



# Kafka Administration

---

David THIBAU – 2023

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



# Agenda

---

## Introduction à Kafka

- Le projet Kafka
- Cas d'usage
- Concepts

## Cluster Kafka

- Cluster
- KRaft
- Distributions / Installation
- Utilitaires Kafka
- Outils graphiques

## Clients Kafka

- Types de clients
- Production/Consommation de messages
- Schema registry
- Mécanismes de réplication / Garanties de livraison
- Latence, Débit, Durabilité
- Kafka Connect

## Sécurité

- Configuration des listeners
- SSL/TLS
- Authentification via SASL
- ACL

## Exploitation

- Gestion des topics
- Stockage et rétention des partitions
- Quotas
- Gestion du cluster
- Dimensionnement
- Monitoring

## Annexes



# Introduction à Kafka

---

## **Le projet Kafka**

Cas d'usage

Concepts



# Origine

---

Initié par *LinkedIn*, mis en OpenSource en 2011

Écrit en *Scala* et *Java*

Au départ, un message broker gérant une file de messages

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

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

Maintenu par *Confluent* depuis 2014



# Objectifs

---

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



# Fonctionnalités

---

Kafka a trois capacités clés:

- Publier et s'abonner à des flux de messages<sup>1</sup> avec certaines garanties de fiabilité.
- Stocker les flux de messages de manière durable et tolérante aux pannes.
- Traiter, transformer les flux de messages au fur et à mesure qu'ils se produisent.

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



# Points forts

---

Très bonne scalabilité et flexibilité

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

Très bonne performance. Capable de traiter des millions de message par secondes

Disponibilité et tolérance aux fautes

Rétention sur disque

Traitement distribué d'évènements

Intégration avec les autres systèmes



# Usage

---

## Suivi d'activité

- Cas d'utilisation d'origine
- Suivi des interactions des utilisateurs avec les applications frontend

## Messagerie

- Échanger des messages entre applications (systèmes de découplage)

## Métriques et journalisation

- Buffering des données utilisés par les systèmes de surveillance ou d'alerte

## Commit log

- Rétention durable pour bufferiser le change log (ex : event sourcing)

## Traitement de flux

- Transformer, agréger, enrichir, ... des données





# Confluent

---

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

Mainteneur principal d'Apache Kafka

Plate-forme *Confluent* :

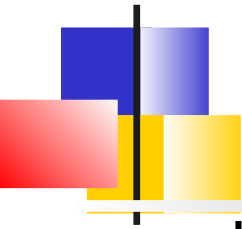
- Une distribution de Kafka
- Fonctionnalités commerciales additionnelles



# Introduction à Kafka

---

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



# Kafka vs Message broker traditionnel

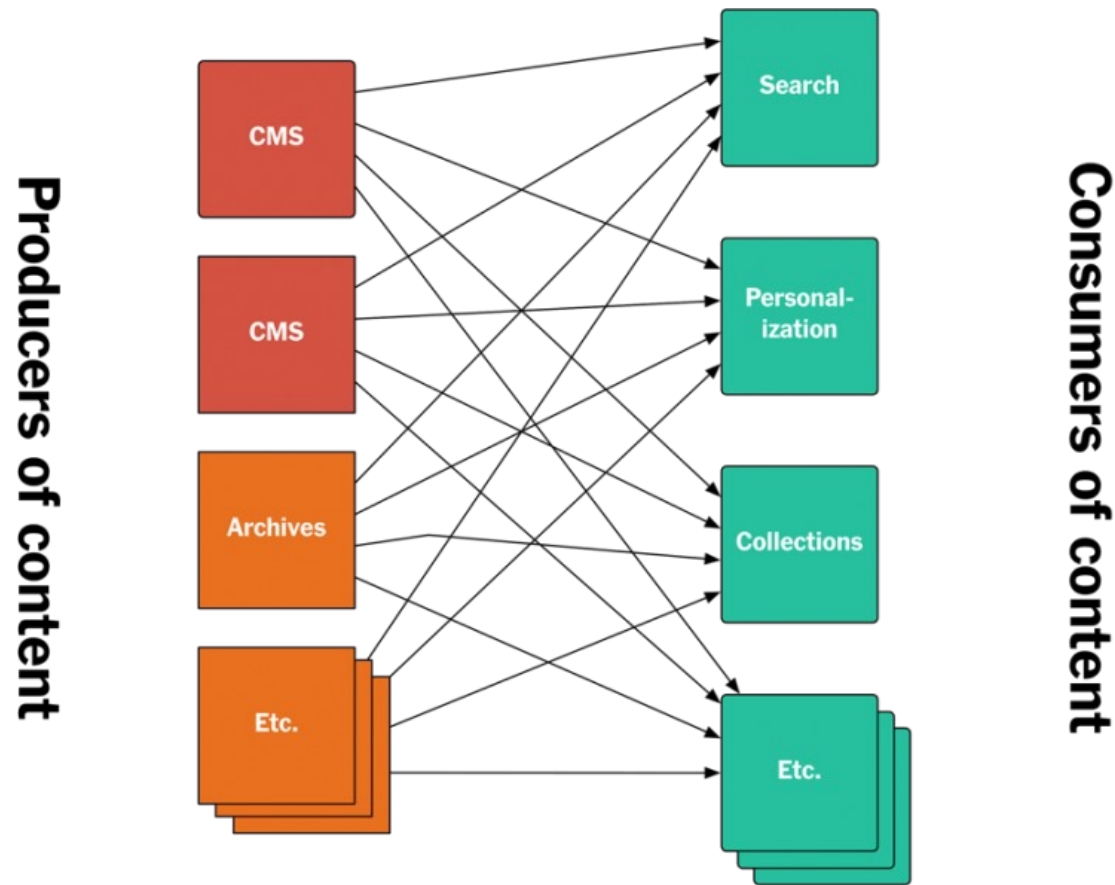
---

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

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

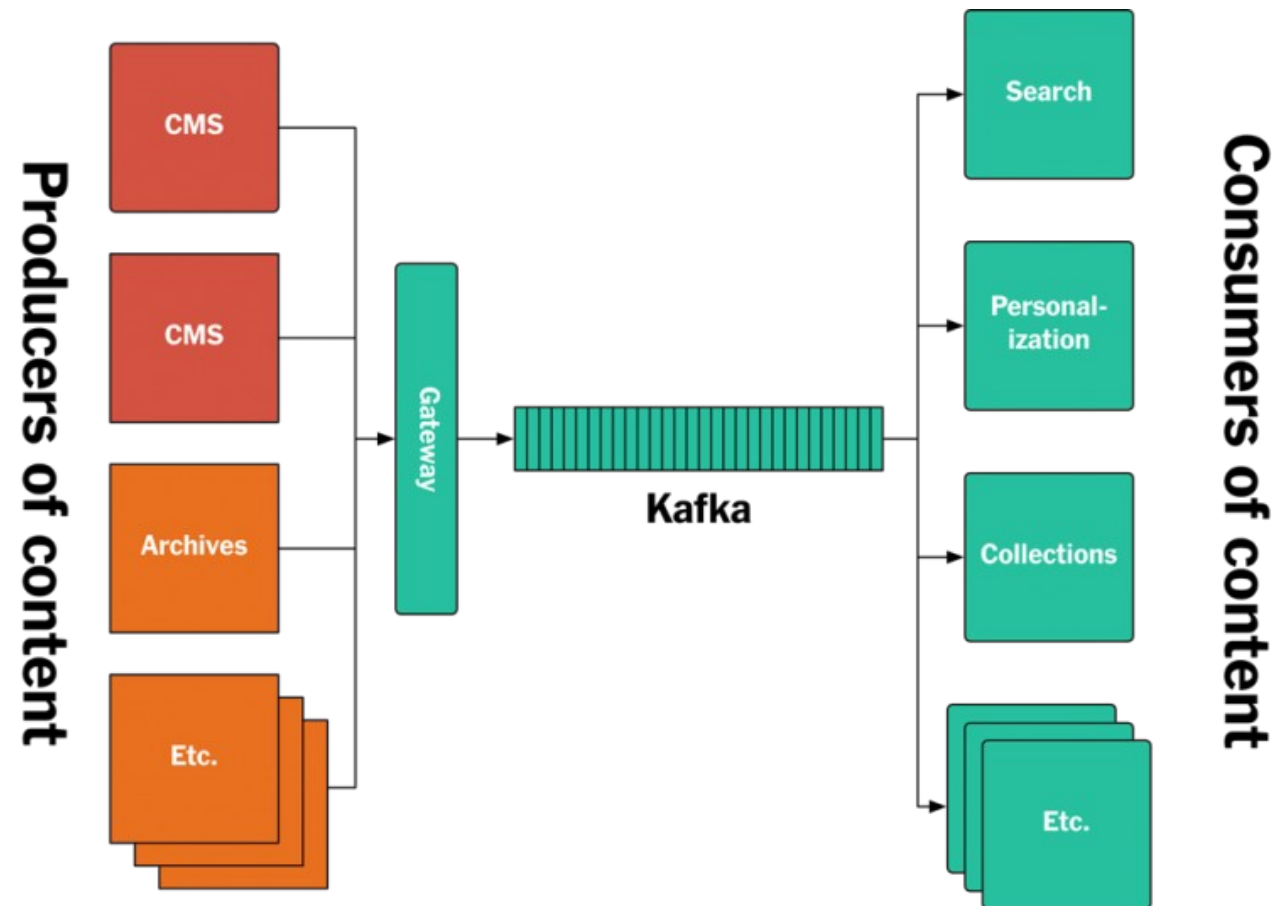
=> Idéal comme plate-forme d'intégration entre services : Architecture micro-services, ESB

# Exemple ESB (New York Times) *Avant*



# Exemple ESB (New York Times)

## *Après*





# Exemple : micro-services

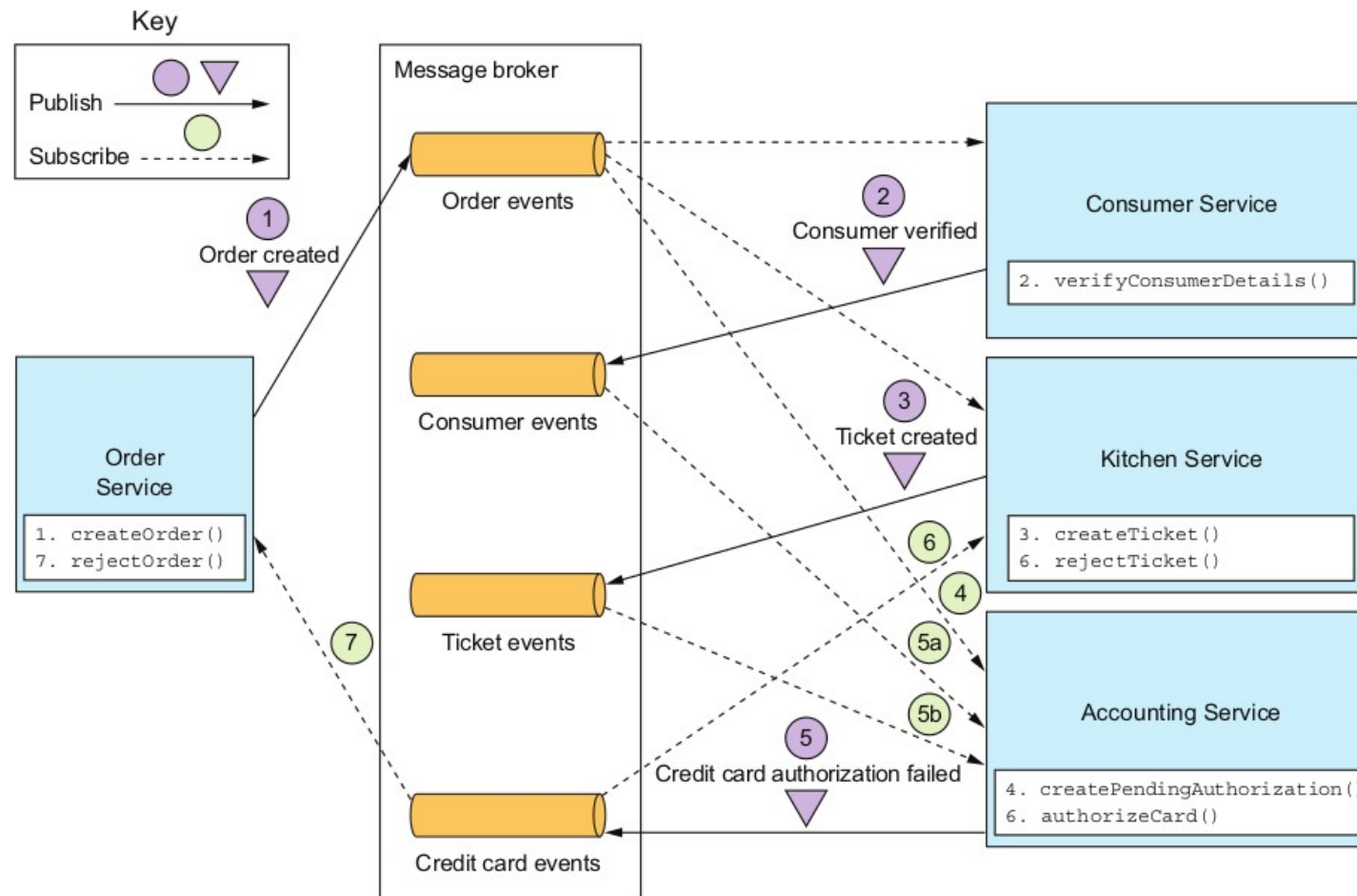
---

Un message broker est souvent utilisé pour permettre les communications entre les services d'une architecture micro-services

- Permet tous les styles d'interaction :
  - Requête/Réponse synchrone ou asynchrone,
  - One way notification,
  - PubAndSub synchrone ou asynchrone
- Certains patterns micro-services sont implémentés via un message broker (exemple SAGA<sup>1</sup>)

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

# Exemple Micro-services Message Broker et SAGA





# Kafka : Métriques et journalisation

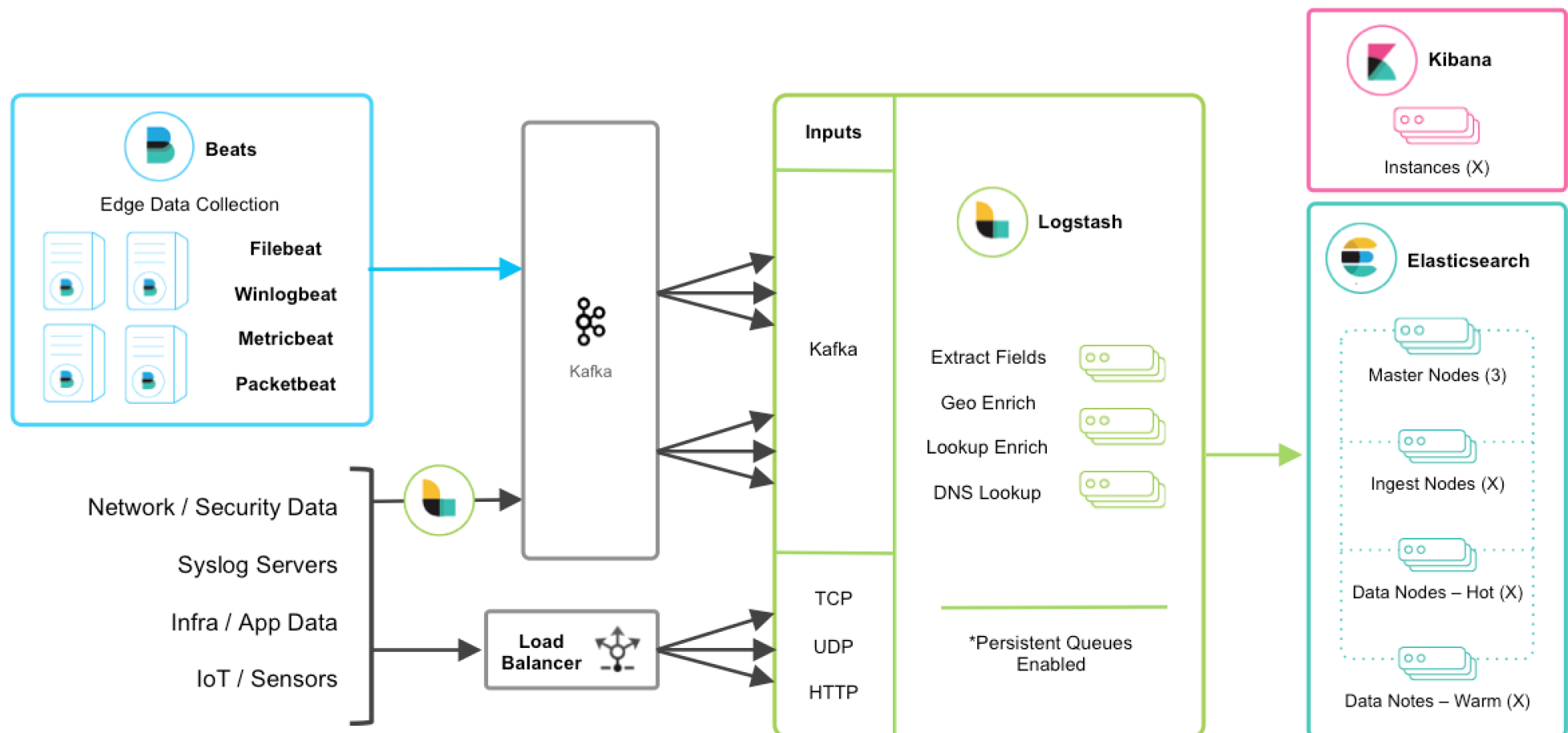
---

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

- Ces événements alimentent des pipelines de traitement et transformation destinées à des solutions d'analyse temps-réel, d'alerting ou de machine learning

