

R5.A.08 :

Qualité de développement

Chouki Tibermacine

Chouki.Tibermacine@univ-ubs.fr



Plan prévisionnel de la ressource

1. Intro aux architectures logicielles
2. Documentation d'architectures en UML
3. Styles et patrons d'architectures
4. Architectures à microservices
5. Design & Implem de frameworks
6. Patrons de conception : Façade, Bridge, MVC et MVVM
7. Patrons de conception : Builder, Proxy et Visitor
8. Autres patrons

Plan prévisionnel de la ressource

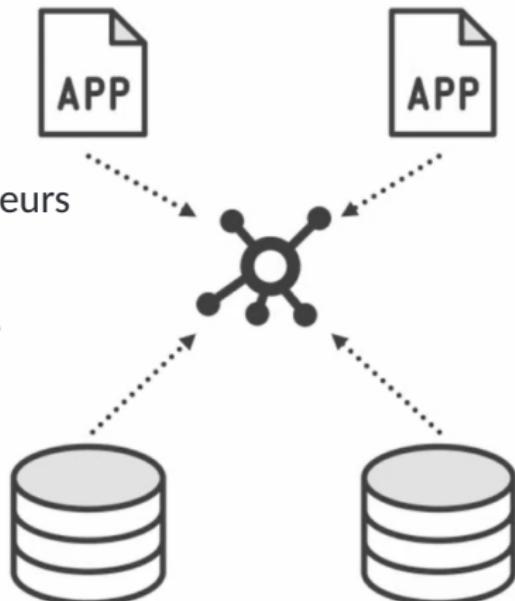
1. Intro aux architectures logicielles
2. Documentation d'architectures en UML
3. Styles et patrons d'architectures – TP
4. Architectures à microservices
5. Design & Implem de frameworks
6. Patrons de conception : Façade, Bridge, MVC et MVVM
7. Patrons de conception : Builder, Proxy et Visitor
8. Autres patrons

Qu'est-ce que "Apache Kafka" ?

- Un système distribué de messagerie publish/subscribe à grand débit
- Spécialement conçu pour un grand débit de messages (des centaines de millions d'utilisateurs et des milliers de milliards de messages par jour)
- A prouvé son efficacité chez LinkedIn, Netflix, Uber, Airbnb, ...
- Garantit une certaine fiabilité en répliquant les messages
- Traditionnellement, on utilisait la réplication de bases de données, mais c'est peu flexible (la migration d'un fournisseur (*vendor*) à un autre ou une évolution des schémas posent des problèmes, parfois complexes)
- Utilisable pour l'interopérabilité entre des sources de données hétérogènes

Un courtier (*broker*) de messages

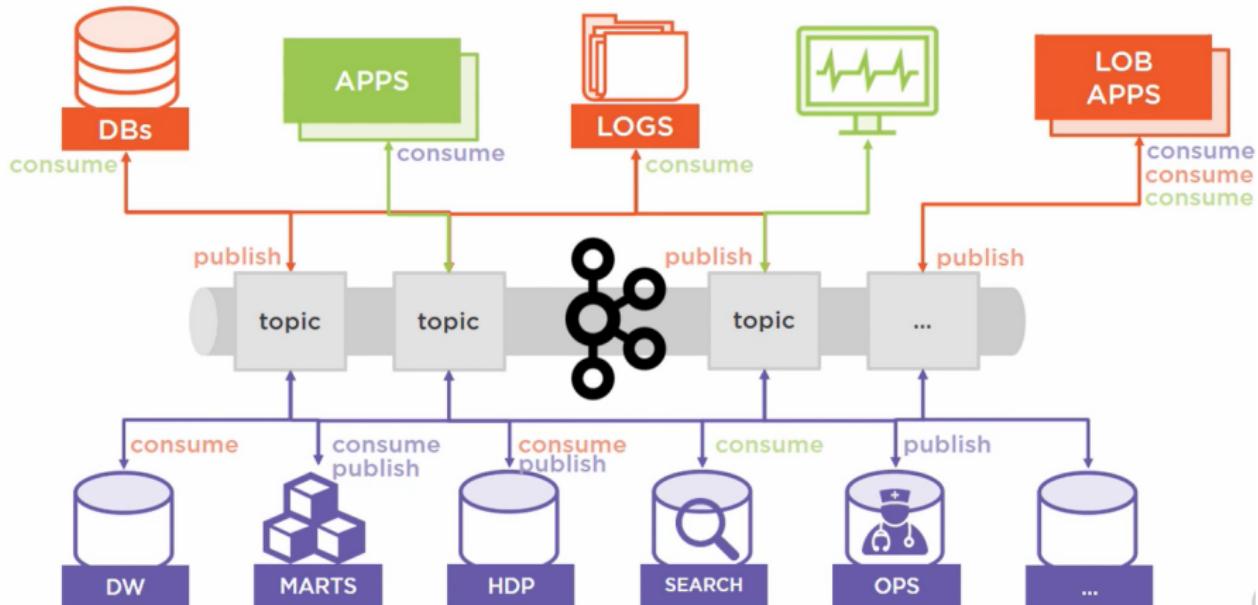
- Permettant de découpler les producteurs et les consommateurs de messages
- Basé sur le pattern publish/subscribe
- Messages durables ou effaçables



