Apache Kafka

Principes et optimisation

Information et communication pour l'ingénieur

Guillaume Laisney

9 mars 2023

le cnam
école d'ingénieur·e·s

Table des matières

En 2011, la publication d'Apache Kafka a créé une rupture dans le domaine de l'ingénierie logicielle, et celui-ci s'est rapidement invité dans de nombreuses entreprises grâce aux performances et aux fonctionnalités qu'il propose. Il est conçu pour répondre à une très large gamme de besoins, et peut se trouver au centre d'architectures guidées par les événements et orientées services.

Architectures guidées par les événements

1 Description

1.1 Éléments principaux

Les architectures guidées par les événements (event driven architectures, EDA) sont construites autour de messages qui reflètent des changements. Ces changements sont des événements, et par métonymie, on donne le nom « événement » aux messages qui les décrivent. Le terme enregistrement est également parfois utilisé.

Les événements déclenchent des communications entre des éléments faiblement couplés qui produisent ou consomment des événements (parfois ils font les deux). On les nomme respectivement producteurs et consommateurs. Enfin, les événements sont immutables et aucun élément de l'architecture ne peut les modifier, il est seulement possible d'y réagir en produisant de nouveaux événements.

Les événements transitent par un broker (routeur ou canal) qui les filtre et les achemine selon les sujets (topics) auxquels ils se rapportent.

Les producteurs et des consommateurs sont faiblement couplés : un producteur ne sait rien des consommateurs situés en aval, comme un consommateur ignore tout de l'émetteur du message qu'il consomme.

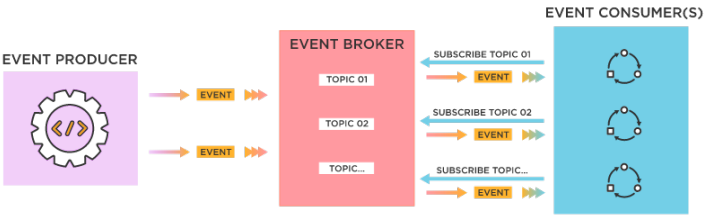


Figure 1: Schéma d'une EDA. Crédit : Tibco

1.2 Couplage

Les EDA offrent un degré de couplage extrêmement faible, leurs composants sont des systèmes autonomes.

1.2.1 Couplage par file

Le routeur peut se comporter comme une simple file (queue) de messages de type « First in, first out », comme le montre le schéma ci-dessous, dans lequel le message m1 est prêt à être consommé.

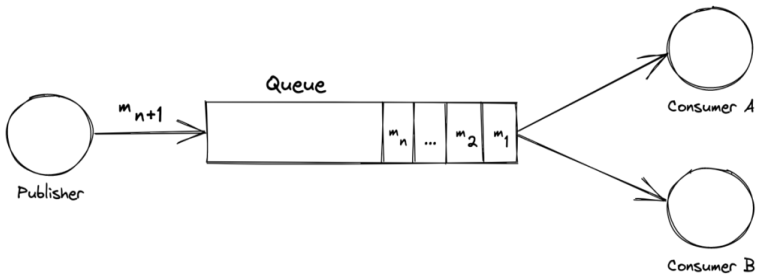


Figure 2: File asynchrone, état initial. Crédit : L. Garvie

Après consommation de m1 par A, l'information n'est plus disponible dans la file.

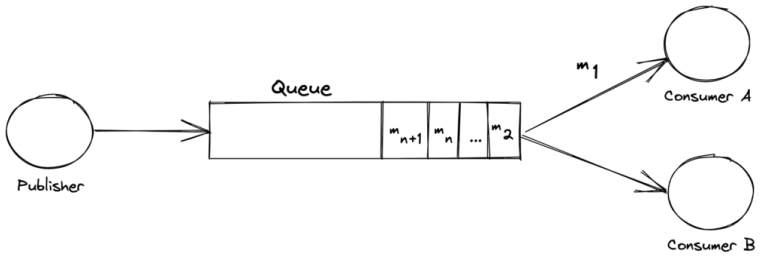


Figure 3: File asynchrone, après consommation de m1. Crédit : L. Garvie

1.2.2 Couplage publish-subscribe

Le routeur peut aussi répartir les messages par sujet (topic) auxquels sont abonnés les consommateurs intéressés, et tout se passe alors comme si chaque consommateur avait accès à son rythme à sa propre file de messages. Ce patron est appelé publish-subscribe.

Quand deux consommateurs sont inscrits à un topic, ils peuvent indépendamment l'un de l'autre consommer leurs messages.

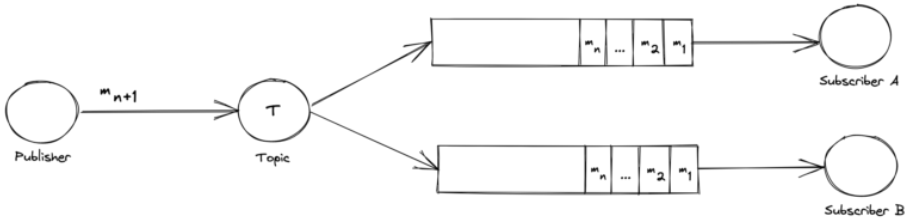


Figure 4: publish-subscribe, avant consommation. Crédit: L. Garvie

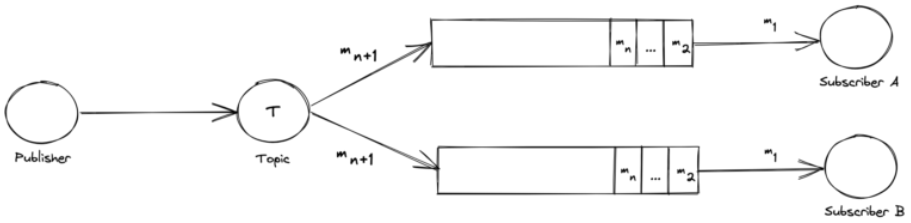


Figure 5: publish-subscribe après consommation. Crédit : L. Garvie

2 Architectures orientées services et EDA

2.1 De l'application monolithique aux services distribués

Le concept d'architecture orientée services est né au début des années 1990 pour répondre à des problèmes d'interopérabilité. On peut y voir une transposition des fondamentaux de la programmation orientée objet : il s'agit principalement de diviser les applications monolithiques en services indépendants. Ces services sont encapsulés et découplés les uns des autres, ils interagissent à l'aide de messages. Chaque service présente une interface qui en définit les méthodes d'accès.

