes totales y promedio de ventas por clientes para cada carrito, desarrollado a través de la información contenida en el topic sales.

```
32  const main = async () => {
33    console.log("Entra stock")
34    await consumer.connect();
35    await consumer.subscribe({ topic: "stock", fromBeginning: true });
36    console.log("producer");
37
38    await consumer.run({
39      eachMessage: async ({ topic, partition, message }) => {
40        value = message.value
41        var algo = JSON.parse(message.value.toString());
42        console.log(algo)
43        json = JSON.parse(value)
44
45        if(json["stock"] <= 20){
46          stock.push(json)
47          if(stock.length==5){
48            console.log('Hay 5 miembros registrados con stock para reposicionar')
49            console.log(stock)
50            stock = [] //stock.lenght=0
51          }
52        }
53      },
54    })
```

Figura 18: Procesamiento 2: Stock para reposición.

De forma mas detallada, lo que se realiza en el código es lo siguiente. El consumer se conecta y subscribe al topic stock, por cada petición de venta recibida se revisa mediante una condicion if si el stock registrado en el venta es menor a 20, si lo es entonces corresponde a una venta con un nivel de stock inferior y por lo tanto se guarda en un array llamado **stock**, este array al momento de que contenga 5 ventas con un nivel inferior de stock se imprimirá en pantalla junto a una alerta.

4.2.2. Procesamiento 1: Ventas diarias

Ahora, se explica el procesamiento 1 llamado ventas diarias. Este se realiza con la información consumida del topic sales.
Como al igual que en el procesamiento recien explicado, el consumer se conecta y subscribe pero esta vez al topic sales.

```
const main = async () => {
  console.log("Entra sale")
  await consumer.connect()
  await consumer.subscribe({ topic: "sales", fromBeginning: true });

  console.log("producer");

  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      value = message.value
      console.log({
        value: message.value.toString(),
      })
      json = JSON.parse(value)
      var count = 0;
      var count1 = 0;
      var count2 = 0;
```

Figura 19: Procesamiento 1: Ventas diarias-Ventas totales por cada carrito.

```
//VENTAS TOTALES DE SOPAIPILLAS POR CARRITO
if(ventas_total_por_patente.length == 0){
  var info = {
    patente:json.patente_carro,
    count_sopaipillas:json.count_sopaipillas
  }
  ventas_total_por_patente.push(info)
}else{
  for(var i in ventas_total_por_patente){
    console.log('SUma')
    count++
    if(ventas_total_por_patente[i].patente == json.patente_carro){
      ventas_total_por_patente[i].count_sopaipillas = ventas_total_por_patente[i].count_sopaipillas + json.count_sopaipillas
      break;
    }else if(ventas_total_por_patente.length == count){
      var info2 = {
        patente:json.patente_carro,
        count_sopaipillas:json.count_sopaipillas
      }
      ventas_total_por_patente.push(info2)
      break;
    }
  }
}
```

Figura 20: Procesamiento 1: Ventas diarias-Clientes totales por cada carrito.

Para las ventas totales realizadas por cada carrito, se procede primero que todo a guardar la primera venta realizada por un carrito en un array llamado **ventas_total_por_patente** en un diccionario llamado **???** el cual contiene la patente del carrito y la cantidad de sopaipillas vendidas. Luego de la primera venta, se recorre el array de diccionarios preguntando si la venta nueva fue realizada por el mismo carrito referenciado con la patente, si es así entonces la cantidad de sopaipillas se suma al total ya almacenado en el diccionario, caso contrario se espera hasta que se recorra todo el array gracias a la variable count, significando que la patente es nueva y por lo tanto será registrada en el array de diccionarios, esto se ejecuta por cada venta obteniendo así el total de ventas realizadas por cada carrito.

```
80        //CLIENTES TOTALES POR CARRITO
81        if(clientes_total_por_patente.length == 0){
82          var info4 = {
83            patente:json.patente_carro,
84            client:json.client,
85            count_client: 1
86          }
87          clientes_total_por_patente.push(info4)
88        }else{
89          for(var k in clientes_total_por_patente){
90            count2++
91            if(clientes_total_por_patente[k].patente == json.patente_carro){
92              if(clientes_total_por_patente[k].client != json.client){
93                clientes_total_por_patente[k].count_client++
94                break;
95              }
96            }else if(clientes_total_por_patente.length == count2){
97              var info5 = {
98                patente: json.patente_carro,
99                client: json.client,
100               count_client:1
101             }
102             clientes_total_por_patente.push(info5)
103             break;
104           }
105         }
106       }
```

Figura 21: Procesamiento 1: Ventas diarias-Promedio de ventas por cada cliente para cada carrito.

