

# Cahier de TP

## « Messagerie distribuée avec Kafka »

### Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOS, Windows 10 (avec Git Bash)
- JDK8+
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode
- Docker, Git

## Atelier 1: Installation

### ***1. Installation Ensemble Zookeeper***

Récupérer une distribution de Zookeeper.

Créer un répertoire (*ZK\_ENSEMBLE*) pour stocker les fichiers de configuration des instance

Y créer 3 sous-répertoire 1,2 et 3

Dans chacun des répertoire créer un sous-répertoire ***data*** y créer un fichier nommé *myid* contenant l'identifiant de l'instance (1,2 et 3)

Copier dans chacun des répertoires le répertoire ***conf*** de la distribution

Dans chacun des répertoire éditer le fichier ***zoo.cfg*** , en particulier

- *dataDir*
- Les ports TCP utilisés

Se mettre au point un fichier sh permettant de démarrer les 3 instances

```
$ZK_HOME/bin/zkServer.sh ---config $ZK_ENSEMBLECONF/1/conf start &&
```

```
$ZK_HOME/bin/zkServer.sh ---config $ZK_ENSEMBLECONF/2/conf start &&
```

```
$ZK_HOME/bin/zkServer.sh ---config $ZK_ENSEMBLECONF/3/conf start
```

### **Données partagées :**

Connecter à l'instance 1 via :

```
./zookeeper-1/bin/zkCli.sh -server 127.0.0.1:2181
```

Créer une donnée et enregistrer un *watcher* :

```
create /dp hello
```

```
get -w /dp
```

Dans un autre terminal, se connecter à l'instance 2 et mettre à jour la donnée partagée

```
./zookeeper-2/bin/zkCli.sh -server 127.0.0.1:2182
```

```
set /dp world
```

Vous devez observer une notification dans le terminal 1

### **Election et quorum**

Afficher les rôles des serveurs avec la commande :

```
echo stat | nc localhost 2181 | grep Mode && echo stat | nc  
localhost 2182 | grep Mode && echo stat | nc localhost 2183 | grep  
Mode
```

Stopper le serveur leader et observer la réélection

Stopper encore un serveur et interroger le serveur restant. L'ensemble n'ayant plus de quorum ne doit pas répondre

Redémarrer un serveur et observer la remise en service de l'ensemble

## **2. Installation broker Kafka**

Récupérer une distribution de *Kafka*

Démarrer un broker via :

```
./kafka-server-start.sh [-daemon] ../config/server.properties
```

Vérifier le bon démarrage via la console

Faites des vérification en créant un topic et envoyant des messages via les utilitaires.

## **3. Mise en place cluster Kafka**

Créer un répertoire **kafka-cluster** et 3 sous-répertoires : **broker-1**, **broker-2**, **broker-3**

Copier le fichier *server.properties* dans les 3 sous-répertoires affecter dans chaque fichier de configuration un identifiant unique au broker (propriété **broker.id**)

Modifier également l'emplacement des logs (**logs.dir**)

Mettre au point un script sh permettant de démarrer les 3 brokers en mode daemon

Visualiser les traces de démarrages :

```
tail -f $KAFKA_HOME/logs/server.log
```

Créer un topic **testing** avec 5 partitions et 2 répliques :

```
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 2 --partitions 5 --topic  
testing
```

Lister les topics du cluster

Démarrer un producteur de messages

```
./kafka-console-producer.sh --broker-list localhost:9092 --topic testing --property "parse.key=true" --property "key.separator=:"
```

Saisir quelques messages

Accéder à la description détaillée du topic

Visualiser les répertoires de logs sur les brokers :

```
./kafka-log-dirs.sh --bootstrap-server localhost:9092 --describe
```

Consommer les messages depuis le début

Dans une autre fenêtre, lister les groupes de consommateurs et accéder au détail du groupe de consommateur en lecture sur le topic ***testing***

## Atelier 2: Producer API

Importer le projet Maven fourni

Le projet est composé de :

- Une classe principale ***KafkaProducerApplication*** qui prend en arguments :
  - ***nbThreads*** : Un nombre de threads
  - ***nbMessages*** : Un nombre de messages
  - ***sleep*** : Un temps de pause
  - ***sendMode*** : Le mode d'envoi : 0 pour Fire\_And\_Forget, 1 pour Synchrone, 2 pour AsynchroneL'application instancie *nbThreads KafkaProducerThread* et leur demande de s'exécuter ; quand toutes les threads sont terminées. Elle affiche le temps d'exécution
- Une classe ***KafkaProducerThread*** qui une fois instanciée envoie *nbMessages* tout les temps de pause selon un des 3 modes d'envoi.  
Les messages sont constitués d'une au format String (*courier.id*) et d'une payload au format JSON (classe *Courier*)
- Le package ***model*** contient les classes modélisant les données
  - ***Position*** : Une position en latitude, longitude
  - ***Courier*** : Un coursier associé à une position
  - ***SendMode*** : Une énumération des modes d'envoi

Compléter les méthodes d'envoi de *KafkaProducerThread*.