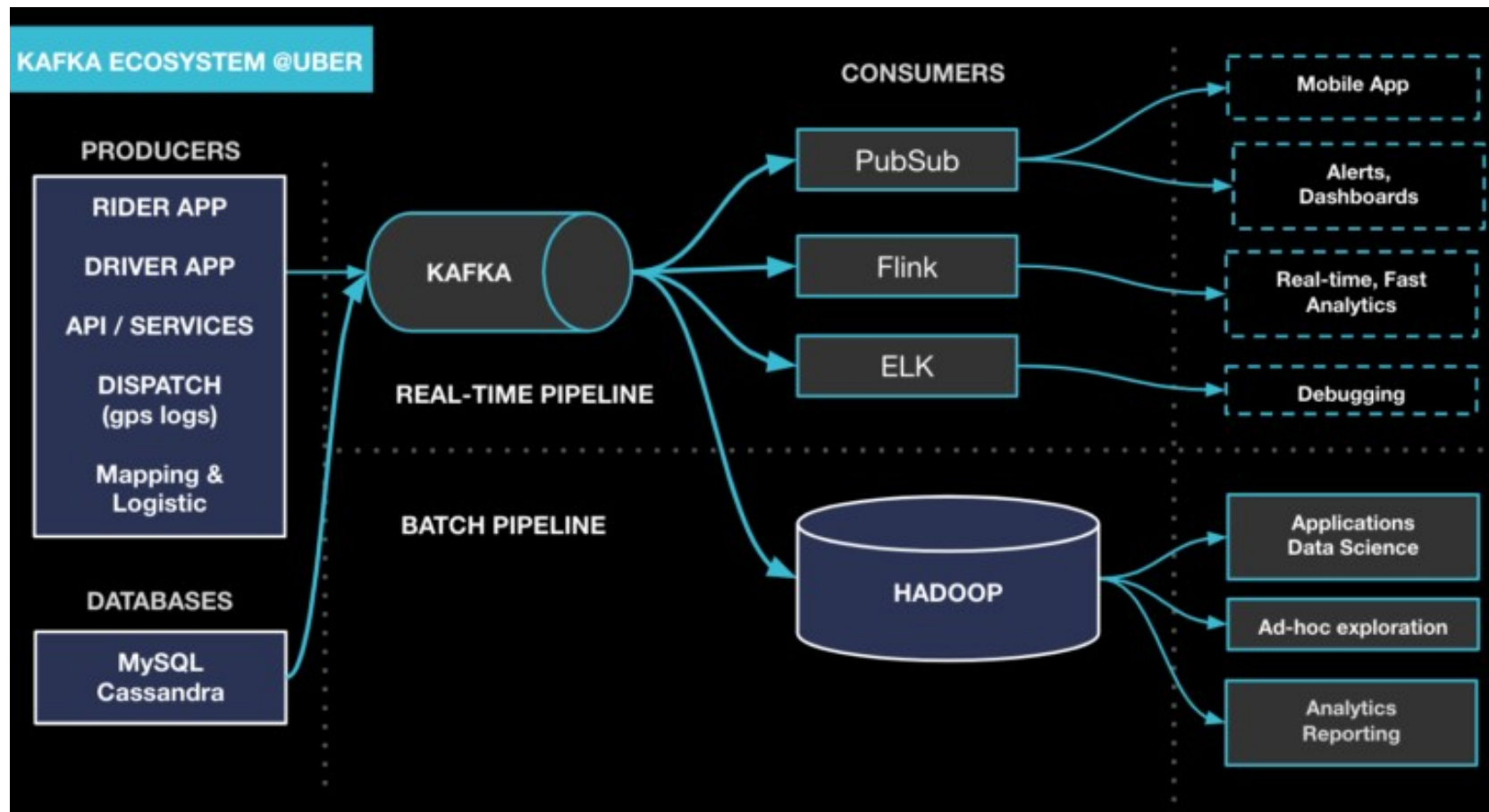


# Exemple Architecture ELK Bufferisation des événements



# Ingestion massive de données

## Exemple Uber





# Kafka comme système de stockage d'évènements

---

Les enregistrements sont écrits et répliqués sur le disque.

La structure de stockage est très scalable.

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

=> Kafka peut être considéré comme un système de stockage d'évènement et permet la mise en place du pattern *Event Sourcing*<sup>1</sup> comme architecture d'application

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



# Event sourcing Pattern

---

***Event sourcing Pattern***<sup>1</sup> : Persiste un agrégat comme une séquence d'événements représentant les changements d'état

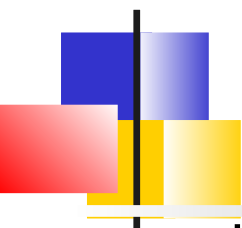
=> Les application peuvent recréer l'état courant d'un agrégat en rejouant les événements

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

# Exemple :

## Mise à jour profil utilisateur





# Architectures Event-driven

---

Les architectures *event-driven* sont des architectures constituées de micro-services consommant en continu des événements.

Chaque micro-service :

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

Cela produit généralement des architectures évolutives, scalables et réactives.

Les micro-services sont simples à développer

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



# Data Stream

## Exemple Spring Cloud Data Flow

### Streams

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

Definitions Create Stream

CREATE STREAM

CLEAR

LAYOUT

AUTO LINK

GRID

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

source

file

ftp

gemfire

gemfire-cq

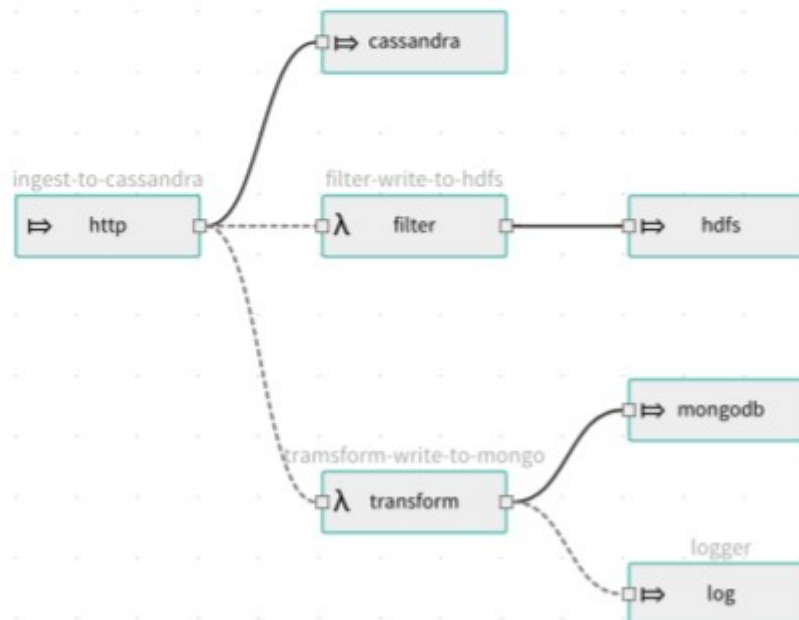
http

jdbc

jms

load-genera...

loggregator





# KafkaStream et kSQLDB

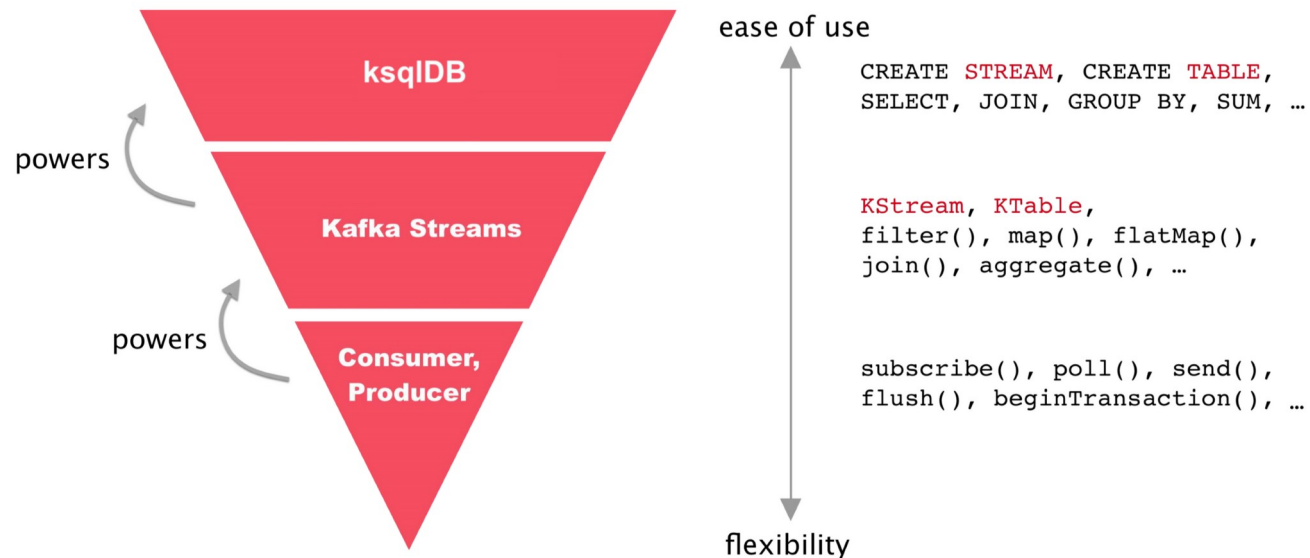
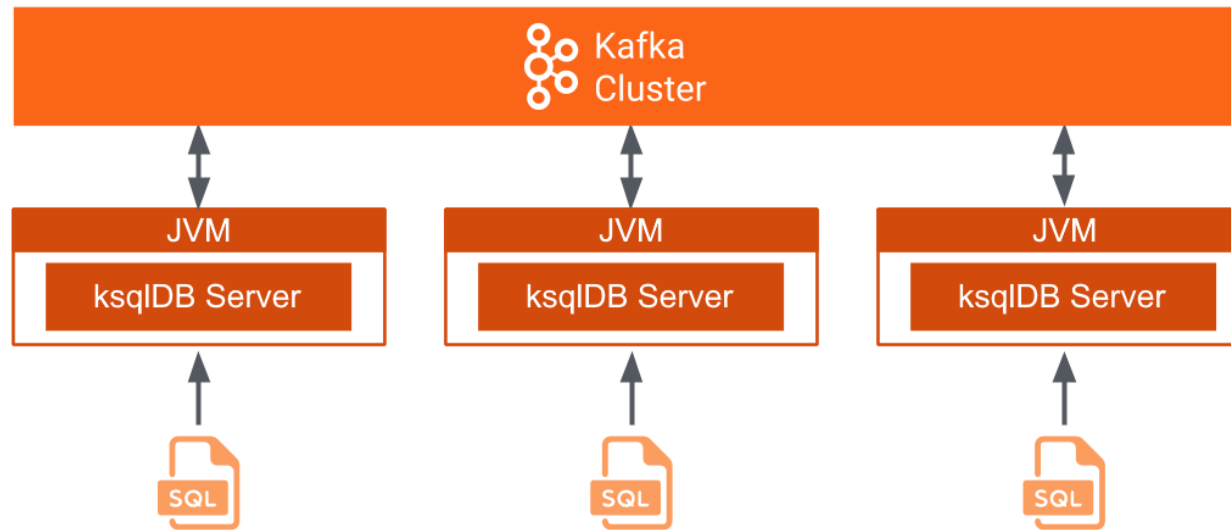
---

KafkaStream est une librairie s'appuyant sur Kafka permettant de manipuler les flux d'événements et de créer des états à partir des flux

KSQLDB s'appuie sur KafkaStream et permet de manipuler les événements avec du SQL



## ksqlDB Standalone Application (Headless Mode)





# Introduction à Kafka

---

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



# Concepts de base

---

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

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

Chaque enregistrement se compose d'une clé éventuelle, d'une valeur, d'un horodatage et éventuellement des entêtes



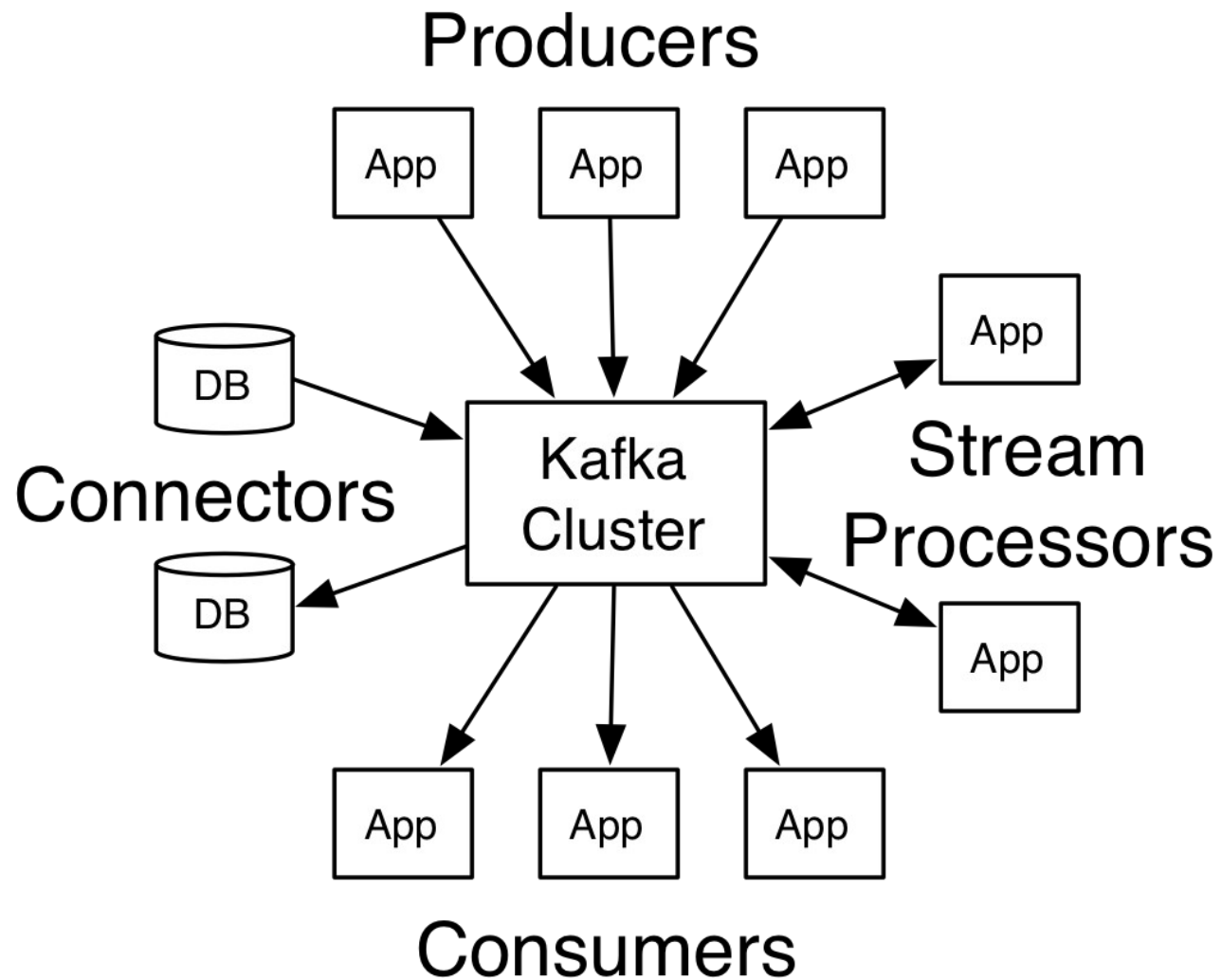
# APIs

---

Kafka propose 5 principales APIs :

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

# APIs





# Protocole Client/Serveur

---

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

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

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

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

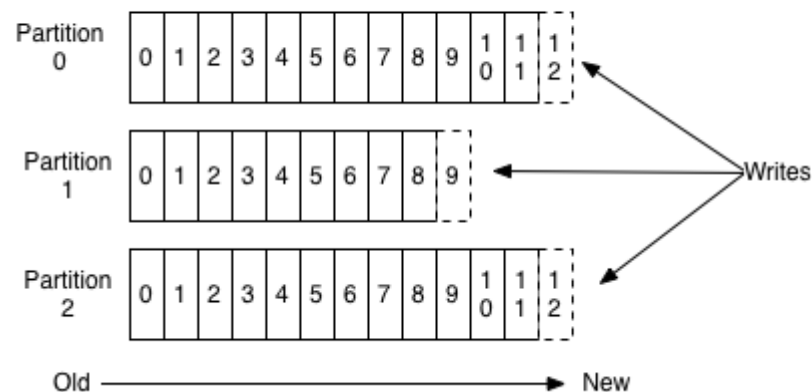
# Topic

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

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

Les topics stockés dans le cluster peuvent être **partitionnés**.

Anatomy of a Topic



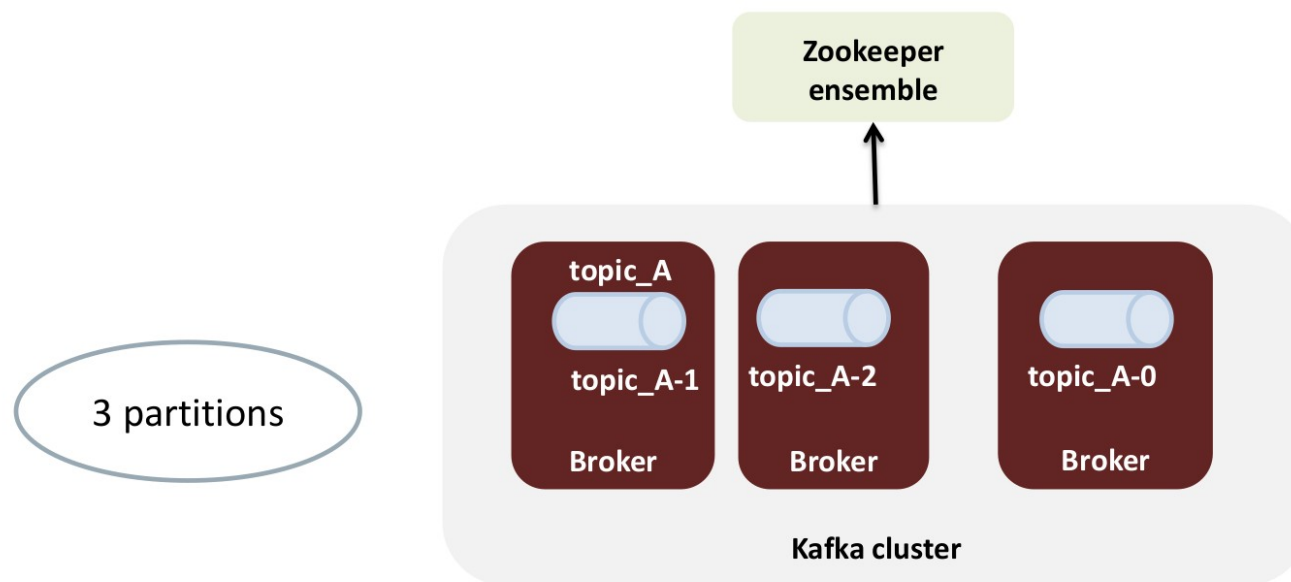


# Apport des partitions

Les partitions autorisent le parallélisme de la consommation et augmentent la capacité de stockage en utilisant les capacités disque de plusieurs nœuds.

