





# Kafka : Messagerie distribuée avec Apache Kafka

---

David THIBAU – 2021

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

## Introduction à Kafka

- Le projet Kafka
- Cas d'usage
- Concepts

## Installation

- Pré-requis et ensemble Zookeeper
- Broker Kafka
- Utilitaires Kafka
- Cluster Kafka

## Kafka APIs

- Producer API
- Consumer API
- Connect API
- Autres APIS
- Frameworks

## Réplication et fiabilité

- Stockage des partitions
- Mécanismes de réplication
- Rôle du contrôleur
- At Most One, At Least One
- Exactly Once

## Administration

- Gestion des topics
- 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

Taillé pour le cluster et 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
- Garanties de fiabilité



# Plateforme de streaming distribuée

---

Kafka a donc trois capacités clés:

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



# 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 :

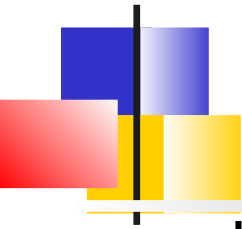
- Une distribution de Kafka
- Fonctionnalité commerciale additionnelle



# 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'a 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

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



# Kafka comme système de stockage

---

Les enregistrements sont écrits et répliqués sur le disque.

Les structure de stockage est très scalable.

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

=> 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

Voir par exemple le projet *ksqlDB*  
(<https://ksqldb.io/overview.html>)



# Kafka pour le traitement de flux

---

Kafka permet de définir des processeurs de flux qui

- consomment en continu des flux de données à partir de topics d'entrée,
- effectuent un certain traitement
- produisent des flux continus de données vers des topics de sortie.

Ces processeurs permettent d'effectuer des transformations complexes comme des calculs d'agrégations ou la fusion de 2 flux distincts

Ils peuvent être combinés en topologie et définir des pipeline de traitement d'événements temps-réel



# Bénéfices

---

La combinaison des fonctionnalités de messagerie, de stockage et de traitement de flux permet la mise en place d'applications de streaming capable de traiter les **messages passés ou futurs** contenant des données critiques (réplication et tolérance aux défaillances)

- Architecture très évolutive. Scalabilité du cluster, ajout progressif de micro-services
- Facilitation maintenance corrective. Upgrade d'applications. (Reprise du traitement des évènements passés)



# Cas d'usage typiques

---

Système de messagerie simple : Comme ActiveMQ ou RabbitMQ

Suivi des activités d'un site Web : Cas d'usage d'origine.  
Enormément de données à traiter

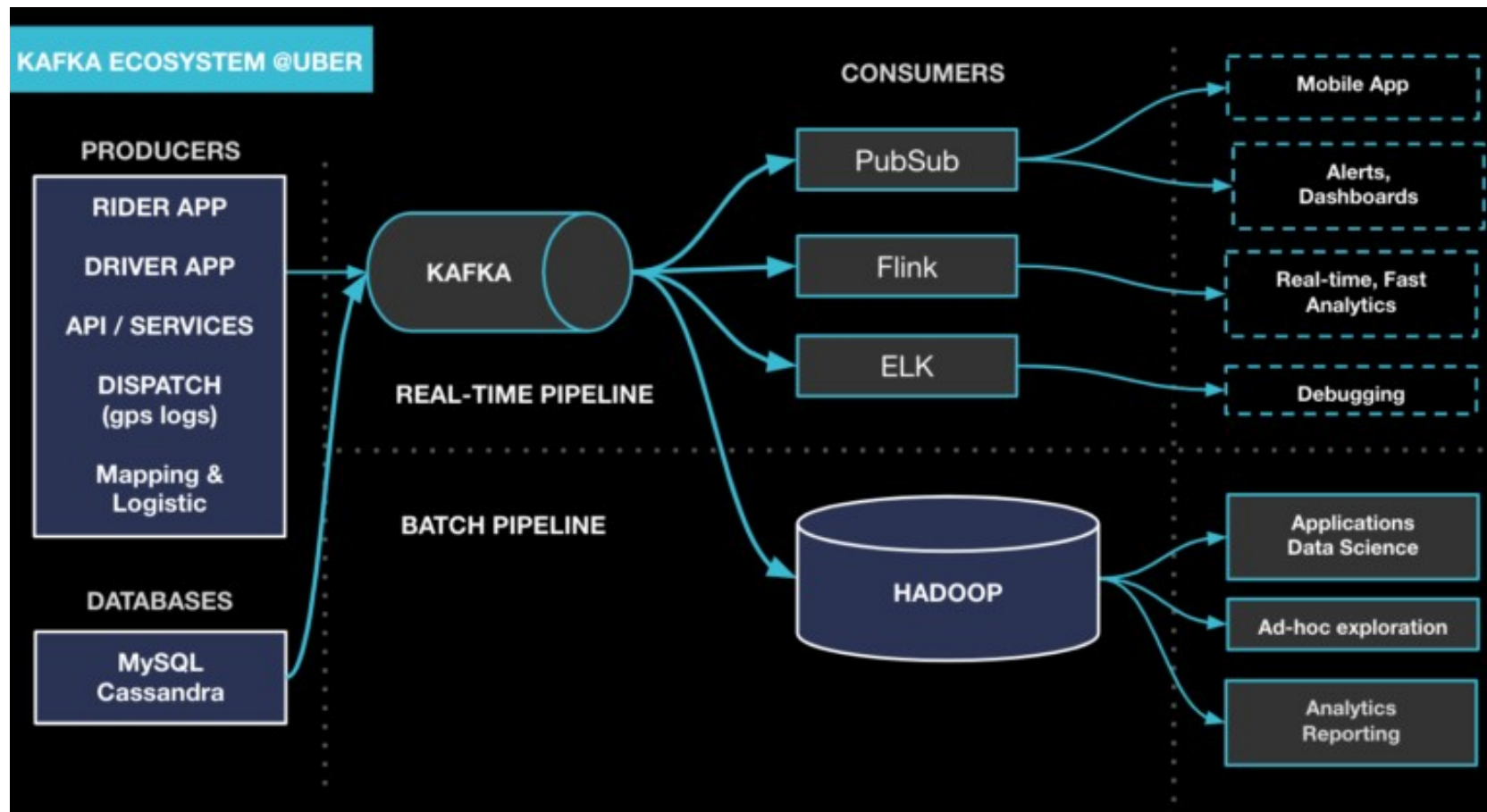
Surveillance de métriques : Centralisé des métriques  
provenant d'un système distribué, les agréger

Agrégation de traces : Agréger les lignes d'un fichier de trace

Traitement flux et BigData : Exécuter les différentes étapes de  
traitement nécessaire dans une approche BigData

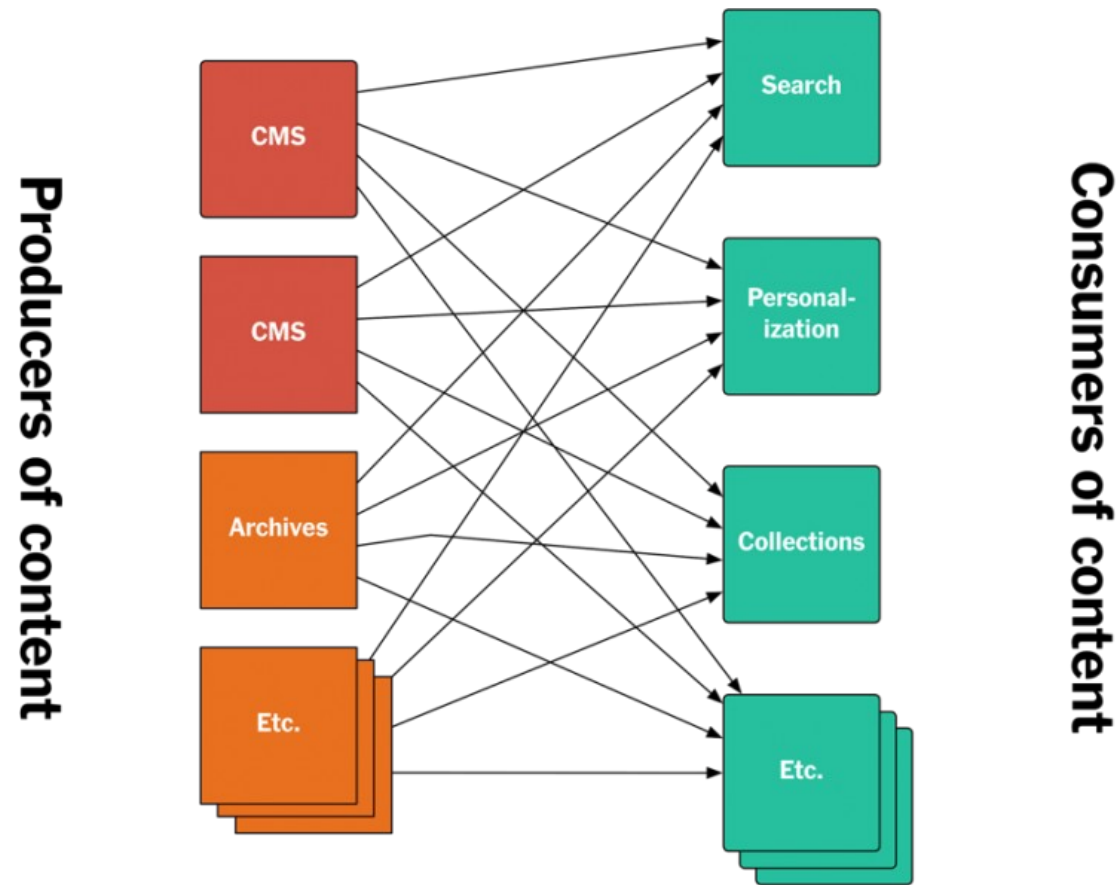
Journal de transaction : Enregistrement des transactions  
permettant de rejouer les transactions pour la réplication ou  
la recovery

