Bases de données et environnements distribués
Chapitre IV : Intergiciels à Messages (Message Oriented Middleware MOM)
Application avec JMS/JORAM et Apache Kafka

Éric Leclercq

IEM/LE2I

révision. nov. 2017 émail : Eric.Leclercq@u-bourgogne.fr
http://ludique.u-bourgogne.fr/leclercq
http://ufrsciencestech.u-bourgogne.fr

Plan du chapitre

Attention

Les exemples sont développés avec JDK 1.4 / J2EE 1.4

Les principes des MOM

- Les systèmes de communication asynchrones reposent sur l'envoi de messages
- Très utilisés pour les applications réparties utilisant Internet ou des équipements mobiles ou encore pour des applications dont le temps de réponse aux requêtes est très long
- Adaptés pour gérer des interactions faiblement couplées entre des systèmes selon deux aspects du couplage :
  - couplage spatial : éloignement des entités
  - couplage temporel : déconnexion temporaire

  Il existe cependant de nombreuses autres formes du couplage (pensez à RMI, CORBA, JDBC, etc.)

Les classes d'applications des MOM

Les MOM constituent une famille d'intergiciels utilisés pour les types d'application suivants :

- systèmes traitant de la mobilité des usages, systèmes ubiquitaires
- intégration de données et d'applications (EAI, EII, ESB, MDM, etc.)
- surveillance et contrôle des réseaux (des équipements)
- traitement de données massives (architectures Big Data Analysis)

Les concepts des MOM

Du point de vue du modèle de communication les MOM utilisent les principes et les modèles de communication suivants :

- Les entités qui communiquent sont découplées : l'émission (production) d'un message est une opération non bloquante, un émetteur (producteur) et récepteur (consommateur) ne communiquent pas directement entre eux, ils utilisent un objet intermédiaire (boîte aux lettres, ou système d'acheminement de messages, ou autre).
- Deux modèles de communication sont proposés :
  - un modèle point à point dans lequel un message produit est consommé par un destinataire unique ;
  - un modèle multi-points dans lequel un message peut être adressé à une communauté de destinataires (communication de groupe).

Le modèle multipoints est généralement complété avec un système de désignation associative : les destinataires d'un message sont identifiés par une propriété du message.

Environnement Technologique

Le modèle multipoints est à l'origine du modèle Publication/Abonnement (*Publish/Subscribe*) largement répandu.

Pendant de nombreuses années les MOM ont été impactés par l'absence de normalisation : ils sont restés propriétaires du point de vue du modèle et de leur implémentation.

Fin des années 90 : rédaction de la spécification JMS (*Java Messaging Service*) et des serveurs d'applications JEE.

JMS définit le modèle de d'interaction et l'API correspondante : en revanche il n'y a pas vraiment de tentative de normalisation/standardisation sur le protocole de l'intergiciel lui-même, ni même sur les mécanismes d'interopérabilité (exemple CORBA avec le protocole IIOP).

Spécification JMS

JMS est la spécification d'un MOM Java : JMS décrit l'interface de programmation pour utiliser un bus à messages asynchrone, c'est-à-dire la manière de programmer les échanges entre composants logiciels.

La structure d'une application JMS comporte :