LinkedIn avant 2010



LinkedIn après 2010

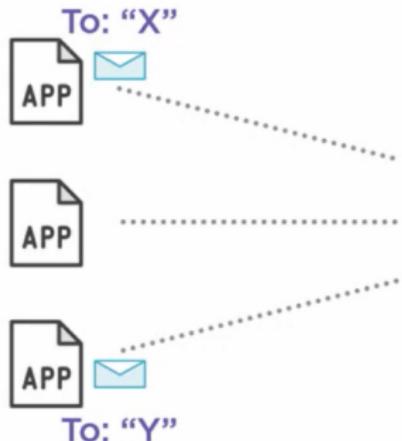


LinkedIn et Kafka

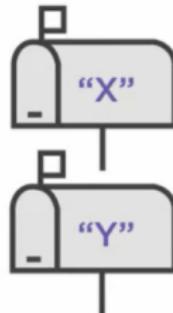
- LinkedIn existe depuis 2003
- Début du développement de Kafka en 2009
- En 2010, c'est opérationnel chez LinkedIn
- En 2011/2012, Kafka est devenu un projet open-source Apache (l'un des plus utilisés de la fondation)
- Aujourd'hui, LinkedIn gère des milliers de milliards de messages / jour avec Kafka

Producers (publishers), Consumers (subscribers) & Topics

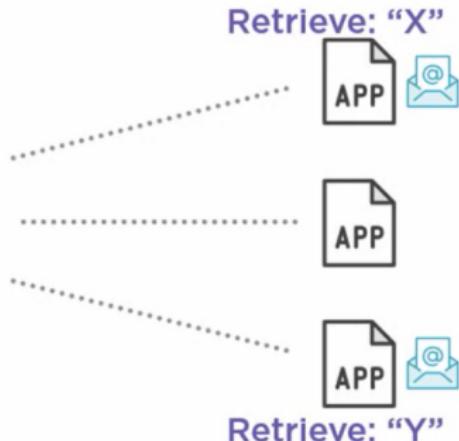
Producers



Topics



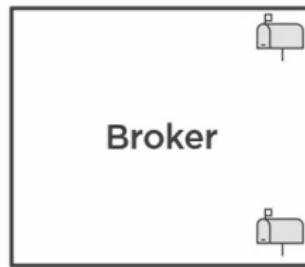
Consumers



Topics = groupes de messages

Courtiers ou *brokers*

Producers



Consumers

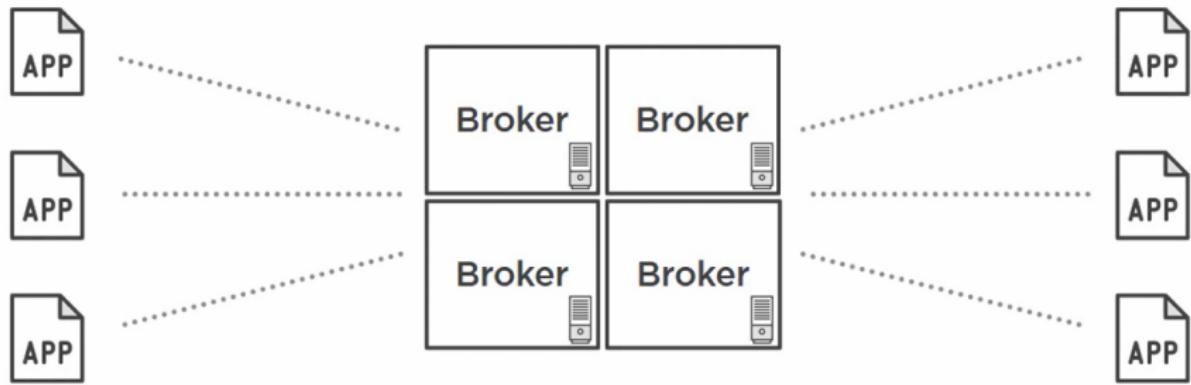


- Broker : gère un ensemble de topics
- C'est un processus (deamon) serveur

On peut créer un nombre quelconque de *brokers*

Producers

Consumers



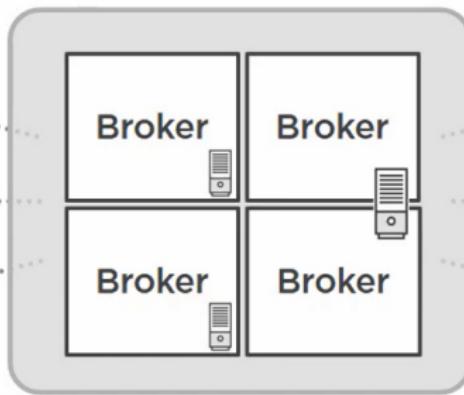
- C'est ce qui permet de gérer les gros débits de messages
- En 10.2019, LinkedIn a déclaré utiliser 4000 brokers avec 100 000 topics pour 7k milliards de messages par jour :
<https://www.linkedin.com/blog/engineering/open-source/apache-kafka-trillion-messages>