Pour cela vous devez :

- Initialiser un *KafkaProducer<String,Courier>* et y positionner des sérialiseurs JSON pour la classe *Courier*
- Construire un *ProducerRecord* pour chaque messages
- Implémenter les 3 méthodes d'envoi

Via les commandes utilitaires de Kafka, vérifier la création du topic et consommer les messages

Supprimer le topic et le recréer avec un nombre de *partitions=3* et un *replication-factor=2*

Envoyer des messages

Construire un jar executable avec :

*mvn package*

# Atelier 3 : Consumer API

## 2.1 Implémentation

Le projet est composé de :

- Une classe principale ***KafkaConsumerApplication*** qui prend en arguments :
  - ***nbThreads*** : Un nombre de threads
  - ***sleep*** : Un temps de pauseL'application instancie *nbThreads* ***KafkaConsumerThread*** et leur demande de s'exécuter. Le programme s'arrête au bout d'un certains temps.
- Une classe ***KafkaConsumerThread*** qui une fois instanciée poll le topic position tout les temps de pause.  
A la réception des messages, il affiche la clé, l'offset et le timesatmp de chaque message. Il met également à jour une Map qui contient le nombre de mise à jour pour chaque coursier
- Le package ***model*** contient les classes modélisant les données
  - ***Position*** : Une position en latitude, longitude
  - ***Courier*** : Un coursier associé à une position

Compléter la boucle de réception des messages

Pour cela, vous devez

- Initialiser un *KafkaConsumer*
- Fournir un Deserialiseur
- Implémenter la boucle de réception

Pour tester la réception, vous pouvez utiliser le programme précédent et le lancer afin qu'il exécute de nombreux message :

Par exemple :

```
producer_home$ java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 100000 500 0
```

## 2.2 Tests

Une fois le programme mis au point, effectuer plusieurs tests

Tester qu'aucun message n'est perdu :

Démarrer le programme avec 1 thread arrêter puis redémarrer avec la même configuration

Tester la réaffectation de partitions :

Démarrer avec 2 threads puis 3 threads et visualiser la répartition des partitions

Démarrer également le programme avec 5 threads

## Atelier 4. Kafka Connect

Objectifs : Déverser le topic *position* dans un index ElasticSearch

### 4.1 Installations ElasticSearch + Connecteur

Démarrer *ElasticSearch* et *Kibana* en se plaçant dans le répertoire du fichier *docker-compose.yml* fourni, puis :

```
docker-compose up -d
```

Récupérer le projet OpenSource ElasticSearchConnector de Confluent et se placer sur une release puis builder.

```
git clone https://github.com/confluentinc/kafka-connect-elasticsearch.git
cd kafka-connect-elasticsearch
git checkout v11.0.3
mvn -DskipTests clean package
```

Copier ensuite toutes les librairies présentes dans ***target/kafka-connect-elasticsearch-11.0.3-package/share/java/kafka-connect-elasticsearch/*** dans le répertoire ***lib*** de Kafka

### 4.1 Configuration Kafka Standalone

Mettre au point un fichier de configuration ***elasticsearch-connect.properties*** contenant :

```
name=elasticsearch-sink
connector.class=io.confluent.connect.elasticsearch.ElasticsearchSinkConnector
tasks.max=1
topics=position
topic.index.map=position:position_index
connection.url=http://localhost:9200
type.name=log
key.ignore=true
schema.ignore=true
```

Démarrer ***kafka-standalone*** et alimenter le topic ***position***

Vous pouvez visualiser les effets du connecteur

- Via ElasticSearch : [http://localhost:9200/position/\\_search](http://localhost:9200/position/_search)
- Via Kibana : <http://localhost:5601>

Optionnel : Améliorer le fichier de configuration afin d'introduire le timestamp

## Atelier 5 : KafkaStream

### Objectifs :

Écrire une mini-application Stream qui prend en entrée le topic ***position*** et écrit en sortie dans le topic ***position-out*** en ajoutant un timestamp aux valeurs d'entrée

Importer le projet Maven fourni, il contient les bonnes dépendances et un package *model* :

- Un champ *timestamp* a été ajouté à la classe *Courier*
- Une implémentation de *Serde* permettant la sérialisation et la désérialisation de la classe *Courier* est fournie

Avec l'exemple du cours, écrire la classe principale qui effectue le traitement voulu

## Atelier 6 : Fiabilité

### **1. At Least Once, At Most Once**

### Objectifs :

Explorer les différentes combinaisons de configuration des producteurs et consommateurs vis à vis de la fiabilité sous différents scénarios de test.

