



Kafka : Messagerie distribuée avec Apache Kafka

David THIBAU – 2023

david.thibau@gmail.com



Agenda

Introduction à Kafka

- Le projet Kafka
- Cas d'usage
- Concepts

Installation

- Kraft
- Broker Kafka
- Utilitaires Kafka
- Cluster Kafka

Kafka APIs

- Producer API
- Consumer API
- Sérialisation Avro
- Frameworks Java
- Connect API
- Autres APIS

Garanties Kafka

- Mécanismes de réplication
- At Most Once, At Least Once
- Exactly Once
- Débit, latence, durabilité

Administration

- Gestion des topics
- Stockage et rétention des partitions
- Gestion du cluster
- Sécurité
- Dimensionnement
- Surveillance



Introduction à Kafka

Le projet Kafka

Cas d'usage

Concepts



Origine

Initié par *LinkedIn*, mis en OpenSource en 2011

Écrit en *Scala* et *Java*

Au départ, un message broker gérant une file de messages

A évolué pour devenir une plate-forme de streaming d'événements temps-réel

Organisé en cluster, taillé pour le BigData, il est basé sur l'abstraction « d'un journal de commit distribué »

Maintenu par *Confluent* depuis 2014



Objectifs

- Découpler producteurs et consommateurs de messages
- Persister les messages afin qu'ils puissent être consommés par de nombreux consommateurs, (éventuellement à posteriori)
- Atteindre de très haut débit et une latence faible
- Scaling horizontal flexible
- Offrir des garanties de fiabilité de la livraison de messages, malgré des défaillances !



Fonctionnalités

Kafka a trois capacités clés:

- Publier et s'abonner à des flux de messages¹ avec certaines garanties de fiabilité.
- Stocker les flux de messages de manière durable et tolérante aux pannes.
- Traiter, transformer les flux de messages au fur et à mesure qu'ils se produisent.

1. Dans la suite des slides on utilise de façon non-différenciés les termes *message*, *événement*, *enregistrement*



Points forts

Très bonne scalabilité et flexibilité

- Gestion des abonnements multiples
- Facilité d'extension du cluster

Très bonne performance

Disponibilité et tolérance aux fautes

Rétention sur disque

Traitement distribué d'évènements

Intégration avec les autres systèmes



Confluent

Créé en 2014 par *Jay Kreps*, *Neha Narkhede*, et *Jun Rao*

Mainteneur principal d'Apache Kafka

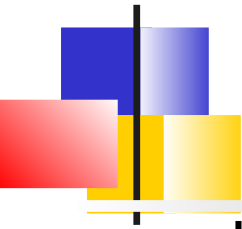
Plate-forme *Confluent* :

- Une distribution de Kafka
- Fonctionnalités commerciales additionnelles



Introduction à Kafka

Le projet Kafka
Cas d'usage
Concepts



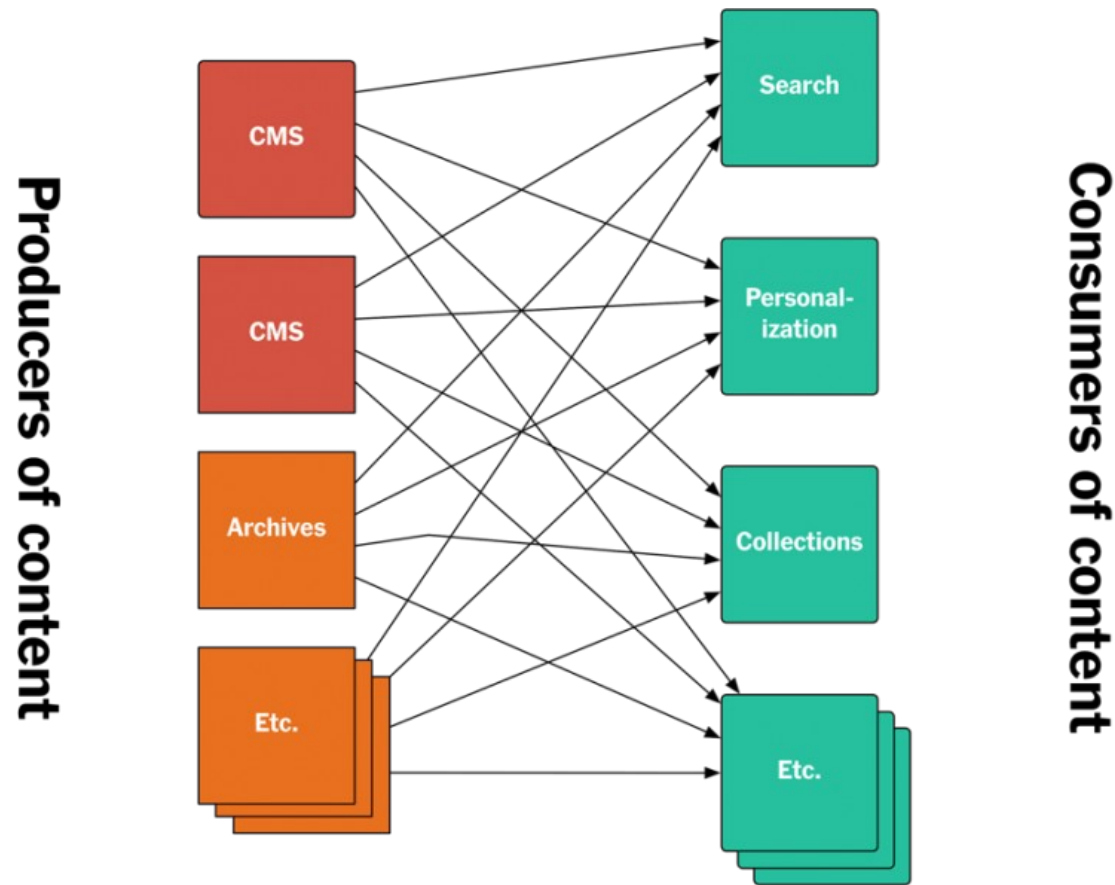
Kafka vs Message broker traditionnel

Kafka peut être utilisé comme **message broker** permettant de découpler un service producteur de services consommateurs

- Kafka n'offre que le modèle **PubSub**.
- Grâce au concept de **groupe de consommateur**, ce modèle est scalable
- Kafka offre une **garantie plus forte** sur l'ordre de livraison des messages
- Kafka ne supprime pas les messages après consommation. Ils peuvent être **consommés à posteriori**

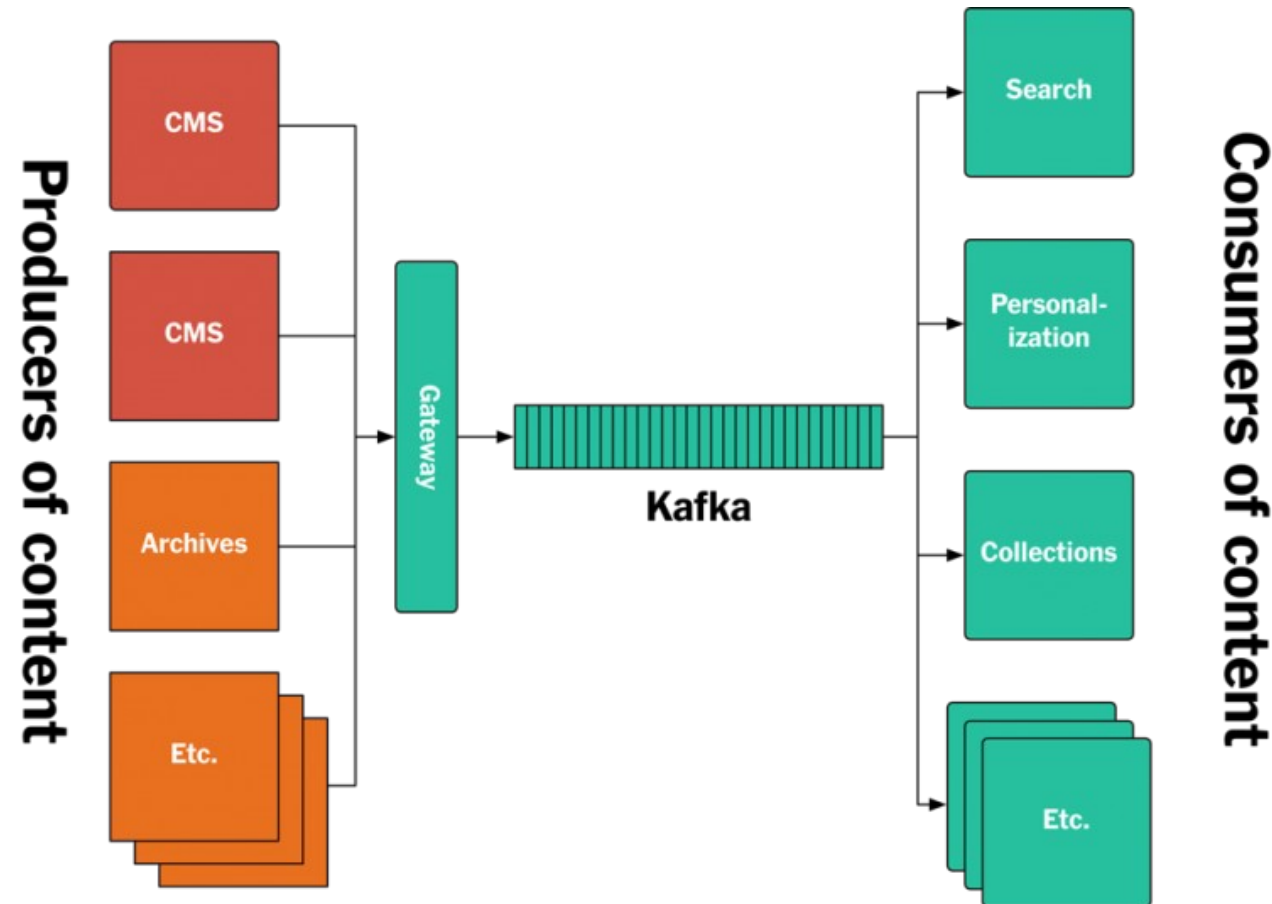
=> Idéal comme plate-forme d'intégration entre services : Architecture micro-services, ESB

Exemple ESB (New York Times) *Avant*



Exemple ESB (New York Times)

Après





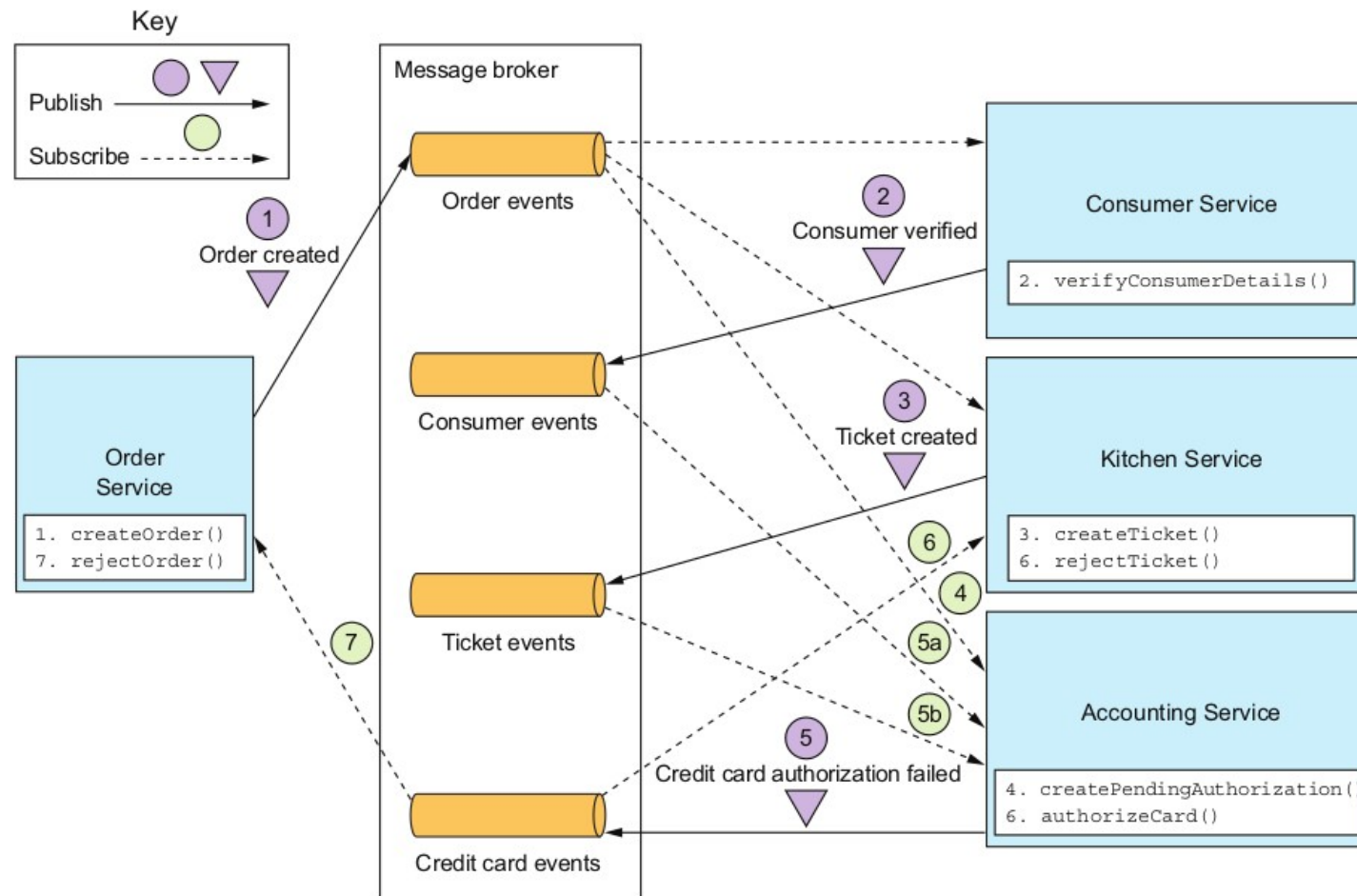
Exemple : micro-services

Kafka est souvent utilisé pour permettre des communications asynchrones entre les services d'une architecture micro-services

- Permet tous les styles d'interaction :
Requête/Réponse synchrone asynchrone, One way notification, Publish and Subscribe, Publish et réponse asynchrones
- Permet l'implémentation de patterns micro-services, par exemple SAGA¹ (~ Transaction distribuée)

1. <https://microservices.io/patterns/data/saga.html>

Exemple Micro-services Message Broker et Design pattern SAGA



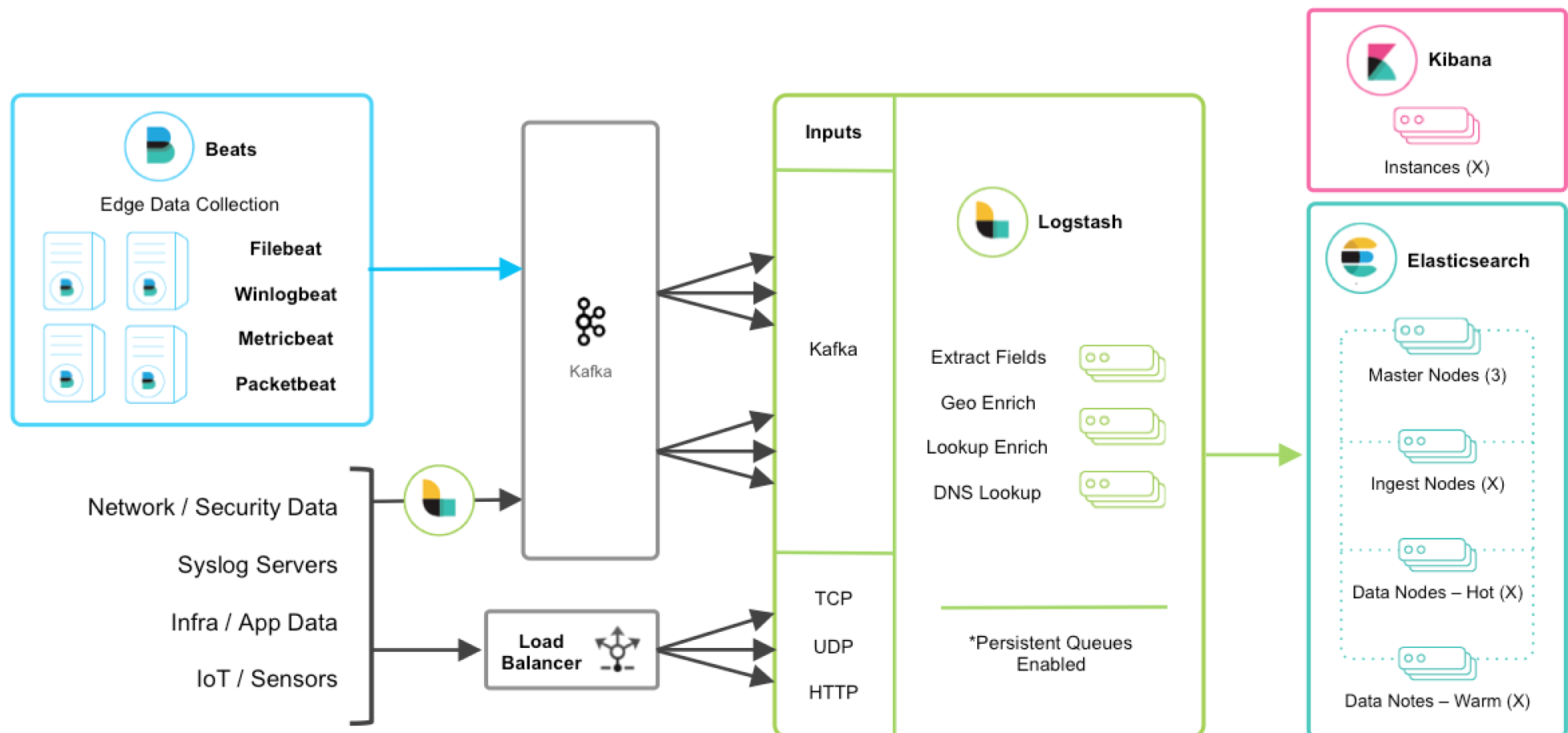


Kafka pour l'ingestion massive de données

Les *topics* Kafka peuvent être utilisés pour bufferiser les événements provenant de multiples sources

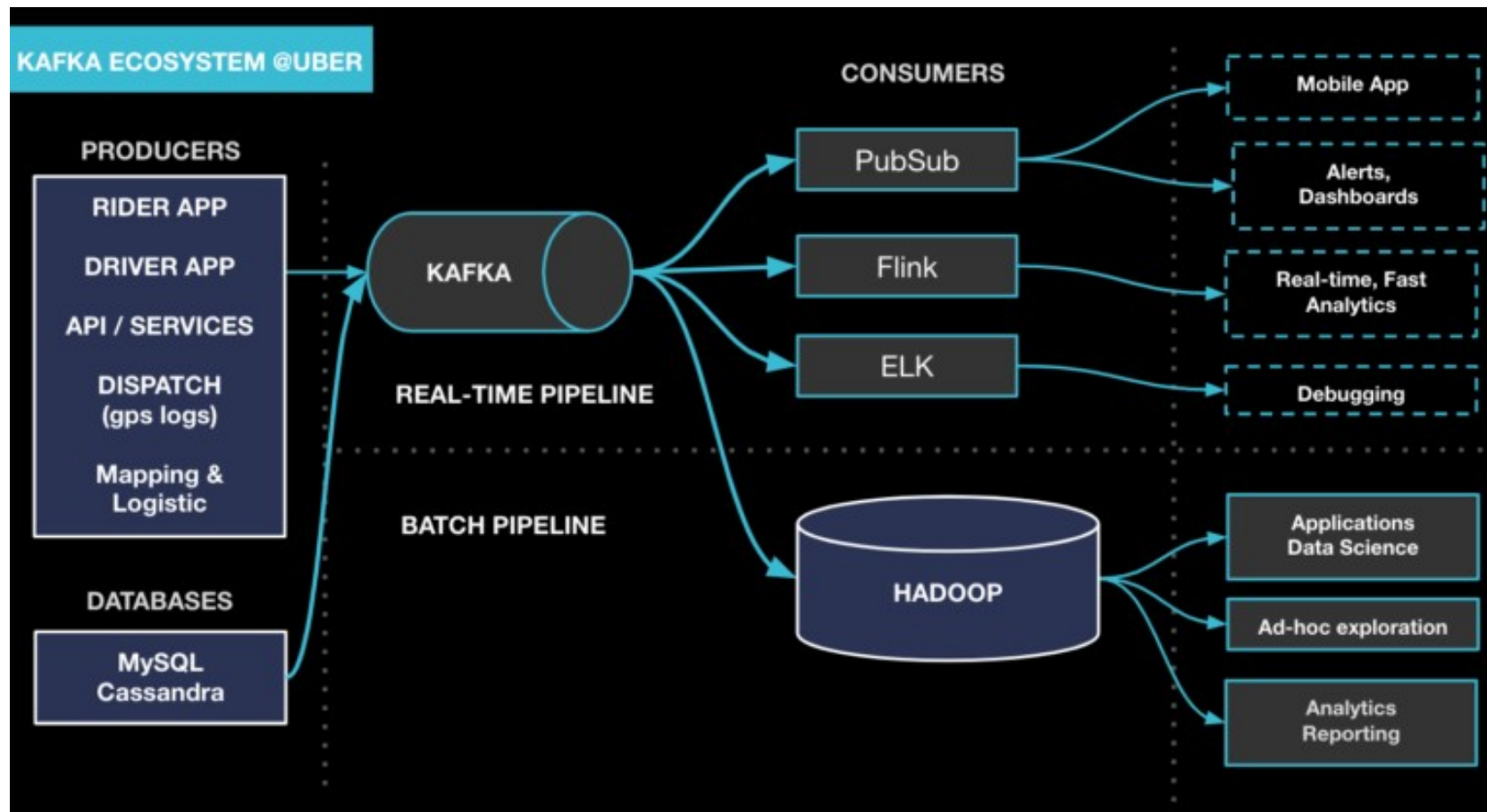
Ces événements peuvent alors être traités par des pipelines généralement destinées au stockage dans des solutions d'analyse temps-réel, d'alerting ou d'archivage souvent relié au machine learning

Exemple Architecture ELK Bufferisation des événements



Ingestion massive de données

Exemple Uber





Architecture Event-driven

Les architectures *event-driven* sont une alternative intéressante dans un contexte micro-services.

- Cela produit généralement des architectures plus souples et plus réactives.

Chaque micro-service consomme en continu des événements :

- Lit un ou plusieurs topics Kafka en entrée
- Effectue un traitement
- Écrit vers un ou plusieurs topics de sortie

Kafka Stream, Spring Cloud Stream ou Spring Cloud Data Flow facilitent ce type d'architecture



Data Stream

Exemple Spring Cloud Data Flow

Streams

Create a stream using text based input or the visual editor.

Definitions Create Stream

CREATE STREAM

CLEAR

LAYOUT

AUTO LINK

GRID

```
1 ingest-to-cassandra=http | cassandra
2 filter-write-to-hdfs=:ingest-to-cassandra.http > filter | hdfs
3 transform-write-to-mongo=:ingest-to-cassandra.http > transform | mongodb
4 logger=:transform-write-to-mongo.transform > log
```

source

file

ftp

gemfire

gemfire-cq

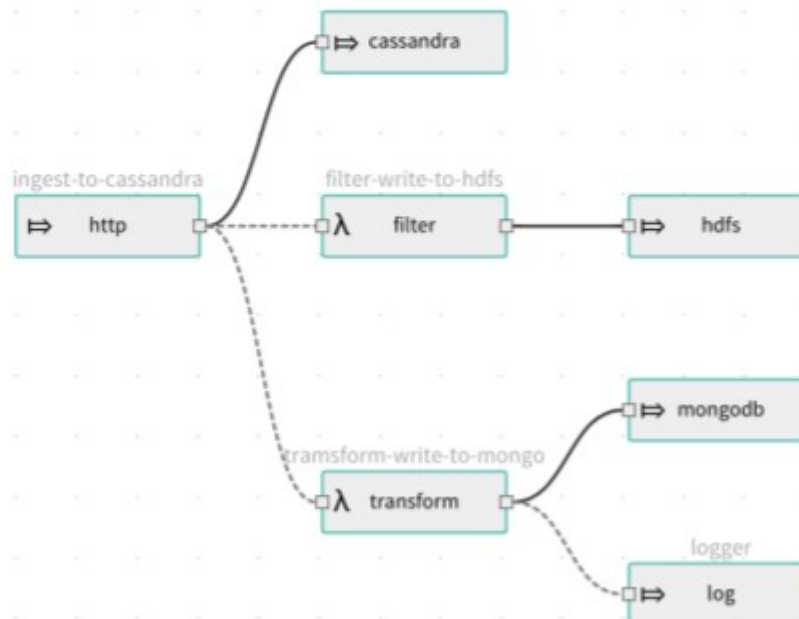
http

jdbc

jms

load-genera...

loggregator





Kafka comme système de stockage

Les enregistrements sont écrits et répliqués sur le disque.

La structure de stockage est très scalable.

Kafka fonctionne de la même manière avec 50 Ko ou 50 To de données.

=> Kafka peut être considéré comme un système de stockage.

