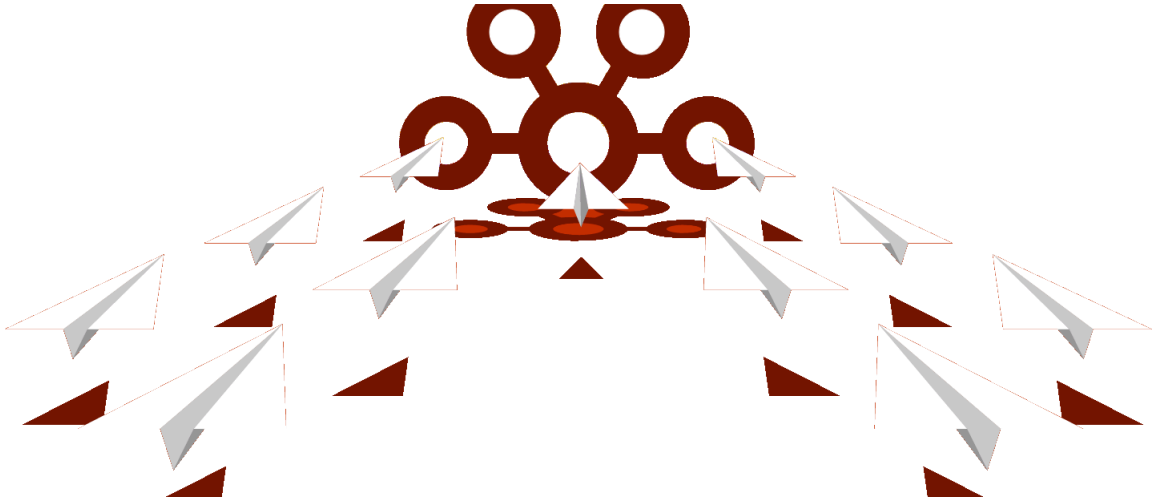
TP3 - La Collecte de Données avec le Bus Kafka



Télécharger PDF

Objectifs du TP

Utilisation de Kafka pour une collecte de données distribuée, et intégration avec Spark.

Outils et Versions

- Apache Kafka Version 2.13-3.6.1
- Apache Hadoop Version: 3.3.6
- Apache Spark Version: 3.5.0
- Docker Version *latest*
- Visual Studio Code Version 1.85.1 (ou tout autre IDE de votre choix)
- Java Version 1.8.
- Unix-like ou Unix-based Systems (Divers Linux et MacOS)

Kafka

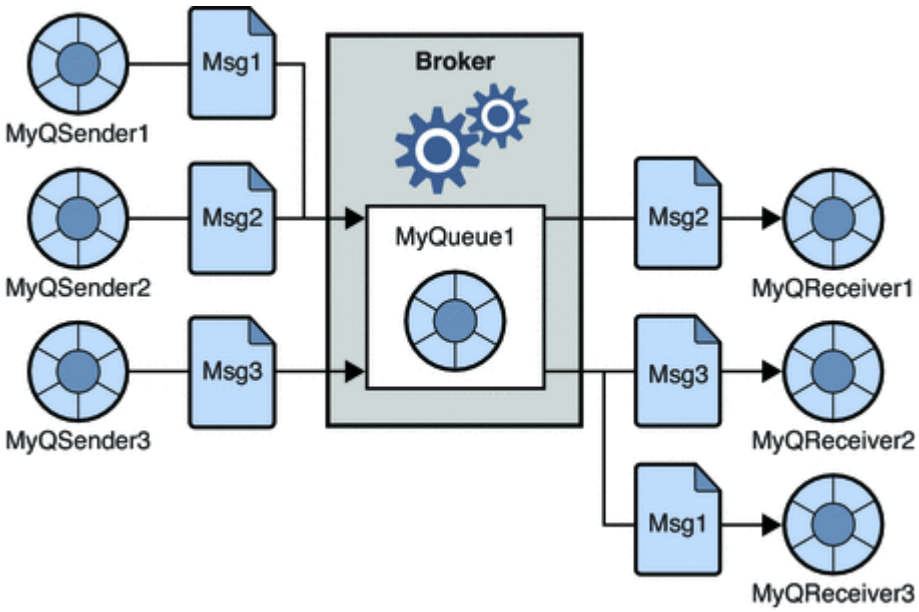
Qu'est-ce qu'un système de messaging?

Un système de messaging (*Messaging System*) est responsable du transfert de données d'une application à une autre, de manière à ce que les applications puissent se concentrer sur les données sans s'inquiéter de la manière de les partager ou de les collecter. Le messaging distribué est basé sur le principe de file de message fiable. Les messages sont stockés de manière asynchrone dans des files d'attente entre les applications clientes et le système de messaging.

Deux types de patrons de messaging existent: Les systèmes "*point à point*" et les systèmes "*publish-subscribe*".

