Gestion de données massives - Kafka

Rémy Barranco
Yoan Dusoleil

Mars 2025

Sommaire

1 Introduction

Au travers de ce projet, nous avons abordé la mise en place d'applications Java communiquant via Apache Kafka. Dans une seconde partie, nous avons exploité les capacités de stream processing de Kafka pour le calcul de valeurs dérivées depuis des valeurs de flux de données.

L'architecture système pour la partie Kafka contient :

- 1 noeud Zookeeper
- 2 noeuds broker Kafka

Les noeuds sont déployés à l'aide de Docker.

Ce rapport se concentrera sur les programmes producteurs et consommateurs en Java permettant de mettre en évidence et d'exploiter les différents mécanismes de communication et de stream processing de Kafka.

Le document est structuré de la manière suivante :

- Exercice 2 : mise en évidence des différences dans le consommation d'une masse de message en focntion du nombre de consommateurs et du nombre de groupes de consommateurs puis comparaison avec des brokers de messages basés sur JMS.
- Exercice 3 : utilisation du stream processing pour calculer une valeur moyenne depuis les valeurs du flux de données sur une fenêtre glissante de 5 minutes.

2 Exercice 2 : Benchmark

Dans cette partie, nous allons aborder l'envoi et la consommation de messages à partir de programmes Java. Différentes stratégies de consommation seront abordées afin de mettre en évidence certaines spécificités de Kafka.

2.1 Objectifs et Architecture

Ces tests se feront sur un topic contenant 2 partitions et avec un facteur de réplication de 2.

L'objectif est de produire un grand nombre de messages (jusqu'à 10^6) dans un topic et de mesurer les différents temps de consommation des différentes stratégies suivantes :

1. Utilisation de 2 consommateurs dans le même groupe de consommateurs.
2. Idem avec 3 consommateurs dans le même groupe.
3. Enfin 2 consommateurs dans 2 groupes différents.

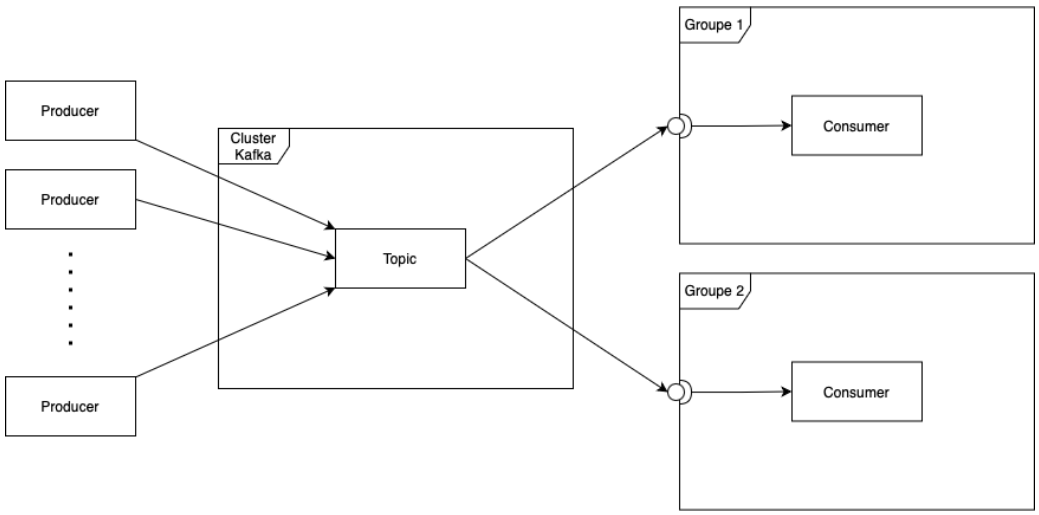

Figure 1: Schéma de l'architecture de test pour un seul consommateur par groupe

2.2 Implémentation du Producteur

Les producteurs sont implémentés sous forme de Threads envoyant une quantité définie à l'instanciation de messages. Par défaut, le programme crée 10 producteurs qui envoient chacun 100 000 messages.

