Les notes ajoutées à certaines diapositives se trouvent à la fin du document.

Master 2 Informatique

Louis ALLAIN

Contenu

- Qu’est ce qu’Apache Kafka ?
- Principes de fonctionnement et d’utilisation
- Pourquoi utilisez Kafka
- Cas d’utilisation
- Présentation de l’API Producer et Consumer
- Mécanismes internes de Kafka
- Exemple simple

Qu’est ce qu’Apache Kafka ?

Apache Kafka est un projet open-source de système de messagerie “publish / subscribe” centralisé. Kafka est décrit comme étant un journal de transactions distribué.

Un journal de transaction est un enregistrement de toutes transactions de sorte que celles-ci peuvent être rejouées afin de construire l’état d’un système de manière consistante. De la même façon, les données dans Apache Kafka sont sauvegardés durablement, dans l’ordre et peuvent être lu de manière déterministe.

De plus les données peuvent être distribuées afin d’ajouter une certaine protection contre les erreurs, et dans le même temps permettre la croissance des applications.

Principes

Message :

- Unité primaire de Kafka
- Tableau d’octets => aucune signification et aucun format pour Kafka

Batch (lot) :

- Pour des raisons d’efficacité, les messages sont groupés par lots
- Un lot = collection de messages
- Créer afin d’éviter de propager message par message à travers tout le réseau
- Plus un lot est grand, plus le nombre de messages gérés par unité de temps sera grand mais la propagation d’un seul message sera plus longue

Principes

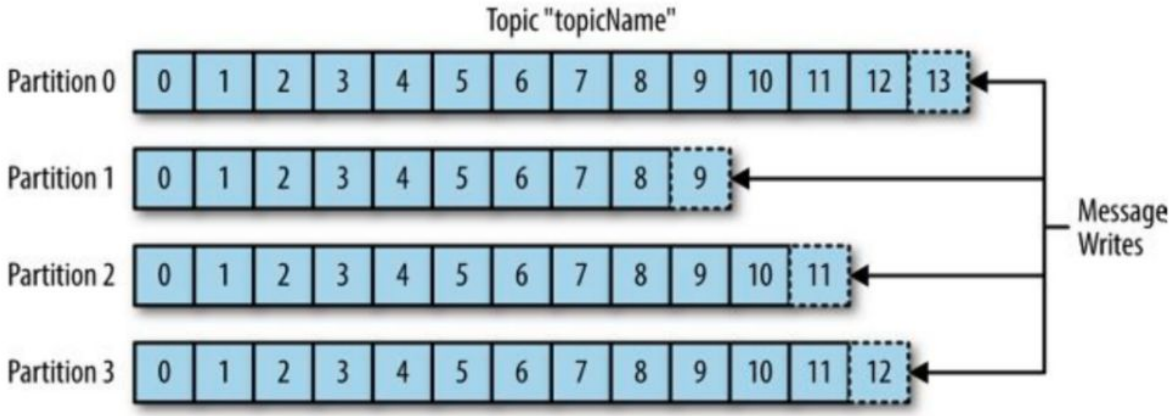
Schémas :

- Structure permettant de décrire le contenu du message
- Important dans Kafka car cela permet de découpler l’écriture de la lecture

Principes

Topics et Partitions :

- Les messages sont catégorisés dans des topics.
- Les topics sont de plus décomposer en plusieurs partitions.
- Permet la redondance et de facilement anticipé la montée en charge d'un système