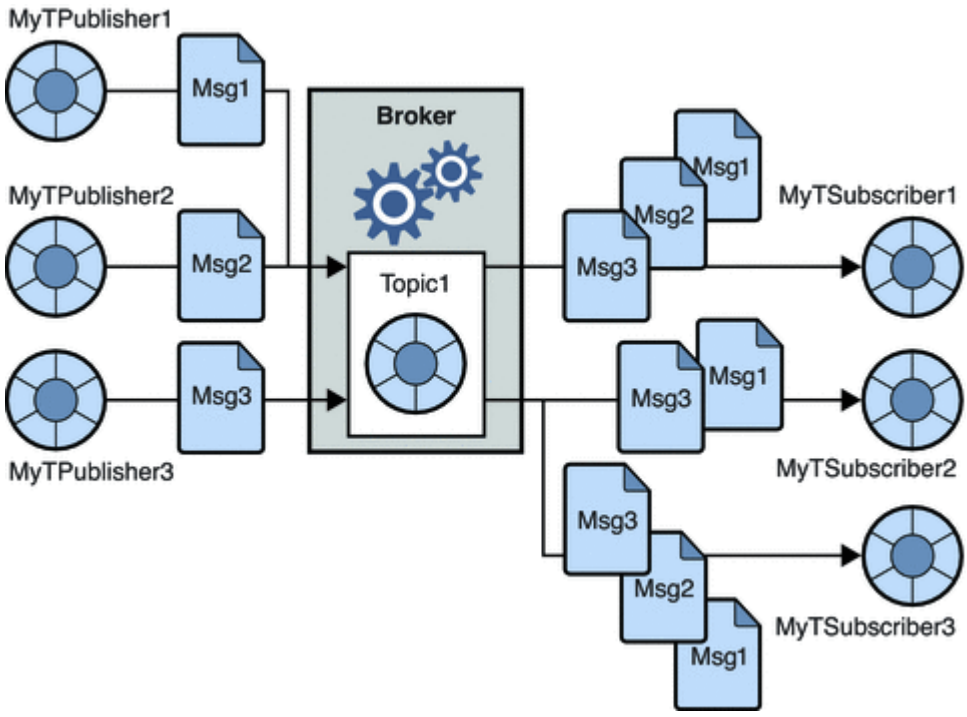
1. SYSTÈMES DE MESSAGING POINT À POINT

Dans un système point à point, les messages sont stockés dans une file. un ou plusieurs consommateurs peuvent consommer les message dans la file, mais un message ne peut être consommé que par un seul consommateur à la fois. Une fois le consommateur lit le message, ce dernier disparaît de la file.



2. SYSTÈMES DE MESSAGING PUBLISH/SUBSCRIBE

Dans un système publish-subscribe, les messages sont stockés dans un "*topic*". Contrairement à un système point à point, les consommateurs peuvent souscrire à un ou plusieurs topics et consommer tous les messages de ce topic.



Présentation de Kafka

Apache Kafka est une plateforme de streaming qui bénéficie de trois fonctionnalités:

1. Elle vous permet de publier et souscrire à un flux d'enregistrements. Elle ressemble ainsi à une file demessage ou un système de messaging d'entreprise.
2. Elle permet de stocker des flux d'enregistrements d'une façon tolérante aux pannes.
3. Elle vous permet de traiter (au besoin) les enregistrements au fur et à mesure qu'ils arrivent.

Les principaux avantages de Kafka sont:

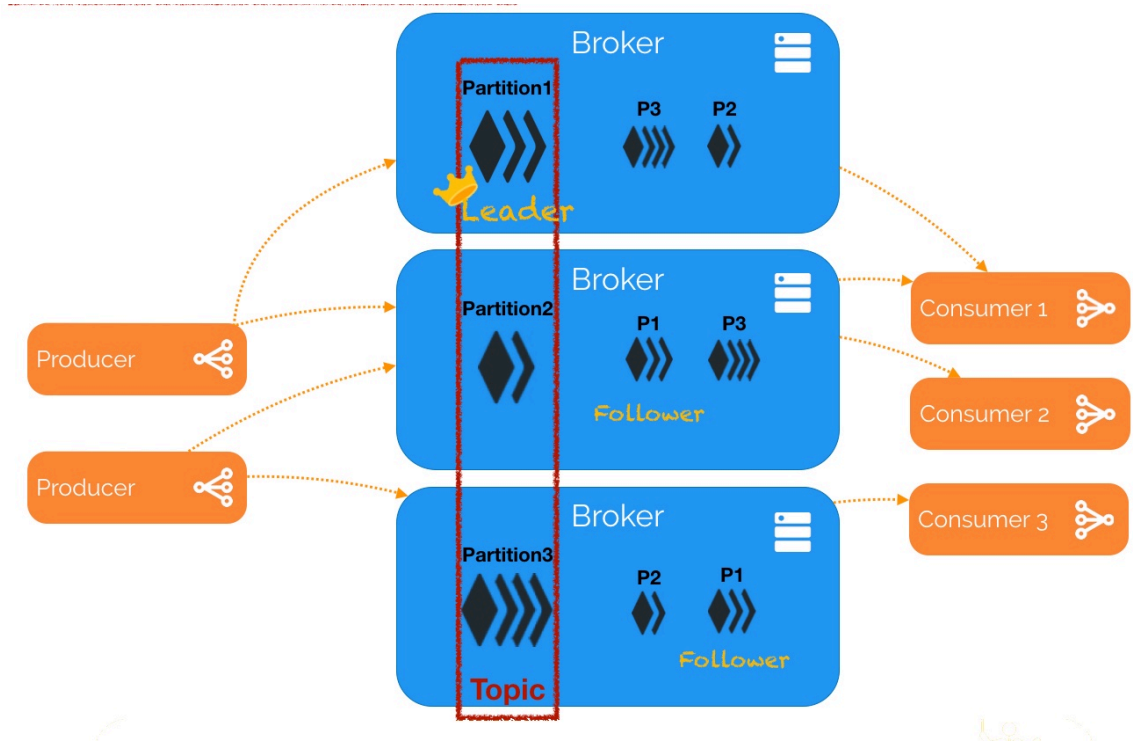
1. ***La fiablitié***: Kafka est distribué, partitionné, répliqué et tolérent aux fautes.
2. ***La scalabilité***: Kafka se met à l'échelle facilement et sans temps d'arrêt.
3. ***La durabilité***: Kafka utilise un *commit log* distribué, ce qui permet de stocker les messages sur le disque le plus vite possible.
4. ***La performance***: Kafka a un débit élevé pour la publication et l'abonnement.

Architecture de Kafka

Pour comprendre le fonctionnement de Kafka, il faut d'abord se familiariser avec le vocabulaire suivant:

1. **Topic**: Un flux de messages appartenant à une catégorie particulière. Les données sont stockées dans des topics.
2. **Partitions**: Chaque topic est divisé en partitions. Pour chaque topic, Kafka conserve un minimum d'une partition. Chaque partition contient des messages dans une séquence ordonnée immuable. Une partition est implémentée comme un ensemble de sègments de tailles égales.
3. **Offset**: Les enregistrements d'une partition ont chacun un identifiant séquentiel appelé *offset*, qui permet de l'identifier de manière unique dans la partition.
4. **Répliques**: Les répliques sont des *backups* d'une partition. Elles ne sont jamais lues ni modifiées par les acteurs externes, elles servent uniquement à prévenir la perte de données.
5. **Brokers**: Les *brokers* (ou courtiers) sont de simples systèmes responsables de maintenir les données publiées. Chaque courtier peut avoir zéro ou plusieurs partitions par topic. Si un topic admet N partitions et N courtiers, chaque courtier va avoir une seule partition. Si le nombre de courtiers est plus grand que celui des partitions, certains n'auront aucune partition de ce topic.
6. **Cluster**: Un système Kafka ayant plus qu'un seul Broker est appelé *cluster Kafka*. L'ajout de nouveau brokers est fait de manière transparente sans temps d'arrêt.
7. **Producers**: Les producteurs sont les éditeurs de messages à un ou plusieurs topics Kafka. Ils envoient des données aux courtiers Kafka. Chaque fois qu'un producteur publie un message à un courtier, ce dernier rattache le message au dernier sègment, ajouté ainsi à une partition. Un producteur peut également envoyer un message à une partition particulière.
8. **Consumers**: Les consommateurs lisent les données à partir des brokers. Ils souscrivent à un ou plusieurs topics, et consomment les messages publiés en extrayant les données à partir des brokers.
9. **Leaders**: Le leader est le noeud responsable de toutes les lectures et écritures d'une partition donnée. Chaque partition a un serveur jouant le rôle de leader.
10. **Follower**: C'est un noeud qui suit les instructions du leader. Si le leader tombe en panne, l'un des followers deviendra automatiquement le nouveau leader.