A la différence d'une BD, il stocke l'intégralité de l'historique des données plutôt qu'un simple instantané

Kafka peut alors être utilisé comme *EventStore* et permet la mise en place du pattern *Event Sourcing*¹ utilisé dans les micro-services

Des abstractions sont proposées pour faciliter la manipulation de l'*EventStore* : Projet **ksqlDB**²

1. <https://microservices.io/patterns/data/event-sourcing.html>

2. <https://ksqldb.io/overview.html>



Event sourcing Pattern

Event sourcing Pattern¹ : Persiste un agrégat comme une séquence d'événements du domaine représentant les changements d'état

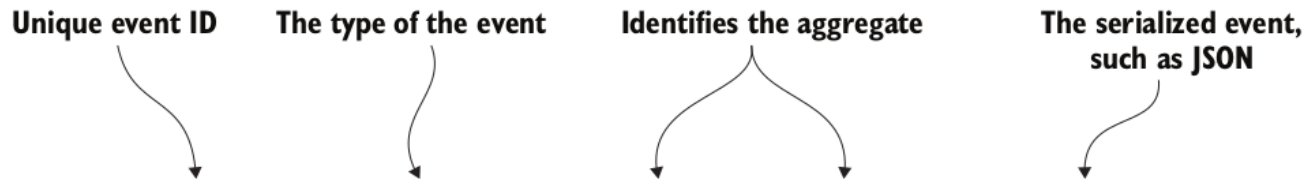
=> Une application recrée l'état courant d'un agrégat en rejouant les événements

1. <http://microservices.io/patterns/data/event-sourcing.html>



Event Store

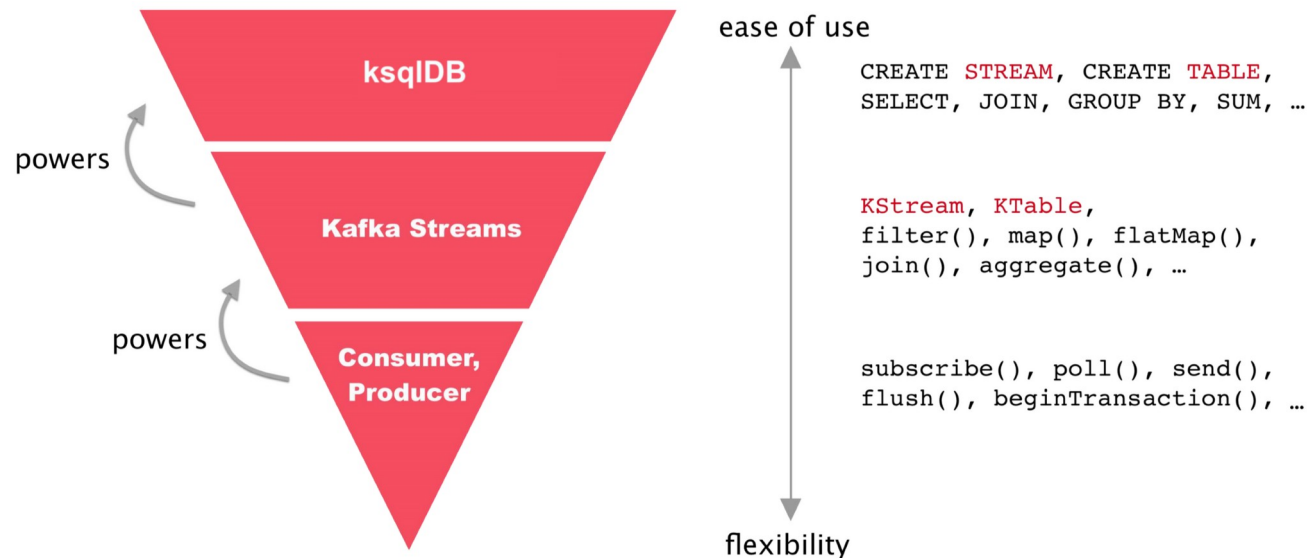
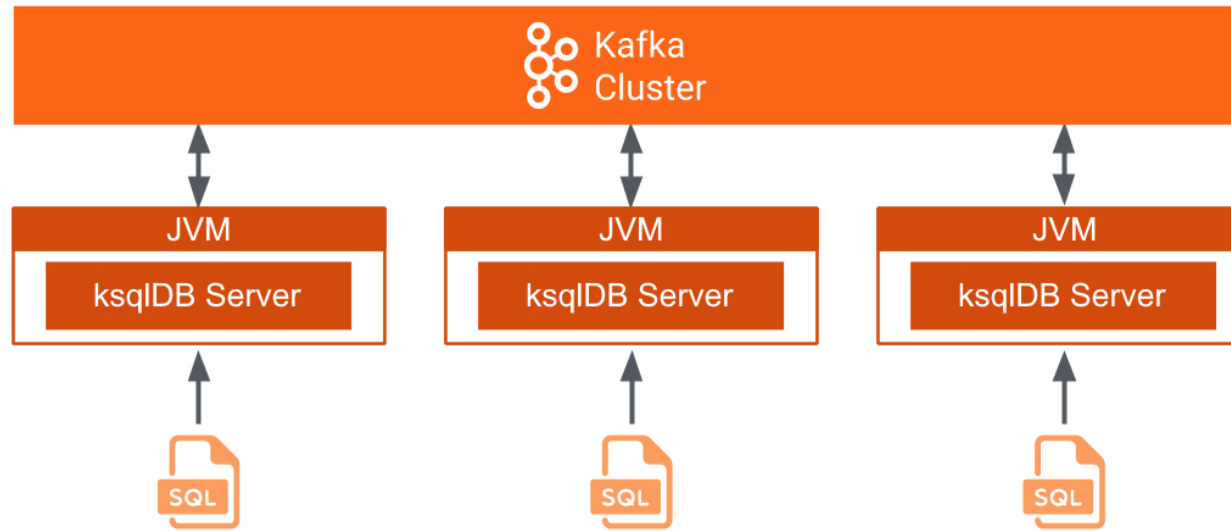
Au lieu de stocker l'agrégat dans un schéma traditionnel classique, l'agrégat est stocké sous forme d'événements dans un ***EventStore***



event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...

EVENTS table

ksqlDB Standalone Application (Headless Mode)





Introduction à Kafka

Le projet *Kafka*
Cas d'usage
Concepts



Concepts de base

Kafka s'exécute en **cluster** sur un ou plusieurs serveurs (**brokers**) pouvant être déployés dans différents data-center.

Le cluster Kafka
stocke des flux d'enregistrements : les **records**
dans des rubriques : les **topics** .

Chaque enregistrement se compose d'une clé éventuelle, d'une valeur et d'un horodatage.

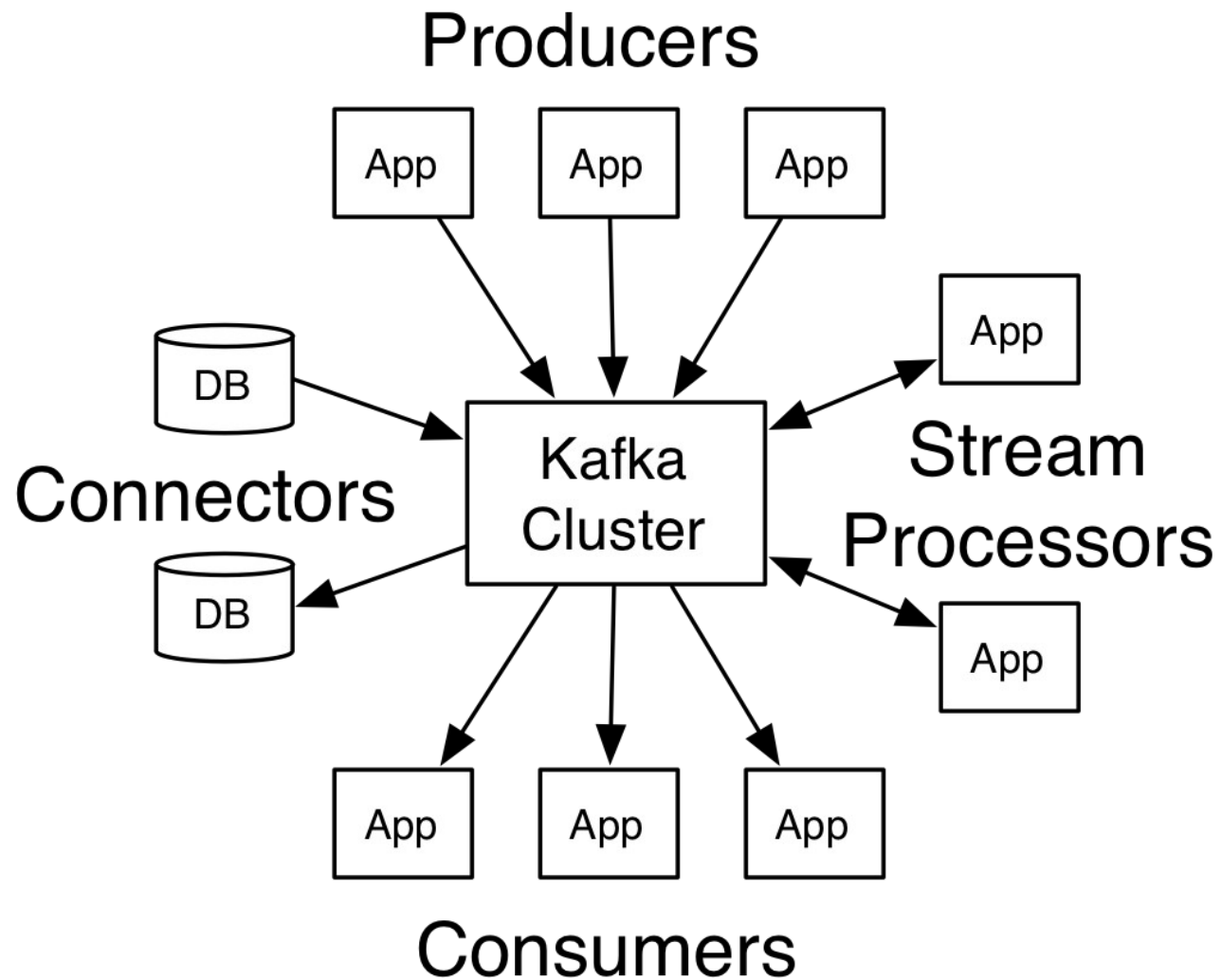


APIs

Kafka propose 5 principales APIs :

- L'API **Producer** permet à une application de publier un flux sur un ou plusieurs topics Kafka.
- L'API **Consumer** permet à une application de s'abonner à un ou plusieurs topics et de traiter le flux d'enregistrements associé.
- L'API **Streams** permet à une application d'agir comme un processeur de flux, consommant un ou plusieurs topic d'entrée et produisant un flux de sortie vers un ou plusieurs topics.
- L'API **Connector** permet de créer et d'exécuter des producteurs ou des consommateurs à partir de système tierces (BD, fichiers, STDOUT, ...)
- L'API **Admin** permet de gérer les topics et le cluster

APIs





Protocole Client/Serveur

Dans Kafka, la communication entre les clients et les serveurs s'effectue via un protocole TCP simple, performant et indépendant du langage.

- Ce protocole est versionné et maintient une compatibilité ascendante avec les versions plus anciennes.

Apache fournit un client Java, mais les clients sont disponibles dans de nombreuses langages.¹

1. <https://cwiki.apache.org/confluence/display/KAFKA/Clients>

Topic

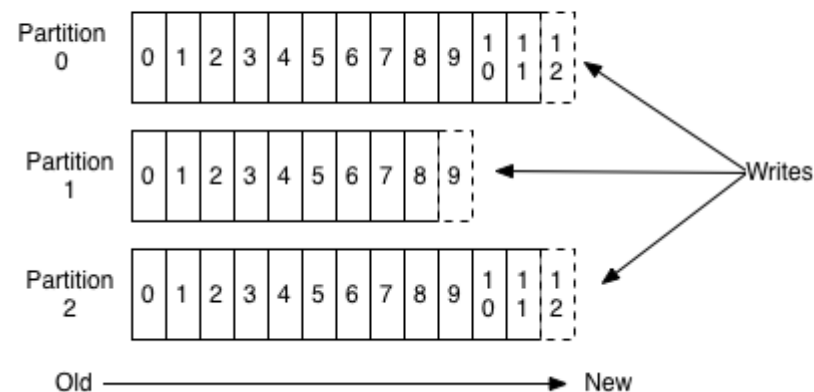
Les *records* sont publiés vers des **topics**.

Les *topics* de Kafka peuvent avoir Zéro, Un ou de multiples abonnés

Les topics peuvent être **partitionnés**.

Le cluster Kafka conserve donc un journal partitionné

Anatomy of a Topic



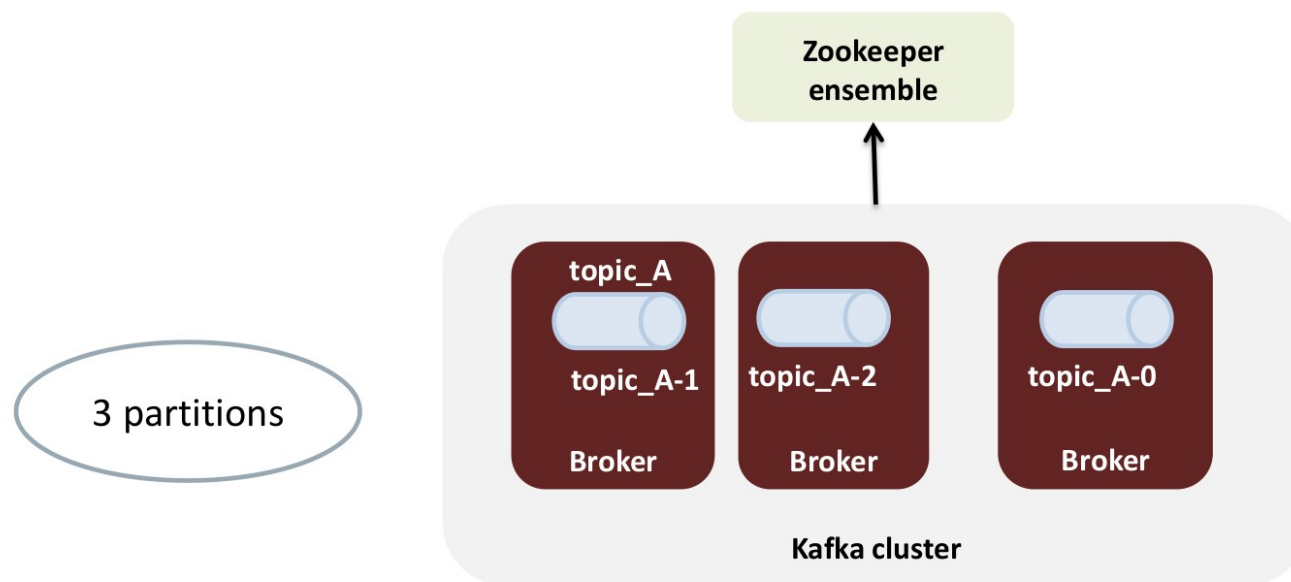


Apport des partitions

Les partitions autorisent le parallélisme et augmentent la capacité de stockage en utilisant les capacités disque de chaque nœud.

L'ordre des messages n'est garanti qu'à l'intérieur d'une partition

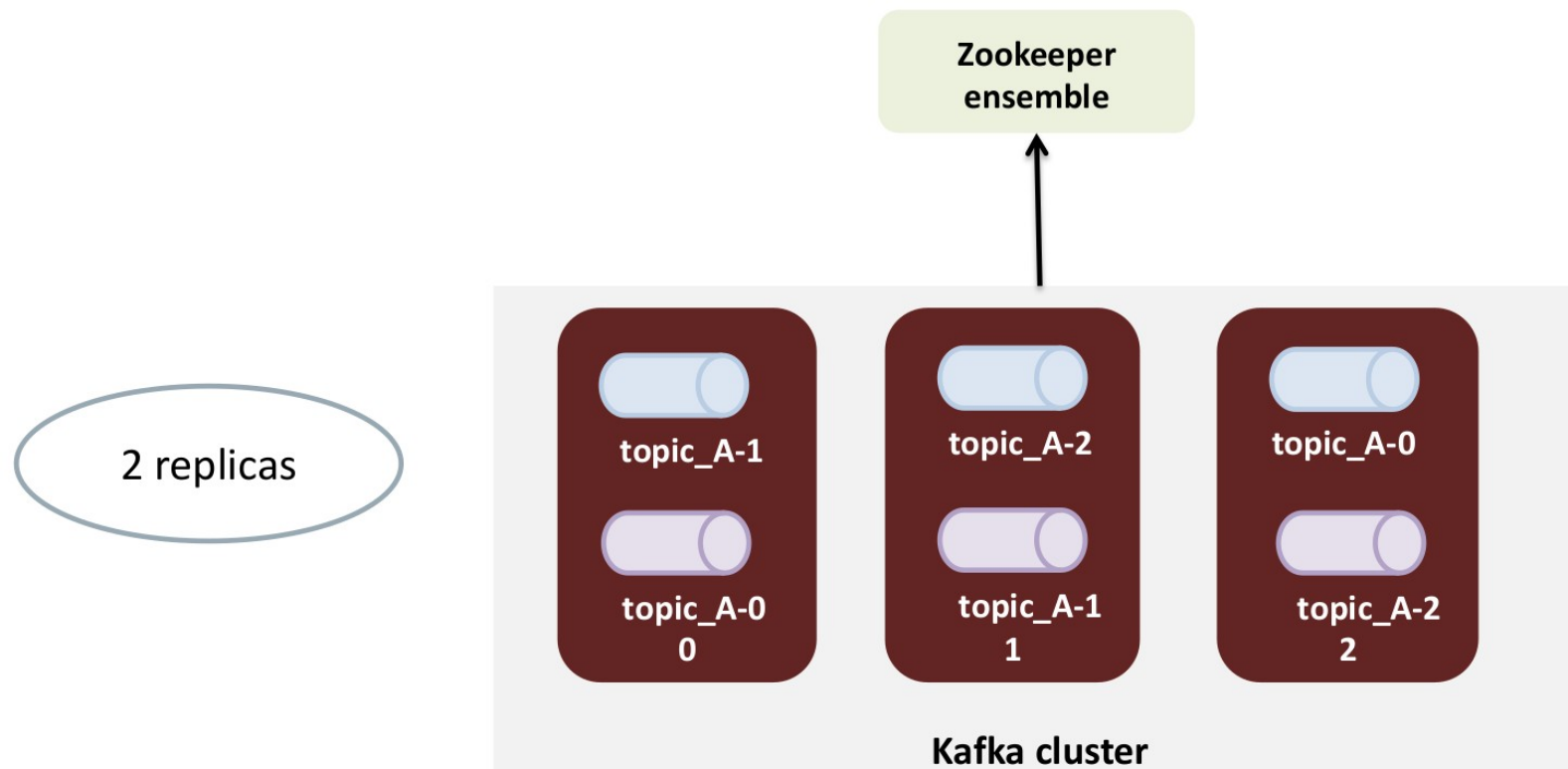
Le nombre de partition est fixé à la création du *topic*



Réplication

Les partitions peuvent être **répliquées**

- La réplication permet la tolérance aux pannes et la durabilité des données





Distribution des partitions

Les partitions sont réparties sur les instances du cluster.

Les répliques sont distribuées sur des instances différentes

Pour chaque partition répliquée, une des instances agit comme **maître (leader)**. Les autres comme **suiveurs (follower)**

- Le maître coordonne les lectures et les écritures sur la partition
- Les suiveurs répliquent passivement le maître
- Si le maître défaille, un processus d'élection choisit un autre maître parmi les répliques



Partition et offset

Chaque partition est une séquence **ordonnée et immuable** d'enregistrements.

Un numéro d'identification séquentiel nommé **offset** est attribué à chaque enregistrement.

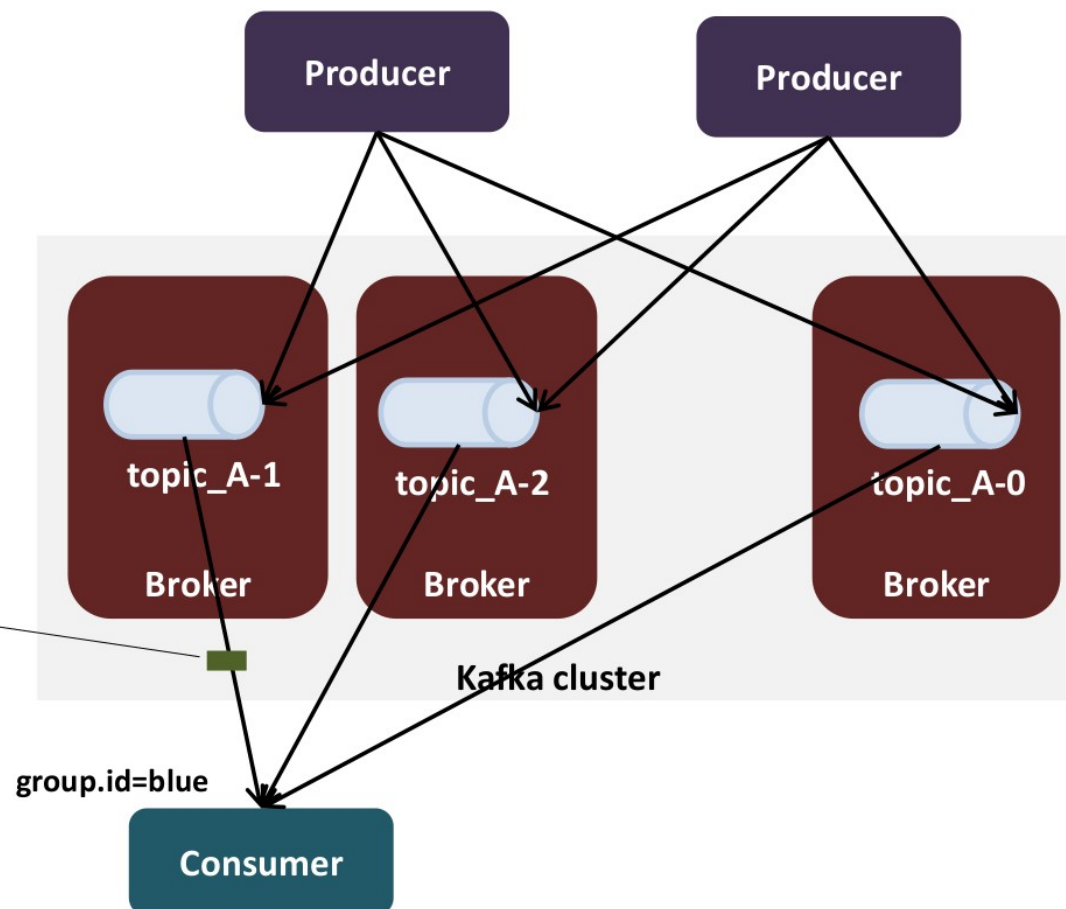
Le cluster Kafka conserve durablement tous les enregistrements publiés, qu'ils aient ou non été consommés, en utilisant une **période de rétention** configurable.

Clients du cluster

Les producteurs et consommateurs sont connectés à tous les brokers détenant le *topic*

message (record, event)

- key-value pair
- string, binary, json, avro
- serialized by the producer
- stored in broker as byte arrays
- deserialized by the consumer



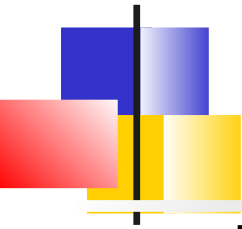


Routing des messages

Les producteurs sont responsables du choix de la partition en fonction de l'enregistrement

Cela peut être fait

- via une stratégie Round-Robin assurant un équilibrage de charge
- En fonction des données de l'enregistrement. Typiquement, la clé



Groupe de consommateurs

Les consommateurs sont taggés avec un nom de **groupe**

- Chaque enregistrement d'un topic est remis à une instance de consommateur au sein de chaque groupe.
- Les instances de consommateurs peuvent se trouver dans des threads, processus ou machines distincts.
=> Scalabilité et Tolérance aux fautes



Offset consommateur

La seule métadonnée conservée pour un groupe de consommateurs est son **offset** du journal.

Cet offset est contrôlé par le consommateur:

- normalement, le consommateur avance son offset au fur et à mesure de sa lecture des enregistrements,
- mais, il peut consommer dans l'ordre qu'il souhaite.

Par exemple, retraiter les données les plus anciennes ou repartir d'un offset particulier.



Consommateur vs Partition

Rééquilibrage dynamique

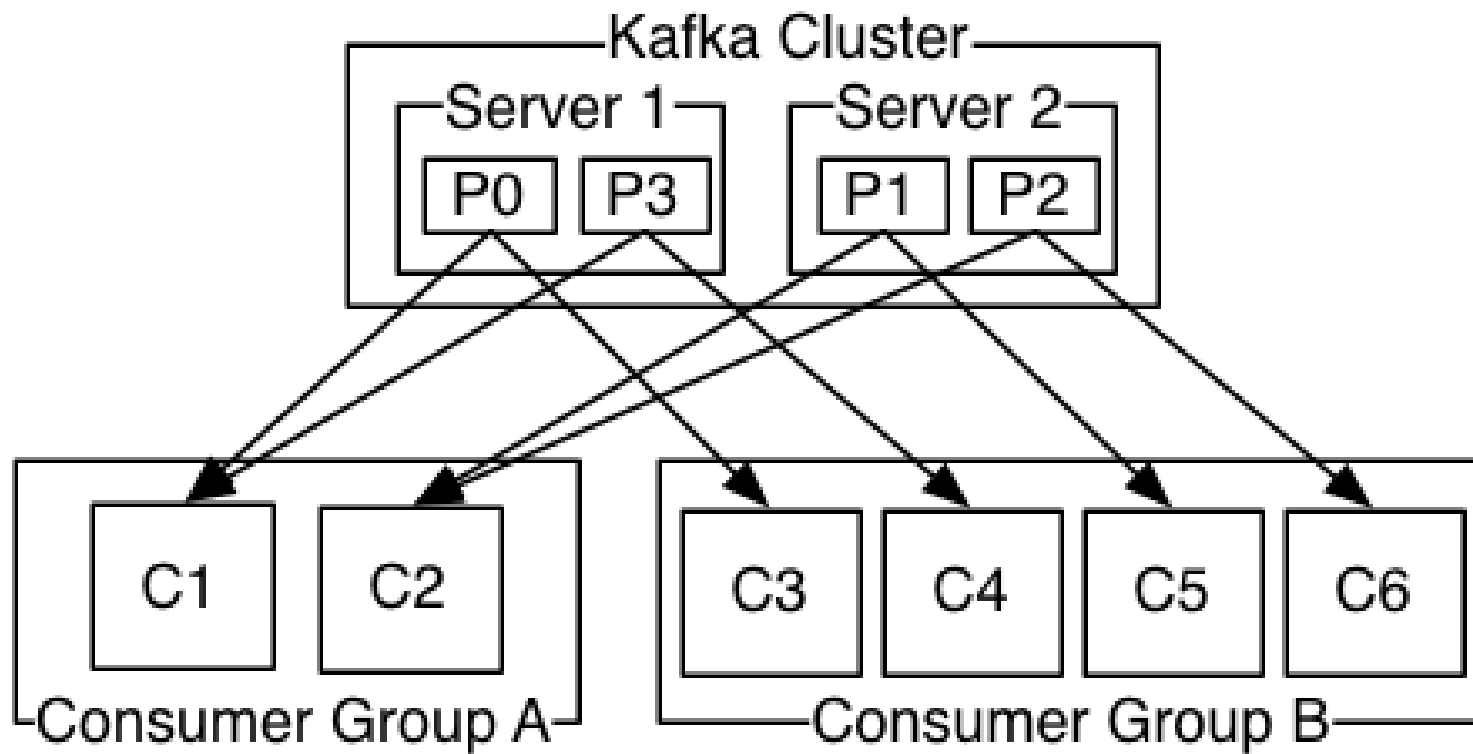
Kafka assigne les partitions à des instances de consommateur d'un même groupe.