Principes

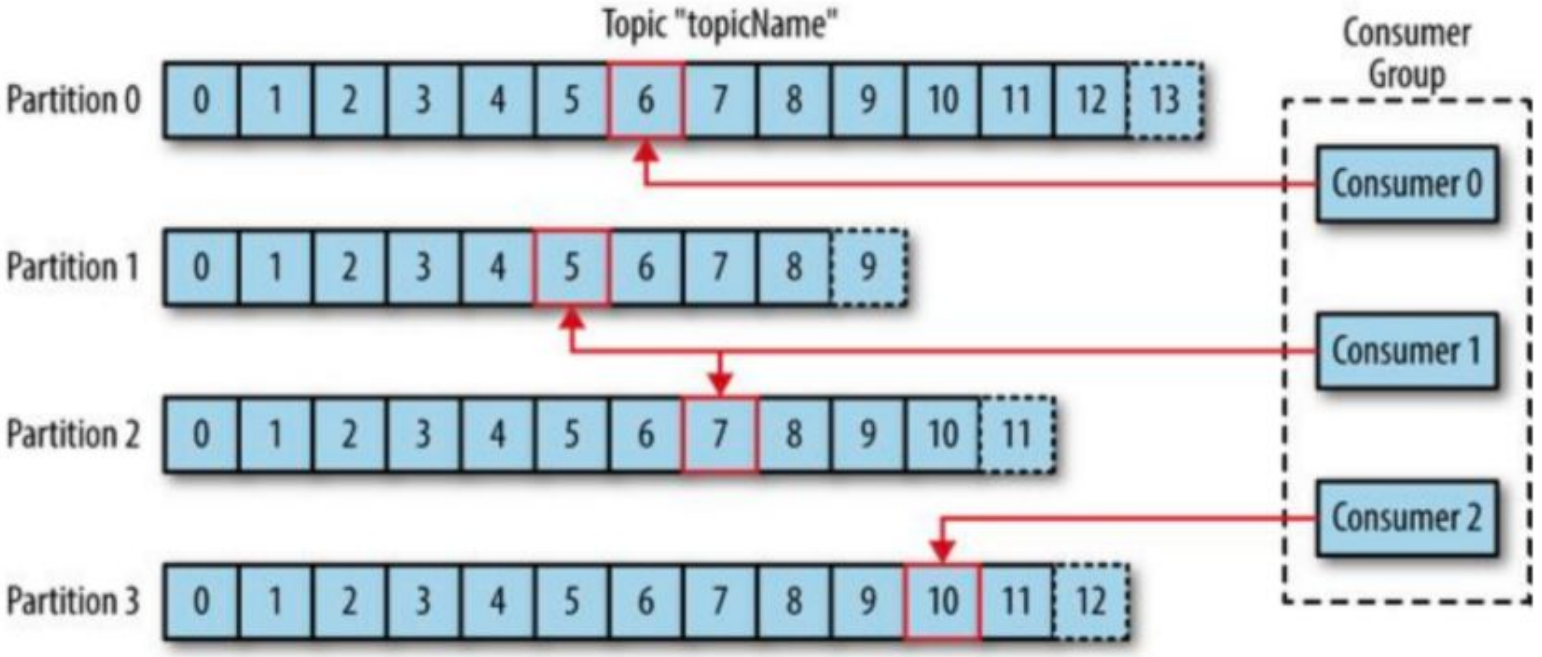
Producteurs :

- Créent de nouveaux messages

Consommateurs :

- S’abonnent à un ou plusieurs topics
- Lisent les messages dans l’ordre dans lequel ils ont été produits.
- Gardent une trace des messages déjà consommés en mémorisant l’offset des messages.
- Les consommateurs travaillent en faisant parti d’un groupe de consommateurs constitué d’au moins un consommateur => permet une meilleure croissance du système / montée en charge.

Principes


Topic "topicName"
Consumer Group
Partition 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13
Partition 1
0 1 2 3 4 5 6 7 8 9
Partition 2
0 1 2 3 4 5 6 7 8 9 10 11
Partition 3
0 1 2 3 4 5 6 7 8 9 10 11 12
Consumer 0
Consumer 1
Consumer 2

Principes

Broker Kafka :