La figure suivante montre un exemple de flux entre les différentes parties d'un système Kafka:

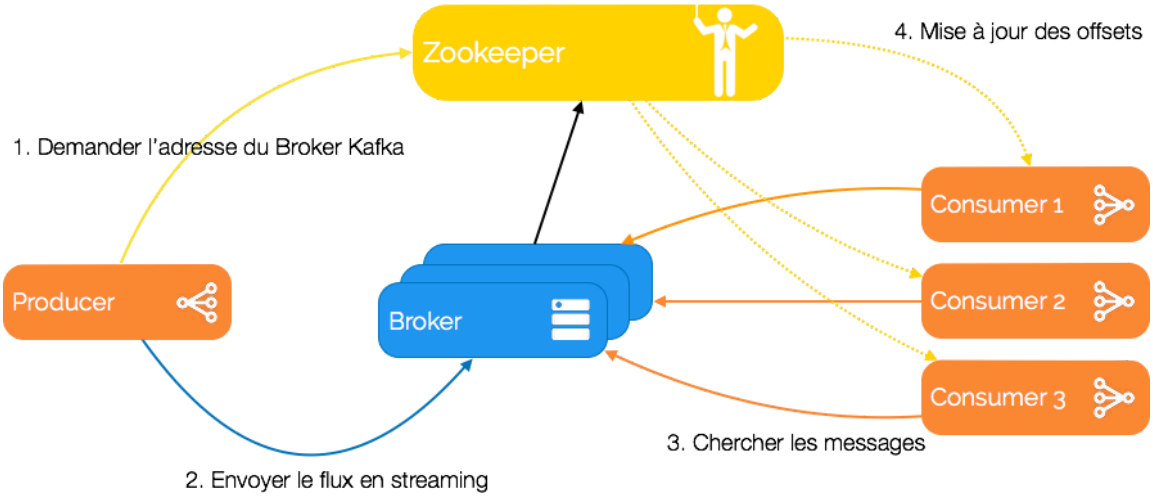


Dans cet exemple, un topic est configuré en trois partitions.

En supposant que, si le facteur de réplication du topic est de 3, alors Kafka va créer trois répliques identiques de chaque partition et les placer dans le cluster pour les rendre disponibles pour toutes les opérations. L'identifiant de la réplique est le même que l'identifiant du serveur qui l'héberge. Pour équilibrer la charge dans le cluster, chaque broker stocke une ou plusieurs de ces partitions. Plusieurs producteurs et consommateurs peuvent publier et extraire les messages au même moment.

Kafka et Zookeeper

Zookeeper est un service centralisé permettant de maintenir l'information de configuration, de nommage, de synchronisation et de services de groupe. Ces services sont utilisés par les applications distribuées en général, et par Kafka en particulier. Pour éviter la complexité et difficulté de leur implémentation manuelle, Zookeeper est utilisé.



Un cluster Kafka consiste typiquement en plusieurs courtiers (Brokers) pour maintenir la répartition de charge. Ces courtiers sont stateless, c'est pour cela qu'ils utilisent Zookeeper pour maintenir l'état du cluster. Un courtier peut gérer des centaines de milliers de lectures et écritures par seconde, et chaque courtier peut gérer des téra-octets de messages sans impact sur la performance.

Zookeeper est utilisé pour gérer et coordonner les courtiers Kafka. Il permet de notifier les producteurs et consommateurs de messages de la présence de tout nouveau courtier, ou de l'échec d'un courtier dans le cluster.

Il est à noter que les nouvelles versions de Kafka abandonnent petit à petit Zookeeper pour une gestion interne des métadonnées, grâce au protocole de consensus appelé KRaft (Kafka Raft).

Installation

Kafka a été installé sur le même cluster que les deux TP précédents. Suivre les étapes décrites dans la partie *Installation* du TP1 pour télécharger l'image et exécuter les trois contenaires. Si cela est déjà fait, il suffit de lancer vos machines grâce aux commandes suivantes:

```
docker start hadoop-master hadoop-worker1 hadoop-worker2
```

puis d'entrer dans le contenaire master:

```
docker exec -it hadoop-master bash
```

Lancer ensuite les démons yarn et hdfs:

```
./start-hadoop.sh
```

Lancer Kafka et Zookeeper en tapant :

```
./start-kafka-zookeeper.sh
```

Les deux démons Kafka et Zookeeper seront lancés. Vous pourrez vérifier cela en tapant `jps` pour voir quels processus Java sont en exécution, vous devriez trouver les processus suivants (en plus des processus Hadoop usuels):

```
2756 Kafka
53 QuorumPeerMain
```

Première utilisation de Kafka

Création d'un topic

Pour gérer les topics, Kafka fournit une commande appelée `kafka-topics.sh`. Dans un nouveau terminal, taper la commande suivante pour créer un nouveau topic appelé "Hello-Kafka".

```
kafka-topics.sh --create --topic Hello-Kafka --replication-factor 1 --
partitions 1 --bootstrap-server localhost:9092
```

> ⚠ **Attention**
>
> Cette commande fonctionne car nous avions rajouté /usr/local/kafka/bin à la variable d'environnement PATH. Si ce n'était pas le cas, on aurait du appeler /usr/local/kafka/bin/kafka-topics.sh

Pour afficher la liste des topics existants, il faudra utiliser:

```
kafka-topics.sh --list --bootstrap-server localhost:9092
```

