Apache Kafka

Mohamed El Marouani

TDIA 2

Event Streaming

Dans un monde de plus en plus connecté, les systèmes doivent traiter des flux continus de données en temps réel.

Events Streaming est une approche qui permet de **collecter**, **traiter** et **analyser** des événements au fur et à mesure qu'ils se produisent.

Cette technologie est au cœur des architectures modernes **réactives**, **scalables** et **orientées données**, utilisée dans des domaines comme :

- Les plateformes de streaming vidéo.
- La surveillance en cybersécurité.
- Le traitement des transactions financières.
- L'IoT et les objets connectés.

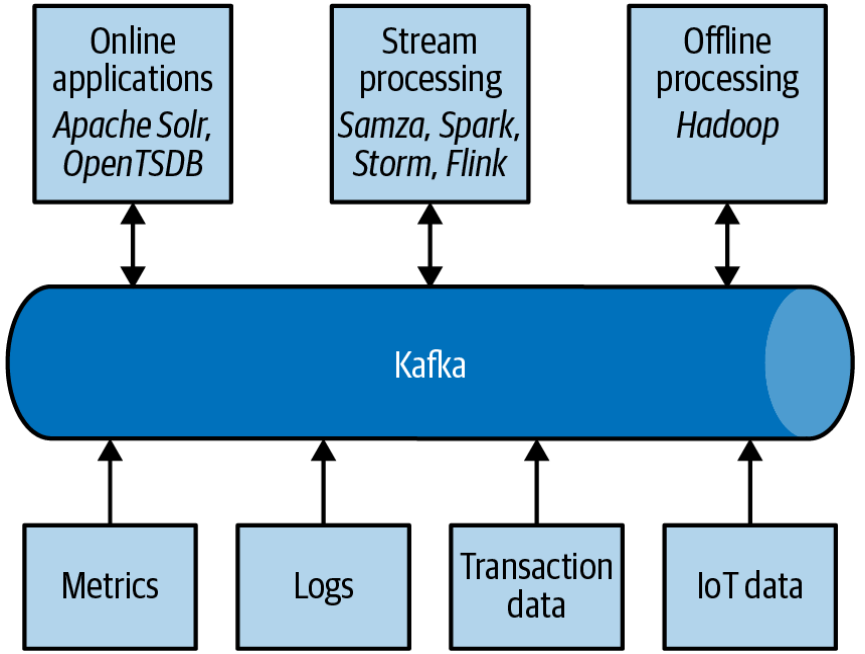
Elle repose sur des solutions comme **Apache Kafka**, **Pulsar** ou **Kinesis**, et change notre manière de penser les systèmes : de statiques à dynamiques, de batch à stream.

Kafka: Introduction

- **Apache Kafka** est un framework open-source distribué conçu pour le streaming d'événements à grande échelle.
- Initialement développé par **LinkedIn** en 2011, il est aujourd'hui un standard industriel pour la gestion de flux de données en temps réel.
- Kafka réunit trois **capacités** clés dans une seule solution éprouvée :
  - **Publier et consommer** des flux d'événements, y compris l'import/export continu de données depuis d'autres systèmes.
  - **Stocker** ces flux de manière fiable et durable, aussi longtemps que nécessaire.
  - **Traiter** les événements en temps réel ou a posteriori.

Kafka: Introduction

- Kafka offre une infrastructure **distribuée**, **scalable**, **élastique**, **tolérante aux pannes** et **sécurisée**.
- Kafka peut être **déployé** :
  - sur des serveurs physiques, machines virtuelles ou conteneurs
  - en local (on-premise) ou dans le cloud
  - en mode autogéré ou via des services managés



Kafka APIs

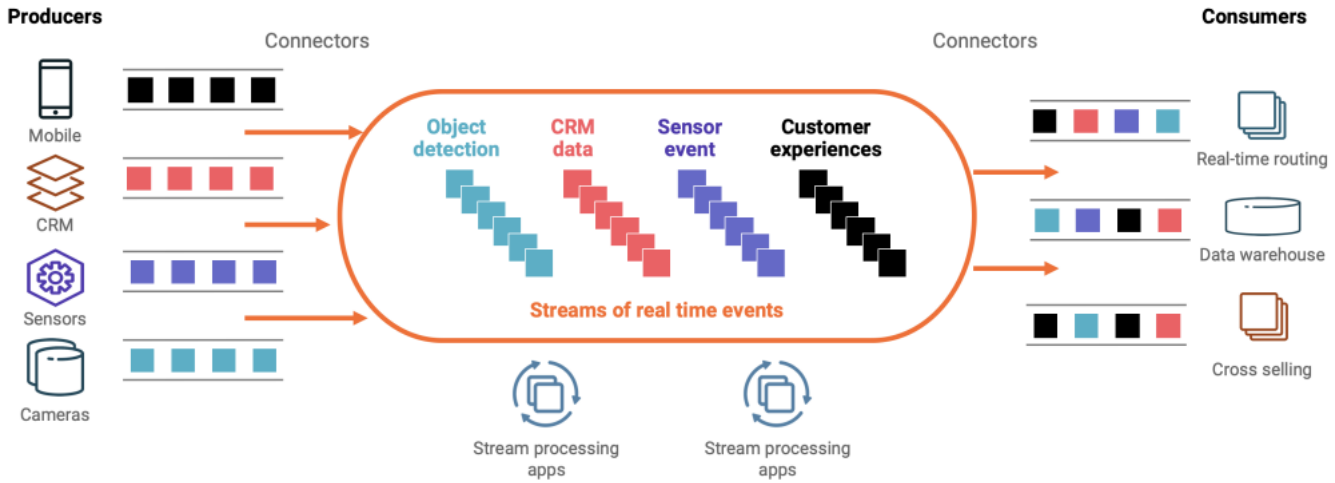
Kafka fournit plusieurs APIs permettant d'interagir avec les flux de données à chaque étape du pipeline :

- **Producer API**
  - Permet d'envoyer des événements dans des topics Kafka.
  - Utilisé par les applications génératrices de données (ex. logs, capteurs IoT, applications web).
- **Consumer API**
  - Permet de lire des événements depuis des topics.
  - Utilisé pour construire des systèmes de traitement ou de réaction aux événements.
- **Streams API**
  - API de traitement intégré au client Kafka, orientée microservices.
  - Permet de transformer, agréger, filtrer et joindre des flux en temps réel.
  - Basée sur une logique déclarative avec une sémantique "event-at-a-time".

Kafka APIs

- **Kafka Connect API**
  - Facilite la connexion de Kafka à des bases de données, systèmes de fichiers, cloud services, etc.
  - Repose sur des connecteurs prêts à l'emploi (source ou sink).

- **Admin API**
  - Permet de créer, configurer et gérer des topics, des quotas, etc.
  - Utile pour l'automatisation et la supervision de l'infrastructure Kafka.