Les messages sont envoyés avec une clé au format String et une valeur au format String. Chaque valeur et clé est différente.

Le code ci-dessous illustre le fonctionnement du producteur :

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());

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

try {
    for (int i = 0; i < messageCount; i++) {
        String key = "Key-" + this.uuid.toString() +  "-" + i;
        String value = "Message-" + i;
        ProducerRecord<String, String> record = new ProducerRecord<>(
   topicName, key, value);
        producer.send(record);
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    producer.close();
}
```

2.3 Implémentation du Consommateur

Afin de pouvoir réaliser les benchmarks sur la partie consommation, les messages sont envoyés dans le topic en amont puis seulement ensuite, les messages sont consommés.

Afin de s'assurer de commencer la lecture du topic depuis le début à chaque fois, on paramètre les propriétés `AUTO_OFFSET_RESET_CONFIG` à `"earliest"` et `ENABLE_AUTO_COMMIT_CONFIG` à `"false`.

Lors de l'utilisation de la méthode `subscribe()` des `KafkaConsumer`, Kafka assigne des partitions en fonction du group_id du consommateur, ce qui peut entraîner dans certains cas un délai avant l'assignation de la partition à un consommateur. Afin de s'assurer que chaque consommateur a une partition qui lui est assignée avant de consommer les messages, le thread est mis en attente tant qu'il n'a pas d'assignation puis on lui force un offset à 0.

```
KafkaConsumer<String, String> consumer =
    new KafkaConsumer<>(properties);
consumer.subscribe(Collections.singletonList(topicName));