Le résultat devrait être :

```
Hello-Kafka
```

Exemple Producteur Consommateur

Kafka fournit un exemple de producteur standard que vous pouvez directement utiliser. Il suffit de taper:

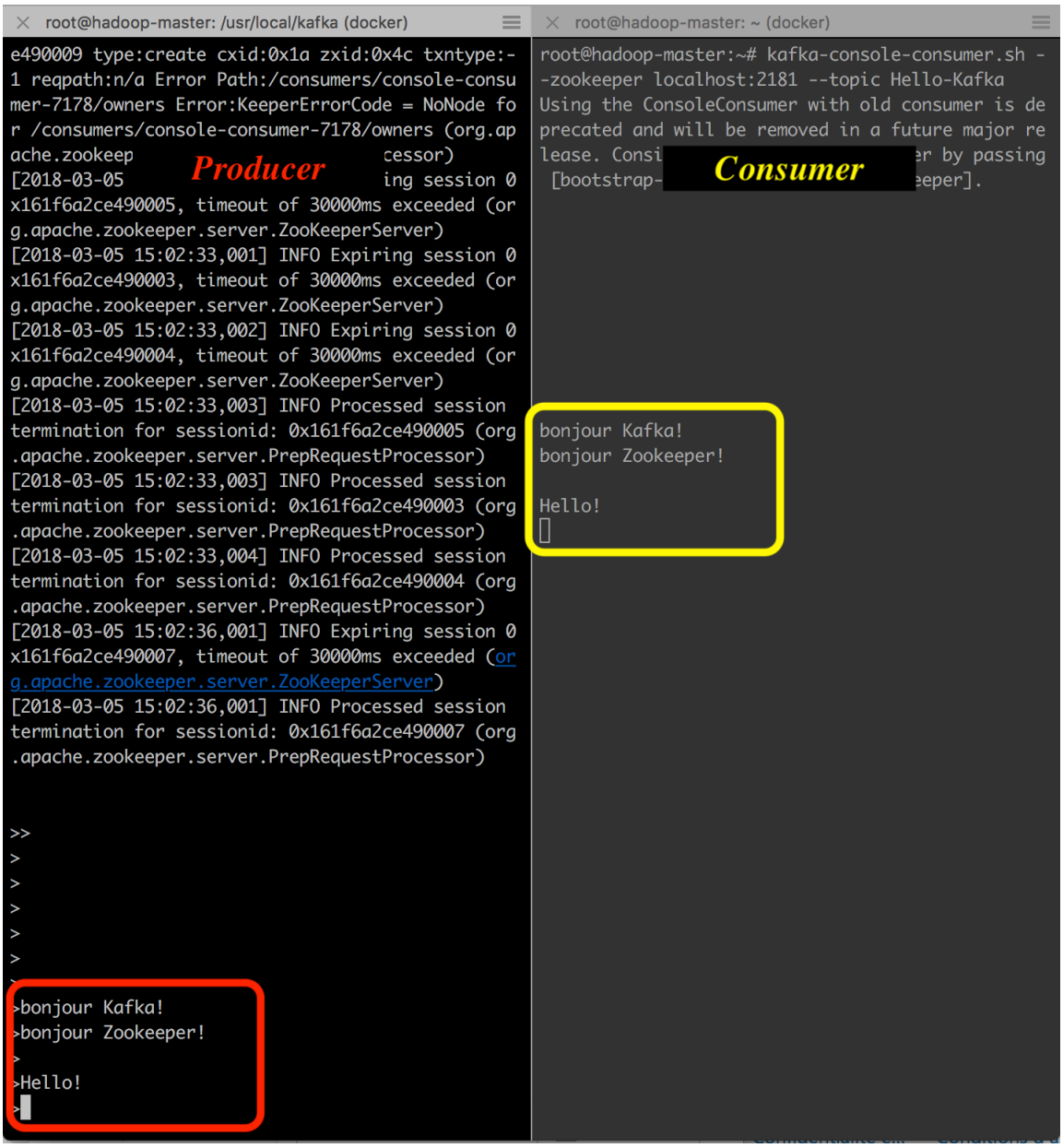
```
kafka-console-producer.sh --broker-list localhost:9092 --topic Hello-Kafka
```

Tout ce que vous taperez dorénavant sur la console sera envoyé à Kafka. L'option `--broker-list` permet de définir la liste des courtiers auxquels vous enverrez le message. Pour l'instant, vous n'en disposez que d'un, et il est déployé à l'adresse localhost:9092.