Kafka: Concepts et terminologie

- **Event**
  - Unité de donnée transmise dans Kafka.
  - Composé généralement d'une clé, d'une valeur, d'un horodatage et de métadonnées.
  - Exemple : {"user_id": 123, "action": "login"}
- **Topic**
  - Canal de communication nommé dans lequel les événements sont publiés.
  - Divisé en partitions pour la scalabilité et la parallélisation.
  - Les events sont ordonnés par partition.
- **Partition**
  - Une sous-division d'un topic ; chaque partition contient une séquence ordonnée d'événements.
  - Permet le traitement parallèle et la montée en charge.
- **Offset**
  - Identifiant unique de chaque événement dans une partition.
  - Utilisé pour garder la position de lecture du consommateur.

Kafka: Concepts et terminologie

- **Le modèle Pub/Sub**

Le modèle Pub/Sub est un style de communication **asynchrone** dans lequel les producteurs (appelés publishers) envoient des messages à un canal nommé (ex. un topic) sans se soucier de qui les recevra.
Les consommateurs (ou subscribers) s'abonnent à ces canaux pour recevoir les messages pertinents, de manière découplée.

Kafka implémente le modèle Pub/Sub de façon **distribuée**, avec persistance des messages et gestion avancée des offsets.

Kafka: Architecture

Kafka repose sur une architecture distribuée composée de deux grandes catégories d'acteurs :

- **Serveurs (Côté Cluster Kafka)**
  - **Brokers :** Nœuds qui reçoivent, stockent et distribuent les événements.
  - **Topics & Partitions :** Les données sont organisées par sujets, découpés en partitions pour la scalabilité.
  - **ZooKeeper / KRaft :** Gère la coordination, l'équilibrage de charge et la tolérance aux pannes (ZooKeeper remplacé progressivement par KRaft).
- **Clients (Côté Applications)**
  - **Producers :** Publient des événements dans un topic Kafka.
  - **Consumers :** Lisent les événements d'un ou plusieurs topics.
  - **Kafka Streams :** Clients intelligents qui consomment, traitent et republient des flux.
  - **Kafka Connect :** Connecte automatiquement Kafka à des systèmes externes (BD, fichiers, cloud, etc.).

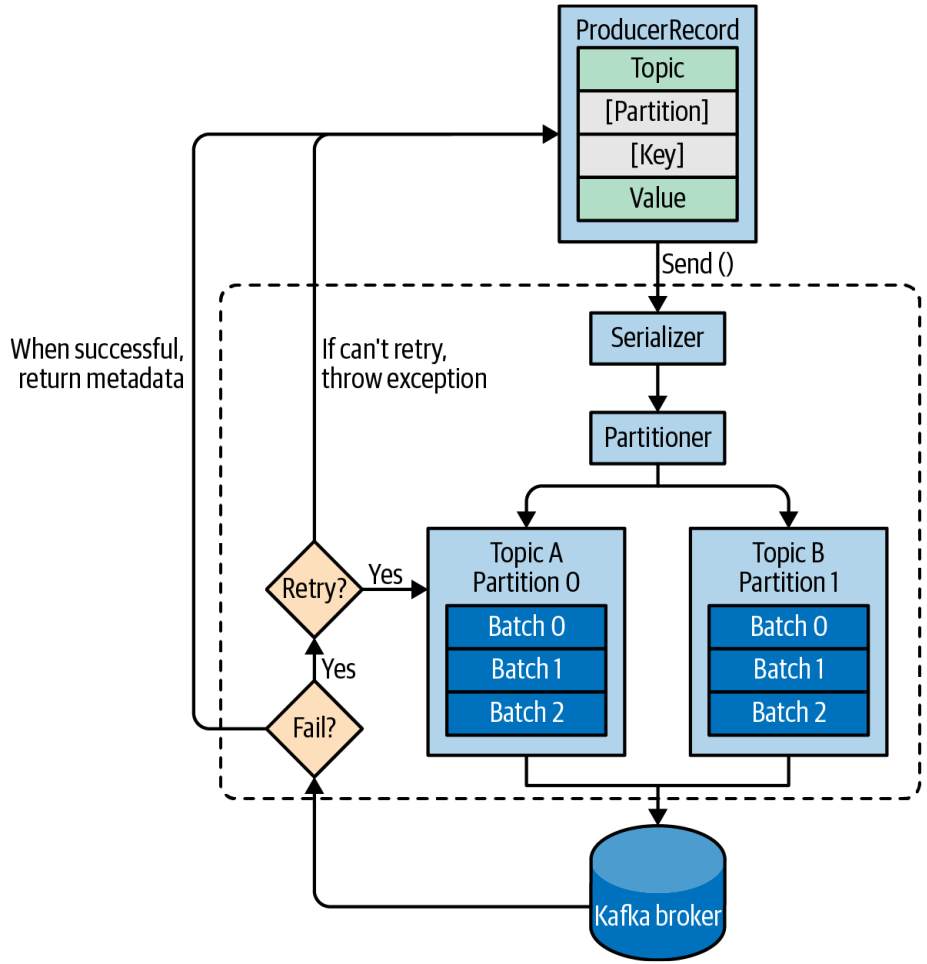
Kafka permet de découpler producteurs et consommateurs, et d'assurer une haute disponibilité grâce à la réplication et à une architecture résiliente.

Apache Kafka vs RabbitMQ

Critère	Apache Kafka	RabbitMQ
Modèle	Distributed Log (log distribué d'événements)	Message Broker (file d'attente orientée message)
Persistance	Stocke les messages durablement sur disque (log)	Messages stockés mais orienté queue, non log
Rétention des messages	Messages conservés pour une durée définie	Supprimés dès qu'ils sont consommés (par défaut)
Performance	Très haut débit (millions de msg/sec)	Débit plus limité, latence faible
Scalabilité	Conçu pour la scalabilité horizontale	Scalabilité plus difficile sans plugins
Modèle Pub/Sub	Pub/Sub natif avec topics & partitions	Pub/Sub via exchanges et bindings
Cas d'usage typique	Event streaming, ETL temps réel, traitement de logs	File d'attente, communication interservices (RPC)
API & Clients	APIs Java natives, Kafka Streams, Connect	Clients AMQP dans plusieurs langages
Écosystème	Kafka Connect, Kafka Streams, KSQL, Confluent, Flink...	Plugins AMQP, Shovel, Federation, UI intégrée
Gestion de l'état	Suivi de l'offset côté consommateur	Suivi de l'état géré par le broker

Kafka: Producers

- Lorsqu'on envoie un message à Kafka via un **Producer**, on commence par créer un **ProducerRecord**. Ce record doit obligatoirement contenir :
  - le topic cible,
  - une valeur (le message).
- On peut aussi ajouter **optionnellement** :
  - une clé,
  - une partition spécifique,
  - un horodatage,
  - et/ou des en-têtes personnalisés.



