

BreizhCamp

06/2015

Retour d'expérience sur Kafka





1. Contexte
2. Concepts
3. Architecture
4. Métriques
5. Retour d'expérience



Contexte



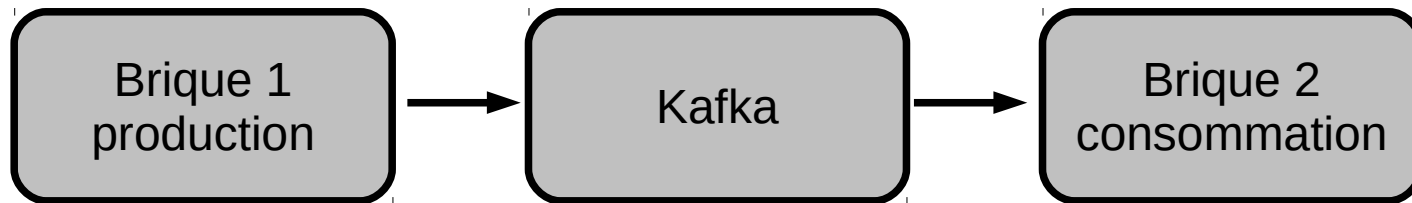
Projet en cours
& en production



Pourquoi un broker de
messages ?



Découplage entre briques logicielles





Robustesse





Persistance sur disque
Pas de perte de message sur
panne ou arrêt



Absorber des vitesses de traitements différentes





Performance : Throughput & latence



Absorber des pics de charge



Consommer très vite



Volumétrie * 100 d'ici 2 ans

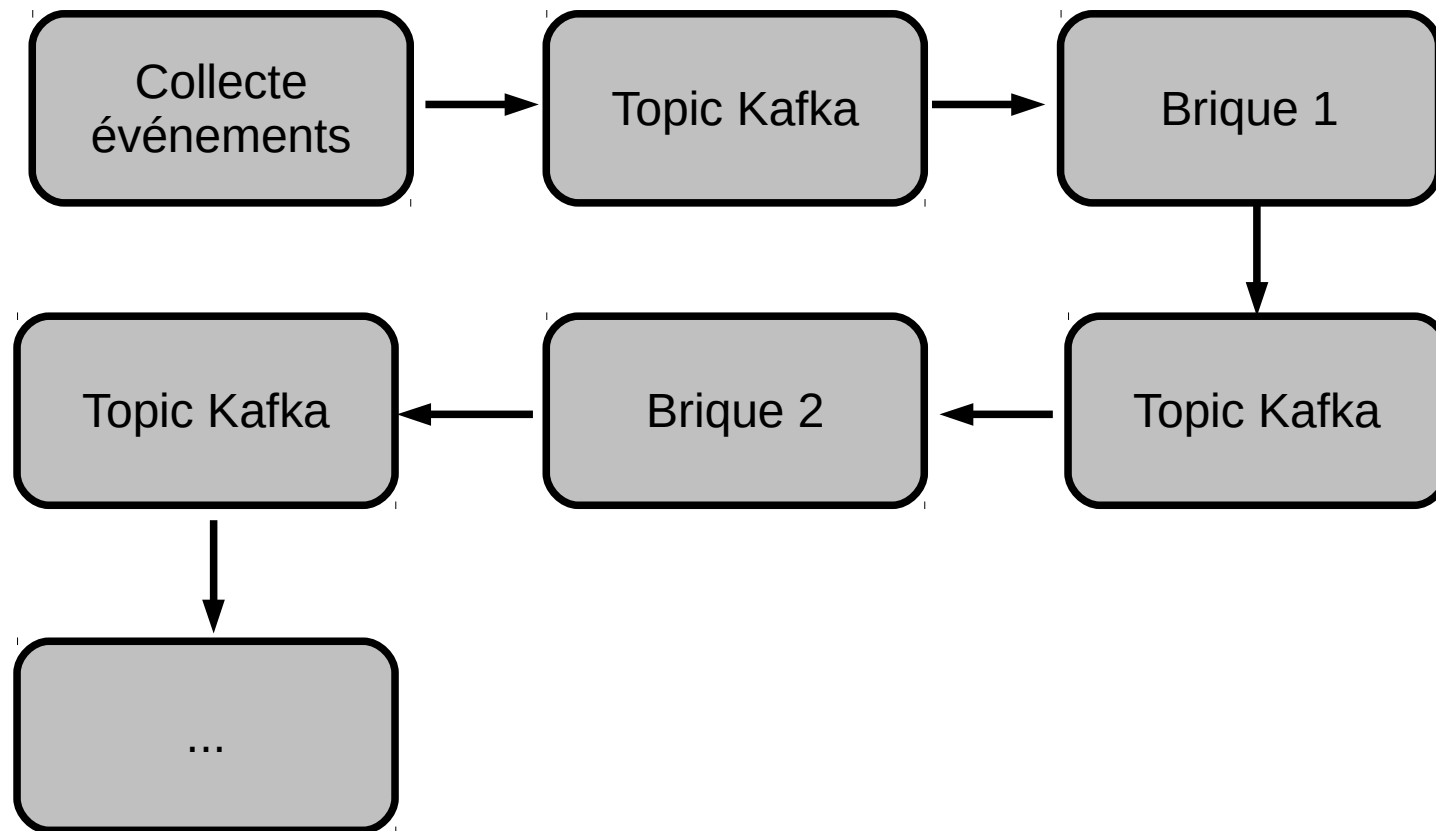


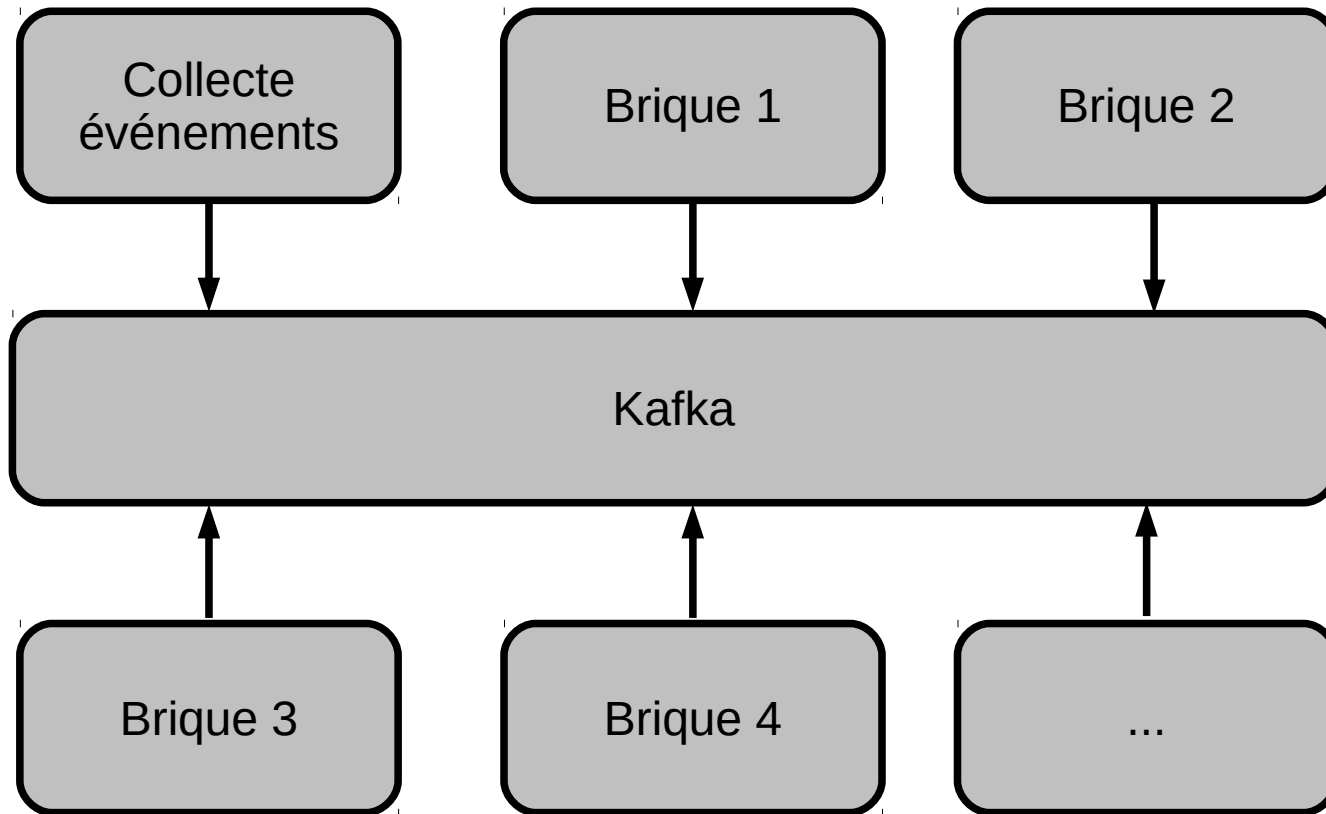
Scalabilité horizontale





Architecture en pipeline







Pas un besoin commun ?