- A tout moment, un consommateur est exclusivement dédié à une partition

Ceci est géré dynamiquement par le protocole Kafka.

- Si de nouvelles instances rejoignent le groupe, elles reprendront certaines partitions des autres membres du groupe;
- si une instance meurt, ses partitions seront distribuées aux instances restantes.

Example





Ordre des enregistrements

Kafka garantit un ordre total sur les enregistrements d'une partition, mais pas sur les différentes partitions d'un topic.

- L'ordre sur les partitions, combiné à la possibilité de partitionner les données par une clé est suffisant pour la plupart des applications.
- Si une application nécessite un ordre strict sur tous les enregistrements. Il faut que le topic n'est qu'une seule partition



Installation

KRaft

Broker *Kafka*
Utilitaires *Kafka*
Cluster *Kafka*



Pré-requis

OS recommandé Linux

JRE8+ ou même mieux JDK8+

Un cluster Kafka comprend :

- Plusieurs serveurs de Kafka
Chaque serveur a un **id**, un **hôte** et un **port** d'écoute
- Plus soit :
 - Un ensemble Zookeeper¹
Stockage des méta-données du cluster
 - Certains serveurs sont configurés comme contrôleur (KRaft)

1. En cours de dépréciation



Kraft mode

Apache Kafka Raft (KRaft) basé sur le protocole de consensus *Raft* simplifie grandement l'architecture de Kafka en se séparant du processus séparé ZooKeeper.

En mode KRaft, chaque serveur Kafka est configuré en tant que contrôleur, broker ou les deux à la fois via la propriété ***process.roles***¹

Si *process.roles* n'est pas définie, il est supposé être en mode ZooKeeper.

1. Les valeurs possibles sont donc : ***broker*** ou ***controller*** ou ***broker,controller***



Contrôleurs

Certains processus Kafka sont donc des contrôleurs, ils participent aux quorums sur les méta-données.

- Une majorité des contrôleurs doivent être vivants pour maintenir la disponibilité du cluster
- Il sont donc typiquement en nombre impair. (3 pour tolérer 1 défaillance, 5 pour 2)

Tous les serveurs découvrent les votants via la propriété ***controller.quorum.voters*** qui les liste en utilisant l'*id*, le *host* et le *port*

controller.quorum.voters=id1@host1:port1,id2@host2:port2,id3@host3:port3



Cluster ID

Chaque cluster a un **ID** qui doit être présent dans les configurations de chaque serveur.

Avec Kraft mode, il faut définir soit même cet ID.

L'outil *kafka-storage.sh random-uuid* permet de générer un ID aléatoire



Installation

Pré-requis et ensemble *Zookeeper*

Broker *Kafka*

Utilitaires *Kafka*

Cluster *Kafka*



Installation broker

Téléchargement, puis

```
# tar -zxf kafka_2.11-0.9.0.1.tgz
# mv kafka_2.11-0.9.0.1 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# cd /usr/local/kafka/bin
# ./kafka-server-start.sh -daemon ../config/server.properties
#
```

Les propriétés de configuration définies dans ***server.properties*** peuvent être surchargées en ligne de commande par l'option ***--override***



Configuration broker

broker.id : Chaque broker doit avoir un identifiant entier unique à l'intérieur du cluster

port : Par défaut 9092

controller.quorum.voters : Listes des contrôleurs

Optionnellement :

process.roles : Le rôle du serveur Par défaut *broker,controller*

log.dirs : Liste de chemins locaux où kafka stocke les messages

auto.create.topics.enable : Création automatique de topic à l'émission ou à la consommation



Vérifications

Création de topic :

```
bin/kafka-topics.sh --create --bootstrap-server  
localhost:9092 --replication-factor 1 --partitions 1 --  
topic test
```

Envois de messages :

```
bin/kafka-console-producer.sh --bootstrap-server  
localhost:9092 --topic test  
This is a message  
This is another message  
^D
```

Consommation de messages :

```
bin/kafka-console-consumer.sh --bootstrap-server  
localhost:9092 --topic test --from-beginning  
This is a message  
This is another message
```



Installation

Pré-requis et ensemble *Zookeeper*

Broker *Kafka*

Utilitaires *Kafka*

Cluster *Kafka*



Introduction

Le répertoire ***\$KAFKA_HOME/bin*** contient de nombreux scripts utilitaires dédiés à l'exploitation du cluster.

Les scripts utilisent l'API Java de Kafka

Une aide est disponible pour chaque script décrivant les paramètres requis ou optionnels



Gestion des topics

Le script ***bin/kafka-topics.sh*** permet de créer, supprimer, modifier visualiser un topic

Exemple création de topic :

```
kafka/bin/kafka-topics.sh --create \  
  --bootstrap-server localhost:9092 \  
  --replication-factor 1 --partitions 13 \  
  --topic my-topic
```

Exemple listing des topics :

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



Configuration des valeurs par défaut

Les valeurs par défaut concernant les topics sont définis dans *server.properties* :

- ***num.partitions*** : Nombre de partitions. *Par défaut 1*
- ***default.replication.factor*** : Nombre de répliques par défaut. *Par défaut 1*
- ***log.retention.ms/minutes/hours*** : Le temps de rétention d'un message. *Par défaut 1 semaine*
- ***log.retention.bytes*** : Critère additionnel pour spécifier une taille par partition pour la rétention
- ***log.segment.bytes*** : Taille d'un segment, i.e. fichier ou sont stockés les messages. *Par défaut 1Go*
- ***log.segment.ms*** : Le délai de fermeture d'un segment. Exclusif avec *log.segment.bytes*
- ***message.max.bytes*** : Taille maximale d'un message. *Par défaut 1Mo*



Envoi de message

Le script ***bin/kafka-console-producer.sh*** permet d'envoyer des messages sur un topic

Exemple :

```
bin/kafka-console-producer.sh \  
  --bootstrap-server localhost:9092 \  
  --topic my-topic
```

...Puis saisir les messages sur l'entrée standard



Lecture de message

Le script ***bin/kafka-console-consumer.sh*** permet de consommer les messages d'un topic

Exemple lecture depuis l'origine :

```
bin/kafka-console-consumer.sh \  
  --bootstrap-server localhost:9092 \  
  --topic my-topic \  
  --from-beginning
```



Autre utilitaires

kafka-configs.sh permet des mises à jour de configuration dynamique des brokers

kafka-consumer-groups.sh permet de gérer les groupes de consommateurs : les lister et manipuler leurs offsets

kafka-reassign-partitions.sh permet de gérer les partitions, déplacement sur de nouveau brokers, de nouveaux répertoire, extension de cluster, ...

kafka-dump-log.sh permet d'afficher les logs

Kafka-metadata-quorum.sh permet d'afficher les informations sur les contrôleurs

***kafka-verifiable-consumer.sh*, *kafka-verifiable-producer.sh*:**
Utilitaires permettant de produire ou consommer des messages et de les afficher sur la console au format JSON



Outils graphiques

Il peut être intéressant de s'équiper d'outils graphiques aidant à l'exploitation du cluster.

Comme produit gratuit, citons :

- **Akhq** (<https://akhq.io/>)
- **Kafka Magic** (<https://www.kafkamagic.com/>)

Ils proposent une interface graphique permettant :

- De rechercher et afficher des messages,
- Transformer et déplacer des messages entre des topics
- Gérer les topics
- Automatiser des tâches.

La distribution commerciale de Confluent a naturellement un outil graphique d'exploitation



Installation

Pré-requis et ensemble *Zookeeper*

Broker *Kafka*

Utilitaires *Kafka*

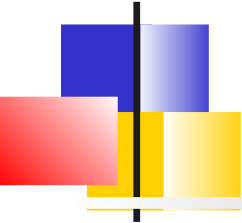
Cluster *Kafka*



Configuration cluster

3 contraintes pour la configuration d'un cluster :

- Tous les brokers doivent avoir le même paramètre ***cluster.id***
- Chaque broker doit avoir une valeur unique pour ***node.id***
- Au moins 1 broker doit avoir le rôle contrôleur



Propriétés d'un broker appartenant à un cluster

node.id : Identifiant unique du broker

process.roles : controller,broker

listeners : Nommage des port TCP ouverts

inter.broker.listener.name : Port utilisé pour la communication inter-broker

controller.listener.names : Port utilisé pour la communication contrôleur

controller.quorum.voters : Liste des contrôleurs sous la forme
<id>@IP:PORT

De plus, l'identifiant du cluster est présent dans le fichier *meta.properties* du répertoire des logs



Nombre de brokers

Pour déterminer le nombre de brokers :

- Premier facteur :
Le niveau de tolérance aux pannes requis
- Second facteur :
La capacité de disque requise pour conserver les messages et la quantité de stockage disponible sur chaque *broker*.
- 3ème facteur :
La capacité du cluster à traiter le débit de requêtes en profitant du parallélisme.



APIs

Producer API

Consumer API

Sérialisation Avro ou ProtoBuf

Frameworks Java

Connect API

Admin et Stream API



Introduction

L'API est simple mais les contraintes applicatives influencent la façon de l'utiliser ainsi que la configuration des *topics*

Les questions devant être posées :

- Chaque message est-il critique, ou peut-on tolérer des pertes de messages?
- La duplication accidentelle de messages est elle autorisée ?
- Y-a-t-il des exigences strictes de latence ou de débit ?



Dépendances

Java

```
<dependency>  
    <groupId>org.apache.kafka</groupId>  
    <artifactId>kafka-clients</artifactId>  
    <version>${kafka-version}</version>  
</dependency>
```

Python

```
pip install confluent-kafka
```

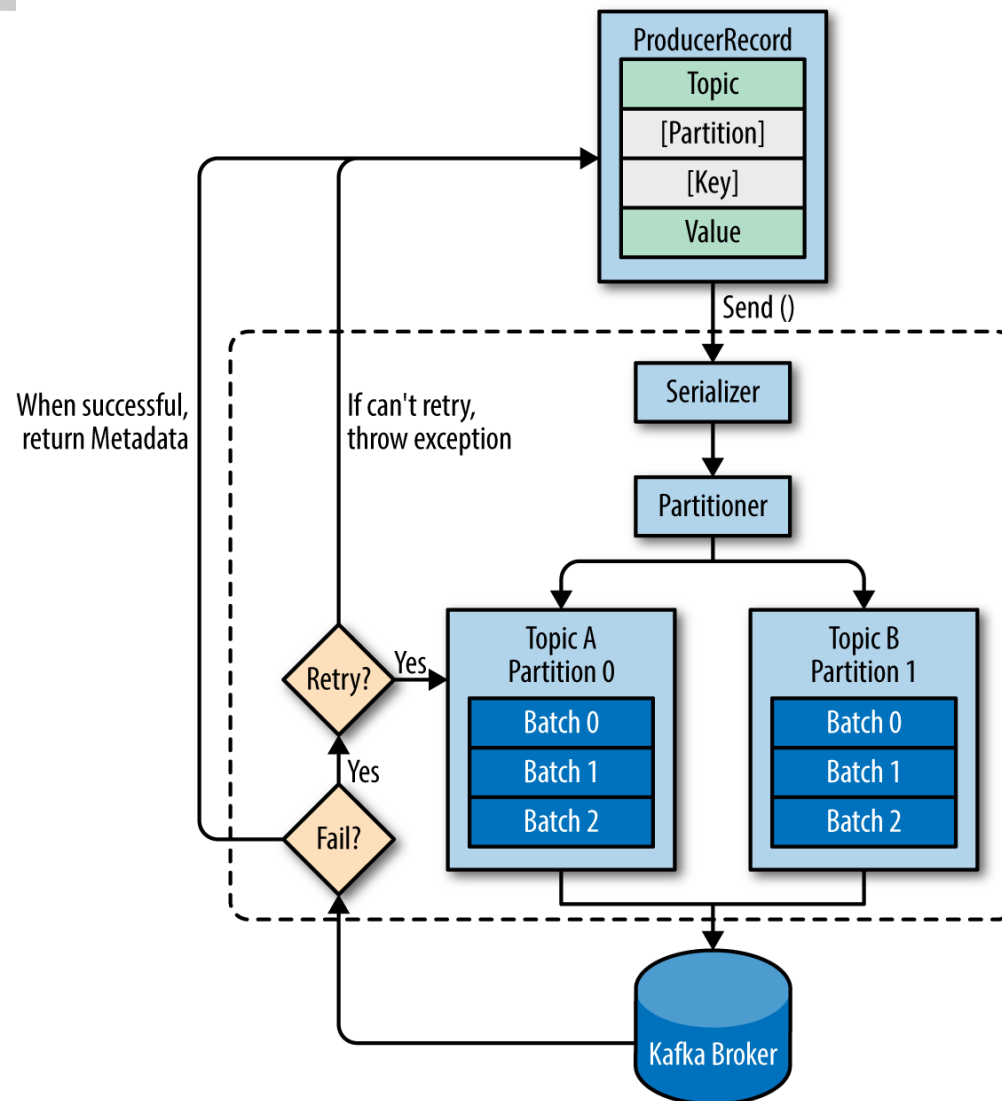



Étapes lors de l'envoi d'un message

L'envoi de message est constitué de plusieurs étapes :

- Création d'un objet **ProductRecord** encapsulant le topic et optionnellement une clé et une partition
- L'objet est **sérialisé** pour préparer sa transmission sur le réseau
- Les données sont ensuite fournies à un **partitionneur** qui détermine la partition de destination, (à partir de la partition indiquée, de la clé du message ou en Round-robin)
- Une fois la partition sélectionnée, le message est ajouté à un **lot de messages** destiné à la même partition. Une thread séparée envoie le lot de messages.
- Lorsque le broker reçoit le message, il renvoie une réponse sous le forme d'un objet **RecordMetadata** encapsulant le *topic*, la partition, la clé et l'offset
- Si le broker n'arrive pas à écrire le message dans le journal, il renvoie une erreur et le producteur peut réessayer un certain nombre de fois

Envoi de message





Construire un Producteur

La première étape pour l'envoi consiste à instancier un ***KafkaProducer***.

3 propriétés de configurations sont obligatoires :

- ***bootstrap.servers*** : Liste de brokers que le producteur contacte au départ pour découvrir le cluster
- ***key.serializer*** : La classe utilisée pour la sérialisation de la clé
- ***value.serializer*** : La classe utilisée pour la sérialisation du message ...

1 optionnelle est généralement positionnée :

- ***client.id*** : Permet le suivi des messages



Exemple Java

```
private Properties kafkaProps = new Properties();  
kafkaProps.put("bootstrap.servers",  
    "broker1:9092,broker2:9092");  
kafkaProps.put("key.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");  
kafkaProps.put("value.serializer",  
    "org.apache.kafka.common.serialization.StringSerializer");  
  
producer = new KafkaProducer<String, String>(kafkaProps);
```



Exemple python

```
from confluent_kafka import Producer  
import socket
```

```
conf = {'bootstrap.servers':  
        "host1:9092,host2:9092"}
```

```
producer = Producer(conf)
```

Rq : Serialiseurs sont fournies « out-of the box »



ProducerRecord

ProducerRecord représente l'enregistrement à envoyer à Kafka.

Il contient le nom du *topic*, une valeur et éventuellement une clé, une partition, un timestamp

Constructeurs :

Sans clé

`ProducerRecord(String topic, V value)`

Avec clé

`ProducerRecord(String topic, K key, V value)`

Avec clé et partition

`ProducerRecord(String topic, Integer partition, K key, V value)`

Avec clé, partition et timestamp

`ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)`



Méthodes d'envoi des messages

Il y a 3 façons d'envoyer des messages :

- ***Fire-and-forget*** : Pas d'acquittement, on s'autorise à perdre quelques messages (même si c'est très rare)
- ***Envoi synchrone*** : La méthode renvoie un objet *Future* sur lequel on appelle la méthode *get()* pour attendre la réponse.
On traite éventuellement les cas d'erreurs
- ***Envoi asynchrone*** : Lors de l'envoi, on passe en argument une fonction de call-back.
La méthode est appelée lorsque la réponse est retournée



Fire And Forget

Java

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry", "Precision", "France");  
  
try {  
    producer.send(record);  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

Python

```
producer.produce(topic, key="key", value="value")
```




Envoi synchrone

Java

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry", "Precision", "France");  
  
try {  
    producer.send(record).get();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

Python

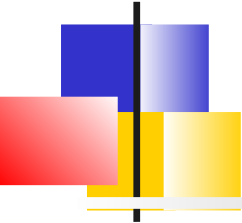
```
producer.produce(topic, key="key", value="value")  
producer.flush()
```



Envoi asynchrone avec callback (Java)

```
private class DemoProducerCallback implements Callback {  
    @Override  
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {  
        if (e != null) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
ProducerRecord<String, String> record =  
    new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA");  
producer.send(record, new DemoProducerCallback());
```



Envoi asynchrone avec callback (Python)

```
def acked(err, msg):
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err)))
    else:
        print("Message produced: %s" % (str(msg)))

producer.produce(topic, key="key", value="value", callback=acked)

# Wait up to 1 second for events. Callbacks will be invoked during
# this method call if the message is acknowledged.
producer.poll(1)
```

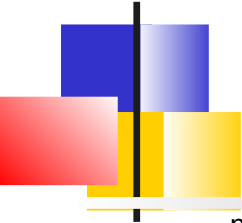


Sérialiseurs

Kafka inclut les classes ***ByteArraySerializer*** et ***StringSerializer*** utile pour types basiques.

Pour des objets du domaine, il faut implémenter ses propres sérialiseurs/désérialiseurs en s'appuyant sur des librairies comme *Avro*, *Thrift*, *Protobuf* ou *Jackson*

La version commerciale de Confluent privilégie le sérialiseur *Avro* et permet une gestion fine des formats de sérialisation via les ***Schema Registry***



Exemple sérialiseur s'appuyant sur Jackson

```
public class JsonPOJOSerializer<T> implements Serializer<T> {
    private final ObjectMapper objectMapper = new ObjectMapper();
    /**
     * Default constructor requis par Kafka
     */
    public JsonPOJOSerializer() { }

    @Override
    public void configure(Map<String, ?> props, boolean isKey) { }

    @Override
    public byte[] serialize(String topic, T data) {
        if (data == null)
            return null;

        try {
            return objectMapper.writeValueAsBytes(data);
        } catch (Exception e) {
            throw new SerializationException("Error serializing JSON message", e);
        }
    }

    @Override
    public void close() { }
}
```



Exemple désérialiseur basé sur Jackson

```
public class JsonPOJODeserializer<T> implements Deserializer<T> {
    private ObjectMapper objectMapper = new ObjectMapper();
    private Class<T> tClass;
    // Default constructor requis par Kafka
    public JsonPOJODeserializer() { }

    @Override
    public void configure(Map<String, ?> props, boolean isKey) {
        tClass = (Class<T>) props.get("JsonPOJOClass");
    }

    @Override
    public T deserialize(String topic, byte[] bytes) {
        if (bytes == null)
            return null;
        T data;
        try {
            data = objectMapper.readValue(bytes, tClass);
        } catch (Exception e) {
            throw new SerializationException(e);
        }
        return data;
    }
    @Override
    public void close() { }
}
```



Envoi de message

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("value.serializer", "org.myappli.JsonPOJOSerializer");

String topic = "customerContacts";

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

Customer customer = CustomerGenerator.getNext();

ProducerRecord<String, Customer> record =
    new ProducerRecord<>(topic, customer.getId(), customer);
producer.send(record);
```



Configuration des producteurs *fiabilité*

Certains paramètres ont un impact significatif sur l'utilisation de la mémoire, les performances et la fiabilité des producteurs.

Fiabilité :

- **acks** : contrôle le nombre de réplicas qui doivent recevoir l'enregistrement avant que le producteur puisse considérer l'écriture comme réussie
- **retries** : Si l'erreur renvoyée est de type Retriable, le nombre de tentative de renvoi. Si > 0 possibilité d'envoi en doublon
- **max.in.flight.requests.per.connection** : Maximum de message en cours de transmission (sans réponse obtenu)
- **enable.idempotence** : Livraison unique de message
- **transactional.id** : Mode transactionnel



Configuration des producteurs *performance*

- ***batch.size*** : La taille du batch en mémoire pour envoyer les messages.
Défaut 16ko
- ***linger.ms*** : la durée d'attente de messages supplémentaires avant d'envoyer le batch courant. Défaut 0ms
- ***buffer.memory*** : Taille buffer pour stocker les messages que l'on ne peut pas envoyé. Défaut 32Mo
- ***compression.type*** : Par défaut, les messages ne sont pas compressés. Valeurs possibles : *snappy* , *gzip* , ou *lz4*
- ***request.timeout.ms***, ***metadata.fetch.timeout.ms*** et ***timeout.ms***:
Timeouts pour la réception d'une réponse à un message, pour obtenir des méta-données (leader, etc..) pour obtenir le ack des répliques.
- ***max.block.ms*** : Temps maximum d'attente pour la méthode *send()*.
Dans le cas ou le buffer est rempli
- ***max.request.size*** : Taille max d'un message
- ***receive.buffer.bytes*** et ***send.buffer.bytes***: Taille des buffers TCP



Garantie sur l'ordre

En absence de failure, *Kafka* préserve l'ordre des messages au sein d'une partition.

- Si des messages ont été envoyés par le producteur dans un ordre spécifique, le broker les écrit sur une partition dans cet ordre et tous les consommateurs les liront dans cet ordre ...

En cas de failure

- Si *retries* > 0 et *max.in.flights.requests.per.session* > 1 .
Il se peut que lors d'un renvoi l'ordre initial soit inversé.
- Pour avoir une garantie sur l'ordre avec tolérance aux fautes, on configure
retries > 0 et *max.in.flights.requests.per.session* = 1
(au détriment du débit global)



APIs

Producer API

Consumer API

Sérialisation Avro ou ProtoBuf

Frameworks Java

Connect API

Admin et Stream API



Introduction

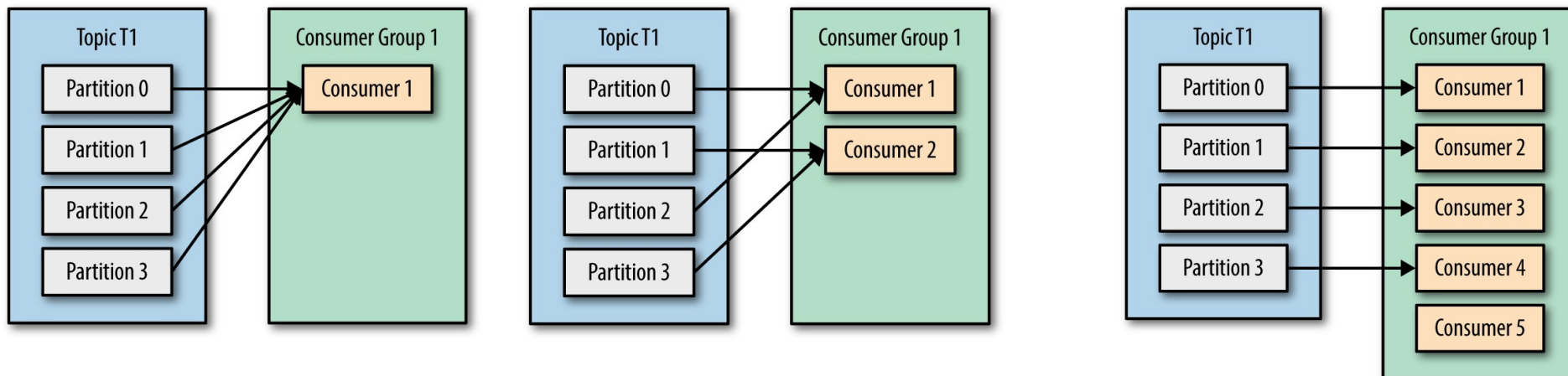
Les applications qui ont besoin de lire les données de Kafka utilisent un ***KafkaConsumer*** pour s'abonner aux topics Kafka

Pour bien comprendre l'API, il faut comprendre la notion de groupe de consommateurs et leurs relations avec les partitions

Groupes de consommateurs

Les consommateurs font généralement partie d'un **groupe de consommateurs**.

- Chaque consommateur d'un groupe reçoit les messages d'un sous-ensemble différent des partitions du topic.





Rééquilibrage dynamique des consommateurs

Lors de l'ajout d'un nouveau consommateur, celui-ci peut se faire affecter une partition consommée précédemment par un autre consommateur du groupe.

Lors de l'arrêt d'un consommateur, la partition qui lui était assignée est réaffectée à un autre consommateur

Cette répartition dynamique offre la scalabilité et la tolérance aux défaillances mais n'est pas spécialement désirable

- car durant le rééquilibrage les messages ne sont pas consommés
- les consommateurs si ils utilisent des caches sont obligés de les rafraîchir



Membership et détection de pannes

Les consommateurs maintiennent leur appartenance à un groupe et leurs attributions de partitions en envoyant régulièrement des *heartbeat* à un broker coordinateur (qui peut être différent en fonction des groupes).