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

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

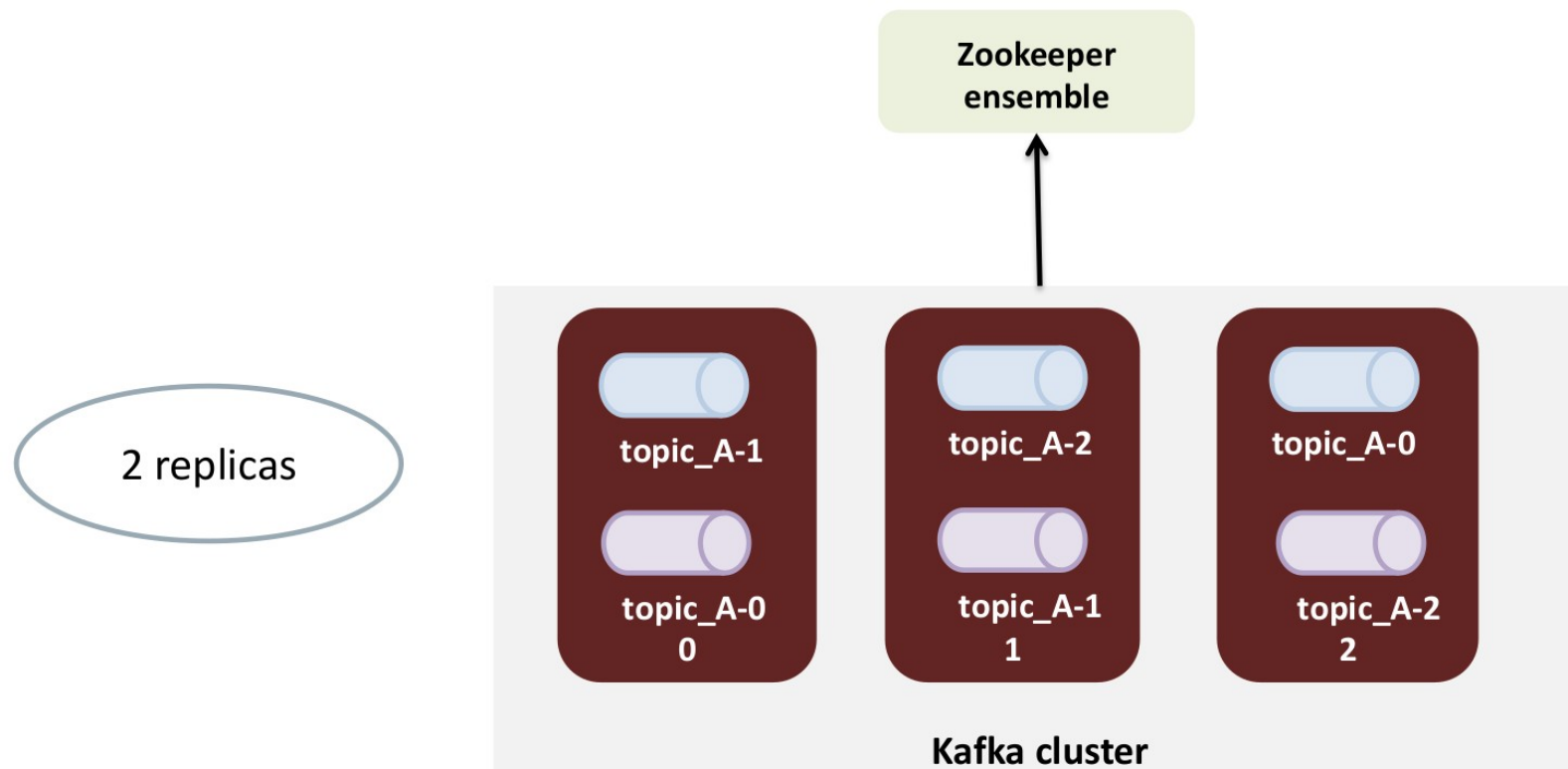


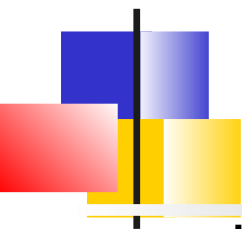


# Réplication

Les partitions peuvent être **répliquées**

- La réplication permet la tolérance aux pannes  
=> durabilité des données





# Distribution des partitions

---

Les partitions d'un topic sont réparties sur les instances du cluster.

Les répliques sont distribuées sur des instances différentes

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

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



# Partition et offset

---

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

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

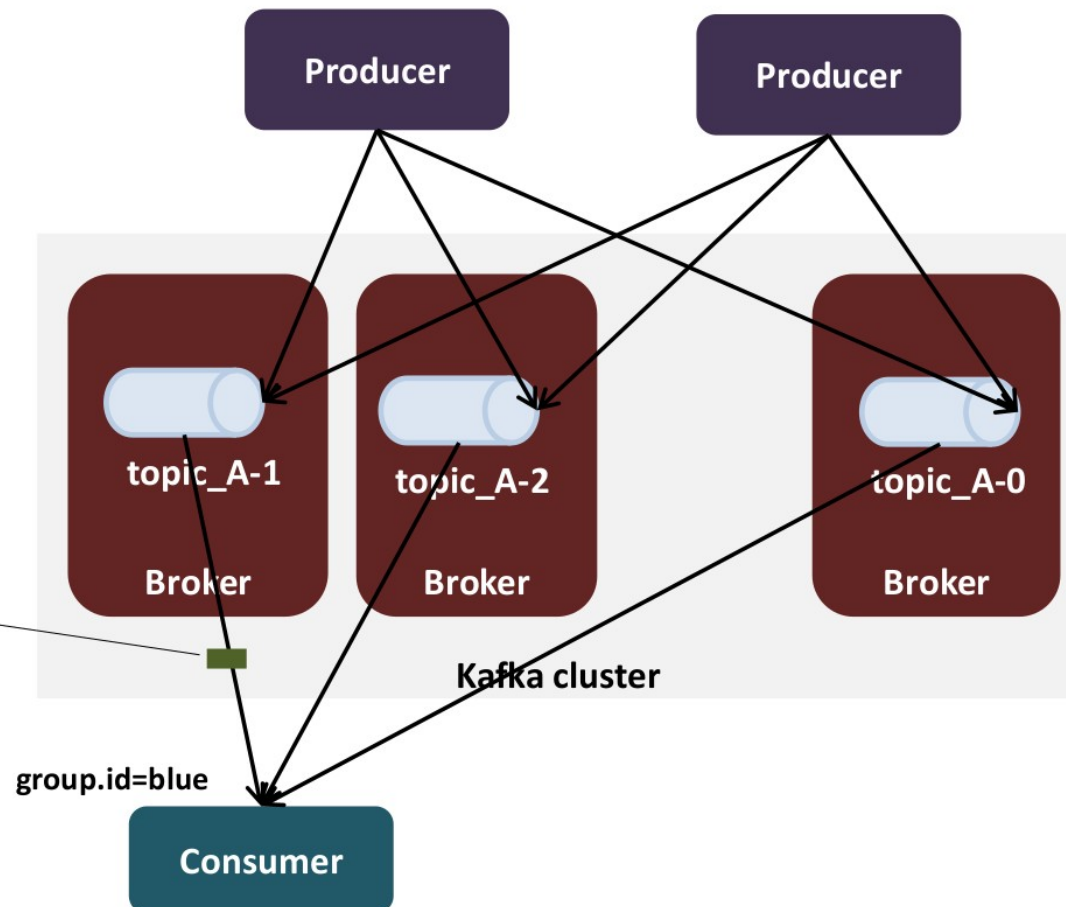
Le cluster Kafka conserve durablement tous les enregistrements publiés, qu'ils aient ou non été consommés, en utilisant une **période de rétention** configurable.

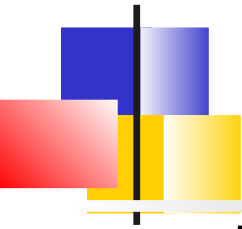
# Clients du cluster

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

**message** (record, event)

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





# Routing des messages

---

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

Cela peut être fait

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



# Groupe de consommateurs

---

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

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



# Offset consommateur

---

Pour chaque groupe de consommateurs, Kafka conserve son **offset**<sup>1</sup> du journal.

Cet offset est contrôlé par le consommateur:

- normalement, le consommateur avance son offset au fur et à mesure des traitements des enregistrements,
- mais, il peut consommer dans l'ordre qu'il souhaite.  
Par exemple, retraiter les données les plus anciennes ou repartir d'un offset particulier.



# Consommateur vs Partition

## Rééquilibrage dynamique

---

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

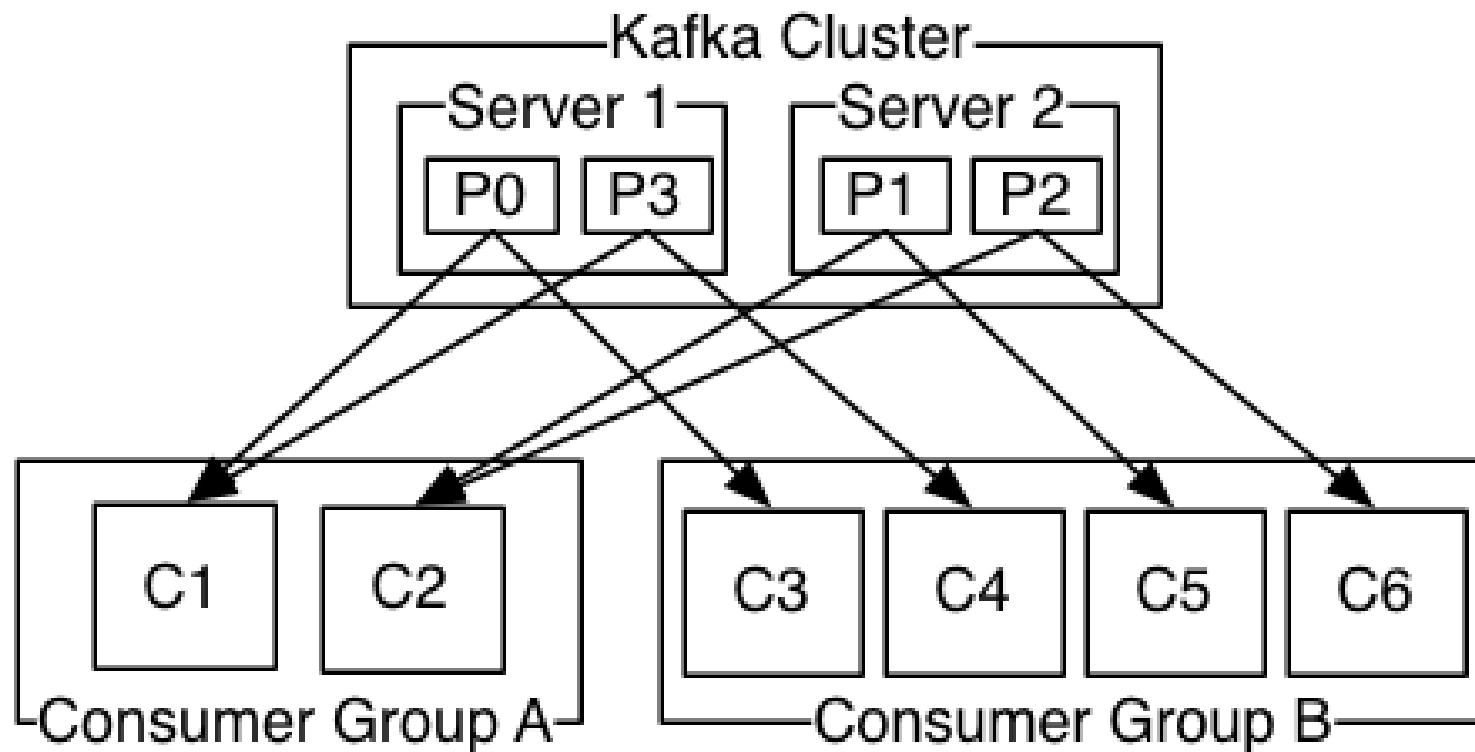
- A tout moment, un consommateur est exclusivement dédié à une partition

Ceci est géré dynamiquement par le protocole Kafka.

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



# Exemple 4 partitions





# Ordre des enregistrements

---

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

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



# Cluster Kafka

---

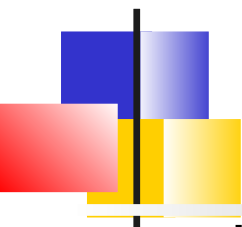
**Cluster**

*KRaft*

Distributions / Installation

Utilitaires *Kafka*

Outils graphiques



# Cluster

---

Kafka est exécuté comme un cluster d'un ou plusieurs **serveurs** pouvant s'étendre sur plusieurs centres de données.

- Certains de ces serveurs appelés les **brokers** forment la couche de stockage.
- Un serveur est désigné **contrôleur**.  
Son rôle est de prendre des décisions concernant le cluster comme l'affectation de partitions

Avant la version 3.x, un cluster Kafka nécessitait également un ensemble Zookeeper permettant de stocker les méta-données nécessaires au contrôleur



# Nombre de brokers

---

Pour déterminer le nombre de brokers :

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



# Configuration et démarrage

---

Kafka fournit un script de démarrage :

***kafka-server-start.sh***

Chaque serveur est démarré via ce script qui lit sa configuration :

- Dans un fichier properties  
***server.properties***
- Certaines propriétés sont généralement surchargées par la commande en ligne  
Option ***--override***



# Principales configuration

---

Les configurations principales sont :

- ***cluster.id***<sup>1</sup> : Identique pour chaque serveur.
- ***broker.id/node.id*** : Différent pour chaque broker
- ***log.dirs*** : Ensemble de répertoires de stockage des enregistrements
- ***listeners*** : Ports ouverts pour communication avec les clients et inter-broker
- ***auto.create.topics.enable*** : Création automatique de topic à l'émission ou à la consommation

*1. Généré automatiquement via Zookeeper, doit être précisé en mode Kraft*



# Cluster Kafka

---

Cluster  
**KRaft**

Distributions / Installation

Utilitaires *Kafka*

Outils graphiques





# Kraft mode

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

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

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

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



# Contrôleurs

---

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

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

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

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



# Cluster ID

---

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

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

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

***bin/kafka-storage.sh random-uuid***



# Formatting des répertoires de log

---

Avec le mode Kraft, il est nécessaire de formater les répertoires des logs.

- Le formatting consiste à créer 2 fichiers de méta-données :
  - ***meta.properties***
  - ***bootstrap.checkpoint***
- Ces fichiers sont dépendants des propriétés *cluster.id* et *node.id*

Pour créer les fichiers :

***bin/kafka-storage.sh format -t <cluster\_id> -c server.properties***



# Cluster Kafka

---

Cluster  
*KRaft*

**Distributions / Installation**

Utilitaires *Kafka*  
Outils graphiques



# Introduction

---

Kafka peut être déployé

- sur des serveurs physiques, des machines virtuelles ou des conteneurs
- sur site ou dans le cloud

Différentes distributions peuvent être récupérées :

- Binaire chez Apache.  
OS recommandé Linux + pré-installation de Java  
(Support de Java 17 pour 3.1.0)
- Images docker ou packages Helm (Bitnami par exemple)
- Confluent Platform (Téléchargement ou cloud)



# Hardware

---

Afin de sélectionner le matériel :

- Débit de disque : Influence sur les producteurs de messages. SSD si beaucoup de clients
- Capacité de disque : Estimer le volume de message \* période de rétention
- Mémoire : Faire en sorte que le système ait assez de mémoire disponible pour utiliser le cache de page. Pour la JVM, 5Go permet de traiter beaucoup de message
- Réseau : Potentiellement beaucoup de trafic. Favoriser les cartes réseau de 10gb
- CPU : Facteur moins important



# Installation à partir de l'archive

---

Téléchargement, puis

```
# tar -zxf kafka_<version>.tgz
# mv kafka_<version> /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk17
# 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***





# Images bitnami

---

## Images basées sur minideb (minimaliste Debian)

```
docker run -d --name kafka-server \  
  --network app-tier \  
  -e ALLOW_PLAINTEXT_LISTENER=yes \  
  bitnami/kafka:latest
```

## Prises en compte de variables d'environnement pour configurer Kafka

- Variables spécifiques bitnami.  
Ex : BITNAMI\_DEBUG, ALLOW\_PLAINTEXT\_LISTENER,  
...
- Mapping des variables d'environnement préfixée par KAFKA\_CFG\_  
Ex : KAFKA\_CFG\_AUTO\_CREATE\_TOPICS\_ENABLE



# Packages Helm

---

Bitnami propose des packages Helm pour déployer vers Kubernetes ou des clouds