Kafka: Producers

Étapes principales :

1. **Sérialisation** :
   - Kafka convertit la clé et la valeur en tableaux d'octets pour les transmettre via le réseau.
2. **Choix de la partition** :
   - Si aucune partition n'est définie, un **partitionneur** détermine automatiquement à quelle partition envoyer le message (souvent basé sur la clé).
3. **Batching** :
   - Kafka groupe les messages destinés à la même partition dans un batch avant de les envoyer.
4. **Transmission** :
   - Un **thread séparé** se charge d'envoyer ces batches aux brokers Kafka.
5. **Réponse du broker** :
   - En cas de succès → le broker renvoie un objet **RecordMetadata** (contenant le topic, la partition et l'offset).
   - En cas d'échec → le broker renvoie une **erreur**, et le producer peut effectuer **des tentatives de renvoi (retries)** avant d'abandonner.

Producers: Construction

Avant d'envoyer des messages à Kafka, il faut **créer un objet producer** avec une configuration adaptée. Il existe **trois propriétés obligatoires** :

1. bootstrap.servers
   - Liste d'adresses hôte:port des **brokers Kafka** pour établir une première connexion.
   - Il **suffit d'en fournir une partie**, car le producer découvre les autres après connexion.
   - Il est conseillé d'en mettre **au moins deux** pour assurer une **tolérance aux pannes**.
2. key.serializer
   - Classe responsable de **sérialiser les clés** des messages en **byte arrays**.
   - Kafka attend des tableaux d'octets, mais on peut envoyer n'importe quel objet Java.
   - Le producer doit donc savoir comment **convertir l'objet clé**.
   - Il faut indiquer une classe qui implémente l'interface Serializer de Kafka.
   - Exemples inclus : StringSerializer, IntegerSerializer, ByteArraySerializer, etc.
   - Même si vous n'envoyez pas de clé, vous devez quand même définir cette propriété (ex. avec VoidSerializer).

Producers: Construction

3. value.serializer
   - Même principe que pour key.serializer, mais pour la **valeur du message**.
   - Permet de transformer l'objet à envoyer en **byte array**.
   - Doit être défini pour garantir que Kafka puisse comprendre les messages produits.

Note : Ces propriétés sont essentielles pour garantir la bonne communication entre l'application et le cluster Kafka.

```
Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

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

Producers: Construction

Une fois le **producer Kafka instancié**, on peut envoyer des messages de **trois façons principales** :

1. **Fire-and-Forget (Envoyer sans retour)**
   - Le message est envoyé **sans attendre de confirmation**.
   - Simple et rapide, mais **aucune garantie de livraison** en cas d'erreur non-récupérable ou de timeout.
   - Kafka est très disponible, donc cela fonctionne bien dans la majorité des cas.
   - Les erreurs ne sont **pas remontées à l'application**.

2. **Synchronous Send (Envoi synchrone)**
   - Le producer reste **asynchrone en interne**, mais on appelle .get() sur le Future retourné par send() pour **attendre le résultat**.
   - Cela permet de savoir si le message a bien été délivré **avant de passer au suivant**.
   - Fiable, mais **plus lent** que l'envoi asynchrone.

3. **Asynchronous Send with Callback (Envoi asynchrone avec rappel)**
   - On appelle send() en fournissant une **fonction de rappel (callback)**.
   - Cette fonction est exécutée **dès que Kafka répond** (succès ou échec).
   - Permet une exécution non bloquante tout en **gérant les erreurs proprement**.

Producers: Construction

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

Producers: Construction

Envoi synchrone d'un message Kafka:

L'envoi **synchrone** permet au producer :

- de **capturer les exceptions** (ex. : erreurs de Kafka ou échec après plusieurs tentatives),
- mais il présente un inconvénient majeur : **la performance**.

- **Inconvénient principal :**
  - Le thread reste **bloqué** en attendant la réponse du broker Kafka (de **2 ms à plusieurs secondes** selon la charge).
  - Pendant ce temps, il ne peut **rien faire d'autre** (pas même envoyer d'autres messages).
  - Cela entraîne une **performance très faible**, raison pour laquelle cette méthode **n'est pas utilisée en production**, mais **très fréquente dans les exemples pédagogiques**.

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

Producers: Construction

Envoi asynchrone dans Kafka:

L'envoi **asynchrone** est **rapide et efficace**, surtout lorsque l'on n'a pas besoin d'attendre les réponses de Kafka.

- **Comparaison avec l'envoi synchrone :**
  - Si le **temps réseau aller-retour** est de 10 ms, envoyer 100 messages **en attendant à chaque fois** prend ~1 seconde.
  - Si on **envoie tout sans attendre**, c'est quasi instantané.
  - En général, l'application n'a **pas besoin de la réponse** (topic, partition, offset) **mais doit savoir si une erreur est survenue**.

- **Solution : Ajouter un callback**
  - Permet d'**envoyer les messages sans blocage** tout en gérant les erreurs.
  - Le callback est exécuté **à la réception de la réponse Kafka**, avec ou sans erreur.

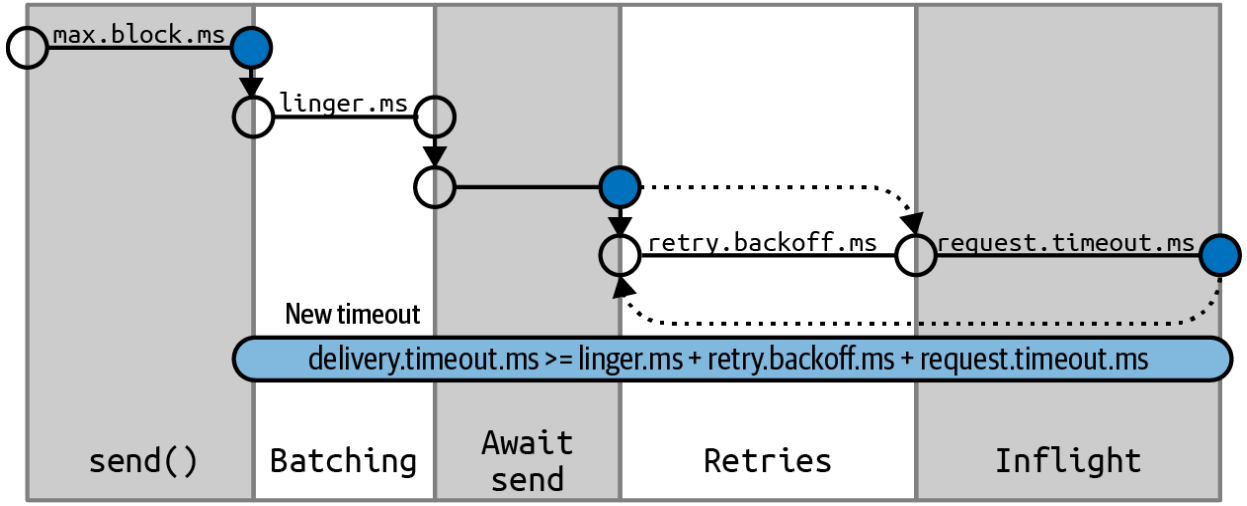
```
private class DemoProducerCallback implements Callback { ❶
    @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {
        if (e != null) {
            e.printStackTrace(); ❷
        }
    }
}

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

Producers: Temps de livraison des messages

Depuis Apache Kafka 2.1, ce temps est divisé en deux phases distinctes :

- Temps jusqu'au retour de l'appel asynchrone à send() :
  - Pendant cette phase, **le thread de l'application est bloqué** (le temps de mise en lot du message, par ex.).
- Temps entre le retour de send() et le déclenchement du callback :
  - Correspond au délai entre le moment où le message a été **placé dans un batch** à envoyer, et la **réponse du broker Kafka** :
    - ✓ succès ,
    - ✓ erreur non-récupérable ,
    - ✓ ou expiration du délai d'envoi maximal configuré.