Cluster

Producers



Cluster
Size: 4

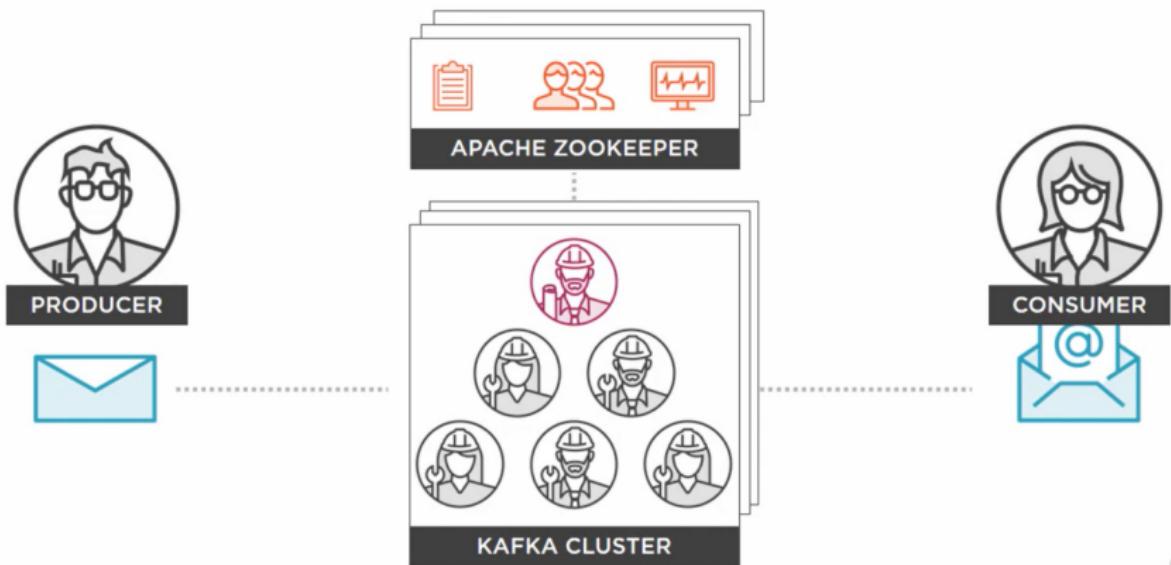


Consumers



- Cluster : ensemble de brokers

Apache Zookeeper



- Un système distribué (serveur) pour gérer les méta-données (état de santé, config, ...) sur les clusters
- C'est lui qui choisit quel broker est responsable de quel topic
- Utilisé dans d'autres projets aussi (Hadoop, Redis, Neo4j, ...)

Une intro différente à Kafka

- <https://www.youtube.com/watch?v=FKgi3n-FyNU>

Installer Kafka à la main

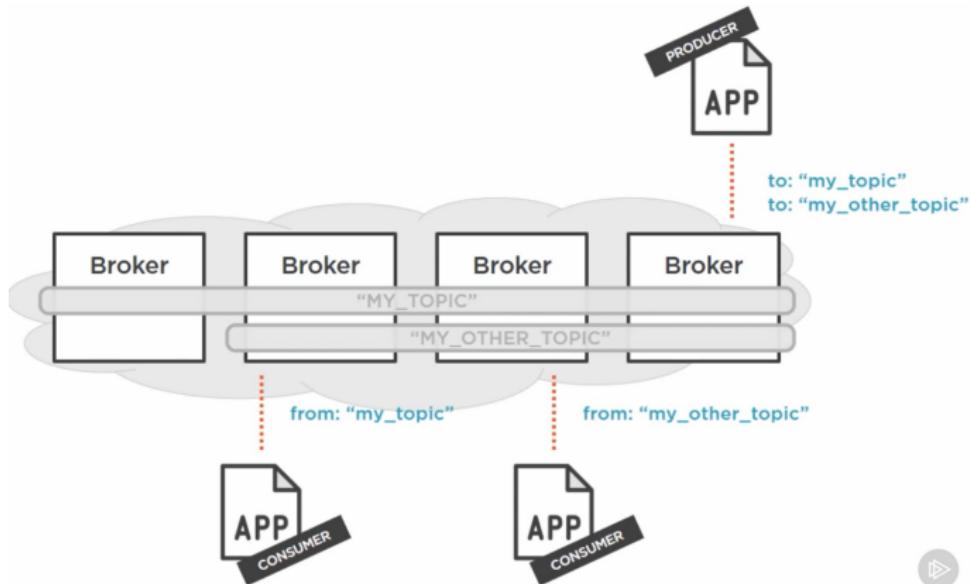
- Ouvrir un terminal
- Installer Scala si vous ne l'avez pas (une partie de Kafka a été écrite avec Scala) : `sudo apt install scala` (sous Linux)
- Télécharger l'archive qui se trouve ici :
<https://kafka.apache.org/downloads>
- Décompresser l'archive : `tar xvzf kafka_XXX.tgz`
- Renommer le dossier de l'archive décompressée (`kafka_XXX`) par `kafka`
- Explorer le dossier, qui a une structure classique : `bin/` (scripts Shell Unix + un dossier `windows/` avec les scripts Batch), `config/`, `libs/`, ...