while (consumer.assignment().isEmpty()) {
    consumer.poll(Duration.ofMillis(100));
}
consumer.assignment()
    .forEach(partition -> consumer.seek(partition, 0));
```

2.4 Benchmarks

La réalisation des benchmarks se fait de la manière suivante :

- Création d'une ConcurrentHashMap dans laquelle les messages lus par les consommateurs seront insérés avec pour clé un UUID aléatoire afin d'avoir une trace de tous les messages lus.
- Instanciation des consommateurs prenant une référence à la ConcurrentHashMap en paramètre du constructeur.
- Démarrage de la mesure du temps puis attente de la fin des Threads.

Pour valider l'installation et mesurer les performances, nous avons réalisé :

1. Un test avec un groupe unique (p.ex. group_id = `"benchmark_group"`), mais plusieurs consommateurs (2 ou 3) dans ce même groupe.
    - Chaque partition du topic est distribuée entre les consommateurs du même groupe.
    - On observe une répartition automatique des messages (chaque partition est gérée par un seul consommateur).
2. Un test avec plusieurs groupes (p.ex. `"benchmark_group_1"`, `"benchmark_group_2"`, etc.).
    - Chaque groupe lit l'intégralité des partitions du topic, car les offsets sont propres à chaque groupe.
    - On observe plus de redondance de consommation, puisque tous les groupes lisent tous les messages.

Stratégie	Temps (ms)	Messages lus	Consommateur par groupe
2 consommateurs, 1 groupe	4910	1,000,000	2
3 consommateurs, 1 groupe	7921	2,000,000	3
2 consommateurs, 2 groupes	6034	2,000,000	1

Table 1: Resultats du benchmark

Quelques points remarquables :

- Avec des consommateurs dans un même groupe (et un facteur de réplication suffisant), la charge est répartie en parallélisant la lecture. On peut néanmoins observé un comportement différent entre le groupe de 2 et celui de 3 consommateurs. Cela se traduit par l'implémentation du consommateur, en effet, avec la propriété `ENABLE_AUTO_COMMIT_CONFIG` à "false", les consommateurs ne commit pas automatiquement leurs nouveaux offset après chaque lecture. Etant donné que les benchmarks sont réalisés sur un topic à 2 partitions, le 3ème consommateur attend que les 2 autres aient fini puis commence la lecture du topic depuis le début car les consommateurs n'ont jamais commit leurs offset. D'où la lecture de 2 000 000 de messages dans ce cas.
- Avec des groupes distincts, on peut s'apercevoir que chaque consommateur lira l'entièreté des 2 partitions et prend donc plus de temps.

2.5 Comparaison avec JMS

JMS (*Java Message Service*) est une spécification standard pour la messagerie asynchrone. Par rapport à JMS, Kafka se distingue entre autres par :

- La persistance native des logs et la rétention configurables (Kafka peut stocker les messages un certain temps, pas uniquement en mémoire).
- L'évolutivité horizontale et la **notion de partition** (répartition sur plusieurs brokers).
- Les **groupes de consommateurs** et la gestion des offsets, permettant de revenir sur l'historique des messages en cas de besoin.
- JMS se base souvent sur une approche *queue* ou *topic* plus classique, et dépend de l'implémentation (ActiveMQ, RabbitMQ, etc.).

En conclusion, Kafka est particulièrement adapté aux cas d'usage orientés *stream* et *big data* (volume massif, relecture possible, haute tolérance aux pannes) alors que JMS est souvent choisi pour des cas d'intégration d'entreprises plus classiques, avec un *broker* JMS répondant aux spécifications JavaEE.

3 Exercice 3 : Stream Processing

Dans cet exercice, il s'agissait de concevoir un système de **monitoring** de données de température, en exploitant **Kafka Streams**. Les bâtiments envoient régulièrement (toutes les 10 secondes) des mesures de température sur un topic Kafka, et un calcul de la **moyenne de température sur une fenêtre de 5 minutes** est réalisé via un job `KafkaStreams`.

3.1 Architecture Globale

La figure suivante illustre l'architecture d'ensemble :

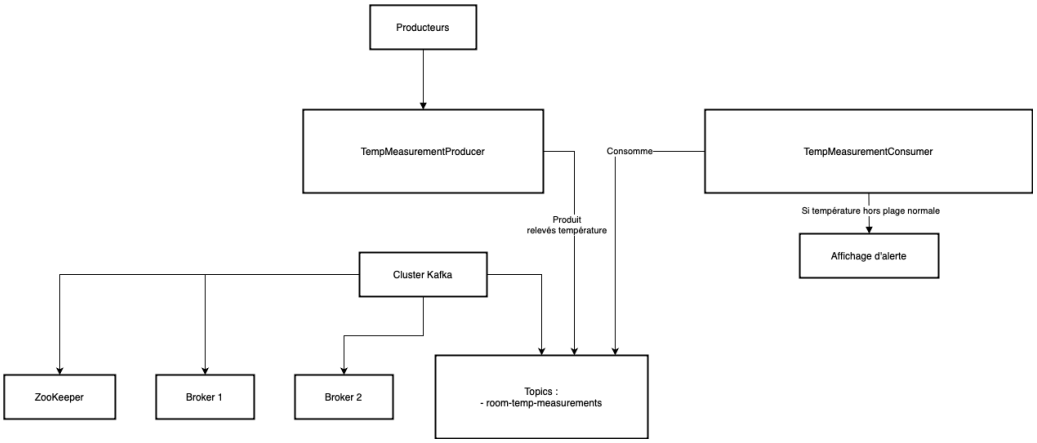

Figure 2: Architecture simplifiée de l'exercice 3 (Stream Processing)

1. **Plusieurs producteurs** Un producteur par salle (afin de simuler un capteur par salle) qui envoient des messages avev pour clé l'identifiant du batiment et pour valeur l'identifiant de la salle et la température mesurée.
2. Les données sont stockées dans un *topic* Kafka, partitionné en 2 partitions et avec un facteur de réplication de 2.
3. Un programme `Kafka Streams` lit les messages de ce *topic* et calcule la moyenne de température par salle sur une fenêtre glissante de 5 minutes.
4. Une **alerte** est affichée si la température est en dehors d'un interval voulu (exemple : $< 16°C$ ou $> 28°C$).

3.2 Producteurs des relevés

Les producteurs sont gérés par la classe `TempMeasurementProducer` (chaque thread correspond à un producteur). Ils envoient toutes les 10 secondes des messages de la forme :

$$\underbrace{\text{buildingId}}_{\text{clé}} \quad \rightarrow \quad \texttt{``roomID,temp''}$$

où `temp` est une valeur simulée démarrant à 20 avec une variation aléatoire dans l'intervalle [-2.5,+2.5] à chaque relevé selon la formule suivante :