Pour lancer le consommateur standard, utiliser:

```
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic Hello-
Kafka --from-beginning
```

Le résultat devra ressembler au suivant:



Configuration de plusieurs brokers

Dans ce qui précède, nous avons configuré Kafka pour lancer un seul broker. Pour créer plusieurs brokers, il suffit de dupliquer le fichier `$KAFKA_HOME/config/server.properties` autant de fois que nécessaire. Dans notre cas, nous allons créer deux autre fichiers: `server-one.properties` et `server-two.properties`, puis nous modifions les paramètres suivants comme suit:

```
### config/server-one.properties
broker.id = 1
listeners=PLAINTEXT://localhost:9093
log.dirs=/tmp/kafka-logs-1

### config/server-two.properties
broker.id = 2
listeners=PLAINTEXT://localhost:9094
log.dirs=/tmp/kafka-logs-2
```

Pour démarrer les différents brokers, il suffit d'appeler `kafka-server-start.sh` avec les nouveaux fichiers de configuration.

```
 kafka-server-start.sh $KAFKA_HOME/config/server.properties &
 kafka-server-start.sh $KAFKA_HOME/config/server-one.properties &
 kafka-server-start.sh $KAFKA_HOME/config/server-two.properties &
```

Lancer `jps` pour voir les trois serveurs s'exécuter.

Création d'une application personnalisée

Nous allons dans cette partie créer une application pour publier et consommer des messages de Kafka. Pour cela, nous allons utiliser KafkaProducer API et KafkaConsumer API.

Producteur

Pour créer un producteur Kafka, créer un fichier dans un répertoire de votre choix dans le contenaire master, intitulé `SimpleProducer.java`. Son code est le suivant:

```
import java.util.Properties;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

 public class SimpleProducer {

   public static void main(String[] args) throws Exception{

      // Verifier que le topic est donne en argument
      if(args.length == 0){
         System.out.println("Entrer le nom du topic");
         return;
      }

      // Assigner topicName a une variable
      String topicName = args[0].toString();

      // Creer une instance de proprietes pour acceder aux configurations du
producteur
      Properties props = new Properties();

      // Assigner l'identifiant du serveur kafka
```

```
      props.put("bootstrap.servers", "localhost:9092");

      // Definir un acquittement pour les requetes du producteur
      props.put("acks", "all");

      // Si la requete echoue, le producteur peut reessayer automatiquemt
      props.put("retries", 0);

      // Specifier la taille du buffer size dans la config
      props.put("batch.size", 16384);

      // buffer.memory controle le montant total de memoire disponible au
producteur pour le buffering
      props.put("buffer.memory", 33554432);

      props.put("key.serializer",
         "org.apache.kafka.common.serialization.StringSerializer");

      props.put("value.serializer",
         "org.apache.kafka.common.serialization.StringSerializer");

      Producer<String, String> producer = new KafkaProducer
         <String, String>(props);

      for(int i = 0; i < 10; i++)
         producer.send(new ProducerRecord<String, String>(topicName,
            Integer.toString(i), Integer.toString(i)));
               System.out.println("Message envoye avec succes");
               producer.close();
   }
 }
```

ProducerRecord est une paire clef/valeur envoyée au cluster Kafka. Son constructeur peut prendre 4, 3 ou 2 paramètres, selon le besoin. Les signatures autorisées sont comme suit:

```
public ProducerRecord (string topic, int partition, k key, v value){...}
public ProducerRecord (string topic, k key, v value){...}
public ProducerRecord (string topic, v value){...}
```

Pour compiler ce code, taper dans la console (en vous positionnant dans le répertoire qui contient le fichier SimpleProducer.java):

```
javac -cp "$KAFKA_HOME/libs/*":. SimpleProducer.java
```

Lancer ensuite le producer en tapant:

```
java -cp "$KAFKA_HOME/libs/*":. SimpleProducer Hello-Kafka
```

Pour voir le résultat saisi dans Kafka, il est possible d'utiliser le consommateur prédéfini de Kafka, à condition d'utiliser le même topic:

```
kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic Hello-
Kafka --from-beginning
```

Le résultat devrait contenir les données suivantes :

```
0
1
2
3
4
5
6
7
8
9
```

Consommateur

Pour créer un consommateur, procéder de même. Créer un fichier `SimpleConsumer.java`, avec le code suivant:

```
import java.util.Properties;
import java.util.Arrays;
import java.time.Duration;

import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class SimpleConsumer {
  public static void main(String[] args) throws Exception {
    if(args.length == 0){
       System.out.println("Entrer le nom du topic");
       return;
    }
    String topicName = args[0].toString();
    Properties props = new Properties();

    props.put("bootstrap.servers", "localhost:9092");
    props.put("group.id", "test");
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("session.timeout.ms", "30000");
    props.put("key.deserializer",
       "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
       "org.apache.kafka.common.serialization.StringDeserializer");

    KafkaConsumer<String, String> consumer = new KafkaConsumer
       <String, String>(props);

    // Kafka Consumer va souscrire a la liste de topics ici
    consumer.subscribe(Arrays.asList(topicName));
```

```
    // Afficher le nom du topic
    System.out.println("Souscris au topic " + topicName);
    int i = 0;

    while (true) {
       ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
       for (ConsumerRecord<String, String> record : records)

       // Afficher l'offset, clef et valeur des enregistrements du
consommateur
       System.out.printf("offset = %d, key = %s, value = %s\n",
          record.offset(), record.key(), record.value());
    }
  }
}
```

Compiler le consommateur avec:

```
javac -cp "$KAFKA_HOME/libs/*":. SimpleConsumer.java
```

Puis l'exécuter:

```
java -cp "$KAFKA_HOME/libs/*":. SimpleConsumer Hello-Kafka
```

Le consommateur est maintenant à l'écoute du serveur de messagerie.

