

udp FACULTAD DE INGENIERÍA Y CIENCIAS

Actividad:

Informe Tarea 2 Sistemas Distribuidos

Integrante: Diego Serrano
Profesor: Nicolás Hidalgo

Introducción

En el siguiente trabajo se tuvo una introducción a los sistemas de procesamiento basados en stream, mediante el uso de una herramienta llamada Apache Kafka, una plataforma distribuida que se utiliza para la transmisión de datos, de código abierto, desarrollado por LinkedIn y donado a Apache Software Foundation, y se destaca por tener un modelo basado en Publish-Subscribe. A través de este trabajo se tiene que comprender el funcionamiento de Kafka, para luego aplicar sus distintos atributos y funcionalidades con el objetivo de realizar un sistema de microservicios que simule el funcionamiento de una plataforma de delivery. El sistema también debe ser capaz de interactuar con peticiones HTTP POST provenientes de puertos específicos para dar comienzo al procesamiento de transacciones, además de interactuar con peticiones HTTP GET para que el usuario sea capaz de saber el estado actual de una transacción en el sistema.

Desarrollo

Para dar comienzo a la actividad, se utilizó el lenguaje de programación Python para implementar el sistema entero. También se contó con archivos como "requirements.txt" para la instalación de dependencias que son necesarias para que el sistema funcione correctamente en el ambiente de trabajo, el archivo "steamgames.csv" que cuenta con una lista extendida de varios videojuegos que se encuentran disponibles en la página de Steam, además de contar con datos como nombre, precio y número identificador para cada videojuego, información que servirá para definir en el sistema, los productos disponibles en la tienda, y también se implementó la herramienta Docker Compose para levantar imágenes como Zookeeper y Confluent Kafka, que cuentan con toda la información necesaria para ejecutar el sistema correctamente. Es importante mencionar que el sistema que se va a desarrollar a continuación fue realizado mediante la ayuda de un video en Youtube llamado "Scalable & Event Driven Food Ordering App with Kafka & Python | System Design", video donde se presenta una demostración de cómo implementar un sistema con microservicios utilizando la herramienta Apache Kafka en Python. El video fue esencial para desarrollar los servicios de solicitudes y procesamiento, además de mostrar cómo se pueden levantar las imágenes de Zookeeper y Confluent Kafka mediante el uso de Docker Compose.

```

version: '3.3'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181

  kafka:
    image: confluentinc/cp-kafka:5.3.1
    depends_on:
      - zookeeper
    ports:
      - 29092:29092
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

```

Figura 1: Imágenes presentes en el docker-compose.yaml del sistema

```

black==21.12b0
click==8.0.3
kafka-python==2.0.2
mypy-extensions==0.4.3
pathspec==0.9.0
platformdirs==2.4.0
tomli==1.2.2
typing-extensions==4.0.1

```

Figura 2: Herramientas presentes en requirements.txt, instaladas mediante pip

Adicionalmente, se instaló la herramienta Flask mediante pip, ya que esta cuenta con funcionalidades que permiten al sistema interactuar con peticiones HTTP provenientes de una terminal.

Una vez implementadas las imágenes en el Docker Compose y los archivos necesarios para levantar el sistema, se procede a desarrollar el código perteneciente al servicio de solicitudes, el cual se llama “peticiones.py” en este sistema. En este código, el servicio cuenta con un Kafka Producer que se encuentra conectado en localhost:9092, y también se encuentra escuchando constantemente en localhost:3000, en la espera de recibir una petición HTTP que le permita saber qué datos deben ser extraídos del archivo “steamgames.csv”. Cuando el servicio recibe una petición HTTP POST, este comienza su ejecución almacenando los datos del juego solicitado, el correo electrónico del usuario y un ID de transacción en un diccionario. Luego de almacenar los datos, se procede a convertir el diccionario en una secuencia de bytes para que Kafka Producer pueda enviar el diccionario en forma de solicitud a través de un topic bajo el nombre de “compra_detalle”, nombre que va a ser referenciado por el Kafka Consumer de “procesamiento.py” para que este pueda recibir el diccionario de “peticiones.py”.

