

Cahier de TP

« Administration Cluster Kafka »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux
- JDK21+, 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.....	3
1.1 Installation cluster.....	3
1.1.1 Option archive.....	3
1.1.2 Option Docker.....	4
1.2 Utilitaires.....	4
1.3 Outils graphiques.....	5
Atelier 2: Production de messages.....	6
Atelier 3 : Consommation de messages.....	7
3.0 Mise en place BD.....	7
3.1 Performance.....	7
3.2 Rebalancing des consommateurs.....	8
Atelier 4 : Exactly Once et Transactions.....	9
4.1 Transaction et ISOLATION_LEVEL.....	9
4.1.1 Producteur transactionnel.....	9
4.1.2 Consommateur.....	9
4.2 Exactly Once.....	9
Atelier 5. Schema Registry et sérialisation Avro.....	10
5.1 Démarrage de Confluent Registry.....	10
5.2 Producteur de message.....	10
5.3 Consommateur de message.....	10
5.4 Evolution du schéma compatible.....	11
5.5 Evolution du schéma incompatible.....	11
Atelier 6. Kafka Connect.....	12
6.1 Installations ElasticSearch + Connecteur.....	12
6.2 Configuration Kafka Standalone.....	12
6.3 Kafka Mirror.....	13
Atelier 7: Sécurité.....	14
7.1 Séparation des échanges réseaux.....	14
7.2 Mise en place de SSL pour crypter les données.....	14
7.2.1 SSL appliqué aux communications inter-broker.....	14
7.2.1 SSL appliqué aux communications externes.....	15
7.3 Authentification via SASL/PLAIN.....	15
7.3.1 Authentification inter-broker.....	16
10.3.2 Authentification client.....	16

7.4 ACL.....	17
7.4.1 Activation des ACLs.....	17
7.4.2 Définition ACLs.....	17
Atelier 8 : Exploitation.....	18
8.1 Reassign partitions.....	18
8.2 Rétention.....	18
8.3 Quota.....	18
8.4 Rolling restart.....	19
Atelier 9: <i>Monitoring avec Prometheus/Grafana</i>	20
9.1 Exporter les informations JMX au format Prometheus.....	20
9.3 Mise en place Prometheus Grafana.....	20
A. Annexes	22
A.1. Installation Ensemble Zookeeper.....	22

Atelier 1: Mise en place

1.1 Installation cluster

1.1.1 Option archive

Étape 1 : Création du cluster ID et formatage des répertoires de logs

Télécharger et dézipper la distribution de Kafka dans un répertoire nommé **\$KAFKA_DIST** dans la suite

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**

Éditer les 3 fichiers **server.properties** afin de modifier les propriétés :

- **node.id**
- **listeners (utiliser les ports 9092, 9192 et 9292)**
- **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 formater les 3 répertoires

Pour chaque broker, formater son répertoire de log avec :

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

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

Mettre au point un script 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=<répertoire de la distribution>  
export KAFKA_CLUSTER=<répertoire des configurations des noeuds>  
export KAFKA_LOGS=<répertoire des 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_DIST/logs/server.log
```

Mettre au point un script d'arrêt

```
#!/bin/sh
```

```
export KAFKA_DIST=<répertoire de la distribution>
```

```
$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. Les messages sont numérotés par Coursier

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 par la propriété **async** activant ou désactivant le mode d'asynchrone d'envoi et en préfixant les propriétés Kafka par *spring.kafka.producer*

Exemple :

```
java -jar SpringProducer.jar --spring.kafka.producer.acks=0 --async=false
```

Créer le topic **position** avec 5 partitions et le facteur de réplication de 3 et un *min.insync.replica* de 2 :

```
./kafka-topics.sh --create --bootstrap-server localhost:9092 --partitions 5 --replication-factor 3 --config min.insync.replicas=1 --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 consommateurs 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 et stocke dans une base postgres l'id du coursier et l'id de l'index du message:

- En permanence, il affiche des informations sur le débit de traitement
- Si la base est réinitialisée, il détecte les doublons de traitements

3.0 Mise en place BD

```
docker compose -f postgres-docker-compose.yml up -d
```

Accéder à *localhost:81* et se logger avec *admin@admin.com/admin*

Enregistre un serveur avec comme paramètres de connexion :

- host : *consumer-postgresql*
- user : *postgres*
- password : *postgres*

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 : Exactly Once et Transactions

Objectifs :

- Visualiser des messages committés et non-committés
- Voir les effets de la propriété *ISOLATION_LEVEL*
- Implémenter un consommateur *ExactlyOnce*

4.1 Transaction et *ISOLATION_LEVEL*

4.1.1 Producteur transactionnel

Le producteur fournit englobe dans une transaction des batchs de messages de taille 10.

Lancer le programme avec :