```
kafka-consumer-groups.sh --bootstrap-server  
localhost:9092 --group sample --describe --state
```

Si un consommateur cesse d'envoyer des *heartbeats*, sa session expire et le coordinateur démarre une réaffectation des partitions



Création de *KafkaConsumer*

L'instanciation d'un ***KafkaConsumer*** est similaire à celle d'une *KafkaProducer*

Plusieurs propriétés doivent être spécifiées dans une classe *Properties* :

- *bootstrap.servers*
- *key.deserializer* , et *value.deserializer*
- *group.id* qui spécifie le groupe de consommateur



Example

Java

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers",
    "broker1:9092,broker2:9092");
kafkaProps.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
kafkaProps.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
kafkaProps.put("group.id", "myGroup");
```

```
consumer = new KafkaConsumer<String, String>(kafkaProps);
```

Python

```
from confluent_kafka import Consumer
conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'group.id': "foo", 'auto.offset.reset': 'smallest'}
consumer = Consumer(conf)
```



Abonnement à un *topic*

Après la création d'un consommateur, il faut souscrire à un *topic*

La méthode ***subscribe()*** prend une liste de *topics* comme paramètre.

Ex :

```
consumer.subscribe(Collections.singletonList("myTopic"));
```

Il est également possible d'utiliser ***subscribe()*** avec une expression régulière

```
consumer.subscribe("test.*");
```



Boucle de Polling

Typiquement, les consommateurs *poll* continuellement les *topics* auxquels ils sont abonnés.

Les objets retournés par *poll* sont une collection de ***ConsumerRecord*** qui encapsule :

- le message :
- La partition
- L'offset
- Le timestamp



Exemple Java

```
try {
while (true) {
    // poll envoie le heartbeat, on bloque pdt 100ms pour récupérer les messages
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> 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.containsValue(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount) ;

        System.out.println(custCountryMap) ;
    }
} finally {
    consumer.close();
}
```



Exemple Python

```
running = True
```

```
def basic_consume_loop(consumer, topics):
```

```
    try:
```

```
        consumer.subscribe(topics)
```

```
        while running:
```

```
            msg = consumer.poll(timeout=1.0)
```

```
            if msg is None: continue
```

```
            if msg.error():
```

```
                if msg.error().code() == KafkaError._PARTITION_EOF:
```

```
                    # End of partition event
```

```
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %  
                                     (msg.topic(), msg.partition(), msg.offset()))
```

```
                elif msg.error():
```

```
                    raise KafkaException(msg.error())
```

```
            else:
```

```
                msg_process(msg)
```

```
    finally:
```

```
        # Close down consumer to commit final offsets and trigger rebalance.
```

```
        consumer.close()
```

```
def shutdown():
```

```
    running = False
```



Thread et consommateur

Il n'est pas possible d'avoir plusieurs consommateurs du même groupe dans la même thread et il n'est pas possible d'utiliser le même consommateur dans plusieurs threads

=> 1 consommateur = 1 thread

Pour exécuter plusieurs consommateurs d'un même groupe dans une application, il est utile d'utiliser les classes *ExecutorService* de Java.



Configuration des consommateurs

Les propriétés les plus importantes :

- ***auto.offset.reset*** : *latest* (défaut) ou *earliest*.
Contrôle le comportement du consommateur si il ne détient pas d'offset valide.
Dernier message ou le plus ancien
- ***enable.auto.commit*** : Le consommateur commit les offsets automatiquement ou non.
Par défaut true
Si true :
 - ***auto.commit.interval.ms*** : Intervalle d'envois des commits



Configuration des consommateurs (2)

D'autres propriétés

- ***fetch.min.bytes*** : Volume minimum de données à recevoir. Permet de réduire la charge sur le broker et le consommateur
- ***fetch.max.wait.ms*** : Attente maximale avant de récupérer les données
- ***max.partition.fetch.bytes*** : Maximum de données par partition ramenées lors d'un poll. Par défaut 1Mo
- ***max.poll.records*** : Maximum de record via un *poll()*
- ***session.timeout.ms*** : Le temps faisant expirer la session et déclarer le consommateur comme down. Par défaut 10s
- ***heartbeat.interval.ms*** : L'intervalle d'envoi des heartbeat. Par défaut 3s
- ***partition.assignment.strategy*** : Stratégie d'affectation des partitions
Range (défaut), *RoundRobin* ou *Custom*
- ***client.id*** : Une chaîne de caractère utilisé pour les métriques.
- ***receive.buffer.bytes*** et ***send.buffer.bytes*** : Taille des buffers TCP



Offsets et Commits

Les consommateurs peuvent suivre leurs offsets de partition en s'adressant à Kafka.

Kafka appelle la mise à jour d'un offset : un ***commit***

Pour committer, un consommateur envoie un message vers un *topic* particulier de Kafka :

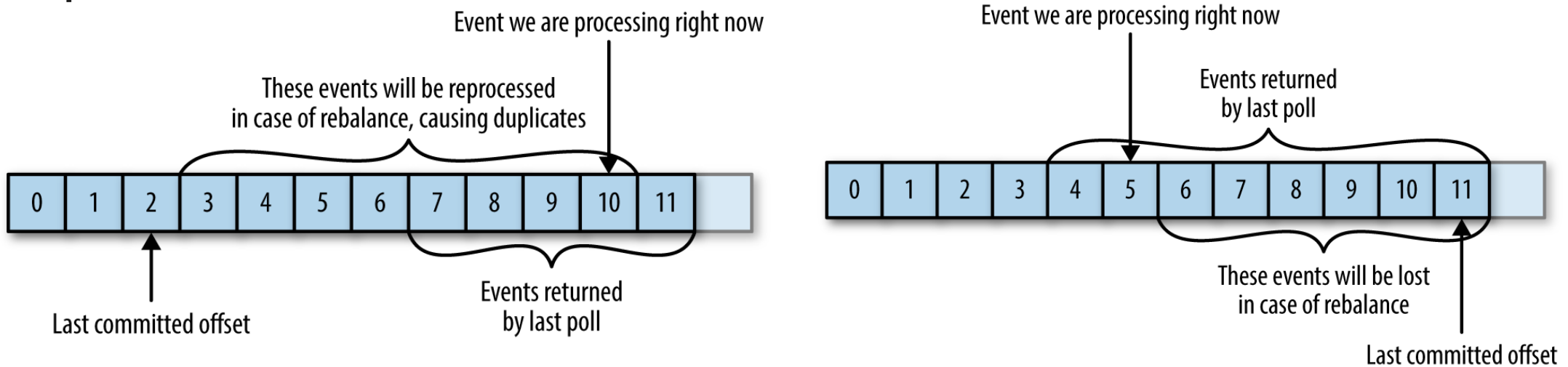
__consumer_offsets

- Ce *topic* contient les offsets de chaque partition.

Lors d'une réaffectation de partitions, 2 risques doivent être pris en compte :

- Traiter 2 fois le même message
- Louper des messages

Risques lors d'une réaffectation



Kafka propose plusieurs façons de gérer les commits



Commit automatique

Si ***enable.auto.commit=true*** ,
le consommateur valide toutes les
auto.commit.interval.ms (par défaut 5000), les
plus grands offset reçus par *poll()*

=> Cette approche (simple) ne protège pas contre
les duplications ou les pertes de message en cas
de ré-affectation

Cela dépend de la grandeur de l'intervalle en
fonction du temps de traitement des messages
ramenés par *poll()*



Commit contrôlé

L'API Consumer permet de contrôler le moment du commit plutôt que de se baser sur un timer.

Si ***auto.commit.offset=false*** ,
l'application doit explicitement committer les offsets

- Soit de façon bloquante avec ***commitSync()***
- Soit en asynchrone avec ***commitAsync()***



Commit Synchrone

La méthode ***commitSync()*** valide les derniers offsets reçus par *poll()*

- La méthode est bloquante et retourne lorsque les offsets sont commités
- Elle lance une exception si un commit échoue

=> En cas de réaffectation, il y a toujours un risque pour que le messages soit traités plusieurs fois



Exemple Java

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        // Si réaffectation, possibilité de duplication
        log.info("topic = %s, partition = %s,
            offset =%d, customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value());
    }
    try {
        consumer.commitSync();
    } catch (CommitFailedException e) {
        log.error("commit failed", e)
    }
}
```



Exemple Python

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(asynchronous=False)

    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```



Commit asynchrone

La méthode ***commitAsync()*** est non bloquante

- En cas d'erreur, 2 cas de figure en fonction de la configuration de *retry* :
 - Soit *retry* et erreur de type *Retriable*, en fonction de la configuration, la méthode peut effectuer un renvoi.
Attention, cela peut provoquer des ordres de commits dans le mauvais ordre
 - Si pas de *retry*, alors c'est le prochain appel à commit qui validera les offsets
- Il est possible de fournir une méthode de callback en argument



Exemple Java

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        Log.info("topic = %s, partition = %s,
            offset = %d, customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
            OffsetAndMetadata> offsets, Exception exception) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
    });
}
```



Exemple Python

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(asynchronous=True)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```



Python Commit avec callback

```
from confluent_kafka import Consumer
```

```
def commit_completed(err, partitions):  
    if err:  
        print(str(err))  
    else:  
        print("Committed partition offsets: " + str(partitions))
```

```
conf = {'bootstrap.servers': "host1:9092,host2:9092",  
        'group.id': "foo",  
        'default.topic.config': {'auto.offset.reset': 'smallest'},  
        'on_commit': commit_completed}
```

```
consumer = Consumer(conf)
```



Committer un offset spécifique

L'API permet également de fournir en argument une *Map* contenant les offsets que l'on veut valider pour chaque partition

- Cela permet de committer sans avoir traité l'intégralité des messages ramenés par *poll()*



Example

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new
    HashMap<>();
int count = 0;
....
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        log.info("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(), record.offset(),
            record.key(), record.value());
        currentOffsets.put(new TopicPartition(record.topic(), record.partition()),
new OffsetAndMetadata(record.offset()+1, "no metadata"));
        if (count % 1000 == 0)
            consumer.commitAsync(currentOffsets, null);
        count++;
    }
}
```



Stocker les offsets hors de Kafka

Il n'y a pas d'obligation à stocker les offsets dans Kafka, un consommateur peut stocker lui même les offsets dans son propre data store.

Si les offsets et le résultat du traitement sont stockés dans la même BD, on peut alors profiter d'une transaction et d'une écriture atomique.

Ce type de scénario permet d'obtenir très facilement des garanties de livraison « *Exactly Once* ».



Réagir aux réaffectations

Lors de réaffectations de partitions, les consommateurs peuvent être prévenus afin de prendre les mesures adéquates (commit, fermeture de ressources, ...)

Lors du *subscribe()*, il faut fournir une classe de type ***ConsumerRebalanceListener*** qui implémente 2 méthodes :

- `public void onPartitionsRevoked(Collection<TopicPartition> partitions)`
- `public void onPartitionsAssigned(Collection<TopicPartition> partitions)`



Example

```
private class HandleRebalance implements
    ConsumerRebalanceListener {

    public void onPartitionsAssigned(
        Collection<TopicPartition> partitions) { }

    public void onPartitionsRevoked(
        Collection<TopicPartition> partitions) {
        log.info("Lost partitions in rebalance.
            Committing current offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets);
    }
}
```




Consommation de messages avec des offsets spécifiques

L'API permet d'indiquer un offset spécifique :

- ***seekToBeginning(TopicPartition tp)*** :
Revenir au début de la partition
- ***seekToEnd(TopicPartition tp)*** :
Se placer à la fin
- ***seek(TopicPartition, long)*** :
Se placer à un offset particulier
Cela permet de stocker les offsets en dehors de Kafka et en cas d'erreur ou redémarrage repartir à partir des offsets sauvegardés



Sortie de boucle

Pour sortir de la boucle de *poll*, il faut qu'une autre thread appelle ***consumer.wakeup()*** qui a pour effet de lancer une *WakeupException* lors de l'appel à *poll*.

Le consommateur doit alors faire un appel explicite à *close()*

On peut utiliser
Runtime.addShutdownHook(Thread hook)



Example

```
Runtime.getRuntime().addShutdownHook(new Thread() {  
    public void run() {  
        System.out.println("Starting exit...");  
        consumer.wakeup();  
        try {  
            mainThread.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
});
```



Example (2)

```
try {
// looping until ctrl-c, the shutdown hook will cleanup on exit
while (true) {
    ConsumerRecords<String, String> records = movingAvg.consumer.poll(1000);
    log.info(System.currentTimeMillis() + "-- waiting for data...");
    for (ConsumerRecord<String, String> record : records) {
        log.info("offset = %d, key = %s,value = %s\n",
            record.offset(), record.key(),record.value());
    }
    for (TopicPartition tp: consumer.assignment())
        log.info("Committing offset atposition:"+consumer.position(tp));
    movingAvg.consumer.commitSync();
}
} catch (WakeupException e) {
} finally {
    consumer.close();
    log.info("Closed consumer and we are done");
}
```



Affectation statique des partitions

Si l'application ne comporte que des consommateurs dédiés statiquement à des partitions, alors la fonctionnalité de groupe n'est pas nécessaire.

Dans ce cas, l'application assigne explicitement les partitions à ses consommateurs.

L'API ***assign()*** est alors utilisée à la place de *subscribe()*



Exemple

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic");

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),partition.partition()));

    consumer.assign(partitions);
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record: records) {
            log.info("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```



APIs

Producer API

Consumer API

Sérialisation Avro ou ProtoBuf

Frameworks Java

Connect API

Admin et Stream API



Introduction

Lors d'évolution des applications, le format des messages est susceptible de changer.

Afin que les consommateurs supportent ces évolutions sans être obligés d'être mis à jour, une simple sérialisation JSON ne suffit pas.

Des bibliothèques de sérialisation comme **Avro** ou **ProtoBuf** adressent ce problème.

Elles offrent également un format de sérialisation binaire plus compact



Apache Avro

Apache Avro est un système de sérialisation de données.

- Il utilise une structure JSON pour définir le **schéma**, permettant la sérialisation entre les octets et les données structurées.
- Les outils associés à Avro sont capables de générer les classes Java¹ correspondantes au schéma.

1. Avro supporte C, C++, C#, Java, PHP, Python, et Ruby



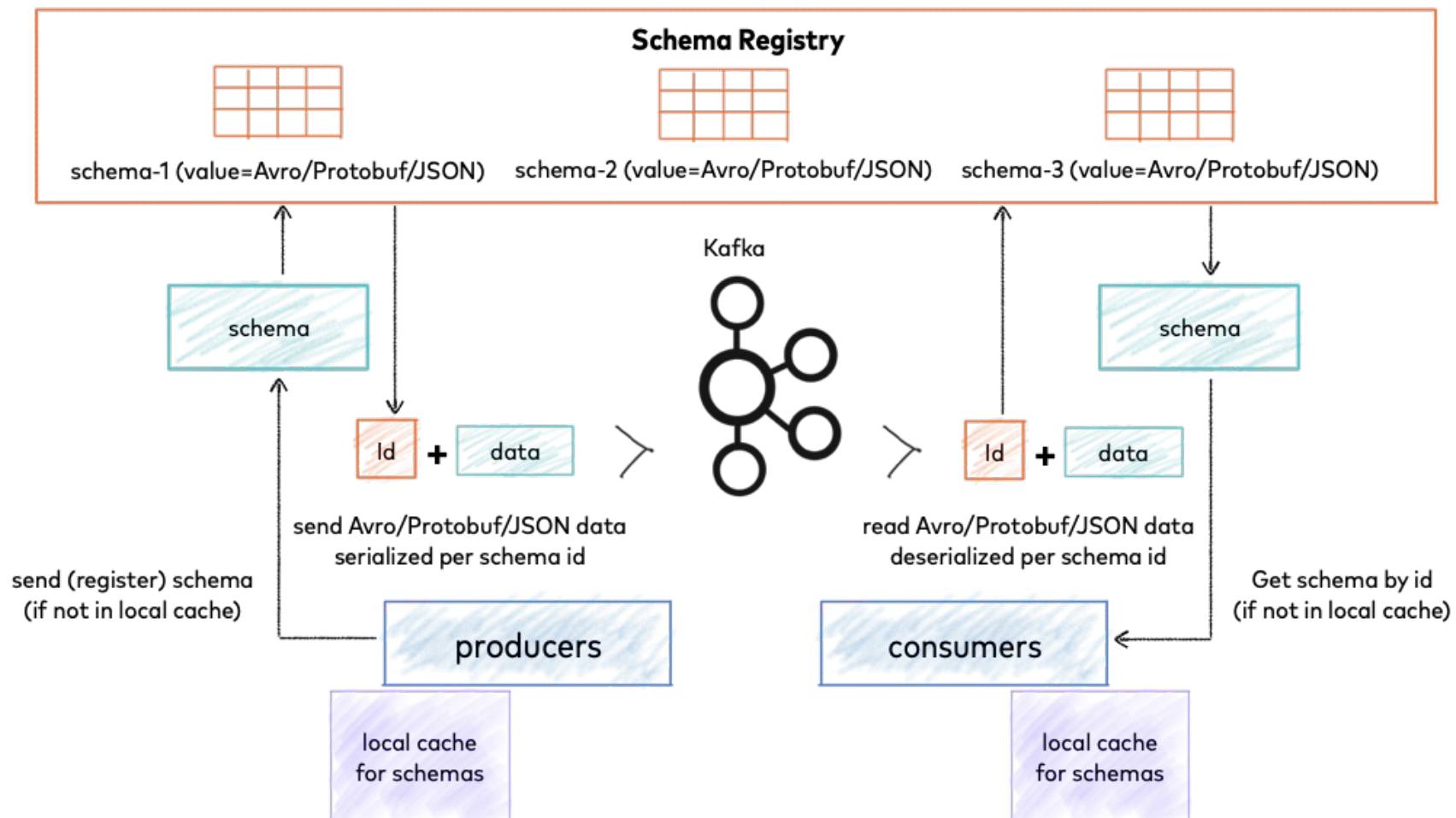
Utilisation du schéma

Avro suppose que le schéma est présent lors de la lecture et l'écriture des fichiers.

- La disponibilité du schéma offre un énorme plus pour l'évolutivité du producteur.
=> Le format du message peut évoluer sans impacter le code des consommateurs

Confluent Platform intègre le composant **Schema Registry** qui offre une API Rest permettant d'accéder aux schémas stockés par les producteurs aux formats Avro, Protobuf et JSON

Schema Registry





Dépendances Confluent

```
<dependencies>
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-schema-registry</artifactId>
    <version>7.2.1</version>
  </dependency>
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>7.2.1</version>
  </dependency>
</dependencies>
<repositories>
  <repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
  </repository>
</repositories>
```



Schéma Avro

```
{
  "type": "record",
  "name": "Courier",
  "namespace": "org.formation.model",
  "fields": [
    {
      "name": "id",
      "type": "int" },
    {
      "name": "firstName",
      "type": "string" },
    {
      "name": "lastName",
      "type": "string" },
    {
      "name": "position",
      "type": [
        {
          "type": "record",
          "name": "Position",
          "namespace": "org.formation.model",
          "fields": [
            {
              "name": "latitude",
              "type": "double" },
            {
              "name": "longitude",
              "type": "double" }
            ]
          }
        ]
      ]
    }
  ]
}
```



Plug-in Avro

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>1.8.2</version>
  <executions>
    <execution>
      <id>schemas</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
        <goal>protocol</goal>
        <goal>idl-protocol</goal>
      </goals>
      <configuration>
        <sourceDirectory>./src/main/resources/</sourceDirectory>
        <outputDirectory>./src/main/java/</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

=> *mvn compile* génère les classes du modèle



Producteur (1)

Le producteur de message doit contenir du code permettant d'enregistrer le schéma en utilisant la librairie cliente de *Schema Registry*

```
// Lecture du schéma
Schema avroSchema = new Schema.Parser().parse(inputStream);
// Instanciation du client
CachedSchemaRegistryClient client = new
    CachedSchemaRegistryClient(REGISTRY_URL, 20);
// Enregistrement du sujet, le nom correspond au nom du topic + suffixe
client.register(topicName + "-value", new AvroSchema(avroSchema));
```



Producteur (2)

Les propriétés du *KafkaProducer* doivent contenir :

- ***schema.registry.url*** :
L'adresse du serveur de registry
- Le sérialiseur de valeur :
io.confluent.kafka.serializers.KafkaAvroSerializer



Consommateur

KafkaConsumer doit également préciser l'URL et le désérialiseur à ***io.confluent.kafka.serializers.KafkaAvroDeserializer***

Il peut récupérer les messages sous la forme de ***GenericRecord*** plutôt que des classes spécialisés.

```
ConsumerRecords<String, GenericRecord> records =  
    consumer.poll(Duration.ofMillis(sleep));  
for (ConsumerRecord<String, GenericRecord> record : records) {  
    System.out.println("Value is " + record.value());  
}
```



APIs

Producer API
Consumer API
Sérialisation Avro ou ProtoBuf
Frameworks Java
Connect API
Admin et Stream API



Introduction

De nombreux frameworks permettent une intégration avec Kafka

Ils facilitent la production/consommation de messages, la configuration des niveaux de fiabilité des topics et proposent des modèles de programmation réactif



Eclipse Vert.x

Vert.x se définit comme une boîte à outil dédiée aux applications réactive basées sur une JVM

Vert.x propose des clients réactifs pour des APIs Web, des Bds, ... et des systèmes de messagerie comme Kafka



Vert.x Consommateur

```
// Création du consommateur : identique à KafkaConsumer API
```

```
...
```

```
// Définition du traitement
```

```
consumer.handler(record -> {  
    System.out.println("Processing key=" + record.key() + ",value=" + record.value() +  
        ",partition=" + record.partition() + ",offset=" + record.offset());  
});
```

```
// S'abonner à plusieurs topics
```

```
Set<String> topics = new HashSet<>();  
topics.add("topic1");  
topics.add("topic2");  
consumer.subscribe(topics);
```

```
// Avec une regex
```

```
Pattern pattern = Pattern.compile("topic\\d");  
consumer.subscribe(pattern);
```

```
// or un simple topic
```

```
consumer.subscribe("a-single-topic");
```



Vert.x Producteur

```
for (int i = 0; i < 5; i++) {  
  
    // Round-robin sur les partitions de destination  
    KafkaProducerRecord<String, String> record =  
        KafkaProducerRecord.create("test", "message_" + i);  
  
    // Envoi asynchrone  
    producer.send(record).onSuccess(rMeta ->  
        System.out.println(  
            "Message " + record.value() + " écrit sur" + rMeta.getTopic() +  
            ", partition=" + rMeta.getPartition() +  
            ", offset=" + rMeta.getOffset()  
        )  
    );  
}
```



Eclipse MicroProfile

Eclipse MicroProfile peut être vu comme un sous-ensemble de la spécification *Jakarta EE (JavaEE)* dédié aux micro-services déployés dans le cloud.

La spécification inclut ***MP Reactive Messaging*** qui fournit une abstraction d'un Message Broker.

Les frameworks implémentant cette API proposent des intégrations avec Kafka.

Par exemple, la librairie réactive *SmallRye*, présente dans *Quarkus*, *Wildfly*, *Open Liberty*, propose une intégration Kafka (qui s'appuie sur Eclipse Vert.x)



MP Reactive Messaging

Objectif : Fournir un support pour le messaging asynchrone basé sur Reactive Streams, indépendamment du message Broker.

Concepts :

- Les applications échangent des **messages** : Payload et méta-données.
Les messages sont acquittés après traitement
- Les messages transitent dans des **canaux** (channels).
Les applications se connectent aux canaux.
- Certains canaux sont internes à l'application d'autres sont associés à des système de messagerie sous-jacent via des **connecteurs (connector)**.



Exemple *SmallRye*

Producteur

```
@ApplicationScoped
public class KafkaPriceProducer {
    private Random random = new Random();

    @Outgoing("prices")
    public Multi<Double> generate() {
        // It emits a price every second
        return Multi.createFrom().ticks().every(Duration.ofSeconds(1))
            .map(x -> random.nextDouble());
    }
}
```

Consommateur

```
@ApplicationScoped
public class KafkaPriceMessageConsumer {

    @Incoming("prices")
    public CompletionStage<Void> consume(Message<Double> price) {
        // Traitement
        // Puis Commit du offset
        return price.ack();
    }
}
```



Envoi avec *Emitter*

La signature des méthodes de *@Outgoing* ne permet pas le passage de paramètre.

L'autre façon d'envoyer des messages est de se faire injecter par le framework un bean ***Emitter***.

Le bean propose la méthode *send()* qui retourne un *CompletionStage*, terminé lorsque le message est acquitté.



Envoi avec *Emitter*

```
@Path("/prices")
public class PriceResource {

    // La configuration par défaut associe le canal "price-create"
    // au topic Kafka "price-create"
    // La surcharge de la config par défaut s'effectue
    // via application.properties
    @Inject
    @Channel("price-create") Associé au topic price-create
    Emitter<Double> priceEmitter;

    @POST
    @Consumes(MediaType.TEXT_PLAIN)
    public void addPrice(Double price) {
        // Exception si nack
        CompletionStage<Void> ack = priceEmitter.send(price);
    }
}
```



Connecteurs

Les **connecteurs** sont responsable
d'associer un canal à un *sink* ou *source* de
messages spécifique à une technologie

La configuration s'effectue via *MP Config*

Elle suit la structure suivante :

```
mp.messaging.incoming.[channel-name].[attribute]=[value]  
mp.messaging.outgoing.[channel-name].[attribute]=[value]  
mp.messaging.connector.[connector-name].[attribute]=[value]
```

Exemples :