Topics



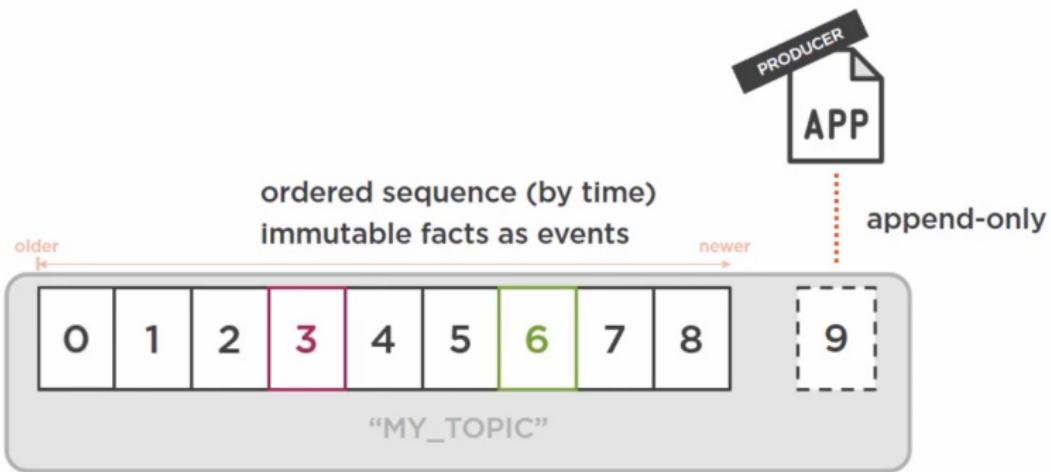
- Une abstraction (entité logique) au cœur du système
- Une catégorie ou un flux nommé de messages
- Les producteurs produisent des messages dans un topic
- Les consommateurs consomment des messages depuis un topic
- Physiquement représentés comme fichiers logs (Kafka = *distributed commit log*)

Topic : entité vue par les prod. & conso.



- Topics sur plusieurs brokers pour la fiabilité/scalabilité
- Mais les producteurs et consommateurs voient des topics (peu importe leur emplacement)

Topic : une séquence ordonnée

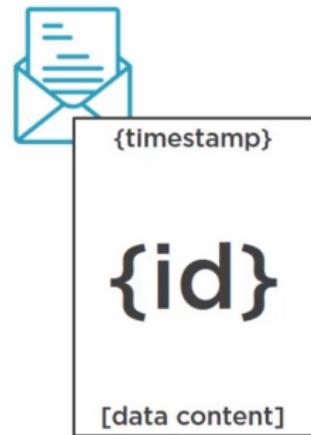


- Topic = séquence ordonnée par le temps
- Messages = événements (enregistrements ou *records*) immuables (système fonctionnant selon le style *Event Sourcing*)
- Messages insérés à la fin du topic et horodatés
- Au consommateur de repérer l'évolution d'un message (producteur incapable de changer un message existant)

Message

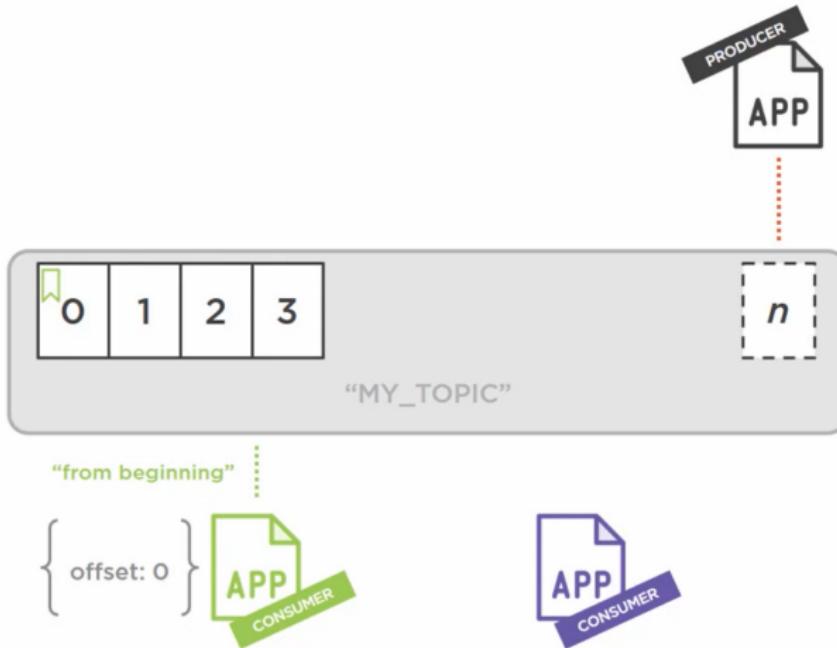
Chaque message :