```
helm install my-release  
oci://registry-1.docker.io/bitnamicharts/kafka
```

De nombreux paramètres sont disponibles :

- *replicaCount* : Nombre de répliques
- *config* : Fichier de configuration
- *existingConfigmap* : Pour utiliser les ConfigMap
- ...



# Configuration broker

---

***cluster.id*** : Chaque cluster a un ID

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

***listeners***: 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



# Configuration broker

---

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

***listeners*** : Par défaut 9092 et 9093 pour la communication inter-contrôleurs

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

***controller.quorum.voters*** : Listes des contrôleurs

***log.dirs*** : Liste de chemins locaux où kafka stocke les messages (Doivent être préformattés)



# Configuration cluster

---

3 contraintes pour la configuration d'un cluster :

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



# Configuration par défaut des topics

---

***num.partitions*** : Nombre de partitions, défaut 1

***default.replication.factor*** : Facteur de réplication, défaut 1

***min.insync.replicas*** : Joue sur les garanties de message.  
Défaut 1

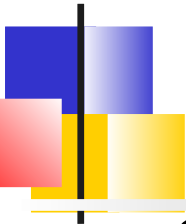
***log.retention.ms*** : Durée de rétention de messages. 7J par défaut

***log.retention.bytes*** : Taille maximale des logs. Pas défini par défaut

***log.segment.bytes*** : Taille d'un segment. 1Go par défaut

***log.roll.ms*** : Durée maximale d'un segment avant rotation. Pas défini par défaut

***message.max.bytes*** : Taille max d'un message. 1Go par défaut



# Vérifications de l'installation

---

## Création de topic :

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

## Envois de messages :

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

## Consommation de messages :

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



# Script d'arrêt

---

La distribution propose également un script d'arrêt permettant d'arrêter les process Kafka en cours d'exécution.

**`bin/kafka-server-stop.sh`**





# Fichiers de trace

---

La configuration des fichiers de trace est définie dans ***conf/log4j.properties***

Par défaut sont générés :

- ***server.log*** : Traces générales
- ***state-change.log*** : Traces des changements d'état (topics, partition, brokers, répliques)
- ***kafka-request.log*** : Traces des requêtes clients et inter-broker
- ***log-cleaner.log*** : Suppression des messages expirés
- ***controller.log*** : Logs du contrôleur
- ***kafka-authorizer.log*** : Traces sur les ACLs



# Cluster Kafka

---

Cluster  
*KRaft*

Distributions / Installation

**Utilitaires Kafka**

Outils graphiques



# Introduction

---

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

Les scripts utilisent l'API Java de Kafka

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



# Gestion des topics

---

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

Les valeurs par défaut des topics sont définis dans *server.properties* mais peuvent être surchargées pour chaque topic

Exemple création de topic :

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

Exemple listing des topics :

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



# Choix du nombre de partitions

---

Une fois un *topic* créé, on ne peut pas diminuer son nombre de partitions. Une augmentation est possible mais peut générer des problèmes.

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 de tester l'envoi des messages sur un topic

Exemple :

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

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



# Lecture de message

---

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

Exemple lecture depuis l'origine :

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



# Autre utilitaires

---

***kafka-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 nouveaux brokers, de nouveaux répertoire, extension de cluster, ...

kafka-replica-verification.sh : Vérifie

***kafka-log-dirs.sh*** : Obtient les informations de configuration des log.dirs du cluster

***kafka-dump-log.sh*** permet de parser un fichier de log et d'afficher les informations utiles pour debugger

***kafka-delete-records.sh*** Permet de supprimer des messages jusqu'à un offset particulier





# Autre utilitaires (2)

---

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

***kafka-cluster.sh*** obtenir l'ID du cluster et sortir un broker du cluster

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

***kafka-acls.sh*** permet de gérer les permissions sur les topics

***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



# Cluster Kafka

---

Cluster  
*KRaft*

Distributions / Installation

Utilitaires Kafka

**Outils graphiques**



# Outils graphiques

---

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

Comme produit gratuit, semi-commerciaux citons :

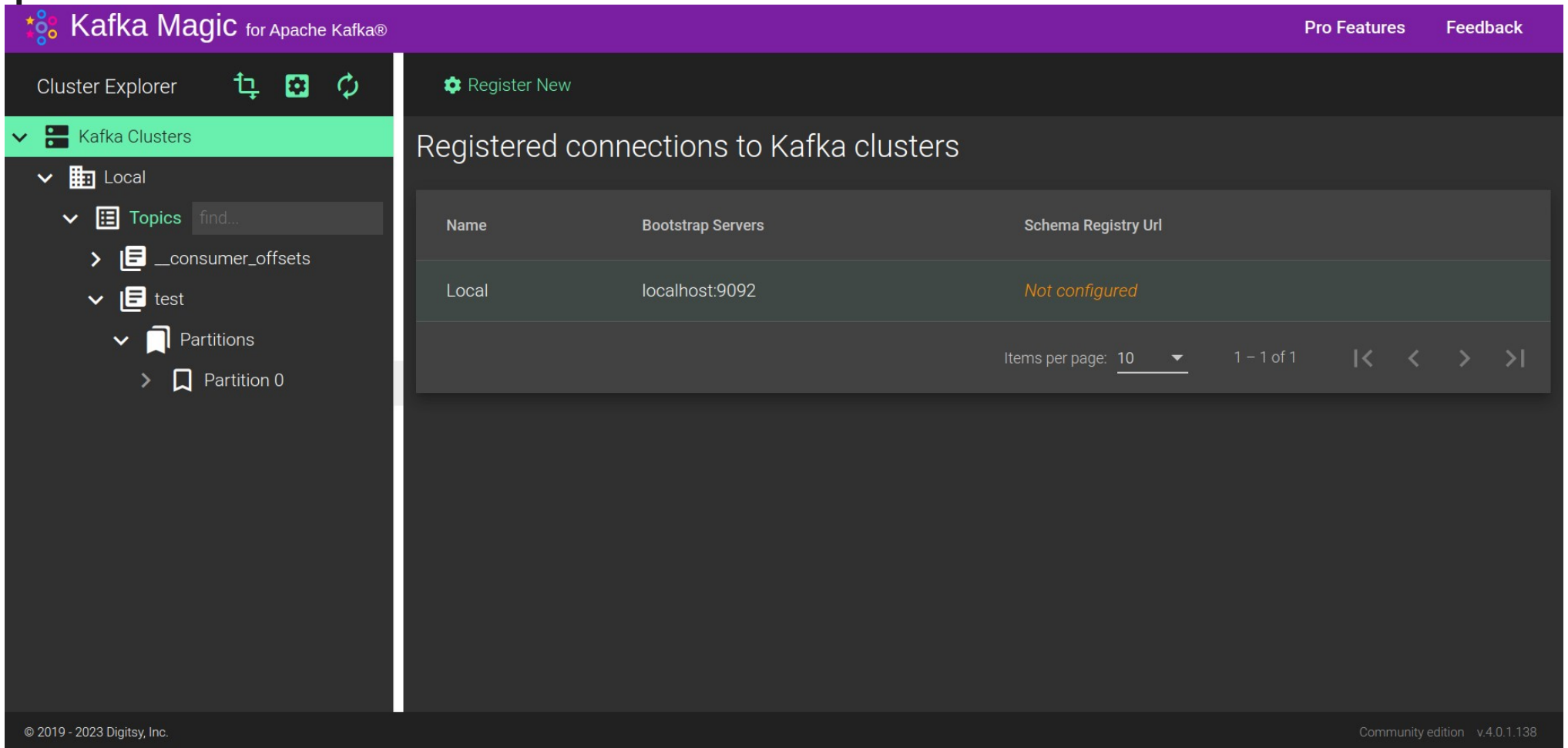
- **Akhq** (<https://akhq.io/>)
- **Kafka Magic** (<https://www.kafkamagic.com/>)
- **Redpanda Console** (<https://docs.redpanda.com/docs/manage/console/>)

Ils proposent une interface graphique permettant :

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

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

# Kafka Magic



The image shows the Kafka Magic web interface. The top navigation bar is purple and contains the 'Kafka Magic for Apache Kafka®' logo, 'Pro Features', and 'Feedback' links. The left sidebar is dark grey and contains a 'Cluster Explorer' section with icons for cluster, settings, and refresh. Below this, the 'Kafka Clusters' section is expanded, showing a tree view with 'Local' selected. Under 'Local', there are 'Topics' (with a search bar) and 'Partitions'. The 'Topics' section shows a list of topics: '\_\_consumer\_offsets' and 'test'. The 'Partitions' section shows 'Partition 0'. The main content area is dark grey and displays 'Registered connections to Kafka clusters'. It features a table with columns 'Name', 'Bootstrap Servers', and 'Schema Registry Url'. The table contains one entry: 'Local' with 'localhost:9092' and 'Not configured'. Below the table, there is a pagination control showing 'Items per page: 10' and '1 - 1 of 1'.

Kafka Magic for Apache Kafka®

Pro Features Feedback

Cluster Explorer

Kafka Clusters

Local

Topics find...

\_\_consumer\_offsets

test

Partitions

Partition 0

Register New

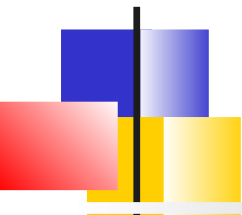
Registered connections to Kafka clusters


Name	Bootstrap Servers	Schema Registry Url
Local	localhost:9092	Not configured

Items per page: 10 1 - 1 of 1


© 2019 - 2023 Digitsy, Inc. Community edition v.4.0.1.138


# akhq





akhq.io


0.24.0


my-cluster ▾


Nodes


Topics

Live Tail

Consumer Groups



ACLs

Schema Registry

Settings

## Nodes

[↗ default](#)

Id ▾	Host ▾	Controller ▾	Partitions (% of total)	Rack ▾	
1	localhost:9092	False	1 (100.00%)		🔍
2	localhost:9192	True			🔍
3	localhost:9292	False			🔍

# Redpanda Console



Redpanda

- Overview
- Topics
- Schema Registry
- Consumer Groups
- Security
- Quotas
- Connectors
- Reassign Partitions

Cluster > Topics



1 1  
Total Topics Total Partitions

Create Topic

☐ Show internal topics

Name	Partitions	Replicas	CleanupPolicy	Size
test	1	1	delete	145 B

Total 1 items < 1 > 50 / page



Redpanda Console (Platform Version v23.1) (built May 03, 2023) FC2BF3B



# Clients Kafka

---

## **Types de clients**

Production de messages

Consommation de messages

Schema registry

Garanties de livraison

Latence, Débit

Kafka Connect



# Types de client

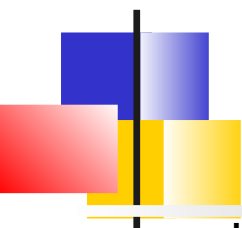
---

Un serveur Kafka a différents types de clients :

- Les autres serveurs (brokers ou contrôleurs)
- Les outils d'administration (Admin API)
- Les producteurs et les consommateurs de message (Client API)
- KafkaConnect
- Les applications KafkaStream

Dans un environnement de production, les flux d'échanges sont séparés en définissant différents listeners (ports différents)





# Configuration des listeners

---

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

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

Ou LISTENER\_NAME est un nom descriptif

Exemple :

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

***advertised.listeners*** : Si l'IP interne n'est pas accessible par les clients, cette propriété doit spécifier l'adresse externe (hôte/IP) utilisable par les clients.

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



# Listeners du cluster

---

***inter.broker.listener.name*** : Nom du listener utilisé pour la communication inter-broker. Si non spécifié, security.inter.broker.protocol, par défaut PLAINTEXT.

***control.plane.listener.name*** : Nom du listener utilisé pour la communication entre le contrôleur et les brokers. Si non spécifié, pas de listener dédié à ce type de communication

***controller.listener.names*** : Une liste de nom des listeners utilisés pour la communication inter controller. Requis en mode Kraft



# Protocoles

---

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

Exemple :

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

Les protocoles supportés sont :

- ***PLAINTEXT*** : Transport en clair
- ***SSL*** : Transport crypté (TLS 1.2 et 1.3)
- ***SASL\_PLAINTEXT*** : Authentification + transport en clair
- ***SASL\_SSL*** : Authentification + crypté



# Configuration commune aux clients

---

Quelque-soit l'API utilisée, un client doit fournir la propriété ***bootstrap.servers***

- Cette propriété est renseignée avec 1 ou plusieurs adresses des serveurs.
- Il suffit que le client en contacte un pour qu'il découvre l'intégralité du cluster.



# Librairies

---

Apache fournit :

- Une librairie Java ***kafka-clients*** : Envoi/Consommation de messages + administration cluster
- Une librairie java ***kafka-stream*** permettant de construire des applications event-stream
- Une distribution de ***KafkaConnect***, un serveur permettant de s'intégrer avec des systèmes externes

Confluent fournit des librairies pour les autres technologies (Python, .NET, ...)

Des tiers fournissent également des librairies en nodejs, php, ...



# Clients Kafka

---

Types de clients

**Production de messages**

Consommation de messages

Schema registry

Garanties de livraison

Latence, Débit

Kafka Connect



# 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 ?



# Configuration Producteur

---

En dehors de ***bootstrap.servers***, le producteur configure 2 sérialiseurs

- ***key.serializer*** : La classe utilisée pour la sérialisation de la clé
- ***value.serializer*** : La classe utilisée pour la sérialisation du corps du message ...

Les sérialiseurs typiques sont Json, Avro, Protobuf

Un producteur peut également préciser

***client.id***. Une chaîne de caractère utilisé pour le monitoring.





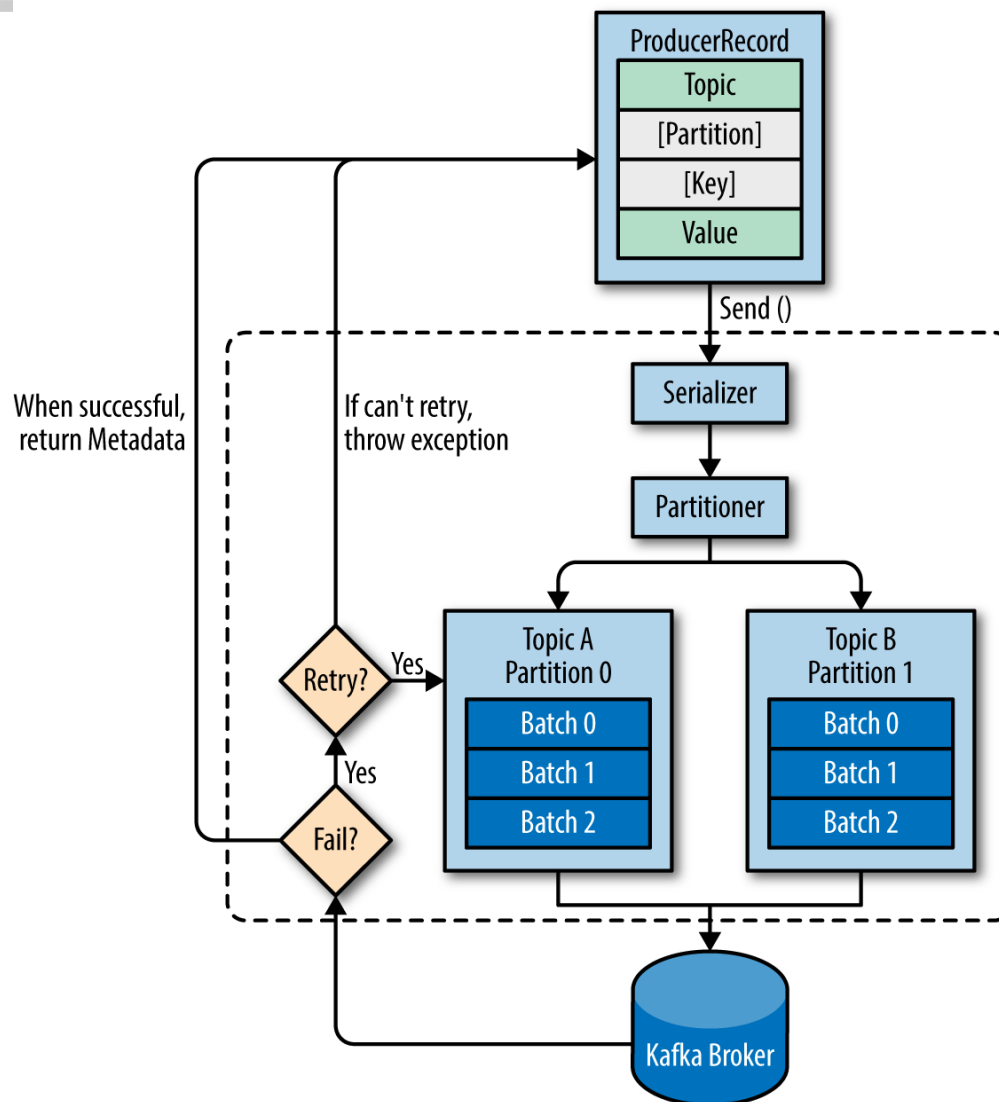
# Étapes lors de l'envoi d'un message

---

L'envoi de message est constitué de plusieurs étapes :

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

# Envoi de message





# Méthodes d'envoi des messages

---

3 façons pour un producteur 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*** : Le producteur attend l'acquittement du message.  
On traite éventuellement les cas d'erreurs
- ***Envoi asynchrone*** : Le producteur précise une fonction de call-back qui sera appelée lorsque la réponse est retournée.



# Écriture sur une partition

---

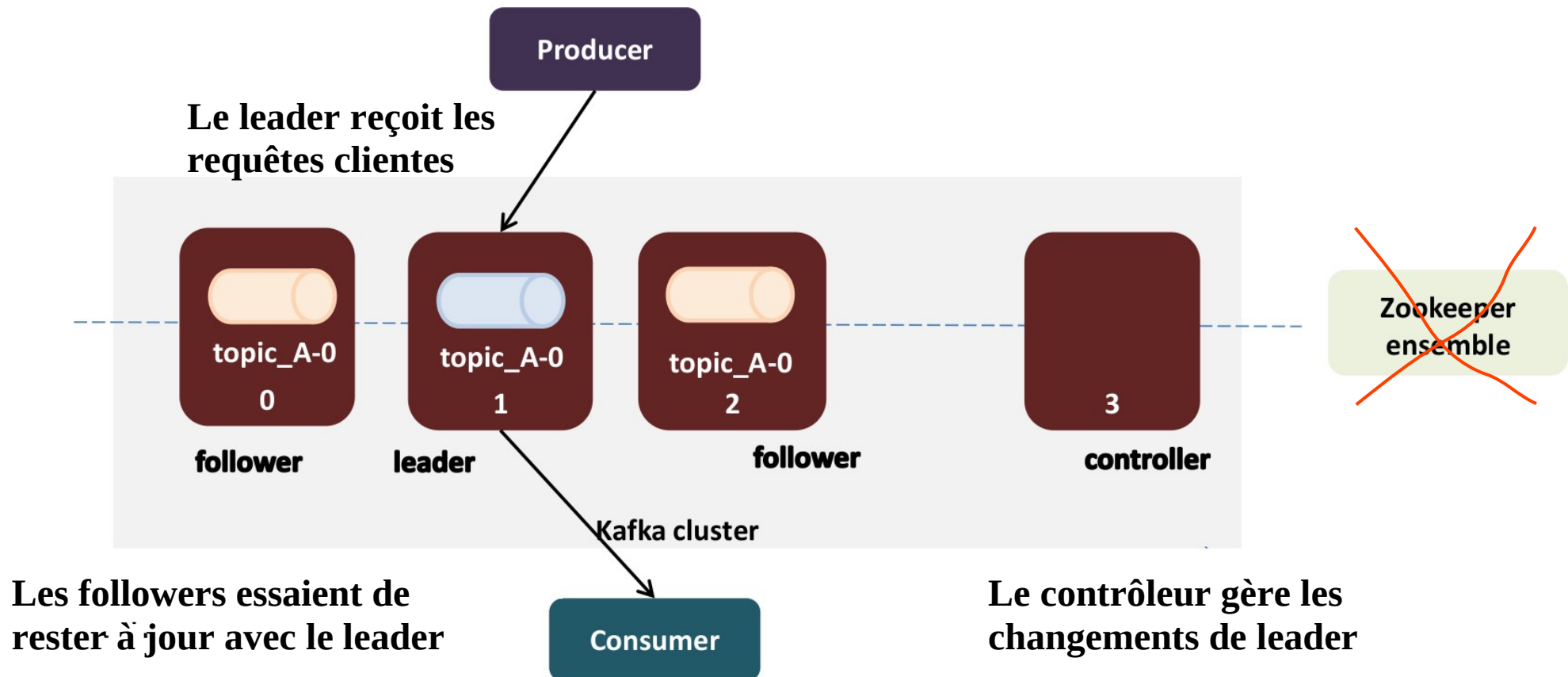
Différents brokers participent à la gestion distribuée et répliquée d'une partition.

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

Au niveau du cluster :

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

# Rôle des brokers gérant un topic





# Garanties Kafka

---

Garanties offertes par Kafka grâce à la réplication :

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



# ISR

---

Contrôlé par la propriété :

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

Si pendant ce délai, le follower

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

Alors, le follower est considéré comme désynchronisé

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

*kafka-topics.sh --describe* : permet de voir les ISR pour chaque partition d'un topic



# *min.insync.replica*

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

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



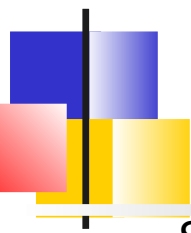


# Conséquences

---

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

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



# Rejet de demande d'émission

---

Si le **nombre de ISR** < *min.insync.replicas* :

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

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

- Mauvaise configuration, Kafka bloque les émissions de message

En conséquences :

$n$  répliques

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

$n$  répliques, *min.insync.replicas* =  $m$

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



# Configuration des producteurs *fiabilité*

---

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

Fiabilité :

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



# Configuration des producteurs *performance*

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



# Garantie sur l'ordre

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

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

En cas de failure

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



# Clients Kafka

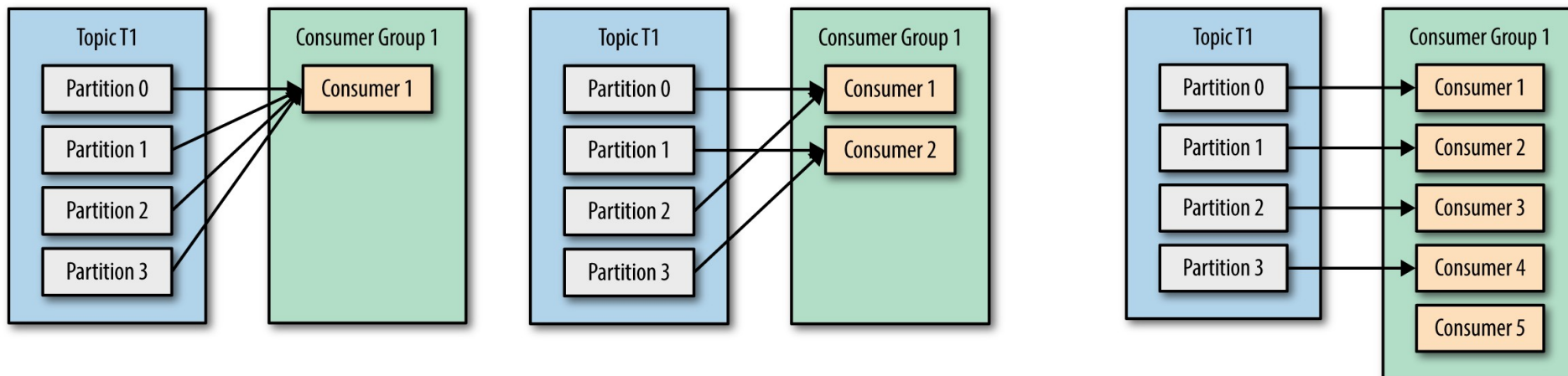
---

Types de clients  
Production de messages  
**Consommation de messages**  
Schema registry  
Garanties de livraison  
Latence, Débit  
Kafka Connect

# 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.





# Configuration Consommateur

---

En dehors de *bootstrap.servers*, le consommateur configure

- ***group.id*** qui spécifie le groupe de consommateur
- ***key.deserializer*** et ***value.deserializer*** qui permettent de transformer le message en objet





# Rééquilibrage dynamique des consommateurs

---

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

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

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

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



# Membership et détection de pannes

---

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

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

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



# Static Group Membership

---

Si une instance de consommateur précise un identifiant d'instance via la propriété ***group.instance.id***, les mêmes partitions lui seront affectés lors des arrêts et redémarrages

La réaffectation de partition ne s'effectuera lorsque le timeout ***session.timeout.ms*** aura expiré



# Boucle de Polling

---

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

Périodiquement, il récupère un lot d'enregistrements.

Chaque enregistrement encapsule :

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



# Exemple Java

---

