Cahier de TP
« Administration Cluster Kafka »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux
- JDK11+, Maven
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode
- Docker, Git

Solutions :
https://github.com/dthibau/kafka-admin-solutions

Table des matières

Atelier 1: Mise en place

1.1 Installation cluster

1.1.1 Option archive

Etape 1 : Création du cluster ID et formattage des répertoires de logs

Télécharger et dézipper la distribution de Kafka dans ***$KAFKA_DIST***

Créer un répertoire ***$KAFKA_LOGS*** qui stockera les messages des 3 brokers

Créer un répertoire ***$KAFKA_CLUSTER*** et 3 sous-répertoires : ***broker-1***, ***broker-2***, ***broker-3***

Copier le fichier de la distribution *$KAFKA_DIST/config/kraft/server.properties* dans les 3 sous-répertoires de $***KAFKA_CLUSTER***

Editer les 3 fichier server.properties afin de modifier les propriétés :

- ***node.id***
- ***listeners***
- ***advertised.listeners***
- ***controller.quorum.voters***
- ***log.dir*** à ***$KAFKA_LOGS/broker-<id>***

Générer un id de cluster :

$KAFKA_DIST/bin/kafka-storage.sh random-uuid

Utiliser l'ID du cluster pour formatter les 3 répertoires

Pour chaque broker, formatter son répertoire de log avec :

bin/kafka-storage.sh format -t <cluster_id> -c $KAFKA_CLUSTER/broker-<id>/server.properties

Etape 2 : Mise au point d'un script de démarrage/arrêt

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

Exemple :

```
#!/bin/sh


export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64/
export KAFKA_DIST=/home/dthibau/Formations/Kafka/MyWork/kafka_2.13-3.3.1
export KAFKA_CLUSTER=/home/dthibau/Formations/Kafka/github/solutions/kafka-cluster
export KAFKA_LOGS=/home/dthibau/Formations/Kafka/MyWork/kafka-logs


$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-1/server.properties
$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-2/server.properties
$KAFKA_DIST/bin/kafka-server-start.sh -daemon $KAFKA_CLUSTER/broker-3/server.properties
```

Visualiser les traces de démarrages :

tail -f $KAFKA_HOME/logs/server.log

Mettre au point un script d'arrêt

```
#!/bin/sh


export KAFKA_DIST=/home/dthibau/Formations/Kafka/MyWork/kafka_2.12-2.4.1
$KAFKA_DIST/bin/kafka-server-stop.sh
```

Effectuer les vérifications de création de topic et envoi/réception de messages

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

1.1.2 Option Docker

Visualiser le fichier ***docker-compose.yml***

Démarrer la stack et observer les logs de démarrages

Effectuer les vérifications de création de topic et envoi/réception de messages

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

1.2 Utilitaires

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

*./**kafka-topics.sh** --create --bootstrap-server localhost:9092 --replication-factor 2 --partitions 5 --topic testing*

Lister les topics du cluster

Démarrer un producteur de message

*./**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

Utiliser ***kafka-dump-log.sh*** sur un fichier de log

1.3 Outils graphiques

Installer l'outil graphique de votre choix

Exemple akhq

Télécharger une distribution d'akhq (akhq-all.jar)

Récupérer le fichier de configuration fourni ***application-basic.yml***

Exécuter le serveur via :
java -Dmicronaut.config.files=./application-basic.yml -jar akhq-0.21.0-all.jar

Exemple Kafka Magic :
docker run -d --rm --network=host digitsy/kafka-magic

Exemple Redpanda Console :

```
docker run --network=host \
  -e KAFKA_BROKERS=localhost:9092 \
  docker.redpanda.com/redpandadata/console:latest
```

Atelier 2: Production de messages

Le projet fourni est un projet développé avec le framework Spring.

Il modélise une flotte de 100 coursiers qui envoient 10000 fois leur position (latitude/longitude) toutes les 10 millisecondes.

Lors du démarrage du programme, le producteur affiche sa configuration sur la console et démarre la production de message après que l'on ait appuyé sur la touche Enter.