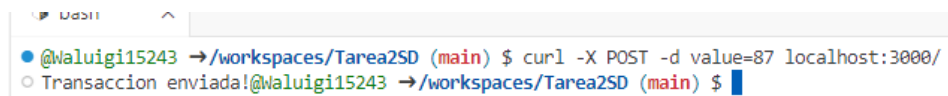
```

@app.route('/', methods=['POST'])
def sendOrder():
    with open("steamgames.csv") as archivo:
        global transaction
        i = int(request.form['value'])
        data = csv.reader(archivo)
        for _ in range(i):
            next(data)
        row = next(data)
        transaction += 1
        order = {
            "orderid": transaction,
            "gameid": row[0],
            "name": row[1],
            "price": row[2],
            "usermail": "diego.serrano1@mail.udp.cl",
        }
        print(order)
        producer.send(topic, json.dumps(order).encode("utf-8"))
        print("Transaccion enviada! \n")
        return "Transaccion enviada!"

if __name__ == '__main__':
    app.run(debug=True, port=3000)

```

Figura 3: Función de peticiones.py que recibe peticiones HTTP para iniciar una transacción



```

@Waluigi15243 →/workspaces/Tarea2SD (main) $ curl -X POST -d value=87 localhost:3000/
○ Transaccion enviada!@Waluigi15243 →/workspaces/Tarea2SD (main) $

```

Figura 4: Respuesta de peticiones.py luego de enviar una petición HTTP POST

Luego de haber desarrollado el código para el servicio generador de solicitudes, se procede a desarrollar la siguiente parte del sistema, esto corresponde al código del servicio de procesamiento llamado “procesamiento.py”. Este servicio tiene dos funciones principales, por una parte cuenta con un Kafka Consumer que se encuentra conectado a localhost:9092, buscando que hayan nuevas solicitudes enviadas a través del topic “compra_detalle” que procesar. La segunda parte corresponde a lo que sucede cuando el servicio de procesamiento recibe nuevas solicitudes en el topic, ya que luego de que se recibe el diccionario enviado por el Kafka Producer en “peticiones.py”, se continua agregando un nuevo atributo al diccionario llamado “estado”, que como su nombre indica, va a ser esencial para entender cuál es el estado actual de la transacción al momento de ser consultada por un usuario. Luego de definir los datos importantes, la transacción procede a pasar por 4 estados (recibido, preparando, entregando y finalizado), durando alrededor de 8 segundos por cada uno de ellos. Entre los intervalos de tiempo de espera de cada estado, el código utiliza un Kafka Producer conectado en localhost:9092 y a través de un topic bajo el nombre de “compra_confirmada”, se envían los datos y el estado actual de la transacción para dar a conocer el avance actual al servicio de notificaciones por medio de su Kafka Consumer.

```

while True:
    print("Buscando transacciones... \n")
    for message in consumer:
        consumed_order = json.loads(message.value.decode('utf-8'))
        orderid = consumed_order["orderid"]
        gameid = consumed_order["gameid"]
        price = consumed_order["price"]
        name = consumed_order["name"]
        usermail = consumed_order["usermail"]
        data = {
            "orderid": orderid,
            "gameid": gameid,
            "name": name,
            "price": price,
            "usermail": usermail,
            "estado": "recibido",
        }
        producer.send(topicNotif, json.dumps(data).encode("utf-8"))
        time.sleep(8)
        data["estado"] = "preparando"
        producer.send(topicNotif, json.dumps(data).encode("utf-8"))
        time.sleep(8)
        data["estado"] = "entregando"
        producer.send(topicNotif, json.dumps(data).encode("utf-8"))
        time.sleep(8)
        data["estado"] = "finalizado"
        producer.send(topicNotif, json.dumps(data).encode("utf-8"))
        time.sleep(8)

```

Figura 5: Función de procesamiento.py que consume solicitudes de un topic para procesarlas y que se comunica con notificaciones.py para indicar el avance actual

Una vez que se ha desarrollado el servicio encargado del procesamiento de transacciones, se procede a finalizar el sistema mediante la implementación de un código llamado “notificaciones.py”. Al igual que con “procesamiento.py”, el servicio de notificaciones tiene dos funciones principales, en primer lugar se implementó una función bajo el nombre de “mail()”, donde se implementó un Kafka Consumer que se encuentra conectado a localhost:9092, esperando a que lleguen nuevas transacciones o actualizaciones en transacciones anteriores dentro del topic “compra_confirmada”. Cuando el servicio recibe una transacción en esta función, se procede a crear un correo electrónico mediante las herramientas de email.mime, utilizando la información incluida en la transacción como el correo electrónico del usuario y ciertos datos de la compra realizada para rellenar el body del correo, y mediante la herramienta smtplib se procede a enviar un correo electrónico de protocolo SMTP al comprador del producto para notificarle sobre el estado actual de su compra. Luego de enviar los datos de la transacción al correo electrónico del usuario, se procede a almacenar la información utilizada en un arreglo de diccionarios, donde se almacenan todas las transacciones del sistema.