```
mp.messaging.outgoing.prices-out.connector=smallrye-kafka  
mp.messaging.outgoing.prices-out.topic=prices
```



Configuration Kafka

Canal d'entrée (Incoming)

- *topic, key.deserializer, value.deserializer, group.id, auto.offset.reset, partitions*
- *fetch.min.bytes, poll-timeout, batch*
- *enable.auto.commit, commit-strategy*
- *retry, retry-attempts, retry-max-wait, failure-strategy*
-

Canal de sortie (Outgoing)

- *topic, key, key.deserializer, value.deserializer,*
- *max-inflight-messages, retries*
- *acks, waitForWriteCompletion*
- ...



Spring Boot

L'écosystème *SpringBoot* propose des starters permettant l'intégration avec Kafka :

- ***org.springframework.kafka:spring-kafka:***
Production/Consommation d'enregistrements Kafka
- ***org.apache.kafka :kafka-streams:***
Intégration de Kafka Streams.
- ***org.springframework.cloud :spring-cloud-stream:***
Abstraction pour développer des micro-services basés sur les événements (Compatible avec Apache Kafka, RabbitMQ ou Solace PubSub+ via des binders)
équivalent à *MicroProfile Messaging*



Exemple *spring-kafka*

Producteur :

```
@RestController
public class Controller {

    @Autowired
    private KafkaTemplate<Object, Object> template;

    @PostMapping(path = "/send/foo/{what}")
    public void sendFoo(@PathVariable String what) {
        this.template.send("topic1", new Foo1(what));
    }
}
```

Consommateur :

```
@KafkaListener(id = "fooGroup", topics = "topic1")
public void listen(Foo2 foo) {
    logger.info("Received: " + foo);
    if (foo.getFoo().startsWith("fail")) { throw new RuntimeException("failed"); }
    this.new SimpleAsyncTaskExecutor().execute(
        () -> System.out.println("Hit Enter to terminate...")
    );
}
```



cloud-stream : production

```
@EnableBinding(Source.class)
```

```
public class SampleSource {
```

```
    private final Log logger = LoggerFactory.getLog(getClass());
```

```
    @Bean
```

```
    @InboundChannelAdapter(value = Source.SAMPLE, poller = @Poller(fixedDelay = "1000", maxMessagesPerPoll = "1"))
```

```
    public MessageSource<String> timerMessageSource() {
```

```
        return new MessageSource<String>() {
```

```
            public Message<String> receive() {
```

```
                logger.info("*****\nAt the Source\n*****");
```

```
                String value = "{\n\"value\":\n\"hi\"}";
```

```
                logger.info("Sending value: " + value);
```

```
                return MessageBuilder.withPayload(value).build();
```

```
            }
```

```
        };
```

```
    }
```

```
    public interface Source {
```

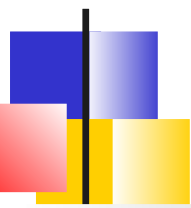
```
        String SAMPLE = "sample-source";
```

```
        @Output(SAMPLE)
```

```
        MessageChannel sampleSource();
```

```
    }
```

```
}
```

cloud-stream : consommation

```
@EnableBinding(SampleSink.Sink.class)
```

```
public class SampleSink {
```

```
    private final Log logger = LoggerFactory.getLog(getClass());
```

```
    // Sink application definition
```

```
    @StreamListener(Sink.SAMPLE)
```

```
    public void receive(Foo foo) {
```

```
        logger.info("*****\nAt the Sink\n*****");
```

```
        logger.info("Received transformed message " + foo.getValue() + " of type " + foo.getClass());
```

```
    }
```

```
    public interface Sink {
```

```
        String SAMPLE = "sample-sink";
```

```
        @Input(SAMPLE)
```

```
        SubscribableChannel sampleSink();
```

```
    }
```

```
}
```



Binding vers topics Kafka

```
spring:
  cloud:
    stream:
      bindings:
        consumer-in-0 :    # Binding sur le nom de méthode
          destination : input
        sample-source:
          destination: testtock
        input:
          destination: testtock
        output:
          destination: xformed
        sample-sink:
          destination: xformed
```



APIs

Producer API
Consumer API
Frameworks

Connect API

Admin et Stream API



Introduction

Kafka Connect permet d'intégrer Apache Kafka avec d'autres systèmes en utilisant des connecteurs.

- => Ingérer des bases de données volumineuses, des données de monitoring dans des topics avec des latences minimales
- => Exporter des topics vers des supports persistants



Fonctionnalités

- Un cadre commun pour les connecteurs qui standardisent l'intégration
- Mode distribué ou standalone
- Une interface REST permettant de gérer facilement les connecteurs
- Gestion automatique des offsets
- Distribué et scalable : Possibilité de scaler les workers, basé sur la gestion de groupe
- Intégration vers les systèmes de streaming ou batch



Mode standalone

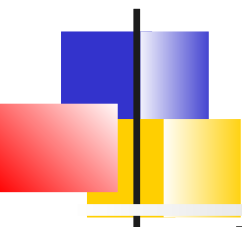
Pour démarrer *Kafka Connect* en mode standalone

```
> bin/connect-standalone.sh config/connect-standalone.properties  
connector1.properties [connector2.properties ...]
```

Le premier paramètre contient la configuration pour le worker. Il inclut entre autres :

- ***bootstrap.servers***
- ***key.converter, value.converter***
- ***offset.storage.file.filename*** : Fichier pour stocker les offsets
- ***plugin.path*** : Une liste de chemins qui contiennent les plugins KafkaConnect (connecteurs, convertisseurs, transformations).

Les autres paramètres définissent les configurations des différents connecteurs



Mode distribué

Le mode distribué gère le scaling dynamique et offre la tolérance aux pannes

Pour démarrer en mode distribué :

```
> bin/connect-distributed.sh  
    config/connect-distributed.properties
```

En mode distribué, *Kafka Connect* stocke les offsets, les configuration et les statuts des taches dans des topics.

- Pour contrôler, le nombre de partitions et le facteur de réplication utilisés, il est recommandé de créer manuellement ces topics



Configurations supplémentaires en mode distribué

group.id (*connect-cluster* par défaut) : nom unique pour former le groupe

config.storage.topic (*connect-configs* par défaut) : *topic* utilisé pour stocker la configuration.

Doit s'agir d'une rubrique à partition unique, hautement répliquée et compactée.

offset.storage.topic (*connect-offsets* par défaut) : *topic* utilisé pour stocker les offsets;

Doit avoir de nombreuses partitions, être répliquée et être configurée pour le compactage

status.storage.topic (*connect-status* par défaut) : *topic* utilisé pour stocker les états;

Peut avoir plusieurs partitions et doit être répliquée et configurée pour le compactage



Configuration des connecteurs

Configuration :

- En mode standalone, via un fichier *.properties*
- En mode distribué, via une API Rest .

Les valeurs sont très dépendantes du type de connecteur.

Comme valeur communes, on peut citer :

- ***name*** : Nom unique.
- ***connector.class*** : Classe Java du connecteur
- ***tasks.max*** : Maximum de tâches créés pour le connecteur.
(degré de parallélisme)
- ***key.converter, value.converter***
- ***topics, topics.regex, topic.prefix*** : Liste, expression régulière ou gabarit spécifiant les topics



Connecteurs

Confluent offre de nombreux connecteurs¹ :

- Active MQ Source Connector
- Amazon S3 Sink Connector
- Elasticsearch Sink Connector
- FileStream Connector (Development and Testing)
- IBM MQ Source Connector
- JDBC Connector (Source et Sink)
- JMS Source Connector
- ...

De nombreux éditeurs fournissent également leur connecteurs Kafka (Amazon, Azure, Google, Salesforce, TIBCO, MongoDB, ...)

1. <https://www.confluent.io/fr-fr/product/connectors/>



Exemple Connecteur JDBC

```
name=mysql-whitelist-timestamp-source
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
tasks.max=10

connection.url=jdbc:mysql://mysql.example.com:3306/my_database?
  user=alice&password=secret
table.whitelist=users,products,transactions

# Pour détecter les nouveaux enregistrements
mode=timestamp+incrementing
timestamp.column.name=modified
incrementing.column.name=id

topic.prefix=mysql-
```



Transformations

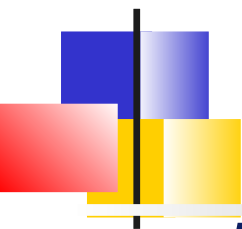
Une chaîne de transformation, s'appuyant sur des ***transformers*** prédéfinis, est spécifiée dans la configuration d'un connecteur.

- ***transforms*** : Liste d'aliases spécifiant la séquence des transformations
- ***transforms.\$alias.type*** : La classe utilisée pour la transformation.
- ***transforms.\$alias.\$transformationSpecificConfig*** : Propriété spécifique d'un *Transformer*



Exemple de configuration

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
# Un sink défini un topic de destination
topic=connect-test
# 2 transformers nommés MakeMap et InsertSource
transforms=MakeMap, InsertSource
# La ligne du fichier devient le champ line
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
# Le champ data-source est ajouté avec une valeur statique
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```



Transformers disponibles

HoistField : Encapsule l'événement entier dans un champ unique de type Struct ou Map

InsertField : Ajout d'un champ avec des données statiques ou des méta-données de l'enregistrement

ReplaceField : Filtrer ou renommer des champs

MaskField : Remplace le champ avec une valeur nulle

ValueToKey : Échange clé/valeur

ExtractField : Construit le résultat à partir de l'extraction d'un champ spécifique

SetSchemaMetadata : Modifie le nom du schéma ou la version

TimestampRouter : Route le message en fonction du timestamp

RegexRouter : Modifie le nom du topic le destination via une *regexp*



API Rest

L'adresse d'écoute de l'API REST peut être configuré via la propriété listeners

`listeners=http://localhost:8080,https://localhost:8443`

Ces listeners sont également utilisés par la communication intra-cluster

- Pour utiliser d'autres Ips pour le cluster, positionner :

`rest.advertised.listener`



API

GET /connectors : Liste des connecteurs actifs

POST /connectors : Création d'un nouveau connecteur

GET /connectors/{name} : Information sur un connecteur (config et statuts et tasks)

PUT /connectors/{name}/config : Mise à jour de la configuration

GET /connectors/{name}/tasks/{taskid}/status : Statut d'une tâche

PUT /connectors/{name}/pause : Mettre en pause le connecteur

PUT /connectors/{name}/resume : Réactiver un connecteur

POST /connectors/{name}/restart : Redémarrage après un plantage

DELETE /connectors/{name} : Supprimer un connecteur



APIs

Producer API
Consumer API
Frameworks
Connect API

Admin et Stream API



Introduction

Kafka propose 2 autres APIs :

- ***Admin API*¹** : Client d'administration permettant de gérer et inspecter les topics, brokers, configurations et ACLs
- ***Streams API*²** : Librairie cliente pour des micro-services dont les entrées/sorties sont des *topics* Kafka

1. API existante dans les autres langages chez Confluent
2. Équivalent Python : Faust



Exemple Admin : Lister les configurations

```
public class ListingConfigs {  
  
    public static void main(String[] args) throws ExecutionException,  
        InterruptedException {  
        Properties config = new Properties();  
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");  
        AdminClient admin = AdminClient.create(config);  
        for (Node node : admin.describeCluster().nodes().get()) {  
            System.out.println("-- node: " + node.id() + " --");  
            ConfigResource cr = new ConfigResource(ConfigResource.Type.BROKER, "0");  
            DescribeConfigsResult dcr = admin.describeConfigs(Collections.singleton(cr));  
            dcr.all().get().forEach((k, c) -> {  
                c.entries()  
                    .forEach(configEntry -> {  
System.out.println(configEntry.name() + "= " + configEntry.value());  
                });  
            });  
        }  
    }  
}
```



Exemple Admin

Créer un topic

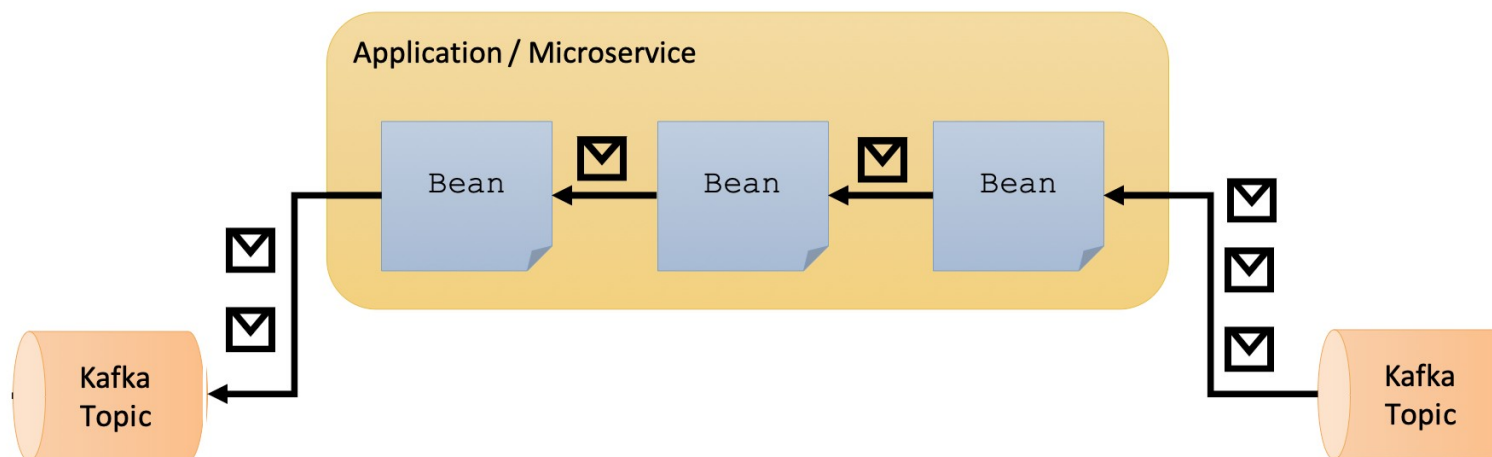
```
public class CreateTopic {
    public static void main(String[] args) throws ExecutionException,
        InterruptedException {
        Properties config = new Properties();
        config.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        AdminClient admin = AdminClient.create(config);
        //Créer un nouveau topics
        System.out.println("-- creating --");
        NewTopic newTopic = new NewTopic("my-new-topic", 1, (short) 1);
        admin.createTopics(Collections.singleton(newTopic));

        //lister
        System.out.println("-- listing --");
        admin.listTopics().names().get().forEach(System.out::println);
    }
}
```



Kafka Streams

Kafka Streams API est une bibliothèque cliente pour la création d'applications et de micro-services, où les données d'entrée et de sortie sont stockées dans des topics Kafka





Apports de *KafkaStream*

- Librairie simple et légère, facilement intégrable, seule dépendance celle d'Apache Kafka.
- Abstractions Kstream et KTable permettant la transformation de flux d'évènements infinis, des jointures et des agrégations
- Peut garantir que chaque évènement soit traité une et une seule fois, même en cas de défaillance.
- Temps de latence des traitements en millisecondes,
- Supporte des opérations de fenêtrage temporel avec l'arrivée des événements dans le désordre.
- Offre les primitives de traitement de flux nécessaires sous forme de DSL haut niveau ou d'une API bas niveau.



Définitions

Un ***KStream*** est une abstraction représentant un ensemble de données illimité, c'est à dire infini et sans cesse croissant car de nouveaux enregistrements continuent d'arriver

D'autres attributs caractérisent ce flux d'événements :

- Les événements sont ordonnés
- Les événements sont immuables
- On peut rejouer un flux d'événements



KStream et KTable

KafkaStream fournit également l'abstraction ***KTable*** qui représente un ensemble de fait qui évoluent.

- Une *KTable* peut être construit à partir d'un *Kstream*. Seule la dernière valeur ou une agrégation d'une clé donnée est conservée
- Un *KStream* peut effectuer des jointures sur une *KTable* et produit alors un *KStream* enrichi
- Une *KTable* peut être transformé en *Kstream* contenant les événements d'update



Agrégation, Jointure, fenêtrage

Une opération **d'agrégation** prend un flux d'entrée ou une table et génère une nouvelle table en combinant plusieurs enregistrements d'entrée en un seul enregistrement de sortie.

– Exemples : Somme, moyenne, ...

Une opération **de jointure** prend plusieurs flux d'entrée et fournit un flux de sortie

Le **fenêtrage** permet de contrôler comment regrouper des enregistrements qui ont une même clé pour des opérations d'agrégations ou des jointures.



State store

Certaines applications (stateful) nécessitent de conserver un état pour grouper, joindre, agréger

Kafka Streams fournit des **State stores** qui permettent aux applications de stocker

On peut y définir des *interactive queries* permettant un accès en lecture à ces données données.

Tout cela dans un contexte de tolérance aux pannes



ksqlDB

ksqlDB fournit donc une couche d'abstraction SQL permettant de manipuler les *KTable* de *KafkaStream*

Via le modèle SQL, il est alors possible de faire tout ce dont on a besoin avec les données en mouvement : acquisition, traitement et interrogation des données.



Application KafkaStream

Une application KafkaStream définit sa logique de traitement à travers une ou plusieurs **topologies de processeurs**, i.e. un graphe de processeurs de flux connectés

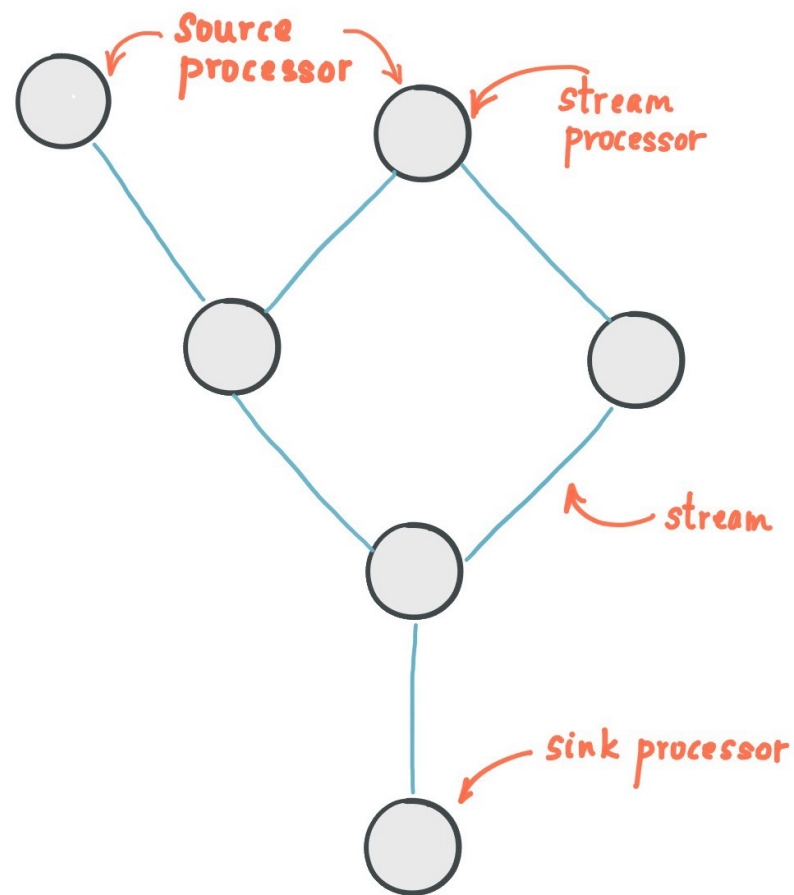
Un processeur représente une étape de traitement qui prend en entrée un événement, le transforme puis produit un ou plusieurs événements

Certains processeurs :

- n'ont pas de connexions entrantes : **Source**
- d'autres n'ont pas de connexions sortantes : **Sink**
- Les autres ont des connexions entrantes et sortantes

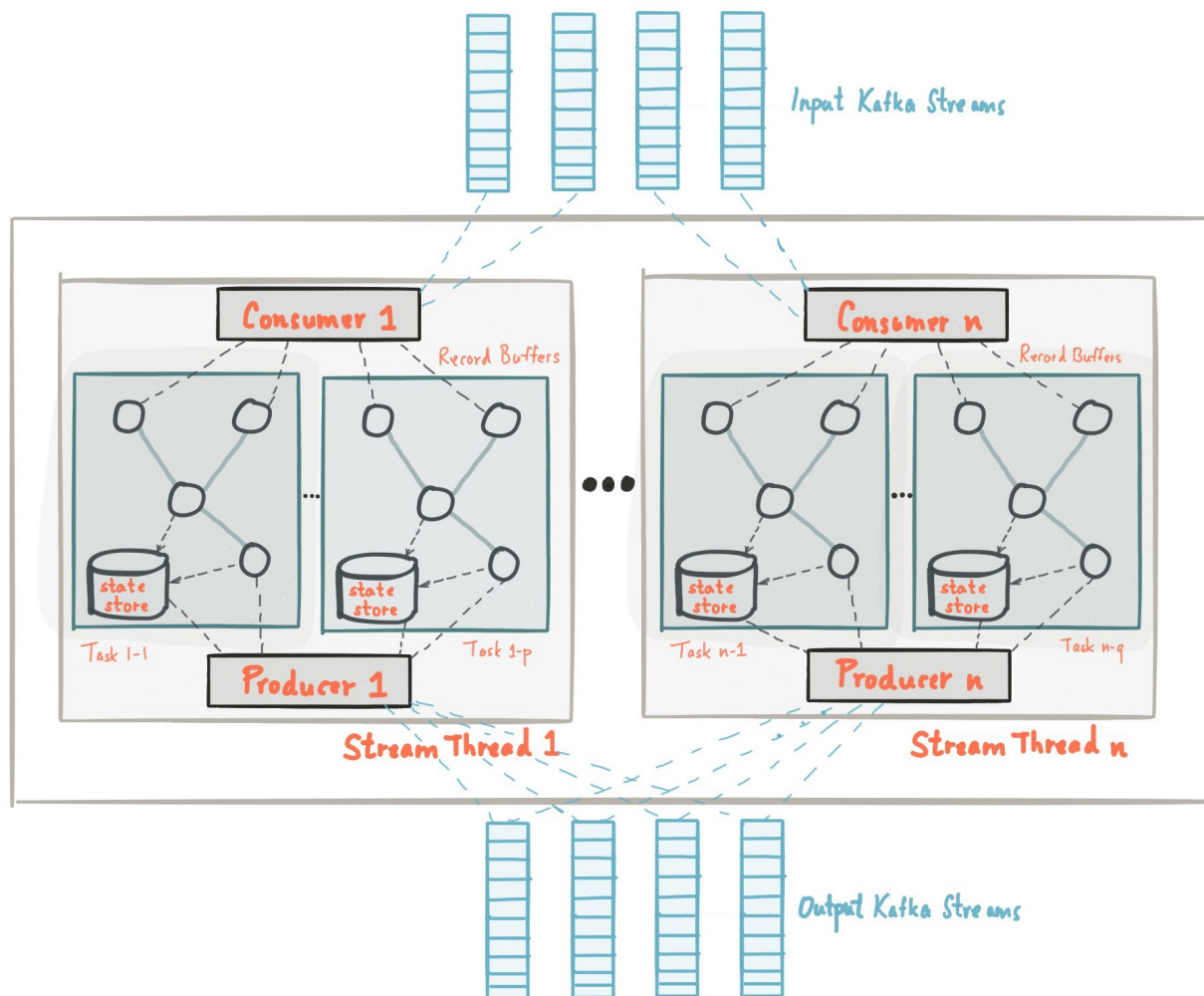
La topologie peut être spécifiée programmatiquement ou par un DSL

Topologie processeurs



PROCESSOR TOPOLOGY

Architecture et scalabilité





Exemple

// Propriétés : ID, BOOTSTRAP, Serialiseur/Désérialiseur

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
```

// Création d'une topologie de processeurs

```
final StreamsBuilder builder = new StreamsBuilder();
builder.<String, String>stream("streams-plaintext-input")
    .flatMapValues(value -> Arrays.asList(value.split("\\W+")))
    .to("streams-linesplit-output");
```

```
final Topology topology = builder.build();
```

// Instanciation du Stream à partir d'une topologie et des propriétés

```
final KafkaStreams streams = new KafkaStreams(topology, props);
```



Exemple (2)

```
final CountDownLatch latch = new CountDownLatch(1);

// attach shutdown handler to catch control-c
Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});

// Démarrage du stream
try {
    streams.start();
    latch.await();
} catch (Throwable e) {
    System.exit(1);
}
System.exit(0);
```




Garanties Kafka

Mécanismes de réplication

At Most One, At Least One

Exactly Once

Débit, latence, durabilité



Introduction

Différents brokers participent à la gestion distribuée et répliquée d'une réplique.