```
try {
while (true) {
    // poll envoie le heartbeat, on bloque pdt 100ms pour récupérer les messages
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        log.debug("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n", record.topic(), record.partition(),
            record.offset(), record.key(), record.value());

        int updatedCount = 1;
        if (custCountryMap.containsValue(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount) ;

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



# Offsets et Commits

---

Les consommateurs synchronise l'état d'avancement de sa consommation avec Kafka. En périodiquement indiquant le dernier offset de messages qu'ils ont traité.

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

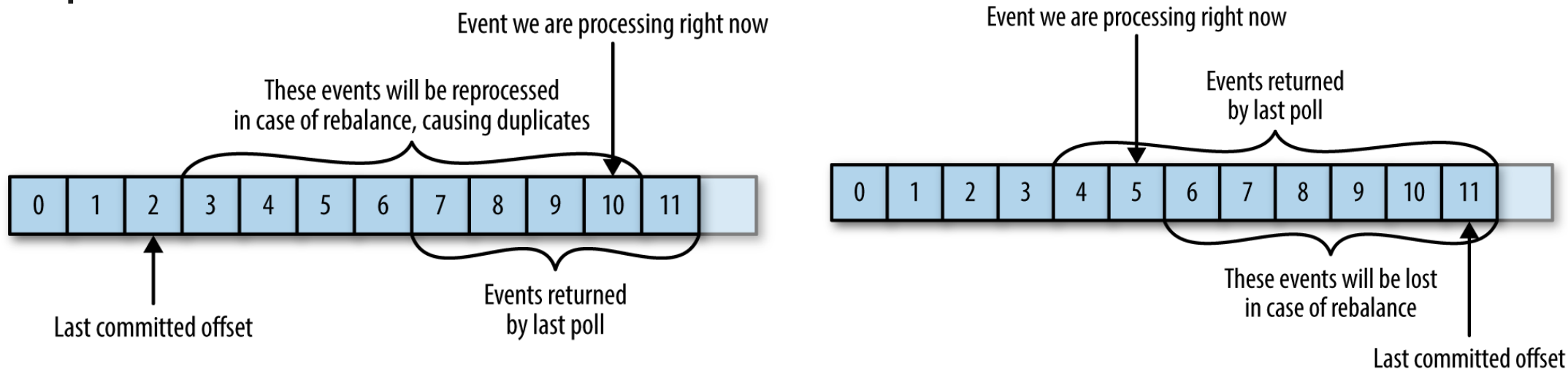
Pour committer, un consommateur envoie un message vers un *topic* particulier de Kafka : ***\_\_consumer\_offsets***

- 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 via la propriété *enable.auto.commit*



# Commit automatique

---

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

- => Cette approche (simple) ne protège pas contre les duplications ou les pertes de message en cas de ré-affectation
- => Cela dépend de la grandeur de l'intervalle en fonction du temps de traitement des messages et si le traitement des messages est asynchrone





# Commit contrôlé

---

L'API Consumer permet de contrôler le moment du commit

Si ***auto.commit.offset=false*** ,  
l'application doit explicitement  
committer les offsets du dernier poll()

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



# 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()*



# Stocker les offsets hors de Kafka

---

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

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

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



# Réagir aux réaffectations

---

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



# Affectation statique des partitions

---

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

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

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



# Configuration des consommateurs

---

Si le consommateur n'a pas d'offset défini dans Kafka, le comportement est piloté par la propriété

***auto.offset.reset :***

- *latest* (défaut), l'offset est initialisé au dernier offset
- *earliest* : L'offset est initialisé au premier offset disponible



# Configuration des consommateurs (2)

---

## D'autres propriétés

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



# Clients Kafka

---

Types de clients  
Production de messages  
Consommation de messages  
**Schema registry**  
Garanties de livraison  
Latence, Débit  
Kafka Connect





# Introduction

---

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

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

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

Elles offrent également un format de sérialisation binaire plus compact



# Apache Avro

---

*Apache Avro* est un système de sérialisation de données.

- Il utilise une structure JSON pour définir le **schéma**, permettant la sérialisation entre les octets et les données structurées.
- Les outils associés à Avro sont capables de générer les classes Java<sup>1</sup> correspondantes au schéma.

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



# Utilisation du schéma

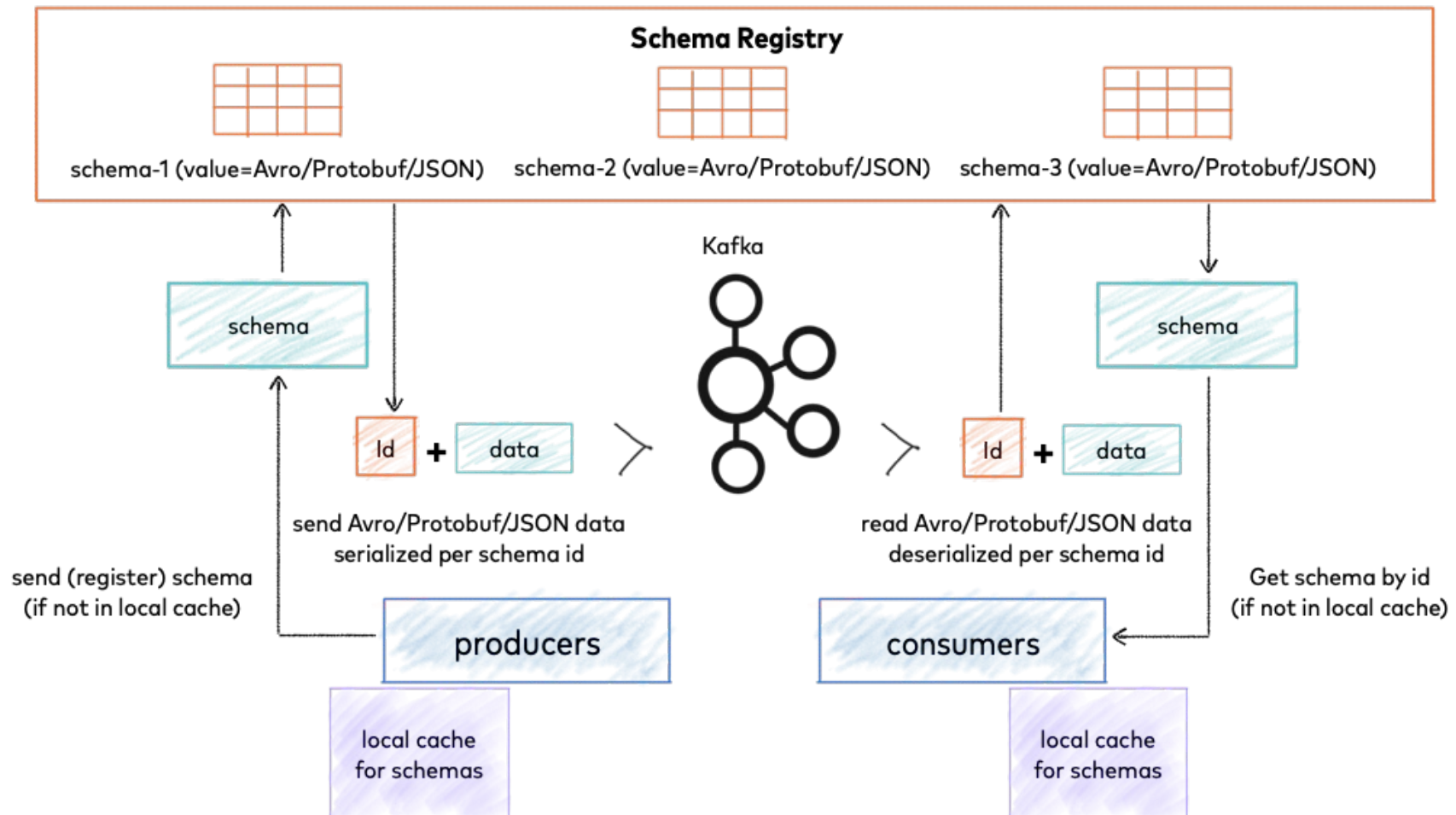
---

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

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

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

# Schema Registry





# Dépendances Confluent

---

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



# Schéma Avro

---

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



# Plug-in Avro

---

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

=> *mvn compile* génère les classes du modèle



# Producteur (1)

---

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

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





# Producteur (2)

---

Les propriétés du *KafkaProducer* doivent contenir :

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



# Consommateur

---

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

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

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



# Clients Kafka

---

Types de clients

Production de messages

Consommation de messages

*Schema registry*

**Mécanismes de réplication**

Garanties de livraison

Latence, Débit

Kafka Connect



# Introduction

---

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

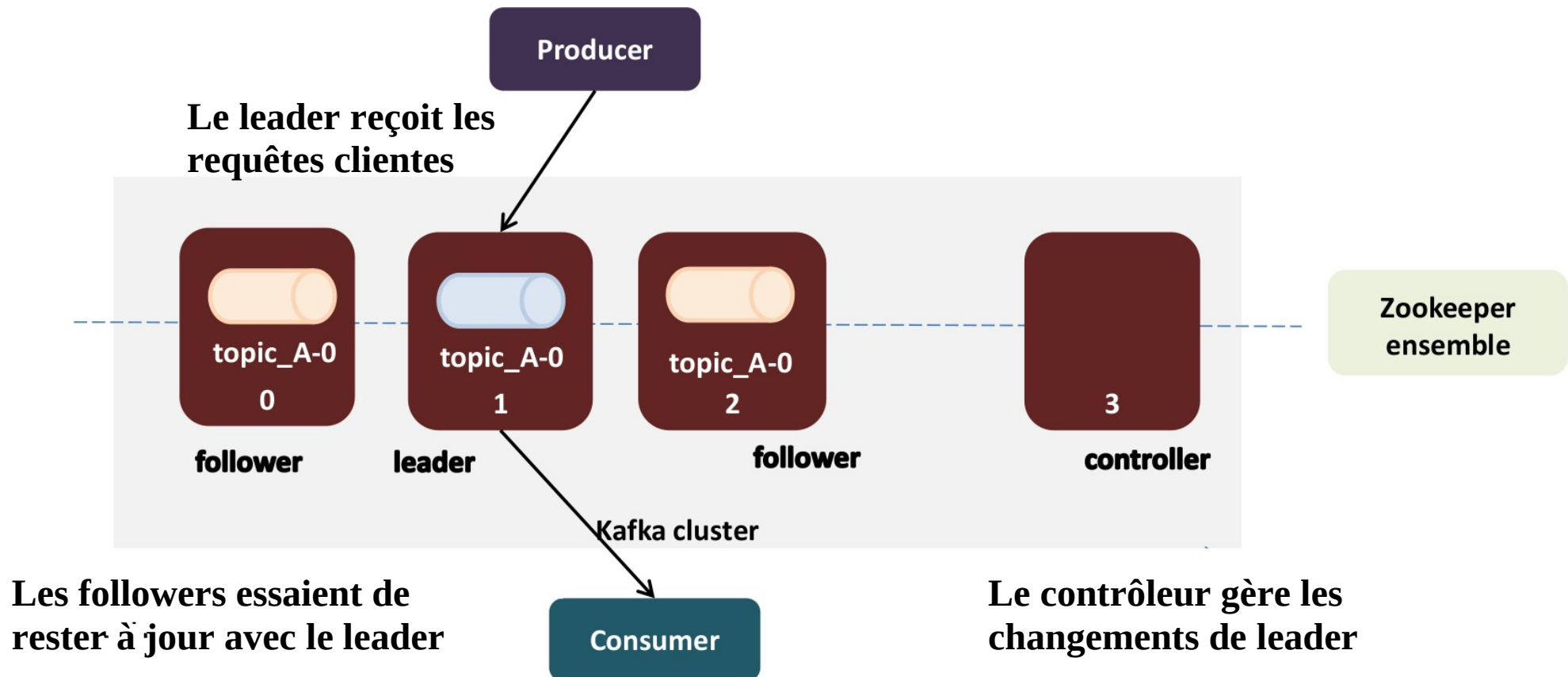
Pour chaque partition d'un topic :

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

Au niveau du cluster :

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

# Rôle des brokers gérant un topic





# Garanties Kafka

---

Garanties offertes par Kafka grâce à la réplication :

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



# Synchronisation des répliques

---

Contrôlé par la propriété :

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

Si pendant ce délai, le follower

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

Alors, le follower est considéré comme désynchronisé

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



# *min.insync.replica*

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

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



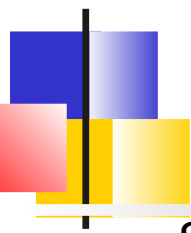


# Conséquences

---

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

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



# Rejet de demande d'émission

---

Si le **nombre de ISR** < ***min.insync.replicas*** :

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

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

- Mauvaise configuration, Kafka bloque les émissions de message

En conséquences :

n répliques

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

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

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



# Configuration

---

2 principales configurations affectent la fiabilité et les compromis liés :

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



# Clients Kafka

---

Types de clients  
Production de messages  
Consommation de messages  
*Schema registry*  
Mécanismes de réplication  
**Garanties de livraison**  
Latence, Débit  
Kafka Connect



# Introduction

---

En fonction des configurations du topic, des producteurs et des consommateurs, Kafka peut garantir différents niveaux de livraison :

- ***At Most Once*** : Un message est livré au plus une fois. On tolère les pertes de message
- ***At Least Once*** : Le message est livré au moins une fois. On tolère les doublons
- ***Exactly once*** : Le message est livré 1 et 1 seule fois.