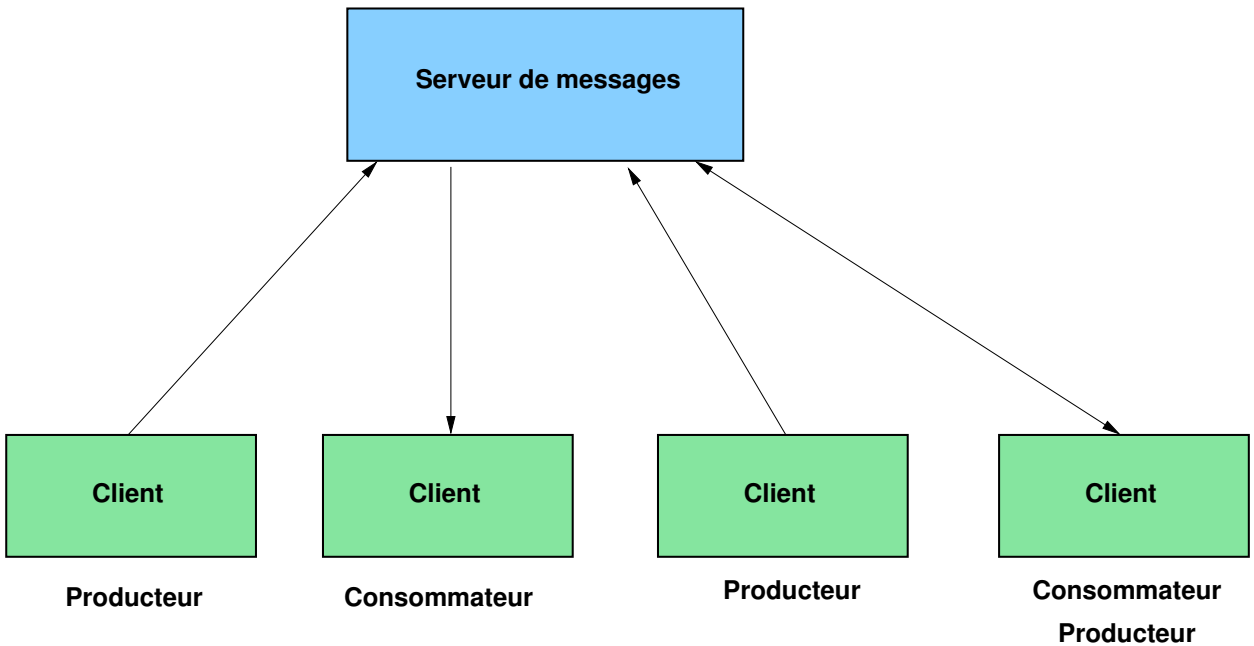
- Le système de messagerie JMS (provider) : on distingue le service de base qui met en œuvre les abstractions du modèle d'interaction et une bibliothèque de fonctions qui met en œuvre l'interface de programmation JMS.
- Les clients JMS (client) : programmes ou applications, qui produisent (émettent) et consomment (reçoivent) des messages JMS