```
java -jar producer-tx.jar 1 105 100 0
```

Ce qui signifie que 1 threads envoie 105 messages, les 100 premiers messages font partie de 10 transactions validées. Les 5 derniers messages ne sont pas validés.

Visualiser les messages dans une console d'administration.

Noter le nombre et les offsets, Visualiser le flag transactionnel

4.1.2 Consommateur

Le programme consommateur peut être démarré en mode *READ_COMMITTED* ou *READ_UNCOMMITTED*

Faire un premier démarrage en mode *READ_UNCOMMITTED*

```
java -jar consumer.jar 1 false
```

Noter le nombre de messages consommés

Démarrer le message en mode *READ_COMMITTED*

```
java -jar consumer.jar 1 true
```

Noter le nombre de messages consommés

4.2 Exactly Once

Reprendre le nouveau consommateur.

Ce nouveau programme lit le topic *position-tx*, ajoute une information de distance à l'enregistrement et publie un message vers le topic *position_distance*.

Exécuter le programme :

```
java -jar consumer-exactly-once.jar 1
```

Vérifier les messages produits et les offsets consommés

Atelier 5. Schema Registry et sérialisation Avro

5.1 Démarrage de Confluent Registry

Visualiser le fichier ***docker-compose.yml*** fourni qui démarre Redpanda Console, akhq et SchemaRegistry de confluent

Démarrer la stack via :

```
docker compose up -d
```

Vérifier le bon démarrage des containers

Accéder à *localhost:8085/subjects*

5.2 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 schema-registry (localhost:8085/subjects)
- ou via votre outil d'admin

Vérifier que l'outil graphique puisse lire les messages Avro

5.3 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

5.4 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

java -jar target/producer-avro-0.0.1-SNAPSHOT-jar-with-dependencies.jar 10 500 10 0

Visualiser ensuite la nouvelle version du schéma dans le registre

Consommer les messages sans modifications du programme consommateur

5.5 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 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"
      }
    }
  }
}
```

Recopier le répertoire `$KAFKA_DIST` dans un répertoire `$KAFKA_CONNECT`

Copier ensuite toutes les librairies du répertoire *kafka-connect-elasticsearch*¹ fourni dans le répertoire *libs* de `$KAFKA_CONNECT`

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_CONNECT/bin/kafka-standalone`

1. Il a été construit à partir du projet *ElasticSearchConnector* de Confluent release v14.0.17
<https://github.com/confluentinc/kafka-connect-elasticsearch.git>)

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

6.3 Kafka Mirror

<https://github.com/framiere/mirror-maker-docker-example>

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. Par exemple :

```
java -jar SpringProducer.jar --spring.kafka.producer.bootstrap-servers=localhost:9094
```

et

```
java -jar SpringConsumer.jar --spring.kafka.consumer.bootstrap-servers=localhost:9094
```

7.2 Mise en place de SSL pour crypter les données

Option1 : Générer keystore et truststore

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)

- Lorsque vous êtes invité à saisir un mot de passe, utilisez le même pour tous.
- Définissez les valeurs Common Name ou FQDN sur **localhost**.
- Après avoir saisi cette valeur, lorsque vous êtes invité « Quels sont votre prénom et votre nom ? », saisissez également **localhost**.

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 kafka.keystore.jks
```

Option2 : Récupérer le répertoire ssl

Il doit contenir des certificats pour localhost valable jusqu'en 2033

7.2.1 SSL appliqué aux communications inter-broker

Configurer ensuite les fichiers **server.properties**

- Ajouter un listener SSL et l'utiliser pour la communication inter-broker

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

```
ssl.keystore.location=<path-to>//server.keystore.jks  
ssl.keystore.password=secret
```

```
ssl.key.password=secret
ssl.truststore.location=<path-to>/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 :

```
[2024-05-23 14:07:08,458] INFO [SocketServer listenerType=BROKER,
nodeId=1] Created data-plane acceptor and processors for
endpoint : ListenerName(SSL) (kafka.network.SocketServer)
```

Vous pouvez également vérifier l’affichage du certificat avec :