La segunda parte de este sistema corresponde a la recepción de peticiones HTTP GET para notificar el estado actual de una transacción en particular. Este proceso se realizó mediante la implementación de una función llamada "index()", la cual se encarga de escuchar constantemente en localhost:5000, en la espera de recibir una petición HTTP que le permita saber qué transacción debe ser impresa en pantalla. Cuando el servicio recibe una petición HTTP GET con el ID de una de las transacciones, se muestra en pantalla el diccionario con todos los datos de la transacción, incluyendo el estado actual de la transacción solicitada mediante la petición

```
def mail():
    while True:
        for message in consumer:
            global diccionarios
            order = json.loads(message.value.decode('utf-8'))
            name = order["name"]
            usermail = order["usermail"]
            price = order["price"]
            fromAddress = ""
            toAddress = order["usermail"]
            subject = "Estado de la Transaccion"
            with diccionariosLock:
                if order["estado"] == "recibido":
                    diccionarios.append(order)
                    body = f"Estimado Usuario: \n Le enviamos este correo para informarle a usted que se ha recibido un  
emailMsg = createMail(subject, body, fromAddress, toAddress)
                    sendMail(emailMsg, fromAddress)
                elif order["estado"] == "preparando":
                    diccionarios[-1]["estado"] = order["estado"]
                    body = f"Estimado Usuario: \n Le enviamos este correo para informarle a usted que su transaccion se  
emailMsg = createMail(subject, body, fromAddress, toAddress)
                    sendMail(emailMsg, fromAddress)
                elif order["estado"] == "entregando":
                    diccionarios[-1]["estado"] = order["estado"]
                    body = f"Estimado Usuario: \n Le enviamos este correo para informarle a usted que su transaccion ha  
emailMsg = createMail(subject, body, fromAddress, toAddress)
                    sendMail(emailMsg, fromAddress)
                elif order["estado"] == "finalizado":
                    diccionarios[-1]["estado"] = order["estado"]
                    body = f"Estimado Usuario: \n Le enviamos este correo para informarle a usted que su transaccion ha  
emailMsg = createMail(subject, body, fromAddress, toAddress)
                    sendMail(emailMsg, fromAddress)
```

Figura 6: Función de notificaciones.py que crea y envía un correo electrónico al comprador

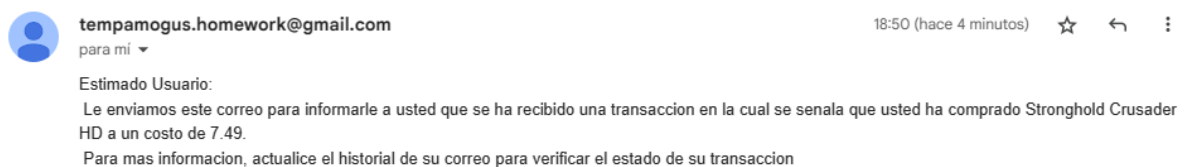


Figura 7: Correo electrónico que recibe el usuario, notificando el estado actual de una transacción

```

@app.route('/')
def index():
    global diccionarios
    i = int(request.args.get('value'))
    with diccionariosLock:
        for transaction in diccionarios:
            if transaction["orderid"] == i:
                return transaction
    return "ERROR: No existe ninguna transaccion con ese numero ID"

```

Figura 8: Función de notificaciones.py que recibe peticiones HTTP GET para mostrar el estado de una transacción en pantalla

```

● @Waluigi15243 → /workspaces/Tarea2SD (main) $ curl -X GET localhost:5000/?value=1
{
  "estado": "preparando",
  "gameid": "438650",
  "name": "Gravity Compass",
  "orderid": 2,
  "price": "1.99",
  "usermail": "diego.serrano1@mail.udp.cl"
}

```

Figura 9: Ejemplo de una transacción en pantalla luego de utilizar una petición HTTP GET