Ouvrir un nouveau terminal et relancer le producteur que vous aviez développé tout à l'heure. Le résultat dans le consommateur devrait ressembler à ceci.

```
offset = 32, key = 0, value = 0
offset = 33, key = 1, value = 1
offset = 34, key = 2, value = 2
offset = 35, key = 3, value = 3
offset = 36, key = 4, value = 4
offset = 37, key = 5, value = 5
offset = 38, key = 6, value = 6
offset = 39, key = 7, value = 7
offset = 40, key = 8, value = 8
offset = 41, key = 9, value = 9
```

Intégration de Kafka avec Spark

Utilité

Kafka représente une plateforme potentielle pour le messaging et l'intégration de Spark streaming. Kafka agit comme étant le hub central pour les flux de données en temps réel, qui sont ensuite traités avec des algorithmes complexes par Spark Streaming. Une fois les données traitées, Spark Streaming peut publier les résultats dans un autre topic Kafka ou les stocker dans HDFS, d'autres bases de données ou des dashboards.

Réalisation

Pour faire cela, nous allons réaliser un exemple simple, où Spark Streaming consomme des données de Kafka pour réaliser l'éternel wordcount.

Dans votre machine locale, ouvrir votre IDE préféré et créer un nouveau projet Maven, avec les propriétés suivantes:

```
groupId: spark.kafka
artifactId: stream-kafka-spark
version: 1
```

Une fois le projet créé, modifier le fichier pom.xml pour qu'il ressemble à ce qui suit:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>spark.kafka</groupId>
    <artifactId>stream-kafka-spark</artifactId>
    <version>1</version>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.12</artifactId>
            <version>3.5.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.12</artifactId>
            <version>3.5.0</version>
            <scope>provided</scope>
        </dependency>
       <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql-kafka-0-10_2.12</artifactId>
            <version>3.5.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.12</artifactId>
            <version>3.5.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
          <groupId>org.apache.kafka</groupId>
          <artifactId>kafka-clients</artifactId>
          <version>3.6.1</version>
      </dependency>
```

```
    </dependencies>

    <build>
        <sourceDirectory>src/main/java</sourceDirectory>
        <testSourceDirectory>src/test/java</testSourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>

<mainClass>spark.kafka.SparkKafkaWordCount</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Le plugin *maven-assembly-plugin* est utile pour pouvoir créer un jar contenant toutes les dépendances du projet.

Créer ensuite une classe *SparkKafkaWordCount* sous le package spark.kafka. Le code de cette classe sera comme suit:

```
package spark.kafka;

import java.util.Arrays;

import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Encoders;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

import scala.Tuple2;


public class SparkKafkaWordCount {
```

```
    public static void main(String[] args) throws Exception {


        if (args.length < 3) {
            System.err.println("Usage: SparkKafkaWordCount <bootstrap-servers>
<subscribe-topics> <group-id>");
            System.exit(1);
        }

        String bootstrapServers = args[0];
        String topics = args[1];
        String groupId = args[2];

        SparkSession spark = SparkSession
                .builder()
                .appName("SparkKafkaWordCount")
                .getOrCreate();

        // Create DataFrame representing the stream of input lines from Kafka
        Dataset<Row> df = spark
                .readStream()
                .format("kafka")
                .option("kafka.bootstrap.servers", bootstrapServers)
                .option("subscribe", topics)
                .option("kafka.group.id", groupId)
                .load();

        df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
                .as(Encoders.tuple(Encoders.STRING(), Encoders.STRING()))
                .flatMap((FlatMapFunction<Tuple2<String, String>, String>)
value -> Arrays.asList(value._2.split(" ")).iterator(), Encoders.STRING())
                .groupByKey((MapFunction<String, String>) value -> value,
Encoders.STRING())
                .count()
                .writeStream()
                .outputMode("complete")
                .format("console")
                .start()
                .awaitTermination();
    }
}
```

Ce code est un exemple d'application Spark utilisant l'intégration de Spark avec Kafka pour effectuer un comptage de mots en temps réel (WordCount) à partir des messages Kafka. Voici ce que fait chaque partie du code :

1. **Initialisation de SparkSession** : Le code commence par créer une instance de SparkSession, nécessaire pour exécuter toute application Spark.
2. **Paramètres d'entrée** : Le programme attend trois arguments - l'adresse des serveurs Kafka (*bootstrapServers*), les topics auxquels s'abonner (*topics*) et l'identifiant du groupe consommateur Kafka (*groupId*).
3. **Lecture des données de Kafka** : Le DataFrame df est créé en lisant le flux de données à partir de Kafka en utilisant les paramètres fournis. Les données lues incluent key et value

pour chaque message Kafka.