- Les messages JMS : objets qui permettent de véhiculer des informations entre les clients JMS (texte structuré XML, format binaire, objets Java sérialisés, etc.)

Modes de communication JMS

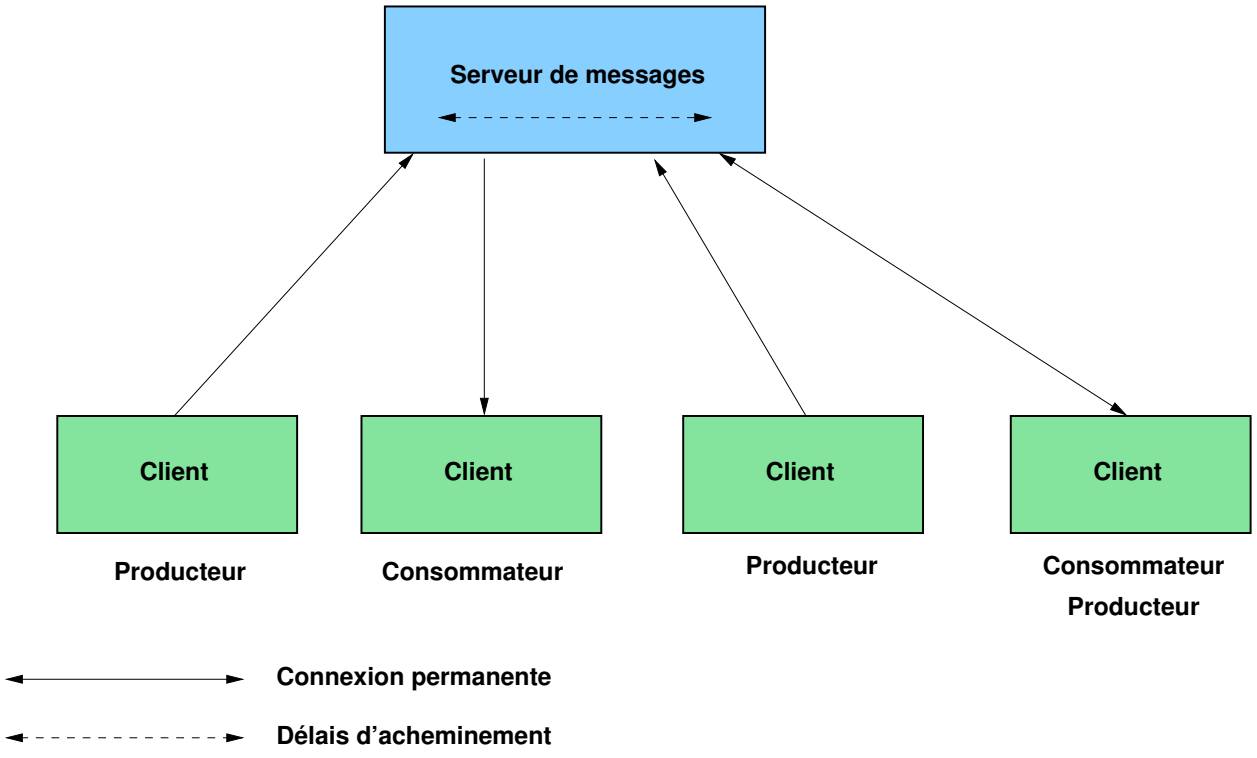
Deux modes de communication (*Messaging Domains*) sont décrits dans la spécification JMS :