# Exemple Uber



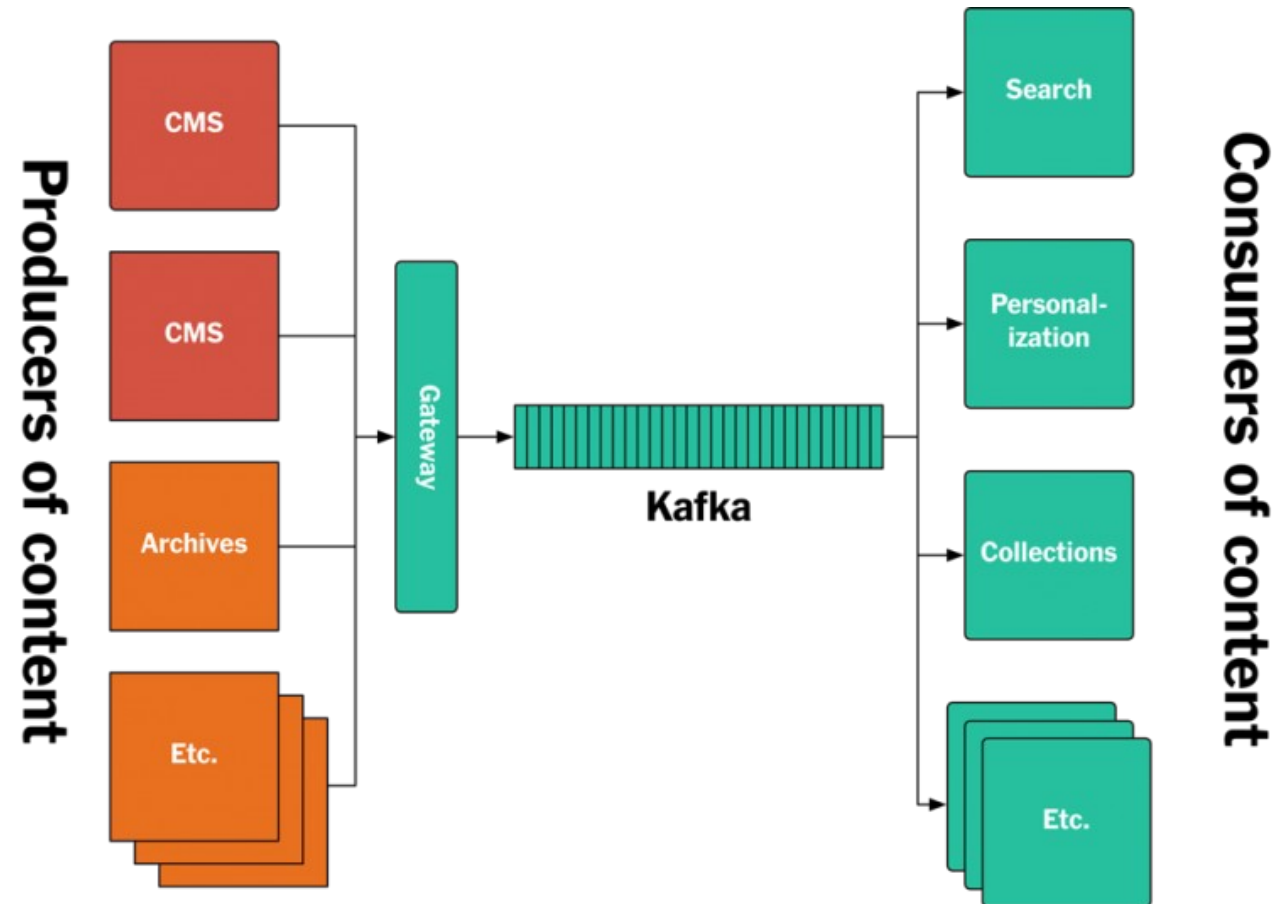


# Exemple New York Times *Avant*



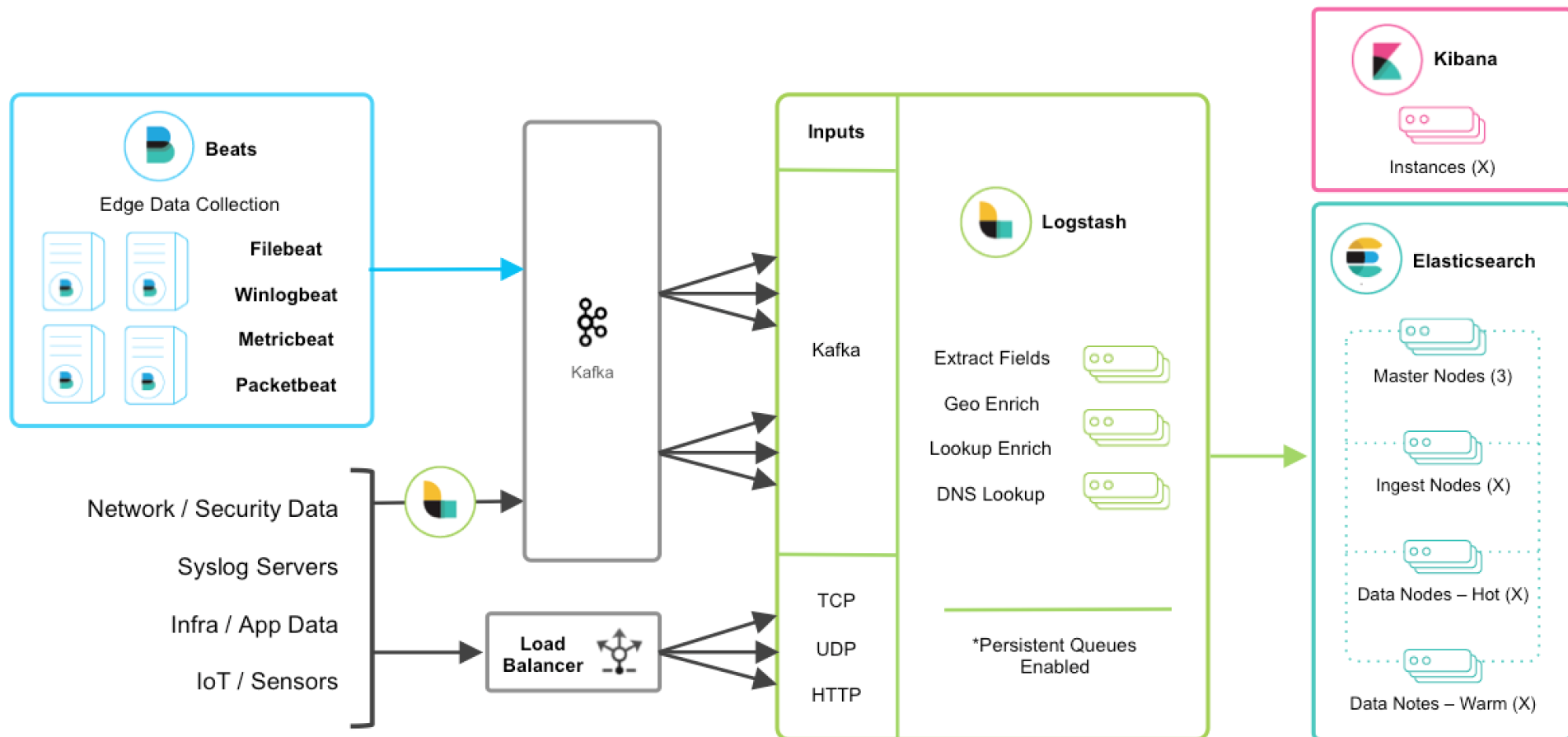
# Exemple New York Times

## *Après*



# Buffer d'évènements

## ELK Cluster





# Introduction à Kafka

---

Le projet *Kafka*  
Cas d'usage  
**Concepts**



# Concepts de base

---

Kafka s'exécute en tant que cluster sur un ou plusieurs serveurs pouvant s'étendre sur plusieurs centres de données.

Le cluster Kafka

stocke des flux d'enregistrements : les **records**  
dans des catégories appelées rubriques : les **topics** .

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



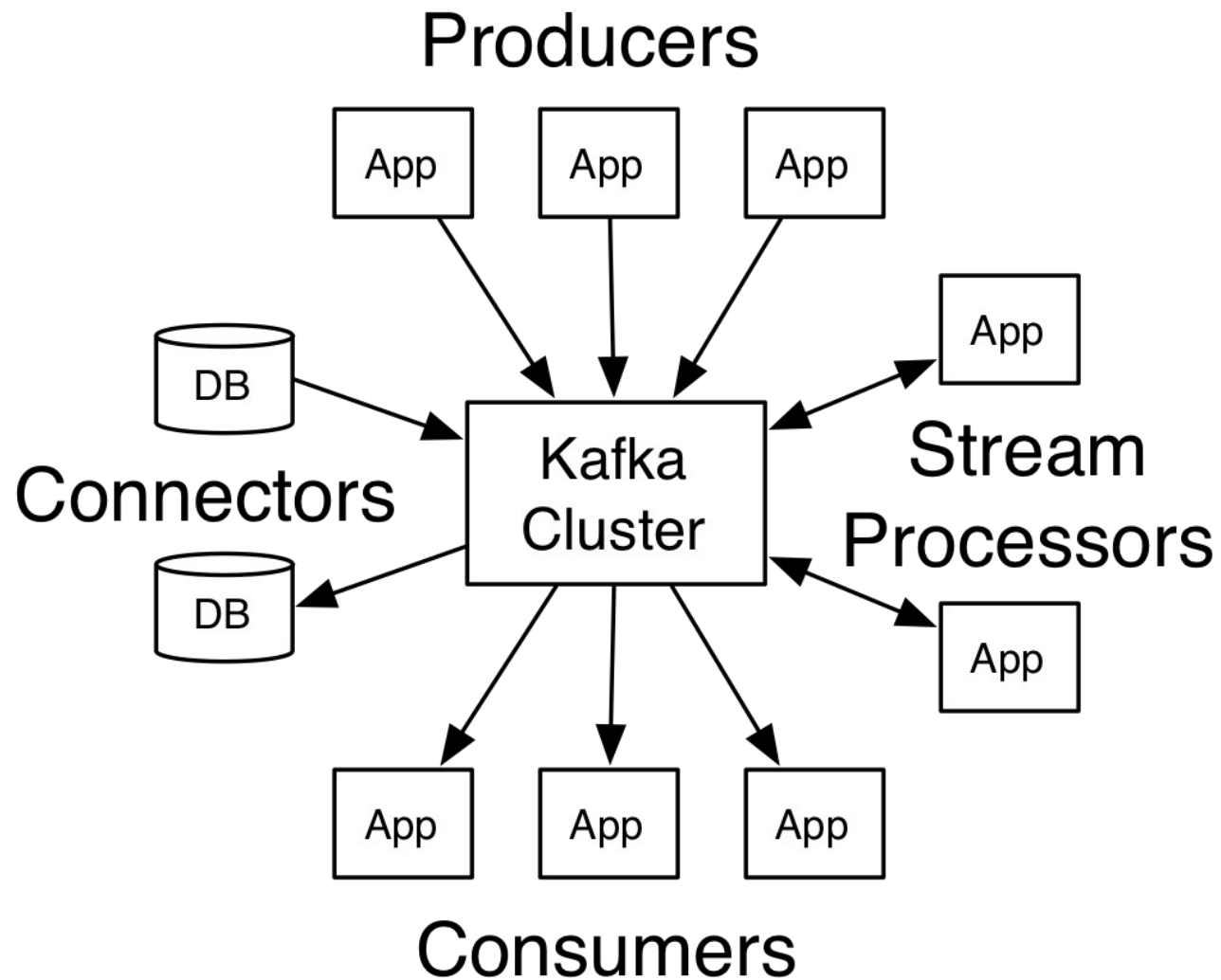
# APIs

---

Kafka propose 4 API :

- 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 réutilisables (BD, STDOUT, ...)

# APIs





# Protocole Client/Serveur

---

Dans Kafka, la communication entre les clients et les serveurs se fait avec un protocole TCP simple, performant et indépendant du langage.

Ce protocole est versionné et maintient une compatibilité ascendante avec une version plus ancienne.

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





# Implémentation cliente

---

Apache ne fournit qu'une implémentation des clients en Java.

Les 4 APIs sont donc disponibles sous forme de jar et dépendance Maven.

Pour les autres technologies, voir :

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

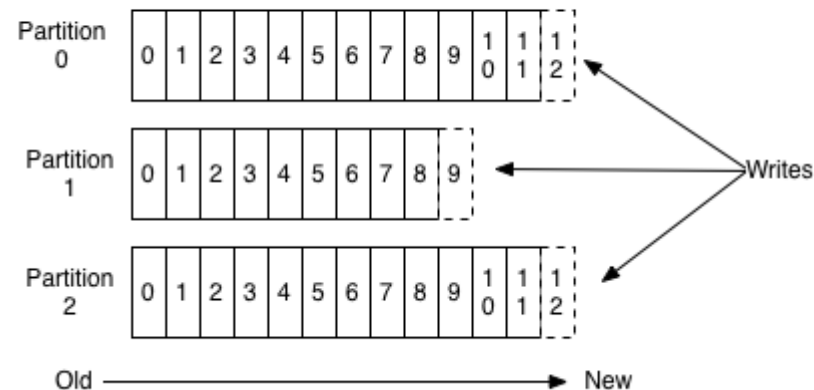
# Topic

Les records sont publiés vers des *topics*.

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

Pour chaque *topic*, le cluster Kafka conserve un journal partitionné

Anatomy of a Topic

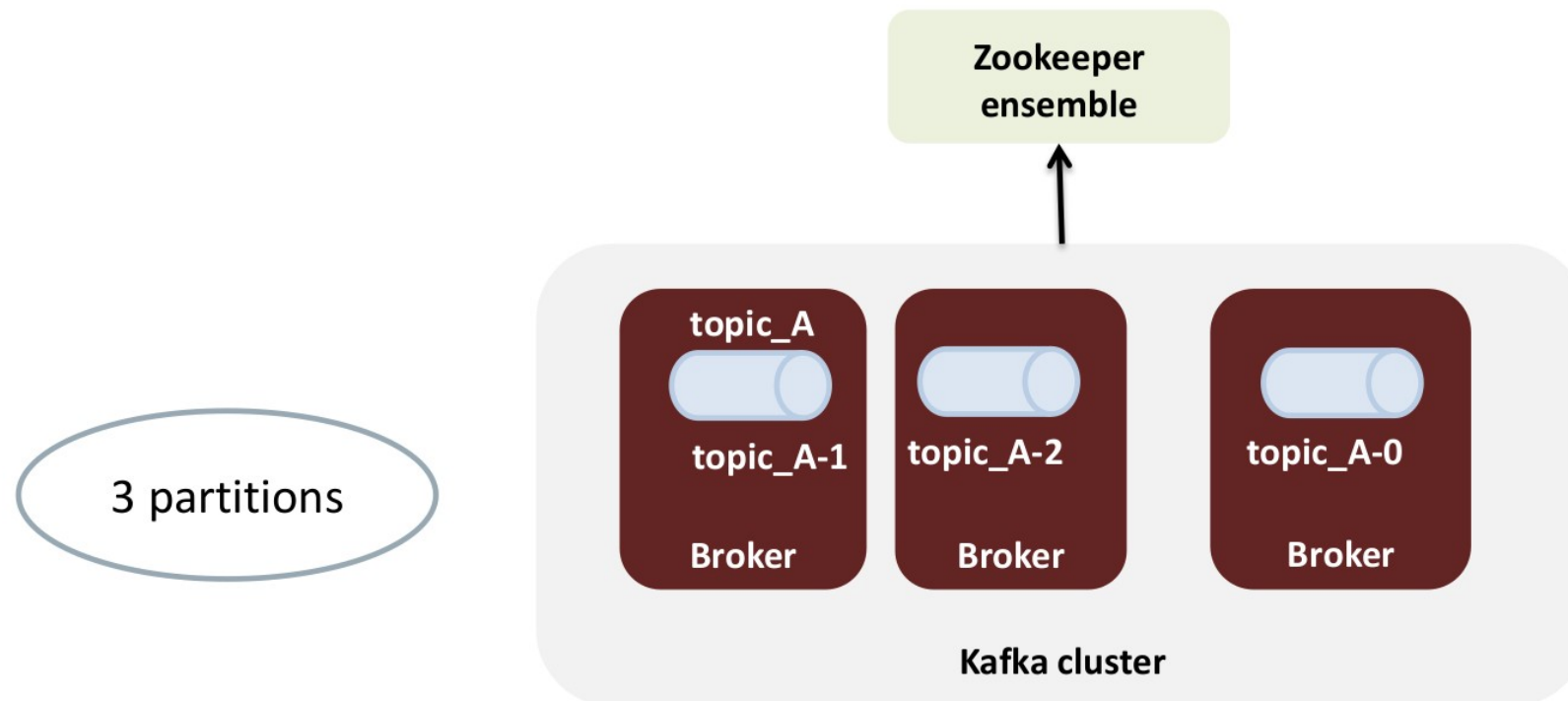


# Partitions

Les *topics* sont **partitionnés**

- Les partitions autorisent le parallélisme et augmentent la capacité de stockage

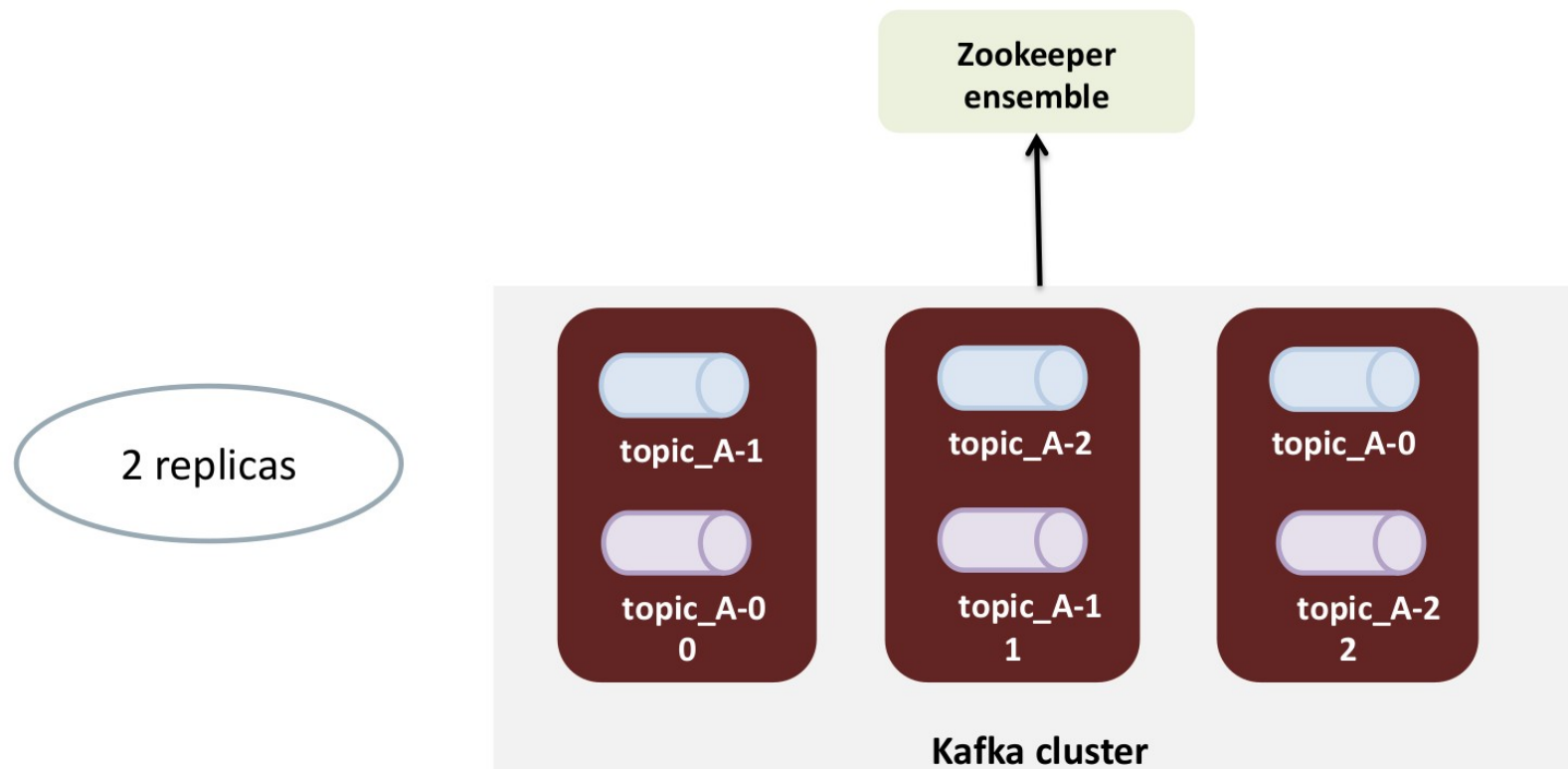
L'ordre des messages n'est garanti qu'à l'intérieur d'une partition



# Réplication

Les partitions sont **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 instance 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.