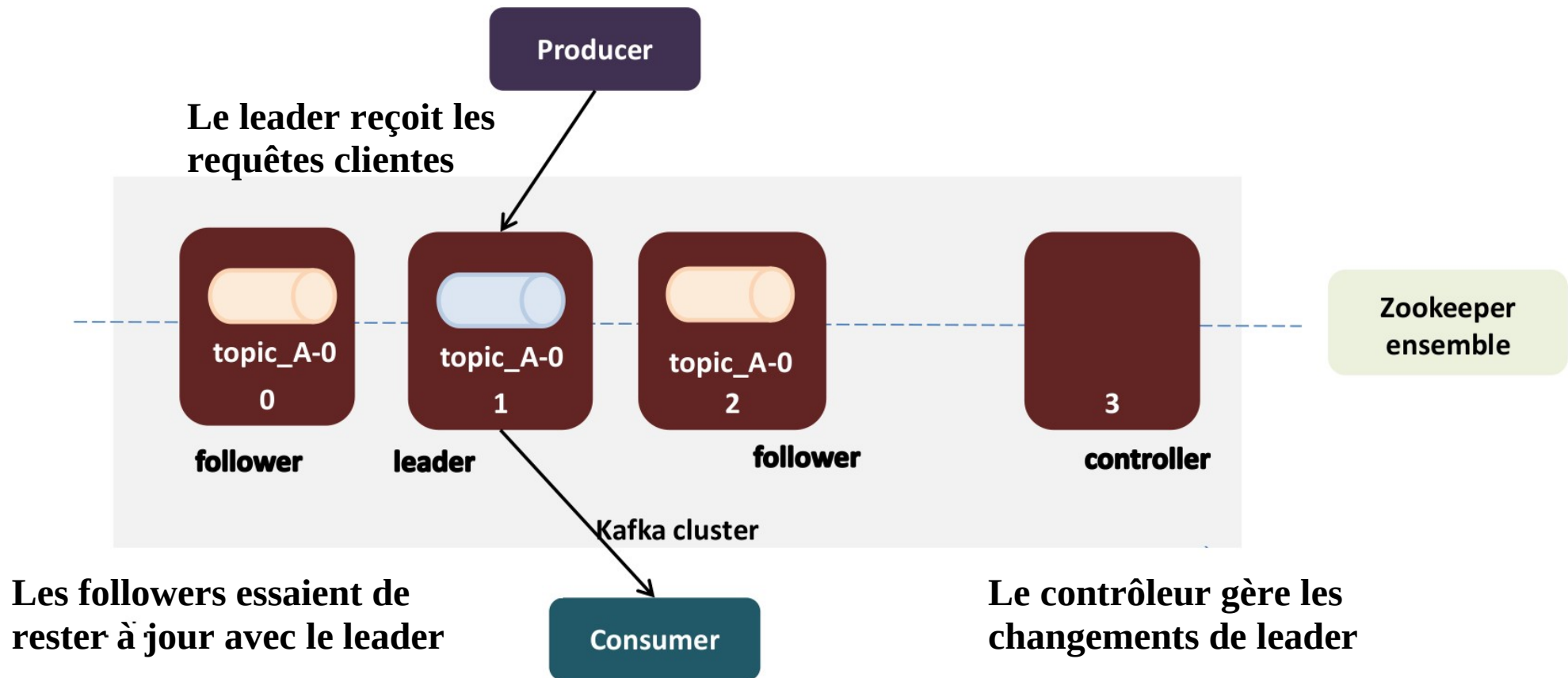
Pour chaque partition d'un topic :

- **1 broker leader** : Détient la partition leader responsable de la validation des écritures
- **N followers** : Suivent les écritures effectuées par le leader. Un décalage est permis mais si il n'arrive plus à suivre la cadence d'écriture, ils sont éliminés

Au niveau du cluster :

- Un **contrôleur** est responsable des élections de leader

Rôle des brokers gérant un topic





Garanties Kafka

Garanties offertes par Kafka grâce à la réplication :

- Garantie de l'ordre à l'intérieur d'une partition.
=> Le consommateur d'une partition lit dans l'ordre d'écriture des messages
- Les messages produits sont considérés “committed” lorsqu'ils sont écrits sur la partition leader et qu'ils ont atteint le minimum de réplication, i.e. minimum de répliques synchronisées
- Les messages validés sont disponibles tant qu'au moins une réplique reste en vie.
- Les consommateurs ne peuvent lire que les messages validés.



Synchronisation des répliques

Contrôlé par la propriété :

replica.lag.time.max.ms (défaut 30 000)

Si pendant ce délai, le follower

- N'envoie pas de requêtes fetch
- N'atteint pas l'offset de fin du leader

Alors, le follower est considéré comme désynchronisé

- Il est supprimé de la liste des **ISR (In Sync Replica)**
- Il peut ensuite rattraper son retard et être réintégré aux ISRs



Rôles des brokers vis à vis de l'ISR

Leader

- Une réplique élue pour chaque partition
- Qui reçoit toutes les requêtes des producteurs et consommateurs
- Gère la liste de ses ISR

Followers

- Essaie de rester à jour avec le leader
- Si le leader tombe, un follower devient le nouveau leader

Controller

- Responsable pour élire les leaders de partition et diffuser l'information au leader et aux ISR
- Persiste le nouveau leader et l'ISR vers *Zookeeper* ou les disques des contrôleurs *Kraft*



min.insync.replica

La propriété ***min.insync.replica***, spécifiée au niveau cluster ou topic, indique le minimum de répliques de l'ISR (incluant le leader) qui doivent avoir écrit un message afin que celui-ci soit considéré comme *committed/validé*

- A la réception d'un message, le leader vérifie si il y a assez d'ISR pour écrire le message, sinon il envoie une *NotEnoughReplicasException*
- Lorsque le message est répliqué par *min.insync.replica* répliques, le leader envoie un acquittement au client.



Conséquences

Un réplique synchronisée légèrement en retard peut ralentir l'acquittement du message ; ce qui peut ralentir le débit.

Une réplique désynchronisée n'a plus d'impact sur les performances mais augmente le risque d'un temps d'arrêt ou d'une perte de données.



Rejet de demande d'émission

Si le **nombre de ISR** $< \text{min.insync.replicas}$:

- Kafka empêche l'acquittement du message.
En fonction de la configuration du producteur, celui-ci peut être bloqué.

Si le **nombre de répliques disponible** $< \text{min.insync.replicas}$

- Mauvaise configuration, Kafka bloque les émissions de message

En conséquences :

n répliques

=> tolère $n-1$ failures pour que la partition soit disponible

n répliques, $\text{min.insync.replicas} = m$

=> Tolère $n-m$ failures pour accepter les envois



Configuration

2 principales configurations affectent la fiabilité et les compromis liés :

- ***default.replication.factor (au niveau cluster)*** et ***replication.factor (au niveau topic)***
Compromis entre disponibilité (valeurs hautes) et matériel (valeur basse)
Valeur classique 3
- ***min.insync.replicas*** (défaut 1, au niveau cluster ou topic)
Le minimum de répliques qui doivent acquitter pour valider un message
Compromis entre perte de données (les répliques synchronisées tombent) et ralentissement
Valeur classique 2/3



Garanties Kafka

Mécanismes de réplication

At Most Once, At Least Once

Exactly Once

Débit, latence, durabilité



Côté producteur

Du côté producteur, 2 premiers facteurs influencent la fiabilité de l'émission

- La configuration des **acks** en fonction du *min.insync.replica* du topic
3 valeurs possibles : *0,1,all*
- Les gestion des **erreurs** dans la configuration et dans le code



acks=0

acks=0 : Le producteur considère le message écrit au moment où il l'a envoyé sans attendre l'acquittement du broker
=> Perte de message potentielle (*At Most Once*)





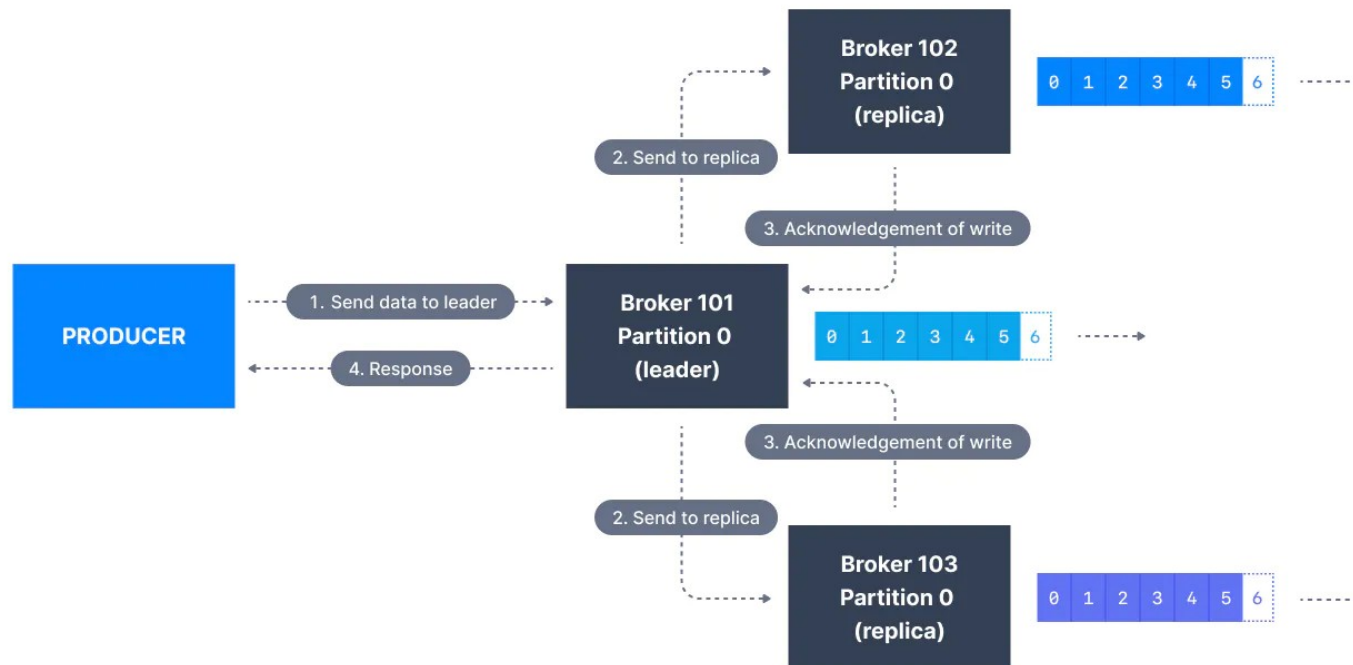
acks=1

acks=1 : Le producteur considère le message écrit lorsque le leader a acquitté l'écriture
=> Si le leader s'arrête et que les répliques n'ont pas eu le temps d'écrire le message, perte de données

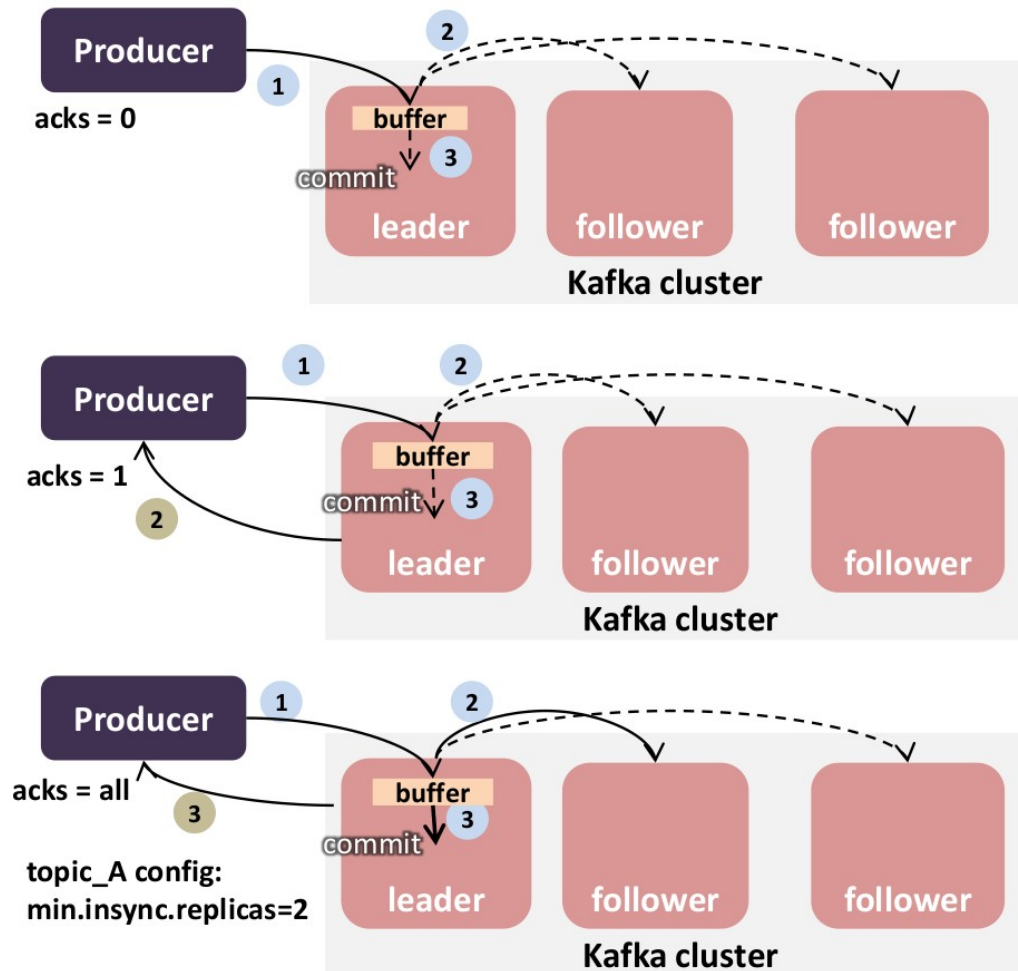


acks=all

acks=all : Le producteur considère le message comme écrit lorsque il a été répliqué par *min.insync.replica* des ISR.
=> Assure le maximum de durabilité
(Nécessaire pour *At Least Once* et *Exactly Once*)



Acquittement et durabilité



Latency

Data loss risk





Gestion des erreurs

2 types d'erreur pour le producteur:

- les erreurs que les producteurs gèrent automatiquement .
Ce sont les erreurs ré-essayable (ex :
LEADER_NOT_AVAILABLE,
NOT_ENOUGH_REPLICA)
Nombre d'essai configurable via **retries**.
=> Attention, peut générer des doublons
- les erreurs qui doivent être traitées par le code. (ex : INVALID_CONFIG,
SERIALIZATION_EXCEPTION)



Rebalancing

En cas de crash ou d'ajout d'un consommateur, Kafka réaffecte les partitions.

Lors d'une réaffectation, un consommateur récupère l'offset de lecture auprès de Kafka

Si l'offset n'est pas synchronisé avec les traitements effectués, cela peut :

- Générer des traitements en doublon
- Perdre des traitements de message

Il est possible de s'abonner aux événements de rebalancing



Côté consommateur

Du point de vue de la fiabilité, la seule chose que les consommateurs ont à faire est de s'assurer qu'ils gardent une trace des offsets qu'ils ont traités en cas de rebalancing.

Pour cela, ils commettent leur offset auprès du cluster Kafka qui stocke les informations dans le topic

_consumer_offsets

=> La seule façon de perdre des messages est alors de committer des offsets de messages lus mais pas encore traités



Configuration

3 propriétés de configuration sont importantes pour la fiabilité du consommateur :

auto.offset.reset : Contrôle le comportement du consommateur lorsqu'aucun offset est commité ou lorsqu'il demande un offset qui n'existe pas

- ***earliest*** : Le consommateur repart au début, garantie une perte minimale de message mais peut générer beaucoup de traitements en doublon
- ***latest*** : Minimise les doublons mais risque de louper des messages

enable.auto.commit : Commit manuel ou non

- Automatique : Si tout le traitement est effectué dans la boucle de poll. Garantie que les offsets commités ont été traités mais pas de contrôle sur le nombre de doublons
Attention si le traitement est fait dans une thread différente de la boucle de poll

auto.commit.interval.ms : Valable en mode automatique

- Diminuer l'intervalle ajoute de la surcharge mais réduit le risque de doublon lors d'un arrêt de consommateur

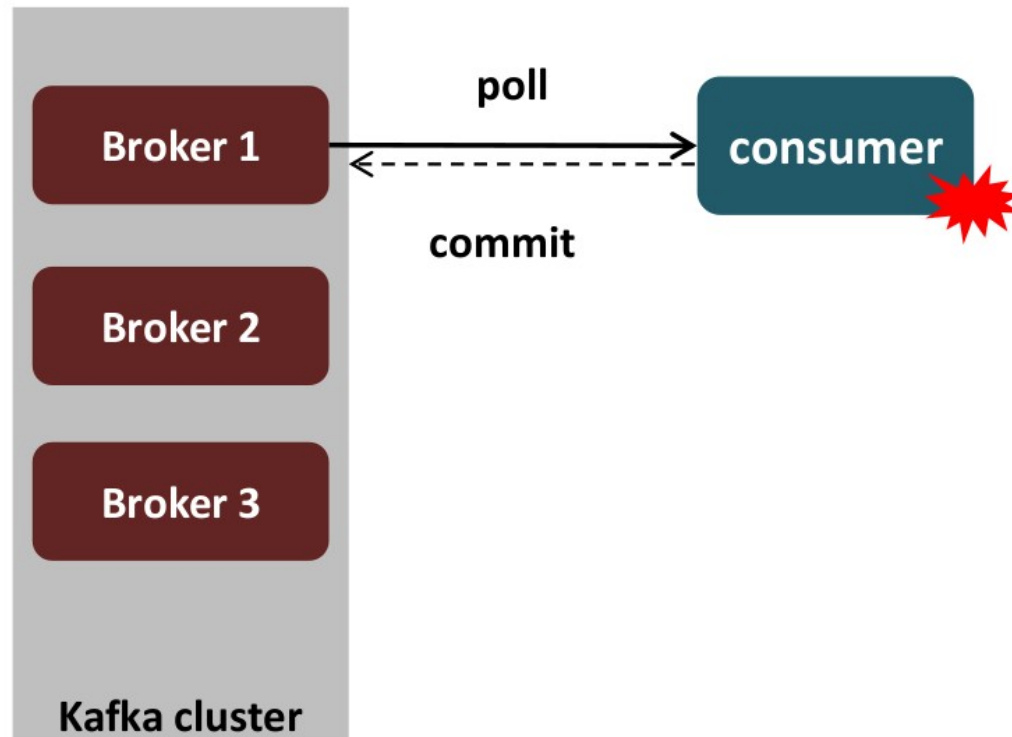


Auto commit

En mode automatique, le commit des offset est effectué lors de l'appel à *poll()*.

Si lors d'un appel à poll, *auto.commit.interval.ms* a été atteint les offsets du dernier poll sont committés

Réception : At Most Once



- L'offset est commité
- Traitement d'un ratio de message puis plantage



Configuration *At Most Once*

- *enable.auto.commit = true.*
- Traitement asynchrone dans la boucle de poll

OU

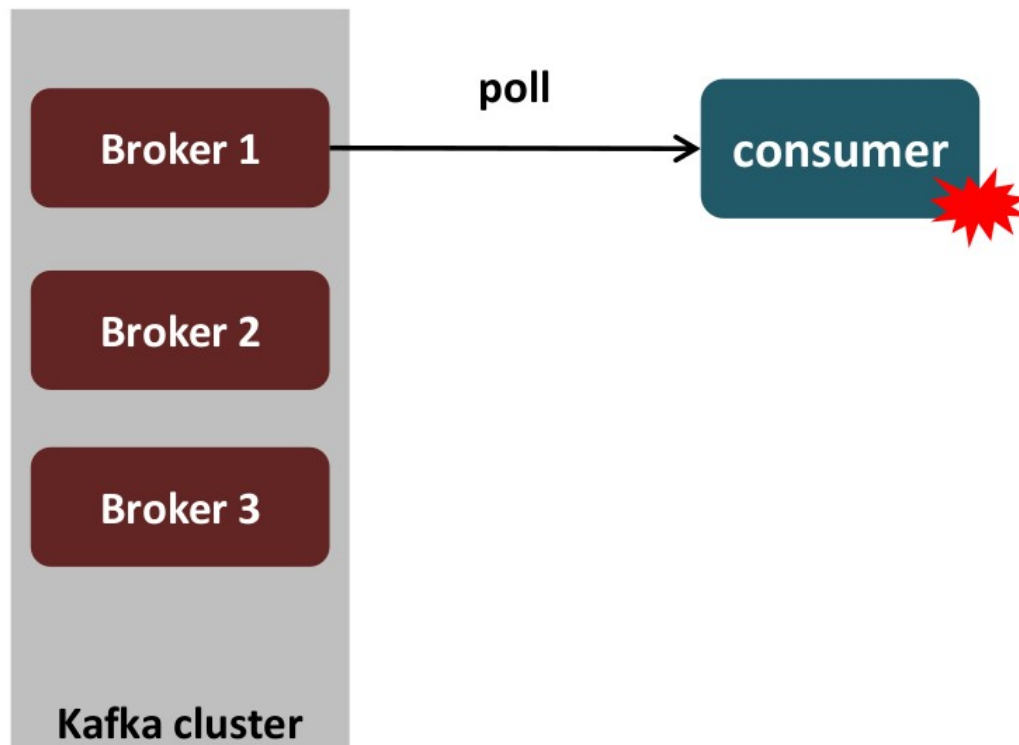
Commit manuel avant le traitement des messages

Exemple : Auto-commit et traitement asynchrone

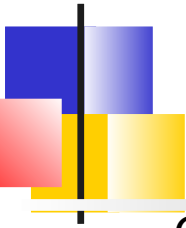
```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    long lastOffset = 0;  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.printf("\n\roffset = %d, key = %s, value = %s", record.offset(), record.key(),  
            record.value());  
        lastOffset = record.offset();  
    }  
    System.out.println("lastOffset read: " + lastOffset);  
    // Traitement asynchrone  
    process();  
}
```



Réception : *At Least Once*



- Traitement d'un ratio de message puis plantage
- L'offset n'est pas commité



Configuration *At Least Once*

Configuration par défaut et traitement synchrone dans la boucle de poll

Ou

enable.auto.commit à false ET Commit explicite après traitement via
consumer.commitSync();

Exemple : Commit manuel après traitement

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    long lastOffset = 0;  
  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.printf("\n\roffset = %d, key = %s, value = %s", record.offset(),  
            record.key(), record.value());  
        lastOffset = record.offset();  
    }  
    System.out.println("lastOffset read: " + lastOffset);  
    process();  
    consumer.commitSync();  
}
```



Validation de la fiabilité

Il est nécessaire de valider si sa configuration remplit les garanties voulues.

La validation s'effectue à 3 niveaux :

- Validation de la configuration
- Validation de l'application
- Surveillance



Validation de la configuration

Kafka fournit 2 outils permettant de tester la configuration en isolation de l'application :

- ***kafka-verifiable-producer.sh*** produit une séquence de messages contenant des nombres en séquence. On peut configurer le nombre de ack, de retry et les cadences des messages
- ***kafka-verifiable-consumer.sh*** consomme les messages et les affiche dans l'ordre de consommation. Il affiche également des informations sur les commit et les rééquilibrage

On peut alors exécuter ces commandes pendant différents scénarios de test : Élection de leader, de contrôleur, redémarrage des brokers, ...



Monitoring

Kafka propose des métriques JMX.

Pour la fiabilité du producteur :

- ***error-rate*** et ***retry-rate*** permettent de déceler des anomalies systèmes.
- Également voir les traces du producteur (WARN)

Pour la fiabilité du consommateur :

- ***consumer lag*** : Indique le décalage des consommateurs



Garanties Kafka

Mécanismes de réplication
At Most Once, At Least Once

Exactly Once

Débit, latence, durabilité

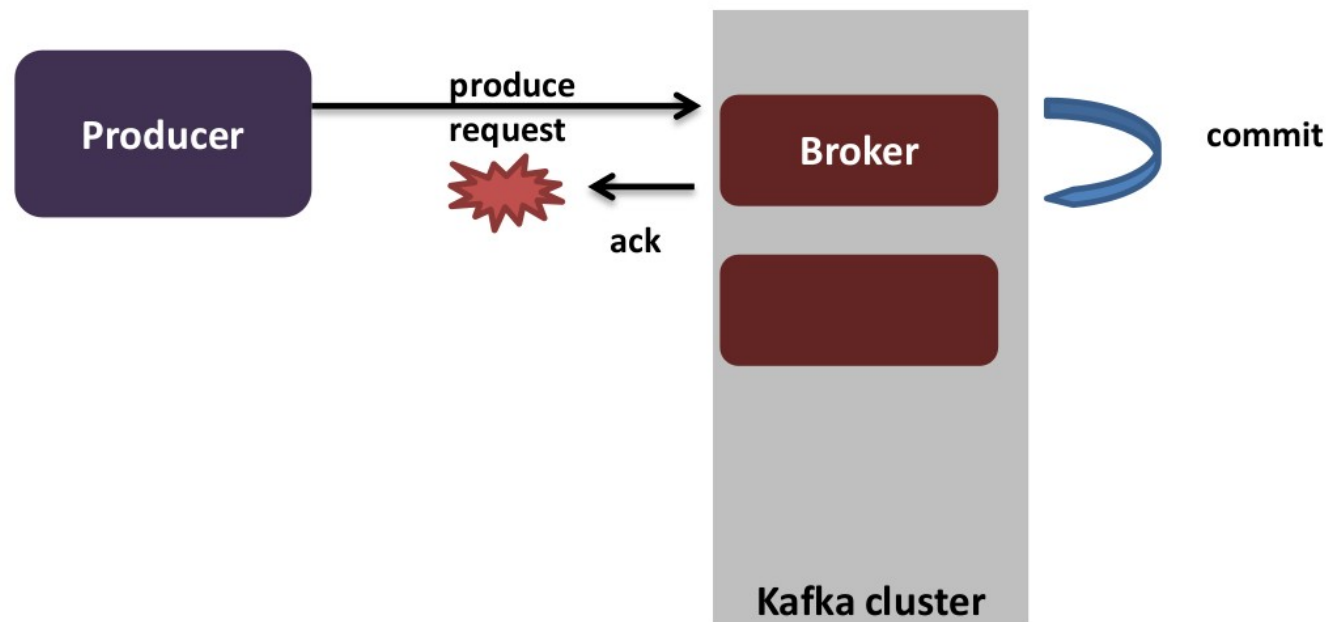


Introduction

La sémantique *Exactly Once* est basée sur *At least Once* et empêche les messages en double en cours de traitement par les applications clientes

Pris en charge par les APIs *Kafka Producer, Consumer et Streams*

Producteur idempotent



Le producteur ajoute une
nombre séquentiel et un ID
de producteur

Le broker détecte le
doublon
=> Il envoie un ack sans le
commit



Configuration

enable.idempotence

L'idempotence nécessite les configurations suivantes :

- *max.in.flight.requests.per.connection* ≤ 5
- *retries* > 0
- *acks* = *all*

Si des valeurs incompatibles sont configurées, une *ConfigException* est lancée



Consommateur

Du côté du consommateur, traiter une et une seule fois les messages consistent à :

- gérer manuellement les offsets des partitions dans un support de persistance transactionnel et partagé par tous les consommateurs et gérer les rééquilibrages
- Si le traitement consiste à produire un message vers le même cluster Kafka, on peut utiliser les transactions¹.

1. C'est le cas de KafkaStream



Exemple

enable.auto.commit=false

```
while (true) {  
    ConsumerRecords<String, String> records = consumer.poll(100);  
    for (ConsumerRecord<String, String> record : records) {  
        System.out.printf("offset = %d, key = %s, value = %s\n",  
            record.offset(),  
            record.key(), record.value());  
  