Para el calculo de clientes totales por cada carrito se realizo un procesamiento similar. Cuando se registra la primera venta se guarda en un array llamado **clientes_total_por_patente** el diccionario llamado **info4** que contiene la patente del carrito, el nombre del cliente y un contador de clientes atendidos inicializado en 1. Luego de la primera venta se procede a recorrer el array de diccionarios preguntando si la patente del carrito de la nueva venta está ya registrado en el array de diccionarios y que el cliente de la venta sea distinto a cualquier otra venta registrada por el carrito, si es así entonces el contador de clientes del carrito subirá en una unidad, en caso contrario esperará a que se recorra todo el array de diccionarios gracias a la variable count2, significando que la patente de la venta es nueva y por lo tanto se registrará esa nueva venta del carrito nuevo al array.

```
108       //PROMEDIO DE VENTAS DE CADA CARRITO POR CLIENTE
109       for(var x in ventas_total_por_patente){
110         if(ventas_total_por_patente[x].patente == clientes_total_por_patente[x].patente){
111           var info6 = {
112             patente: ventas_total_por_patente[x].patente,
113             promedio: ventas_total_por_patente[x].count_sopaipillas / clientes_total_por_patente[x].count_client
114           }
115           promedio_ventas_por_cliente.push(info6)
116           break;
117         }
118       }
119     },
120   })
121   .catch(console.error)
122 };
```

Figura 22: Procesamiento 1: Ventas diarias-Promedio de ventas por cada cliente para cada carrito.

Por último, se realiza el cálculo del promedio de ventas por cliente para cada carrito. Esto se intento realizar dividiendo las ventas totales de cada carrito entre la cantidad total de clientes atendidos en cada carrito. Esto con un ciclo for que recorra el array **ventas_total_por_patente** donde se almacenan las ventas totales de cada carrito, preguntando si la patente del carrito es igual a la patente del carrito que esta registrado en el array **clientes_total_por_patente**, el cual almacena los clientes totales atendidos por cada carrito. El resultado de esta división es el promedio de ventas por clientes para cada carrito almacenado en un diccionario llamado **promedio_ventas_por_cliente** en conjunto con la patente del carrito correspondiente al promedio. Desgraciadamente no se logro un correcto resultado de este procedimiento.

```
app.get('/ventadiaria', (req, res) => {

  console.log('Ventas totales por cada carrito')
  for(var b in ventas_total_por_patente){
    console.log(`Patente: ${ventas_total_por_patente[b].patente} , Ventas totales: ${ventas_total_por_patente[b].count_sopaipillas}`)
  }
  console.log('\n')
  console.log('Promedio de ventas a clientes')
  for(var c in promedio_ventas_por_cliente){
    console.log(`Patente: ${promedio_ventas_por_cliente[c].patente} , Promedio de ventas: ${promedio_ventas_por_cliente[c].promedio}`)
  }
  console.log('\n')
  console.log('Clientes totales por cada carrito')
  for(var a in clientes_total_por_patente){
    console.log(`Patente: ${clientes_total_por_patente[a].patente} , Clientes totales: ${clientes_total_por_patente[a].count_client}`)
  }
  res.send(ventas_total_por_patente)
})
```

Figura 23: Procesamiento 1: Ventas diarias-Una vez al dia se imprime por pantalla las ventas totales, clientes totales y promedio de ventas por clientes para cada carrito.

Como ultimo detalle, la actividad solicita mostrar las ventas totales, clientes totales promedio de ventas para cada carrito una vez al día, para esto ultimo se crea una función en el consumer donde al realizar una petición GET '/ventadiaria´ se imprimirá por pantalla las ventas totales, clientes totales y promedio de ventas para cada carrito. Por lo tanto, se creo el supuesto de que la acción una vez al día es lo mismo que realizar la petición GET '/ventadiaria´ una vez al día.