De plus en plus :
grosses données
collecte de logs et de métriques
event sourcing
stream processing
IoT



Concepts





LinkedIn engineering



Opensourcé en 2011
chez Apache





Nom « académique » : Commit
log distributed over several
systems in a cluster



Broker de messages



Topics



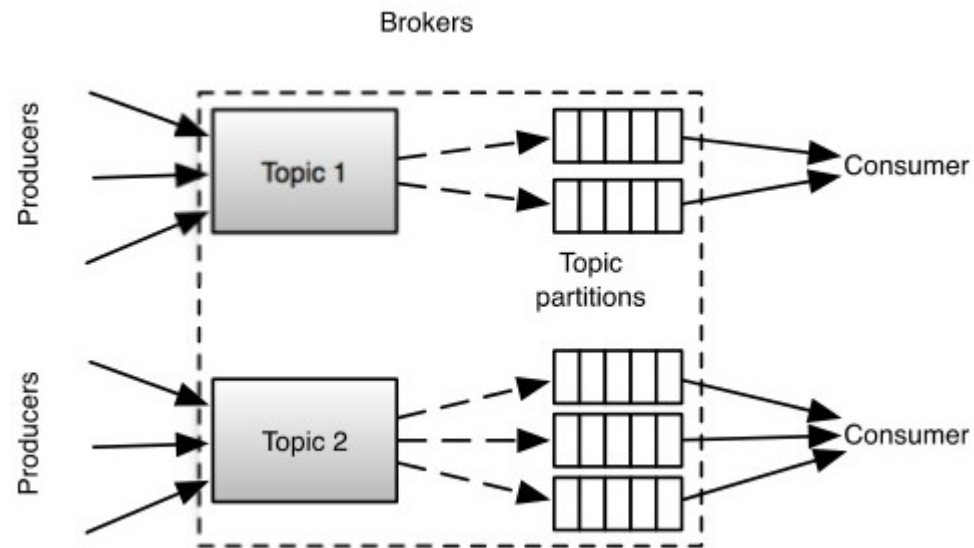
Topic producer



Topic consumer



Un topic est partitionné





Message = clé + valeur
Clé optionnelle



Un message consommé
n'est pas effacé



Conservation des messages dans
des fichiers de « logs » par topic &
partition



Purge par expiration
et/ou taille maximale
et/ou par compaction basée sur
la clé



message == byte[]
serializer encode et
deserializer décode



Format des messages libre :

string

json

byte[]

avro

...



Le broker ne « connaît »
pas les consumers



Ce n'est pas le broker qui push
Le consumer pull



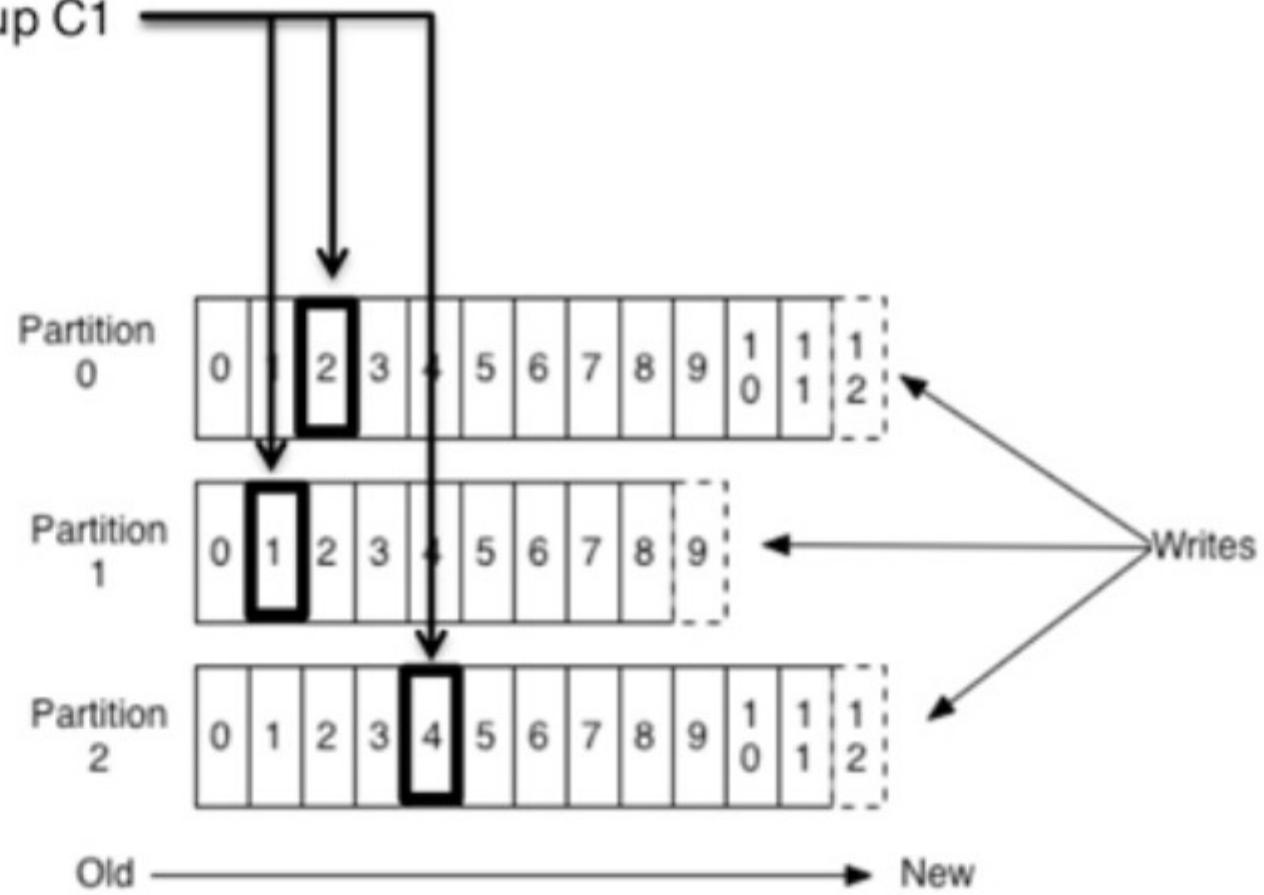
Pas
d'acknowledge/commit/rollback
du consumer



Un consumer maintient
ses offsets de
consommation



Consumer group C1





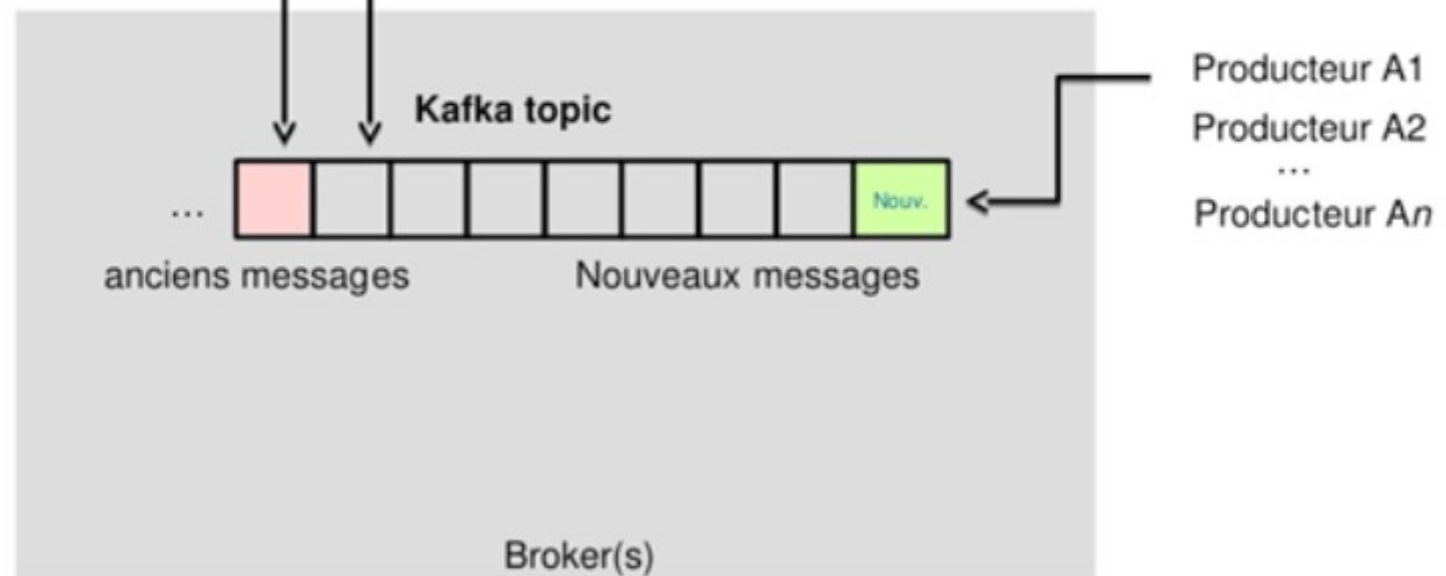
Offset :
séquence
index ordonné



Consommateur groupe C1

Consommateur groupe C2

*Les consommateurs emploient un « pointeur de lecture »
("offset pointer") pour tracer et contrôler là où il en est
(et décide du rythme de consommation)*