Producers: Serializers

Kafka requiert des **sérialiseurs** pour convertir les objets Java en tableaux d'octets avant envoi. Kafka fournit des sérialiseurs par défaut (ex : chaînes, entiers, tableaux d'octets), mais pour des objets plus complexes, vous devrez créer **votre propre sérialiseur** ou utiliser une bibliothèque de sérialisation.

Custom Serializer

Prenons une classe simple **Customer** avec un ID et un nom. Pour l'envoyer via Kafka, on crée un sérialiseur personnalisé : CustomerSerializer.

```
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() {
        return customerID;
    }

    public String getName() {
        return customerName;
    }
}
```

Producers: Serializers

Extrait du fonctionnement :

- L'objet Customer est converti en un tableau d'octets (byte[]) via un ByteBuffer.
- Format de sérialisation :
  - 4 octets : ID client
  - 4 octets : taille du nom (en UTF-8)
  - N octets : le nom lui-même

Limites :

- Le code est fragile : tout changement dans la classe (ex : ajouter un champ) casse la compatibilité.
- Le débogage des problèmes de compatibilité entre versions est complexe.
- Si plusieurs équipes utilisent Kafka, elles doivent toutes maintenir la même logique de sérialisation/désérialisation.

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer<Customer> {

    @Override
    public void configure(Map configs, boolean isKey) {
        // nothing to configure
    }
```

Producers: Serializers

Recommandations:

- Éviter les sérialiseurs personnalisés quand possible.
- Préférer des bibliothèques standardisées et robustes comme **Avro, Protobuf, Thrift ou JSON**.
- **Avro** est fortement recommandé : il gère le schéma, l'évolution de données, la compatibilité et permet une meilleure interopérabilité.

```
**/
public byte[] serialize(String topic, Customer data) {
    try {
        byte[] serializedName;
        int stringSize;
        if (data == null)
            return null;
        else {
            if (data.getName() != null) {
                serializedName = data.getName().getBytes("UTF-8");
                stringSize = serializedName.length;
            } else {
                serializedName = new byte[0];
                stringSize = 0;
            }
        }

        ByteBuffer buffer ⊨ ByteBuffer.allocate(4 + 4 + stringSize);
        buffer.putInt(data.getID());
        buffer.putInt(stringSize);
        buffer.put(serializedName);

        return buffer.array();
    } catch (Exception e) {
        throw new SerializationException(
            "Error when serializing Customer to byte[] " + e);
    }
}

@Override
public void close() {
    // nothing to close
}
}
```

Producers: Sérialisation avec Apache Avro

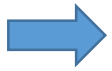
Apache **Avro** est un format de sérialisation de données **neutre vis-à-vis du langage**. Conçu par Doug Cutting (créateur de Hadoop), il facilite le partage de fichiers de données avec un grand nombre d'utilisateurs.

- Les données sont décrites à l'aide d'un **schéma JSON**.
- La sérialisation est **binaire** (ou JSON si besoin).
- Le schéma est **inclus avec les données** pour permettre lecture/écriture sans ambiguïté.

Pourquoi Avro avec Kafka ?
Avro est particulièrement adapté à Kafka car il **gère l'évolution du schéma**. Lorsqu'un producteur change de schéma de manière *compatible*, les consommateurs n'ont **pas besoin d'être mis à jour** pour continuer à traiter les messages.

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "faxNumber", "type": ["null", "string"], "default": "null"}
 ]
}
```

Original schema



```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
     {"name": "email", "type": ["null", "string"], "default": "null"}
 ]
}
```

New schema

Producers: Sérialisation avec Apache Avro

Compatibilité des applications

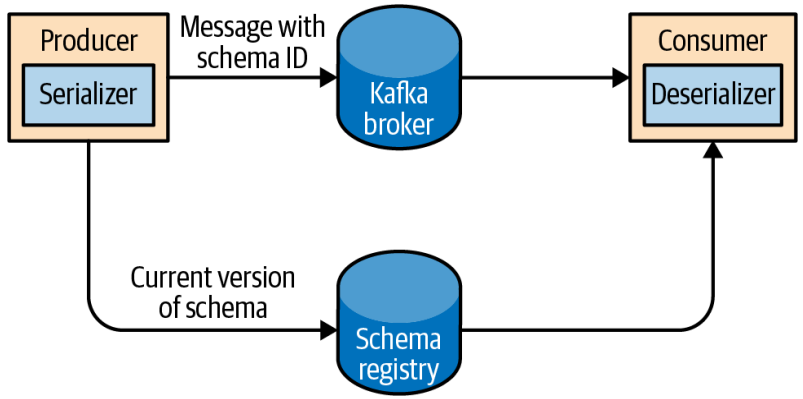
- **Ancienne application** : peut lire les nouveaux messages, mais getFaxNumber() renverra null car ce champ n'existe plus.
- **Nouvelle application** : peut lire les anciens messages, mais getEmail() renverra null car l'email n'était pas présent à l'époque.
- Résultat : **aucune erreur bloquante**, même si les schémas sont différents mais compatibles.

Producers: Sérialisation avec Apache Avro

- **Problème : stocker le schéma dans chaque message**
  - Dans un **fichier Avro**, inclure le schéma complet ne pose pas trop de problème.
    Mais dans **Kafka**, inclure le schéma dans chaque message **doublerait sa taille**, ce qui est inefficace.
  - **Solution : utiliser un *Schema Registry***

- **Schema Registry – Comment ça marche ?**
  - Le **Schema Registry** est un **service externe** (pas intégré à Kafka) qui stocke tous les schémas utilisés.
  - Lorsqu'un message est produit :
    On **n'envoie pas le schéma complet**, mais simplement **un identifiant du schéma**.
  - Le consommateur utilise cet identifiant pour **récupérer le schéma** dans le registry et **désérialiser le message**.
  - **Toute cette logique est gérée automatiquement** par les sérialiseurs/désérialiseurs Avro. Le producteur Kafka n'a rien à faire de spécial, il utilise juste un **AvroSerializer** comme un sérialiseur classique.

- **Exemple utilisé : Confluent Schema Registry**
  - Outil open-source (ou intégré à la plateforme Confluent).
  - Permet de stocker, versionner et gérer les schémas utilisés pour Kafka.
  - La documentation Confluent est recommandée pour la mise en œuvre.

