UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Laboratorio 2 - Kafka y Spark

Integrantes: Esteban Arenas
Bryan Salgado

Asignatura: Sistemas Distribuidos

25 de Agosto de 2024

Índice de Contenidos

1. Introducción

En el contexto de los sistemas distribuidos, la escalabilidad es esencial para permitir que las aplicaciones manejen un aumento en la carga de trabajo sin comprometer su rendimiento. La replicación de datos distribuye copias de la información en múltiples ubicaciones, lo que no solo mejora la confiabilidad del sistema al proporcionar tolerancia a fallos, sino que también aumenta la disponibilidad de un servicio al permitir la redundancia de nodos.

Sin embargo, la replicación introduce el desafío de mantener la consistencia de datos. A medida que los datos se replican en diferentes nodos, implica que todas las réplicas reflejan el mismo estado de información en cualquier momento. Siempre que es modificada una copia, ésta se vuelve diferente al resto de copias, por lo que para poder garantizar la consistencia, las modificaciones deben realizarse en todas las copias.

En la práctica, grandes plataformas como GitHub, Twitter y Netflix manejan enormes volúmenes de datos y usuarios, por lo que deben hacer uso de técnicas de replicación para poder hacer entrega de un servicio disponible en todo momento, considerando las diferencias geográficas de sus usuarios para la reducción de la latencia y, lo más importante, garantizar que la información sea consistente en todas las copias existentes del sistema.

En este informe, serán explorados los conceptos de replicación y consistencia de datos, a través de un diseño de baja fidelidad de un sistema de control de versiones como GitHub. Se discutirá la arquitectura del sistema, la metodología utilizada para evaluar su ejecución, y los resultados obtenidos serán analizados junto con los desafíos y consideraciones asociadas con la replicación y la consistencia en sistemas distribuidos.

2. Caso de estudio

En el desarrollo de proyectos informáticos, GitHub es la plataforma líder en el control de versiones, que permite a equipos de desarrolladores colaborar manteniendo un control sobre un repositorio compartido que recibe modificaciones constantemente de diferentes usuarios.

En este caso de estudio, se ha diseñado una solución para similar al comportamiento de un sistema de control de versiones distribuido, tomando como ejemplo a GitHub. El problema principal radica en la gestión de las operaciones *push* y *pull* en un entorno distribuido, donde múltiples clientes pueden interactuar con un repositorio que se encuentra replicado en varias instancias.

Los clientes pueden manejar dos tipos de operaciones, en primer lugar, la operación *push* permite al cliente subir una modificación hecha en el repositorio local al servidor donde se encuentra el repositorio remoto. Para que una solicitud *push* sea aceptada, la versión del repositorio en la máquina cliente debe coincidir con la versión del repositorio en la instancia del servidor.

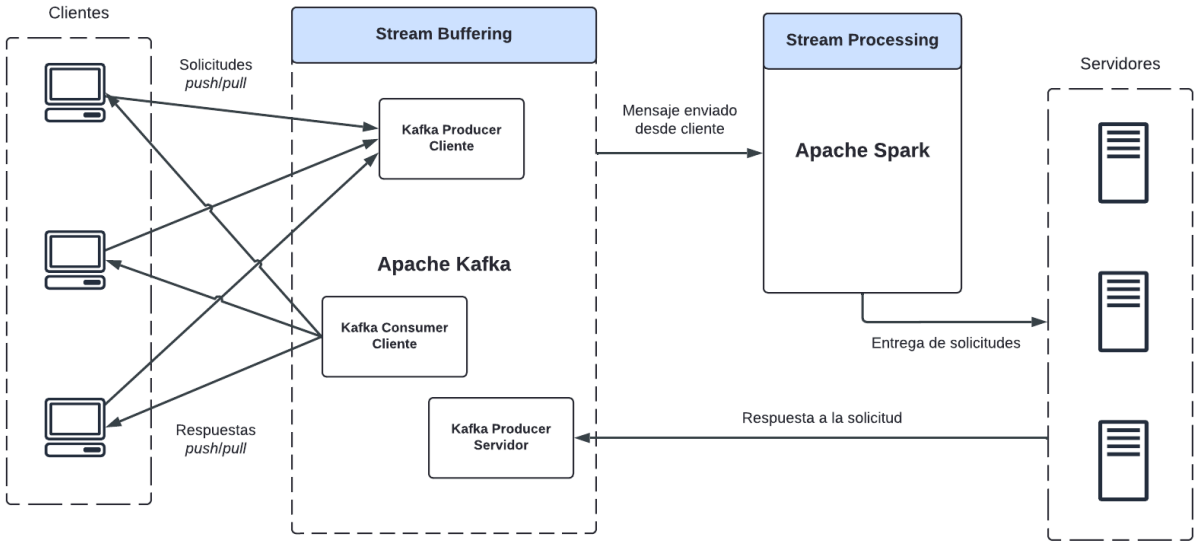
Por otra parte, la operación *pull* permite al cliente obtener la última versión del repositorio, migrando toda esta información a la máquina local. El cliente es quien debe asegurarse de mantener la última versión del repositorio para evitar futuros conflictos de versiones.