Il suffit donc de reculer les
offsets pour faire du replay

Relire un message en
particulier via son offset



Consumer group identiques
=> queue

Consumer group différents
=> topic (publish subscribe)



Delivery : at least one
(messages jamais perdu
mais peuvent être
redélivrés)



Chaque message a une clé
Peut être null



Le producer route les
messages vers les
partitions



Si la clé est null, routage
aléatoire
Sinon $\text{hash}(\text{key}) \% \text{nb partitions}$
Partitioner custom possible



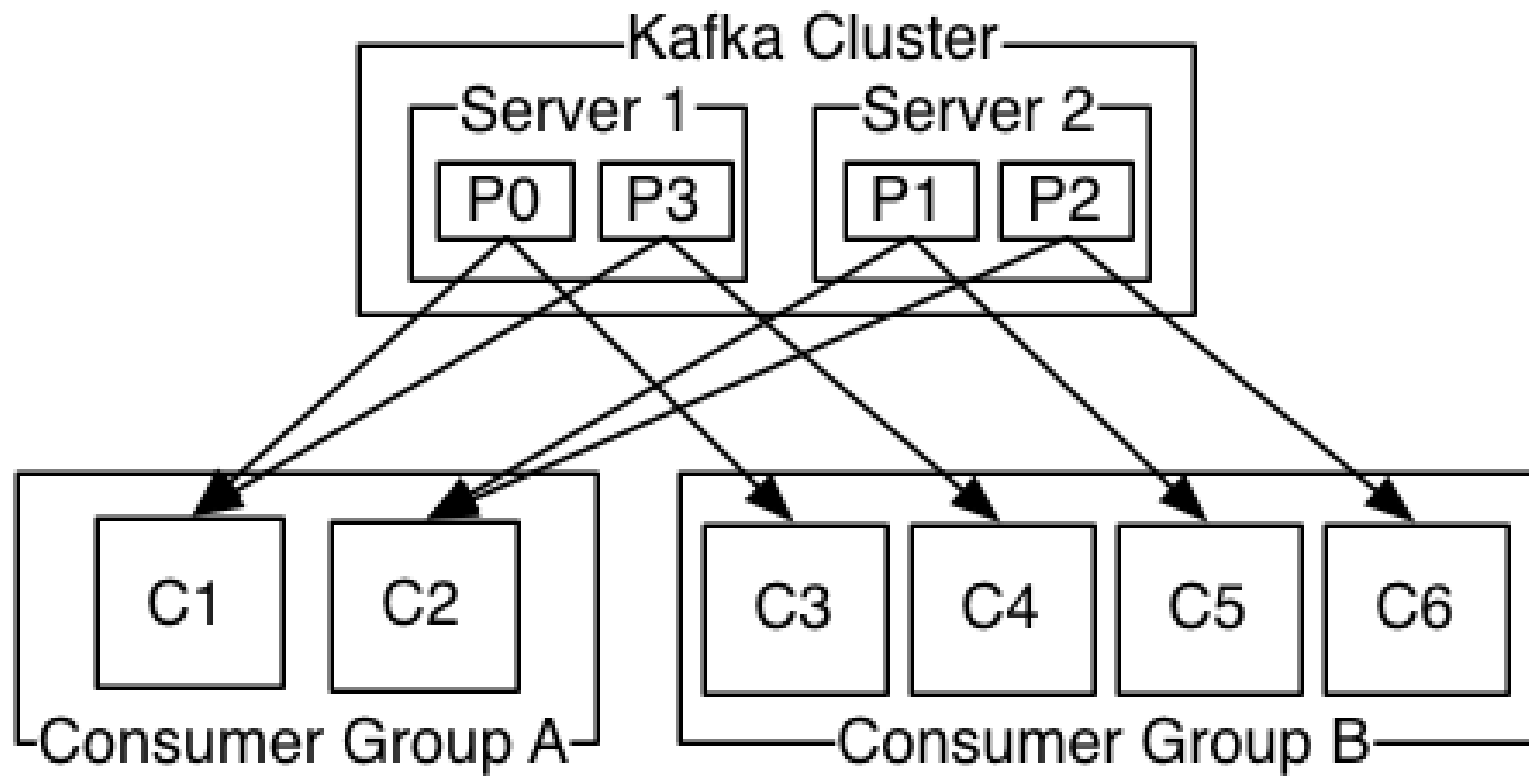
Conservation de l'ordre par partition



Possibilité de consommer en
parallèle par partition pour les
performances
tout en gardant une notion d'ordre
entre les messages

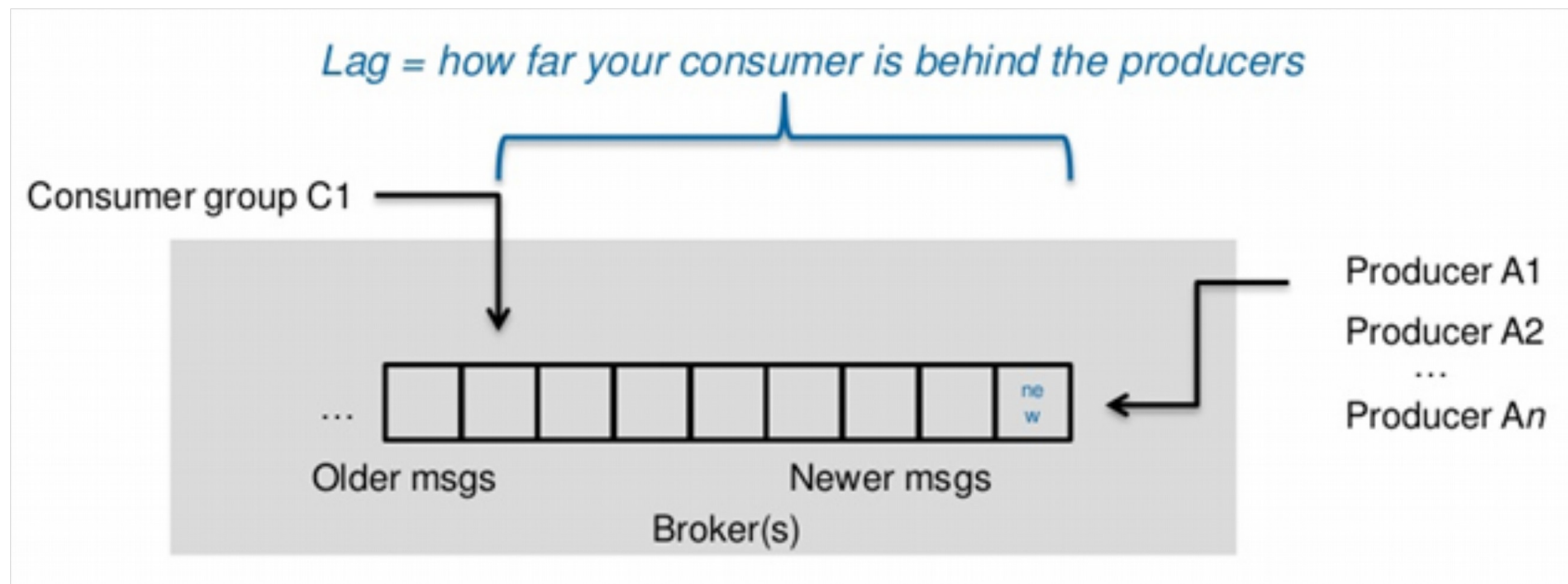


Mais pas inter partition





Consumer lag = retard





YAHOO!