Les architectures orientées services fonctionnent de manière distribuée et les messages sont échangés par des protocoles de communication standardisés.

La division de tâches complexes en services simples et interopérables assure ainsi une forte réutilisabilité, et simplifie considérablement l'évolution des systèmes.

2.2 Communications entre les services

2.2.1 Couplage fort

Les services nécessitent une infrastructure de communication, qui a longtemps été un bus de service synchrone ou une de ses nombreuses variantes (middleware messaging, integration platform, microservice gateway, API management...). Avec ces infrastructures, le fort couplage entre les services pouvait mener à ce qui est parfois appelé une architecture « spaghetti ».

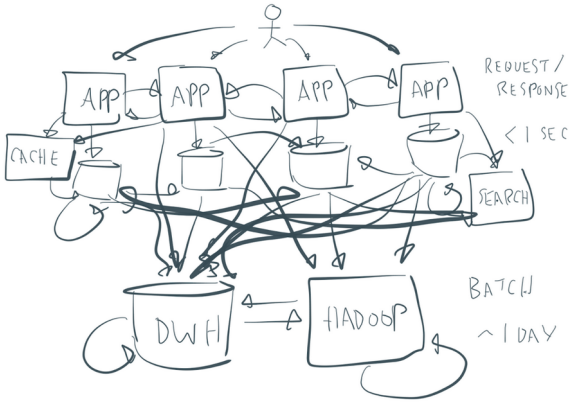


Figure 6: Architecture spaghetti. Crédit : Confluent

2.2.2 Event streaming

Il existe donc un besoin de découplage des services auquel il est possible de répondre par l'utilisation d'une architecture à la fois orientée services *et* guidée par les événements. Néanmoins, le concept d'architecture guidée par les événements est un paradigme qui ne précise pas le détails des échanges d'événements.

Ceux-ci peuvent être gérés par un mécanisme d'event streaming, qui implémente la fonction du broker des architectures guidées par les événements. Ce mécanisme :

- fournit des interfaces entre les éléments de l'architecture (entre des services notamment)
- propose différents modes de livraison des messages (delivery semantics)
- permet le parallélisme de la gestion des événements,

- gère la persistance des informations dans le système
- assure la cohérence de l'ordre des événements.

L'event streaming n'est pas le seul mécanisme existant pour mettre en place une architecture guidée par les événements, mais il a été conçu pour cela et répond parfaitement aux besoins de découplage extrême des architectures orientées services.

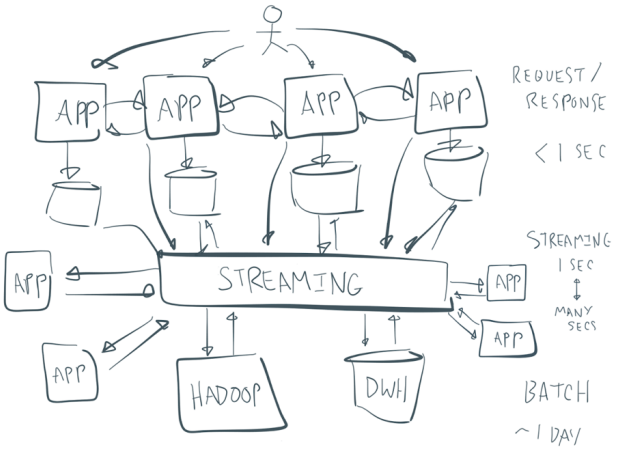


Figure 7: Event streaming. Crédit : Confluent

Apache Kafka

3 Description

Kafka est une plate-forme d'event streaming écrite en Java et en Scala, créée par la société LinkedIn en 2009 et publiée sous licence open source en 2011.

Un point clef du succès que rencontre Kafka est qu'il est capable d'assurer les fonction d'event streaming à très grande échelle de manière extrêmement performante, comme le montre le diagramme ci-après.

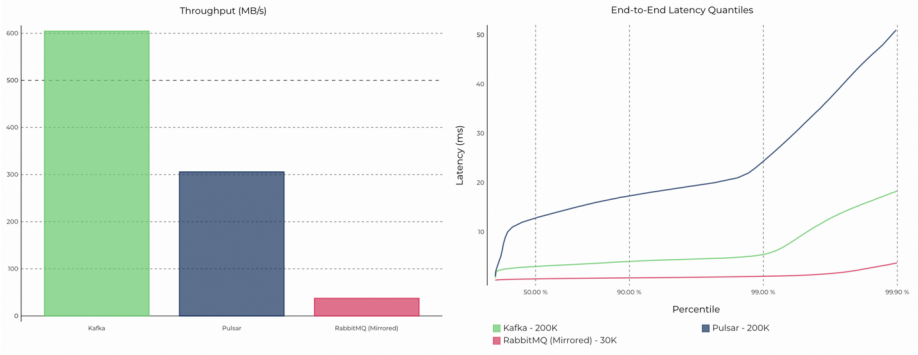


Figure 8: Comparatif de performances Kafka/Pulsar/RabbitMQ.
Crédit : Confluent

3.1 Exemples d'utilisation

Kafka rend des services dans de très nombreux domaines dont voici quelques exemples :

- Traitement guidé par les événements de grandes quantités de données dans des contextes tels que ceux de l'internet des objets, des réseaux sociaux.
- Applications temps-réel critiques : transactions financières, détection de fraude, expérience client.
- Intégration découplée de nouvelles applications dans des systèmes anciens.
- Apprentissage statistique

3.2 Vue d'ensemble

Kafka est un système distribué de serveurs (les brokers) et de clients (producteurs et consommateurs) qui communiquent via un protocole binaire encapsulé dans des segments TCP.

La partie brokers peut être installée sur des machines Linux, Mac OS et même Windows, physiques ou virtuelles, et via des containers. Elle peut bien entendu résider en local ou dans un cloud. Elle est également prévue pour fonctionner en clusters dont il est simple de modifier l'envergure, au sein desquels la chute d'un serveur n'entraîne pas de perte de données ni d'interruption de service.

Kafka fournit la partie producteurs/consommateurs sous la forme d'une interface de programmation d'applications (API). Cette API est disponible dans de nombreux langages (Java et Scala mais aussi Python, C/C++, Go...), elle permet d'écrire des clients producteurs et consommateurs - un client pouvant être les deux simultanément - sous forme d'application ou de service.