Le producteur peut être configuré via la ligne de commande en préfixant les propriétés Kafka par *spring.kafka.producer*

Exemple :

java -jar SpringProducer.jar --spring.kafka.producer.acks=0

Créer le topic ***position*** avec 3 partitions et le facteur de réplication par défaut :

./kafka-topics.sh --create --bootstrap-server localhost:9092 --partitions 3 --topic position

Faire plusieurs essais de production de message en réinitialisant le topic et en changeant les paramètres par défaut, Vérifier la configuration à chaque fois

- ***acks***
  Pour comparer les valeurs de débits
- ***batch.size*** et ***linger.ms***

  Pour comparer les valeurs de débits
- ***retries*** / ***max.in.flights.requests.per.session***
  Avec arrêt/redémarrage du cluster ou d'un serveur durant les tests.
  Vérifier le nombre de messages produits

Atelier 3 : Consommation de messages

Le projet fourni est également un projet développé avec le framework Spring.

Il permet de consommer le topic ***position***.

Le nom du groupe est ***consumer***

Il peut être surchargé via la ligne de commande :

--spring.kafka.consumer.group-id

La propriété de configuration ***spring.kafka.listener.concurrency*** fixe le nombre de consommateur du groupe.

Elle est fixée à 3 et peut être surchargée via la ligne de commande.

--spring.kafka.listener.concurrency

Le programme consomme les messages effectue un traitement de 10ms puis affiche :

- En permanence, des informations sur le débit de traitement
- Un message de log si il s'aperçoit que les messages ne sont pas consécutifs (perte de message ou message non-ordonné)

3.1 Performance

Démarrer le programme avec les valeurs de concurrence par défaut :

java -jar SpringConsumer.jar

Visualiser les logs de démarrage de l'application qui affichent :

- La configuration du consommateur
- Les allocations de partition

Supprimer les informations du groupe dans Kafka afin que les consommateurs repartent à zéro.

Modifier des paramètres pouvant affecter la performance et les commits d'offset :

- max.poll.records

3.2 Rebalancing des consommateurs

Démarrer 3 processus avec un degré de concurrence à 1

java -jar SpringConsumer.jar --spring.kafka.listener.concurrency=1

Arrêter un processus et observer la réaffectation de partition

Redémarrer le processus

Atelier 4. Sérialisation Avro

4.1 Démarrage de Confuent Registry

Option archive
Télécharger une distribution de la Confluent Platform version communautaire :
curl -O http://packages.confluent.io/archive/7.2/confluent-community-7.2.1.zip
Dézipper

Démarrer le seveur de registry via :
./schema-registry-start ../etc/schema-registry/schema-registry.properties

Accéder à localhost:8081/subjects

Vous pouvez également configurer le schema registry dans votre outil graphique d'administration

Option Docker

Visualiser le fichier ***docker-compose.yml*** fourni qui démarre RedpandaConsole et SchemaRegistry de confluent

Démarrer la stack

Accéder à *localhost:8085/subjects*

4.2 Producteur de message

Récupérer le projet ***producer-avro***

Visualiser le schéma Avro :
src/main/resources/Courier.avsc

Lancer la commande ./***mvnw compile*** et regarder les classes générées par le plugin Avro

Lancer ensuite ***./mvnw package*** pour construire l'exécutable

Démarrer l'exécutable par
java -jar target/producer-avro-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 500 10 0

Le programme, après avoir enregistré le schéma, envoie 10*500 messages qui sont sérialisés avec Avro sur le topic ***avro-position***

Après un premier démarrage, le schéma doit être consultable sur la registre (localhost:8085/subjects ou via votre outil d'admin)

Vérifier que l'outil graphique puisse lire les messages. (Si option non docker, vérifier la configuration, vous devez indiquer l'URL du registre de schéma dans la configuration)

4.3 Consommateur de message

Récupérer le projet ***consumer-avro***
Visualiser le code source et en particulier la boucle de poll et l'utilisation de GenericMessage

Lancer ensuite ***./mvnw package*** pour construire l'exécutable

Consommer les messages du topic précédent avec la commande
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 2 1

Laisser le programme tourner

4.4 Evolution du schéma compatible

Mettre à jour le schéma en ajoutant les champs optionnels : ***firstName*** dans la structure ***Coursier***

```
{
      "name": "first_name",
      "type": "string",
      "default": "undefined"
},
```

Fixer les problèmes de compilation :
Dans ***src/main/java/org/formation/KafkaProducerThread.java***
Changer la ligne 29 par :

```
this.courier = new Courier(id, "David", new Position(Math.random() + 45,
Math.random() + 2));
```