criteo

sematext

NETFLIX

loggly



Hotels.com





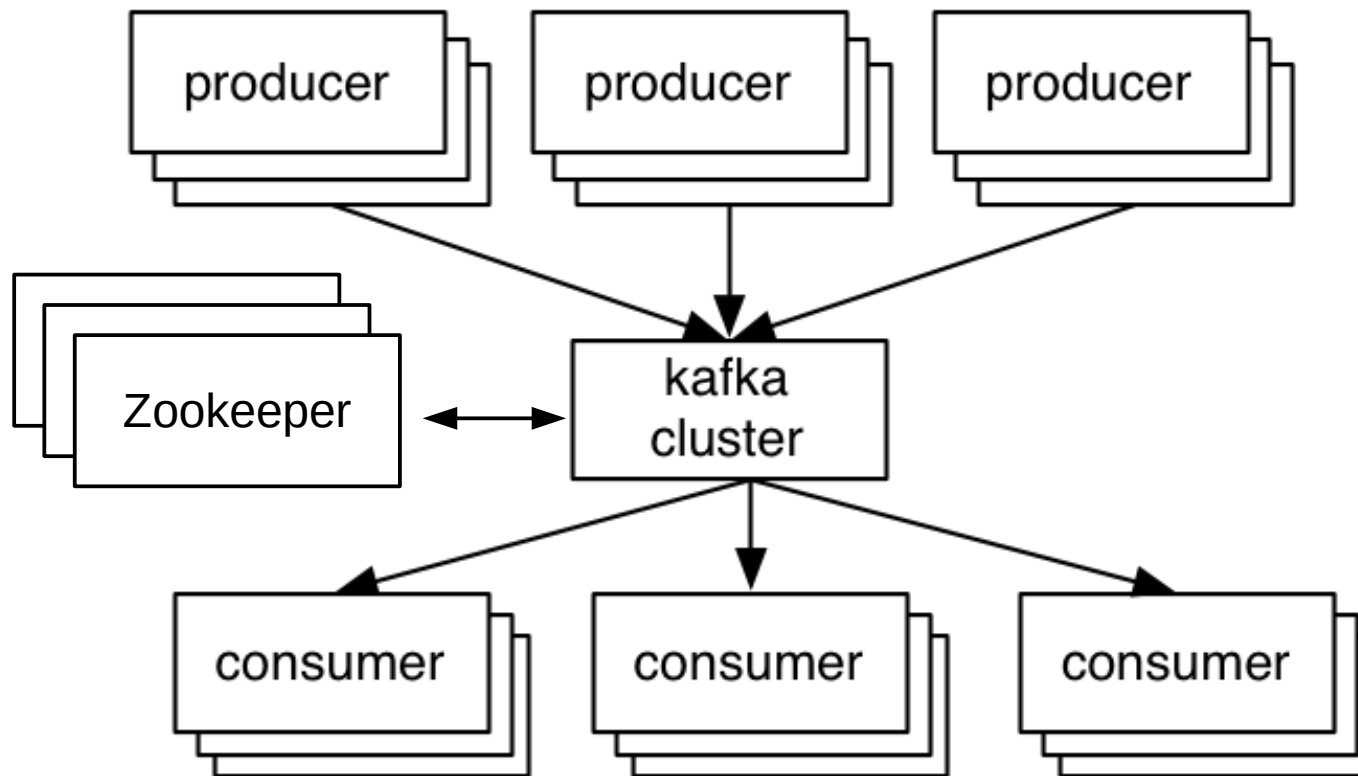
Architecture



Écrit en Scala
Tourne sur une simple JVM



Architecture globale



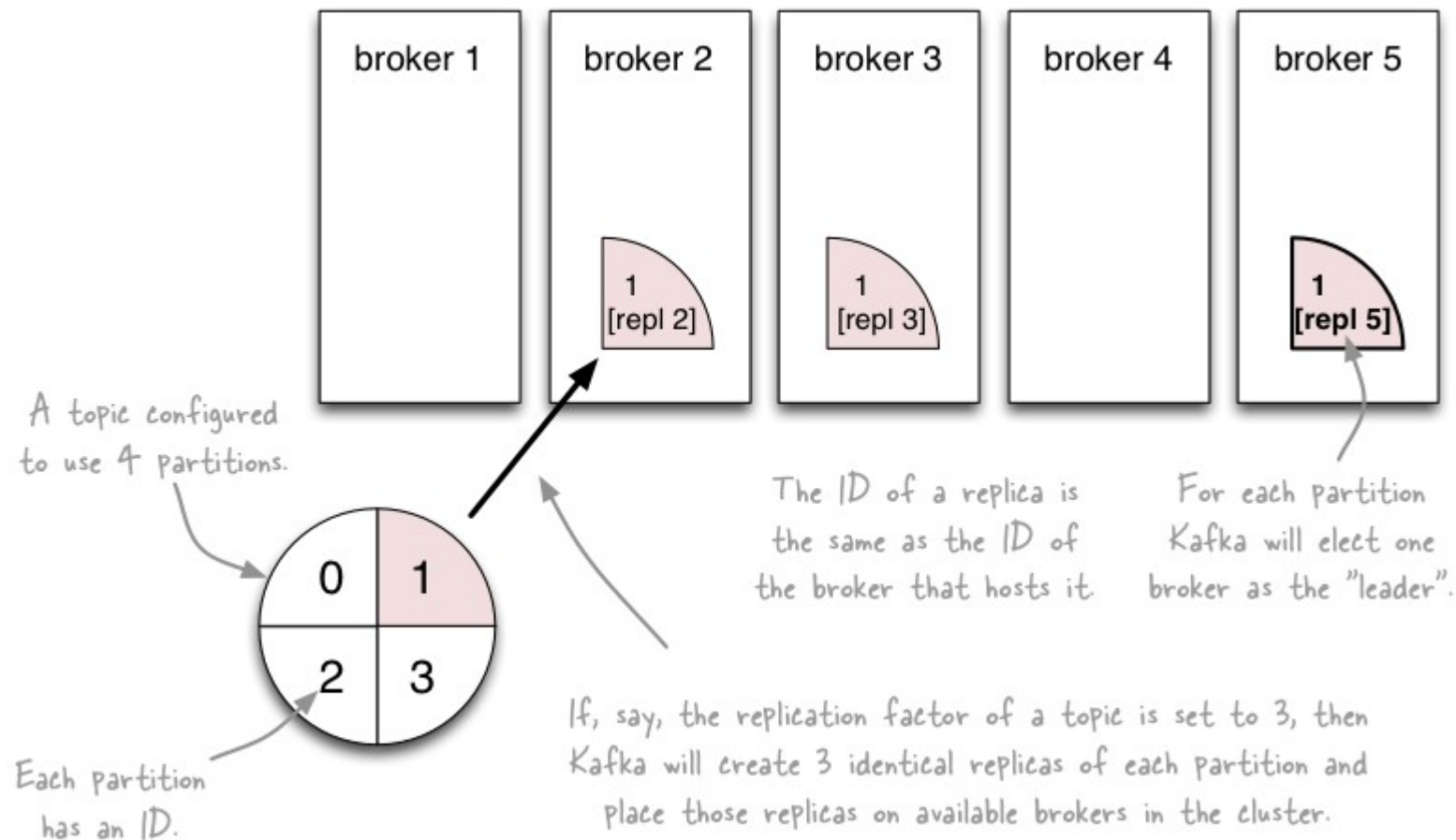


Topics partitionnés
Les partitions sont répliquées



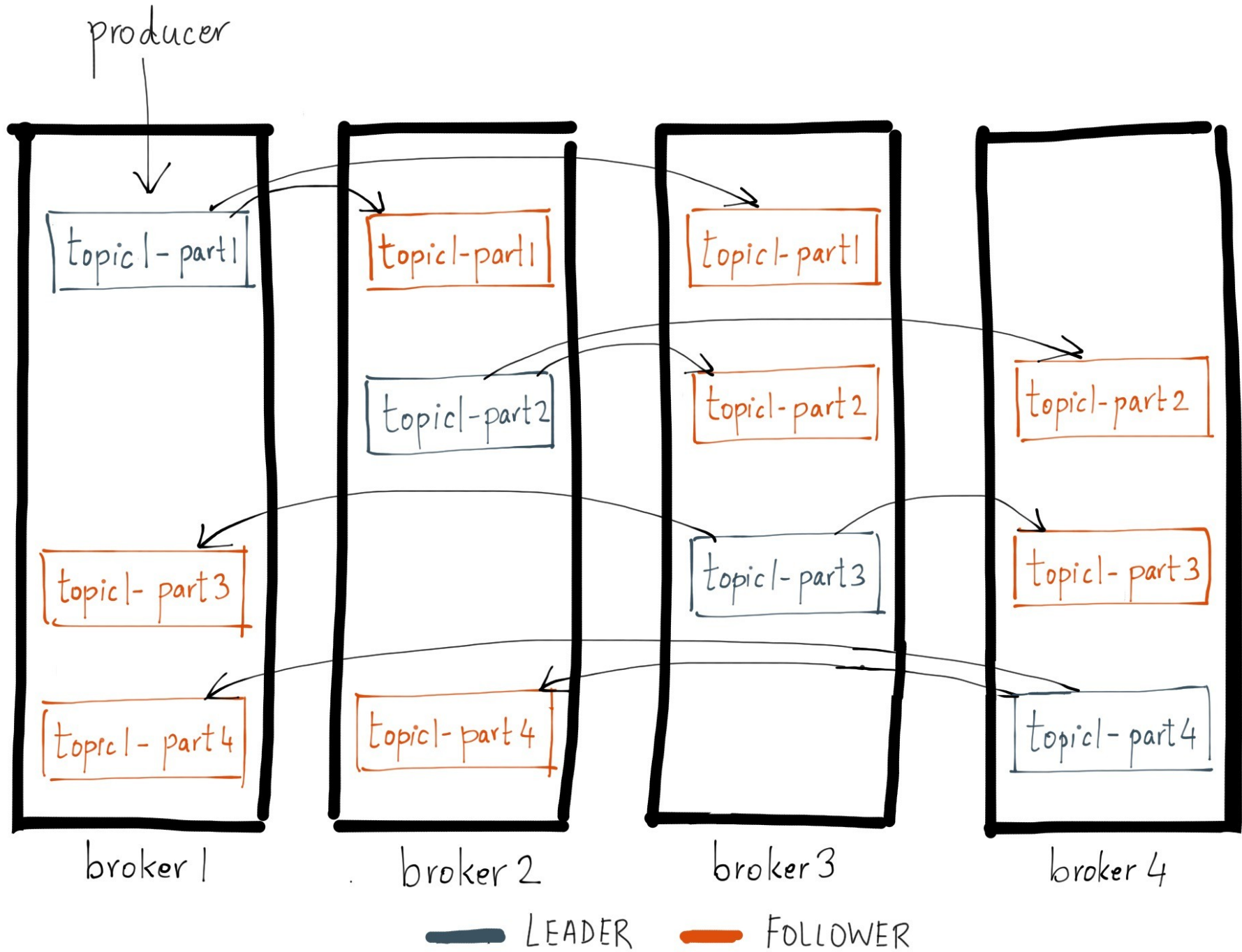
Un leader et N répliquas

Répartis sur les brokers du cluster





Un des nœuds est élu controler
Responsable de la répartition
des leaders





Partition = unité de parallélisme
+ partitions

=>

+ throughput en production & consommation

=>

scalabilité



Mais aussi

=>

- + ressources système
- + de temps pour bascule leaders en failover
- + de latence si ack asynchrone
- + de mémoire côté client



Réplication synchrone
tout
leader