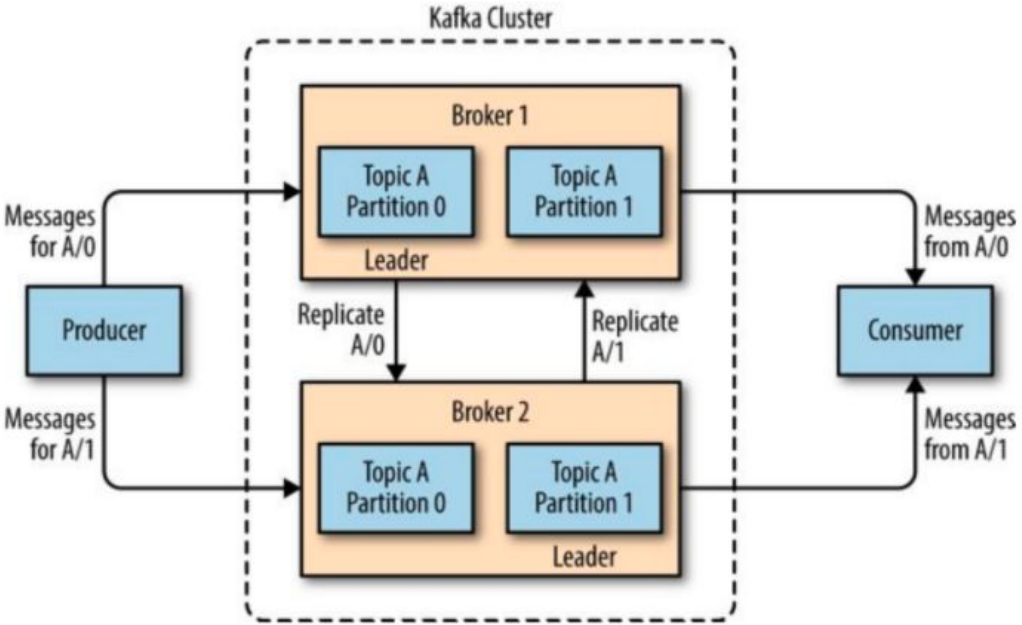
- Reçoit les messages des producteurs, leurs assignent un offset, et sauvegarde ces messages sur disque
- Sert les consommateurs en cherchant les partitions et en répondant avec les messages provenant du disque.
- Peut gérer des milliers de partitions et des millions de messages par secondes

Principes

Cluster (ensemble de brokers) :

- Les brokers sont conçus pour faire parti d'un cluster
- Parmi ce cluster : un contrôleur (élu automatiquement) chargé des opérations administratives (assigner les partitions aux Brokers, détecter les pannes, …)
- Une partition appartient à un seul broker (Leader d'une partition)
- Une partition peut être assignée à plusieurs brokers, ainsi les partitions sont répliquées entraînant une redondance des messages. Cette redondance permet en cas de panne d'un leader d'un partition de le remplacer par un autre broker.

Principes


Kafka Cluster
Broker 1
Topic A Partition 0
Topic A Partition 1
Leader
Broker 2
Topic A Partition 0
Topic A Partition 1
Leader
Producer
Messages for A/0
Messages for A/1
Replicate A/0
Replicate A/1
Consumer
Messages from A/0
Messages from A/1

Principes

Rétention :

- Fonctionnalité clef de Apache Kafka
- Sauvegarde durable des messages pour une certaine période
- Configuration de cette période pour les brokers :
  - Période de temps (ex : 7 jours) ou taille maximale d’un topic (ex : 1 Go)
- Une fois cette limite atteinte, les messages sont supprimés
- Possibilité de configurer cette période topic par topic
  - Ex : Topic de prise de mesure => données valables quelques heures

Pourquoi utiliser Kafka ?

Producteurs multiple :

- Kafka est capable de gérer beaucoup de Producteurs, qu'ils utilisent plusieurs topics différents ou le même. Idéal pour agréger des données provenant de beaucoup de sources.

Consommateurs multiple :

- Kafka est conçu de sorte que beaucoup de Consommateurs puissent lire n'importe quel flux de données sans interférer.
- Des Consommateurs Kafka peuvent choisir de faire parti d'un groupe partageant un flux de données et ainsi ils s'assurent que tout le groupe traite une seule fois un message.

Pourquoi utiliser Kafka ?

Rétention sur disque :

- Tous les messages sont sauvegardés (comme vu précédemment) selon des règles configurées pour chaque broker.
- Les consommateurs ne sont donc pas obligés de traiter les messages en temps réel
- Un consommateur peut tomber en panne sans être inquiéter de la perte de données
- Possibilité donc pour les applications consommants des données de s’arrêter pour maintenance pour une courte période de temps

Pourquoi utiliser Kafka ?