Reconstruire l'application via :

./mvnw clean package

Relancer le programme de production et visualiser la nouvelle version du schéma dans le registre

Consommer les messages sans modifications du programme consommateur

4.5 Evolution du schéma incompatible

Mettre à jour le schéma en ajoutant un champs obligatoire : ***vehicle_id*** dans la structure ***Coursier***

```
      {
            "name": "vehicle_id",
            "type": "int"
      },
```

Fixer les problèmes de compilation
Dans ***src/main/java/org/formation/KafkaProducerThread.java***
Changer la ligne 29 par :

```
this.courier = new Courier(id, "David", 1, new Position(Math.random() + 45,
Math.random() + 2));
```

Relancer le programme de production et visualiser l'exception au moment de l'enregistrement du nouveau schéma.

Atelier 5 : Garanties de livraison

Reprendre les projets SpringProducer et SpringConsumer fournis

5.1. At Least Once

Objectifs :

Mettre en évidence la garantie *At Least Once.*

On utilisera un cluster de 3 nœuds avec un topic de 3 partitions, un mode de réplication de 2 et un *min.insync.replica* de 1.

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

- Ré-équilibrage des consommateurs

Les métriques surveillés

- Consommateur : Trace Doublon ou messages loupés

Méthodes :

Supprimer le topic ***position***

Le recréer avec

```
./kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 2 --partitions 3 --topic position
```

Vérifier le *min.insync.replicas*

Vérifier l'affectation des partitions et des répliques via :

```
./kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic position
```

Récupérer les programmes fournis

Dans 2 terminal, démarrer 2 consommateurs :

```
java -jar SpringConsumer.jar --spring.kafka.listener.concurrency=1 >> log1.csv
```

```
java -jar SpringConsumer.jar --spring.kafka.listener.concurrency=1 >> log2.csv
```

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

```
java -jar SpringProducer.jar --spring.kafka.producer.enable.idempotence=true
```

Pendant la consommation des messages, arrêter et redémarrer un consommateur (plusieurs fois)

Visualisation résultat :

Concaténer les fichiers résultas :

```
cat log1.csv >> cat log2.csv >> log.csv
```

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

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

Atelier 6. Kafka Connect

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

6.1 Installations ElasticSearch + Connecteur

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

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

Se connecter à kibana (http://localhost:5601) et dans la DevConsole exécuter :

```
PUT /position
{
  "mappings":{
     "properties": {
         "@timestamp": {
             "type": "date",
             "format": "epoch_millis"
         }
     }
 }
}
```

Récupérer le répertoire libs fourni (il a été construit à partir du projet ***ElasticSearchConnector*** de Confluent release v14.5.0 `https://github.com/confluentinc/kafka-connect-elasticsearch.git`)

Copier ensuite toutes les librairies dans le répertoire ***libs*** de Kafka

Redémarrer le cluster

6.2 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***

Alimenter le topic ***position*** en utilisant le programme fourni
java -jar producer-with-dependencies.jar 10 500 10 0

Vous pouvez visualiser les effets du connecteur

- Via ElasticSearch : http://localhost:9200/position/_search
- Via Kibana : http://localhost:5601

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

Atelier 7: Sécurité

7.1 Séparation des échanges réseaux

Créer 3 listeners plain text dénommé PLAIN_TEXT, CONTROLLER et EXTERNAL en leur affectant des ports différents

Pour l'instant utiliser des communications en clair pour les 3

Avec un programme précédent utiliser le listener EXTERNAL

7.2 Mise en place de SSL pour crypter les données

Travailler dans un nouveau répertoire ***ssl***

Générer un keystore et un truststore pour les serveurs en utilisant le script ***generate-ssl.sh*** fourni (Provient de Bitnami)

A la fin de l'opération, vous devez avoir 2 fichiers .jks :

- ***keystore/kafka.keystore.jks*** : Certificats utilisé par les brokers
- ***truststore/kafka.truststore.jks*** : Truststore pour les serveurs

Vous pouvez vérifier les contenus des store avec :

keytool -v -list -keystore server.keystore.jks

Configurer les fichiers *server.properties*

- Ajouter un listener SSL et l'utiliser pour la communication inter-broker
- Utiliser également SSL pour le listener EXTERNAL