```
1     double temp = previousTemp + Math.random() * 5 - 2.5;
```

La simulation prend en compte une légère variation aléatoire pour approcher le comportement d'un capteur réel.

3.3 Calcul de la moyenne sur fenêtre Kafka Streams

Côté consommation/agrégation, la classe `TempMeasurementConsumer` met en place le pipeline Kafka Streams :

- `builder.stream(topicName)` permet de récupérer le flux `KStream<cle,valeur>`.
- La valeur est parsée pour séparer `roomID` et `température` puis on regroupe par clé composite `(buildingID, roomID)`.
- On applique un `TimeWindow` de 5 minutes (`TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(5))`).
- La fonction `aggregate(...)` cumule la somme de températures et le nombre de lectures (`TemperatureData`).
- Enfin, on map les valeurs en une moyenne finale.

Un extrait de code simplifié :

```
KStream<Integer, String> source = builder.stream(topicName);

KStream<String, Double> roomTemperatures = source.flatMap((buildingId,
    value) -> {
   String[] parts = value.split(",");
   String roomId = parts[0];
   double temperature = Double.parseDouble(parts[1]);
   String compositeKey = buildingId + "," + roomId;
   return Collections.singletonList(KeyValue.pair(compositeKey,
   temperature));
});

TimeWindows fiveMinuteWindow = TimeWindows.ofSizeWithNoGrace(Duration.
   ofMinutes(5));

KTable<Windowed<String>, TemperatureData> aggregated =
   roomTemperatures
  .groupByKey()
  .windowedBy(fiveMinuteWindow)
  .aggregate(
     () -> new TemperatureData(),
     (room, temperature, agg) -> agg.addTemperature(temperature),
     Materialized.with(Serdes.String(), new JsonSerdes<TemperatureData
   >())
  );

aggregated.mapValues(agg -> agg.computeAverageTemperature())
          .toStream()
          .foreach((windowedKey, avg) -> {
             if(avg < 15 || avg > 25) {
               System.out.println("[ALERTE] Temperature anormale : " +
                                  avg + " pour la salle " +
                                  windowedKey.key()
                                  );
             }
          });
```

4 Conclusion

Dans ces deux exercices, nous avons :

- Mise en pratique de l'API Java de Kafka (producteur et consommateur).
- Observé comment la notion de *consumer group* permettait de paralléliser la consommation et d'implémenter différents cas d'usage.
- Comparé brièvement Kafka à JMS, en soulignant la persistance et la gestion d'historique de Kafka, ainsi que sa scalabilité.
- Mis en place un **Stream Processing** pour collecter des températures, calculer des moyennes sur des fenêtres de temps, et détecter des valeurs hors-seuil.

Ces exercices illustrent la puissance de Kafka, non seulement comme système de messagerie distribué, mais également comme plate-forme de **stream processing**, bien adaptée à l'analyse de flux de données temps réel (IOT, logs, métriques, etc.).

5 Perspective d'évolution

Lors de l'exercice 3, en cas de température moyenne en dehors de la plage de températures autorisées, l'alerte remontée est faite uniquement sous la forme d'un message affiché dans la console. Il pourrait être intéressant de générer cette alerte sous forme d'un message à envoyer dans un nouveau topic Kafka, permettant par exemple de contrôler en conséquence des objets connectés (actionneurs de fenêtres, thermostats, volets motorisés, etc...) afin de ramener la température dans la plage de températures normales.