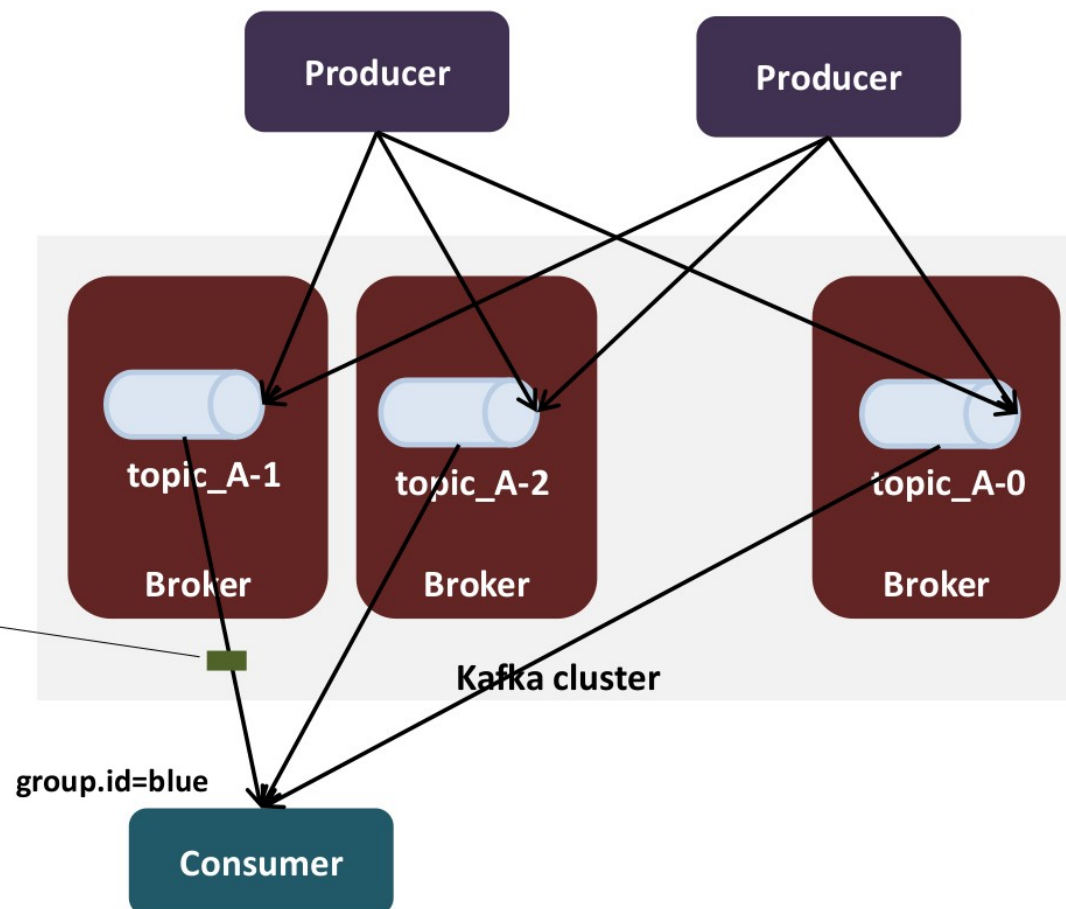
Le cluster Kafka conserve durablement tous les enregistrements publiés, qu'ils aient ou non été consommés, en utilisant une ***période de rétention*** configurable.

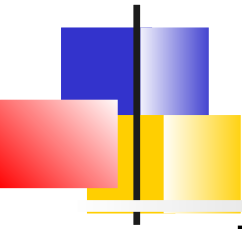
# Clients du cluster

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 responsable 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 consommateur peuvent se trouver dans des processus distincts ou sur des machines distinctes.  
=> Scalabilité et Tolérance aux fautes



# Offset consommateur

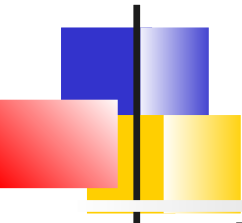
---

La seule métadonnée conservée par groupe de consommateurs est l'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 d'un offset particulier.



# Instance vs Partition

## Rééquilibrage dynamique

---

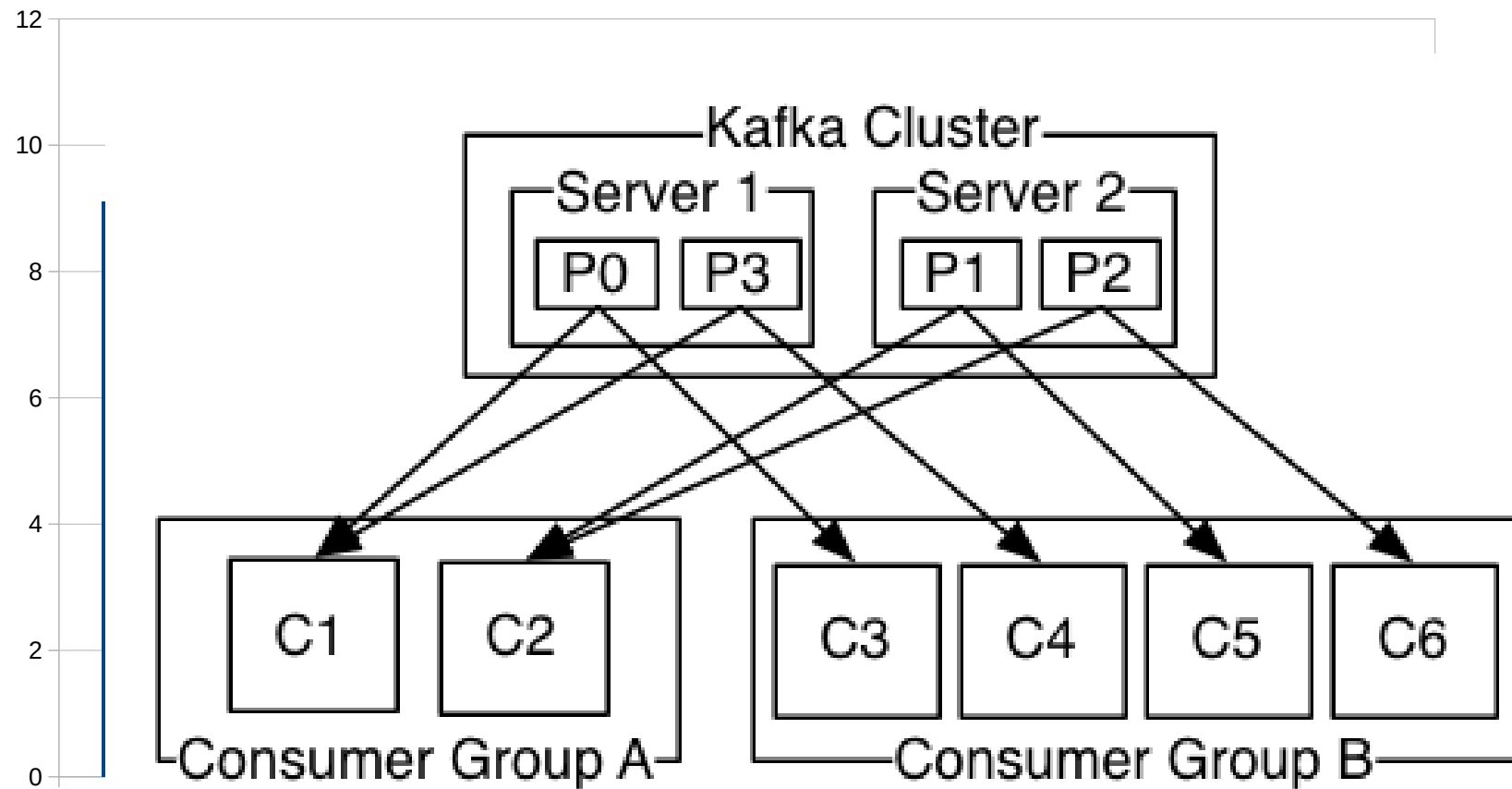
Kafka assigne les partitions à des instances de consommateur d'un même groupe.

- A tout moment, un consommateur exclusif est 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



ne  
ne  
ne



# 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 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

---

## **Pré-requis et ensemble *Zookeeper***

Broker *Kafka*

Cluster *Kafka*

Utilitaires *Kafka*



# Pré-requis

---

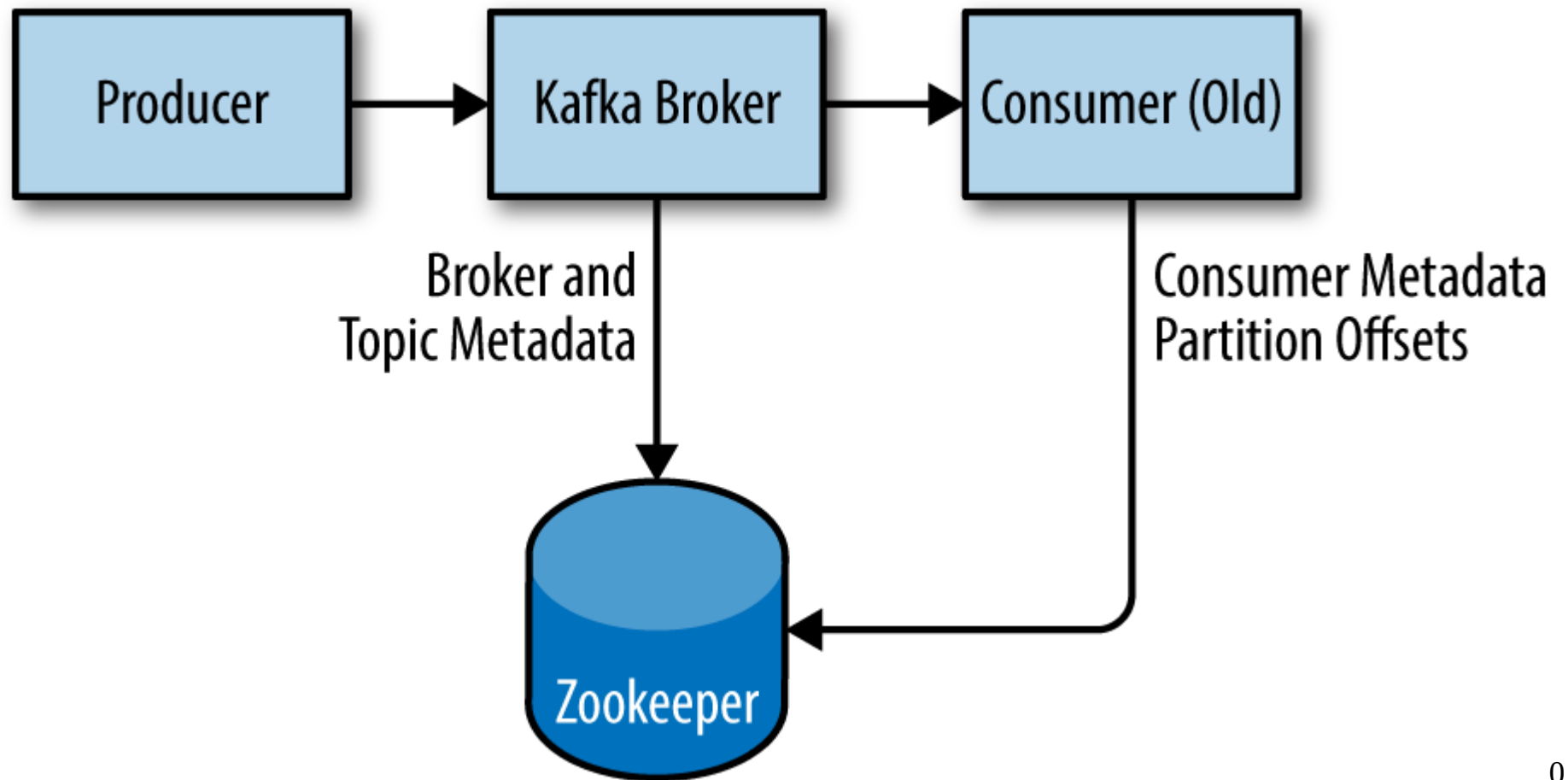
OS recommandé Linux

JRE8+ ou même mieux JDK8+

Un cluster Kafka comprend :

- Un ensemble Apache Zookeeper :  
Stockage des méta-données du cluster
- Et des instances de Kafka : les  
messages brokers

# Kafka et Zookeeper







# Principes

“High-performance coordination service for distributed applications”

Utilisé par Kafka pour la gestion de configuration et la synchronisation

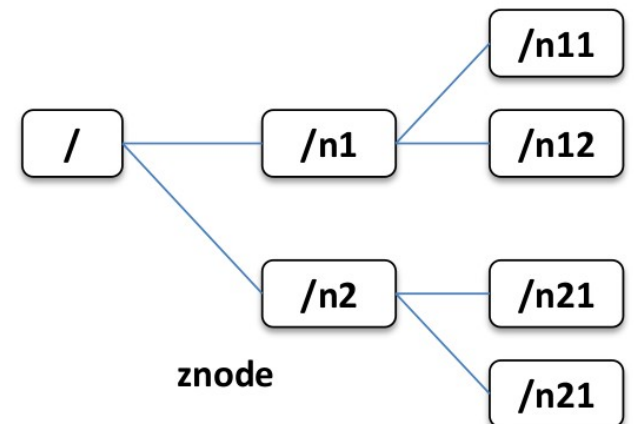
Stocke les méta-données du cluster Kafka

Répliqué sur plusieurs hôtes formant un *ensemble*

Fournit un espace de noms hiérarchiques

Exemples de nœuds pour Kafka :

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





# Vocabulaire Zookeeper

---

**Noeud** : identifié par un chemin

**Client** : Utilisateur du service Zookeeper

**Session** : Etablie entre le client et le service Zookeeper

**Noeud éphémère** : le nœuds 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 znodes



# *Zookeeper*

---

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).