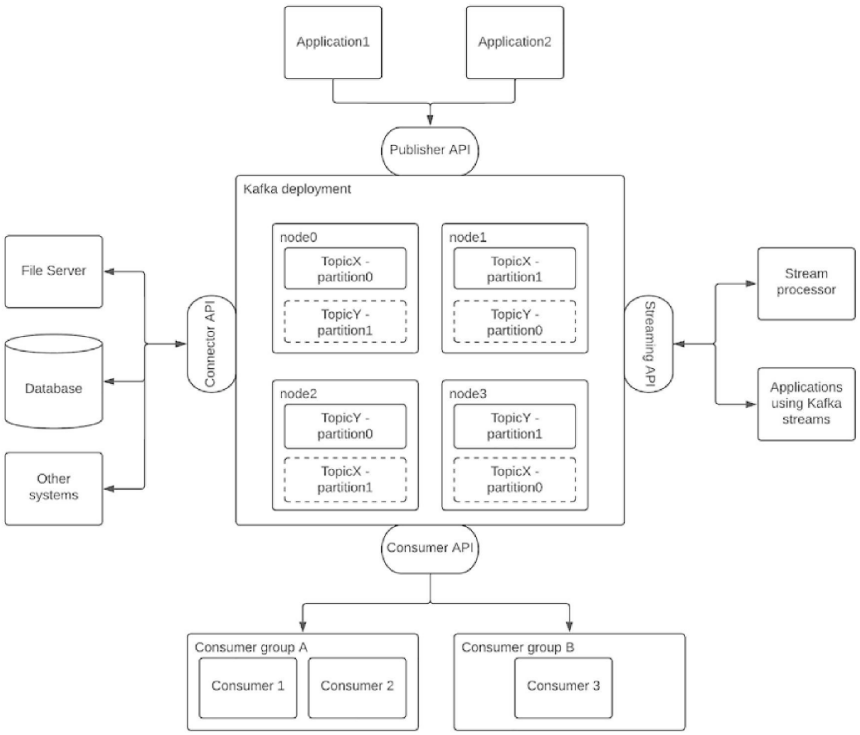


Figure 9: Kafka - description générale. Crédit : C. Fernando

4 Fonctionnement

4.1 Événements

L'unité d'écriture et de lecture de Kafka est l'événement : il s'agit d'un objet qui possède :

- une clef, optionnelle, qui identifie l'événement et qui peut déterminer sa destination à l'intérieur de son topic. (item n°2 sur le schéma)
- une valeur, qui pour Kafka, est un simplement tableau d'octets. (4)
- un horodatage (timestamp). Selon la configuration retenue, l'horodatage est l'heure de création de l'événement par son producteur ou celle de son stockage (ingestion) par le système. (3)

D'autres information liées principalement au transport peuvent être ajoutées dans l'entête de l'événement (1).

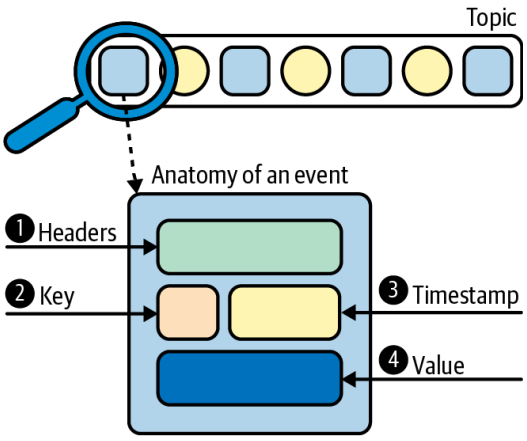


Figure 10: Structure d'un événement. Crédit M. Seymour

4.2 Stockage et lecture des événements dans un topic

4.2.1 Groupes de consommateurs

Pour maximiser le débit des informations (throughput) et minimiser la latence d'accès, plusieurs consommateurs peuvent être groupés. Cela permet de répartir la charge de traitement d'une file sur plusieurs processus : si des consommateurs font partie du même groupe, ils consomment la même file.

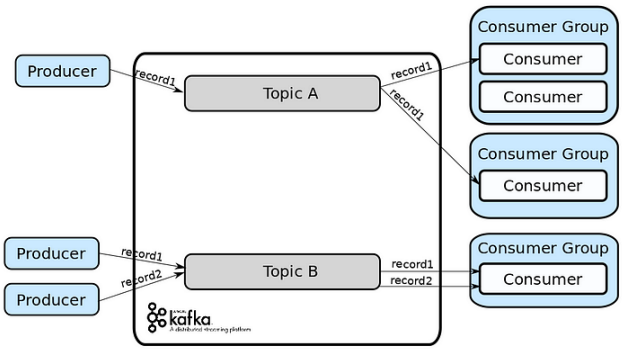


Figure 11: Production et consommation des topics.
Crédit : M. Valcam

Chaque groupe se voit attribuer un broker coordinateur qui est chargé de suivre l'état de santé des consommateurs ; ces derniers lui envoient des « battements de cœur » régulièrement, signifiant ainsi qu'ils sont vivants. Dans le cas contraire, les données qu'ils allaient consommer sont affectées à un autre membre du groupe. Les groupes de consommateurs participent ainsi à la disponibilité du système distribué.

4.2.2 Structure de données

Kafka stocke les événements dans des structures de données appelées « logs » ou « commit logs ». Ce sont des files d'événements auxquelles on ne peut qu'ajouter des éléments ; elles sont totalement ordonnées afin d'assurer un traitement déterministe par tous les processus consommateurs.

Les groupes de consommateurs parcourent les logs indépendamment. Cela est rendu possible par un nombre appelé « offset » associé à chaque groupe et conservé dans un topic interne du système. La consommation d'un événement ne le supprime donc pas du log, elle fait simplement varier l'offset associé au consommateur pour ce log. Cela est peu coûteux et permet à un grand

nombre de consommateurs de travailler simultanément, sans stresser le cluster.