Preguntas de la Entrega

1. Describa en detalle cómo la arquitectura Apache Kafka permite brindar tolerancia a fallos en el sistema propuesto.

Como la arquitectura Apache Kafka está basada en modelos de tipo Publisher-Subscriber, significa que la arquitectura es capaz de brindar tolerancia a fallos gracias a que se pueden recibir mensajes incluso cuando la comunicación entre un productor y un consumidor se ve interrumpida debido a que una de las dos entidades se encuentra fuera de línea o debido a que no se logra establecer una conexión entre ambos. Esto se debe a que Apache Kafka cuenta con un servidor bajo el nombre de broker, que es capaz de almacenar los mensajes enviados por un productor, y aunque el broker líder llegue a fallar bajo ciertas condiciones y circunstancias, Apache Kafka es capaz de recuperar los datos automáticamente gracias a que cuenta con múltiples réplicas del broker líder, los cuales toman su lugar en el caso de que se caiga. Esta arquitectura también es capaz de brindar tolerancia a fallos mediante la formación de grupos de consumidores, los cuales pueden encargarse de procesar un subconjunto selecto de particiones de un topic, y en el caso de que se detecte una falla en uno de ellos, otro consumidor es capaz de tomar su lugar y continuar con el procesamiento del consumidor caído. Dichas características aseguran que la arquitectura se encuentre disponible por grandes intervalos de tiempo y sea capaz de escalar de forma horizontal.

2. Considerando el sistema propuesto ¿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.

En el caso de tener que manejar un número mayor de solicitudes en el sistema, lo que se recomienda hacer en términos de ajustes de configuración es aumentar el número de particiones por grupo de consumidores en un tema. Esta acción permite que en el sistema se pueda dividir la carga de trabajo entre más entidades, logrando así el paralelismo en el procesamiento de mensajes del sistema. También es posible modificar el tamaño que tiene cada una de estas particiones con el fin de evitar que se causen cuellos de botella en los brokers. Adicionalmente, se puede asegurar el escalamiento del sistema mediante el aumento del número de brokers, asegurando así que existan más servidores de Kafka encargados de una parte del tráfico de solicitudes y que se mejore la capacidad total de solicitudes en el sistema.

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 encontrar solución a esta problemática, se tuvo que implementar la librería time de Python, la cual cuenta con herramientas que permiten medir los tiempos de ejecución para cada función implementada en el sistema, de tal manera que se puede medir el rendimiento del sistema con respecto a diferentes cargas de trabajo. A continuación se presenta en una tabla la latencia del sistema con respecto al número de solicitudes por carga:

Número de solicitudes por carga	Latencia promedio (segundos)
5	0.009792
10	0.010524
15	0.009948

La tabla anterior muestra cómo la latencia promedio del sistema de gestión de pedidos va variando según el número de solicitudes que se reciben en procesamiento.py a través del tópico "compra_detalle". Si bien los tiempos de latencia promedio no varían demasiado por caso, es posible notar que la latencia tiende a incrementar cuando la carga del sistema aumenta, sugiriendo que la latencia del sistema es directamente proporcional al número de solicitudes que se encuentren en las colas de mensajería.

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

Para sistemas como el implementado en esta actividad, donde se almacenan los pedidos de la tienda mediante el uso de colas de mensajería, se pueden implementar políticas de auto escalado que basados en patrones de tráfico y el histórico de demanda de productos del sistema en relación a cómo fueron las ventas promedio en años anteriores. Dicha información permite al sistema predecir aquellos períodos de baja demanda y le permite prepararse para momentos donde existen picos de demanda. Confluent Cloud es una plataforma de transmisión de datos ideal para el auto escalado de sistemas realizados bajo esta arquitectura, ya que ofrece un servicio completamente gestionado y nativo en la nube que permite ajustar automáticamente la carga de clústeres en Kafka basándose en las cargas de trabajo, mediante la adición o eliminación de brokers para manejar períodos de alta demanda y para optimizar el uso de recursos.

5. Considerando la diversidad de fuentes 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.

Uno de los mayores desafíos al momento de integrar el sistema de por sí fue asegurar que los servicios con Kafka Consumer fueran capaces de recibir y consumir los mensajes enviados a través de un tópico por parte de un Kafka Producer, debido a que en ocasiones el consumidor no respondía a las solicitudes enviadas por el servicio de solicitudes y se quedaba esperando indefinidamente a que existiera una nueva transacción que procesar. Es por esto que se tuvo que implementar parámetros adicionales en los Kafka Consumer para asegurar que se pudiera recibir mensajes en la cola de manera persistente, evitando que las caídas de estos consumidores causen que se perdieran múltiples peticiones en el sistema. Uno de los parámetros utilizados fue `auto_offset_reset`, que permite definir el orden en que los consumidores leen el log de mensajes en un tópico, de tal forma que para este caso se decidió que los consumidores deben comenzar desde el inicio del log. Otro parámetro implementado fue `enable_auto_commit=True`, donde se especifica para este caso que existe un offset de 1000 milisegundos entre los mensajes leídos por los consumidores. Y finalmente se utilizó el parámetro `group_id` para determinar los grupos de consumidores existentes en el sistema, donde se decidió que tanto los consumidores del servicio de procesamiento como aquellos del servicio de notificaciones van a compartir el mismo identificador de grupo, lo que significa que los mensajes del sistema fueron distribuidos en un mismo grupo de consumidores.