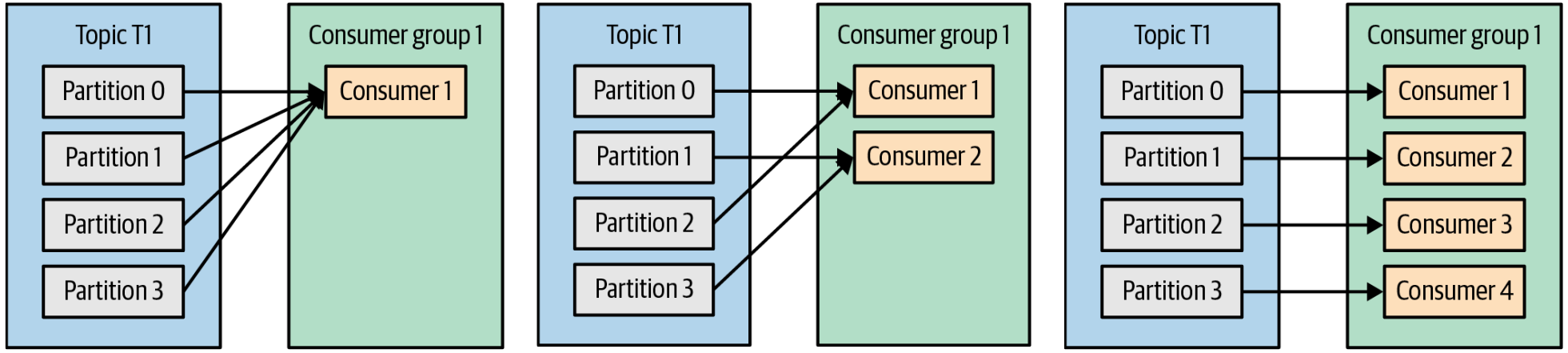


Kafka: Consumers

- L'API Kafka Consumers est une interface fournie par Apache Kafka pour consommer des messages publiés dans des topics Kafka. Un consumer est un client Kafka qui se connecte à un ou plusieurs topics, lit les messages disponibles dans ces topics et les traite.
- **Problème :** Si le flux de messages est élevé, un consommateur unique peut être submergé et accumuler un retard important.
- Les consumers sont regroupés dans des consumer groups. Chaque consumer d'un groupe lit les messages d'un ensemble de partitions spécifique, garantissant ainsi que chaque message est consommé par un seul consumer du groupe.

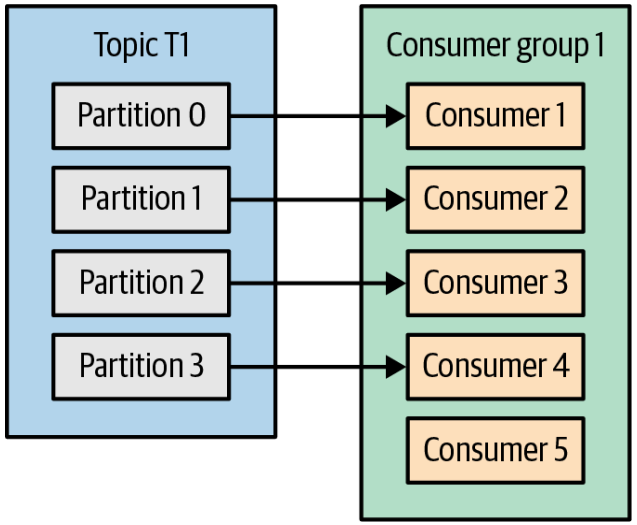
Kafka Consumers: Concepts

- **Consumer Groups:**
  - Un groupe de consommateurs permet de distribuer le traitement des messages entre plusieurs consommateurs.
  - Chaque consommateur du groupe reçoit les messages d'un sous-ensemble des partitions d'un topic.



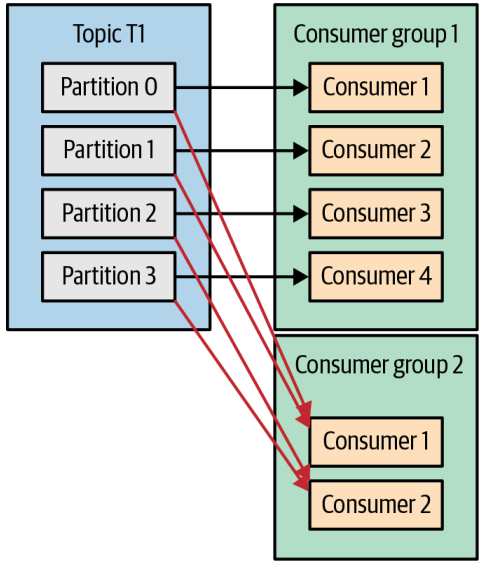
Kafka Consumers: Concepts

- Pour augmenter la capacité de traitement, on peut ajouter des consommateurs à un groupe.
- Limite : Le nombre de partitions d'un topic détermine le nombre maximal de consommateurs actifs.
- Plus le nombre de partitions est élevé, plus il est possible d'ajouter des consommateurs.
- Exemple : Un topic avec 8 partitions peut supporter jusqu'à 8 consommateurs actifs dans un groupe.
- Attention : Si un groupe a plus de consommateurs que de partitions, certains resteront inactifs.