El desafío está en cómo mantener la coherencia del estado del repositorio entre todas las instancias que lo replican. Cuando se realiza una operación *push* el servidor que recibe la solicitud presenta una actualización de la copia que maneja, esta actualización debe ser informada o propagada a todas las demás instancias del repositorio.

3. ARQUITECTURA PROPUESTA

El objetivo es simular la interacción de uno o varios clientes con un repositorio replicado en múltiples instancias, imitando las operaciones de *pull* y *push* características de una plataforma de control de versiones como GitHub. La arquitectura propuesta asegura la consistencia entre las réplicas y maneja las solicitudes de los clientes mediante un esquema que emplea tecnologías como Apache Kafka y Apache Spark para el manejo de la mensajería entre cliente y servidor, y procesamiento de datos respectivamente.

En la lógica de negocio, cada cliente puede tener una copia local del repositorio que puede contar con la última versión disponible o estar desactualizado, también se puede dar el caso en que aún no se ha clonado el repositorio por primera vez. El cliente interactúa con el sistema a través de las operaciones disponibles (*pull* y *push*), estas se traducen en solicitudes gestionadas por un sistema de mensajería basado en Apache Kafka, como se muestra en el diagrama de la arquitectura de la solución.



Las solicitudes enviadas son procesadas en tiempo real por Apache Spark, este sistema captura y gestiona el flujo de datos que proviene de Kafka, transformando cada mensaje en un formato estructurado que permite aplicar la lógica correspondiente. Para simular el acceso de una réplica, se introduce la aleatoriedad para definir el servidor al cual se realizará la solicitud, imitando la función de un balanceador de carga.

La lógica decide la acción a tomar según la operación solicitada, evaluando el estado actual del repositorio y la versión del cliente. En caso de un *pull*, Spark determina si el cliente necesita actualizar su repositorio local, crearlo por primera vez, o si ya posee la última versión de este. Para un *push*, verifica si el cliente cuenta con la versión más

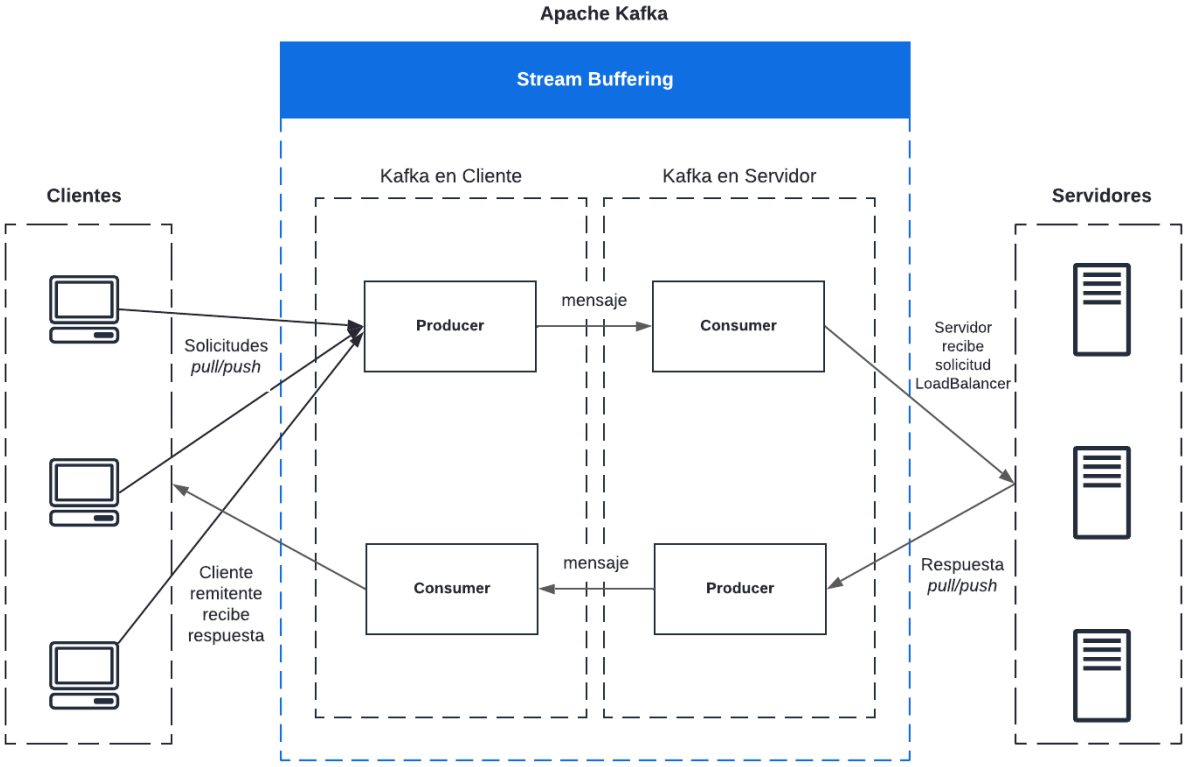
actualizada del repositorio, determinando si es posible realizar la modificación global del repositorio.