1. point à point utilisant des files de message (*Queues*) : un message est adressé à une file par un client producteur, il est extrait par un client consommateur. Un message est consommé par un seul client.
   - Le message est stocké dans la file jusqu'à sa consommation ou jusqu'à l'expiration d'un délai d'existence
   - La consommation d'un message peut être synchrone/asynchrone
   - La consommation du message est confirmée par un accusé de réception généré par le système ou le client
2. multipoints avec modèle *Publish/Subscribe* : un client producteur émet un message concernant un sujet prédéterminé (*Topic*) . Tous les clients préalablement abonnés à ce Topic reçoivent le message correspondant.

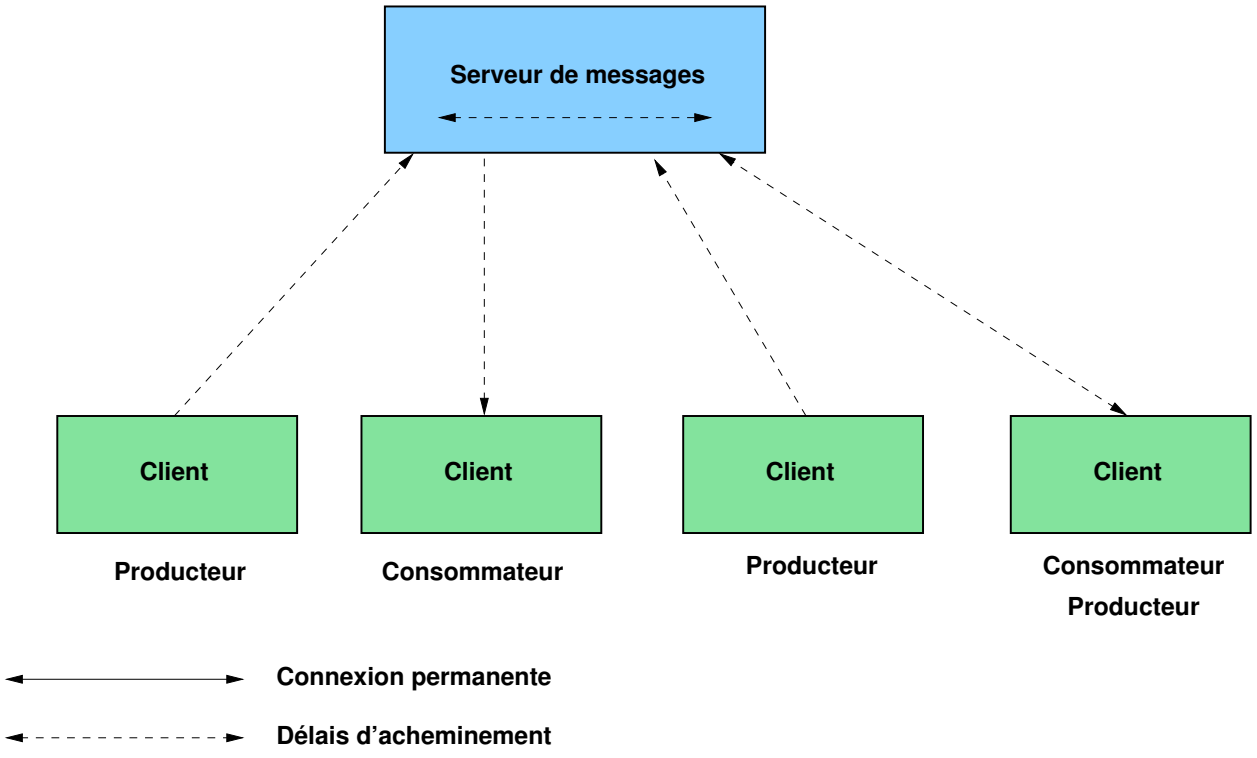
Architectures envisageables



Architectures envisageables



Architectures envisageables



L'API JMS

Les caractéristiques de la dernière version de la spécification JMS (1.1) de 2002 :

- Unifie la manipulation des deux modes de communication
- Le terme *Destination* représente soit une *Queue* de message, soit un *Topic*
- L'unification simplifie l'API (les primitives de production, consommation sont syntaxiquement fusionnées) et permet une optimisation de la communication.
- Elle doit permettre d'utiliser le mode transactionnel dans les deux modèles de communication.
- La spécification JMS propose également des options de qualité de service, en particulier pour ce qui concerne les abonnements (temporaires ou durables) et la garantie de délivrance des messages via la propriété de persistance.

Les limites des spécifications JMS

- JMS ne donne aucune indication sur la mise en œuvre du service de messagerie qui est donc une implantation spécifique (propriétaires souvent) aux éditeurs
- La portabilité est réduite du fait que les fonctions d'administration sont propres à chaque plate-forme
- L'interopérabilité entre deux clients JMS implantés sur des plates-formes différentes n'est pas garantie
- La spécification JMS n'est pas complète : cas en particulier pour la configuration et l'administration du service de messagerie, la sécurité (intégrité et chiffrage, confidentialité des messages) et certains paramètres de qualité de service.

Caractéristiques principales de JORAM

- JORAM (Java Open Reliable Asynchronous Messaging) est une implémentation de la spécification JMS 1.1 (incluse dans la spécification J2EE 1.4).
- JORAM est un composant libre faissant partie des projets ObjectWeb sous licence LGPL - `http://joram.objectweb.org`
- JORAM peut être utilisé de deux façons complémentaires :
  - Comme un système de messagerie Java autonome entre des applications JMS développées pour des environnements variés (de J2EE à J2ME).
  - Comme un composant de messagerie intégré dans un serveur d'application J2EE (inclus dans JOnAS - `http://jonas.objectweb.org`)

Structuration et communications

- Comme toutes les autres implémentations de JMS, JORAM est structuré en deux parties :
  - une partie serveur qui gère les objets JMS : `Queues`, `Topics` , `connexions`, etc.
  - une partie client JORAM liée à l'application cliente JMS
- le serveur JORAM peut être mis en œuvre de façon centralisée ou de façon distribuée.
- La communication s'appuie sur le protocole TCP/IP, une variante, consiste à utiliser le protocole HTTP/SOAP pour les clients JMS développés dans l'environnement J2ME.
- La communication entre deux serveurs JORAM peut utiliser différents types de protocoles selon les besoins : TCP/IP, HTTP, SOAP, sécurisation via SSL.