Montée en charge (Scalability) :

- Comme vu précédemment, les concepts de Kafka sont faits de sorte à permettre une croissance facile des applications.
- Possibilité d'étendre le nombre de brokers en fonction du nombre de données sans impacter la disponibilité des applications.

Haute performance :

- Apache Kafka est un agent de messagerie "publish/subscribe" très performant à forte charge

Cas d’utilisation de Kafka

Suivi d’activité :

- Interactions des utilisateurs sur un site générant des messages en fonction des actions telles que des click, des pages consultées, modification d’informations sur la page, etc.
- Ces messages sont publiés sur un ou plusieurs topics et sont ensuite consommés par des applications comme du machine learning, des rapports, ...

Cas d’utilisation de Kafka

Messagerie :

- Applications nécessitant d’envoyer des notifications aux utilisateurs
- Certaines applications produisent des messages de n’importe quel format
- Une application lit tous ces messages et les traite de manière à les formater proprement, combiner plusieurs messages afin de les transformer en une seule notification à envoyer, appliquer des préférences utilisateurs sur la manière dont ils souhaitent recevoir les messages.

Mesures et journal d’activité :

- Kafka peut être aussi utilisé pour agréger des données provenant d’applications distribuées.

Utilisation de Kafka : l’API Producer

Création d’un producteur :

```
private Properties kafkaProps = new Properties();

// Liste des brokers auxquels le Producer tente de se connecter; Il n’est pas nécessaire de tous les mettre puisque le
// Producer obtient plus d’informations sur le cluster après la connexion à un broker
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

// Nom de la classe permettant de sérialiser la clef des données produites
kafkaProps.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");

// Nom de la classe permettant de sérialiser la valeur des données produites
kafkaProps.put("value.serializer","org.apache.kafka.common.serialization.StringSerializer");


// Création du Producer avec les paramètres définis précédemment
producer = new KafkaProducer<String, String>(kafkaProps);
```

Utilisation de Kafka : l’API Producer

Méthode permettant d’envoyer des données à Kafka :

Trois types de méthode :

- Le “Fire-and-forget” : envoi du message sans se soucier de son arrivée
- L’envoi synchrone : bloque jusqu’à obtention du réponse
- L’envoi asynchrone : ajoute une méthode de callback à l’envoi appelée lorsqu’une réponse est arrivée

Utilisation de Kafka : l’API Producer

Le type d’envoi “Fire-and-forget” :

```
// Commence par créer un ProducerRecord : c’est cet objet qui sera envoyé à Kafka.
// Il exésite plusieurs constructeur (voir la doc). Ici on utilise celui demandant dans l’ordre :
// Le nom du topic où l’on envoie des données, la clef et la valeur.
ProducerRecord<String, String> record = new ProducerRecord<>("CustomerCountry", "Precision Products", "France");


try {
   producer.send(record); // Envoi de la donnée sans vérification (“Fire-and-forget”)
} catch (Exception e) {
   e.printStackTrace();
}
```

Utilisation de Kafka : l’API Producer

Le type d’envoi synchrone :

```
// Commence par créer un ProducerRecord : c’est cet objet qui sera envoyé à Kafka.
// Il exésite plusieurs constructeur (voir la doc). Ici on utilise celui demandant dans l’ordre :
// Le nom du topic où l’on envoie des données, la clef et la valeur.
ProducerRecord<String, String> record = new ProducerRecord<>("CustomerCountry", "Precision Products", "France");


// Utilisation de la méthode “get()” bloquante permettant d’attendre une réponse de Kafka
// Cette méthode renvoie une exception si la donnée n’a pas été envoyée correctement
try {
   producer.send(record).get();
} catch (Exception e) {
   e.printStackTrace();
}
```