On utilisera un cluster de 3 nœuds avec un topic de 3 partitions et un mode de réplication de 2.

Le scénarios de test envisagé (Choisir un scénario parmi les 2):

- Redémarrage de broker(s)
- Ré-équilibrage des consommateurs

Les différentes combinaisons envisagées

- Producteur : *ack=0* ou *ack=all*
- Consommateur : auto-commit ou commits manuels

Les métriques surveillés

- Producteur : Trace WARN ou +
- Consommateur : Trace Doublet ou messages loupés

### Méthodes :

Supprimer le topic ***position***

Le recréer avec

```
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 2 --partitions 3 --topic verify
```

Vérifier l'affectation des partitions et des répliques via :

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic position
```

Récupérer les sources fournis et construire pour les 2 clients l'exécutable via

mvn clean package

Dans 2 terminal, démarrer 2 consommateurs :

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 1000 >> log1.csv
```

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 1000 >> log2.csv
```

Dans un autre terminal, démarrer 1 producteur multi-threadé :

```
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 20 5000 10 <0|1|2> <0|all>
```

Pendant la consommation des messages, en fonction du scénario : arrêter et redémarrer un broker ou un consommateur.

Un utilitaire **check-logs** est fourni permettant de détecter les doublons ou les offsets.

```
java -jar check-logs.jar <log.csv>
```

## 2. *Exactly One*

Modifier la configuration du producer afin d'autoriser l'idempotence.

Rejouer les scénarios de tests précédents

# Atelier 7 : Administration

Exécuter les producteurs et les consommateurs pendant les opérations d'administration

## 7.1 *Reassign partitions, Retention*

Modifier le nombre de partitions de position à 8

Vérifier avec

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic position
```

Ajouter un nouveau broker dans le cluster.

Réexécuter la commande

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic position
```

Effectuer une réaffectation des partitions en 3 étapes

Visualiser les segments et apprécier la taille

Diminuer le **retention.bytes** à une taille inférieure et visualiser les conséquences



## 7.2 Rolling restart

Sous charge, effectuer un redémarrage d'un broker.

Vérifier l'état de l'ISR

## 7.3 Mise en place monitoring Prometheus, Grafana

Dans un premier temps, démarré une **JConsole** et visualiser les Mbeans des brokers, consommateurs et producteurs

Dans un répertoire de travail **prometheus**

```
wget
https://repo1.maven.org/maven2/io/prometheus/jmx/jmx\_prometheus\_javaagent/0.6/jmx\_prometheus\_javaagent-0.6.jar
```

```
wget
https://raw.githubusercontent.com/prometheus/jmx\_exporter/master/example\_configs/kafka-2\_0\_0.yml
```

Modifier le script de démarrage du cluster afin de positionner l'agent Prometheus :

```
KAFKA_OPTS="$KAFKA_OPTS -javaagent:$PWD/jmx_prometheus_javaagent-0.2.0.jar=7071:$PWD/kafka-0-8-2.yml" \
./bin/kafka-server-start.sh config/server.properties
```

Attention, Modifier le port pour chaque broker

Redémarrer le cluster et vérifier <http://localhost:7071/metrics>

Récupérer et démarrer un serveur prometheus

```
wget
https://github.com/prometheus/prometheus/releases/download/v2.0.0/prometheus-2.0.0.linux-amd64.tar.gz
tar -xzf prometheus-*.tar.gz
cd prometheus-*
cat <<'EOF' > prometheus.yml
global:
  scrape_interval: 10s
  evaluation_interval: 10s
scrape_configs:
  - job_name: 'kafka'
    static_configs:
      - targets:
        - localhost:7071
        - localhost:7072
        - localhost:7073
        - localhost:7074
EOF
./prometheus
```

Récupérer et démarrer un serveur Grafana

```
sudo apt-get install -y adduser libfontconfig1
```

```
wget https://dl.grafana.com/oss/release/grafana_7.4.3_amd64.deb
```

```
sudo dpkg -i grafana_7.4.3_amd64.deb
```

```
sudo /bin/systemctl start grafana-server
```

Accéder à <http://localhost:3000> avec **admin/admin**

Déclarer la datasource Prometheus

Importer le tableau de bord : <https://grafana.com/grafana/dashboards/721>

Les métriques des brokers devraient s'afficher