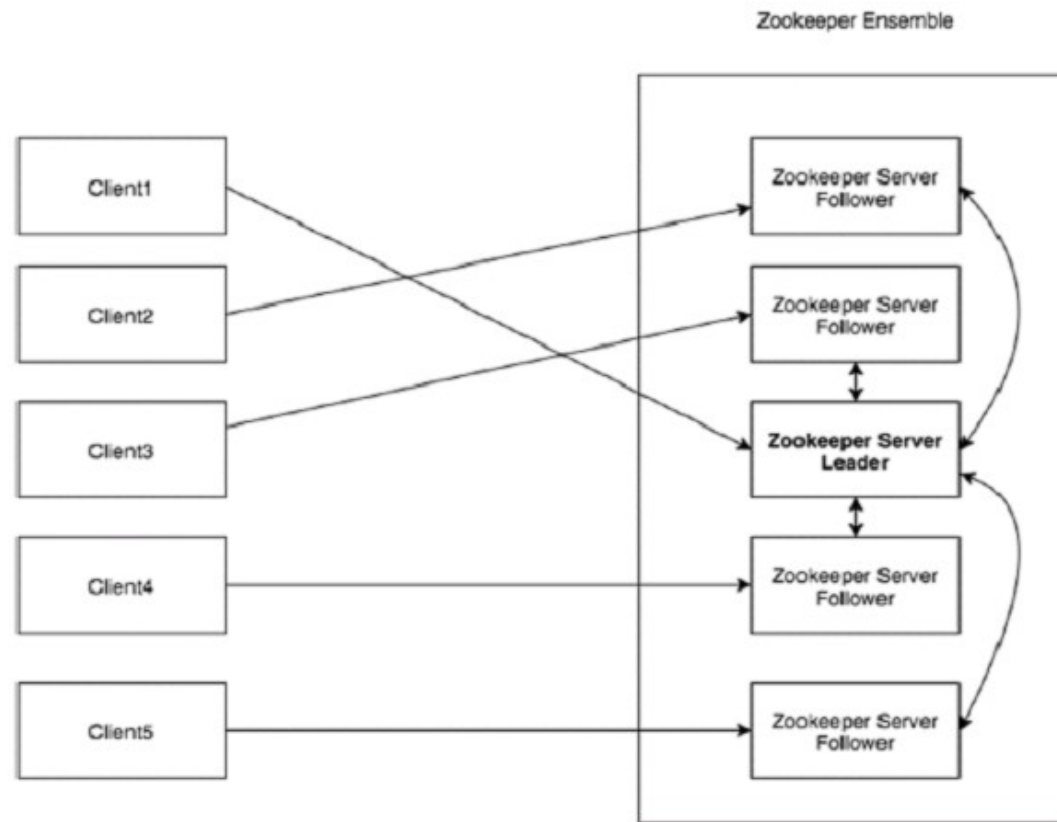
- Le nombre dépend du niveau de tolérance aux pannes voulu :

- 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
```



# Vérifications 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
```



# Installation

---

Pré-requis et ensemble *Zookeeper*

**Broker *Kafka***

Cluster *Kafka*

Utilitaires *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*



# 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 --broker-list 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
```



# Configuration broker

---

***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



# Installation

---

Pré-requis et ensemble *Zookeeper*

Broker *Kafka*

**Utilitaires *Kafka***

Cluster *Kafka*



# Gestion des topics

---

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

Exemple création de topic :

```
kafka/bin/kafka-topics.sh --create \  
  --zookeeper localhost:2181 \  
  --replication-factor 1 --partitions 13 \  
  --topic my-topic
```

Exemple listing des topics :

```
kafka/bin/kafka-topics.sh --list \  
  --zookeeper localhost:2181
```





# Configuration topic

---

Le fichier *.properties* définit également les valeurs de configuration par défaut concernant les topics

- ***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 où 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*



# Détail sur le mécanisme d'expiration des messages

Les paramètres de rétention s'appliquent sur des **segments** (fragments du log) et non sur les messages.

- Au fur et à mesure de leur arrivée, les messages sont ajoutés au segment courant de la partition.
- Une fois que le segment a atteint la taille spécifiée par **log.segment.bytes** il est fermé et un nouveau s'ouvre.
- Une fois fermé, le segment peut être considéré pour expiration.

=> Une taille de segment petite signifie que les fichiers doivent être fermés et alloués plus souvent, ce qui réduit l'efficacité globale des écritures sur disque.

=> Si le débit d'écriture est faible et la taille de segment important, les messages peuvent être retenus plus longtemps que la valeur de configuration



# Choix du nombre de partitions

---

Une fois un *topic* créé, on ne peut pas diminuer son nombre de partitions

Lors du choix, il faut tenir compte de :

- Du débit des écritures
- Le débit maximum de consommation
- L'espace disque et la bande passante réseau sur chaque broker
- En général en augmentant le nombre de partitions, on augmente le débit global de l'application en s'autorisant plus de consommateurs
- Mais évitez de surestimer, car chaque partition utilise de la mémoire et le nombre de partitions augmente le temps du processus d'élection du maître



# Envoi de message

---

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

Exemple envoi vers un topic :

```
bin/kafka-console-producer.sh \  
  --broker-list localhost:9092 \  
  --topic my-topic
```

...Puis saisir les topics 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 en utilisant l'ensemble ZooKeeper

***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-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



# Installation

---

Pré-requis et ensemble *Zookeeper*

Broker *Kafka*

Utilitaires *Kafka*

**Cluster *Kafka***



# 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***.

D'autres paramètres de configuration sont utilisés lorsque de l'exécution d'un cluster - en particulier, les paramètres qui contrôlent la réplication.





# Nombre de brokers

---

Pour déterminer le nombre de brokers :

- Premier facteur : la capacité de disque requise pour conserver les messages et la quantité de stockage disponible sur chaque *broker*.
- Second facteur à considérer est la capacité du cluster à traiter le débit de requêtes.



# 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 (en utilisant un chemin Zookeeper chroot pour chaque cluster).



# APIs

---

**Producer API**

Consumer API

Frameworks

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 ?



# Dépendance Maven

---

```
<dependency>  
  <groupId>org.apache.kafka</groupId>  
  <artifactId>kafka-clients</artifactId>  
  <version>2.7.0</version>  
</dependency>
```



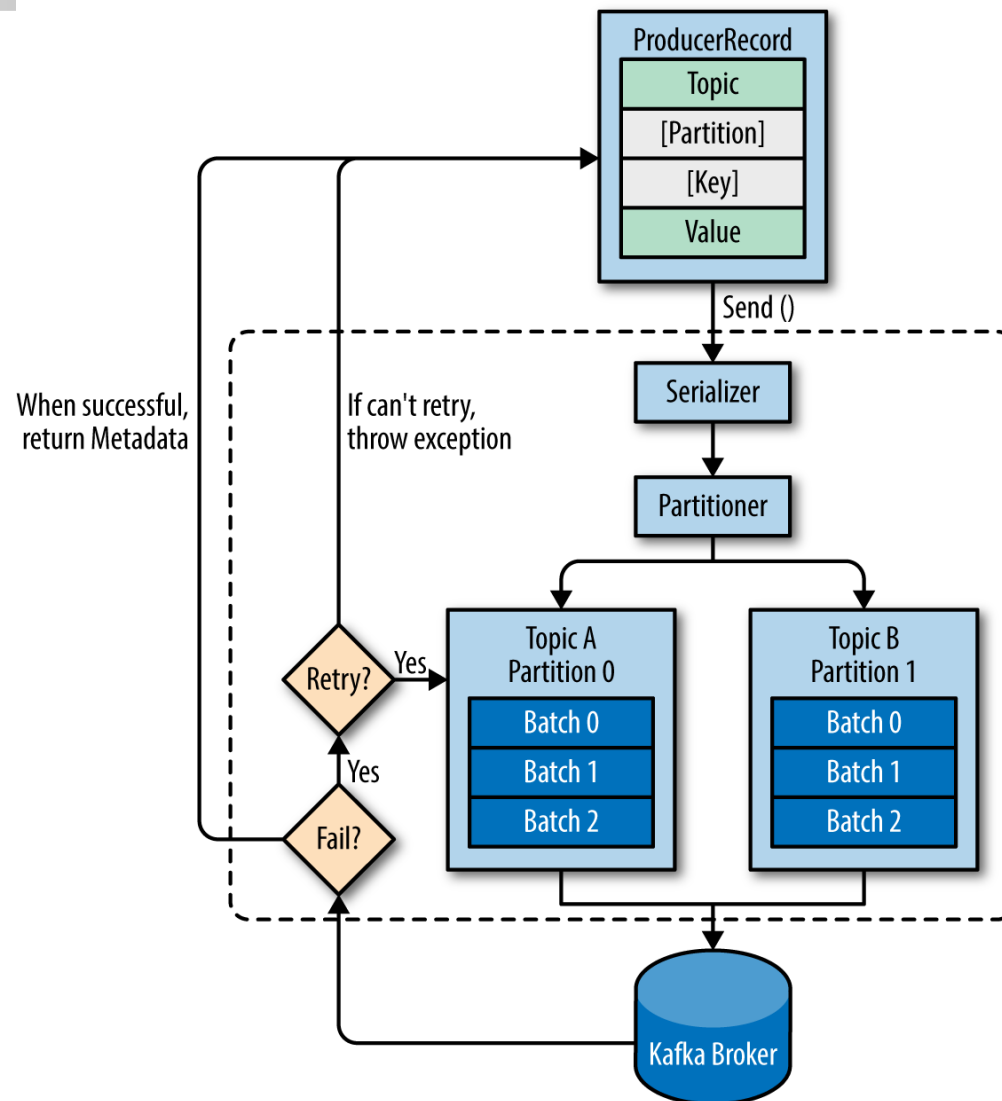
# Étapes lors de l'envoi d'un message

---

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 transmission sur le réseau
- Les données sont envoyées à un **partitionneur** qui détermine la partition de destination, soit à partir de la partition indiquée, soit de la clé du message, soit 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, 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 trouver le cluster
- ***key.serializer*** : La classe utilisée pour la sérialisation de la clé
- ***value.serializer*** : La classe utilisée pour la sérialisation du message ...





# Exemple

---

```
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");
kafkaProps.put("group.id", "myGroup");

producer = new KafkaProducer<String, String>(kafkaProps);
```



# *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

Quelques constructeurs :

- `ProducerRecord(String topic, V value)` : Sans clé
- `ProducerRecord(String topic, K key, V value)` : Avec clé
- `ProducerRecord(String topic, Integer partition, K key, V value)` : Avec clé et patition
- `ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)` : Avec clé, partition et timestamp



# 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

---

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry",  
    "Precision Products", "France");
```

```
try {  
    producer.send(record);  
} catch (Exception e) {  
    e.printStackTrace();  
}
```



# Envoi synchrone

---

```
ProducerRecord<String, String> record =  
new ProducerRecord<>("CustomerCountry",  
    "Precision Products", "France");
```

```
try {  
    producer.send(record).get();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```



# Envoi asynchrone

---

```
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());
```



# 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*



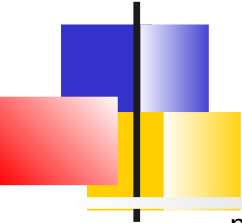
# Workflow

---

Le workflow de développement consiste à :

- Implémenter l'interface :  
*org.apache.kafka.common.serialization.Serializer*
- Implémenter l'interface :  
*org.apache.kafka.common.serialization.Deserializer*
- Pour Kafka Stream, Implémenter l'interface  
*org.apache.kafka.common.serialization.Serde*
  - manuellement
  - ou en utilisant des méthode utilitaire de *Serdes*  
comme :  
*Serdes.serdeFrom(Serializer<T>, Deserializer<T>)*





# 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

---

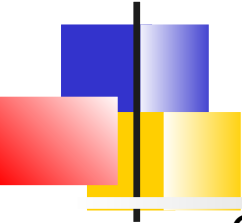
```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("value.serializer", "org.myappli.JsonPOJOSerializer");