Utilisation de Kafka : l’API Producer

Le type d’envoi asynchrone :

```
// Définition d’une classe implémentant l’interface “org.apache.kafka.clients.producer.Callback”
private class DemoProducerCallback implements Callback {

   // L’objet recordMetada permet d’obtenir des informations sur la donnée sauvegardée dans Kafka
   @Override public void onCompletion(RecordMetadata recordMetadata, Exception e) {
      if (e != null) e.printStackTrace();
   }
}

ProducerRecord record = new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");
// Envoi de la donnée en précisant un callback.
producer.send(record, new DemoProducerCallback());
```

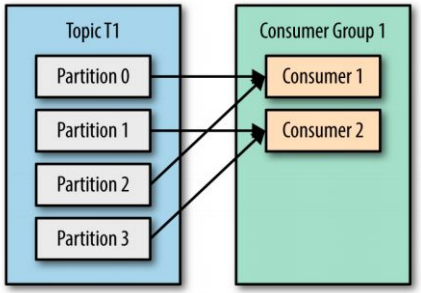
Utilisation de Kafka : l’API Producer

Les clefs :

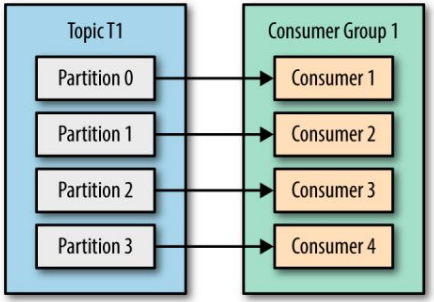
- Il est possible de ne pas ajouter de clefs aux données produites
- Les clefs peuvent servir à deux choses :
  - Informations supplémentaires
  - Permettent de décider dans quelle partition une donnée sera écrite. Tous les messages avec la même clef iront dans la même partition. Permet d’optimiser la lecture de donnée sur une même partition par exemple.

Utilisation de Kafka : l'API Consumer

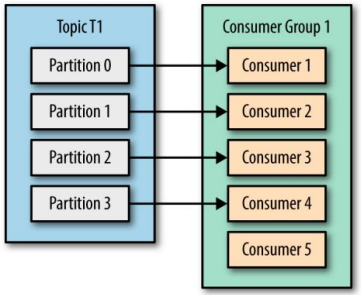
- Un consommateur fait parti d'un groupe de consommateurs. C'est ce qui permet de faire monter en charge un système sans modification à par l'ajout de consommateurs.
- Un ensemble de consommateurs faisant parti du même groupe et s'abonnant à un topic vont recevoir des messages provenant de partitions. Et les consommateurs au sein de ce groupe vont se répartir la lecture des messages entre toutes les partitions du topic.



Exemple avec 2 consommateurs et 4 partitions (deux chacun)



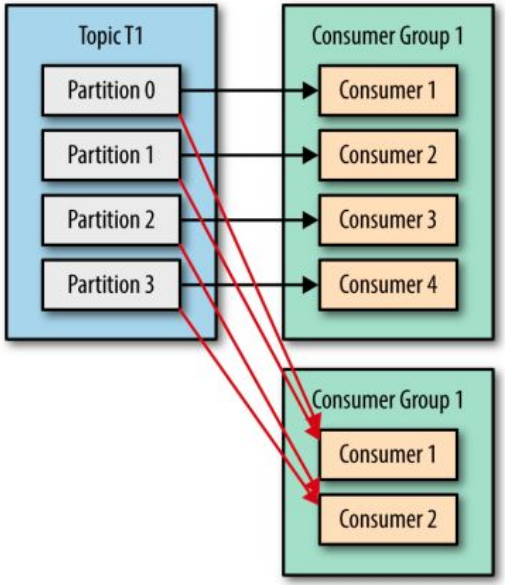
Exemple avec 4 consommateurs et 4 partitions (une chacun)



Si il y a plus de consommateurs que de partitions dans un topic, alors le surnombre ne fera aucun travail.

Utilisation de Kafka : l'API Consumer

- Plusieurs applications peuvent lire un même topic



Dans cet exemple, deux groupe de consommateurs lisent les messages d'un même topic.