n

Réplication asynchrone



Replica :

Jamais lu ni écrit par un
consumer ou un producer

Uniquement pour la tolérance
aux pannes



Un topic avec une réplication
de N tolère la perte de N-1 serveurs
(min.isr = 1)



Le consumer commite ses
offsets dans un topic Kafka
dédié (Zookeeper avant)



Topic Kafka avec compaction

A la compaction, on ne garde que
la dernière version par clé
(Event sourcing, data change
capture)



Compression par le producer
Décompression par le consumer
transparente



Nativement et toujours persistant
sur filesystem

byte[] stocké directement sur fs
sans transformation



Écrit mais pas flushé
fsync sur intervalle max et/ou
nombre de messages max



Mais alors pas ultra safe

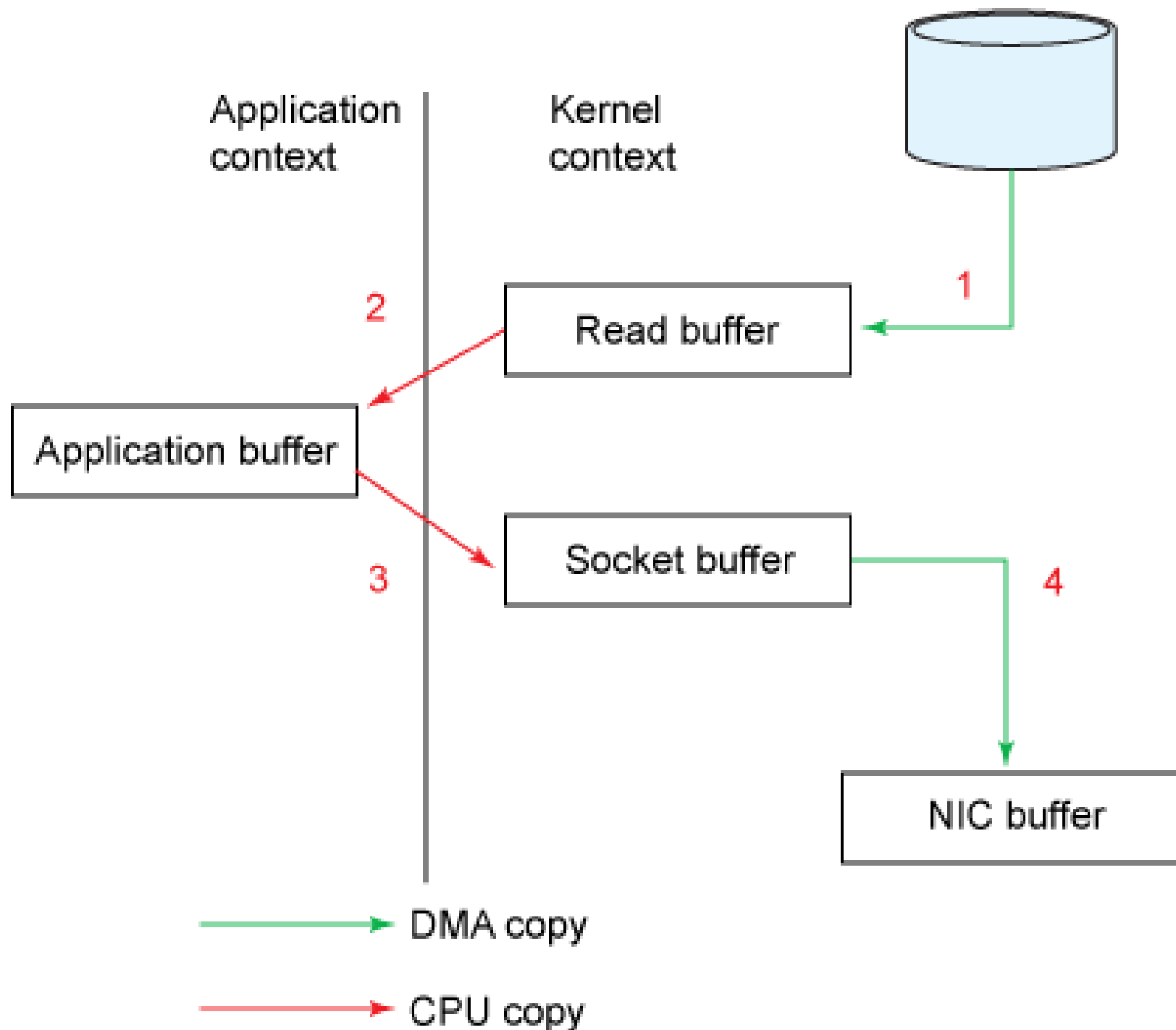
...