4.3. Agente Extraño

4.3.1. Registro de Aviso extraño

Para esta petición, se solicita denunciar carros prófugos, según sea la información que el usuario de. Para este se creó un producer con la siguiente estructura.

```
app.post("/ubication", (req, res) => {
  (async () => {
      const producer = kafka.producer();

      await producer.connect();
      const { patente,ubicacion } = req.body;
      var denuncia = Math.floor(Math.random()*2)
      var time = Math.floor(new Date() / 1000);

      let ubication = {
        patente:patente,
        ubicacion: ubicacion,
        denuncia: denuncia,
        time: time
      }
```

Figura 24: Petición 3: Aviso de agente extrano

Si bien, se solicitó solo indicar la coordenada, se agregaron variables para trabajar con ellas por detrás, lo que permitirá asignarle información al carro y así saber si estos serán prófugos o no. Con la función **Math.floor(Math.randon** se asigna un valor aleatorio entero entre 0 y 1, si el valor de la llave "denuncia" del diccionario "ubication" es 1 entonces corresponderá a un carrito prófugo denunciado, caso contrario la llave "denuncia" tendrá un valor de 0, por lo tanto corresponderá a un carrito no prófugo.

```
49        if(ubication["denuncia"] == 1){
50          console.log("Este carrito ha sido denunciado, es profugo")
51
52           const topicMessages = [{
53              topic: 'ubication',
54              partition:1,
55              messages:[{value:JSON.stringify(ubication),partition: 1}]
56            },
57          ]
58          await producer.sendBatch({topicMessages})
59          console.log("Envie", JSON.stringify(ubication))
```

Figura 25: Petición 3: Condiciones Ubication

Si el carro resulta ser prófugo es enviado al tópico: "ubication" y a una partición asignada para los prófugos, que en este caso será la partición 1, esta información se encuentra dentro de la variable llamada topicMessages.

```
60        }else{
61          console.log("Carrito Limpio.")
62
63           const topicMessages = [
64           {
65            topic: 'ubication',
66            partition:0,
67            messages:[{value:JSON.stringify(ubication),partition: 0}]
68           },
69         ]
70         await producer.sendBatch({topicMessages})
71         console.log("Envie", ubication)
72       }
73       await producer.disconnect();
74       res.json(ubication);
75      })();
76    });
```

Figura 26: Petición 3: Condiciones Ubication

En caso contrario, si el carrito esta limpio y no está siendo denunciado, será enviado a la partición 0 a través de la misma variable topicMessages.

Finalmente, independiente de que el carrito sea denunciado como prófugo o no, el producer realiza la acción de publicar gracias al evento **sendBatch**.

4.3.2. Procesamiento 3: Ubicación

Para este procesamiento al igual que en todos los demás, se conecta y subscribe el consumer al topic específico para el procesamiento, este topic es llamado 'ubication´.

```
33  const main = async () => {
34    const consumer = kafka.consumer({ groupId: "ubication" });
35    console.log("Entra Ubication")
36    await consumer.connect();
37    await consumer.subscribe({ topic: "ubication", fromBeginning: true });
38    console.log("EL SIGUIENTE ES EL MESSAGE")
39    await consumer.run({
40      eachMessage: async ({ topic, partition, message }) => {
41        console.log("CONDICIONES SIGUIENTES")
42
43        if(partition == 0){
44          var algo = JSON.parse(message.value.toString());
45          var count = 0;
46
47          var info = {
48            patente:algo.patente,
49            ubicacion: algo.ubicacion,
50            time: algo.time
51          }
```

Figura 27: Procesamiento 3: Ubicación

Si el registro fue de un carrito no prófugo entonces el consumer ingresará a la partición 0, se crea un diccionario llamado **info** el cual contendrá la información justa y necesaria para poder procesar la información requerida por la actividad.

```
53        console.log("Este carrito no fue denunciado")
54        if(carritos_patente_noprofugos.length == 0){
55          carritos_patente_noprofugos.push(info)
56        }
57        for(var i in carritos_patente_noprofugos){
58          if(carritos_patente_noprofugos[i].patente == info.patente){
59            if((info.time - carritos_patente_noprofugos[i].time) > 60){
60              console.log(`Carrito con patente ${carritos_patente_noprofugos[i].patente} desaparecio`)
61              carritos_patente_noprofugos.splice(i,1)
62              break;
63            }else{
64              carritos_patente_noprofugos[i].ubicacion = info.ubicacion
65              console.log(`Carrito con patente ${carritos_patente_noprofugos[i].patente} esta en la ubicacion ${carritos_patente_noprofugos[i].ubicacion}`
66              carritos_patente_noprofugos[i].time = info.time
67              break;
68            }
69          }else{
70            carritos_patente_noprofugos.push(info)
71            console.log(`Carrito con patente ${carritos_patente_noprofugos[i].patente} esta en la ubicacion ${carritos_patente_noprofugos[i].ubicacion}`)
72            break;
73          }
74        }
```