1. est horodaté (a un *timestamp*)
2. a un identifiant unique utilisé par les producteurs et les consommateurs
La paire (*timestamp*,identifiant) est unique
3. a un contenu (*payload*) binaire



La défaillance d'un consommateur n'a aucun impact sur les messages, les autres consommateurs ou les producteurs

Offset



- Une sorte de marque-page qui correspond à la position du dernier message consommé
- Chaque consommateur peut évoluer indépendamment des autres dans la consommation des messages

Politique de rétention des messages

- Kafka garde tous les messages produits, qu'ils soient consommés ou pas
- La durée de rétention des messages est configurable en heures
- Par défaut, la durée de rétention est de 168 heures (7 jours)
- La période de rétention est définie par topic
- Le stockage physique peut contraindre la durée de rétention

A vos claviers

1. Démarrer un serveur Zookeeper :

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

* Un processus serveur est démarré sur le port 2181

2. Démarrer un broker :

```
bin/kafka-server-start.sh config/server.properties
```

* Un processus serveur est démarré sur le port 9092

3. Créer un topic :

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

4. Lister les topics :

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

A vos claviers -suite-

1. Produire un message :

```
bin/kafka-console-producer.sh --broker-list localhost:9092  
--topic my_topic
```

Ceci permet d'obtenir un Shell dans lequel on peut écrire plusieurs messages, séparés par des validations sur la touche "Entrée"

2. Consommer un message :

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

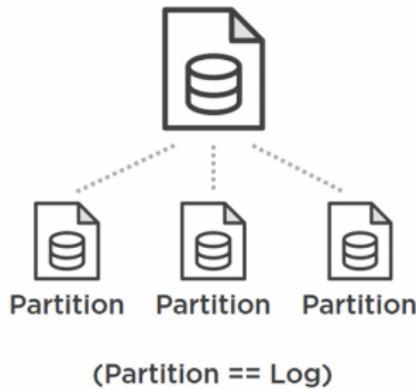
Les anciens messages s'affichent et chaque nouveau message produit par un producteur va apparaître

Pour supprimer un topic :

```
bin/kafka-topics.sh --delete --bootstrap-server localhost:9092  
--topic my_topic
```

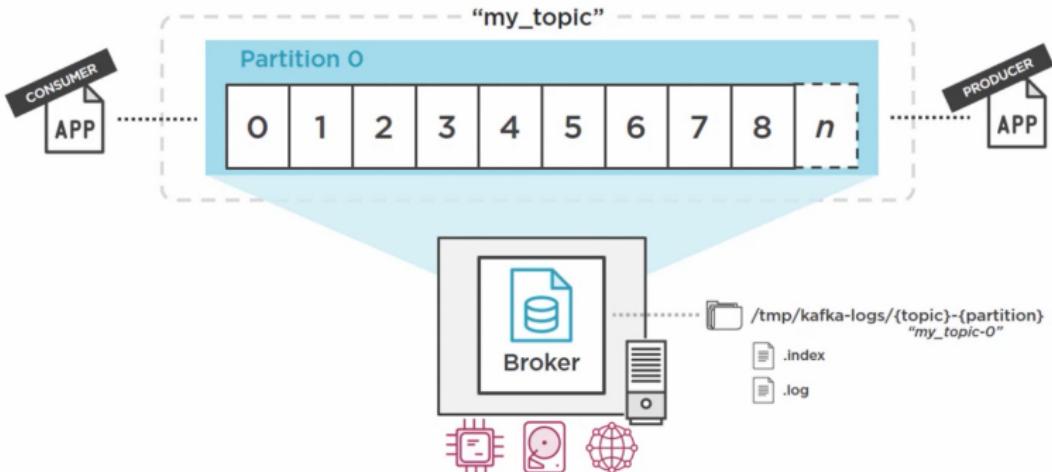
Partitions Kafka

- Une partition = un log physique (fichiers)
- Un topic a une ou +ieurs partitions
- Une partition est la base pour Kafka pour gérer la scalabilité, la tolérance aux fautes et la gestion d'un gros débit de messages
- Chaque partition est maintenue sur au moins un broker
- Dans l'exemple précédent, nous avons créé un topic avec une seule partition (- -partitions 1)