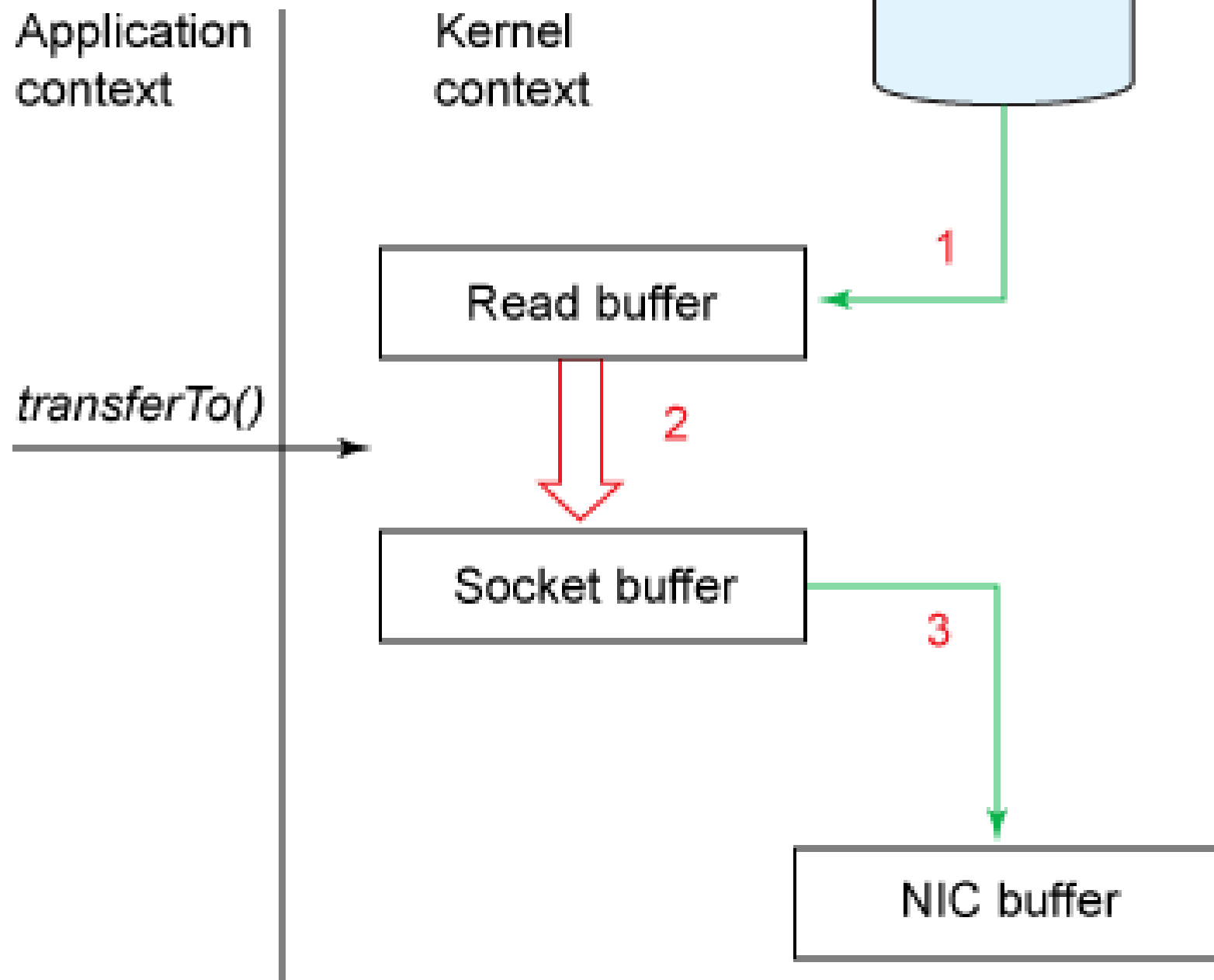
En fait si car réplica, acks ...



write : écritures disques séquentielles
read : OS cache, API sendfile (zero copy)

Avec une consommation == production,
quasiment pas d'activité en lecture disque,
tout depuis le cache OS