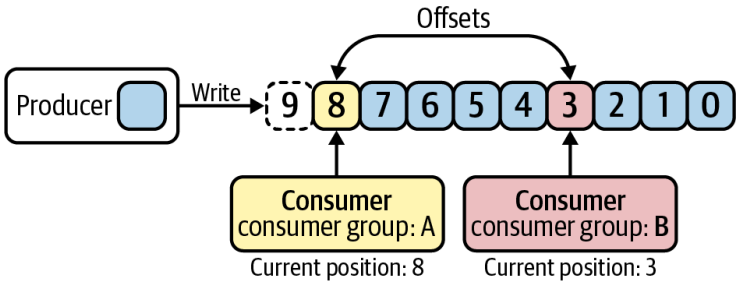


Figure 12: Utilisation d'offsets pour parcourir un log. Crédit : M. Seymour

4.3 Partitions

Les topics peuvent devenir très grands et il est souvent souhaitable de les partitionner pour améliorer les performances et la scalabilité du système. Cela revient concrètement à utiliser plusieurs logs par topic, appelés partitions.

4.4 Clusters de brokers

Un broker est en pratique un serveur Kafka fonctionnant sur une machine. L'architecture de Kafka permet de répartir les partitions sur plusieurs brokers pour former ainsi un cluster de brokers, avec pour effet d'améliorer la tolérance au pannes et la disponibilité du système. Le broker est l'un des éléments de scalabilité de Kafka.

4.4.1 Zookeeper

Zookeeper est une boîte à outils pour systèmes distribués qui est utilisée pour répartir les responsabilités entre les brokers, ainsi que pour effectuer un certain nombre de tâches administratives : suivi de l'état des leaders/followers, informations relatives aux utilisateurs, listes de contrôle d'accès, quotas, etc. Zookeeper est un programme qui fonctionne de manière répartie, souvent sur les mêmes machines que les brokers Kafka.

Kafka deviendra totalement indépendant de Zookeeper à partir de sa version 4.

5 Optimisation

5.1 Capacité à monter en charge

« capacité d'un produit à s'adapter à un changement d'ordre de grandeur de la demande (montée en charge), en particulier sa capacité à maintenir ses fonctionnalités et ses performances en cas de forte demande »

5.1.1 Scalabilité par partitionnement des topics

Si l'on est en capacité de répartir la charge de calcul sur un groupe de N consommateurs, il faudra idéalement diviser un topic en au moins N partitions. S'il y a davantage de consommateurs que de partitions, certains seront en attente, alors que le contraire ne sera pas gênant. La partition est l'unité de stockage de Kafka, on ne peut la répartir sur plusieurs brokers ni même sur plusieurs disques, ce qui rend sa taille limitée par celle du point de montage sur lequel elle se trouve.

Néanmoins, un nombre trop élevé de partitions induirait une surcharge inutile sur le système et il est recommandé de limiter le nombre de partitions par broker à $100 \times b \times r$, avec b nombre de brokers du cluster, et r le facteur de réplication.

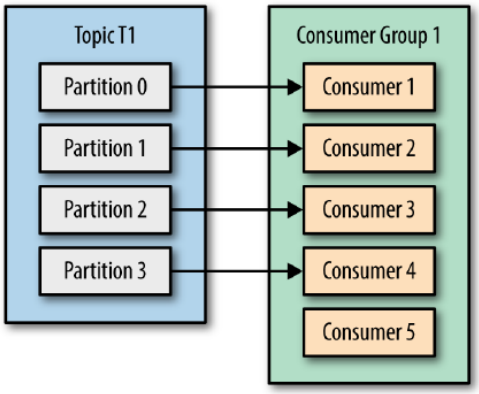


Figure 13: Consommateur privé de travail.
Crédit : N. Narkhede

5.1.2 Répartition de la charge des partitions

La répartition peut être automatique. Dans ce cas, des clefs sont attribuées aux événements par les producteurs, et elles sont utilisées pour choisir la partition de destination ; schématiquement, c'est le reste de la division de leur hashcode (MurmurHash2) par le nombre de partitions du

topic qui est utilisé. Ainsi, tous les événements qui ont la même clef sont groupés sur une même partition, sauf si des partitions sont ajoutées à un topic au cours de son utilisation. Le nombre de partitions doit donc être bien choisi dès la construction des topics.

La responsabilité de lire et d'écrire une partition précise peut aussi être prise par les clients s'ils spécifient la référence de cette partition dans leurs échanges avec le broker.

Enfin, si aucune clef ni partition ne sont spécifiées, une partition est choisie par le système, mais il est également possible d'implémenter un algorithme de partitionnement personnalisé, par exemple d'après le contenu des messages.

5.1.3 Scalabilité par gestion des groupes de consommateurs

La composition des groupes peut évoluer : il est possible d'ajouter des consommateurs à un groupe pour répondre à une augmentation de charge, comme il est possible d'en retirer, par exemple pour une opération de maintenance ou une lors d'une panne.

Une partition ne peut être lue simultanément par plusieurs consommateurs du même groupe. Ceci est à la fois un mécanisme de répartition de charge et de sécurité : c'est le gage qu'un événement ne soit géré que par un processus du groupe à la fois.

5.2 Intégrité et disponibilité des données

La protection de l'intégrité des données et leur disponibilité sont rendues possibles par plusieurs mécanismes relatifs aux partitions : la réplication et la synchronisation.

5.2.1 Rétention

Les événement sont stockés pour une durée limitée par défaut à 7 jours. Il est bien entendu possible de modifier cette valeur, voire de ne jamais supprimer les données. Il est aussi possible de spécifier une taille maximale de partition au delà de laquelle les événements les plus anciens sont supprimés pour faire de la place au nouveaux.

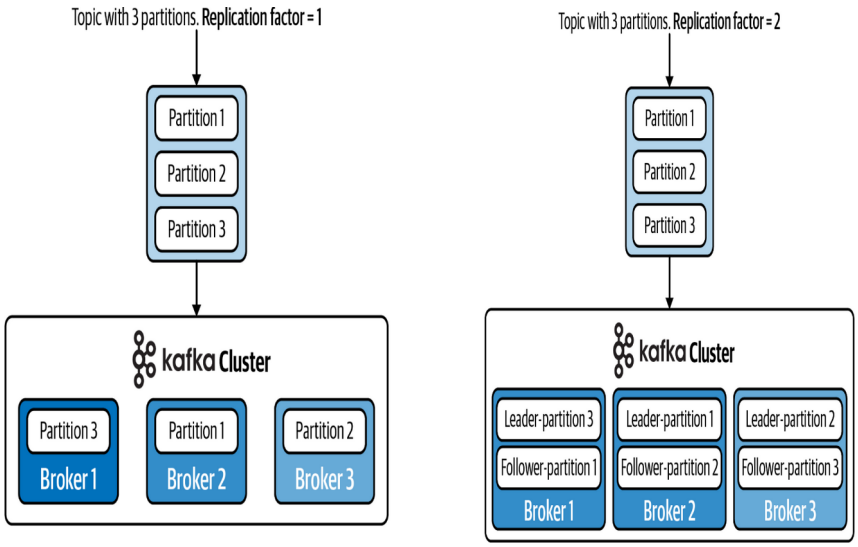
Une compression automatisée des logs est également implémentée ; elle permet de ne conserver que la version la plus récente d'un état régulièrement enregistré dans une partition.