Dichas acciones permitieron que los Kafka Consumer del sistema fueran capaces de recibir y leer mensajes de manera consistente, evitando así que hubieran más pérdidas de transacciones.

Análisis

A continuación se va a diseñar un escenario de testeo del sistema para poner a prueba qué tan robusto es el sistema al momento de recibir y leer mensajes almacenados en sus colas de mensajería. Para ello se va a tener en cuenta un escenario donde se recibe un número de 20 solicitudes de pedido al mismo tiempo, para comprobar la latencia de cada uno de estos pedidos y para saber cuántos de estos mensajes fueron efectivamente recibidos por los consumidores del sistema. Se pueden observar los resultados obtenidos en el escenario en las siguientes tablas:

Número de Petición	Latencia (segundos)
1	0.050393104553222656
2	0.004365682601928711
3	0.00819706916809082
4	0.009479761123657227
5	0.005126953125
6	0.008307456970214844
7	0.010029077529907227
8	0.031049251556396484
9	0.011096477508544922
10	0.00985860824584961
11	0.00516057014465332
12	0.007317781448364258
13	0.024838924407958984
14	0.014763355255126953
15	0.04493904113769531
16	0.010666608810424805
17	0.004806995391845703
18	0.004913806915283203
19	0.006757974624633789
20	0.004799842834472656

Tabla 1: Valores de latencia registrados en el servicio de solicitudes al momento de enviar pedidos

Número de Petición	Latencia (segundos)
1	12.071831464767456
2	12.01012921333313
3	12.007921695709229
4	12.008578538894653
5	12.01165246963501
6	12.007928609848022
7	12.012027978897095
8	12.010571956634521
9	12.008625984191895
10	12.00697660446167
11	12.00757360458374
12	12.010785102844238
13	12.010788202285767
14	12.01300048828125
15	12.00991678237915
16	12.011250019073486
17	12.009714603424072
18	12.010466814041138
19	12.012500047683716
20	12.012743711471558

Tabla 2: Valores de latencia registrados en el servicio de procesamiento luego de terminar con el procesado de transacciones

Número de Petición	Latencia Promedio (segundos)
1	0.710975
2	0.566025
3	0.594925
4	0.643925
5	0.716225
6	0.518075
7	0.64825
8	0.6366
9	0.58125
10	0.6502
11	0.591825
12	0.53695
13	0.552125
14	0.5885
15	0.61365
16	0.614375
17	0.5387
18	0.550825
19	0.659225
20	0.5231

Tabla 3: Valores de latencia promedio registrados en el servicio de notificaciones luego enviar correos electrónicos al comprador

En las tablas anteriores se puede observar la latencia registrada para los servicios de pedidos, procesamiento y notificaciones luego de que se hayan enviado 20 solicitudes de compra por parte de los usuarios. En primer lugar, se puede notar que se han procesado todas las solicitudes de los clientes a pesar de que el procesamiento de las solicitudes haya sido lento a causa de los cambios de estado de cada transacción, por lo que se puede concluir que el sistema tiene una alta disponibilidad al momento de tratar con grandes cargas de solicitudes al mismo tiempo.

Por otro lado, se puede observar que la latencia general para cada una de las solicitudes no sufre demasiadas diferencias con respecto a los tiempos de ejecución, procesamiento y envío de correos electrónicos, sin embargo existen ciertos puntos donde la latencia aumenta considerablemente en comparación a otras latencias registradas del sistema, especialmente en casos como la petición 1 y 15 en el servicio de solicitudes, la primera petición ingresada en el servicio de procesamiento, y la petición 1 y 5 ingresada en el servicio de notificaciones. Para los primeros registros de latencia, se puede concluir que los tiempos de respuesta en estos casos fue mayor debido a que el sistema recién se está adaptando al proceso de gestión de pedidos, por lo que la latencia suele ser mayor. En el caso de las otras peticiones, es posible concluir que la saturación en las colas de mensajería están causando que los consumidores deban repartirse el número de transacciones en grupos de consumidores más pequeños, lo que resulta en que los consumidores del sistema se demoren más tiempo en leer mensajes, aumentando así la latencia para dichos casos.