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

7.2.1 SSL appliqué aux communications externes

Utiliser également SSL pour le listener EXTERNAL, il suffit de modifier la propriété ***listener.security.protocol.map***

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

```
security.protocol=SSL
ssl.truststore.location=<path-to>/kafka.truststore.jks
ssl.truststore.password=secret
```

Vérifier la connexion cliente avec les outils Kafka

```
$KAFKA_DIST/bin/kafka-console-producer.sh --bootstrap-server
localhost:EXTERNAL_PORT --topic ssl --producer.config client-
ssl.properties
```

```
$KAFKA_DIST/bin/kafka-console-consumer.sh --bootstrap-server
localhost:9094 --topic ssl --consumer.config client-ssl.properties
--from-beginning
```

Ou en utilisant un des programmes clients précédents

7.3 Authentication via SASL/PLAIN

7.3.1 Authentication inter-broker

Mettre au point un fichier **kafka_server_jaas.conf** définissant 2 utilisateurs **admin** et **alice** et indiquant que le serveur utilise l'identité admin 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 le script de démarrage afin qu'il utilise le fichier :

```
export KAFKA_OPTS="-Djavax.net.debug=ssl:handshake:verbose  
-Djava.security.auth.login.config=<path-to>/  
kafka_server_jaas.conf"
```

Modifier les fichiers *server.properties* afin que la communication inter-broker utilise le listener **SASL_SSL**.

Vous devez modifier les propriétés de configuration :

- *listeners*
- *inter.broker.listener.name*
- *advertised.listeners*

Ajouter également les configurations :

```
sasl.mechanism.inter.broker.protocol=PLAIN  
sasl.enabled.mechanisms=PLAIN
```

Redémarrer le cluster et vérifier son bon démarrage

10.3.2 Authentication client

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_DIST/bin/kafka-console-producer.sh --bootstrap-server  
localhost:9094 --topic ssl --producer.config client-ssl.properties
```


7.4 ACL

7.4.1 Activation des ACLs

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

```
authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer
```

Définir les super users et la règle

```
allow.everyone.if.no.acl.found=true
```

```
super.users=User:admin
```

Modifier le protocole du listener CONTROLLER à SASL_PLAIN

Indiquer la propriété

```
sasl.mechanism.controller.protocol=PLAIN
```

Ajouter une section dans le fichier JAAS :

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

Démarrer le cluster

7.4.2 Définition ACLs

Définir une ACL via Redpanda Console ou via `$KAFKA_DIST/bin/kafka-acls.sh` interdisant à l'utilisateur ***alice*** d'écrire sur les topics.

Tester le refus d'envoi de message avec :

```
$KAFKA_DIST/bin/kafka-console-producer.sh --bootstrap-server  
localhost:9094 --topic ssl --producer.config client-ssl.properties
```

Atelier 8 : Exploitation

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 Rétenion

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 Quota

Définition d'un quota pour l'utilisateur **alice**

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094  
--alter --add-config  
'producer_byte_rate=1024,consumer_byte_rate=1024' --entity-type  
users --entity-name alice --command-config client-admin-  
ssl.properties
```

Relancer SpringProducer

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094  
--alter --delete-config producer_byte_rate --entity-type users --  
entity-name alice --command-config client-admin-ssl.properties
```

```
$KAFKA_DIST/bin/kafka-configs.sh --bootstrap-server localhost:9094  
--alter --delete-config consumer_byte_rate --entity-type users --
```

```
entity-name alice --command-config client-admin-ssl.properties
```

8.4 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

```
$JAVA_HOME/bin/jconsole &
```

9.1 Exporter les informations JMX au format Prometheus

Dans un répertoire de travail *monitoring*

```
wget
https://repo1.maven.org/maven2/io/prometheus/jmx/jmx\_prometheus\_javaagent/0.20.0/jmx\_prometheus\_javaagent-0.20.0.jar
wget
https://raw.githubusercontent.com/confluentinc/jmx-monitoring-stacks/main/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-0.20.0.jar=7071:$PWD/kafka_broker.yml" \
```

Attention, Modifier le port pour chaque broker

Redémarrer le cluster et vérifier <http://localhost:7071/metrics> (Peut prendre du temps avant de s'afficher)

9.3 Mise en place Prometheus Grafana

Dans le répertoire grafana des données du TP, lancer la stack via :

```
docker compose up -d
```

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

Vérifier les cibles de prometheus

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

Vérifier 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 doivent s'afficher

A. Annexes

A.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