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);
```

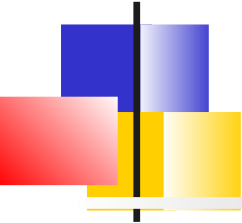


# Configuration des producteurs

---

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

- **acks** : contrôle le nombre de réplikas qui doivent recevoir l'enregistrement avant que le producteur puisse considérer l'écriture comme réussie
  - 0 : On attend pas
  - all : On attend toutes les répliques
- **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*
- **retries** : Si l'erreur renvoyée est de type *Retriable*, le nombre de tentative de renvoi. Si > 0 possibilité d'envoi en doublon



# Configuration des producteurs (2)

---

...

- ***client.id*** : Optionnel, pour permettre le suivi des messages
- ***max.in.flight.requests.per.connection*** : Maximum de message en cours de transmission (sans réponse obtenu)
- ***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 où 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

---

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

Attention :

- Si *retries* > 0 et *max.in.flights.requests.per.session* > 1 .  
Il se peut que lors d'un renvoi l'ordre initial soit inversé.

=> Si l'ordre et la fiabilité sont importantes, on configure généralement *retries* > 0 et *max.in.flights.requests.per.session* =1 au détriment du débit global



# Propriété Idempotent et transactions

---

A partir de Kafka 0.11, le *KafkaProducer* prend en charge 2 modes supplémentaires: idempotence et transactionnel.

- Un producteur **idempotent** garantit la livraison unique de chaque message (pas de doublon)
- Un producteur **transactionnel** permet à une application d'envoyer des messages à plusieurs partitions (et topics!) de façon atomique



# Idempotence

---

Pour autoriser l'idempotence :  
**`enable.idempotence=true`**

Dans ce cas,

- *retries* a par défaut *Integer.MAX\_VALUE*
- et *acks* a par défaut *all*.

L'idempotence n'a pas d'incidence sur le code applicatif.

Par contre, l'application ne doit pas essayer de renvoyer elle-même les messages en cas d'erreur





# 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*  $\geq 3$  et *min.insync.replicas* = 2
- Les consommateurs doivent eux être configurés pour faire du *read committed*
- L'API *send()* devient bloquante



# Exemple

---

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new
    StringSerializer());
```

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

```
try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>("my-topic", ""+i, ""+i));
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // Impossible de rattraper, la seule solution est de fermer le producer and exit.
    producer.close();
} catch (KafkaException e) {
    // Pour les autres exceptions, abort et essayer à nouveau.
    producer.abortTransaction();
}
producer.close();
```



# APIs

---

Producer API

**Consumer API**

Frameworks

Connect API

Admin et Stream API



# Introduction

---

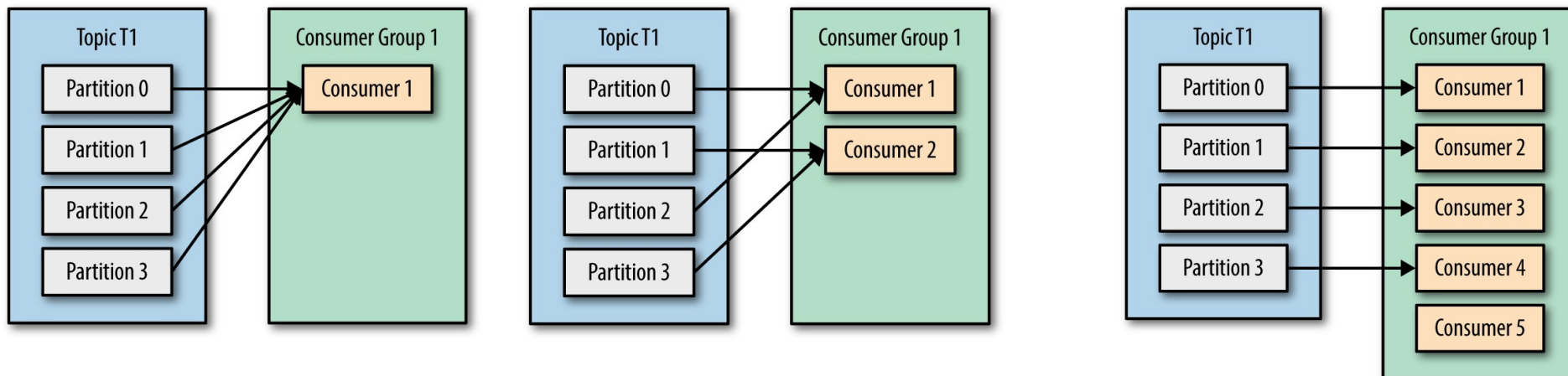
Les applications qui ont besoin de lire les données de Kafka utilisent un ***KafkaConsumer*** pour s'abonner aux topics Kafka

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 commence par consommer les messages d'une partition consommée précédemment par un autre consommateur du groupe.

Lors de l'arrêt d'un consommateur, la partition qui lui été 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 la réaffectation les messages ne sont pas consommés
- les consommateurs si ils utilisent des caches sont obligés de les rafraîchir



# Membership

---

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).

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





# Exemple

---

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



# 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 applications consommatrices *poll* continuellement les *topics* auxquels elles sont abonnées.

Les objets retournés par *poll* sont une collection de ***ConsumerRecord*** contenant en plus du message :

- La partition
- L'offset
- Le timestamp

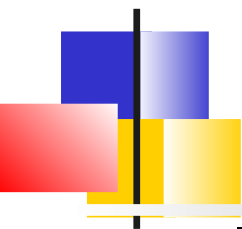


# Exemple

---

```
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) ;
        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString()) ;
    }
} finally {
    consumer.close();
}
```



# 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 plus importantes propriétés de configuration :

- ***fetch.min.bytes*** : Minimum volume 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 3s
- ***heartbeat.interval.ms*** : L'intervalle d'envoi des heartbeat durant la méthode *poll()*



# Configuration des consommateurs (2)

---

...

- ***auto.offset.reset*** : ***latest*** (défaut) ou ***earliest***.  
Contrôle le comportement du consommateur si il ne détient pas d'offset valide
- ***enable.auto.commit*** : Le consommateur commit les offsets automatiquement. Par défaut *true*
- ***partition.assignment.strategy*** : Stratégie d'affectation des partitions *Range* (défaut), *RoundRobin* ou *Custom*
- ***client.id*** : Une chaîne de caractère utilisé pour les métriques. Par défaut 3s
- ***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***

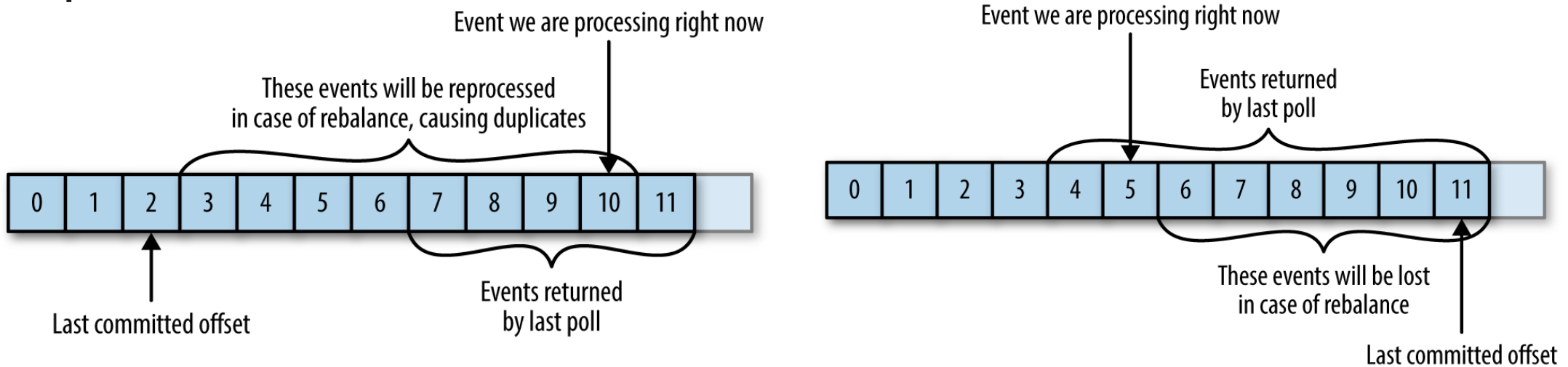
- Ce *topic* contient les offsets de chaque partition.

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()*

Lors d'un appel à *poll()*, le consommateur vérifie  
si il est temps de commit, si c'est le cas il  
committe les offsets du précédent appel à  
*poll()*

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



# 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

---

```
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)
    }
}
```



# 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



# Example

---

```
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);
        }
    });
}
```



# 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++;
    }
}
```



# 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 différentes façons pour 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");
}
```



# Consommateur standalone

---

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

**Frameworks**

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 Reactive 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.

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)



# 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();
    }
}
```





# 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+).



# 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)**

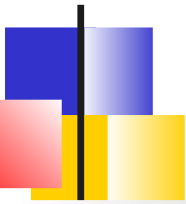
```
public class SampleSource {

    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 = "{\"value\":\"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:
        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.

=> Avec *Kafka Connect*, on peut ingérer des bases de données volumineuses, des données de monitoring avec des latences minimales



# Fonctionnalités

---

- Un cadre commun pour les connecteurs qui standardise 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

---

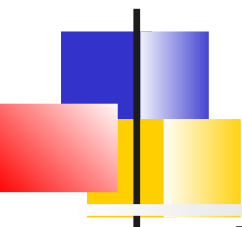
Pour démarrer *Kafka Connect* en mode standalone (single process)

```
> 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

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 démarre 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

---

En mode standalone, la configuration est défini dans un fichier *.properties*

En mode distribué, c'est via une API Rest et un format JSON que la configuration est mise à jour.

Les valeurs sont très dépendantes du type de connecteur.

Comme valeurs communes, on peut citer :

- ***name*** : Nom unique.
- ***connector.class*** : Classe Java du connecteur
- ***tasks.max*** : Maximum de tâches créées pour le connecteur. (degré de parallélisme)
- ***key.converter***, ***value.converter***

Les connecteurs de type *Sink* ont en plus :

- ***topics*** : Liste des topics servant d'entrée
- ***topics.regex*** : RegExp pour les topics d'entrée



# Connecteurs

---

*Confluent* offre de base les connecteurs suivants :

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

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



# 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 *transformer* prédéfinis, peut être 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 pour ce transformer



# Exemple de configuration

---

```
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=test.txt
topic=connect-test
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka.connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache.kafka.connect.transforms.InsertField$Value
transforms.InsertSource.static.field=data_source
transforms.InsertSource.static.value=test-file-source
```





# *Transformers disponibles*

---

***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

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

***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

---

Le serveur d'API REST peut être configuré via la propriété listeners

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

Ces listener 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*** : Client d'administration permettant de gérer et inspecter les topics, brokers, configuration et ACL
- ***Streams API*** : Librairie cliente pour des micro-services dont les entrées/sorties sont des *topics* Kafka



# 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);
    }
}
```



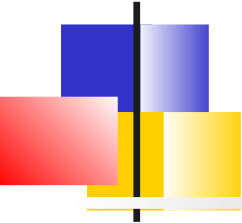
# Kafka Streams

---

Depuis la version 0.10.0, Kafka inclut dans son API une librairie de traitement de flux ***Kafka Streams API***

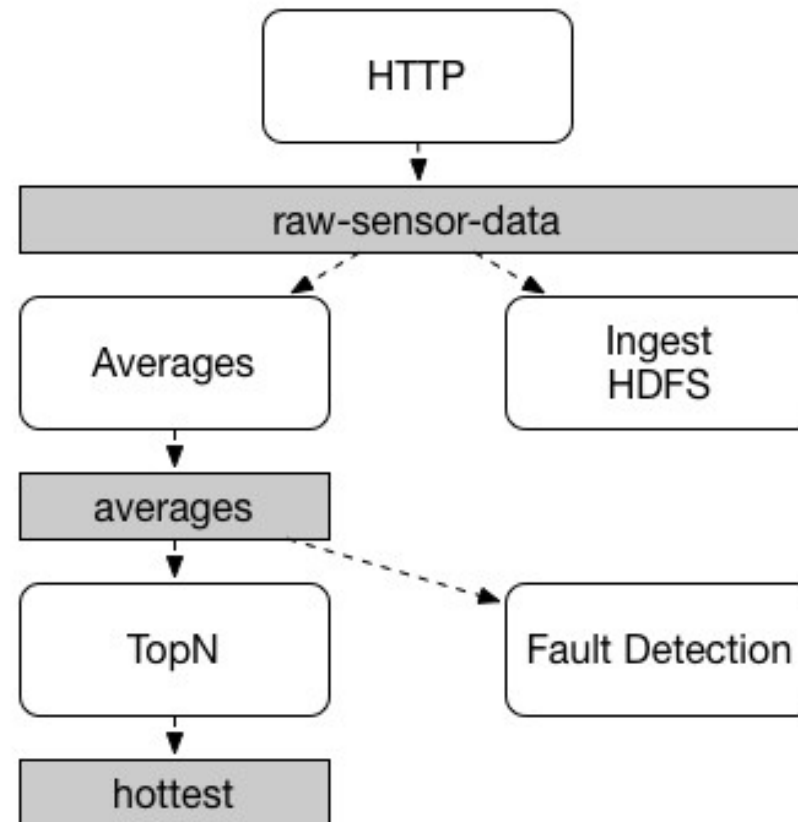
C'est une bibliothèque cliente pour la création d'applications et de micro-services, où les données d'entrée et de sortie sont stockées dans des clusters Kafka





# Exemple d'architecture

---





# Apports de *KafkaStream*

---

- Librairie simple et légère, facilement intégrable
- Seule dépendance celle d'Apache Kafka.
- Permet de conserver un état local tolérant aux pannes permettant des opérations d'agrégation et de jointures très rapides.
- 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 ***data stream*** 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



# Concepts Kafka

---

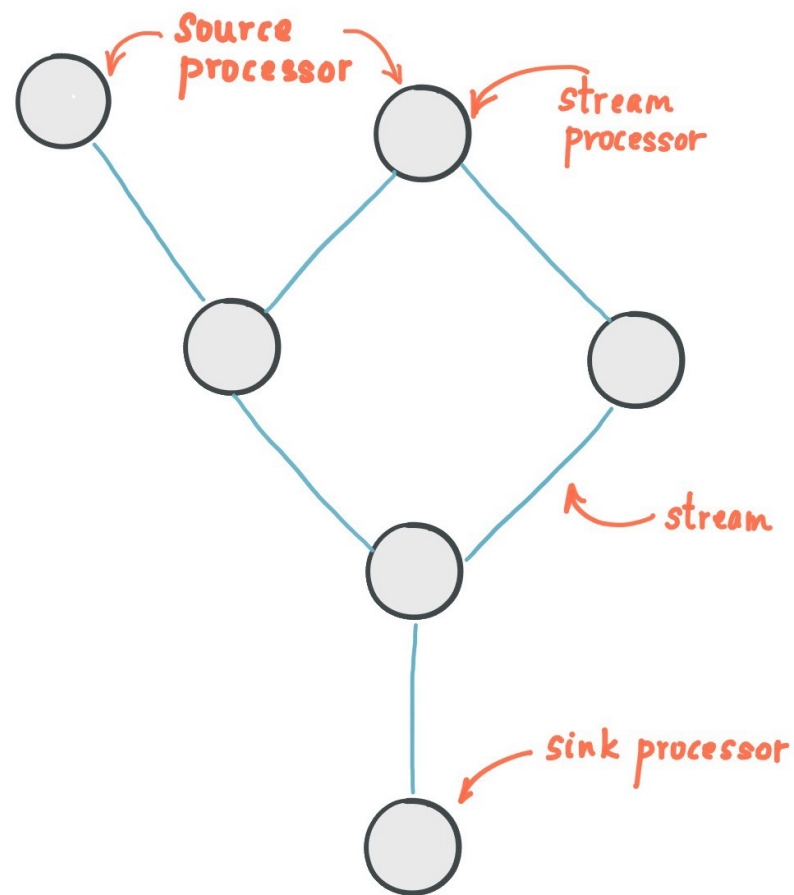
Une application de traitement de flux est un programme qui définit sa logique de calcul via une ou plusieurs **topologies de processeurs**, i.e. un graphe de processeurs de flux connectés

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

Un processeur effectue une transformation sur ces données d'entrée et alimente soit un *topic* Kafka, soit un système externe

# Topologie processeurs



PROCESSOR TOPOLOGY



# Flux et Table

---

En plus de l'abstraction *Stream*,  
*KafkaStream* fournit l'abstraction  
***Table*** permettant les agrégations.