Client Java natif

Clients scala, node.js, .NET,
clojure, go, python, ruby, php,
erlang, c/c++



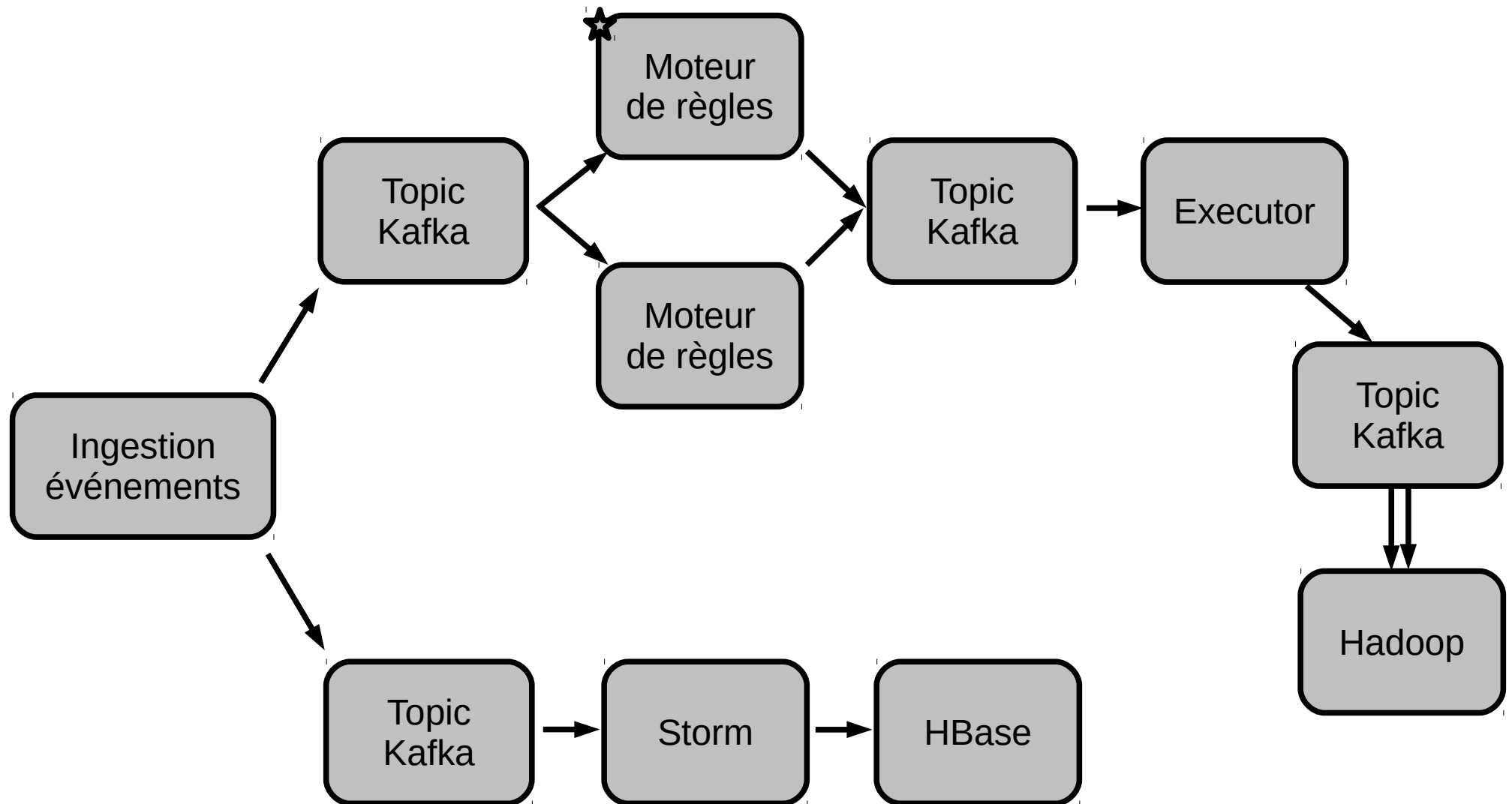
Proxy REST

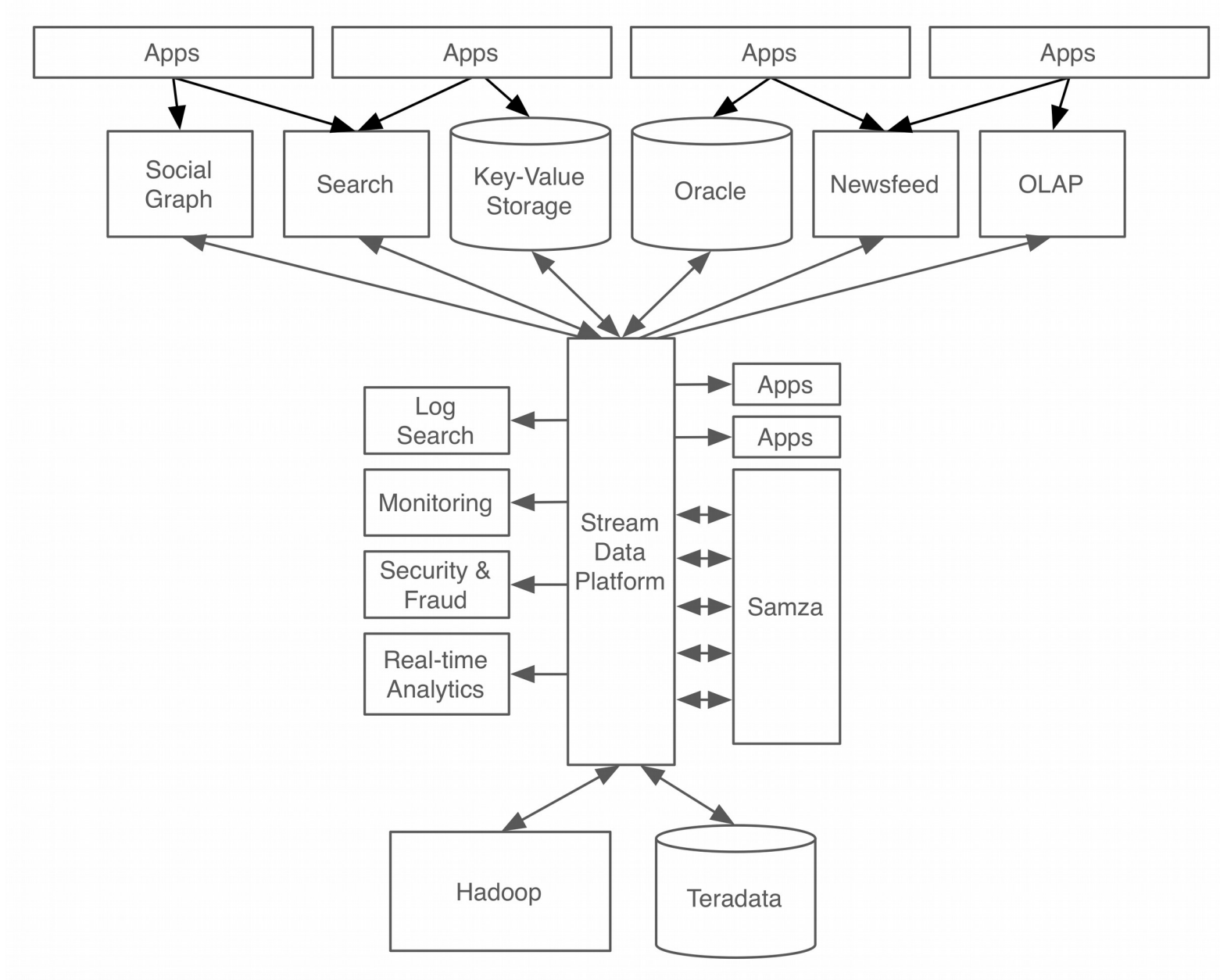
Interface simple
Performance moindre

Pour les autres langages ou si le
client n'est pas à jour



Architecture





@LinkedIn : A modern stream-centric data architecture built around Apache Kafka



Les deux approches sont
combinables : temps réel et batch
en relisant tout ou partie



Dans les distributions Hadoop
Cloudera et Hortonworks





Flume avec Flafka

Source & sink

Implémentation de channel



Kafka -> Hadoop
via Camus et Sweeper
(compaction des petits fichiers
Avro sur HDFS)
ou Kangaroo





Agrégation de logs
Appender logback, log4j2, ...
Logstash à partir 1.5



Stream processing :
Storm
Flink
Spark streaming
Samza



STORM



Spark

samza



Métriques



Sur un PC portable (sans SSD), 1 producer multithreadé, 1 consumer, pas de réplication :

20 000 messages/s en production synchrone

330 000 messages/s en production asynchrone

200 000 messages/s en consommation

< 2 ms en latence



Kafka@LinkedIn

300 brokers

18 000 topics & 140 000 partitions

220 milliards messages/j

40 Tbits/j in & 160 Tbits/j out



1 cluster @LinkedIn
15 brokers
15 000 partitions
réplication *2
400 000 messages/s
70 MB/s in & 400 MB/s out



Benchmark LinkedIn

3 serveurs

Intel Xeon 2.5 GHz 6 core

7200 RPM SATA drives

32GB of RAM

1Gb Ethernet

1 topic & 6 partitions



Producer

1 thread, no réplication
=> 820 000/s (78 MB/s)

1 thread, 3* réplication asynchrone
=> 780 000/s (75 MB/s)

1 thread, 3* réplication synchrone
=> 420 000/s (40 MB/s)

3 producers, 3 serveurs, 3* réplication asynchrone
=> 2 000 000/s (190 MB/s)



Danger des brokers
OK tant que ça tient en mémoire

Performances KO quand les
messages ne sont pas consommés
assez vite et qu'ils doivent être
écrits/paginéés sur disque



Embêtant car c'est justement le but
d'un broker que d'absorber du lag

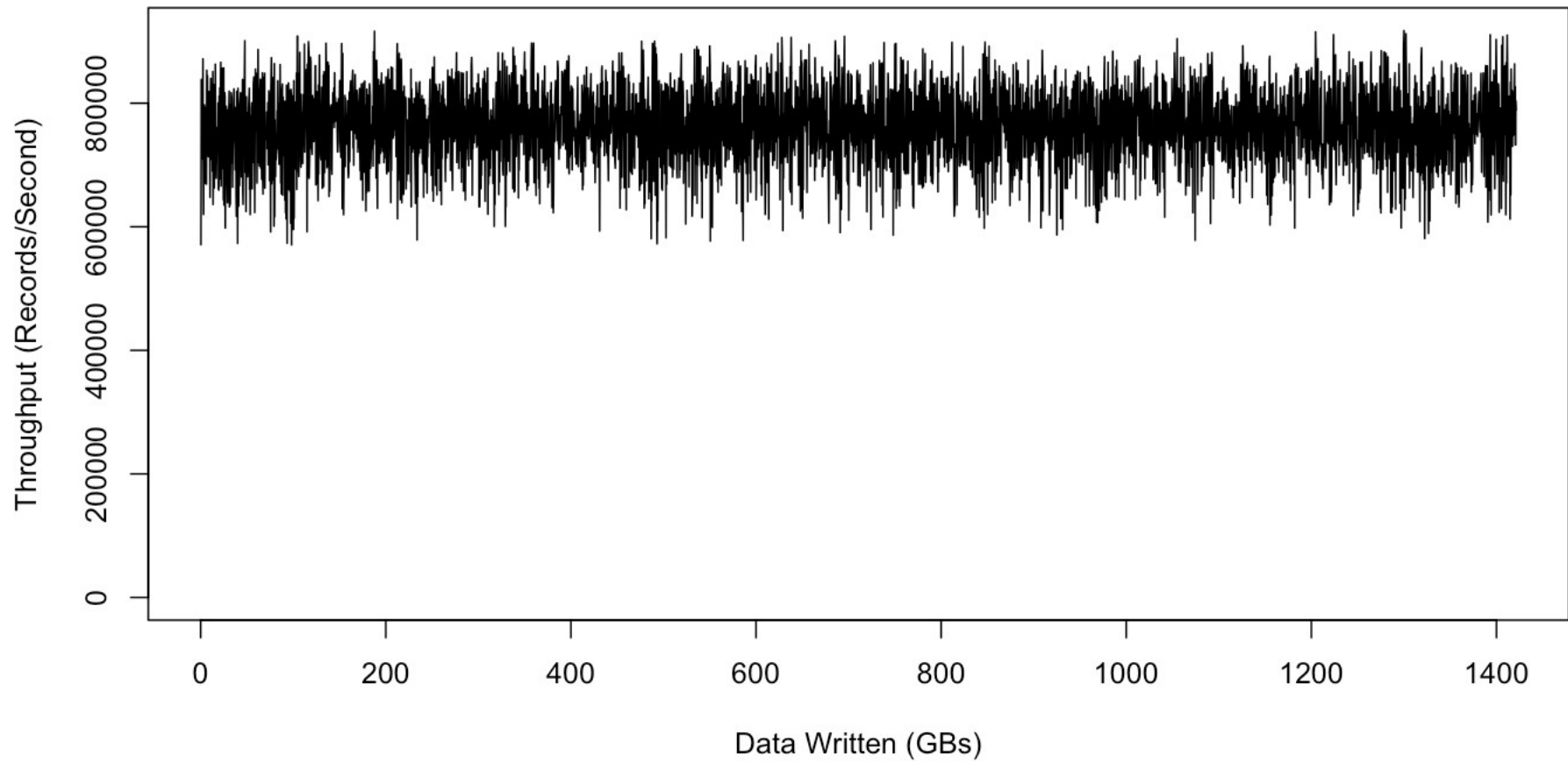


Couteau suisse mais flow control
puis blocked

Kafka



Throughput vs Size





Consumer

1 consumer (1 seul thread), début du topic (donc pas dans le cache de l'OS)
=> 940 000/s (90 MB/s)

3 consumers, 3 serveurs
=> 2 600 000/s (250 MB/s)



Latence

2 ms (median)

3 ms (99th percentile)

14 ms (99.9th percentile)



Producer & consumer

1 producer, 3* réplication asynchrone
1 consumer
=> 795 000/s (75 MB/s)

Quasi identique au cas du producer seul
La consommation est efficace et peu impactante