Structuration des communications

JMS strcuture les deux modèles de communication Point-à-Point et Publish/Subscribe autour des éléments suivants :

- `ConnectionFactory` : objet d'administration utilisé par le client JMS pour créer une connexion avec le système de messagerie
- `Connection` : une connexion active avec le système de messagerie
- `Session` : un contexte (mono-thread) pour émettre et recevoir des messages
- `Destination` : l'objet de communication entre deux clients JMS désignant
  - la destination des messages pour un producteur
  - la source des messages pour un consommateur.

  La destination désigne une `Queue` ou un `Topic`

Structuration des communications

- `MessageProducer` : un objet créé par une session et utilisé pour émettre des messages à un objet Destination
- `MessageConsumer` : un objet créé par une session et utilisé pour recevoir les messages déposés dans un objet Destination

Note :

Pour déterminer les noms des classes préfixer par Queue ou Topic les classes du modèle général respectivement pour les communications PàP et Publish / Subscribe

Communication côté client

Un client JMS va exécuter la séquence d'opérations suivante :

1. Rechercher un objet `ConnectionFactory` dans un annuaire en utilisant l'API JNDI (Java Naming and Directory Interface)
2. Utiliser l'objet `ConnectionFactory` pour créer une connexion JMS, il obtient alors un objet `Connection`
3. Utiliser l'objet `Connection` pour créer une ou plusieurs sessions JMS, il obtient alors des objets `Session`
4. Utiliser le répertoire pour trouver un ou plusieurs objets `Destination`
5. Créer les objets `MessageProducer` et `MessageConsumer` pour émettre et recevoir des messages (à partir d'un objet `Session` et des objets `Destination`)

Les messages

Un message JMS est composé de trois parties :

- Un en-tête : contient l'ensemble des données utilisées à la fois par le client et le système pour l'identification et l'acheminement du message
- Des propriétés : en plus des champs standard de l'en-tête, les messages permettent d'ajouter des champs optionnels sous forme de propriétés normalisées, ou de propriétés spécifiques à l'application ou au système de messagerie
- Un corps : JMS définit différents types de corps de message afin de répondre à l'hétérogénéité des systèmes de messagerie.

En-tête des messages

Un en-tête de message JMS est composé des champs suivants :

- `JMSDestination` : contient la destination du message, il est fixé par la méthode d'envoi de message en fonction de l'objet Destination spécifié
- `JMSDeliveryMode` : définit le mode de délivrance du message (persistant ou non) ; il est fixé par la méthode d'envoi de message en fonction des paramètres spécifiés
- `JMSMessageId` : est un identificateur pour chaque message envoyé par un système de messagerie. Il est fixé par la méthode d'envoi de message et peut-être consulté après envoi par l'émetteur
- `JMSTimeStamp` : contient l'heure de prise en compte du message par le système de messagerie, il est fixé par la méthode d'envoi de message
- `JMSReplyTo` : contient la Destination à laquelle le client peut éventuellement émettre une réponse. Il est fixé par le client dans le message.

En-tête des messages

- `JMSExpiration` : ce champs est calculé comme la somme de l'heure courante (GMT) et de la durée de vie d'un message (time-to-live). Lorsqu'un message n'est pas délivrée avant sa date d'expiration il est détruit ; aucune notification n'est définie pour prévenir de l'expiration d'un message.
- il existe également `JMSCorrelationId`, `JMSPriority`, `JMSRedelivered`, `JMSType`

Propriétés et Corps

Les propriétés permettent à un client JMS de sélectionner les messages en fonction de critères applicatifs :

- Un nom de propriété doit être de type String ; une valeur peut être : null, boolean, byte, short, int, long, float, double et String.
- Un client peut ainsi définir des filtres en réception dans l'objet `MessageConsumer` : la syntaxe est basée sur un sous-ensemble de la syntaxe d'expression de conditions du langage SQL.