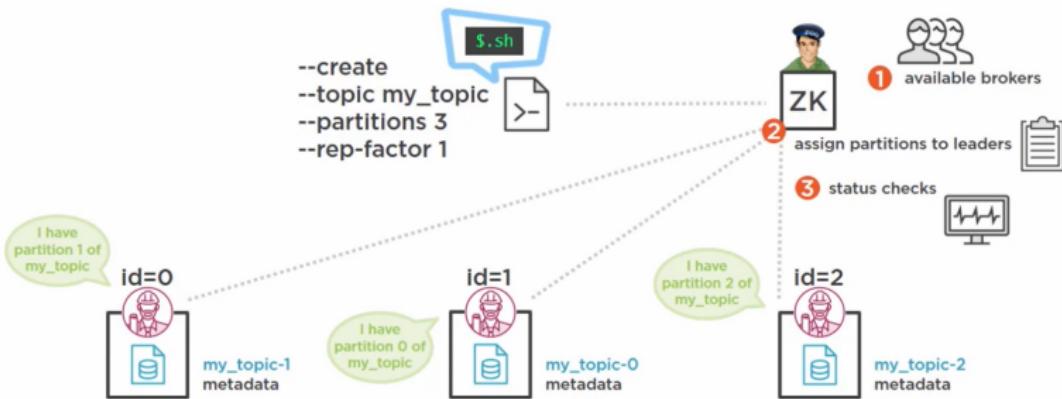
(Partition == Log)

Une partition sur une machine



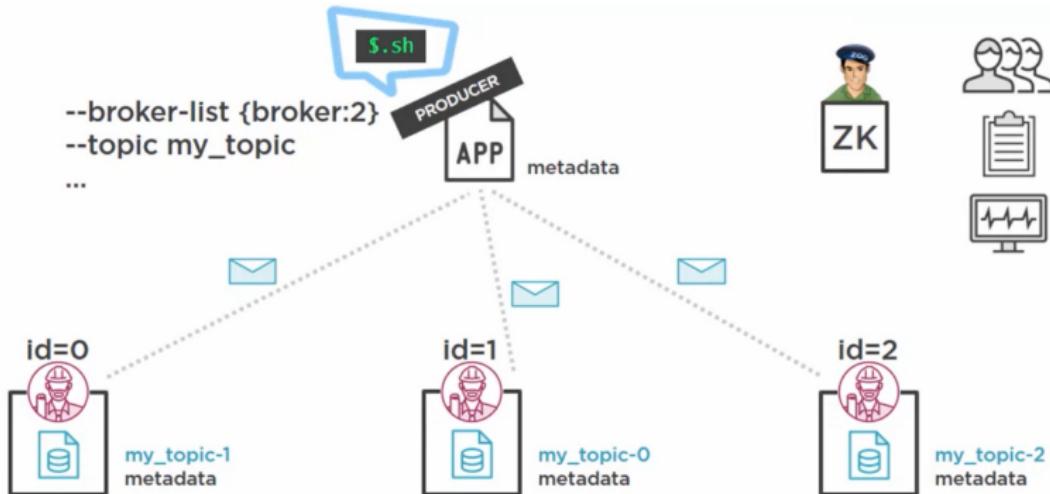
- Vérifier le contenu du dossier `/tmp/kafka-logs`
- Il y a un dossier pour votre topic, `my_topic-0`, avec des fichiers binaires `.log` (qui contient les messages), `.index`, ...
- Chaque partition doit tenir dans une machine

Plusieurs partitions pour la scalabilité



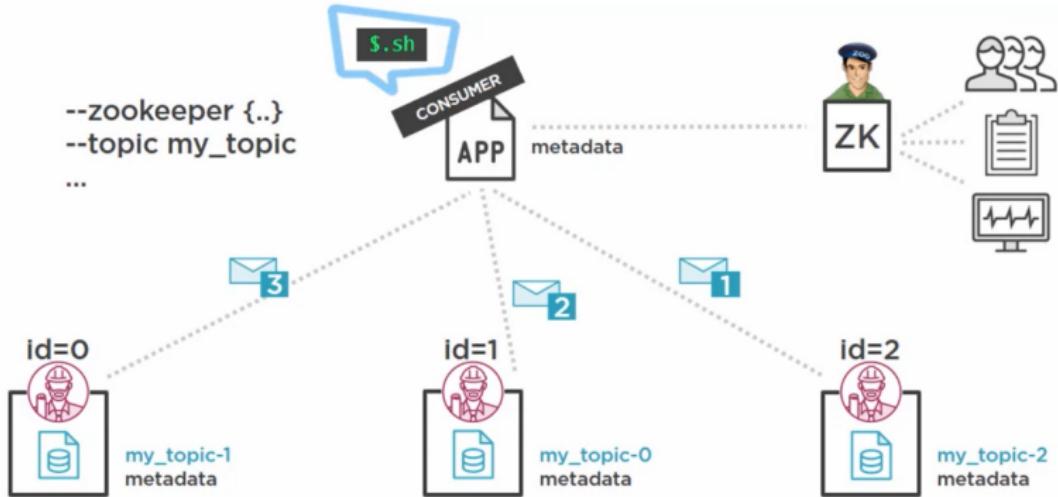
- Le serveur Zookeeper (ZK) gère l'affectation des partitions aux brokers
- Dans l'exemple, il va choisir 3 brokers, un pour chaque partition
- Les brokers remontent les informations sur leur état à ZK