Option Archive

Configurer le listener SSL et les propriétés SSL suivante dans *server.properties*

```
ssl.keystore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/server.keystore.jks
ssl.keystore.password=secret
ssl.key.password=secret
ssl.truststore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/server.truststore.jks
ssl.truststore.password=secret
security.inter.broker.protocol=SSL
ssl.endpoint.identification.algorithm=
ssl.client.auth=none
```

Démarrer le cluster kafka et vérifier son bon démarrages
Dans les traces doivent apparaître :

Option Docker

Monter le répertoire ssl sur

```
/opt/bitnami/kafka/config/certs/kafka.truststore.jks
```

Vérifier également l'affichage du certificat avec :

```
openssl s_client -debug -connect localhost:9192 -tls1_2
```

Mettre au point un fichier ***client-ssl.properties*** avec :

security.protocol=SSL
ssl.truststore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/client.truststore.jks
ssl.truststore.password=secret

Vérifier la connexion cliente avec par exemple

./kafka-console-producer.sh --broker-list localhost:EXTERNAL_PORT --topic ssl
--producer.config client-ssl.properties

Ou en utilisant un des programmes clients précédents

7.3 Authentification via SASL/PLAIN

Mettre au point un fichier comme suit :

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

Modifier les fichiers *server.properties* afin que la communication inter-broker utilise le listener ***SASL_SSL***.
Ajouter les configurations :
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN

Redémarrer le cluster

Configurer le listener EXTERNAL pour qu'il utilise également SASL_SSL comme protocole

Mettre à jour le fichier ***client-ssl.properties*** comme suit :
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
*sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required *
*username="alice" *

password="alice-secret";

Tester avec :
./kafka-console-producer.sh --broker-list localhost:EXTERNAL_PORT --topic ssl
--producer.config client-ssl.properties

7.4 ACL

Activer les ACL avec la classe org.apache.kafka.metadata.authorizer.StandardAuthorizer

Définir les super users et la règle allow.everyone.if.no.acl.found=true

Démarrer le cluster

Voir :

https://github.com/bitnami/containers/issues/23237

https://stackoverflow.com/questions/74671787/kafka-cluster-using-kraft-mtls-and-standardauthorizer-not-starting-up-getting

Atelier 8 : Administration

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

8.1 Reassign partitions

Extension du cluster et réassignement des partitions

Modifier le nombre de partitions de position à 8

Vérifier avec

`./kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic position`

Ajouter un nouveau broker dans le cluster. (Ne pas l'inclure dans les contrôleurs possible)

Option Docker : un docker-compose est fourni

Réexécuter la commande

`./kafka-topics.sh --bootstrap-server localhost:909 --describe --topic position`

Effectuer une réaffectation des partitions en 3 étapes

8.2 Rétention

Visualiser les segments et apprécier la taille

Pour le topic ***position*** modifier le ***segment.bytes*** à 512ko

Diminuer le ***retention.bytes*** afin de voir des segments disparaître

8.3 Rolling restart

Sous charge, effectuer un redémarrage d'un broker.

Vérifier l'état des ISR

Atelier 9: *Monitoring avec Prometheus/Grafana*

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

Dans un répertoire de travail ***monitoring***

```
wget https://repo1.maven.org/maven2/io/prometheus/jmx/jmx_prometheus_javaagent/0.18.0/jmx_prometheus_javaagent-0.18.0.jar
wget https://github.com/confluentinc/jmx-monitoring-stacks/raw/6.1.0-post/shared-assets/jmx-exporter/kafka_broker.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_broker.yml" \
```

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/v2.45.0/prometheus-2.45.0.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
```

Se connecter sur ***http ://localhost:9090***

Vérifier les cibles de prometheus

Récupérer et démarrer un serveur Grafana :

Sur Debian/Ubuntu

```
wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -
sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"
sudo apt update
sudo apt install grafana
sudo systemctl start grafana-server
sudo systemctl status grafana-server
```

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

Déclarer la datasource Prometheus

Importer le tableau de bord fourni (Provient de https://github.com/confluentinc/jmx-monitoring-stacks/tree/6.0.1-post/jmxexporter-prometheus-grafana/assets/prometheus/grafana/provisioning/dashboards)

Les métriques des brokers devraient s’afficher

Annexes

1.1. Installation Ensemble Zookeeper

Récupérer une distribution de Zookeeper.

Créer un répertoire (dans la suite nommé *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 script 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 le l’ensemble