4. **Traitement des données** :
    - Le code transforme les données en castant *key* et *value* en chaînes de caractères.
    - Ensuite, il utilise la fonction *flatMap* pour diviser chaque valeur (chaque ligne de texte des messages Kafka) en mots individuels.
    - Il utilise *groupByKey* pour regrouper les mots identiques.
5. **Comptage et écriture** : Il compte le nombre d'occurrences de chaque mot (*count*) et écrit le résultat du comptage en continu à la console (*writeStream.format("console")*).
6. **Exécution** : L'application démarre le traitement du flux avec *start*() et attend que le traitement soit terminé avec *awaitTermination*().

Nous allons ensuite créer une configuration Maven pour lancer la commande:

```
mvn clean compile assembly:single
```

Dans le répertoire target, un fichier `stream-kafka-spark-1-jar-with-dependencies.jar` est créé. Copier ce fichier dans le contenaire master, en utilisant le terminal comme suit:

```
docker cp target/stream-kafka-spark-1-jar-with-dependencies.jar hadoop-
master:/root/stream-kafka.jar
```

Revenir à votre contenaire master, et lancer la commande `spark-submit` pour lancer l'écouteur de streaming spark.

```
spark-submit --class spark.kafka.SparkKafkaWordCount --master local  stream-
kafka.jar localhost:9092 Hello-Kafka mySparkConsumerGroup >> out
```

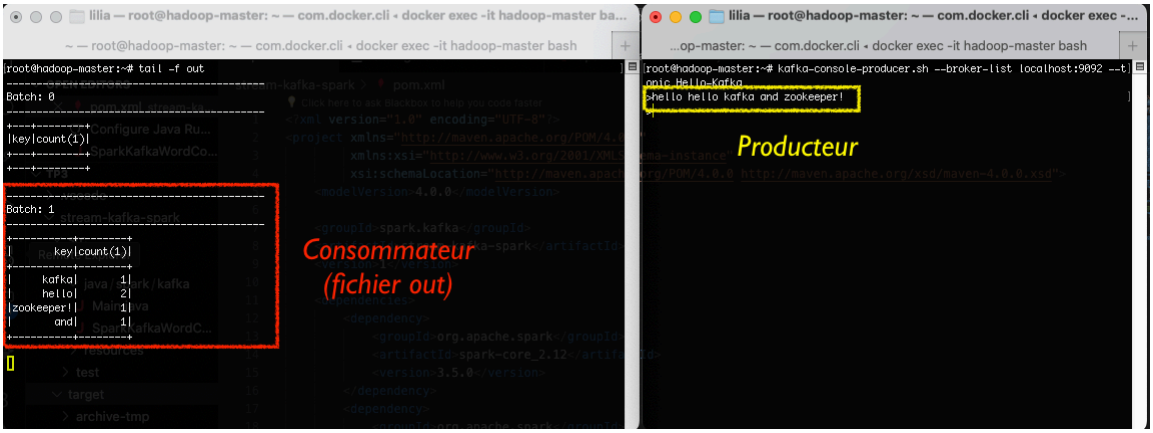
Les trois options à la fin de la commande sont requises par la classe *SparkKafkaWordCount* et représentent respectivement l'adresse du broker, le nom du topic et l'identifiant du groupe de consommateurs Kafka (ensemble de consommateurs (processus ou threads) qui s'abonnent aux mêmes topics et qui travaillent ensemble pour consommer les données).

> **Remarque**
>
> >>out est utilisé pour stocker les résultats produits par spark streaming dans un fichier appelé *out*.

Dans un autre terminal, lancer le producteur prédéfini de Kafka pour tester la réaction du consommateur spark streaming:

```
kafka-console-producer.sh --broker-list localhost:9092 --topic Hello-Kafka
```

Ecrire du texte dans la fenêtre du producteur. Ensuite, arrêter le flux de spark-submit, et observer le contenu du fichier out. Il devra ressembler à ce qui suit:



Projet

Étape 3

Pour votre projet, vous allez utiliser Kafka pour gérer les flux entrants et les envoyer à Spark. Ces mêmes données (ou une partie de ces données) peuvent également être stockées dans HDFS pour un traitement par lot ultérieur. Réaliser les liaisons nécessaires entre Kafka et Spark, puis Kafka et HDFS.

Last update: 2024-03-24