Para finalizar, Apache Spark se encarga de sincronizar las réplicas del repositorio según sea necesario y envía una respuesta al cliente mediante Kafka, este mensaje describe el estado de la solicitud. En caso de una respuesta *pull* y *push* satisfactoria, la máquina cliente actualizará el estado de su repositorio local con la información envíada por el servidor.

Hasta ahora se ha hablado de una arquitectura que no es la que se ha implementado, sino que corresponde a una propuesta de arquitectura, creada en la primera fase de diseño de la solución. En esta etapa se estaban conociendo las tecnologías propuestas sin aún comenzar a escribir código.

Si bien, se trabajó en estudiar las tecnologías de Apache, pero especialmente Spark presentó una gran dificultad para ser implementado, llegando al punto de decir buscar dar solución al problema haciendo uso solamente de Kafka.

Tomar este nuevo enfoque significa también modificar la arquitectura, que ahora solo depende de Kafka, haciendo uso *producer* y *consumer* por ambas partes, logrando así la comunicación entre clientes y servidor. En cuanto a la lógica para asegurar la consistencia, esta se encuentra desarrollada en el mismo código que maneja los servidores.



No se ha mencionado anteriormente, pero la forma de manejar a cuál servidor se redirigirá la solicitud es mediante una simulación de balanceador de carga. Luego de cada operación, las copias deben ser sincronizadas con las modificaciones respectivas de la copia seleccionada, esto ocurre siempre antes de recibir un nuevo mensaje.

Durante la implementación, se encontraron varios problemas al intentar conectar Kafka con Spark:

1. **Compatibilidad de Versiones**: Las versiones de Kafka y Spark utilizadas presentaban incompatibilidades, lo que generaba errores en la conexión y en la transmisión de datos.
2. **Configuración Compleja**: La configuración de los conectores y la integración entre Kafka y Spark resultó ser más compleja de lo anticipado, requiriendo ajustes constantes y resolución de conflictos.
3. **Errores de Conexión**: Se presentaron errores recurrentes en la conexión entre los clústeres de Kafka y Spark, lo que impedía el flujo continuo de datos.

Debido a estos problemas, se decidió simplificar la arquitectura eliminando Spark y utilizando únicamente Kafka.

1. Kafka:
    - Productores: Aplicaciones que generan eventos y los publican en temas de Kafka.
    - Tópicos: Canales de comunicación donde se almacenan los eventos..
    - Consumidores:Leen los eventos de los temas y los procesan.
2. Flujo de Datos:
    - Generación de Eventos: Los productores generan eventos que representan acciones o cambios en el sistema, como commits, pull.
    - Publicación en Temas: Los eventos generados se publican en los temas de Kafka correspondientes. Cada tipo de evento puede tener su propio tema.
    - Gestión de Consistencia: Kafka asegura la consistencia de los datos mediante la configuración de réplicas y la gestión de offset.