Figura 28: Procesamiento 3: Ubicación

Primero que todo, si es el primer registro en la partición 1 del topic entonces se agregará al array de diccionarios llamado **carritos_patente_noprofugos**, luego del primer registro se empezará a recorrer el array preguntando si la patente del carrito ya está registrada como un carrito no prófugo, si es así entonces pasará por otra condición que preguntara si la petición para actualizar la ubicación del carrito se realiza en menos de 60 segundos, si es asi entonces se actualiza la ubicación y el tiempo en el que se hizo la actualización, en caso contrario el carro simplemente se borra del array que almacena los carritos no prófugos llamado **carritos_patente_noprofugos**. Ahora, si un carrito no denunciado en nuevo, entonces se agrega al array de carrito no prófugos.

Si el registro fue de un carrito prófugo entonces el consumer ingresara a la particion 1 y se procede a una lógica muy similar a la usada para un carrito no profugo. Se crea un diccionario llamado **info2** el cual contendrá

la información justa y necesaria para poder procesar la información requerida por la actividad.

```
78        else if(partition == 1)
79        {
80          var algo = JSON.parse(message.value.toString());
81
82          var info2 = {
83            patente: algo.patente,
84            ubicacion: algo.ubicacion
85          }
86
87          console.log("Este carrito fue denunciado, ES PROFUGO ATRAPENLOC CTMRE")
88            if(carritos_patente_profugos.length == 0){
89              carritos_patente_profugos.push(info2)
90            }
91            for(var i in carritos_patente_profugos){
92              count++
93              if(carritos_patente_profugos[i].patente == info2.patente){
94                console.log('Carrito denunciado ya está registrado')
95                break;
96              }else if(carritos_patente_profugos.length == count){
97                carritos_patente_profugos.push(info2)
98                console.log('Carritos profugos')
99                console.log(carritos_patente_profugos)
100               break;
101             }
102           }
103       }
104     },
105   })
106   console.log("Sali del Mesagge")
107 }
```

Figura 29: Procesamiento 3: Ubicación

Al igual que para los carritos no prófugos, si es el primer registro en la partición 0 del topic entonces se agregará al array de diccionarios llamado **carritos_patente_profugos**, luego del primer registro se empezará a recorrer el array preguntando si la patente del carrito ya está registrada como un carrito prófugo, si es así entonces simplemente avisara por pantalla que el carrito prófugo ya esta registrado. En caso contrario significa que reviso todo el array de diccionarios que contiene a los carrito prófugos y no lo contiene, esto con ayuda de la variable count. Finalmente el diccionario que contiene la información del carrito prófugo nuevo es agregado al array y mostrado por pantalla.

5. Kafka

5.1. ¿Que es?

Antes de proceder a explicar el funcionamiento o configuración de kafka, es necesario saber que es kafka, al menos en grandes rasgos.

Kakfa, es una tecnología utilizada para realizar una transición de datos y procesamiento en tiempo real, tecnología que principalmente empezó solo como mensajería.

En el mismo Kafka, se encuentra Zookeeper, el cual, es utilizado como medio para almacenar datos dentro de un cluster, detalles de consumidores e incluso coordinar los brokers, que en este caso utilizaremos solo uno. Antes de llegar al uso de Kafka, es necesario tener instalado y ejecutando Zookeper ya que es considerado como una dependencia de Kafka.

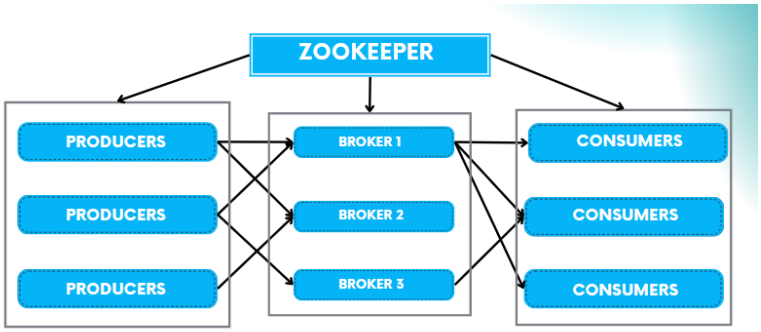

Figura 30: Kafka

Como se mencionó anteriormente, en este caso, se utilizó un solo broker, en donde se realizaron todos los trabajos, dentro de este broker, se encuentra el almacenamiento y distribución de datos mediante topicos y particiones, dentro de cada topico, podemos encontrar las particiones.