- Il existe en fait une relation étroite entre les flux et les tables : une dualité

=> Un stream peut être vu comme une table, et une table peut être vue comme un stream.



# 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'agréations ou des jointures.

- Possibilité d'indiquer une *grace period* : période pendant laquelle les événements arrivant en retard sont pris en compte



# State store

---

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

*Kafka Streams* fournit des ***State stores*** qui permettent aux applications de stocker et requêter les données

- *KStream* offre des fonctionnalités de tolérance aux pannes et une récupération automatique.
- Les state stores expose des *interactive queries* permettant un accès en lecture aux données ;





# Garantie de traitement

---

2 principales garanties de traitement accessibles par simple configuration de *KafkaStream*

- ***At-least-one*** (défaut) : Tout événement est traité au moins 1 fois. Tolérance vis à vis des doublons de traitement
- ***Exactly-once*** : S'appuie sur les capacités de transactions et d'idempotence de Kafka



# Partition de flux

---

Kafka Streams utilise les concepts de **partitions de flux** basé sur les partitions des topics Kafka.

- Chaque partition de flux est une séquence totalement ordonnée d'événement et correspond à une partition d'un topic Kafka.
- Un événement dans le flux correspond à un message Kafka du topic.
- Les clés des événements déterminent le partitionnement des données dans les flux et comment les données sont acheminées vers des partitions spécifiques dans les topics.



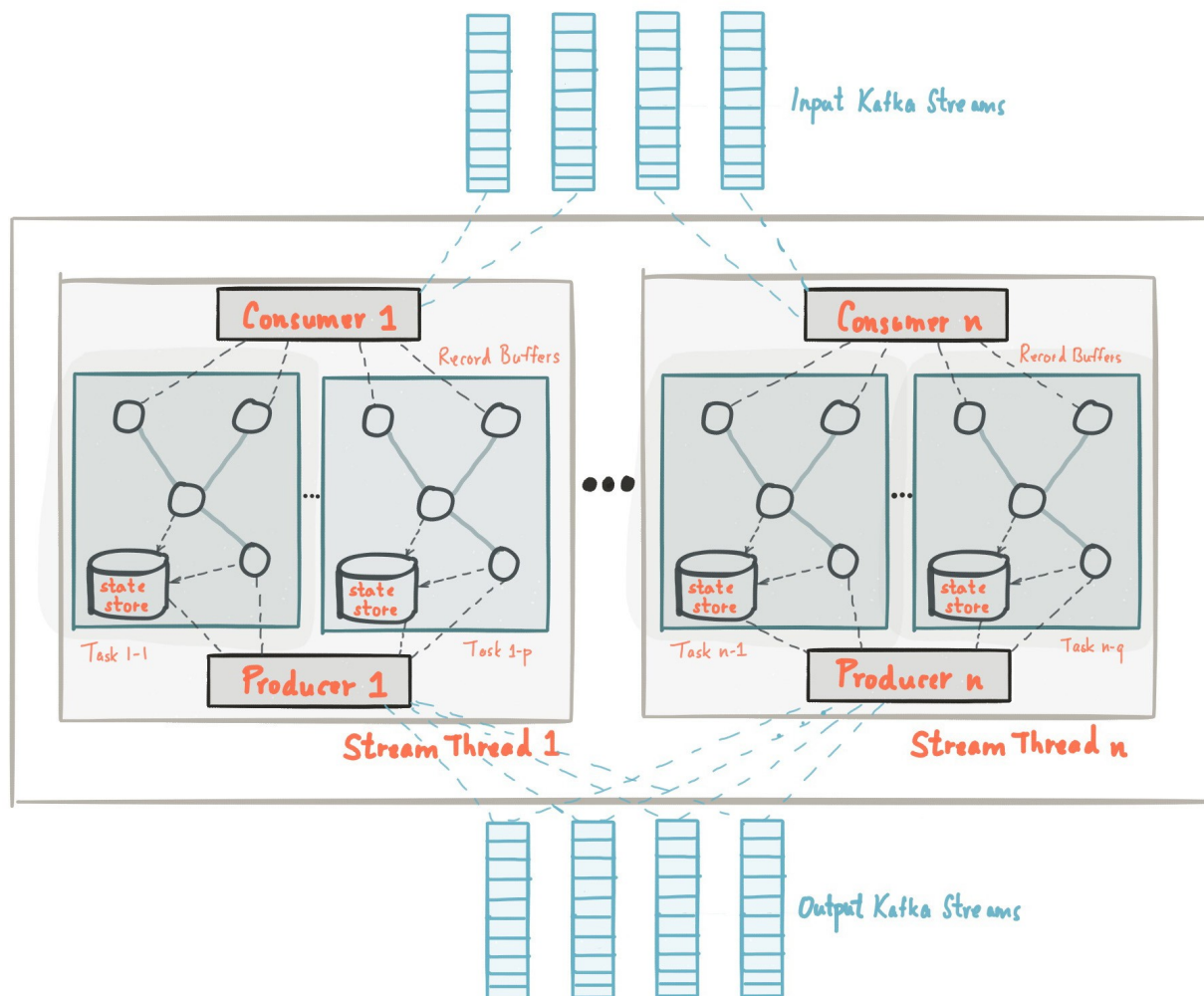
# Tâches

---

La scalabilité d'une topologie de traitement de flux est assuré par les **tâches**

- *Kafka Streams* crée un nombre fixe de tâches en fonction des partitions de flux d'entrée pour l'application, chaque tâche se voit attribuer une liste de partitions à partir des flux d'entrée.
- L'affectation de partitions aux tâches ne change jamais, chaque tâche est une unité fixe de parallélisme de l'application
- Les tâches conservent un buffer tampon pour chacune des partitions qui leur sont affectées et traitent les messages un par un à partir de ces buffers