Tout ceci avec des défaillances



# Côté producteur

---

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

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



# acks=1

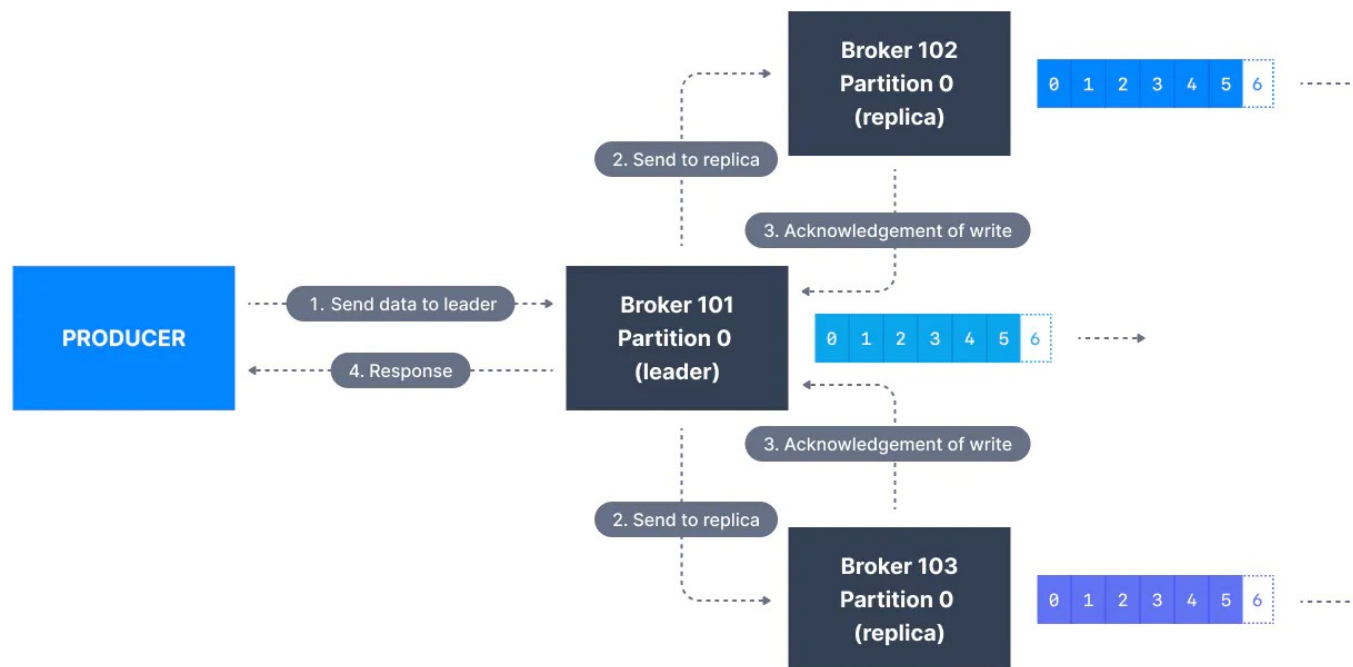
**acks=1** : Le producteur considère le message écrit lorsque le leader a acquitté l'écriture  
=> Si le leader s'arrête et que les répliques n'ont pas eu le temps d'écrire le message, perte de données  
(At most once avec diminution du risque)



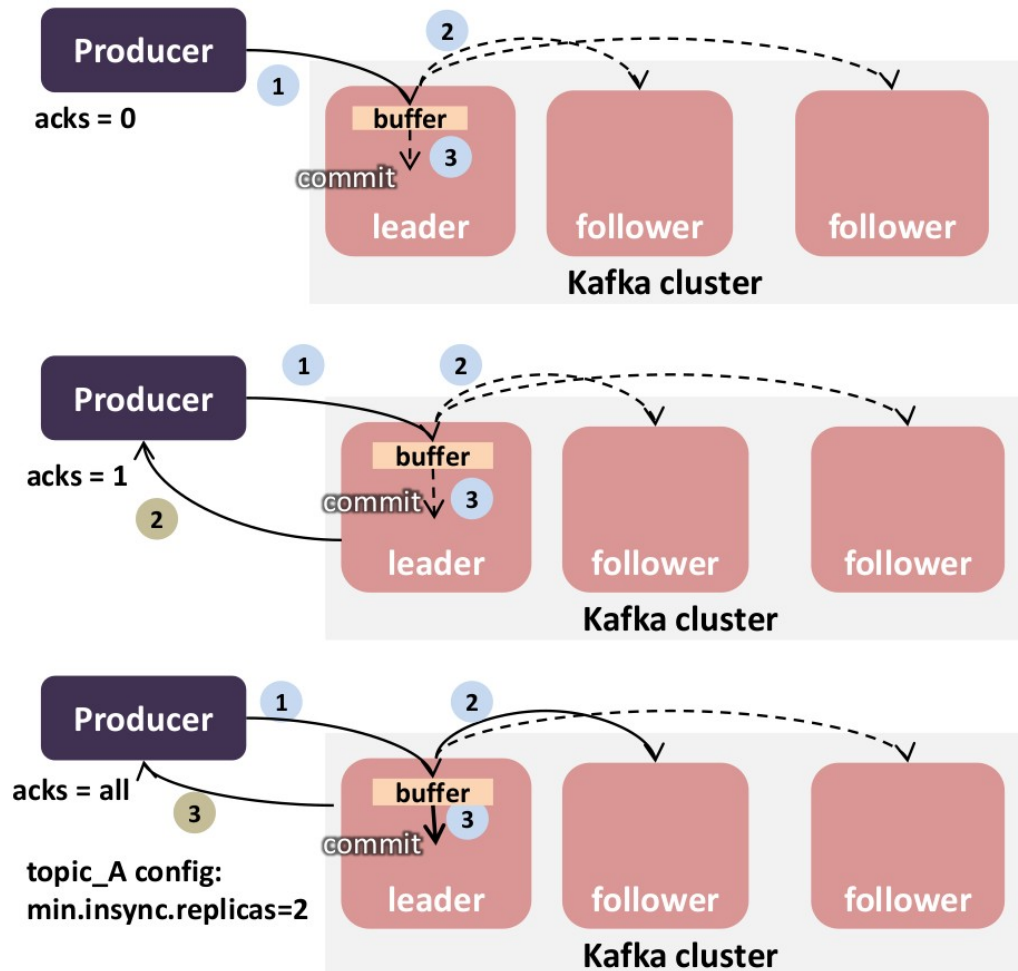


# acks=all

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



# Acquittement et durabilité



Latency

Data loss risk





# Gestion des erreurs

---

2 types d'erreur pour le producteur:

- les erreurs que les producteurs gèrent automatiquement .  
Ce sont les erreurs ré-essayable (ex :  
LEADER\_NOT\_AVAILABLE, NOT\_ENOUGH\_REPLICA)  
Nombre d'essai configurable via **retries**.  
=> Attention, peut générer des doublons
- les erreurs qui doivent être traitées par le code. (ex :  
INVALID\_CONFIG, SERIALIZATION\_EXCEPTION)  
Typiquement, envoyé vers une dead-letter-queue



# Rebalancing

---

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

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

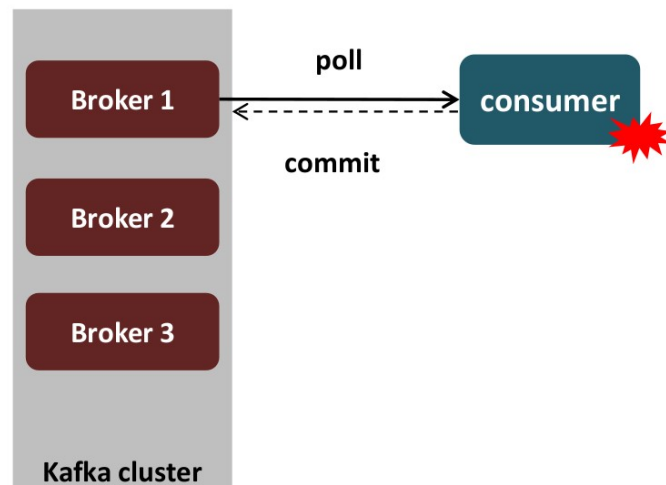
Si l'offset n'est pas synchronisé avec les traitements effectués, cela peut :

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

Il est possible de s'abonner aux événements de rebalancing



# Consommateur: At Most Once



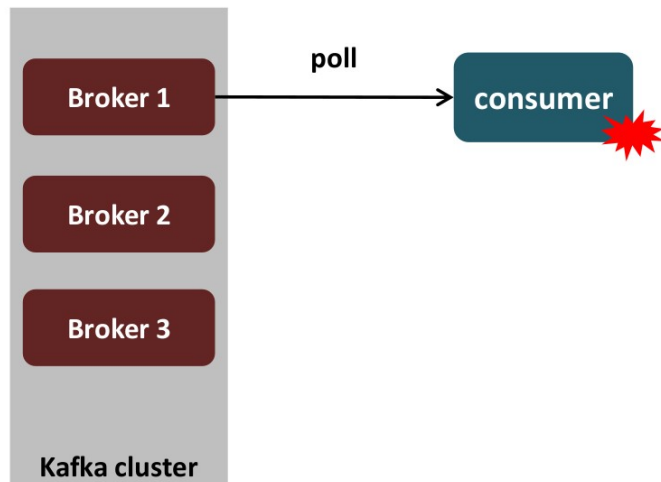
- L'offset est commité
- Traitement d'un ratio de message puis plantage

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

OU

Commit manuel avant le traitement des messages

# Réception : *At Least Once*



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

Configuration par défaut et traitement synchrone

Ou

*enable.auto.commit* à false ET Commit explicite après le traitement



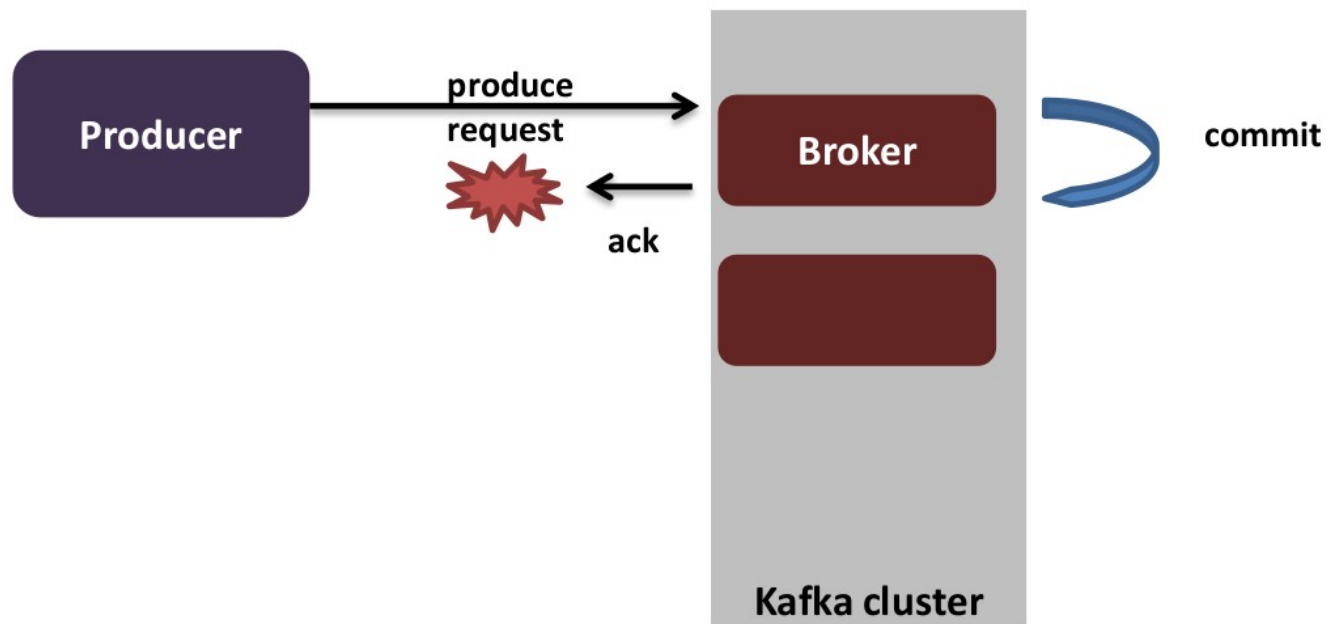
# Introduction

---

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

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

# Producteur idempotent



Le producteur ajoute une  
nombre séquentiel et un ID  
de producteur

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





# Configuration

---

*enable.idempotence*

L'idempotence nécessite les configurations suivantes :

- *max.in.flight.requests.per.connection*  $\leq 5$
- *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 manuellement les offsets des partitions dans un support de persistance transactionnel et partagé par tous les consommateurs + gérer les rééquilibrages ou forcer des assignations statiques
- Si le traitement consiste à produire un message vers le même cluster Kafka, on peut utiliser les transactions<sup>1</sup>.

*1. C'est le cas de KafkaStream*



# 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 avoir la propriété *isolation.level* à *read committed*
- L'API *send()* devient bloquante



# Transaction

---

Permet la production atomique de messages sur plusieurs partitions

Les producteurs produisent l'ensemble des messages ou aucun

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



# Configuration du consommateur

---

Afin de lire les messages transactionnels validés, les consommateurs doivent modifier la propriété ***isolation.level*** à ***read\_committed***

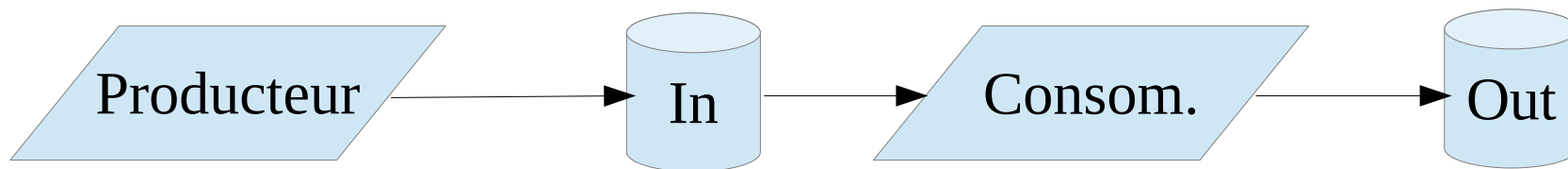
- *read\_committed*: Messages (transactionnels ou non) validés
- *read\_uncommitted*: Tous les messages (même les messages transactionnels non validés)



# Exactly Once et Transaction

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

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





# Producteur

---

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

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



# Consommateur

---

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





# Clients Kafka

---

Types de clients  
Production de messages  
Consommation de messages  
*Schema registry*  
Mécanismes de réplication  
**Garanties de livraison**  
Latence, Débit, Durabilité  
Kafka Connect



# Configuration pour favoriser le débit

---

## Côté producteur :

### Augmenter

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

### Puis

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

## Côté consommateur

### Augmenter

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



# Configuration pour favoriser la latence

---

## Cluster

Si followers pas suffisamment rapide, augmenter leur nombre

- *num.replica.fetchers* : (défaut 1)

## Côté producteur

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

## Côté consommateur

- *fetch.min.bytes*: 1



# Configuration pour la durabilité

---

## Cluster

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

## Producteur

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

## Consommateur

- *isolation.level: read\_committed*



# Clients Kafka

---

Types de clients  
Production de messages  
Consommation de messages  
*Schema registry*  
Mécanismes de réplication  
Garanties de livraison  
Latence, Débit, Durabilité  
**Kafka Connect**



# Introduction

---

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

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



# Fonctionnalités

---

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



# Mode standalone

---

Pour démarrer *Kafka Connect* en mode standalone

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

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

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

Les autres paramètres définissent les configurations des différents connecteurs





# Mode distribué

---

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

Pour démarrer en mode distribué :

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

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

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



# Configurations supplémentaires en mode distribué

---

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

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

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

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

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

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

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



# Configuration des connecteurs

---

Configuration :

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

Les valeurs sont très dépendantes du type de connecteur.

Comme valeur communes, on peut citer :

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



# Connecteurs

---

*Confluent* offre de nombreux connecteurs<sup>1</sup> :