JMS définit cinq formes de corps de messages :

- `StreamMessage` : message dont le corps contient un flot de valeurs de types primitifs Java ; il est rempli et lu séquentiellement
- `MapMessage` : le corps est composé d'un ensemble de couples noms-valeurs
- `TextMessage` : le corps est une chaîne de caractère (String)
- `ObjectMessage` : le corps contient un objet Java sérialisable

Les objets JMS

- Objets d'administration :
  - l'objet Destination qui encapsule les différents formats d'adresses des systèmes de messagerie
  - l'objet ConnectionFactory encapsule l'ensemble des paramètres de configuration définit par l'administrateur, il est utilisé par le client pour initialiser une connexion avec le système de messagerie.
- Objet Connection : représente une connexion active avec le système de messagerie.
  - Lors de sa création, le client peut devoir s'authentifier
  - L'objet Connection permet de créer une ou plusieurs sessions
  - Lors de sa création une connexion est dans l'état stoppé, elle doit être explicitement démarrée pour recevoir des messages.

Les objets JMS

- Objet Session : c'est un contexte mono-thread pour produire et consommer des messages.
  - construit les objets MessageProducer et MessageConsumer
  - crée les objets Destination et Message
  - gère l'acquittement des messages
  - réalise l'ordonnancement des messages reçus et envoyés
  - supporte les transactions et permet de grouper plusieurs opérations de réception et d'émission dans une unité atomique qui peut être validée (resp. invalidée) par la méthode Commit (resp. Rollback)

Note :

Une session permet la création de plusieurs objets `MessageProducer` et `MessageConsumer` cependant ils ne doivent être utilisés que par un flot d'exécution (thread). Dans le mode Publish/Subscribe, si deux sessions s'abonnent à un même sujet, chaque client abonné (TopicSubscriber) reçoit tous les messages émis sur le Topic.

Les objets JMS

- Objet MessageConsumer :
  - Un client utilise un objet MessageConsumer pour recevoir les messages envoyés à une destination particulière.
  - Créé en appelant la méthode CreateConsumer de l'objet Session avec un objet Destination en paramètre.
  - Un objet MessageConsumer peut être créé avec un sélecteur de message pour filtrer les messages à consommer.
  - JMS propose deux modèles de consommation des messages :
    - le modèle synchrone : un client demande le prochain message en utilisant la méthode Receive de l'objet MessageConsumer (mode de consommation Pull).
    - le mode asynchrone : le client enregistre au préalable un objet qui implémente la classe MessageListener dans l'objet MessageConsumer ; les messages sont alors délivrés lors de leur arrivée par appel de la méthode onMessage sur cet objet (mode de consommation Push).

Les objets JMS

- Objet MessageProducer
  - Un client utilise un objet MessageProducer pour envoyer des messages à une destination.
  - Un objet MessageProducer est créé appelant la méthode CreateProducer de l'objet Session avec un objet Destination en paramètre.
  - Si aucune destination n'est spécifiée un objet Destination doit être passé à chaque envoi de message (en paramètre de la méthode Send).
  - Un client peut spécifier le mode de délivrance, la priorité et la durée de vie par défaut pour l'ensemble des messages envoyés par un objet MessageProducer. Il peut aussi les spécifier pour chaque message.

Communication en mode Point à point

- Un objet Queue encapsule un nom de file de message du système de messagerie.
- JMS ne définit pas de fonctions pour créer, administrer ou détruire des queues : ces fonctions sont réalisées par des outils d'adminsitration propres à chaque plate-forme JMS.
- Un client utilise un objet QueueConnectionFactory pour créer une connexion active (objet QueueConnection) avec le système de messagerie.
- L'objet QueueConnection permet au client d'ouvrir une ou plusieurs sessions (objets QueueSession) pour envoyer et recevoir des messages.
- L'interface des objets QueueSession fournit les méthodes permettant de créer les objets QueueReceiver, QueueSender et QueueBrowser.

Communication en mode Publish/Subscribe

- Un client JMS publie vers et s'abonne à un nœud dans une hiérarchie de sujets Topics.
- Un objet Topic encapsule un nom de sujet du système de messagerie.
- Un client utilise un objet TopicConnectionFactory pour créer un objet TopicConnection qui représente une connexion active avec le système de messagerie.
- Un objet TopicConnection permet de créer une ou plusieurs sessions (objet TopicSession) pour produire et consommer des messages.
- Un client utilise un objet TopicPublisher pour publier des messages concernant un sujet donné.