Utilisation de Kafka : l’API Consumer

La création d’un consommateur :

```
Properties props = new Properties();

// Pour ce paramètre, voir la création d’un Producer
props.put("bootstrap.servers", "broker1:9092,broker2:9092");

// Ici, on précise l’id du groupe auquel appartient ce Consumer (sous forme d’une chaine de caractère)
props.put("group.id", "CountryCounter");

// A l’inverse de la création d’un Producer, ici, on précise une classe permettant de désérialiser la clef
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

// A l’inverse de la création d’un Producer, ici, on précise une classe permettant de désérialiser la valeur
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");


// Création du Consumer avec les paramètres définis précédemment
KafkaConsumer consumer = new KafkaConsumer(props);
```

Utilisation de Kafka : l’API Consumer

La souscription à un topic :

```
// Méthode permettant de souscrire à une liste de Topics.
consumer.subscribe(Collections.singletonList("customerCountries"));

// Possibilité d’utiliser des expressions régulières pour s’abonner à une ensemble de Topics correspondants.
consumer.subscribe("test.*");
```

Utilisation de Kafka : l'API Consumer

Lecture des messages :

```
try {
   while (true) {
      // Un Consumer doit "poll" en permanence. En effet, s'il ne le fait pas, il sera considéré comme "mort" et son
      // travail sera donné à un autre Consumer du groupe. Le paramètre donné à cette fonction est le nombre de ms
      // où le Consumer sera bloqué s'il n'y a pas de donnée arrivant du broker.
      ConsumerRecords records = consumer.poll(100);
      // La méthode "poll"  retourne une liste de données contenant chacune le topic et la partition d'où vient la
      // la donnée, l'offset de la donnée depuis la partition et la clef et la valeur de la donnée.
      for (ConsumerRecord record : records) {
         log.debug("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n", record.topic(),
record.partition(), record.offset(), record.key(), record.value()); int updatedCount = 1;
         if (custCountryMap.countainsValue(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
         }
         custCountryMap.put(record.value(), updatedCount) JSONObject json = new JSONObject(custCountryMap);
         System.out.println(json.toString(4))
      }
   }
}
finally { consumer.close(); }
```

Utilisation de Kafka : l’API Consumer

Les commits et les off-sets :

- Un commit dans Kafka et l’action de mettre à jour la position courante (l’off-set) dans la partition.
- Si un consommateur tombe en panne ou si un un nouveau consommateur entre en jeu, alors chaque consommateur peut être assigné à d’autres tâches (“rebalance”). Ainsi, pour savoir depuis où reprendre le travail, un consommateur va lire le dernier off-set de chaque partition et continuer à partir de là.
- Si le dernier off-set est plus petit que l’off-set du dernier message alors les messages compris entre ces deux off-set seront lus deux fois.
- A l’inverse, le dernier off-set est plus grand alors les messages entre ces deux off-set ne seront pas lus.
- => Gestion des commits important dans Kafka.

Utilisation de Kafka : l'API Consumer

Commits automatique :

- Façon la plus simple d'effectuer des commits.
- Activation en configurant le paramètre *enable.auto.commit=true*
- Gestion de la période à de commit automatique en configurant le paramètre *auto.commit.interval.ms*
  Par défaut, il est de 5 secondes.
- Problème : si une réattribution des tâches des consommateurs intervient 2 secondes après le dernier commit, alors les 2 secondes de tâches réalisées avant cela seront réalisées deux fois.

Utilisation de Kafka : l’API Consumer

Commits synchrone :

- La méthode commitSync() permet de “commit” le dernier off-set retourner par la méthode poll(). Il est important donc, de traiter tous les messages reçus de poll() AVANT de d’appeler la méthode commitSync()
- Exemple :

```
while (true) {
   ConsumerRecords records = consumer.poll(100);
   for (ConsumerRecord record : records) {
      System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n", record.topic(),
record.partition(),record.offset(), record.key(), record.value());
   }
   try {
      // Une fois le traitement des données finie (ici juste affichage) alors “commit” du dernier off-set
      consumer.commitSync();
   } catch (CommitFailedException e) {
      log.error("commit failed", e)
   }
}
```