# Architecture





# 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);
```



# Réplication et Fiabilité

---

## **Stockage des partitions**

Mécanismes de réplication

Rôle du contrôleur

*At Most Once, At Least Once*

Exactly Once



# 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 répertoire de stockage des partitions





# Allocation des partitions

---

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

Ces objectifs sont :

- Répartir uniformément les répliques entre les courtiers
- 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***

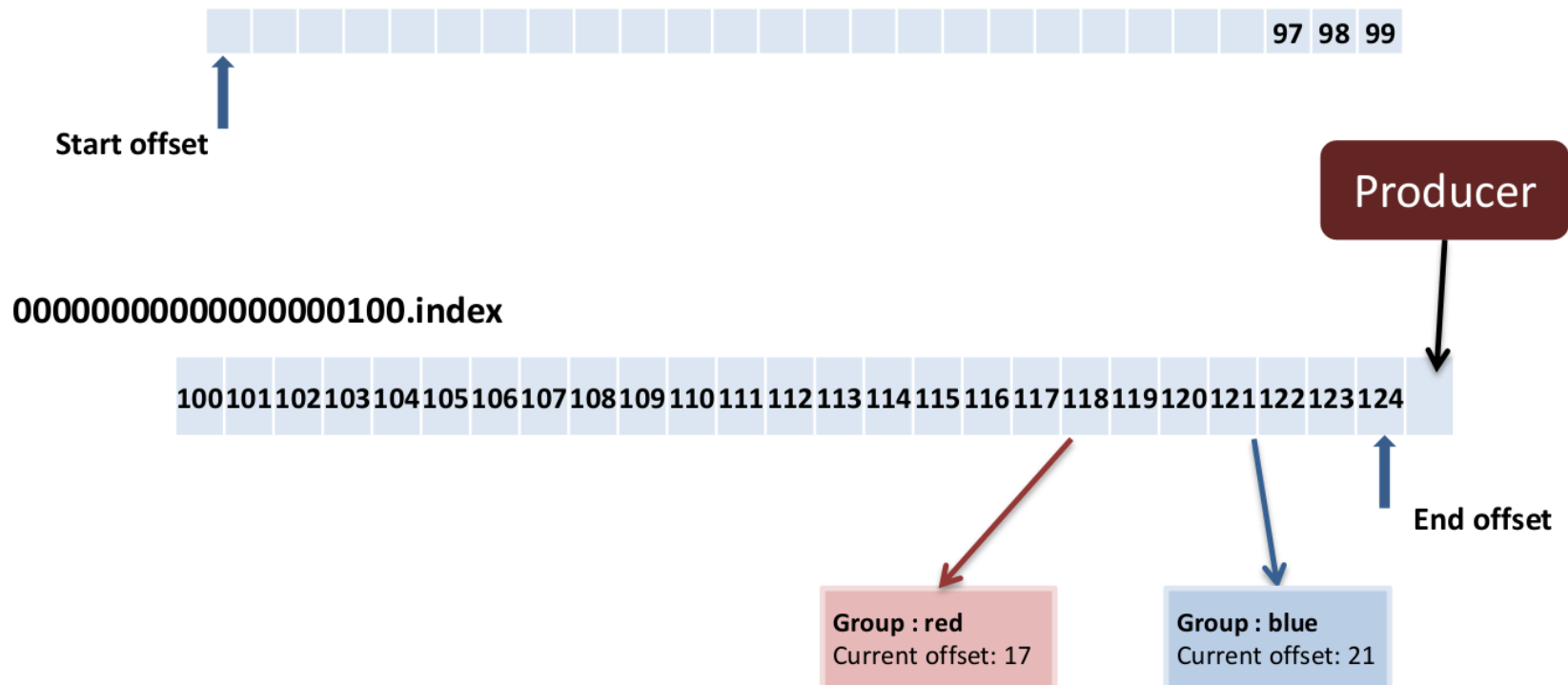
- Les segments sont des fragments de partition au maximum de 1Go et au maximum d'une semaine de données
- Lors de l'écriture d'un segment, lorsque la limite est atteinte, le broker ferme le fichier. Il devient éligible pour la purge
- Il existe donc un seul *segment actif* qui d'ailleurs ne peut pas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

- Propriété ***log.cleaner.enabled***
- Les événements doivent alors contenir une clé
- Le compactage est effectué par une thread séparé qui périodiquement purge les messages *dirty*



# Réplication et Fiabilité

---

Stockage des partitions

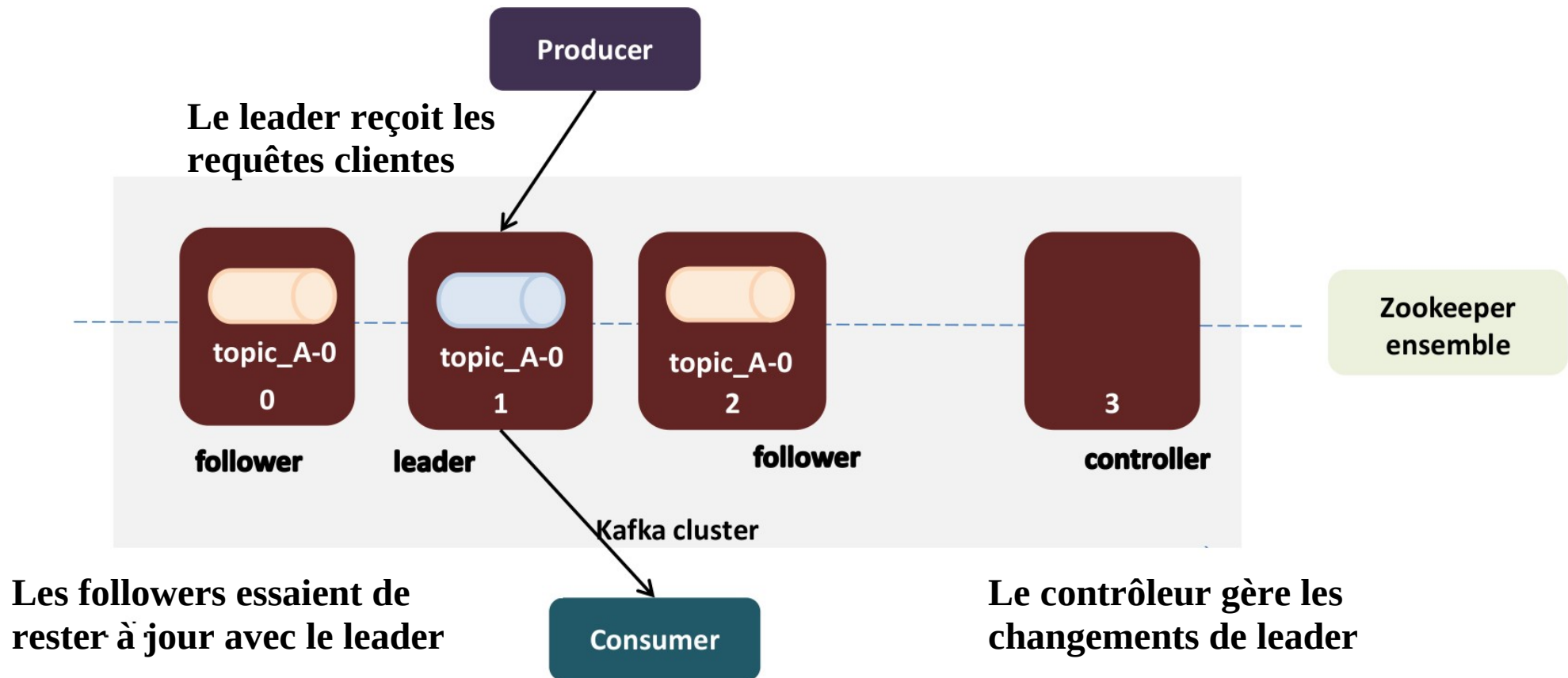
**Mécanismes de réplication**

Rôle du contrôleur

*At Most One, At Least One*

Exactly Once

# Rôle des brokers gérant un topic







# Synchronisation des répliques

---

Contrôlé par la propriété :

***replica.lag.time.max.ms*** (défaut 10 000)

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)



# Les rôles des brokers

---

## **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
- Gère les changements de leader
- Envoie le nouveau leader et tous les ISRs à tous les brokers
- Persiste le nouveau leader et l'ISR vers *Zookeeper*



# 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 écrit sur la partition et toutes les répliques en synchronisation
- Les messages validés ne seront pas perdus tant qu'au moins une réplique reste en vie.
- Les consommateurs ne peuvent lire que les messages validés.

=> Il y a des compromis à faire pour construire un système fiable, les administrateurs et développeurs doivent décider de la fiabilité dont ils ont besoin au détriment du débit, de la latence et du coût.



# In-Sync

---

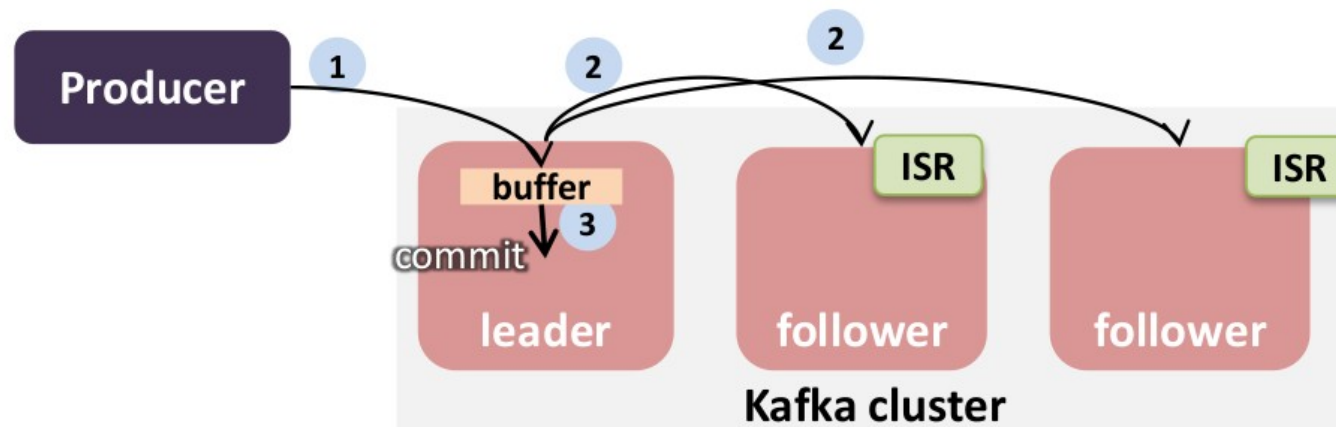
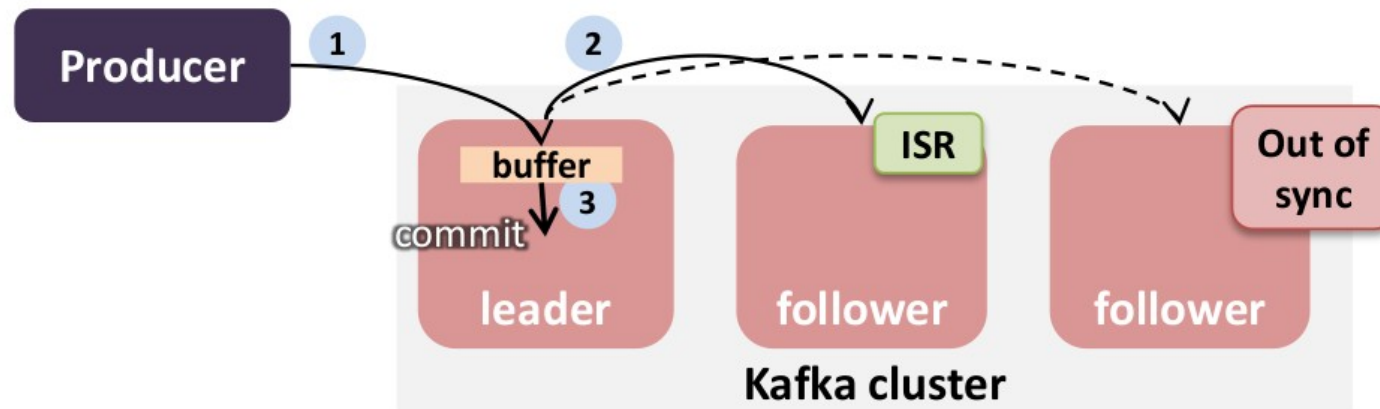
Les répliques doivent rester *in-sync* avec le leader

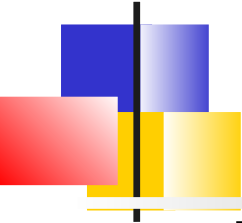
Les conditions sont :

- Avoir une session active avec ZooKeeper (heartbeat par défaut 6s)
- Avoir récupéré des messages du leader dans les dernières 10s (par défaut)
- Avoir récupéré les messages les plus récents du leader au cours des 10 dernières secondes. Pas de décalage

Un réplique désynchronisée se re-synchronise lorsqu'elle se connecte à nouveau à Zookeeper et rattrape son décalage

# Réplication et commit



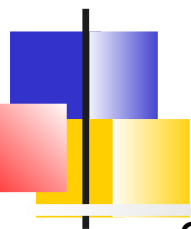


# Conséquences

---

Un réplique synchronisée légèrement en retard peut ralentir les producteurs et les consommateurs, car ils attendent qu'un certain nombre de répliques synchronisées reçoivent le message avant de le valider.

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** < ***min.insync.replicas*** :

- Kafka empêche l'acquittement

Si le **nombre de réplique disponible** < ***min.insync.replicas***

- 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



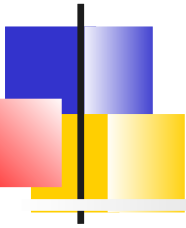
# Configuration

---

3 principales configuration affecte 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
- ***unclean.leader.election.enable*** (défaut false, au niveau cluster)  
Autorise les répliques non présentes dans l'ISR de devenir un leader,  
Compromis entre pertes de données potentielles (true) et plus basse disponibilité (false)





# Fichiers de point de contrôle

---

## ***recovery-point-offset-checkpoint***

- Indique les derniers offsets qui ont été flushés de la mémoire vers le disque

## ***replication-offset-checkpoint***

- “high water mark”, utilisé par le leader pour persister le dernier offset validé
- Indique le dernier offset pouvant être consommé

## ***leader-epoch-checkpoint***

- Le broker (ancien leader) devient un follower et utilise ce point de contrôle pour tronquer les données non validées



# Réplication et Fiabilité

---

Stockage des partitions

Mécanismes de réplication

**Rôle du contrôleur**

*At Most Once, At Least Once*

*Exactly Once*



# 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 le leader est ce broker, 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



# Réplication et Fiabilité

---

Stockage des partitions  
Mécanismes de réplication  
Rôle du contrôleur

***At Most Once, At Least Once***  
*Exactly Once*



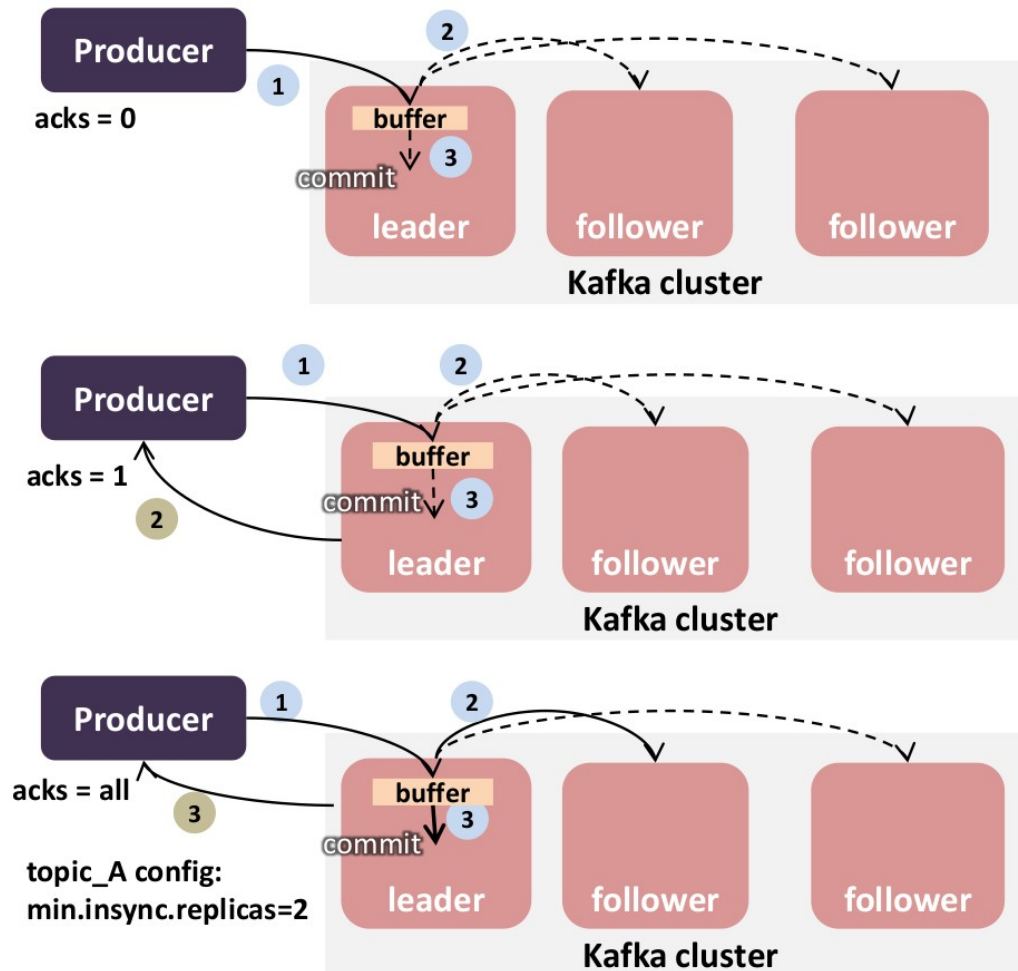
# Côté producteur

---

Du côté producteur, 2 premiers facteurs influencent la fiabilité

- La configuration des **ack**
- Les gestion des **erreurs** dans la configuration et dans le code

# 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)  
Nombre d'essai configurable via **retries**.  
=> Attention, peut générer des doublons
- les erreurs qui doivent être traitées par le code. (ex : INVALID\_CONFIG)





# 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 offset qu'ils ont traités en cas de crash.

Pour cela, ils committent leur offset auprès du cluster Kafka.

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

Visualisation des offsets pour un groupe :

```
./kafka-consumer-groups.sh --bootstrap-server  
localhost:9092 --describe --group <my-group>
```



# 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

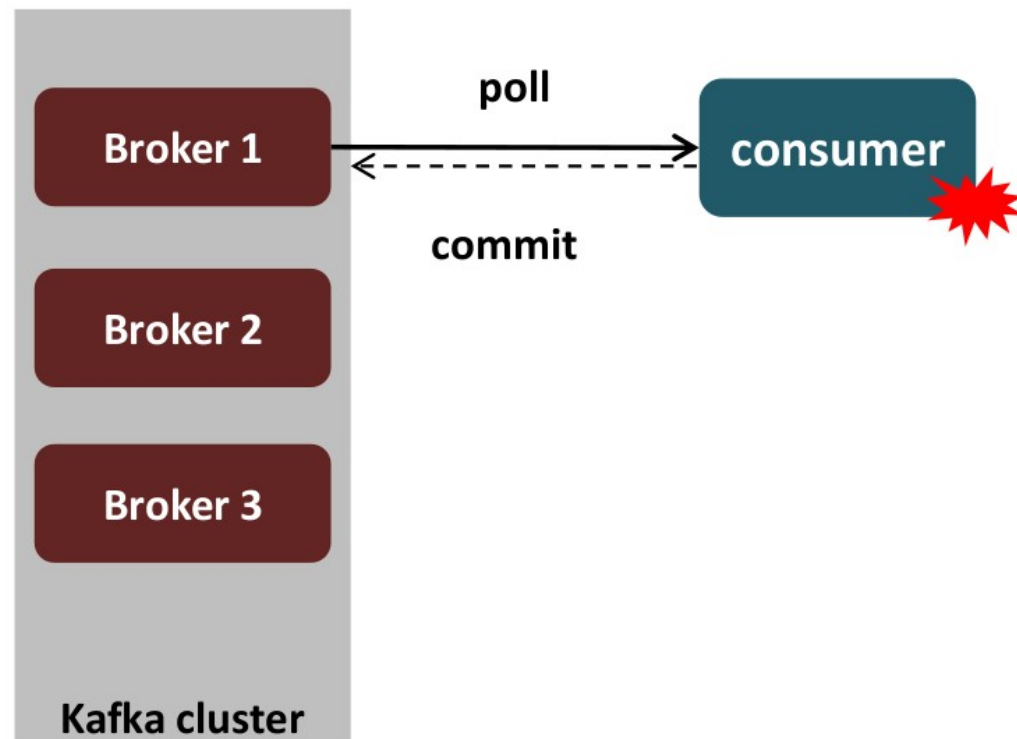
***enable.auto.commit*** : Commit manuel ou non

- 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

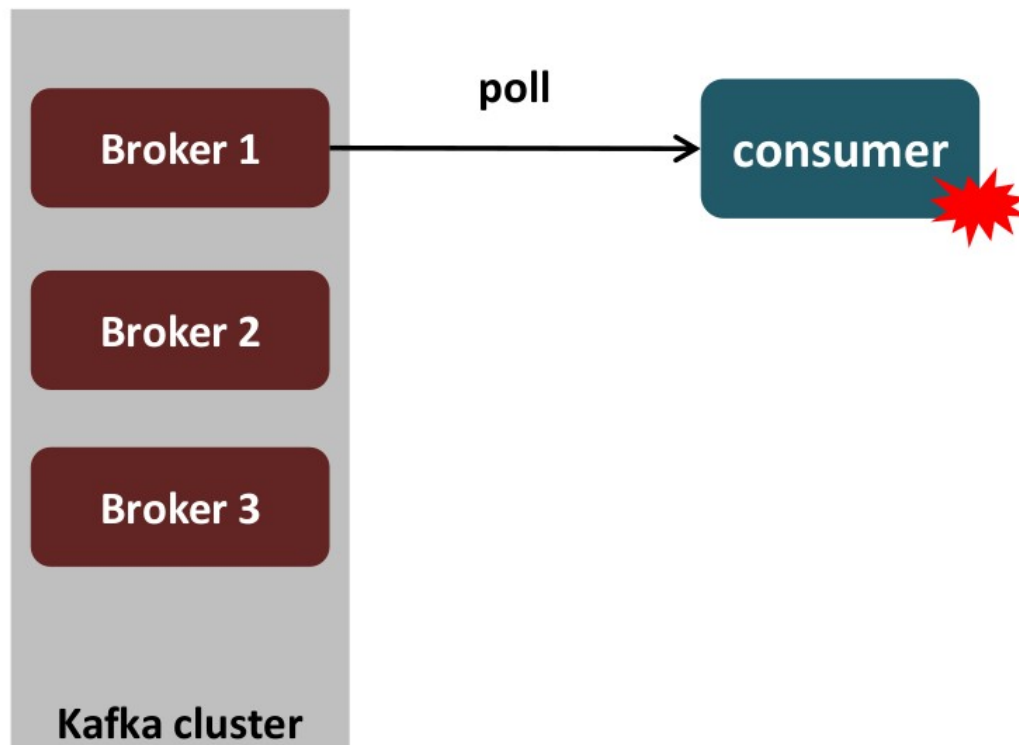
# Réception : At Most Once



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



# Réception : *At Least Once*



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



# Configuration *At Least One*

---

`enable.auto.commit` à false

Ou

`enable.auto.commit=true` ET '`auto.commit.interval.ms`' à une valeur haute .

Commit explicite après traitement via `consumer.commitSync()`;