5.2.2 Réplication des partitions

Le facteur de réplication est un nombre que l'on spécifie lors de la création d'un topic. Il correspond au nombre de brokers chargés de répliquer chacune des partitions du topic. Par exemple, un facteur de réplication de 3 permettra à chaque partition d'être stockée sur 3 brokers simultanément.

Quand une partition est répliquée, l'un des brokers est désigné leader pour celle-ci, les autres deviennent des followers. Le leader traite les demandes de lectures/écritures sur la partition, les followers se chargent de copier les données depuis le leader vers les répliques, et en cas de défaillance du leader, l'un des followers est promu leader.

Quand un nouveau broker est ajouté au cluster, le facteur de réplication peut être incrémenté.



5.2.3 Synchronisation des données répliquées

Un mode de synchronisation simple consisterait à attendre que toutes les répliques soient à jour pour accuser réception au producteur, auquel cas il

faudrait attendre la synchronisation de la dernière réplique pour avancer. Pour remédier à cela, Kafka utilise le concept de répliques synchronisées (In-Sync Replicas) : un ensemble dynamiquement alloué de répliques qui montre une bonne capacité à suivre la partition leader et dont les répliques lentes sont automatiquement retirées. Le nombre minimal de répliques synchronisées est paramétrable à l'aide de min.insync.replicas (par défaut, ce paramètre vaut 1). Le délai de réaction avant de considérer qu'une réplique est lente est également ajustable (paramètre replica.lag.time.max.ms)

Le niveau de durabilité peut également être paramétré par un producteur à l'aide du nombre d'accusés de réception requis (paramètre acks). Trois cas sont possibles :

- pas d'accusé de réception (fire and forget) ;
- accusé de réception de la leader seulement ;
- accusé de réception de tous les membres de l'ISR.

5.2.4 Modes de livraison

Le mode de livraison « au plus une fois » garantit qu'un événement ne sera jamais livré en double exemplaire à son destinataire.

Le mode « au moins une fois » signifie qu'un événement ne sera considéré comme reçu par son consommateur. Dans le cas contraire, il lui est envoyé à nouveau.

Le mode « exactement une fois » est plus compliqué à mettre en œuvre puisqu'il implique un couplage fort entre les éléments du système, ce qui n'est pas l'objectif quand on utilise Kafka. Pour parvenir à délivrer les messages exactement une fois sans couplage fort, la solution consiste à rendre les consommateurs *idempotents* : cela signifie que de traitement répété du même événement par ceux-ci n'a au final aucun effet. C'est la combinaison d'un mode de livraison « au moins une fois » et de consommateurs idempotents qui permet d'accéder au mode « exactement une fois ». L'idempotence est implémentée à l'aide d'un mécanisme basé sur des numéros de séquence de croissance monotone et des identifiants attribués aux producteurs.

En fonction des besoins de l'application, on pourra configurer les clients dans l'un des trois modes après avoir choisi un compromis entre faible latence et fiabilité.

5.2.5 Transactions

Les échanges transactionnels donnent accès à un niveau de fiabilité supérieur encore à celui du mode idempotent, au prix d'une complexité accrue.

En mode transactionnel, les itérations d'un client (ici un producteur) peuvent ressembler au code suivant :

```
producer.beginTransaction();
try {

// Production/traitement des événements
//...

//envoi des données traitées
producer.send(...);

// ...

//fin de la transaction
producer.sendOffsetsToTransaction(...);
producer.commitTransaction();

} catch (KafkaException e) {
producer.abortTransaction();
throw e;
}
```

C'est donc sur les clients que repose la charge de bien implémenter la gestion des transactions, mais certaines API comme Kafka Streams les prennent en charge de manière transparente pour le développeur.

5.3 Répartition sur plusieurs centres de données

Plusieurs architectures ont été utilisées avec succès pour répartir Kafka sur plusieurs centres de données. Chacune induit un certain nombre de compromis, et il peut se trouver nécessaire de les panacher.

5.3.1 Architecture en étoile

Cette architecture peut être utilisée quand des données sont produites dans des lieux différents et que certains consommateurs nécessitent un accès à la totalité de celles-ci. Les données sont toujours produites localement, et sont répliquées une fois seulement vers le site central et cette architecture est simple à mettre en œuvre. En revanche, l'accès direct aux données d'un centre producteur par un autre n'est pas possible.

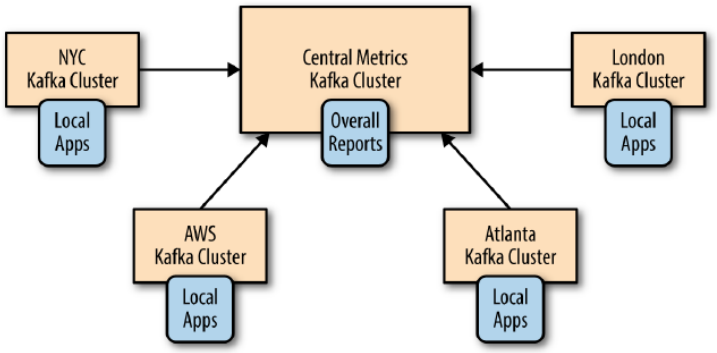


Figure 14: Architecture "Hub and spokes". Crédit : N. Narkhede

5.3.2 Architecture actif-actif

Ce modèle peut être utilisé quand plusieurs centres partagent, produisent et consomment des événements. La multiplication des centres de données augmente la redondance et la résilience du système. D'autre part, les utilisateurs proches d'un centre de données obtiendront des performances correctes sans limitations des fonctionnalités de Kafka. L'inconvénient principal de ce modèle réside dans la complexité induite par la gestion des lectures/écritures asynchrones réparties, et la cohérence des données.