Utilisation de Kafka : l’API Consumer

Commits asynchrone :

- Le commit synchrone limite le rendement puisqu’on attend une réponse.
- La méthode commitAsync() permet d’envoyer la requête de commit et de continuer.
- Exemple :

```
while (true) {
   ConsumerRecords records = consumer.poll(100);
   for (ConsumerRecord record : records) {
      System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n", record.topic(),
record.partition(),record.offset(), record.key(), record.value());
   }
   // Commit asynchrone
   consumer.commitAsync();
}
```

Utilisation de Kafka : l’API Consumer

Pattern connu : combinaison du commit synchrone et asynchrone

- Permet de s’assurer qu’un dernier commit soit fait avec une réattribution des tâches (rebalance)

```
try {
   while (true) {
      ConsumerRecords records = consumer.poll(100);
      for (ConsumerRecord record : records) {
         System.out.printf("topic = %s, partition = %s, offset = %d, customer = %s, country = %s\n", record.topic(),
record.partition(),record.offset(), record.key(), record.value());
      }
      // Commit asynchrone
      consumer.commitAsync();
   }
} catch(Exception e) {
   log.error("Unexpected error", e);
}
finally {
   try {
      consumer.commitSync();
   } finally { consumer.close(); }
}
```

Mécanismes internes de Kafka

Cluster :

- Utilisation de Zookeeper afin de maintenir une liste de brokers appartenant à un cluster
- Chaque broker a un identifiant unique
- Au démarrage, un broker s’enregistre en créant un noeud éphémère dans ZK

Mécanismes internes de Kafka

Le contrôleur :

- Le contrôleur est un broker Kafka avec plus de responsabilités
- Le premier broker qui démarre devient le contrôleur du cluster en créant un noeud éphémère ZK. Les autres broker ajoute un ZK Watcher sur ce noeud et lorsqu'il disparaît (donc plus de contrôleur) alors les autres brokers tentent de créer ce noeud ZK, le premier à réussir devient le contrôleur
- Le contrôleur est responsable d'élire les leaders des partitions et des répliques lorsqu'il détecte qu'un broker rejoint ou quitte le cluster.

Mécanismes internes de Kafka

La réplication :

- Garantie la disponibilité des données
- Chaque partition d'un Topic peut être répliquée plusieurs fois. Ces répliques sont sauvegardées dans les brokers
- Deux types de répliques :
  - La réplique "leader" : gère les requête de production / consommation des données
  - La réplique "suiveuse" : ne gère aucune requête. Se met à jour par rapport à la réplique leader. En cas de panne, possibilité de devenir la réplique leader.

Mécanismes internes de Kafka

La gestion des requêtes :

- Protocol binaire basé sur TCP
- Types de requêtes les plus communes :
  - Requêtes “produce” : envoyés par les producteurs, contient les messages à écrire sur les brokers
  - Requêtes “fetch : envoyés par les consommateurs et les répliques “suiveuses” pour la lecture des messages depuis les brokers
- Ces deux types de requêtes doivent être envoyées à la réplique “leader”.

Mécanismes internes de Kafka

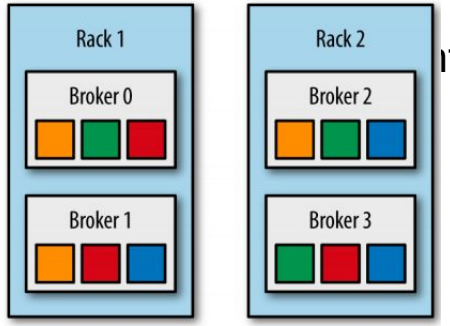
La gestion des requêtes :

- Protocol binaire basé sur TCP
- Types de requêtes les plus communes :
  - Requêtes “produce” : envoyés par les producteurs, contient les messages à écrire sur les brokers
  - Requêtes “fetch : envoyés par les consommateurs et les répliques “suiveuses” pour la lecture des messages depuis les brokers
- Ces deux types de requêtes doivent être envoyées à la réplique “leader”.
  - Retrouve les leaders en envoyant un autre type de requête “metadata”. Cette requête permet à un client de savoir, entre autres, pour un serveur donné quels sont les partition dont il est le leader.