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

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

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



# Exemple Connecteur JDBC

---

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

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

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

topic.prefix=mysql-
```



# Transformations

---

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

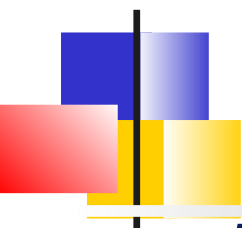
- ***transforms*** : Liste d'aliases spécifiant la séquence des transformations
- ***transforms.\$alias.type*** : La classe utilisée pour la transformation.
- ***transforms.\$alias.\$transformationSpecificConfig*** : Propriété spécifique d'un *Transformer*



# Exemple de configuration

---

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



# *Transformers disponibles*

---

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

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

***ReplaceField*** : Filtrer ou renommer des champs

***MaskField*** : Remplace le champ avec une valeur nulle

***ValueToKey*** : Échange clé/valeur

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

***SetSchemaMetadata*** : Modifie le nom du schéma ou la version

***TimestampRouter*** : Route le message en fonction du timestamp

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





# API Rest

---

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

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

Ces listeners sont également utilisés par la communication intra-cluster

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

`rest.advertised.listener`



# API

---

**GET /connectors** : Liste des connecteurs actifs

**POST /connectors** : Création d'un nouveau connecteur

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

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

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

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

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

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

**DELETE /connectors/{name}** : Supprimer un connecteur



# Sécurité

---

## **Configuration des listeners**

SSL/TLS

Authentication via SASL

ACLs



# Introduction

---

Différentes alternatives supportées pour la sécurité :

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

Naturellement, dégradation des performances avec SSL



# Listeners

---

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

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

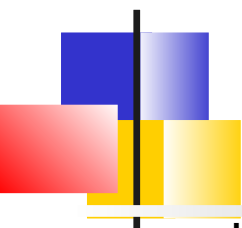


# Combinaisons des protocoles

---

Chaque protocole combine une couche de transport (PLAINTEXT ou SSL) avec une couche d'authentification optionnelle (SSL ou SASL) :

- **PLAIN\_TEXT** : En clair sans authentification. Ne convient qu'à une utilisation au sein de réseaux privés pour le traitement de données non sensibles
- **SSL** : Couche de transport SSL avec authentification client SSL en option. Convient pour une utilisation dans réseaux non sécurisés car l'authentification client et serveur ainsi que le chiffrement sont prise en charge.
- **SASL\_PLAINTEXT** : Couche de transport PLAINTEXT avec authentification client SASL. Ne prend pas en charge le cryptage et convient donc uniquement pour une utilisation dans des réseaux privés.
- **SASL\_SSL** : Couche de transport SSL avec authentification SASL. Convient pour une utilisation dans des réseaux non sécurisés.



# Configuration des listeners

---

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

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

Ou LISTENER\_NAME est un nom descriptif

Exemple :

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

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

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

Les protocoles supportés sont : *PLAINTEXT*, *SSL*, *SASL\_PLAINTEXT*, *SASL\_SSL*

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



# Sécurité

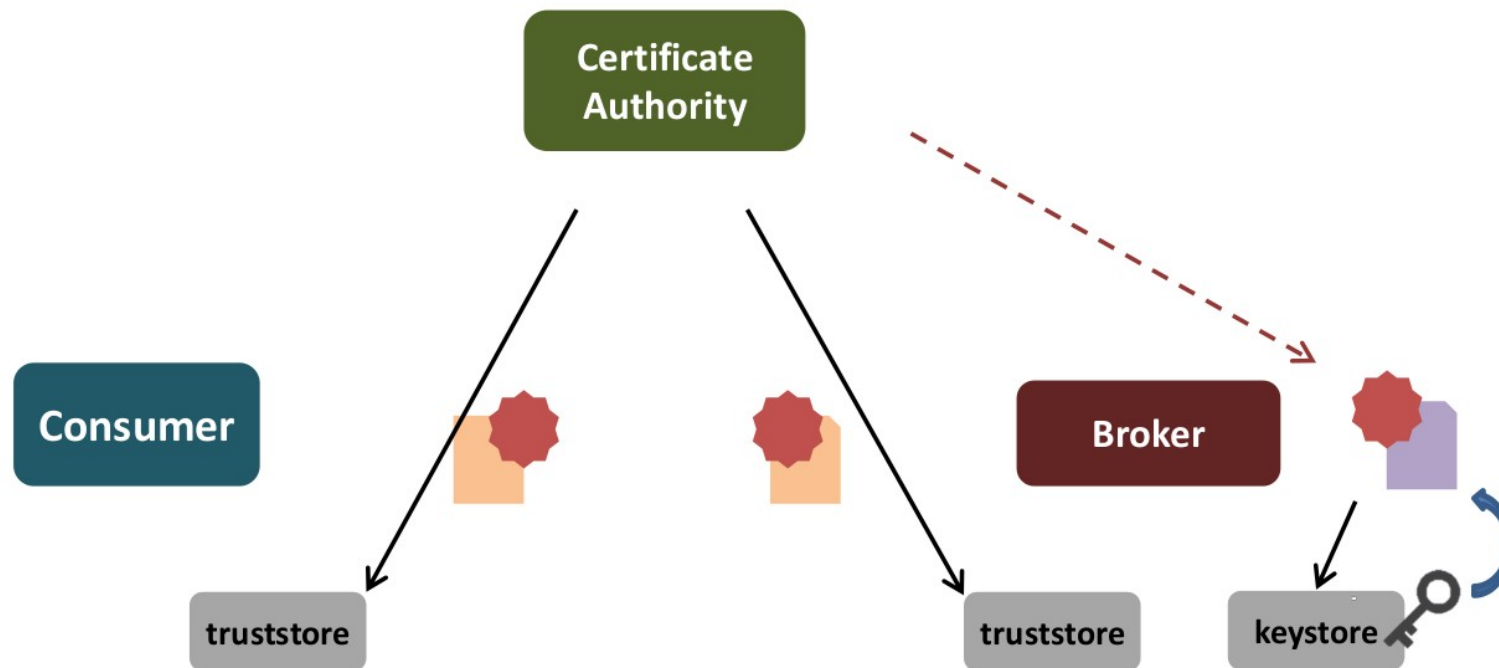
---

Configuration des listeners  
**SSL/TLS**  
Authentication via SASL  
ACLs



# SSL

## SSL pour le cryptage et l'authentification





# Configuration TLS et Vérification d'hôte

---

Les certificats des brokers doivent contenir le nom d'hôte du broker en tant que nom alternatif du sujet (SAN) extension ou comme nom commun (CN) pour permettre aux clients de vérifier l'hôte du serveur.

Les certificats génériques utilisant les wildcards peuvent être utilisés pour simplifier l'administration en utilisant le même keystore pour tous les brokers d'un domaine



# Préparation des certificats

---

## # Générer une paire clé publique/privé pour chaque serveur

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

## # Créer son propre CA (Certificate Authority)

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

## # Importer les CA dans les truststore

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

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

## # Créer une CSR (Certificate signed request)

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

## # Signer la CSR avec le CA

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

## # Importer le CA et la CSR dans le keystore

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

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



# Configuration du broker

---

*server.properties :*

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

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

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

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

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

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



# Configuration du client

---

```
security.protocol=SSL
```

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

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

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

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



# Authentification des clients via SSL

---

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

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



# Sécurité

---

Configuration des listeners  
SSL/TLS

**Authentication via SASL**  
ACLs



# SASL

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

Kafka utilise JAAS pour la configuration SASL.

Mécanismes:

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





# Configuration JAAS

---

La configuration JAAS s'effectue :

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

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

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

–



# Configuration JAAS

---

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

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



# Mécanismes

---

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

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

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

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



# SASL PLAIN

---

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

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

3. Configurer SASL dans *server.properties*

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



# Configuration du client

---

Créer le fichier Jaas

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

Ajouter un paramètre JVM

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

*client.properties*

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



# SASL / PLAIN en production

---

Doit être utilisé avec SSL

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

***sasl.server.callback.handler.class***

et

***sasl.client.callback.handler.class***.

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

***sasl.server.callback.handler.class***



# Sécurité

---

Configuration des listeners  
SSL/TLS  
Authentication via SASL  
**ACLs**



# Introduction

---

Les brokers Kafka gèrent le contrôle d'accès à l'aide d'une API définie par l'interface

*org.apache.kafka.server.authorizer.Authorizer*

L'implémentation est configurée via la propriété :

***authorizer.class.name***

Kafka fournit 2 implémentations

- Pour Zookeeper,  
***kafka.security.authorizer.AclAuthorizer***
- En kraft mode  
***org.apache.kafka.metadata.authorizer.StandardAuthorizer***





# Autorisation

---

Les ACLs sont stockées dans les méta-données gérées par les contrôleurs et peuvent être gérées par l'utilitaire ***kafka-acls.sh***

Chaque requête Kafka est autorisée si le *KafkaPrincipal* associé à la connexion a les autorisations pour effectuer l'opération demandée sur les ressources demandées.

Les règles peuvent être exprimées comme suit :

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



# Exemple

---

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

```
bin/kafka-acls.sh --bootstrap-servers
localhost:9092 --add --allow-principal
User:Bob --allow-principal User:Alice --
allow-host 198.51.100.0 --allow-host
198.51.100.1 --operation Read --operation
Write --topic Test-topic
```



# Ressources et opérations

---

Chaque ACL consiste en :

- **Type de ressource** : *Cluster | Topic | Group | TransactionalId*
- **Type de pattern** : *Literal | Prefixed*
- **Nom de la ressource** : Possibilité d'utiliser les wildcards
- **Opération** : *Describe | Create | Delete | Alter | Read | Write | DescribeConfigs | AlterConfigs*
- **Type de permission** : *Allow | Deny*
- **Principal** : De la forme *<principalType>:<principalName>*  
Exemple : *User:Alice, Group:Sales, User :\**
- **Host** : Adresse IP du client, \* si tout le monde

Exemple Complet :

*User:Alice has Allow permission for Write to Prefixed  
Topic:customer from 192.168.0.1*



# Règles

---

AclAuthorizer autorise une action s'il n'y a pas d'ACL de DENY qui corresponde à l'action et qu'il y a au moins une ACL ALLOW qui correspond à l'action.

- L'autorisation Describe est implicitement accordée si l'autorisation Read, Write, Alter ou Delete est accordée.
- L'autorisation Describe Configs est implicitement accordée si l'autorisation AlterConfigs est accordée.



# Permissions clientes

---

Brokers : ***Cluster:ClusterAction*** pour autoriser les requêtes de contrôleur et les requêtes de fetch des répliques.

Producteurs simples : ***Topic:Write***

- idempotents sans transactions : ***Cluster:IdempotentWrite***.
- Transactionnels : ***TransactionalId:Write*** à la transaction et ***Group:Read*** pour que les groupes de consommateurs valident les offsets.

Consommateurs : ***Topic:Read*** et ***Group:Read*** s'ils utilisent la gestion de groupe ou la gestion des offsets.

Clients admin : ***Create, Delete, Describe, Alter, DescribeConfigs, AlterConfigs*** .



# Exceptions

---

2 options de configuration permettant d'accorder un large accès aux ressources permet de simplifier la mise en place d'ACL à des clusters existants :

- ***super.users*** : Permet de définir les utilisateurs ayant droit à tout
- ***allow.everyone.if.no.acl.found=true*** : Tous les utilisateurs ont accès aux ressources sans ACL.



# Exemple

---

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

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



# Audit

---

Les brokers peuvent être configurés pour générer des traces d'audit.

La configuration s'effectue dans *conf/log4j.properties*

- Le fichier par défaut est *kafka-authorizer.log*
- Le niveau INFO trace les entrées pour chaque refus
- Le niveau DEBUG pour chaque requête acceptée





# Administration

---

## **Gestion des topics**

Stockage et rétention des partitions

Quotas

Gestion du cluster

Dimensionnement

Monitoring



# Introduction

---

L'outil ***kafka-topics.sh*** permet d'effectuer facilement la plupart des opérations sur les topics (créer, supprimer, décrire)

Les commandes de modification de la configuration ont été dépréciées au profit de l'outil générique ***kafka-config.sh***



# *describe*

---

```
./kafka-topics.sh --bootstrap-servers localhost:9092 --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
```



# Gestion des groupes de consommations

---

***kafka-consumer-groups.sh*** peut être utilisé pour :

- Lister les groupes
- Décrire un groupe
- Supprimer des groupes
- Réinitialiser les informations d'offsets

Exemple :

Exporter les offsets en CSV

```
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --export  
--group my-consumer --topic my-topic --reset-offsets --to-current  
--dry-run > offsets.csv
```

Importer à partir de CSV

```
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --reset-  
offsets --group my-consumer --from-file offsets.csv --execute
```



# 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 entités

---

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

Exemple, modification propriété d'un topic

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



# Suppression de Topic

---

La suppression doit être autorisée sur les brokers

***delete.topic.enable=true***

Il faut arrêter tous les producteurs/consommateurs avant la suppression

Les offsets des consommateurs ne sont pas supprimés immédiatement



# Ajout de partition

---

L'ajout de partition permet de scaler horizontalement et d'augmenter le parallélisme de la consommation

- Attention au topic contenant des messages avec des clés

```
kafka-topics.sh --bootstrap-server localhost:9092  
--alter --topic my-topic --partitions 16
```

Il n'est pas possible de réduire le nombre de partitions





# Extension du cluster

---

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

Il faut manuellement exécuter  
***kafka-reassign-partitions***

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



# Usage de *kafka-reassign-partition*

---

*kafka-reassign-partition* peut être utilisé pour :

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



# Étape 1

## Génération proposition

---

```
$ cat topics-to-move.json
```

```
{"topics":[{"topic" : "foo1"}, {"topic" : "foo2"}], "version":1}
```

```
$ ./kafka-reassign-partitions.sh --bootstrap-servers localhost:9092 \  
--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 --bootstrap-servers localhost:9092 \
--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 \  
  --bootstrap-servers localhost:9092 \  
  --reassignment-json-file expand-cluster-reassignment.json \  
  --verify
```

Status of partition reassignment :

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



# Administration

---

Gestion des topics  
**Stockage et rétention des partitions**  
Quotas  
Gestion du cluster  
Dimensionnement  
Monitoring



# Introduction

---

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

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

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



# Allocation des partitions

---

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

Ses objectifs sont :

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





# Rétention des données

---

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

- Soit une durée
- Soit un volume

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

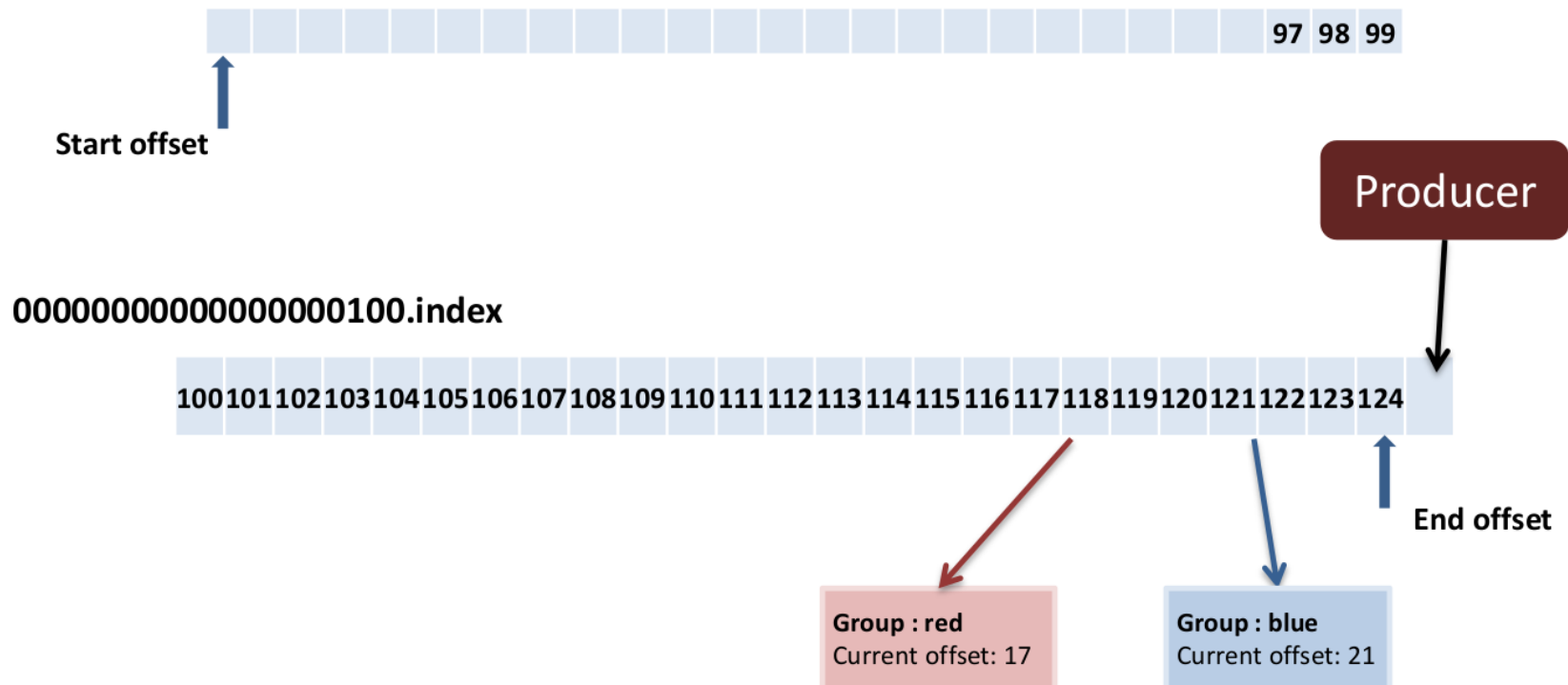
- Les segments sont des fragments de partition au maximum de 1Go et au maximum d'une semaine de données
- Lors de l'écriture d'un segment, lorsque la limite est atteinte, le broker ferme le fichier. Il devient éligible pour la purge
- Il existe donc un seul *segment actif* qui d'ailleurs ne peut pas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

# Segments

## Log segment

closing segment parameters :

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





# Indexation

---

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

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

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



# Principales configurations

---

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

***log.retention.minutes*** (défaut null),

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

Période de rétention des vieux segment avant de les supprimer

***log.retention.bytes (défaut -1)***

La taille maximale du log

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

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



# Compactage

---

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

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



# Nettoyage des logs

---

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

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

Possibilité de combiner les 2 stratégies (delete and compact)



# Stratégie *delete*

---

La stratégie *delete* s'appuie sur :

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

=> Meilleur contrôle de l'usage disque



# Stratégie compact

---

2 propriétés de configuration importante pour cette stratégie :

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

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

Le pool est configurable :

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

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

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

=> Consomme CPU et RAM





# Exemple *compact*

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



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



# Administration

---

Gestion des topics  
Stockage et rétention des partitions

## **Quotas**

Gestion du cluster  
Dimensionnement  
Monitoring



# Introduction

---

Kafka permet d'appliquer des quotas sur les requêtes pour contrôler les ressources du broker utilisées par les clients.

Deux types de quotas peuvent être appliqués par groupe de client :

- Les quotas de bande passante réseau définissent des seuils de débit
- Les quotas de taux de requête définissent les seuils d'utilisation du processeur en pourcentage du réseau et des threads d'E/S
- Quotas du taux de connexion par IP

L'identité du client correspond au `KafkaPrincipal` dans un cluster sécurisé ou la propriété applicative `client-id`.

Tous les clients ayant la même identité partagent leur configuration de quotas



# Quota de bande passante

---

Le seuil de débit d'octets pour chaque groupe de clients.

Chaque groupe peut publier/consommer un maximum de X octets/sec par broker avant d'être limités.

```
kafka-configs.sh --bootstrap-server $B00T --alter --  
add-config  
  'producer_byte_rate=1024,consumer_byte_rate=1024' --  
entity-type users --entity-name <authenticated-  
principal> --command-config /tmp/client.properties
```



# Taux de requêtes

---

Définis en pourcentage de temps sur les threads d'I/O et de réseau de chaque broker dans une fenêtre temporelle.

```
kafka-configs.sh --bootstrap-server $BOOT --  
  alter --add-config 'request_percentage=150'  
  --entity-type users --entity-name big-time-  
tv-show-host --command-config  
/tmp/client.properties
```



# Application des quotas

---

Lorsqu'un broker constate une violation de quota, il calcule une estimation du délai nécessaire pour ramener le client sous son quota.

Le broker peut alors :

- Répondre au client avec une demande de délai
- Désactiver le canal de la socket

Selon le client (récent ou non), le client peut soit respecter ce délai, soit l'ignorer.



# Administration

---

Gestion des topics  
Stockage et rétention des partitions  
Quotas  
**Gestion du cluster**  
Dimensionnement  
Monitoring



# 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 -bootstrap-servers  
localhost:9092 --describe --topic my_topic_name
```

Exemple d'outillage :

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





# Mise à jour du cluster

---

Avant la mise à niveau:

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

Garanties:

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



# Étapes de mise à jour

---

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

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

Mettre à niveau les brokers un par un (redémarrage)

- Attendre l'état stable (ISR = réplicas)

Mettre à jour en dernier le contrôleur

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

Mettre à niveau les clients

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



# Administration

---

Gestion des topics  
Stockage et rétention des partitions  
Quotas  
Gestion du cluster  
**Dimensionnement**  
Monitoring



# Kafka Broker

---

CPU : Pas d'utilisation intensive du CPU

Disque :

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

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

- 16 Gb - 64 Gb

JVM :

- 4 Gb - 6 Gb

Réseau:

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



# Configuration système

---

Étendre la limite du nombre de fichiers ouverts :

- *ulimit -n 100000*

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

- *vm.swappiness=1*
- *vm.dirty\_background\_ratio=5*
- *vm.dirty\_ratio=60*



# JVM

---

KAFKA\_HEAP\_OPTS peut être utilisée

Heap size :

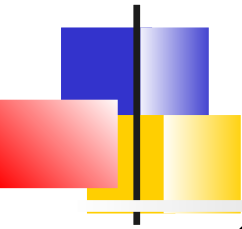
– -Xms4G -Xmx4G or -Xms6G -Xmx6G

Options JVM

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

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

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



# Dimensionnement cluster

---

Généralement basée sur les capacités de stockage :

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

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

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



# Contrôleur

---

Un nombre impair de serveurs  
contrôleurs pour le Quorum

- 3 nœuds permet une panne
- 5 nœuds permet 2 pannes





# Partitionnement des topics

---

Augmenter le nombre de partitions  
permet plus de consommateurs

Mais augmente généralement la taille du  
cluster

Évaluer l'utilisation d'un autre topic ?