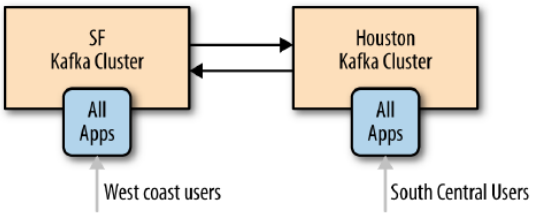


Figure 15: Architecture "actif-actif". Crédit: N. Narkhede

5.3.3 Architecture « actif-passif »

Utiliser deux centres de données distincts peut servir à remédier aux catastrophes qui pourraient advenir dans l'un d'eux. Dans ce cas, un cluster dit « passif» est uniquement chargé de répliquer les données du cluster « actif ». En cas de défaillance de celui-ci, un appel à un administrateur systèmes permet de basculer vers le cluster de secours. Ce modèle est simple à mettre en œuvre mais ne fait qu'un usage minimaliste des capacités du cluster de secours. De plus, quelques événements peuvent ne pas avoir été répliqués au moment de la défaillance et cette perte potentielle de données est rédhibitoire pour certains usages.

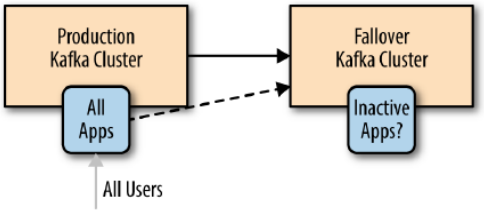


Figure 16: Architecture "actif-passif". Crédit: N. Narkhede

5.3.4 Cluster étiré

Cette architecture est composée d'un seul cluster, réparti sur plusieurs centres de données. Dans cette configuration, c'est Kafka qui gère la réplication de l'information et celle-ci peut alors être synchrone. En contrepartie, si elle protège bien le système d'une défaillance d'un centre de données, elle ne le protège pas d'un problème lié aux applications ou à Kafka. Enfin, il faut en pratique au moins trois centres de données distincts, pourvus d'une interconnexion à très grand bande passante et à très petite latence.

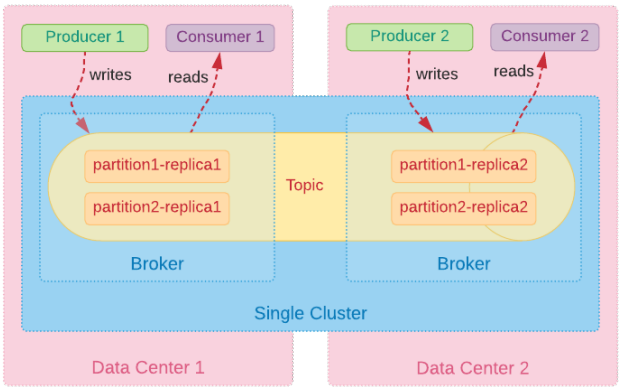


Figure 17: Cluster étiré. Crédit : Mateus Bukowicz

5.4 Performances

5.4.1 Optimisation par « zéro copie »

Pour gagner en performances, Kafka met en pratique une idée simple : éviter toute copie inutile d'informations par les CPU en mémoire centrale. Les événements qui transitent par les brokers sont donc copiés des contrôleurs réseaux vers les contrôleurs disques sans passage en mémoire centrale, avec un nombre de passages minimal en mode noyau. Ce mode de transfert est pris en charge par le package java.nio.channels.FileChannel avec la fonction transfertTo() qui peut aller jusqu'à tripler le débit de données par rapport à un transfert après passage en mémoire centrale, tout en évitant de solliciter les CPU. Le choix des contrôleurs réseau et disque revêt donc une certaine importance pour obtenir des brokers performants.

5.4.2 Lecture/écriture

L'utilisation de logs séquentiels append-only a une conséquence importante sur les performances de Kafka : les entrées sorties sont en très grande majorité séquentielles, or les périphériques de stockages sont beaucoup plus performants avec cette forme d'accès.

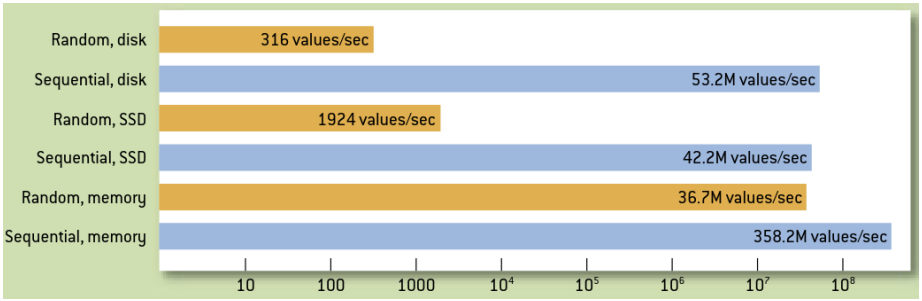


Figure 18: Performances en séquentiel vs random. Crédit : A. Jacobs

Des tests en lecture/écriture séquentielles pourront donc être réalisés sans exclure les disques mécaniques lors de l'optimisation matérielle du cluster.

5.4.3 Compression

Les débits atteints sur les machines grâce au stockage séquentiel des événements son tels que les réseaux deviennent souvent le goulot d'étranglement du système.

Pour contrecarrer cet effet, les événements sont groupés lors de leur transmission (batching) dès que la charge du système augmente. Ils sont groupés par les producteurs et stockés tel-quels sans intervention du broker en respectant le principe « zéro copie », puis acheminés vers les consommateurs groupés de la même manière. Il est possible d'optimiser manuellement le groupage des messages (paramètres batch.size et linger.ms)

Pour aller plus loin, un algorithme de compression peut être adjoint au batching. Cela permet d'atteindre des quotients de compression de 7 sur des messages à faible entropie (par exemple JSON) quand ils sont groupés. Les effets positifs sont très importants, non seulement sur le trafic réseau mais également sur les disques des brokers. Enfin, la charge induite par les opérations de compression/décompression est très majoritairement portée par les clients, à condition de paramétrer les brokers pour qu'ils acceptent le type de compression proposée par ceux-ci.