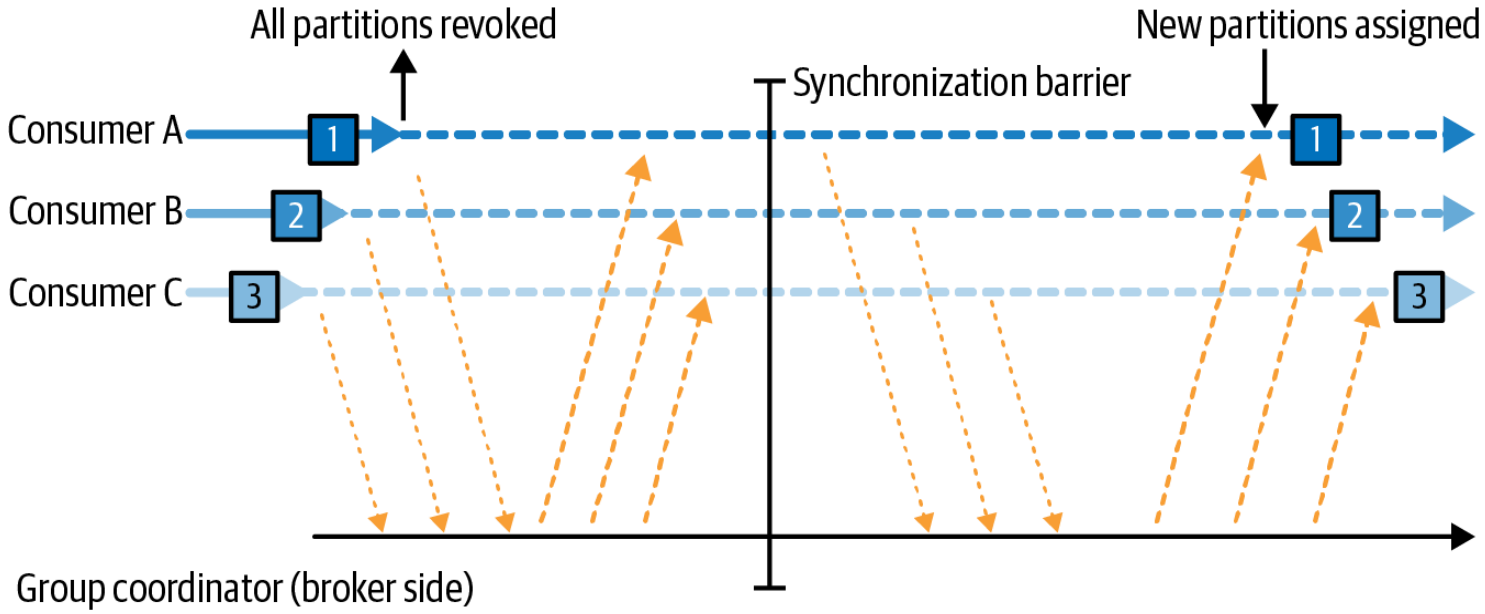
Kafka Consumers: Concepts

- Créer des groupes de consommateurs distincts pour chaque application qui doit traiter tous les messages.
- **Exemple :**
  - Application A utilise le groupe G1 pour traiter le topic T1.
  - Application B utilise le groupe G2 pour traiter le même topic T1.
  - Les consommateurs de G1 et G2 travaillent indépendamment sans interférer.
- **Avantage :** Permet d'isoler le traitement des données pour différentes applications tout en partageant le même topic.



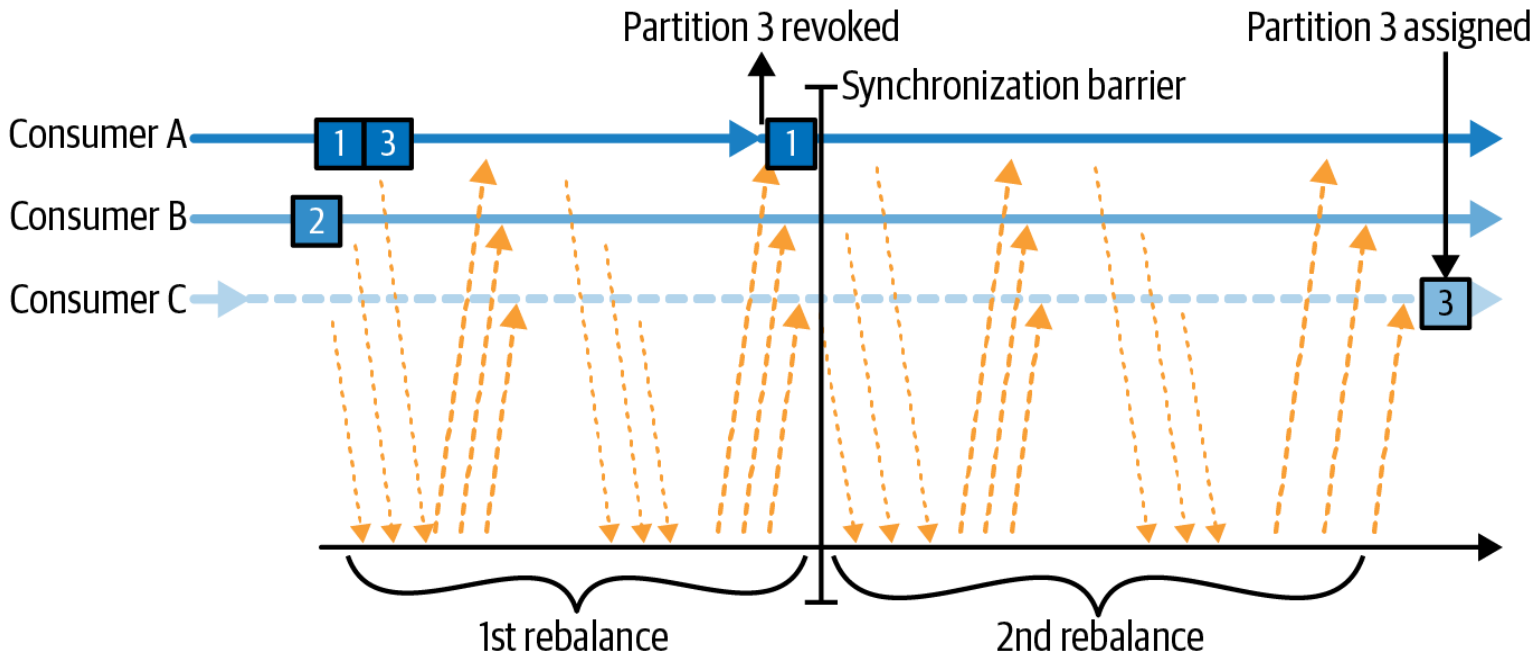
Kafka Consumers: Rebalancement des partitions

- Lorsqu'un consommateur rejoint ou quitte un groupe, les partitions sont réassignées automatiquement.
- **Objectif :** Garantir la haute disponibilité et l'équilibre des charges.
- Types de rebalancements :
  - **Eager Rebalance** : Tous les consommateurs stoppent, libèrent leurs partitions, puis récupèrent de nouvelles partitions. Risque de latence accrue.



Kafka Consumers: Rebalancement des partitions

- **Cooperative Rebalance** : Redistribution progressive des partitions pour éviter les interruptions complètes. Recommandé pour les grands groupes.



Kafka Consumers: Rebalancement des partitions

- Les consommateurs envoient des heartbeats pour signaler qu'ils sont actifs.
- Le coordinateur de groupe surveille ces heartbeats pour détecter les pannes.
- Si un consommateur cesse d'envoyer des heartbeats, le coordinateur le considère comme inactif et déclenche un rebalancement.
- Paramètres essentiels :
  - **heartbeat.interval.ms** : Intervalle entre deux heartbeats.
  - **session.timeout.ms** : Durée maximale sans heartbeat avant qu'un consommateur soit considéré comme mort.

Kafka Consumers: Adhésion statique des groupes

- Par défaut, un consommateur est identifié de manière temporaire dans un groupe.
- Avec group.instance.id, un consommateur devient un membre statique :
  - Ses partitions ne sont pas réassignées après un redémarrage.
  - Il conserve ses partitions tant que le délai session.timeout.ms n'est pas dépassé.
- **Avantage** : Maintien de l'état local (cache, traitement en cours).
- **Inconvénient** : En cas de redémarrage long, risque de retard important.

- **Recommendations** :
  - Utiliser cooperative rebalance pour minimiser les interruptions.
  - Configurer group.instance.id pour les applications nécessitant des états locaux persistants.
  - Prévoir un nombre de partitions suffisant pour permettre une montée en charge progressive.
  - Surveiller les heartbeats et ajuster les timeouts pour équilibrer réactivité et robustesse.