Producteur de messages vers les brokers



- Un producteur de messages peut connaître 1 seul broker (broker 2 ci-dessus)
- Ce broker indique au producteur quels autres brokers détiennent les autres partitions (transparent pour le dev)
- La production du message envoie le message à tous les brokers

Consommateur de messages depuis les brokers

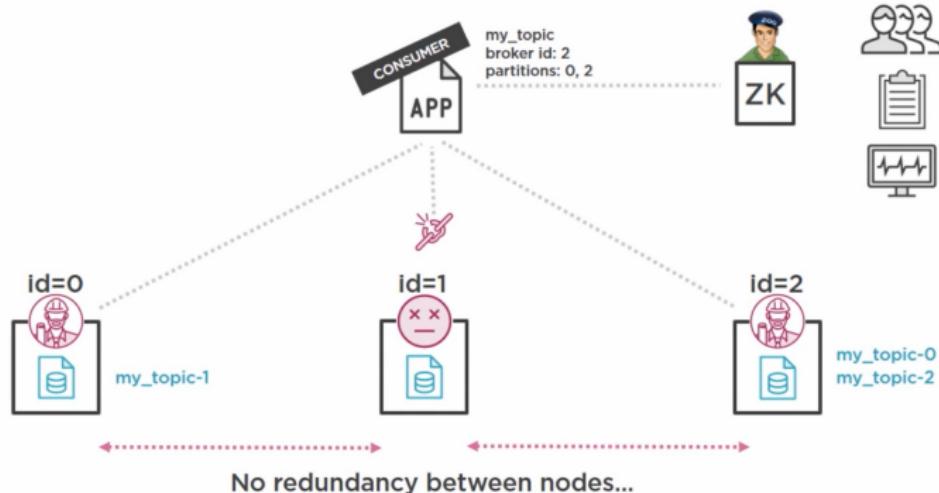


- Un consommateur collecte des méta-données depuis Zookeeper (liste et état des brokers)
- Il sollicite ensuite les différents brokers pour récupérer les messages
- Ces messages peuvent être ordonnés différemment sur chaque broker (le consommateur est responsable de gérer cela)

Compromis sur le nombre de partitions

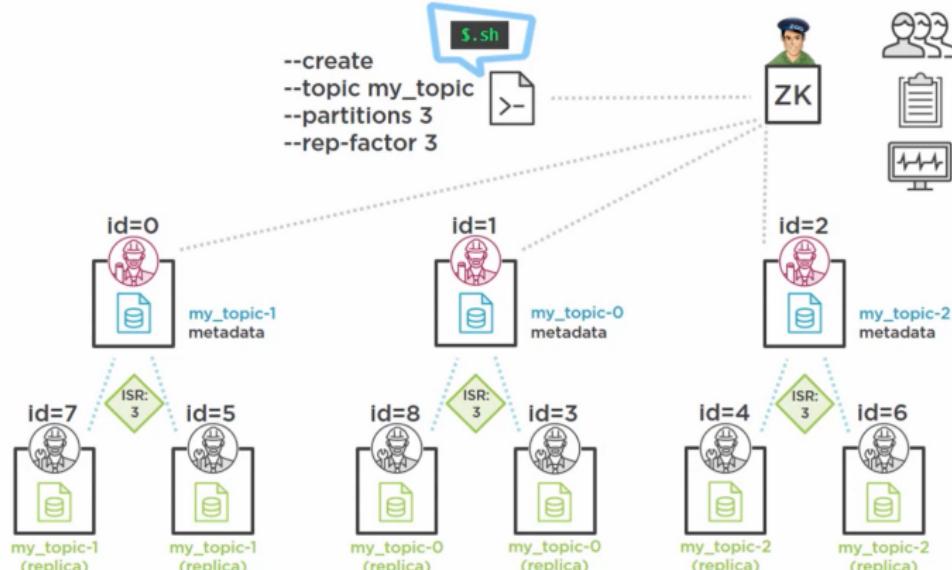
- Un nombre important de partitions a un impact négatif (surcharge) sur le Zookeeper (dans la coordination des brokers)
- Si l'on veut avoir un ordre total sur les messages, utiliser une seule partition par topic (sinon, ordres partiels / partition), ce qui peut être complexe à gérer dans certains cas

Tolérance aux fautes (ou aux pannes)