Retour d'expérience



Pas un remplaçant instantané de
JMS

Pas d'acquittement, de
request/reply, de sélecteur, de XA



Absorption de pics de
charge sereinement





Très performant





Utilisé depuis longtemps
chez LinkedIn sous forte
charge





Très stable





Bonne documentation
Communauté assez active





Confluent
Support entreprise possible





Une release par an
Release sur patch important
rapide





Montée en compétences
relativement rapide





Packaging tar.gz
RPM avec les distributions
Hadoop ou parcel
docker





CLI ligne de commande simple
création, modification,
suppression de topics





```
kafka-topics.sh --zookeeper localhost:2181 --create  
--topic topic-test --partitions 4 --replication-factor 1
```

```
kafka-topics.sh --zookeeper localhost:2181 --list
```

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



API de production
super simple





Contention sur le producer
en multithread en 0.8.1
=> Un producer par thread





Option : utiliser la production
asynchrone mais gestion des
retry nécessaire



Option : envoi par batch
(liste de messages en un appel
réseau)





Producer 0.8.2 asynchrone
par défaut
Synchrone via Future.get()
Callback possible





Exemple production synchrone 0.8.2

```
Properties config = new Properties();  
config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");  
config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());  
config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());  
  
Producer<String, String> producer = new KafkaProducer<String, String>(config);  
ProducerRecord<String, String> record = new ProducerRecord<>("topic-test", "une-valeur");  
producer.send(record).get();  
  
producer.close();
```





APIs de consommation :
High level consumer
Low level (simple) consumer





High level API plus compliqué
que l'API de production
(rewrite en cours pour la 0.9)





Le commit est automatique
sur délai

Commit « manuel » possible
pour plus de contrôle



Exemple consommation v0.8.1



```
public void consume() {
    Properties props = new Properties();
    props.put("zookeeper.connect", "localhost:2181");
    props.put("group.id", "un-consumer");
    ConsumerConfig consumerConfig = new ConsumerConfig(props);

    ConsumerConnector consumer = Consumer.createJavaConsumerConnector(consumerConfig);
    String topic = "topic-test";
    int threadCount = 4;

    Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
    topicCountMap.put(topic, threadCount);
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer.createMessageStreams(topicCountMap);
    List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);

    ExecutorService executor = Executors.newFixedThreadPool(threadCount);
    for (final KafkaStream stream : streams) {
        executor.submit(new ConsumerTest(stream));
    }
}

public class ConsumerTest implements Runnable {
    private KafkaStream m_stream;

    public ConsumerTest(KafkaStream a_stream) {
        m_stream = a_stream;
    }

    public void run() {
        ConsumerIterator<byte[], byte[]> it = m_stream.iterator();
        while (it.hasNext()) {
            System.out.println("Consuming " + new String(it.next().message()));
        }
    }
}
```





Simple API n'a rien de simple
mais offre plus de contrôle

Vraiment très très bas niveau



API consumer 0.9 en cours



```
Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:2181");
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
for (int i = 0; i < 8; i++) {
    consumer.subscribe(new TopicPartition("test-topic2", i));
}
int count = 0;

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.println("partition:" + record.partition() + ", offset:" + record.offset() + ", message:" + record.value());
        if (count++ >= 10) {
            consumer.commit(CommitType.SYNC);
            count = 0;
        }
    }
}
```





n threads de consommation == n partitions
+ de threads => threads consomment pas
(intéressant pour le HA)
- de threads => threads consomment
plusieurs partitions



8 partitions par topic
Recommandations de sur
partitionner



Retry



API changeante
Compatibilité entre 0.7 et 0.8 KO
OK désormais (Confluent &
Cloudera API Compatibility
Testing)





Commencer petit



3 nœuds en production
2 replicas
producer avec ack du
leader et du réplica



Curseur latence et durabilité
flush toutes les secondes & tous
les 10 000 messages



Maintenant :
de 500 000 à 10M messages/j
8000/s en pic
Très vite :
*100



Très vite 5 nœuds
3 replicas



Taille de fetch > taille du plus
gros message





Cluster Zookeeper nécessaire Élection de leader robuste





Zookeeper n'est pas fait
pour des fortes charges
en écriture





Commit des offsets à batcher dans Zookeeper





Manque d'unité
entre Kafka et Zookeeper





Avec le stockage des offsets dans
Kafka en 0.8.2, possibilité de
diminuer drastiquement l'intervalle
de commit





Principalement stream
Replay pour un scénario de HA
Consommation batch export Hadoop





Des messages avec une même clé
sont routés dans la même partition
Traitement dans un ordre précis





Si pas de clé métier et pas notion
d'ordre, clé == null
Routage aléatoire avec sticky entre 2
refresh
Corrigé en 0.8.2





Pas de timeout sur le close du
producer en 0.8.2
Corrigé sur master





Veiller à une bonne
répartition des messages
par partition



Répartition automatique
des leaders par broker
marche bien





Serializer custom
Utilisé pour plugger
élégamment un mécanisme
de sérialisation custom





Asséchement avant
déploiement
(lag des consumers == 0)

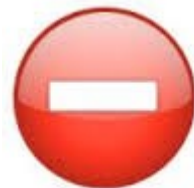


Ecrit en Scala ... lisible



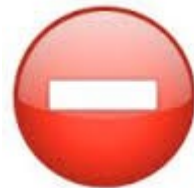


Dépendance sur Scala
peut entrer en conflit avec
d'autres briques en Scala dans
des versions différentes **non
compatibles**





Dépendances Maven à assembler





Pas évident sous
Windows en 0.8.1
OK en 0.8.2





Breakpoint en debug pas simple
car les timeouts Zookeeper sautent
(breakpoint type thread possible)





EmbeddedKafkaServer à écrire
pour les tests unitaires/intégration
Pas très compliqué

Projet GitHub en cours





Facile de simuler un cluster
localement en démarrant
plusieurs brokers localement





Robustesse
approuvée





Empreinte mémoire
et CPU légère
même en charge





En cas de perte d'un broker pour
une longue période, script de
réassignation de partitions à
passer

Travaux en cours 0.8.3
ou Mesos





Pas encore de rack awareness
Netflix y travaille





Settings par défaut
+ quelques paramètres
recommandés

Durée rétention, min isr,
réplication, ... importants



20j de rétention
le temps de pouvoir analyser
sereinement un bug si nécessaire





Tolérant aux partitions
réseaux dans certaines limites
en 0.8.1 (voir Jepsen)





Paramétrage 0.8.2 pour éviter
des « unclean » élections
&
min.isr





Script de démarrage de base
service /etc/init.d à écrire

Bientôt plus simple avec SystemD





Installation facilement
automatisable
Paramétrage simple





Logs corrects
Log beaucoup en cas de
failover
Certains restent cryptiques





Doublon possible :
rebalancing
retry du producer





Ceinture & bretelle via
idempotence
(utilisé pour d'autres objectifs
dans l'architecture)



Idempotent producer
en réflexion



Pas d'IHM
Pas indispensable
mais pratique





Quelques contributions
Certaines indéployables
(rubygem, ...)
D'autres incomplètes
(manque le lag)






Kafka Web Console

 Zookeepers

 Brokers

 Topics

Zookeeper	Topic	Partitions
Test SNAPSHOT	dead-letter-queue	1
Latest	exceptions	1
Latest	metrics	10
Latest	stats	2
Latest	logs	5
Latest	notifications	10
Live	exceptions	1
Live	metrics	10
Live	stats	2
Live	logs	5
Live	notifications	10

Kafka Web Console



Kafka Offset Monitor



Monitoring JMX complet et complexe
Pas d'agrégation niveau cluster





Quelques scripts à
écrire/assembler pour avoir
une vision consolidée des
topics et offsets





Sécurité

Pas le focus du design originel
JIRAs en cours (SSL, encryption, ...)





On a bien sûr pas tout vu
mais l'essentiel



Conclusion



Complexité limitée
Un peu « roots »

Encore quelques défauts
de jeunesse (APIs, ops)
... surmontables

Très performant
Scalable
Sûr





Questions ?



Merci





Sources

<http://kafka.apache.org>

http://fr.slideshare.net/dave_revell/nearrealtime-analytics-with-kafka-and-hbase

<http://fr.slideshare.net/edwardcapriolo/apache-kafka-demo>

<http://aphyr.com/tags/jepsen>

<http://www.michael-noll.com>

http://fr.slideshare.net/Hadoop_Summit/building-a-realtime-data-pipeline-apache-kafka-at-linkedin

<http://www.ibm.com/developerworks/library/j-zero-copy/>

<http://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines>

<http://fr.slideshare.net/carolineboison/apache-kafka-40973171>

<http://blog.confluent.io>