        // Sauvegarder l'offset traité .  
        offsetManager.saveOffsetInExternalStore(record.topic(),  
            record.partition(), record.offset());  
    }  
}
```



Example (2)

```
public class MyConsumerRebalancerListener implements
    org.apache.kafka.clients.consumer.ConsumerRebalanceListener {

    private OffsetManager offsetManager = new OffsetManager("storage2");
    private Consumer<String, String> consumer;

    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            offsetManager.saveOffsetInExternalStore(partition.topic(),
                partition.partition(), consumer.position(partition));
        }
    }

    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        for (TopicPartition partition : partitions) {
            consumer.seek(partition,
                offsetManager.readOffsetFromExternalStore(partition.topic(),
                partition.partition()));
        }
    }
}
```



Transaction

Pour utiliser le mode transactionnel et l'API correspondante, il faut positionner la propriété ***transactional.id*** à une chaîne de caractères unique par producteur

- Le mode transactionnel inclut le mode idempotent
- Les topics impliqués doivent être configurés avec *replication.factor* ≥ 3 et *min.insync.replicas* = 2
- Les consommateurs doivent avoir la propriété *isolation.level* à *read committed*
- L'API *send()* devient bloquante



Transaction

Permet la production atomique de messages sur plusieurs partitions

Les producteurs produisent l'ensemble des messages ou aucun


```
// initTransaction démarre une transaction avec l'id.  
// Si une transaction existe avec le même id, les messages sont roll-backés  
producer.initTransactions();  
try {  
    producer.beginTransaction();  
    for (int i = 0; i < 100; ++i) {  
        ProducerRecord record = new ProducerRecord("topic_1", null, i);  
        producer.send(record);  
    }  
    producer.commitTransaction();  
} catch (ProducerFencedException e) { producer.close(); } catch (KafkaException  
e) { producer.abortTransaction(); }
```



Configuration du consommateur

Afin de lire les messages transactionnels validés :

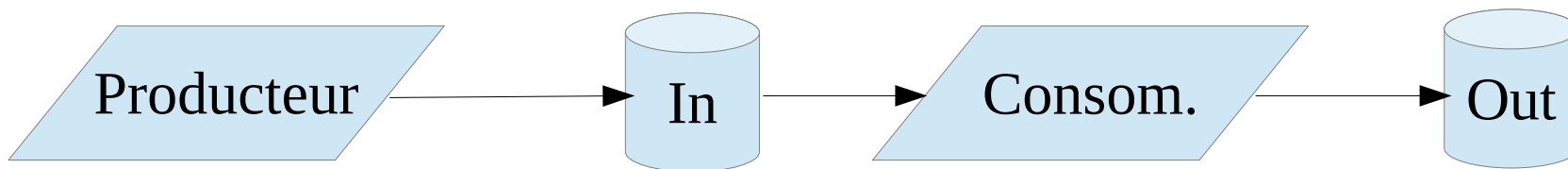
- ***isolation.level=read_committed***
(Défaut: *read_uncommitted*)
 - *read_committed*: Messages (transactionnels ou non) validés
 - *read_uncommitted*: Tous les messages (même les messages transactionnels non validés)



Exactly Once pour le transfert de messages entre topics

Lorsque un consommateur produit vers un autre topic, on peut utiliser les transactions afin d'écrire l'offset vers Kafka dans la même transaction que le topic de sortie

Si la transaction est abandonnée, la position du consommateur reviendra à son ancienne valeur et les données produites sur les topics de sortie ne seront pas visibles pour les autres consommateurs, en fonction de leur "niveau d'isolement".





Producteur

```
producer.initTransactions();
```

```
try {  
    producer.beginTransaction();  
    Stream.of(DATA_MESSAGE_1, DATA_MESSAGE_2)  
        .forEach(s -> producer.send(new ProducerRecord<String, String>("input", null, s)));  
    producer.commitTransaction();  
} catch (KafkaException e) {  
    producer.abortTransaction();  
}
```




Consommateur

```
while (true) {  
  
    ConsumerRecords<String, String> records = consumer.poll(ofSeconds(60));  
  
    // Transform message  
    Map<String, Integer> wordCountMap = ...  
  
    producer.beginTransaction();  
  
    wordCountMap.forEach((key, value) ->  
        producer.send(new ProducerRecord<String, String>(OUTPUT_TOPIC, key, value.toString())));  
  
    Map<TopicPartition, OffsetAndMetadata> offsetsToCommit = new HashMap<>();  
  
    // Retrieve offsets for each partition  
    for (TopicPartition partition : records.partitions()) {  
        List<ConsumerRecord<String, String>> partitionedRecords = records.records(partition);  
        long offset = partitionedRecords.get(partitionedRecords.size() - 1).offset();  
        offsetsToCommit.put(partition, new OffsetAndMetadata(offset + 1));  
    }  
    // Commit Offset for consumer associated with the commit of the transaction  
    producer.sendOffsetsToTransaction(offsetsToCommit, CONSUMER_GROUP_ID);  
    producer.commitTransaction();  
}
```



Garanties Kafka

Mécanismes de réplication
At Most Once, At Least Once
Exactly Once
Débit, latence, durabilité



Configuration pour favoriser le débit

Côté producteur :

Augmenter

- *batch.size*: ex : 100000 (défaut 16384)
- *linger.ms*: ex : 100 (default 0)
- *buffer.memory*: si il y a beaucoup de partitions (défaut 32 Mb)

Puis

- *compression.type*=lz4 (défaut none)
- *acks*=1 (défaut all)

Côté consommateur

Augmenter

- *fetch.min.bytes*: ex : 100000 (défaut 1)



Configuration pour favoriser la latence

Cluster

Si followers pas suffisamment rapide, augmenter leur nombre

- *num.replica.fetchers* : (défaut 1)

Côté producteur

- *linger.ms*: 0
- *compression.type*=none
- *acks*=1

Côté consommateur

- *fetch.min.bytes*: 1



Configuration pour la durabilité

Cluster

- *replication.factor*: 3
- *min.insync.replicas*: 2 (défaut 1)
- *unclean.leader.election.enable* : false (défaut false)
- *broker.rack*: rack du broker (défaut null)

Producteur

- *acks:all* (défaut all)
- *enable.idempotence:true* (défaut false)
- *max.in.flight.requests.per.connection*: ≤ 5

Consommateur

- *isolation.level: read_committed*



Administration

Gestion des topics

Stockage et rétention des partitions

Gestion du cluster

Sécurité

Dimensionnement

Surveillance



Introduction

La distribution Kafka offre de nombreux scripts CLI pour les différentes tâches d'administration.

Certains outils UI tentent de rassembler tous ses CLI en une interface utilisateur :

- UI for Apache Kafka
<https://github.com/provectus/kafka-ui>
- CMAK
<https://github.com/yahoo/CMAK>
- Confluent CC
<https://github.com/confluentinc>
- Conduktor
<https://github.com/conduktor>
- ...



kafka-topics.sh

L'utilitaire ***kafka-topics.sh*** permet de créer, supprimer, modifier et visualiser en détail.

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic position
Topic: position PartitionCount: 3    ReplicationFactor: 2    Configs:
Topic: position Partition: 0      Leader: 3    Replicas: 3,1    Isr: 3,1
Topic: position Partition: 1      Leader: 1    Replicas: 1,2    Isr: 1,2
Topic: position Partition: 2      Leader: 2    Replicas: 2,3    Isr: 2,3
```




Détection de problème

Certaines options utilisées conjointement avec *--describe* permet de mettre en lumière des problèmes

- ***--unavailable-partitions*** : Seulement les partitions qui n'ont plus de leader
- ***--under-min-isr-partitions*** : Seulement les partitions dont l'ISR est inférieur à une valeur
- ***--under-replicated-partitions*** : Les partitions sous-répliquées



Modification de topics

Il est possible de modifier un *topic* existant.

- Par exemple, modifier le nombre de partitions

```
bin/kafka-topics.sh --zookeeper zk_host:port --alter --  
topic my_topic_name --partitions 40
```

- Ne déplace pas les données sur les partitions existantes
- Peut causer des problèmes pour les consommateurs de données stateful avec des clés

- Ou modifier une configuration

```
bin/kafka-topics.sh --zookeeper zk_host:port --alter --  
topic my_topic_name --config retention.ms=
```



Modification entités

Le script ***kafka-configs.sh*** permet de visualiser, modifier la configuration d'un topic, broker, client

Exemple, modification de topic

```
./kafka-configs.sh --bootstrap-server  
localhost:9092 --entity-type topics --  
entity-name position --alter --add-config  
retention.ms=-1
```



Suppression de Topic

La suppression doit être autorisée sur les brokers

delete.topic.enable=true

Il faut arrêter tous les producteurs/consommateurs avant la suppression

Les offsets des consommateurs ne sont pas supprimés immédiatement



Extension du cluster

Kafka ne déplace pas automatiquement les données vers les Brokers ajoutés au cluster

Il faut manuellement exécuter
kafka-reassign-partitions

- Il n'y a pas d'interruption de service
- Il est impossible d'annuler le processus
- A un temps T, une seule réaffectation est possible



Usage de *kafka-reassign-partition*

kafka-reassign-partition peut être utilisé pour :

- Réduire ou étendre le cluster
- Augmenter le facteur de réplication d'un *topic*
- Résoudre un déséquilibre de stockage entre brokers ou entre répertoire d'un même broker



Étape 1

Génération proposition

```
$ cat topics-to-move.json
{"topics":[{"topic" : "foo1"}, {"topic" : "foo2"}], "version":1}
```

```
$ ./kafka-reassign-partitions.sh --zookeeper localhost:2181 \
--topics-to-move-json-file topics-to-move.json \
--broker-list "0,1,2" --generate
```

Current partition replica assignment

```
{"version" : 1, "partitions" : [
{"topic" : "foo1", "partition" : 0, "replicas": [0,1]},
{"topic" : "foo2", "partition" : 0, "replicas": [1,0]}, ... ]}
```

Proposed partition reassignment configuration

```
{"version" : 1, "partitions" : [
{"topic" : "foo1", "partition" : 0, "replicas": [0,1]},
{"topic" : "foo2", "partition" : 0, "replicas": [1,2]}
```



Etape 2

Exécution

```
$ ./kafka-reassign-partition.sh --zookeeper localhost:2181 \  
--reassignment-json-file expand-cluster-reassignment.json \  
--execute
```

Current partition replica assignment

```
{"version" : 1, "partitions" : [  
{"topic" : "foo1", "partition" : 0, "replicas": [0, 1]},  
{"topic" : "foo2", "partition" : 0, "replicas": [1, 0]}, ... ]}
```

Save this to use as the --reassignment-json-file option during rollback

Successfully started reassignments of partitions

```
{"version" : 1, "partitions" : [  
{"topic" : "foo1", "partition" : 0, "replicas": [0, 1]},  
{"topic" : "foo2", "partition" : 0, "replicas": [1, 2]}
```




Exemple Vérification

```
./kafka-reassign-partition.sh --zookeeper localhost:2181 \  
--reassignment-json-file expand-cluster-reassignment.json \  
--verify
```

Status of partition reassignment :

```
Reassignment of partition [foo1,0] completed successfully  
Reassignment of partition [foo1,1] is in progress  
Reassignment of partition [foo1,2] is in progress  
Reassignment of partition [foo2,0] completed successfully  
Reassignment of partition [foo2,1] completed successfully  
Reassignment of partition [foo2,2] completed successfully
```



Administration

Gestion des topics

Stockage et rétention des partitions

Gestion du cluster

Sécurité

Dimensionnement

Surveillance



Introduction

L'unité de stockage de Kafka est une réplique de partition.

- => Les partitions ne peuvent pas être divisées entre plusieurs brokers ni entre plusieurs disques du même broker

La propriété ***log.dirs*** définit les répertoires de stockage des partitions



Allocation des partitions

A la création des *topics*, Kafka décide comment allouer les partitions sur les brokers

Ses objectifs sont :

- Répartir uniformément les répliques entre les brokers
- S'assurer que chaque réplique d'une partition se trouve sur un broker différent
- Si les brokers ont des informations sur le rack, s'assurer que les répliques sont affectés à des racks différents si possible



Rétention des données

L'administrateur Kafka configure une période de rétention pour chaque topic

- Soit une durée
- Soit un volume

Pour accélérer la purge, Kafka utilise les ***segments***

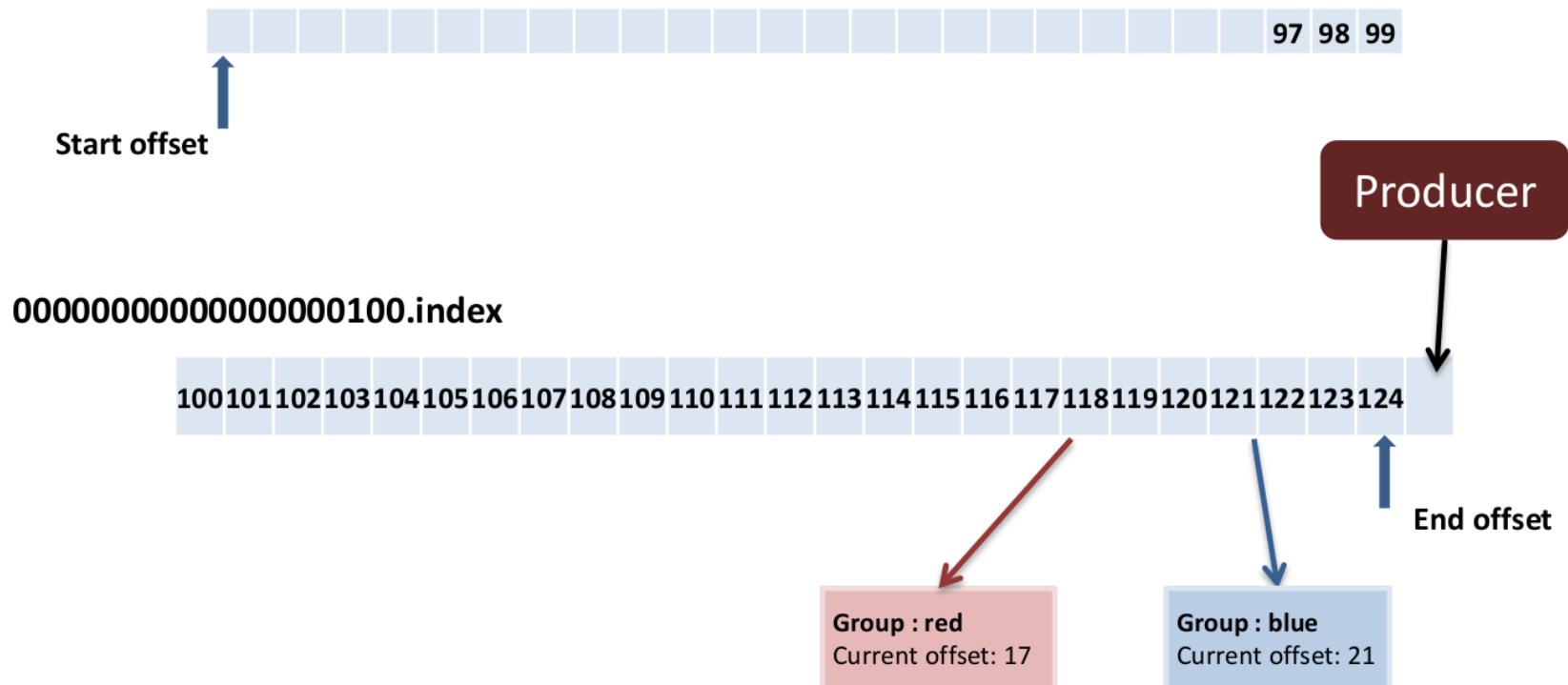
- Les segments sont des fragments de partition au maximum de 1Go et au maximum d'une semaine de données
- Lors de l'écriture d'un segment, lorsque la limite est atteinte, le broker ferme le fichier. Il devient éligible pour la purge
- Il existe donc un seul *segment actif* qui d'ailleurs ne peut pas être purgé

Segments

Log segment

closing segment parameters :

- log.roll.ms (ignored if log.roll.hours is set)
- log.roll.hours (default 168)
- log.segment.bytes (default 1073741824)





Indexation

Kafka permet de récupérer des messages à partir de n'importe quel offset disponible.

Pour accélérer cette opération, Kafka maintient un ***index*** pour chaque partition

- L'index associe les offsets aux segments et aux positions dans le fichier
- Les index sont également divisés en segments
- En cas de suppression, ils peuvent être régénérés automatiquement par Kafka



Principales configurations

log.retention.hours (défaut 168 : 7 jours),

log.retention.minutes (défaut null),

log.retention.ms (défaut null, si -1 infini)

Période de rétention des vieux segment avant de les supprimer

log.retention.bytes (défaut -1)

La taille maximale du log

offsets.retention.minutes (défaut 10080 : 7 jours)

Le nombre de minutes pour conserver l'index d'offset d'un groupe n'ayant plus de consommateurs



Compactage

Kafka autorise également la stratégie de rétention ***compact***, qui ne stocke que la valeur la plus récente pour chaque clé du *topic*.

- Propriétés ***cleanup.policy=compact*** et ***log.cleaner.enabled=true***
- Les événements doivent alors contenir une clé
- Le compactage est effectué par une thread séparé qui périodiquement purge les messages *dirty*



Nettoyage des logs

Propriété `log.cleanup.policy`, 2 stratégies disponible :

- ***delete*** (défaut) :
 - Suppression des vieux segments en fonction de l'âge et la taille du log (partition)
- ***compact***
 - Suppression basée sur les clés des messages quelque soit l'âge et la taille des données

Possibilité de combiner les 2 stratégies (delete and compact)



Stratégie *delete*

La stratégie *delete* s'appuie sur :

- ***log.retention.bytes*** (défaut -1 : infinite)
- ***log.retention.ms*** (défaut null)
- ***log.retention.minutes*** (défaut null)
- ***log.retention.hours*** (défaut 168, 1 semaine)

=> Meilleur contrôle de l'usage disque



Stratégie compact

2 propriétés de configuration importante pour cette stratégie :

- *cleaner.min.compaction.lag.ms* : Le temps minimum qu'un message reste non compacté
- *cleaner.max.compaction.lag.ms* : Le temps maximum qu'un message reste inéligible pour la compactage

Le nettoyeur (*log cleaner*) est implémenté par un pool de threads.

Le pool est configurable :

- *log.cleaner.enable* : doit être activé si stratégie compact
- *Log.cleaner.threads* : Le nombre de threads
- *log.cleaner.backoff.ms* : Le temps de pause lorsqu'il n'y pas de travail (défaut 15 secondes)
-

Lors du nettoyage, le segment actif est fermé (un nouveau segment est créé)

Sauvegarde la dernière valeur (dernière mise à jour) pour chaque clé

=> Consomme CPU et RAM



Exemple *compact*

1	2	3	4	5	6	7
K1	K2	K3	K4	K4	K5	K1
6	3	2	2	1	3	2



2	3	5	6	7
K2	K3	K4	K5	K1
3	2	1	3	2



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



Redémarrage du cluster

Redémarrage progressif, broker par broker

Attendre que l'état se stabilise
(isr=replicas)

Vérification de l'état des topics

```
bin/kafka-topics.sh --zookeeper zk_host:port --  
describe --topic my_topic_name
```

Exemple d'outillage :

<https://github.com/deviceinsight/kafkactl>



Mise à jour du cluster

Avant la mise à niveau:

- Pour toutes les partitions:
liste de répliques = liste ISR

Garanties:

- Pas de temps d'arrêt pour les clients (producteurs et consommateurs)
- Les nouveaux brokers sont compatibles avec les anciens clients Kafka



Étapes de mise à jour

server.properties: définissez la configuration suivante (redémarrage progressif)

- *inter.broker.protocol.version* : version actuelle
- *log.message.format.version* : version actuelle du format de message

Mettre à niveau les brokers un par un (redémarrage)

- Attendre l'état stable (ISR = réplicas)

Mettre à jour en dernier le contrôleur

Mettre à jour *inter.broker.protocol.version* (redémarrage progressif)

Mettre à niveau les clients

Mettre à jour *log.message.format.version* (redémarrage progressif)



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



Introduction

Différentes alternatives supportées pour la sécurité :

- Authentification des connections clients vers les brokers via SSL ou SASL
- Authentification des connections des brokers vers *Zookeeper*
- Cryptage des données transférées avec les clients via SSL
- Autorisation des opérations read/write/create/delete/... par client
- Possibilité d'intégration avec d'autres mécanismes d'authentification et d'autorisation

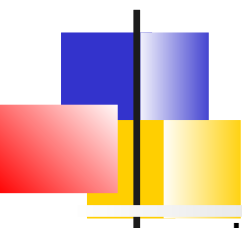
Naturellement, dégradation des performances avec SSL



Listeners

Chaque serveur doit définir l'ensemble des *listeners* utilisés pour recevoir les requêtes des clients ainsi que des autres serveurs.

Chaque listener peut être configuré pour authentifier les clients et pour garantir que le trafic entre le serveur et le client est crypté.



Configuration des listeners

Les listeners sont déclarés via la propriété ***listeners*** :

`{LISTENER_NAME}://{hostname}:{port}`

Ou LISTENER_NAME est un nom descriptif

Exemple :

```
listeners=CLIENT://localhost:9092,BROKER://localhost:9095
```

Le protocole utilisé pour chaque listener est spécifié dans la propriété ***listener.security.protocol.map***

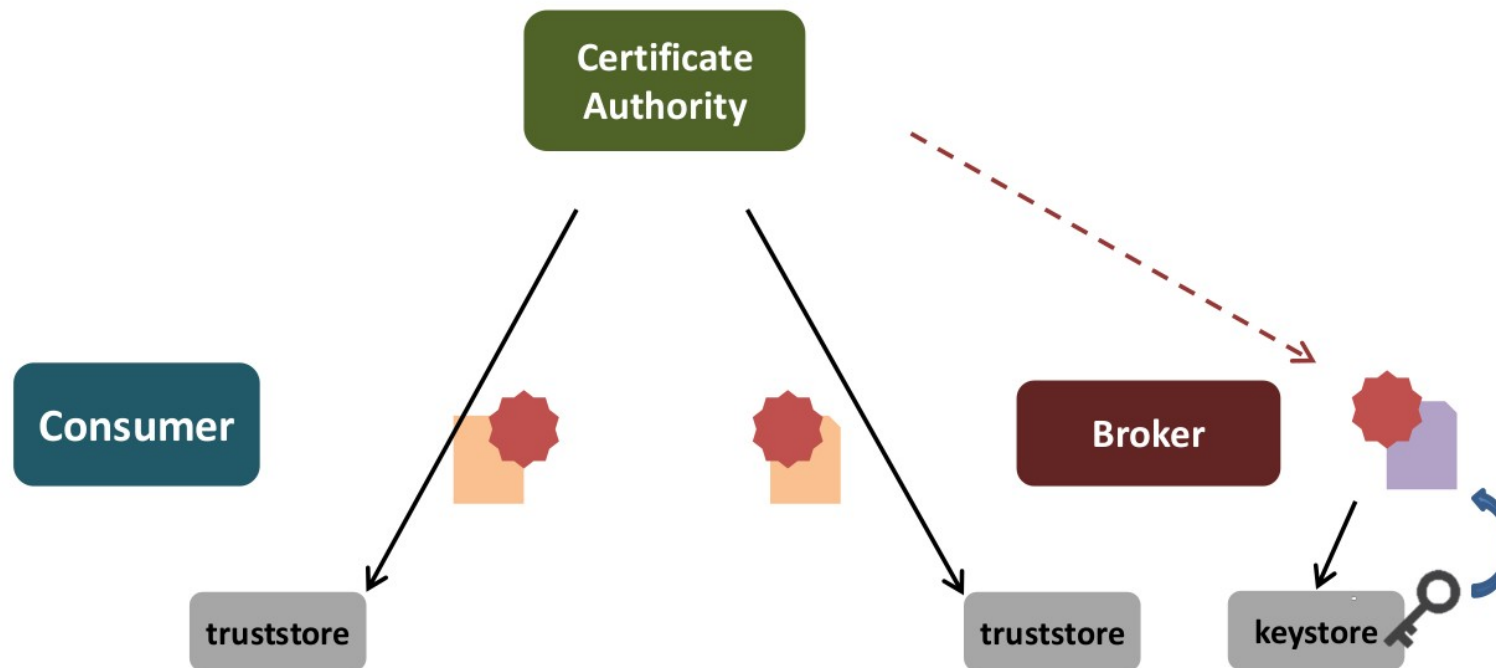
Exemple : `listener.security.protocol.map=CLIENT:SSL,BROKER:PLAINTEXT`

Les protocoles supportés sont : *PLAINTEXT*, *SSL*, *SASL_PLAINTEXT*, *SASL_SSL*

Il est possible de déclarer le listener utilisé pour la communication inter broker via ***inter.broker.listener.name*** et ***controller.listener.names***

SSL

SSL pour le cryptage et l'authentification





Préparation des certificats

Générer une paire clé publique/privé pour chaque serveur

```
keytool -keystore server.keystore.jks -alias localhost -validity 365 -keyalg RSA -genkey
```

Créer son propre CA (Certificate Authority)

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

Importer les CA dans les truststore

```
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert
```

```
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert
```

Créer une CSR (Certificate signed request)

```
keytool -keystore server.keystore.jks -alias localhost -certreq -file cert-file
```

Signer la CSR avec le CA

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days 365 -  
CAcreateserial -passin pass:servpass
```

Importer le CA et la CSR dans le keystore

```
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert
```

```
keytool -keystore server.keystore.jks -alias localhost -import -file cert-signed
```



Configuration du broker

server.properties :