4. Metodología

4.1 Hardware Utilizado

El entorno de experimentación se configuró en una máquina local con las siguientes especificaciones de hardware:

- **Procesador**: Intel(R) Core(TM) i5
- **Memoria RAM**: 24,0 GB
- **Sistema Operativo**: Windows
- **Versión de Python**: Python 3.12 o superior
- **Kafka**: Apache Kafka 2.13.0 o superior

4.2 Software y Librerías

Se instalaron y configuraron las siguientes herramientas y librerías para la ejecución de los experimentos:

- **Apache Kafka**: Se utilizó para la mensajería entre los componentes del sistema. La configuración de Kafka incluye un único nodo con el broker corriendo en localhost:9092.
- **Librerías de Python**:
    - kafka: para la interacción con Kafka.
    - json: para la serialización y deserialización de mensajes.

4.3 Configuración de Kafka

El broker de Kafka se configuró utilizando los valores por defecto. Se crearon dos tópicos:

- Tópico para Solicitudes: repo-requests
- Tópico para Respuestas: repo-responses

4.4 Descripción del Código

El sistema se compone de dos scripts principales:

- **Server (server.py)**:

Escucha mensajes en el tópico *repo-requests*, procesa las solicitudes de los clientes (tales como *push* y *pull*), y responde a través del tópico *repo-responses*.

Mantiene un estado del repositorio y gestiona versiones de los mismos, aplicando una lógica de aceptación o rechazo de las solicitudes de *push* y *pull*.

- **Cliente (main.py)**:

  Envía solicitudes de *push* y *pull* para un conjunto de clientes simulados.

  Espera y procesa las respuestas del servidor, actualizando el estado local del cliente según corresponda.

4.5 Procedimiento

El experimento se realizó en los siguientes pasos:

1. **Inicialización del Servidor**:

   Se ejecutó el script server.py, que permanece activo y a la espera de recibir mensajes.El servidor procesa cada mensaje recibido y envía la respuesta correspondiente.

2. **Ejecución de Clientes**:

   Se ejecutó el script main.py, el cual envía una serie de mensajes de *push* y *pull* simulando las acciones de diferentes usuarios.El cliente espera las respuestas del servidor y procesa los resultados.

4.6 Parámetros de Entrada

- **Clientes Simulados**: Tres clientes con diferentes versiones de un repositorio simulado.
- **Acciones Realizadas**: Cada cliente intenta realizar un *push* al servidor. Las respuestas varían según el estado del repositorio en el servidor y la versión que el cliente posee.

4.7 Recolección de Resultados

Se realizaron 3 ejecuciones del experimento para garantizar la consistencia de los resultados. Durante cada ejecución, se observó:

- Las respuestas generadas por el servidor en relación con cada solicitud.
- El estado final de cada cliente tras la recepción de todas las respuestas.

Los logs generados en cada ejecución fueron guardados para análisis posterior. Estos logs contienen la secuencia de mensajes enviados y recibidos, así como las decisiones tomadas por el servidor.

5. Resultados y discusión

Tal como fue mencionado, la forma en que se llevó a cabo la comunicación fue haciendo uso exclusivo de Apache Kafka, debido a la dificultad presentando en la aplicación de Apache Spark. Debido a esto, los resultados fueron obtenidos mediante un *producer* y un *consumer*, siendo el primero el cual empaqueta la información en un mensaje, para que luego el segundo lo reciba y transforme tipo de dato manipulable.

Si bien, los resultados obtenidos por consola no muestran constantemente el estado del repositorio luego de cada acción, pero mediante las salidas existentes es posible interpretar el correcto funcionamiento de la lógica. Es decir, el repositorio mantiene una consistencia a lo largo de la ejecución del código encargado de hacer las solicitudes (*main.py*).

```
PS C:\Users\esteb\Desktop\Esteban - USACH\2024.1\DESTRI\DistribuidosLaboratorio2\GitHub> python .\main.py
Sent push message for arenasesteban with version None
Received response for arenasesteban: pull - Pull completed: Repository created. Version: 1.2000000000000002
Sent push message for BryanSalgado with version 1.0
Received response for BryanSalgado: pull - Pull completed: Updated to version 1.2000000000000002
Sent push message for TheWillyrex with version 1.1
Received response for TheWillyrex: pull - Pull completed: Updated to version 1.2000000000000002
Sent pull message for arenasesteban with version 1.2000000000000002
Received response for arenasesteban: push - Push denied: No repository found.
Sent pull message for BryanSalgado with version 1.2000000000000002
Received response for BryanSalgado: push - Push denied: Version mismatch. Please pull the latest version.
Sent pull message for TheWillyrex with version 1.2000000000000002
Received response for TheWillyrex: push - Push accepted. New version: 1.2000000000000002
PS C:\Users\esteb\Desktop\Esteban - USACH\2024.1\DESTRI\DistribuidosLaboratorio2\GitHub>
```

Cada salida mostrada por consola muestra la resolución de la acción, pudiendo ser aceptada o denegada. Además, añade un *status* en el cual describe lo que ocurrió luego de ser evaluada la ejecución, esto depende justamente de la versión presente en la máquina del cliente comparada con las diferentes copias del servidor.