Kafka permet d'utiliser plusieurs algorithmes de compression qui peuvent être plus ou moins performants suivant le type de données à transmettre : il s'agit de gzip, snappy, lz4 et zstd qu'il peut être intéressant d'évaluer sur des données réelles.

5.5 Confidentialité de l'information

Kafka stocke des données potentiellement sensibles, est peut être utilisé comme un moyen de communication central par une organisation. Cela fait de lui une cible attractive pour des attaquants potentiels.

5.5.1 Confidentialité par défaut

Par défaut, la confidentialité de Kafka est... nulle. Les premières versions n'intégraient pas d'éléments de sécurité, et pour des raisons de compatibilité avec les anciens clients, les réglages par défaut sont restés en mode « ouvert », dans lequel les performances sont maximales mais dont l'usage est limité à des systèmes fermés.

Ainsi, par défaut :

- N'importe quel client peut établir une connexion vers un broker ou un zookeeper, non seulement sur les ports 2181 et 9092 mais aussi sur les ports de diagnostic « Java management extensions » ;
- Les connexions TCP vers les brokers ne sont pas cryptées, et il est facile de sniffer les paquets pour en obtenir et éventuellement modifier le contenu ;
- Les connections entre les brokers sont également non cryptées ;
- Les connections vers les brokers ont lieu sans authentification ;
- Il n'y a pas de contrôle d'autorisation, et même avec l'authentification activée, un client peut effectuer n'importe quelle action sur un broker.

5.5.2 Ségrégation réseau

L'utilisation de firewalls pour segmenter l'architecture et de réseaux privés virtuels pour les clients distants est une première mesure efficace qui permet de largement réduire la surface d'attaque disponible.

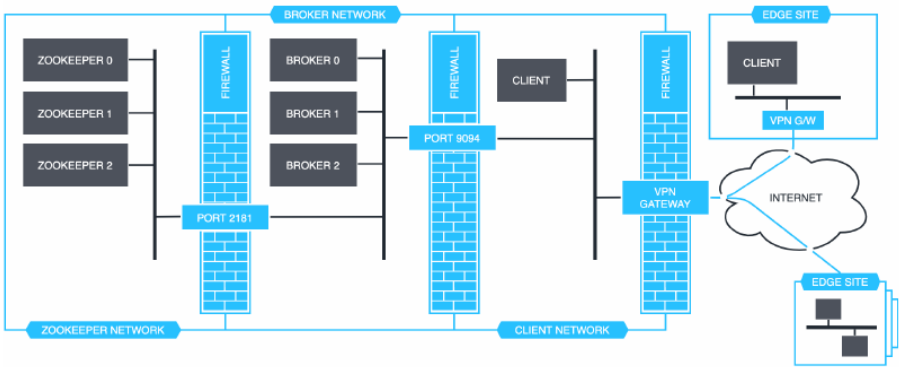


Figure 19: Ségrégation réseau. Crédit : E. Koutanov

5.5.3 Chiffrement du trafic réseau

Kafka permet de chiffrer les informations transmises à l'aide de TLS 1.3 (Transport Socket Layer) sous réserve d'utiliser une machine virtuelle Java de version supérieure à 11. les connexions clients-brokers, inter-brokers et brokers-zookeepers peut être sécurisées de cette manière.

5.5.4 Chiffrement des données stockées

Kafka ne propose nativement pas moyen de chiffrement des logs. On peut, de manière traditionnelle, chiffrer les disques ou les systèmes de fichiers mais pour protéger les données de processus malveillants s'exécutant sur les serveurs, un chiffrement point à point est recommandé. Il est alors possible de mettre en place soit même le chiffrement dans le code des clients, ou d'utiliser des frameworks comme QuickSign Kafka Encryption et Kafka End2End encryption.

Le chiffrement point à point augmente l'entropie des messages de manière considérable et rend la compression nuisible, car celle-ci demande une certaine charge de calcul et augmente en général la taille des messages.

5.5.5 Authentification

Kafka propose plusieurs modes d'authentification des clients : TLS mutuel, SASL (Simple Authentication and Security Layer) et GSSAPI (Generic Security Service API).

Il est aussi possible de mettre en place une authentification entre les brokers ainsi qu'entre les brokers et zookeeper en SASL.

5.5.6 Autorisations

Les autorisations sont gérées par des listes de contrôle d'accès (ACL), qui permettent de décrire finement ce qu'une application a la possibilité de faire et ce qui lui est interdit. Il est possible de gérer une dizaine d'opérations différentes en précisant sur quels hôtes, sur quels types de ressources elles peuvent ou non être réalisées.

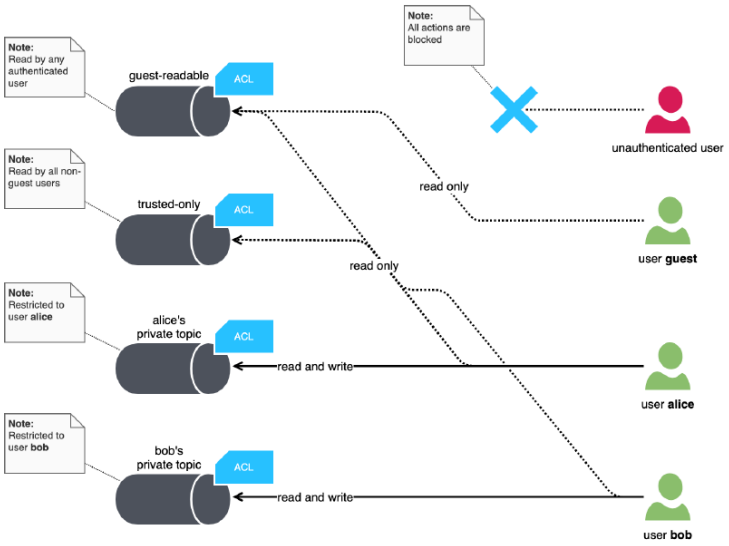


Figure 20: Exemple de gestion des autorisations. Crédit : E. Koutanov

5.6 Disponibilité et quotas

La disponibilité des données peut être un enjeu de sécurité dans certaines applications, et Kafka est une cible a priori facile pour une attaque par déni de service (DoS). Plusieurs applications d'importances différentes sont susceptibles de partager les mêmes ressources, et il est facilement imaginable qu'une application accessoire, volontairement ou non d'ailleurs, sature des ressources nécessaires au fonctionnement d'une application critique.