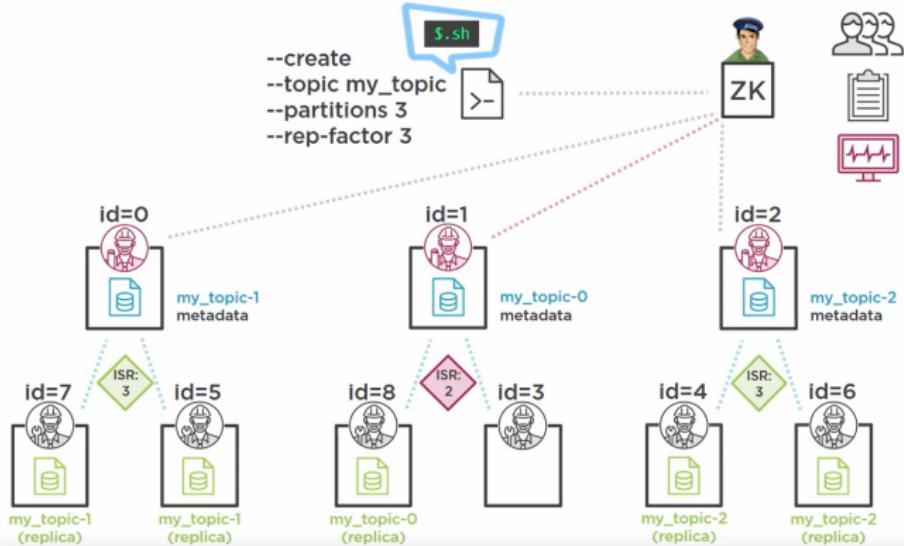
- Plusieurs fautes peuvent se produire (broker HS, coupure réseau, crash disque, ...)
- Le zookeeper gère l'inaccessibilité d'un broker pour informer les producteurs/consommateurs de messages (voir ci-haut)
- Mais les messages du broker en panne, qui n'ont pas été consommés, peuvent être perdus

RéPLICATION pour tolérer les fautes/pannes



- La tolérance aux fautes est atteinte par la réPLICATION
- Le paramètre **replication-factor** permet d'indiquer le nombre de répliq**ats par topic**
- Chaque broker (nommé leader) négocie avec d'autres brokers la gestion de la réPLICATION

RéPLICATION pour tolérer les fautes/pannes -suite-



- ISR veut dire *in-sync replicas*
- Tant que ISR est égale à replication-factor, le cluster est considéré comme en bonne santé (*healthy cluster*)
- ISR < rep-factor : Kafka continue de fonctionner mais ne fait pas de compensation

A vos claviers

Tester la création d'un cluster de 3 brokers :

- 3 exécutions de kafka-server-start avec 3 fichiers server.properties distincts (à copier, renommer et éditer)
server-0.properties, server-1.properties, ... avec :
 1. des id de brokers différents : 0, 1 et 2
 2. des dossiers différents pour les logs : /tmp/kafka-logs-0, /tmp/kafka-logs-1 ...
 3. et des ports différents : <votre-hostname>:9092, <votre-hostname>:9093, ...
- Créer un topic avec un facteur de réPLICATION = 3 :
`bin/kafka-topics.sh --create --topic replicated_topic
--bootstrap-server <votre-hostname>:9092
--replication-factor 3 --partitions 1`

A vos claviers -suite-

- Afficher les méta-données : (option --describe)

```
bin/kafka-topics.sh --describe --bootstrap-server  
<votre-hostname>:9092 --topic replicated_topic
```

- Ceci vous permet d'avoir :

```
Topic: replicated_topic
PartitionCount: 1 ReplicationFactor: 3
Configs:
Topic: replicated_topic Partition: 0 Leader: 1
Replicas: 1,0,2 Isr: 1,0,2
```

- Le leader de la réPLICATION est le broker dont l'id est 1 (brokers 0 et 2 participant)
- La ligne Topic aurait été affichée plusieurs fois si on avait plusieurs partitions

On peut altérer les méta-données d'un topic existant avec l'option --alter (pour ajouter par ex la réPLICATION à un cluster existant)

A vos claviers -suite-

- Créer un producteur de messages et produire quelques messages
- Créer un consommateur de messages
- Simuler une panne de l'un des brokers (le leader dont l'ID est 1), en tuant le processus de démarrage du broker (sur le terminal de démarrage de ce broker, faire un Ctrl-C)
- Ré-exécuter la commande `--describe` vue précédemment
- Que remarquez-vous dans le résultat affiché ?
- Le consommateur affiche une exception disant que le leader s'est arrêté, mais il continue de recevoir les messages
- Produire un message pour le vérifier
- Redémarrer le broker arrêté. Il va rejoindre le cluster. Pour le vérifier exécuter la commande `--describe`

Références bibliographiques

- Site web Apache Kafka :
<https://kafka.apache.org/>
- Producteurs et consommateurs de messages comme classes annotées Spring (à tester si vous connaissez Spring) :
<https://docs.spring.io/spring-kafka/reference/>
- Tutos sur Pluralsight et Baeldung