Création d'un Kafka Consumer

Pour consommer des enregistrements dans Kafka, la première étape consiste à créer une instance de **KafkaConsumer**. La création d'un KafkaConsumer est similaire à celle d'un KafkaProducer : il faut d'abord créer une instance de Properties pour définir les propriétés à utiliser. Les trois propriétés obligatoires pour démarrer sont :

1. **bootstrap.servers** : la chaîne de connexion au cluster Kafka (comme pour le producteur).
2. **key.deserializer** : classe pour convertir les clés des enregistrements de tableau d'octets en objets Java.
3. **value.deserializer** : classe pour convertir les valeurs des enregistrements de tableau d'octets en objets Java.

Une quatrième propriété, non obligatoire mais couramment utilisée, est **group.id**, qui indique le groupe de consommateurs auquel appartient l'instance de KafkaConsumer.

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer =
    new KafkaConsumer<String, String>(props);
```

Consumers: Souscription à un topic

- Après avoir créé un **KafkaConsumer**, l'étape suivante consiste à s'abonner à un ou plusieurs topics.
  La méthode subscribe() accepte une liste de topics en paramètre.

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

- Il est également possible d'utiliser une expression régulière avec subscribe(). Cela permet de s'abonner à plusieurs topics dont les noms correspondent au motif spécifié. Si un nouveau topic correspondant au motif est créé, un **rééquilibrage** se produit immédiatement et les consommateurs commencent à consommer les données de ce nouveau topic.

- Cette méthode est particulièrement utile pour les applications qui consomment des données de plusieurs topics ou pour les applications de traitement de flux.

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

Consumers: la boucle poll()

Au cœur de l'API Consumer se trouve une boucle simple qui interroge le serveur pour obtenir des données. Le code typique d'un consommateur Kafka ressemble à ceci :

```
Duration timeout = Duration.ofMillis(100);
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(timeout);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %d, offset = %d, " +
                          "customer = %s, country = %s\n",
                          record.topic(), record.partition(), record.offset(),
                          record.key(), record.value());

        int updatedCount = 1;
        if (custCountryMap.containsKey(record.value())) {
            updatedCount = custCountryMap.get(record.value()) + 1;
        }
        custCountryMap.put(record.value(), updatedCount);
        JSONObject json = new JSONObject(custCountryMap);
        System.out.println(json.toString());
    }
}
```

Cette boucle est infinie car les consommateurs sont généralement des applications **longues durées** qui consomment continuellement des données de Kafka.

Consumers: la boucle poll()

- **Importance de poll() :**
  - Les consommateurs doivent **appeler poll() en permanence** pour être considérés comme actifs.
  - Si poll() n'est pas invoqué pendant plus de max.poll.interval.ms, le consommateur est marqué comme inactif et ses partitions sont réassignées à un autre consommateur.
  - Le paramètre timeout détermine le temps d'attente si aucun enregistrement n'est disponible dans le buffer.
- **Fonctionnement de poll() :**
  - poll() retourne une liste de **ConsumerRecords**.
  - Chaque enregistrement contient : le topic, la partition, l'offset, la clé et la valeur.
  - Le traitement des enregistrements consiste généralement à écrire les résultats dans une base de données ou à mettre à jour un enregistrement existant.
- **Gestion des rééquilibrages :** Le premier appel à poll() gère :
  - La recherche du **GroupCoordinator**.
  - L'adhésion au groupe de consommateurs.
  - L'assignation des partitions.
  - La gestion des rééquilibrages.

Configuration des consommateurs

Les consommateurs Kafka peuvent être configurés à l'aide de plusieurs propriétés. Voici les principales configurations :

- **fetch.min.bytes :**
  - Définit la **quantité minimale de données** que le consommateur souhaite recevoir du broker lors d'un poll().
  - Par défaut : 1 octet.
  - Augmenter cette valeur peut **réduire la charge CPU** en cas de faible activité sur le topic, mais peut également augmenter la latence.
- **fetch.max.wait.ms :**
  - Temps d'attente maximal pour obtenir fetch.min.bytes.
  - Par défaut : 500 ms.
  - Un compromis entre **latence et volume de données** récupérées.
  - Exemple : Si fetch.min.bytes = 1 MB et fetch.max.wait.ms = 100 ms, Kafka renverra les données soit après 1 MB de données, soit après 100 ms, selon la première condition atteinte.
- **fetch.max.bytes :**
  - Quantité maximale de données retournées par poll().
  - Par défaut : 50 MB.
  - Contrôle la mémoire utilisée par le consommateur.

Configuration des consommateurs

- **max.poll.records :**
  - Nombre maximal de **messages traités par appel à poll()**.
  - Permet de **limiter le traitement** des données lors de chaque itération de la boucle poll().
- **max.partition.fetch.bytes :**
  - Limite la quantité de données par partition lors d'un poll().
  - Par défaut : 1 MB.
  - Recommandé d'utiliser fetch.max.bytes pour un meilleur contrôle.

Gestion des temps et des délais:

- **session.timeout.ms :** Temps pendant lequel le consommateur peut rester inactif sans être considéré comme mort.
- **heartbeat.interval.ms :** Fréquence d'envoi des heartbeats.
- **max.poll.interval.ms :** Durée maximale entre deux appels poll() avant que le consommateur ne soit considéré comme mort.
- **default.api.timeout.ms :** Délai par défaut pour **toutes les requêtes API** du consommateur.
- **request.timeout.ms :** Délai maximal pour attendre une réponse du broker.

Configuration des consommateurs

Gestion des offsets :

- **auto.offset.reset :** Comportement lorsque le consommateur **ne trouve pas d'offset valide** :
  - "latest" (par défaut) : commence à lire les nouveaux messages.
  - "earliest" : commence à lire depuis le début du topic.
  - "none" : lève une exception si l'offset est invalide.
- **enable.auto.commit :**
  - Contrôle le **commit automatique des offsets**.
  - Par défaut : true.
  - Pour un contrôle manuel des offsets, définir sur false.

Configuration des consommateurs

Stratégies d'assignation des partitions

- **Range :**
  - Attribue les partitions de manière **consécutive**.
  - Peut créer des déséquilibres si le nombre de partitions n'est pas divisible par le nombre de consommateurs.
- **RoundRobin :**
  - Attribue les partitions de manière **circulaire**, garantissant une répartition plus équilibrée.
- **Sticky :**
  - Objectif : **minimiser le déplacement des partitions** lors des rééquilibrages.
- **Cooperative Sticky :**
  - Identique à Sticky, mais permet aux consommateurs de **continuer à consommer les partitions non réassignées**.

Configuration des consommateurs

Autres propriétés importantes

- **client.id :** Identifiant du consommateur, utilisé dans les logs et les métriques.
- **client.rack :** Permet d'indiquer le datacenter ou la zone où se trouve le consommateur.
- **group.instance.id :** Identifiant unique d'un consommateur pour une **adhésion statique au groupe**.
- **receive.buffer.bytes / send.buffer.bytes :** Taille des buffers TCP. À ajuster pour des communications inter-datacenters.
- **offsets.retention.minutes :** Durée pendant laquelle les offsets sont conservés après que le groupe soit devenu inactif.
  - Par défaut : 7 jours.

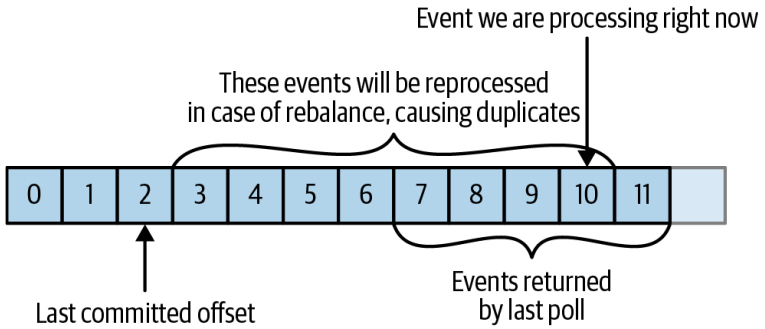
Consumers: Commits & Offsets

Lorsque le consommateur appelle poll(), il reçoit les enregistrements non encore lus par le groupe de consommateurs. Kafka permet aux consommateurs de suivre leur position dans chaque partition à l'aide des offsets, mais il ne gère pas les accusés de réception.

L'action d'actualiser la position dans une partition est appelée **commit d'offset**. Les offsets sont stockés dans le topic spécial __consumer_offsets. En cas de crash ou d'ajout d'un nouveau consommateur, un rééquilibrage se produit et les consommateurs se basent sur les offsets précédemment commités pour reprendre la consommation.

Risques :

- Si le dernier offset commité est inférieur à l'offset du dernier message traité, certains messages seront traités deux fois.
- Si le dernier offset commité est supérieur à l'offset du dernier message traité, certains messages seront perdus.



Consumers: Commits & Offsets

1. **Commit automatique :**
   - En activant enable.auto.commit=true, le consommateur commit automatiquement l'offset toutes les 5 secondes (par défaut).
   - Problème : Si le consommateur plante avant le prochain commit, des messages peuvent être traités deux fois.
2. **Commit manuel des offsets actuels :**

Pour plus de contrôle, il est possible de désactiver le commit automatique (enable.auto.commit=false) et de commiter manuellement à l'aide de commitSync().

Exemple:

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, offset = %d\n", record.topic(), record.offset());
    }
    consumer.commitSync();
}
```