Généralement 24 est suffisant



# Administration

---

Gestion des topics  
Stockage et rétention des partitions  
Quotas  
Gestion du cluster  
Dimensionnement  
**Monitoring**



# Introduction

---

Un cluster Kafka peut gérer une quantité importante de données, il est important de monitorer pour maintenir des performances fiables des applications qui en dépendent.

3 types de métriques sont à surveiller :

- Métriques des serveurs Kafka
  - Métriques JMX émis par Kafka
  - Métriques hôtes
  - Métriques JVM
- Métriques des producteurs
- Métriques des consommateurs



# Principaux métriques JMX des brokers

Métrique	Description	Alerte
kafka.server:type=ReplicaManager,name= <b>UnderReplicatedPartitions</b>	Nombre de partition sous-répliquée	Si > 0
kafka.controller:type=KafkaController,name= <b>OfflinePartitionsCount</b>	Nombre de partitions qui n'ont pas de leader actif	Si > 0
kafka.controller:type=KafkaController,name= <b>ActiveControllerCount</b>	Nombre de contrôleur actif dans le cluster	Si != 1
kafka.server:type=ReplicaFetcherManager,name= <b>MaxLag</b> ,clientId=Replica	Retard maximal des messages entre les répliques et le leader	
kafka.server:type=ReplicaManager,name= <b>IsrShrinksPerSec/IsrExpandsPerSec</b>	Cadence de shrink/expansion des ISR	Si != 0



# Principaux métriques JMX des brokers

Métrique	Description	Alerte
kafka.controller:type=ControllerStats,name= <b>=LeaderElectionRateAndTimeMs</b>	Cadence d'élection de leaders	Si > 0
kafka.controller:type=ControllerStats,name= <b>=UncleanLeaderElectionsPerSec</b>	Cadence d'élection de leaders parmi les répliques non synchronisée	Si > 0
kafka.network:type=RequestMetrics,name= <b>TotalTimeMs</b> ,request={Produce FetchConsumer FetchFollower}	Total en ms pour les requêtes de production ou du fetch	Si changement significatif
kafka.network:type=RequestMetrics,name= <b>=RequestsPerSec</b> ,request={Produce FetchConsumer FetchFollower}	Nombre de requêtes par seconde, pour produire et récupérer	Si changement significatif
kafka.server:type=DelayedOperationPurgatory,delayedOperation=Produce,name= <b>PurgatorySize</b>	Nombre de requêtes d'émission en attente dans le purgatoire. !=0 lorsque acks=all	Si élevé
kafka.server:type=BrokerTopicMetrics,name= <b>=BytesInPerSec/BytesOutPerSec</b>	Débit In/Out du point de vue des clients	Si pas bon, compression ?



# Métriques des hôtes

---

4 principaux métriques à surveiller :

- **Page cache reads ratio** : Ratio lecture page cache/disque, une valeur élevée équivaut à des meilleures performances. Si le taux de lecture reste inférieur à 80 %, pensez à ajouter des brokers.
- **Usage disque** : Surveiller le pourcentage utilisé et anticiper
- **Usage CPU** : Rarement la source du problème à part si usage intensif de la compression
- **Traffic réseau** : Une utilisation élevée peut être le symptôme d'une performance dégradée (retransmissions TCP, perte de paquets).



# Métriques JVM

---

Le principal métrique à surveiller concerne la collecte et les temps de pause qu'elle peut imposer.

Les métriques JMX concernés sont :

*java.lang:type=GarbageCollector,name=G1 (Young|Old) Generation : **CollectionTime/CollectionCount***

Remède : Upgrade du JDK et de l'algorithme de collecte, Tuning des options JVM



# Métriques producteur

Les processus de production expose des métriques JMX permettant de surveiller le débit :

- ***compression-rate-avg*** : Taux de compression moyen des lots envoyés  
=> Plus c'est faible plus c'est performant.  
=> Si cette métrique augmente, peut indiquer un problème avec la forme des données
- ***response-rate*** : Moyenne du nombre de réponses reçues par seconde  
Dépend de la configuration *ack*
- ***request-rate*** : Moyenne du nombre de requêtes envoyées par seconde  
Un taux élevé sans limitation de quota peut surcharger les brokers
- ***request-latency-avg*** : Moyenne de la latence des requêtes en ms  
Dépendant de *linger.ms* et *batch.size*
- ...





# Métriques producteur (2)

---

- ***outgoing-byte-rate*** : Moyenne du nombre d'octets envoyés par seconde  
Permet de déterminer si il faut améliorer le réseau et à identifier les sources de trafic excessif.
- ***io-wait-time-ns-avg*** : Moyenne du temps passé par la thread d'IO en attente de socket (en ns)  
Si excessif, peut être améliorer les vitesses de disques
- ***batch-size-avg*** : Le nombre moyen d'octets envoyé par requête  
Si différent de la configuration `batch.size`, diminuer `linger.ms`

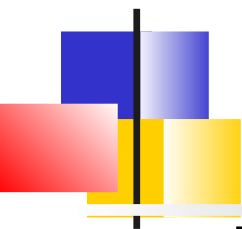


# Métriques consommateur

---

Les consommateurs expose des métriques JMX permettant de surveiller le débit et la performance :

- **records-lag** : Retard par rapport à la production  
=> Pb pour le temps-réel
- **records-lag-max** : Retard Max  
=> Pb pour le temps-réel
- **bytes-consumed-rate** : Nombre moyen d'octets consommés par seconde
- **records-consumed-rate** : Nombre moyen de messages consommés par seconde
- **fetch-rate** : Nombre de requêtes fetch par secondes  
=> Si tend vers zéro pb sur le consommateur



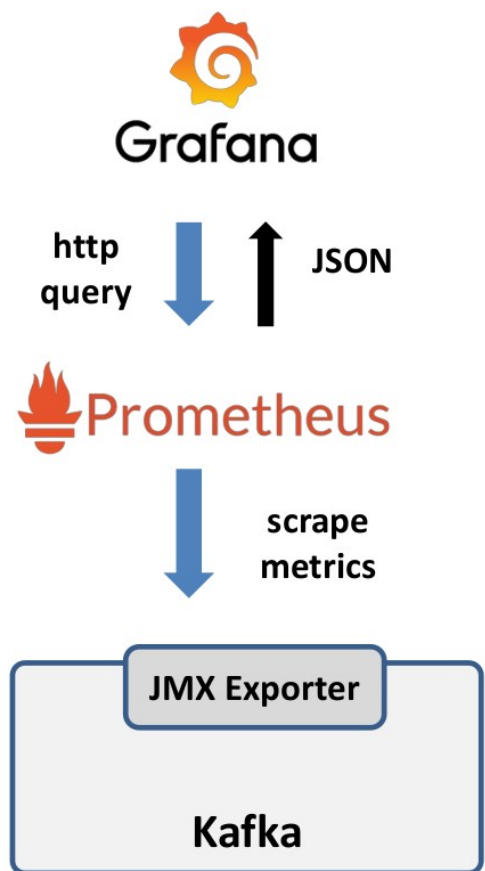
# Collecte et visualisation

---

Pour visualiser les métriques JMX de Kafka, plusieurs outils

- JConsole
- API JMX
- Burrow (LinkedIn) dédié aux consommateurs
- Confluent Control Center
- Graphite
- DataDog
- SMM of Data Flow (Hortonworks)
- Prometheus / Grafana

# Prometheus/Grafana



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)

Tdb Grafana dispo :

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



# Annexes

---

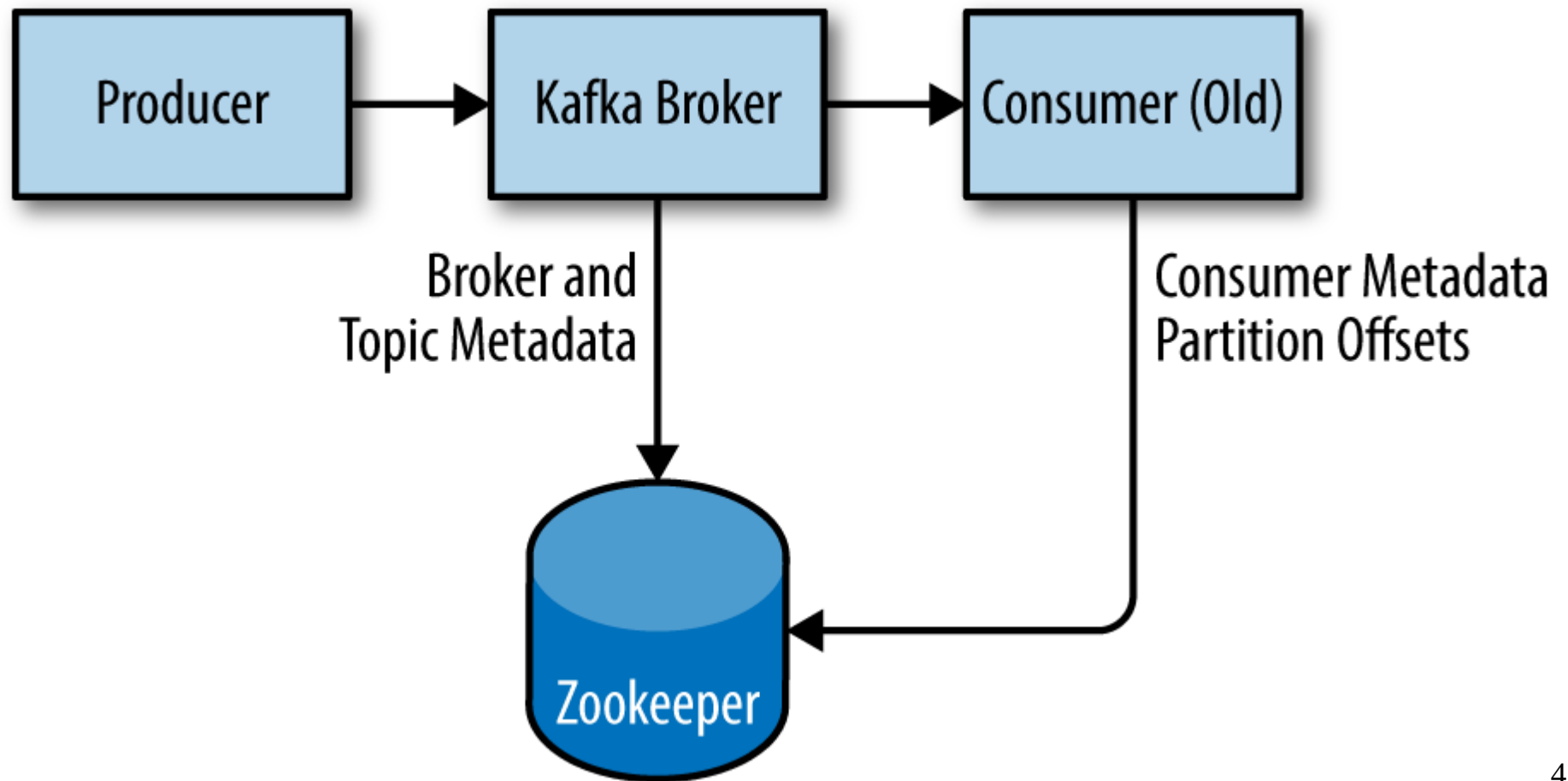
## **Apache Zookeeper**

Contrôleur et Zookeeper

SASL SCRAM

Dimensionnement ZooKeeper

# Kafka et Zookeeper



# Zookeeper

## Principes

*“High-performance coordination service for distributed applications”*

Utilisé par Kafka pour la gestion de configuration et la synchronisation

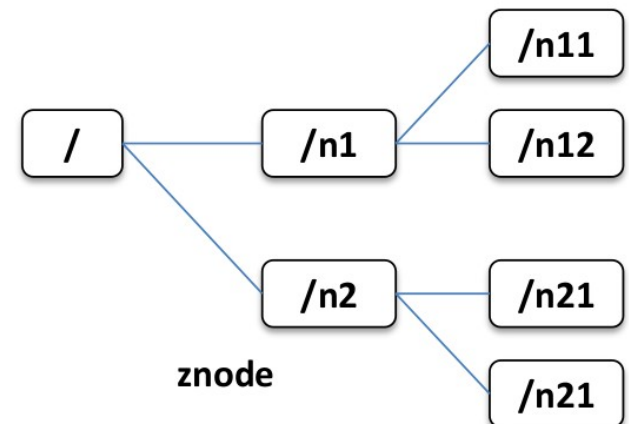
Stocke les méta-données du cluster Kafka

Répliqué sur plusieurs hôtes formant un *ensemble*

Fournit un espace de noms hiérarchiques

Exemples de nœuds pour Kafka :

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





# Vocabulaire Zookeeper

---

**Nœud (zNode)** : Donnée identifiée par un chemin

**Client** : Utilisateur du service Zookeeper

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

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

**Watch** :

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





# *Zookeeper* Distribution

---

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

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



# Exemple installation standalone

---

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



# Vérification Zookeeper

---

```
# telnet localhost 2181
```

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

```
Connected to localhost.
```

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

```
srvr
```

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

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

```
Received: 1
```

```
Sent: 0
```

```
Connections: 1
```

```
Outstanding: 0
```

```
Zxid: 0x0
```

```
Mode: standalone
```

```
Node count: 4
```

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

```
#
```



# Ensemble Zookeeper

---

Un cluster Zookeeper est appelé un **ensemble**

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

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

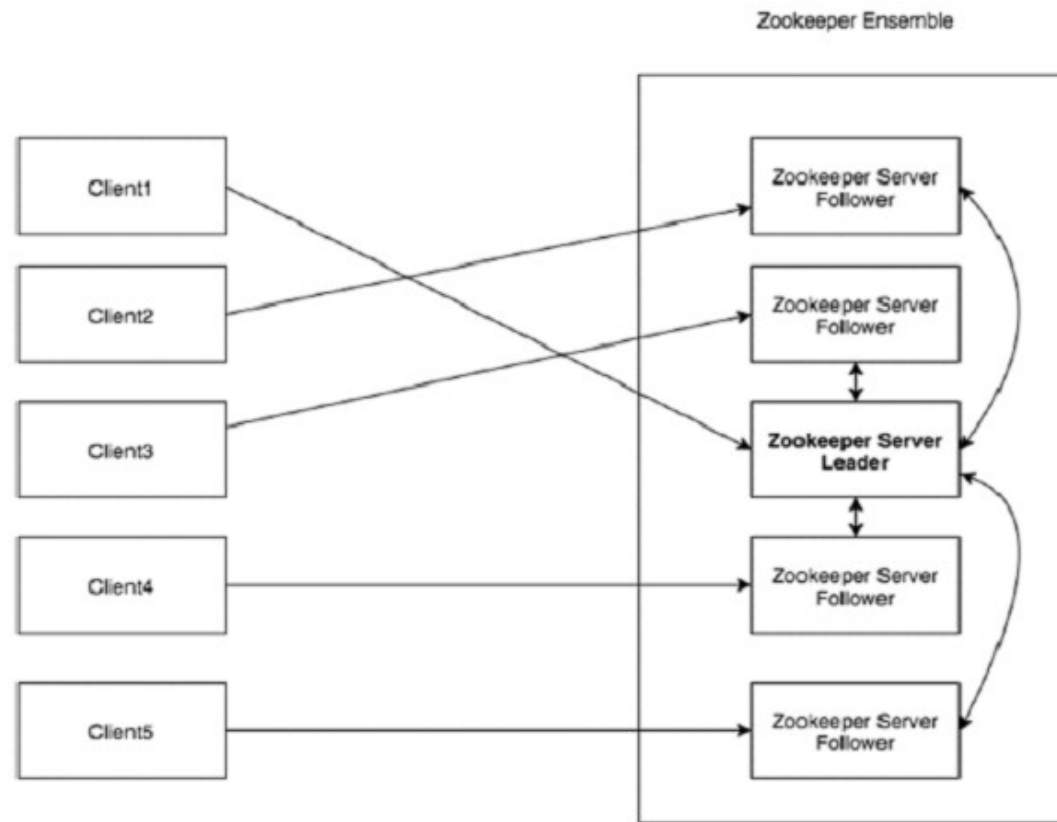
– Le nombre dépend du niveau de tolérance aux pannes  
voulu :

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

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

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

# Architecture *ZooKeeper*





# Propriétés de config

---

**tickTime** : L'unité de temps en ms

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

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

**dataDir** : Répertoire de stockage des données

**clientPort** : Port utilisé par les clients

La configuration répertorie également chaque sous la forme :

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

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

Optionnel :

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



# Configuration d'un ensemble

---

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

## Exemple de config :

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



# 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
```





# Quelques commandes

---

***ls [path]*** : Lister un zNode

***create [zNode]*** : Créer un nœud

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

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

***history*** : Historique des commandes

***quit*** : Quitter zkCli



# 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***.



# Configuration cluster

---

3 contraintes pour la configuration d'un cluster :

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



# Nombre de brokers

---

Pour déterminer le nombre de brokers :

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



# 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.



# Annexes

---

Apache Zookeeper  
**Contrôleur et Zookeeper**  
SASL SCRAM



# Rôles du contrôleur

---

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

Pour le visualiser :

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

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

Si le contrôleur échoue:

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





# Responsabilités

---

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

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

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



# Annexes

---

Apache Zookeeper  
Contrôleur et Zookeeper  
**SASL SCRAM**



# SASL SCARM

---

Utilisé avec SSL pour une  
authentification sécurisée

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



# Configuration des créden*t*iels SASL SCRAM

---

Communication Inter-Broker : user “admin”

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

```

Communication Client-Broker : user “user”

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

```



# Configuration du broker

---

Créer le fichier Jaas

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

Ajouter un paramètre JVM

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

*server.properties*

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



# Configuration du client

---

Créer le fichier Jaas

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

Ajouter un paramètre JVM

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

*client.properties*

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



# Administration

---

Gestion des topics  
Stockage et rétention des partitions  
Quotas  
Gestion du cluster  
**Dimensionnement**  
Monitoring



# 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





# Zookeeper

---

Un nombre impair de serveurs  
*Zookeeper*

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