Para poder generar una explicación que sea sencilla de entender, consideremos un broker como la ciudad entera, dentro de la ciudad existen calles principales o avenidas, estas serían consideradas como nuestros topicos, y si queremos realizar algo más especifico, están las particiones, que serán como subcalles.

En el proyecto, se consideraron topicos especifico para cada tema a tratar, por ejemplo, para el caso de los miembros, se creó un topico especifico de "members", el cual, se creaba al momento de utilizarlo.

Esto, se repitió para todos los casos ya sean miembros, ventas y ubicación. Por otro lado, dentro del topico de miembros y ubicación, se crearon particiones, las cuales, nos permitieron ir controlando de una manera más especifica la información.

De igual manera, fue necesario establecer una configuración dentro del docker para kafka, ya que aquí fue donde se genero todo nuestro entorno de trabajo.

```yaml
kafka:
  container_name: elkafka
  image: 'bitnami/kafka:latest'
  networks:
   - lared
  depends_on:
    - zookeeper
  environment:
    KAFKA_ADVERTISED_HOST_NAME: "kafka"
    KAFKA_ADVERTISED_PORT: "9092"
    ALLOW_PLAINTEXT_LISTENER: "yes"
    KAFKA_CFG_ZOOKEEPER_CONNECT: "zookeeper:2181"
    KAFKA_CFG_NUM_PARTITIONS: "2"

  ports:
    - 9092:9092
```

Figura 31: Docker : Kafka

Como se observa en la imagen anterior, se realizaron ciertas configuraciones que significan lo siguiente:

- **networks: lared**, red de tipo bridge creada de manera local para la correcta comunicación entre los servicios levantados en cada contenedor de docker.
- **depends_on: zookeeper**, dependencia del contenedor donde esta alojado el servicio kafka hacia el contenedor donde esta alojado el servicio zookeeper.
- **ports: 9092**, puerto en el que se aloja el contenedor que aloja el servicio de kafka. De manera estándar es el 9092.
- **image: bitanmi/kafka:latest**, imagen de bitnami mas reciente para la instalación de kafka en el contenedor de docker.
- **environment: KAFKA_ADVERTISED_HOST_NAME: kafka**, declarar nombre del sistema kafka.
- **environment: KAFKA_ADVERTISED_PORT: 9092**, declarar el puerto que escuchara el sistema kafka.
- **environment: KAKFAA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181**, declarar el socket con el que se comunicara el sistema kafka con el sistema zookeeper.
- **environment: KAKFA_CFG_NUM_PARTITIONS: 2**, declarar la cantidad de particiones que contendrá cada topic al momento de que se creen.

6. Preguntas

6.1. ¿Cómo Kafka puede escalar tanto vertical como horizontalmente? Relacione su respuesta con el problema asociado, dando un ejemplo para cada cada uno de los tipos de escalamiento

Colocando en contexto la pregunta, escalar kafka no es tan como redis, donde se le puede colocar más memoria al cache, en el caso de kafka hay que tener en cuenta en donde esta funcionando kafka. En este caso se esta realizando en un contenedor de docker, entonces el escalamiento vertical que por definición es, aumentar las capacidades del recurso, se puede ver ejemplificado al momento de dedicar mas recursos al contenedor donde se esta ejecutando kafka.

Para el caso de escalamiento horizontal que por definición es, aumentar la cantidad de los recursos, este se puede ver ejemplificado al momento de aceptar más de un nodo en un clúster y no proporcionar tiempo de inactividad para las actualizaciones necesarias del sistema.

En lo que consta con Kafka, como se evidencia en esta tarea, se puede generar un escalamiento al momento de segmentar o bien, distribuir la información y como esta es procesada. Para nuestro caso, utilizamos topicos, los cuales, tienen dentro de ellos particiones o canales que permiten distribuir la información de una manera útil y segura, así como controlar el uso de recursos.

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

En Kafka, se hace notar el hecho de ser un sistema distribuido al momento de trabajar con un productores o cliente, este envía y trabaja con la información mediante varios consumidores o servidores, los cuales funcionarán tanto de manera independiente como en conjunto.

Dentro de la tarea que se desarrolló previamente, se evidencia de manera superficial al analizar la figura ” Estructura del Sistema´´ que demuestra como es que existen varios servidores trabajando de manera simultanea para que posteriormente sean analizados estos datos.

Referencias

[1] Github de la tarea

[2] Kafka

[3] Servidor Kafka Node $|||KafkaJS$