```
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();  
}
```

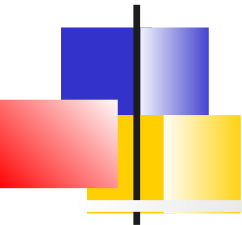


# Commits manuels

---

## Recommandations :

- Committer après que les messages aient été traités
- La fréquence de commit est un compromis entre les performances et le nombre de doublon potentiel
- Gérer les réaffectations de partition (rebalance)
- Éventuellement, maintenir un état côté consommateur
- Lors de long traitement, continuer à appeler régulièrement *poll()* afin que le client continue à envoyer ses heartbeats (Déprécié depuis 0.10.1.0)



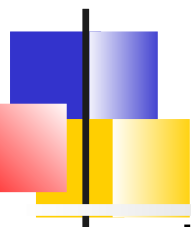
# Validation de la fiabilité

---

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 cadence des messages
- ***kafka-verifiable-consumer.sh*** consomme les messages et les affiche dans l'ordre de consommation. Il affiche également des information sur les commit et les rééquilibrage

On peut alors exécuter ses commandes pendant différents scénarios de test : Election de leader, de contrôleur, redémarrage des brokers, unclean leader election, ...





# Monitoring

---

Kafka propose des métriques JMX.

Pour la fiabilité du producteur :

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

Pour la fiabilité du consommateur :

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



# Réplication et Fiabilité

---

Stockage des partitions  
Mécanismes de réplication  
Rôle du contrôleur  
*At Most Once, At Least Once*  
***Exactly Once***



# Introduction

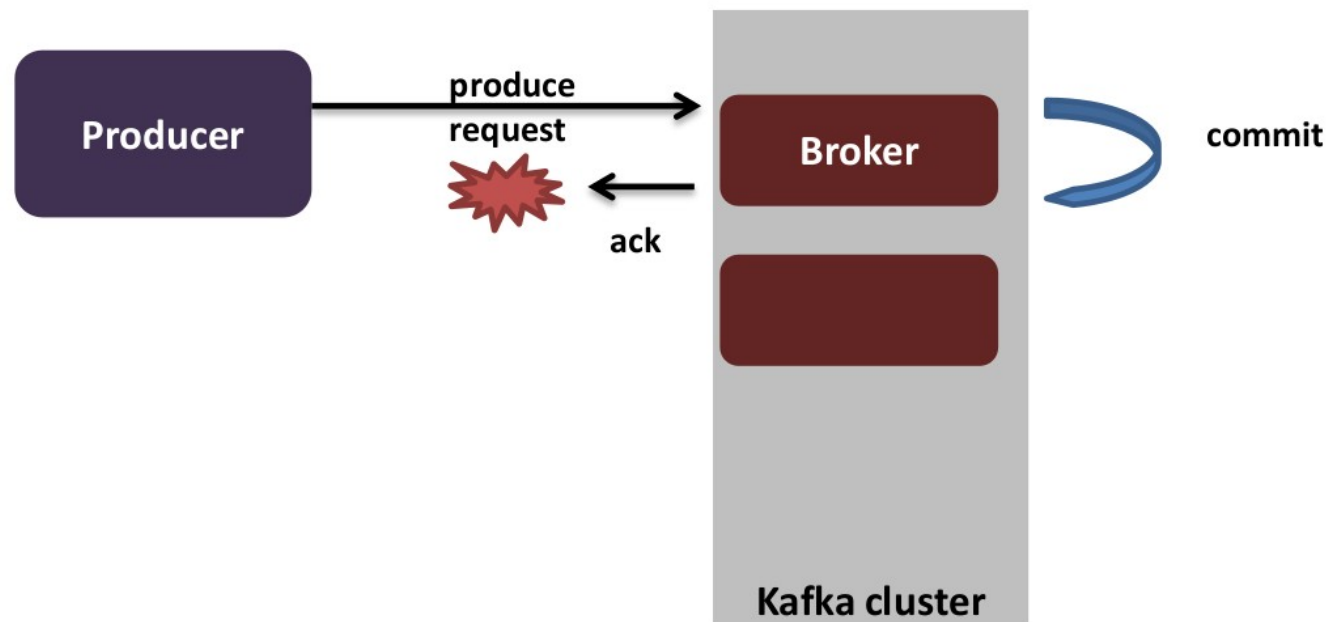
---

Avant la version 0.11, seule la sémantique «au plus une fois» ou «au moins une fois» était prise en charge par Kafka

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* (défaut false)

L'idempotence nécessite les configurations suivantes :

- *max.in.flight.requests.per.connection*  $\leq 5$
- *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 les offsets des partitions dans un support de persistance transactionnel et partagé par tous les consommateurs
- Gérer les rééquilibrages



# 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

---

Permet la production atomique de messages sur plusieurs partitions

Les producteurs produisent l'ensemble des messages ou aucun

```
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)



# 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 1)

## 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 1)
- *enable.idempotence:true* (défaut false)
- *max.in.flight.requests.per.connection*:  $\leq 5$

## Consommateur

- *isolation.level*: *read\_committed*



# Administration

---

## **Gestion des topics**

Gestion du cluster

Sécurité

Dimensionnement

Surveillance



# *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 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 --zookeeper localhost:2181  
--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



# Etape 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
```





# 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



# 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

---

Le *nettoyeur* (threads) est basé sur :

- *log.cleaner.backoff.ms* (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  |



# Conséquences stratégie *compact*

---

Consommateurs «à jour» (offset courant dans le segment actif), récupèrent tous les messages

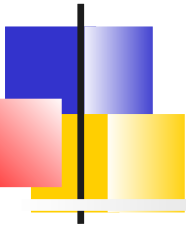
- Le segment actif n'est pas compacté

=> Le compactage n'empêche pas le consommateur de lire les données en double

L'ordre des messages est sauvegardé

Les offsets des messages sont sauvegardés

Les messages supprimés sont toujours disponible pour les consommateurs actif jusqu'à ce que *delete.retention.ms* soit expiré (24h par défaut)



# Paramètres de compactage

---

***segment.ms*** (7 jours)

***segment.byte*** (1G)

***min.compaction.lag.ms*** (défaut 0)

***delete.retention.ms*** (défaut 1 jour)

***min.cleanable.dirty.ratio*** (défaut 0.5) : réduire pour un nettoyage plus efficace



# Administration

---

Gestion des topics  
**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
```

Outillage :

<https://kafka-utils.readthedocs.io/en/latest/>





# 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

Gestion du cluster

**Sécurité**

Dimensionnement

Surveillance



# Introduction

---

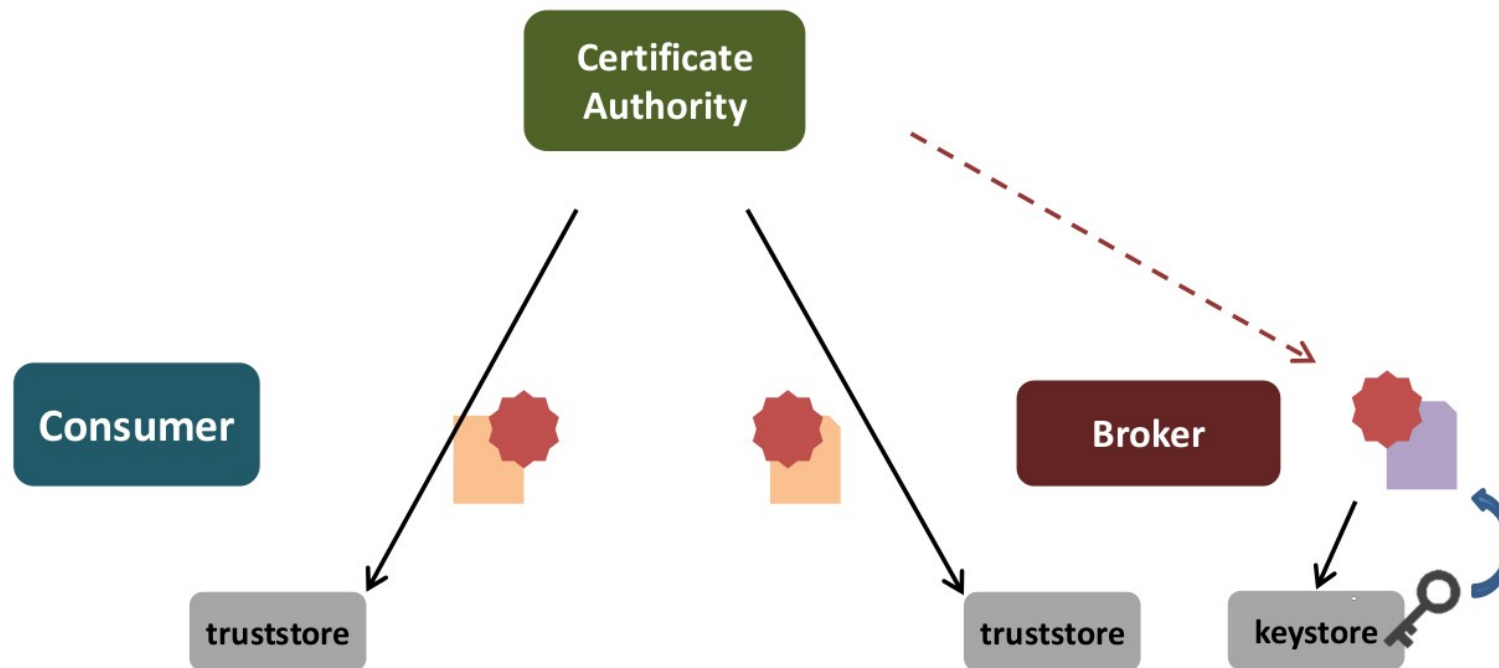
Différentes alternatives supportées pour la sécurité :

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

Naturellement, dégradation des performances avec SSL

# 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
```





# SASL

---

*Simple Authentication and Security  
Layer pour **l'authentification***

Mécanismes:

- GSSAPI: Kerberos
- SCRAM-SHA-256, SCRAM-SHA-512:  
hashed passwords
- PLAIN: username/password en clair



# 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
```



# Administration

---

Gestion des topics

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 compactiage)

- 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

---

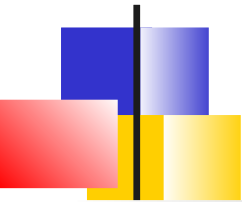
Gestion des topics

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

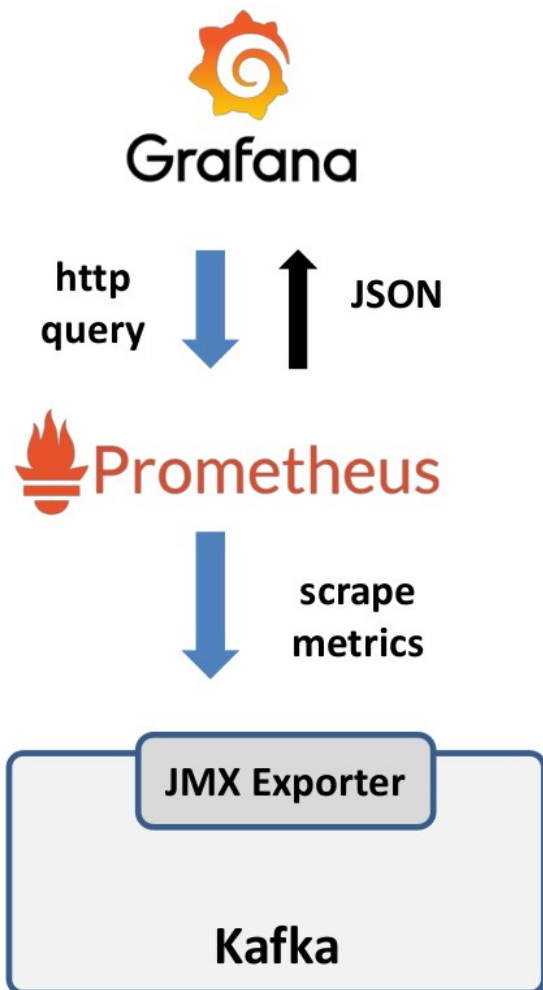
| Métrique  | Description                                       |
|---|---|
| kafka.producer:type=producermetrics,client-id=([-w]+),name=io-ratio       | Fraction de temps de la thread passé dans les I/O |
| kafka.producer:type=producer-metrics,client-id=([-w]+),name=io-wait-ratio | Fraction de temps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

...