L'utilisation de quotas est gérée nativement par Kafka. Elle permet de se protéger d'attaques DoS, mais aussi de répondre à des besoins de qualité de service (QoS) et peut être intégrée dans la gestion de capacité du système.

Il existe deux types de quotas :

- Les quotas sur bande passante qui empêchent les clients de dépasser un taux de transfert donné sur un broker (en octets par seconde) ;
- Les quotas sur taux de requêtes limitent l'utilisation des CPU des brokers.

6 Synthèse performance - sécurité de l'information

Ce chapitre présente une rapide synthèse des choix à effectuer pour maximiser performances ou sécurité.

6.1.1 Synchronisation

Un premier choix devra être fait sur le mode de synchronisation des partitions répliquées : pas de synchronisation, partition leader seule ou encore tous les membres de l'ISR, et dans ce dernier cas faire varier le nombre minimal de partitions synchrones.

6.1.2 Mode de livraison

Le mode de livraison est également un curseur à ajuster. Il faudra choisir entre une sécurité maximale avec le mode « exactement une fois », une rapidité maximale en mode « au plus une fois », ou un compromis avec possibilité de doublons en mode « au moins une fois ».

6.1.3 Transactions

La sécurité des données peut encore être augmentée au détriment des performances à l'aide du mécanisme de transactions proposé par Kafka.

6.1.4 Compression ou chiffrement

Les capacités du réseau seront beaucoup moins limitantes et les lectures/écritures seront plus rapides en activant la compression point-à-point, mais elle n'est pas compatible avec le chiffrement des événements du point de vue des performances.

6.1.5 Architectures

Choix de l'architecture de répartition sur plusieurs centres de données et l'architecture réseau, notamment les performances des firewalls seront très déterminants.

7 Exemples de clients Java

7.1 Exemple de producteur

Cet exemple simple produit toutes les 3 secondes un nouvel événement dont la clef est « maClef », qui a pour valeur une chaîne de caractères de forme « Publication d'un nouvel événement du [date] » .

Le mode idempotent est activé, le broker est local, sur le port 9092.

Le topic demo-ENG221 sera créé s'il n'existe pas à condition que le broker le permette.

```
public static void main(String[] args)
        throws InterruptedException {
    final var topic = "demo-ENG221";

    //Création d'une map pour les paramètres client
    final Map<String, Object> config =
            Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                    v1: "localhost:9092",
                    ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                    StringSerializer.class.getName(),
                    ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                    StringSerializer.class.getName(),
                    ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
                    v4: true);

    try (var producer = new KafkaProducer<String, String>(config)) {
        while (true) {
            // Boucle de production
            final var key = "maClef";
            final var value = new Date().toString();
            out.format("Publication d'un nouvel événement du %s%n",
                    value);

            final Callback callback = (metadata, exception) -> {
                out.format("Publié, metadata: %s, erreur: %s%n", metadata, exception);
            };
            producer.send(new ProducerRecord<>(topic, key, value), callback);
            Thread.sleep( l: 3000);
        }
    }
}
}
```

7.2 Exemple de consommateur

Ce consommateur vérifie 10 fois par seconde la présence d'un nouvel événement dans le topic demo-ENG221 et, le cas échéant, le consomme en affichant sa valeur. Il fait partie du groupe « exemple-consommateur » et lira les messages les plus anciens si son groupe est nouvellement créé (auto.offset.reset=earliest). Enfin l'enregistrement du changement d'offset ne sera pas effectué automatiquement mais après chaque lecture et de manière asynchrone.

```
public static void main(String[] args) {
    final var topic = "demo-ENG221";

    final Map<String, Object> config =
            Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                    "localhost:9092",
                    ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                    StringDeserializer.class.getName(),
                    ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                    StringDeserializer.class.getName(),
                    ConsumerConfig.GROUP_ID_CONFIG,
                    "exemple-consommateur",
                    ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
                    "earliest",
                    ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
                    false);

    try (var consumer = new KafkaConsumer<String, String>(config)) {
        consumer.subscribe(Set.of(topic));
        while (true) {
            final var records = consumer.poll(Duration.ofMillis(100));
            for (var record : records) {
                out.format("Reçu événement de valeur%s%n", record.value());
            }
            consumer.commitAsync();
        }
    }
```

Conclusion

Kafka permet de manipuler des abstractions de haut niveau et soulève des questions architecturales, mais les optimisations dont il bénéficie sont souvent basées sur des mécanismes très proches du matériel, et tout cela rend son étude très riche et intéressante.

Enfin, en plus de très probables rencontres professionnelles futures avec Kafka, une suite à court terme de ce travail consistera certainement à refondre le code d'un projet personnel dans lequel il pourrait être d'un grand service. Ce projet mélange des parties temps réel avec de l'apprentissage statistique et souffre d'une certaine rigidité dans sa structure. De plus, Kafka permettra d'intégrer plus facilement des parties écrites en Python au code Java existant.

Bibliographie

Neha Narkhede, Gwen Shapira, and Todd Palino, Kafka: The Definitive Guide, 2017

Chanaka Fernando, Solution Architecture Patterns for Enterprise, A Guide to Building Enterprise Software Systems, 2023

Pethuru Raj, Anupama Raman, Harihara Subramanian. Architectural Patterns, Uncover essential patterns in the most indispensable realm of enterprise architecture

Dylan Scott, Viktor Gamov, Dave Klein, Kafka in Action, 2022

Emil Koutanov, Effective Kafka, A Hands-On Guide to Building Robust and Scalable Event-Driven Applications, 2021

https://www.confluent.io/blog/apache-kafka-vs-enterprise-service-bus-esb-friends-enemies-or-frenemies/

https://www.baeldung.com/pub-sub-vs-message-queues

https://kafka.apache.org/

https://www.quora.com/What-are-the-benefits-of-Apache-Kafkas-native-binary-TCP-protocol-over-its-restful-API

https://medium.com/hacking-talent/kafka-all-you-need-to-know-8c7251b49ad0

https://www.confluent.io/blog/kafka-fastest-messaging-system/

https://stackoverflow.com/questions/66863170/encrypt-data-in-kafka

http://mbukowicz.github.io/kafka/2020/08/31/kafka-in-multiple-datacenters.html