Mécanismes internes de Kafka

Allocation des partitions :

- La création d'un topic implique de choisir le nombre de partitions pour ce topic et le facteur de réplication.
  - Exemple : On a 6 brokers, on créer un topic avec 10 partitions et un facteur de réplication de 3. Alors Kafka devra répartir 30 partitions entre 6 brokers
- Le but :
  - Répartir ces partitions de manière équitable entre tous les brokers.
  - S'assurer que les partitions leader soient au même endroit que les partitions suiveuses.



Exemple simple

- Télécharger la dernière version depuis : https://kafka.apache.org/downloads
- Lancer le serveur ZK intégré :

```
kafka_2.12-0.10.2.1>.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```

- Lancer un broker Kafka :

```
kafka_2.12-0.10.2.1>.\bin\windows\kafka-server-start.bat .\config\server.properties
```

- Créer un topic :

```
kafka_2.12-0.10.2.1>.\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic m2_info-topic
```

- Produire un message sur ce topic :

```
kafka_2.12-0.10.2.1>.\bin\windows\kafka-console-producer.bat --broker-list localhost:9092
--topic m2_info-topic Hello World
```

- Consommer un message du topic :

```
kafka_2.12-0.10.2.1>.\bin\windows\kafka-console-consumer.bat --bootstrap-server
localhost:9092 --topic m2_info-topic --from-beginning
```

Notes - Diapo n°5

Tandis que le format des messages est opaque pour Kafka puisque ce sont des tableaux d’octets, il est recommandé d’ajouter une structure ou un schéma au contenu du message de sorte qu’il puisse être facilement compris. (Voir Apache Avro qui est un framework de sérialisation par exemple).

Découpler écriture / lecture : Permet d’éviter qu’une application ayant souscrit à des messages ne soit obligée d’être mise à jour afin de gérer un nouveau format de donnée en plus de l’ancien. En utilisant un schéma et en le sauvegardant dans un endroit commun, les messages peuvent être compris de tous sans coordination.

Notes - Diapo n°6

Les messages sont ajoutés à un topic et sont lus dans l’ordre du début à la fin. A noter, puisque les topics sont décomposer en partitions, il n’y a aucune garantie de l’ordre temporel des messages au sein d’un topic.

Redondance et montée en charge : Chaque partition peut être sauvegardée sur des serveurs différents ce qui veut dire qu’un seul topic peut être étendu à travers différents serveurs. Permettant de fournir des performances bien supérieur la capacité d’un seul serveur.

Notes - Diapo n°7

Par défaut les producteurs ne s’occupent pas de savoir dans quelles partitions les messages seront écrits.

L’offset des messages : une métadonnée (un entier) qui augmente constamment ajouté par Kafka à chaque message produit. Chaque message dans une partition donnée possède un offset unique. En sauvegardant l’offset du dernier message consommé pour chaque partition, un consommateur peut s’arrêter et redémarrer sans perdre sa place.

Groupe de consommateurs : Assure que chaque partition n’est consommée que par un seul membre du groupe. De cette façon le nombre de consommateur peut facilement évoluer. De plus, si un consommateur tombe en panne alors le reste du groupe va distribuer sa charge de travail.

Notes - Diapo n°8

Dans cet exemple, il y trois consommateurs dans un groupe consommant un topic. Deux Consumers travaillent sur une partition et un autre consommateur travaille sur deux partitions.

Notes - Diapo n°13

Producteurs Multiple : exemple : Un site servant du contenu aux utilisateurs via un certain nombre de services peut avoir un seul topic où tous ces services écriront en utilisant le même format. Les applications consommants ces données pourront recevoir un seul flux de données pour toutes les applications du site sans avoir besoin de coordonner la consommation des données depuis plusieurs topics.

Notes - Diapo n°18

A noter qu’il est possible de produire des données qui n’ont pas de clefs, mais même dans ce cas, il est nécessaire de préciser une classe qui permet de sérialiser la clef.