```
listeners=PLAINTEXT://host.name:port,SSL://  
host.name:port
```

```
ssl.keystore.location=/home/ubuntu/ssl/server.keystore.jks
```

```
ssl.keystore.password=servpass
```

```
ssl.key.password=servpass
```

```
ssl.truststore.location=/home/ubuntu/ssl/server.truststore.jks
```

```
ssl.truststore.password=servpass
```




Configuration du client

```
security.protocol=SSL
```

```
ssl.truststore.location=/var/private/ssl/  
client.truststore.jks
```

```
ssl.truststore.password=clipass
```

```
--producer.config client-ssl.properties
```

```
--consumer.config client-ssl.properties
```



Authentification des clients via SSL

Si l'authentification du client est requise, un *keystore* doit également être créé et la configuration de *client.properties* doit contenir :

```
ssl.keystore.location=/var/private/client.keystore.jks  
ssl.keystore.password=test1234  
ssl.key.password=test1234
```



SASL

Simple Authentication and Security Layer pour
l'authentification

Kafka utilise JAAS pour la configuration SASL.

Mécanismes:

- GSSAPI: Kerberos
- SCRAM-SHA-256, SCRAM-SHA-512: hashed passwords
- PLAIN: username/password en clair
- OAUTHBEARER (Depuis Kafka 3.x)



Configuration JAAS

La configuration JAAS s'effectue :

- Soit via un **fichier jaas** contenant :
 - Une section *KafkaServer* pour l'authentification auprès d'un broker
 - Une section *Client* pour s'authentifier auprès de Zookeeper

L'exemple suivant définit 2 utilisateurs admin et alice qui pourront accéder au broker et l'identité avec laquelle le broker initiera les requêtes inter-broker

```
KafkaServer {  
    org.apache.kafka.common.security.plain.PlainLoginModule required  
    username="admin"  
    password="admin-secret"  
    user_admin="admin-secret"  
    user_alice="alice-secret";  
};
```

–



Configuration JAAS

La configuration JAAS peut également se faire via la propriété ***sasl.jaas.config*** préfixé par le mécanisme SASL

```
listener.name.sasl_ssl.scram-sha-  
256.sasl.jaas.config=org.apache.kafka.common.security.scram  
.ScramLoginModule required \  
  username="admin" \  
  password="admin-secret";
```



Mécanismes

SASL/Kerberos : Nécessite un serveur (Active Directory par exemple), d'y créer les Principals représentant les brokers. Tous les hosts kafka doivent être atteignables via leur FQDNs

SASL/PLAIN est un mécanisme simple d'authentification par login/mot de passe. Il doit être utilisé avec TLS. Kafka fournit une implémentation par défaut qui peut être étendue pour la production

SASL/SCRAM (256/512) (Salted Challenge Response Authentication Mechanism) : Mot de passe haché stocké dans Zookeeper

SASL/OAUTHBEARER : Basé sur OAuth2 mais pas adapté à la production



SASL PLAIN

1. Configurer le fichier JAAS
2. Passer l'emplacement du fichier de configuration JAAS en tant que paramètre JVM à chaque broker

```
-Djava.security.auth.login.config=/etc/kafka/kafka_server_jaas.conf
```

3. Configurer SASL dans *server.properties*

```
listeners=SASL_SSL://host.name:port
security.inter.broker.protocol=SASL_SSL
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN
```



Configuration du client

Créer le fichier Jaas

```
KafkaClient {  
    org.apache.kafka.common.security.plain.PlainLoginModule  
    required  
    username="alice"  
    password="alice-secret"  
} ;
```

Ajouter un paramètre JVM

```
-Djava.security.auth.login.config=/home/ubuntu/ssl/client_jaas.conf
```

client.properties

```
security.protocol=SASL_SSL  
sasl.mechanisms=PLAIN
```




SASL / PLAIN en production

Doit être utilisé avec SSL

Possibilité d'éviter de stocker des mots de passe clairs en configurant des *callback handler* qui obtiennent le nom d'utilisateur et le mot de passe d'une source externe via :

sasl.server.callback.handler.class

et

sasl.client.callback.handler.class.

Possibilité d'intégrer des *callback handler* utilisant des serveurs d'authentification externes pour la vérification du mot de passe via

sasl.server.callback.handler.class



Autorisation

Kafka fournit une implémentation permettant de définir des ACL

Pour l'activer en mode Kraft :

```
authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer
```

Les ACLs sont stockés dans les méta-données gérées par les contrôleurs

Les règles peuvent être exprimées comme suit :

Principal P is [Allowed/Denied] Operation O From Host H on any Resource R matching ResourcePattern RP

L'utilitaire ***kafka-acls.sh*** permet de gérer les ACLs



Exemple

Principals User:Bob et User:Alice sont autorisés à effectuer les opérations Read et Write sur le Topic Test-Topic à partir des IP 198.51.100.0 et IP 198.51.100.1

```
bin/kafka-acls.sh --authorizer-properties
zookeeper.connect=localhost:2181 --add --
allow-principal User:Bob --allow-principal
User:Alice --allow-host 198.51.100.0 --
allow-host 198.51.100.1 --operation Read --
operation Write --topic Test-topic
```



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



ZooKeeper

CPU : Typiquement pas un goulot d'étranglement

- 2 - 4 CPU

Disque : Sensible à la latence I/O, Utilisation d'un disque dédié. De préférence SSD

- Au moins 64 Gb

Mémoire : Pas d'utilisation intensive

- Dépend de l'état du cluster
- 4 Gb - 16 Gb (Pour les très grand cluster : plus de 2000 partitions)

JVM : Pas d'utilisation intensive de la heap

- Au moins 1 Gb pour le cache de page
- 1 Gb - 4 Gb

Réseau : La bande passante ne doit pas être partagée avec d'autres applications



Kafka Broker

CPU : Pas d'utilisation intensive du CPU

Disque :

- RAID 10 est mieux
- SSD n'est pas plus efficace•

Mémoire : Pas d'utilisation intensive (sauf pour le compactage)

- 16 Gb - 64 Gb

JVM :

- 4 Gb - 6 Gb

Réseau:

- 1 Gb - 10 Gb Ethernet (pour les gros cluster)
- La bande passante ne doit pas être partagée avec d'autres applications



Configuration système

Étendre la limite du nombre de fichiers ouverts :

- *ulimit -n 100000*

Configuration Virtual Memory
(*/etc/sysctl.conf pour ubuntu*)

- *vm.swappiness=1*
- *vm.dirty_background_ratio=5*
- *vm.dirty_ratio=60*



JVM

KAFKA_HEAP_OPTS peut être utilisée

Heap size :

– -Xms4G -Xmx4G or -Xms6G -Xmx6G

Options JVM

-XX:MetaspaceSize=96m -XX:+UseG1GC -
XX:MaxGCPauseMillis=20

-XX:InitiatingHeapOccupancyPercent=35 -
XX:G1HeapRegionSize=16M

-XX:MinMetaspaceFreeRatio=50 -
XX:MaxMetaspaceFreeRatio=80



Dimensionnement cluster

Généralement basée sur les capacités de stockage :

$\text{Nb de msg-jour} * \text{Taille moyenne} * \text{Rétention} * \text{Replication} / \text{stockage par Broker}$

Les ressources doivent être surveillées et le cluster doit être étendu plus de 70% est atteint pour :

- CPU
- L'espace de stockage
- La bande passante



Zookeeper

Un nombre impair de serveurs
Zookeeper

- Nécessité d'un Quorum (vote majoritaire)
- 3 nœuds permet une panne
- 5 nœuds permet 2 pannes



Partitionnement des topics

Augmenter le nombre de partitions
permet plus de consommateurs

Mais augmente généralement la taille du
cluster

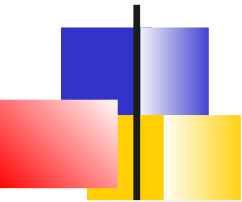
Évaluer l'utilisation d'un autre topic ?

Généralement 24 est suffisant



Administration

Gestion des topics
Stockage et rétention des partitions
Gestion du cluster
Sécurité
Dimensionnement
Surveillance



Principaux métriques brokers accessibles via JMX

Métrique	Description	Alerte
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions	Nombre de partition sous-répliquée	Si > 0
kafka.controller:type=KafkaController,name=OfflinePartitionsCount	Nombre de partitions qui n'ont pas de leader actif	Si > 0
kafka.controller:type=KafkaController,name=ActiveControllerCount	Nombre de contrôleur actif dans le cluster	Si != 1
kafka.network:type=RequestMetrics,name=RequestsPerSec,request={Produce FetchConsumer FetchFollower}	Nombre de requêtes par seconde, pour produire et récupérer	Si changement significatif
kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica	Retard maximal des messages entre les répliques et le leader	
kafka.server:type=ReplicaManager,name=IsrShrinksPerSec	Cadence de shrink des ISR	
kafka.server:type=ReplicaManager,name=IsrExpandsPerSec	Cadence d'expansion des ISR	



Métriques clients

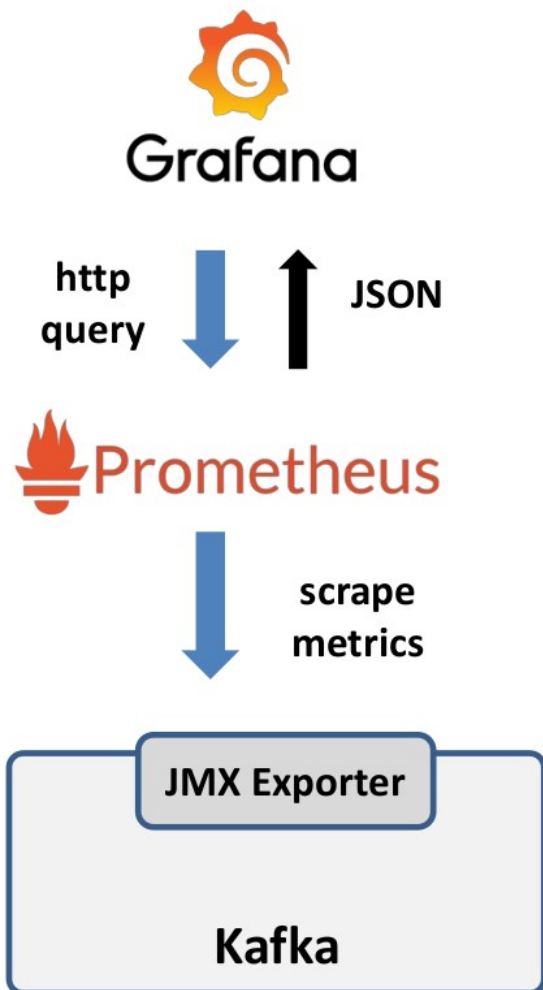
Producteur

Métrique	Description
kafka.producer:type=producermetrics,client-id=([-w]+),name=io-ratio	Fraction de temps de la thread passé dans les I/O
kafka.producer:type=producer-metrics,client-id=([-w]+),name=io-wait-ratio	Fraction de temps de la thread passé en attente

Consommateur

Métrique	Description
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=([-w]+),records-lag-max	Le décalage maximum en termes de nombre d'enregistrements pour n'importe quelle partition

Outil de visualisation



Tdb dispo :

<https://grafana.com/grafana/dashboards/721>

Exportateur JMX

https://github.com/prometheus/jmx_exporter/blob/master/example_configs/kafka-2_0_0.yml



Autres outils

Confluent Control Center

JConsole

Graphite

CloudWatch

DataDog

SMM of Data Flow (Hortonworks)

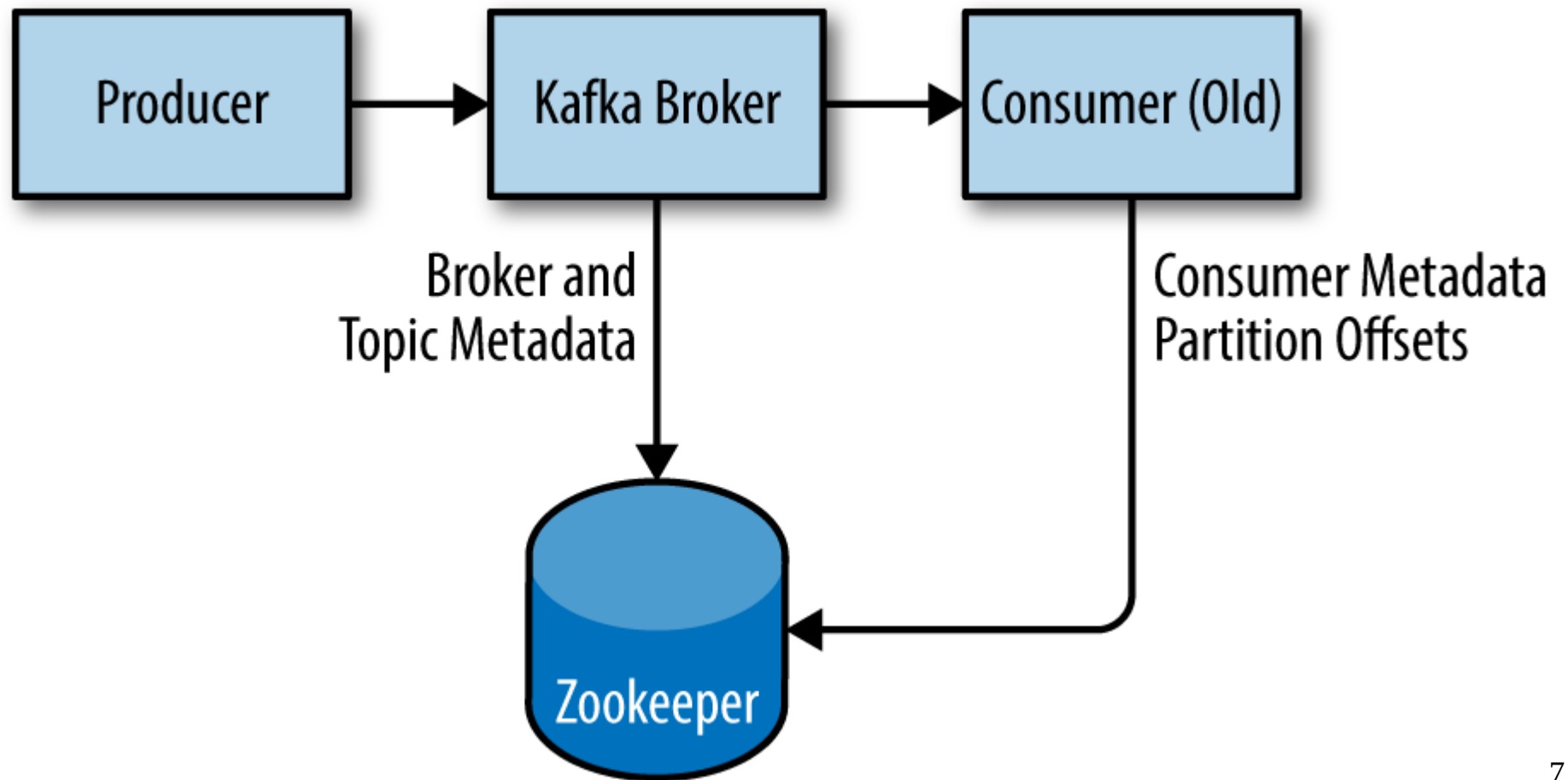
...



Annexes

Apache Zookeeper Contrôleur et Zookeeper SASL SCRAM

Kafka et Zookeeper



Zookeeper

Principes

“High-performance coordination service for distributed applications”

Utilisé par Kafka pour la gestion de configuration et la synchronisation

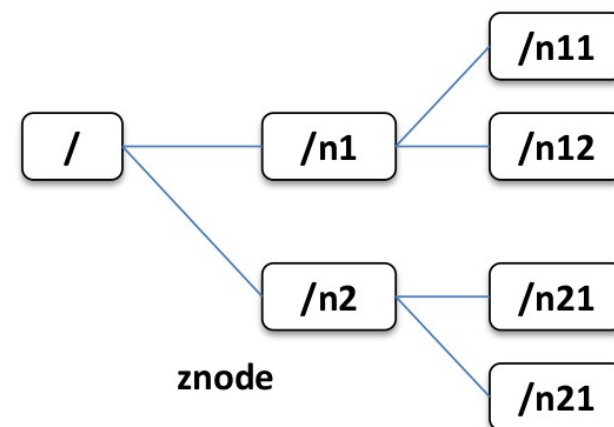
Stocke les méta-données du cluster Kafka

Répliqué sur plusieurs hôtes formant un *ensemble*

Fournit un espace de noms hiérarchiques

Exemples de nœuds pour Kafka :

- */controller*
- */brokers/topics/*
- */config*





Vocabulaire Zookeeper

Nœud (zNode) : Donnée identifiée par un chemin

Client : Utilisateur du service Zookeeper

Session : Établie entre le client et le service Zookeeper

Noeud éphémère : le nœud existe aussi longtemps que la session qui l'a créé est active

Watch :

- Déclenché et supprimé lorsque le nœud change
- Clients peuvent positionné un watch sur un nœud *znode*



Zookeeper Distribution

Kafka contient des scripts permettant de démarrer une instance de Zookeeper mais il est préférable d'installer une version complète à partir de la distribution officielle de *Zookeeper*

<https://zookeeper.apache.org/>



Exemple installation standalone

```
# tar -zxf zookeeper-3.4.6.tar.gz
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```



Vérification Zookeeper

```
# telnet localhost 2181
```

```
Trying ::1...
```

```
Connected to localhost.
```

```
Escape character is '^['.
```

```
srvr
```

```
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
```

```
Latency min/avg/max: 0/0/0
```

```
Received: 1
```

```
Sent: 0
```

```
Connections: 1
```

```
Outstanding: 0
```

```
Zxid: 0x0
```

```
Mode: standalone
```

```
Node count: 4
```

```
Connection closed by foreign host.
```

```
#
```



Ensemble Zookeeper

Un cluster Zookeeper est appelé un **ensemble**

Une instance est élue comme ***leader***

L'ensemble contient un nombre impair d'instances
(algorithme de consensus basé sur la notion de quorum).

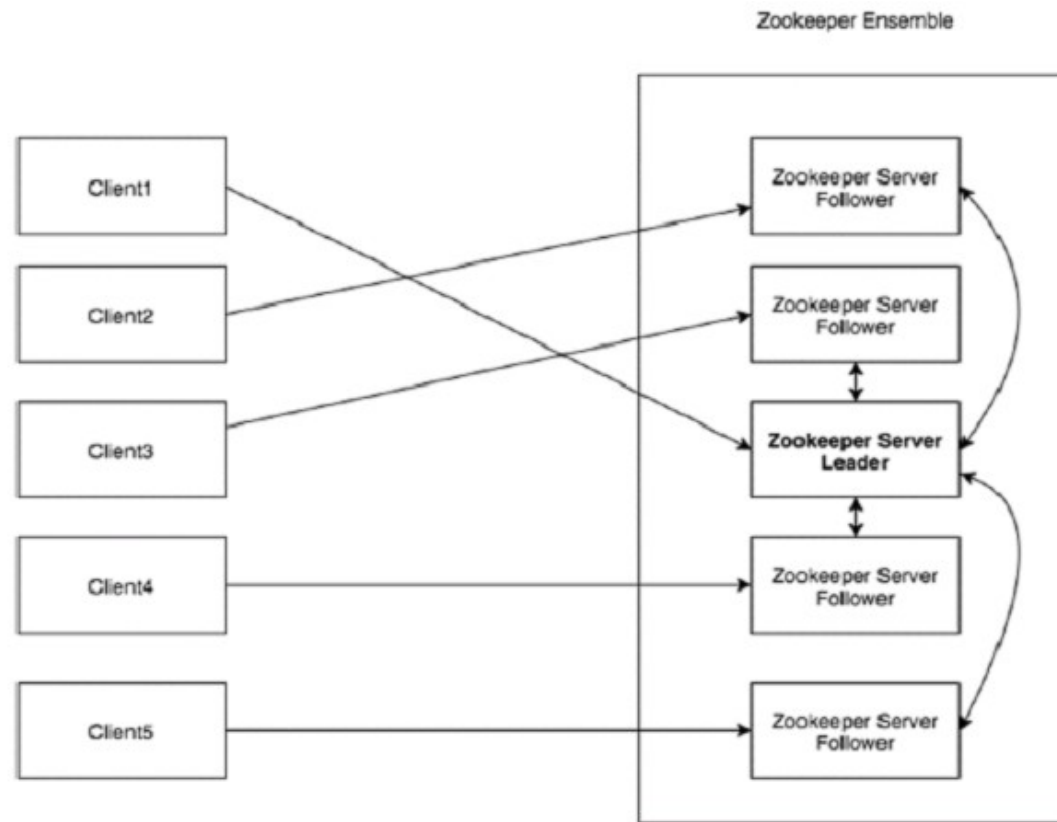
– Le nombre dépend du niveau de tolérance aux pannes
voulu :

- 3 nœuds => 1 défaillance
- 5 nœuds => 2 défaillances

Les clients peuvent s'adresser à n'importe quelle instance
pour lire/écrire les données

Lors d'une écriture, le *leader* coordonne les écritures sur
les *followers*

Architecture *ZooKeeper*





Propriétés de config

tickTime : L'unité de temps en ms

initLimit : nombre de tick autorisé pour la connexion des suiveurs au leader

syncLimit : nombre de tick autorisé pour la synchronisation suiveur/leader

dataDir : Répertoire de stockage des données

clientPort : Port utilisé par les clients

La configuration répertorie également chaque sous la forme :

server.X = nom d'hôte: peerPort: leaderPort

- **X** : numéro d'identification du serveur.
- **nom d'hôte** : IP
- **peerPort** : Port TCP pour communication entre serveurs
- **leaderPort** : Port TCP pour l'élection.

Optionnel :

4lw.commands.whitelist : Les commandes d'administration autorisées



Configuration d'un ensemble

Les serveurs doivent partager une configuration commune listant les serveurs et chaque serveur doit contenir un fichier *myid* dans son répertoire de données contenant son identifiant

Exemple de config :

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```



Vérfications Ensemble Zookeeper

Se connecter à une instance :

```
./zkCli.sh -server 127.0.0.1:2181
```

Voir le mode (leader ou suiveur) d'une instance

si la commande *stat* est autorisée

```
echo stat | nc localhost 2181 | grep Mode
```



Quelques commandes

ls [path] : Lister un zNode

create [zNode] : Créer un nœud

delete/deleteall : Suppression (récursive) d'un nœud

get/set [zNode] : Lire/écrire la valeur du nœud

history : Historique des commandes

quit : Quitter zkCli



Annexes

Apache Zookeeper
Cluster avec Zookeeper
SASL SCRAM



Configuration broker

cluster.id : Chaque cluster a un ID

broker.id : Chaque broker doit avoir un identifiant entier unique à l'intérieur du cluster

port : Par défaut 9092

zookeeper.connect : Listes des serveurs Zookeeper sous la forme *hostname:port/path*
path chemin optionnel pour l'environnement *chroot*

log.dirs : Liste de chemins locaux où kafka stocke les messages

auto.create.topics.enable : Création automatique de topic à l'émission ou à la consommation



Configuration cluster

2 contraintes pour la configuration d'un cluster :

- Tous les brokers doivent avoir le même paramètre ***zookeeper.connect***.
- Chaque broker doit avoir une valeur unique pour ***broker.id***.



Cluster et ensemble Zookeeper

Kafka utilise *Zookeeper* pour stocker des informations de métadonnées sur les brokers, les topics et les partitions.

Les écritures sont peu volumineuses et il n'est pas nécessaire de dédier un ensemble *Zookeeper* à un seul cluster Kafka.

- => Un seul ensemble pour plusieurs clusters Kafka.



Rôles du contrôleur

Un des brokers Kafka (nœud éphémère dans *Zookeeper*)

Pour le visualiser :

```
./bin/zookeeper-shell.sh [ZK_IP] get /controller
```

- Gère le cluster en plus des fonctionnalités habituelles d'un broker
- Détecte le départ / l'arrivée de broker via *Zookeeper* (*/brokers ids*)
- Gère les changements de Leaders

Si le contrôleur échoue:

- un autre broker est désigné comme nouveau contrôleur
- les états des partitions (liste des leaders et des ISR) sont récupérés à partir de *Zookeeper*



Responsabilités

Lors d'un départ de broker, pour toutes les partitions dont il est le leader, le contrôleur :

- Choisit un nouveau leader et met à jour l'ISR
- Met à jour le nouveau Leader et l'ISR (État des partitions) dans *Zookeeper*
- Envoie le nouveau Leader/ISR à tous les brokers contenant le nouveau leader ou les followers existants

Lors de l'arrivée d'un broker, le contrôleur lui envoie le Leader et l'ISR



Annexes

Apache Zookeeper
Contrôleur et Zookeeper
SASL SCRAM



SASL SCARM

Utilisé avec SSL pour une
authentification sécurisée

- *Zookeeper* est utilisé pour stocker les
crédentiels
- Sécurisé via l'utilisation d'un réseau
privé



Configuration des créden*t*iels SASL SCRAM

Communication Inter-Broker : user “admin”

```
kafka-configs.sh --zookeeper host:port --alter \  
--add-config 'SCRAM-SHA-256=[password=adminpass],SCRAM-  
SHA-512=[password=adminpass]' \  
--entity-type users --entity-name admin \  

```

Communication Client-Broker : user “user”

```
kafka-configs.sh --zookeeper host:port --alter \  
--add-config 'SCRAM-SHA-256=[password=userpass],SCRAM-  
SHA-512=[password=userpass]' \  
--entity-type users --entity-name user\  

```



Configuration du broker

Créer le fichier Jaas

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="adminpass"  
} ;
```

Ajouter un paramètre JVM

```
-Djava.security.auth.login.config=/home/ubuntu/ssl/kafka_jaas.conf
```

server.properties

```
listeners=SASL_SSL://host.name:port  
security.inter.broker.protocol=SASL_SSL  
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512  
sasl.enabled.mechanisms=SCRAM-SHA-512
```



Configuration du client

Créer le fichier Jaas

```
KafkaClient {  
    org.apache.kafka.common.security.scram.ScramLoginModule  
    required  
    username="alice"  
    password="alice-secret"  
} ;
```

Ajouter un paramètre JVM

```
-Djava.security.auth.login.config=/home/ubuntu/ssl/client_jaas.conf
```

client.properties

```
security.protocol=SASL_SSL  
sasl.mechanisms=SCRAM-SHA-512
```