Limite : Si le commit est effectué avant le traitement complet des messages, certains d'entre eux peuvent être ignorés en cas de crash.

Consumers: Commits & Offsets

3. Commit asynchrone:

Pour éviter le blocage de l'application pendant le commit, Kafka propose commitAsync().

- Avantage : Non bloquant, améliore le débit.
- Inconvénient : En cas de défaillance, il ne réessaie pas, ce qui peut causer des duplications ou des pertes de messages.
- Exemple avec callback :

```
consumer.commitAsync(new OffsetCommitCallback() {
    public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception e) {
        if (e != null) log.error("Commit échoué pour les offsets {}", offsets, e);
    }
});
```

4. Combinaison des commits synchrone et asynchrone:

Il est recommandé de combiner commitAsync() pour le traitement normal et commitSync() juste avant la fermeture du consommateur pour garantir la persistance des offsets.

Consumers: Commits & Offsets

5. Commit d'offsets spécifiques:

Pour des cas où il est nécessaire de commiter des offsets avant la fin d'un batch, il est possible de passer une map d'offsets spécifiques :

- Exemple:

```
Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records) {
        currentOffsets.put(
            new TopicPartition(record.topic(), record.partition()),
            new OffsetAndMetadata(record.offset() + 1)
        );
        consumer.commitAsync(currentOffsets, null);
    }
}
```

Cette approche permet un contrôle fin des engagements et réduit le risque de pertes ou de duplications de messages.

Consumers: Deserializers

- Dans Kafka, les producteurs utilisent des **sérialiseurs** pour convertir des objets en tableaux d'octets avant de les envoyer à Kafka. De même, les consommateurs utilisent des **désérialiseurs** pour convertir ces tableaux d'octets en objets Java.
- Dans les exemples précédents, nous avons utilisé le StringDeserializer par défaut pour les clés et les valeurs des messages. Cependant, pour des objets personnalisés, il est nécessaire de créer des désérialiseurs personnalisés ou d'utiliser des formats standardisés comme **Avro**.
- Il est essentiel que le **sérialiseur utilisé par le producteur** corresponde au **désérialiseur utilisé par le consommateur**. Par exemple, un message sérialisé avec IntSerializer ne pourra pas être désérialisé correctement avec StringDeserializer.
    - Avro et le **Schema Registry** permettent de gérer cette compatibilité en validant les schémas des messages produits et consommés.

Consumers: Deserializers

Voici un exemple de désérialiseur personnalisé pour un objet Customer

```java
import org.apache.kafka.common.serialization.Deserializer;
import org.apache.kafka.common.errors.SerializationException;
import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements Deserializer<Customer> {
    @Override
    public void configure(Map configs, boolean isKey) {}

    @Override
    public Customer deserialize(String topic, byte[] data) {
        try {
            if (data == null) return null;
            if (data.length < 8) throw new SerializationException("Data size too short");

            ByteBuffer buffer = ByteBuffer.wrap(data);
            int id = buffer.getInt();
            int nameSize = buffer.getInt();
            byte[] nameBytes = new byte[nameSize];
            buffer.get(nameBytes);
            String name = new String(nameBytes, "UTF-8");

            return new Customer(id, name);
        } catch (Exception e) {
            throw new SerializationException("Error deserializing", e);
        }
    }

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

```java
public class Customer {
    private int customerID;
    private String customerName;

    public Customer(int ID, String name) {
        this.customerID = ID;
        this.customerName = name;
    }

    public int getID() { return customerID; }
    public String getName() { return customerName; }
}
```

Consumers: Deserializers

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", CustomerDeserializer.class.getName());

KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList("customerCountries"));

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, Customer> record : records) {
        System.out.println("Customer ID: " + record.value().getID());
        System.out.println("Customer Name: " + record.value().getName());
    }
    consumer.commitSync();
}
```

Limite :
Les désérialiseurs personnalisés sont fragiles car ils dépendent fortement de la structure des objets. Les erreurs de compatibilité peuvent survenir si les schémas changent.

Consumers: Deserializers

L'utilisation d'Avro permet de **standardiser les messages** et d'assurer la compatibilité des schémas.
Exemple de désérialisation avec Avro :

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
props.put("specific.avro.reader", "true");
props.put("schema.registry.url", "http://schema-registry-url");

KafkaConsumer<String, Customer> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList("customerContacts"));

while (true) {
    ConsumerRecords<String, Customer> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, Customer> record : records) {
        System.out.println("Customer Name: " + record.value().getName());
    }
    consumer.commitSync();
}
```

- schema.registry.url : Indique l'emplacement du **Schema Registry**.
- KafkaAvroDeserializer : Désérialise les messages Avro.
- specific.avro.reader : Permet d'utiliser des objets Avro générés.