Por otro lado, en el código *server.py*, en cada lectura de un mensaje es mostrado su contenido, junto con una salida por la consola que confirma que este mensaje será procesado. Es cierto que, en este apartado, podría ser añadida un método para comprobar que la consistencia se está siendo lograda.

```
PS C:\Users\esteb\Desktop\Esteban - USACH\2024.1\DESTRI\DistribuidosLaboratorio2\GitHub> python .\server.py
Message received: {"action": "push", "client": {"username": "arenasesteban", "repository": null}}
Processing message from arenasesteban with action push
Message received: {"action": "push", "client": {"username": "BryanSalgado", "repository": {"name": "LaboratorioDistribuidos", "version": "1.0"}}}
Processing message from BryanSalgado with action push
Message received: {"action": "push", "client": {"username": "TheWillyrex", "repository": {"name": "LaboratorioDistribuidos", "version": "1.1"}}}
Processing message from TheWillyrex with action push
Message received: {"action": "pull", "client": {"username": "arenasesteban", "repository": null}}
Processing message from arenasesteban with action pull
Message received: {"action": "pull", "client": {"username": "BryanSalgado", "repository": {"name": "LaboratorioDistribuidos", "version": "1.0"}}}
Processing message from BryanSalgado with action pull
Message received: {"action": "pull", "client": {"username": "TheWillyrex", "repository": {"name": "LaboratorioDistribuidos", "version": "1.1"}}}
Processing message from TheWillyrex with action pull
```

Pero hay que destacar que la lógica detrás de esto no es más que una simulación, la elección del servidor que será el que procese la solicitud, es manejada por un random. Este escoge un servidor aleatoriamente, y luego el repositorio que recibió cambio es copiada su información en las demás copias, así simulando una sincronización entre réplicas.

Es por esto, que basta con revisar las salidas mostradas hasta ahora, con las cuales confirmamos que la consistencia está siendo lograda con éxito. Si no fuese así el caso, no se obtendrían los resultados esperados, que son justamente los mostrados. Como trabajamos con ejemplos de datos pequeños, es posible predecir los resultados correctos.

6. CONCLUSIONES

Como fue mencionado en un inicio, hoy se tienen plataformas de uso cotidiano que, debido a su demanda y la cantidad de datos que entran y salen, deben adoptar este tipo de técnicas de replicación asegurando la consistencia de los datos en cada caso.

A pesar de que el objetivo del laboratorio era analizar estos conceptos en un caso de estudio, llevando al equipo de trabajo a desarrollar un "clon" del manejo de versiones presente en GitHub. El caso es realmente atractivo al presentar diferentes obstáculos, tal como es la abstracción de cómo funciona este sistema, pero el laboratorio le suma complejidad al tener que implementarlo utilizando tecnologías que no son manejadas.

Apache Kafka y Apache Spark son herramientas muy potentes, especialmente para lograr la comunicación y asegurar la consistencia de datos aplicando lógica antes de hacer alguna modificación. La verdad es que significaron un gran desafío, especialmente Spark, lo que implicó no conseguir el objetivo de hacer uso de esta tecnología para el procesamiento de datos. Como grupo, queda al debe este objetivo, ya que se conoce la importancia de manejar una tecnología como esta.

Sin embargo, fue posible lograr simular el funcionamiento de GitHub a una pequeña escala, pero queda claro que llevar este caso a la realidad significa un desafío mucho mayor. Esta plataforma es capaz de proporcionar su servicio a diferentes usuarios sin importar su ubicación geográfica, esto es logrando mediante la replicación en lugar de tener un servidor centralizado. Esta solución también permite reducir la latencia al presentar un servidor más cercano al usuario y, a su vez, al existir un mayor nuevo de servidor aumenta la tolerancia a posibles fallos.

Por último, mantener la misma información en todas las réplicas, sin duda significa un gran costo, que iría en aumento a medida que crezca el número de copias, lo que sería recomendable buscar alguna técnica de replicación más avanzada que cubra esta problemática.