Initialisation d'une session Point-to-Point

```
1  // We need a pre-configured ConnectionFactory.
2  // Get it by looking it up using JNDI.
3  Context messaging = new InitialContext();
4  QueueConnectionFactory connectionFactory =
5              (QueueConnectionFactory) messaging.lookup("␣&");
6
7  // Gets Destination objects using JNDI.
8  Queue queue1 = (Queue) messaging.lookup("␣&");
9  Queue queue2 = (Queue) messaging.lookup("␣&");
10
11 // Creates QueueConnection, then use it to create a session.
12 QueueConnection connection = connectionFactory.createQueueConnection();
13 QueueSession session = connection.createQueueSession(
14       false,  // not transacted
15       Session.AUTO_ACKNOWLEDGE);      // acknowledge on receipt
16
17 // Getting a QueueSender and a QueueReceiver.
18 QueueSender sender = session.createSender(queue1);
19 QueueReceiver receiver = session.creatReceiver(queue2);
20
21 // Getting a QueueReceiver with a selector message.
22 Queue queue3 = (Queue) messaging.lookup("␣&");
23 String selector = new String("(name␣=␣'Bull')␣or␣(name␣=␣'IBM'))");
24 QueueReceiver receiver2 = session.creatReceiver(queue3, selector);
25 }
```

Initialisation d'une session Publish/Subscribe

```
1  // We need a pre-configured ConnectionFactory.
2  // Get it by looking it up using JNDI.
3  Context messaging = new InitialContext();
4  TopicConnectionFactory connectionFactory =
5          (TopicConnectionFactory) messaging.lookup("␣&");
6  // Gets Destination object using JNDI.
7  Topic Topic = (Topic) messaging.lookup("␣&");
8
9  // Creates TopicConnection, then use it to create a session.
10 TopicConnection connection = connectionFactory.createTopicConnection();
11 TopicSession session =
12         connection.createTopicSession(false, Session.CLIENT_ACKNOWLEDGE);
13
14 // Getting a TopicPublisher  &
15 TopicPublisher publisher = session.createPublisher(Topic);
16 //  & then a TopicSubscriber.
17 TopicSubscriber subscriber = session.createSubscriber(Topic);
18 Subscriber.setMessageListener(listener);
```

Creation d'un message

```
byte[] data
BytesMessage message = session.createByteMessage();
message.writeBytes(data);
Utilisation d'un TextMessage
StringBuffer data
TextMessage message = session.createTextMessage();
message.setText(data);
```

Envoi de messages

En mode Point-to-Point

```
1  // Sending of all of these message types is done in the same way:
2  sender.send(message);
3  // Receiving of all of these message types is done in the same way. Here is how
       to
4  // receive the next message in the queue.
5  // Note that this call will block indefinitely until a message arrives on the
       queue.
6  StreamMessage message;
7  message = (StreamMessage)receiver.receive();
```

En mode Publish/Subscribe

```
1  // Sending (publishing) of all of these message types is done in the same way:
2  publisher.publish(message);
3  // Receiving of all of these message types is done in the same way. When the
4  // client subscribed to the Topic, it registered a message listener. This
       listener
5  // will be asynchronously notified whenever a message has been published to
6  // the Topic. This is done via the onMessage() method in that listener class.
7  // It is up to the client to process the message there.
8  void onMessage(Message message) throws JMSException {
9          // unpack and handle the messages we receive.
10         ...
11 }
```

Lecture d'un message

```
1  byte[] data;
2  int length;
3  length = message.readBytes(data);
4  TextMessage
5  StringBuffer data;
6  data = message.getText();
7  MapMessage                // Note : l'ordre de lecture des champs est quelconque.
8  String name = message.getString("Name");
9  double value = message.getDouble("Value");
10 long time = message.getLong("Time");
11 StreamMessage   // Note : l'ordre de lecture des champs
12                                // doit etre identique a l'ordre d'ecriture.
13 String name = message.readString();
14 double value = message.readDouble();
15 long time = message.readLong();
16 ObjectMessage
17 obj = message.getObject();
```

Principes de Kafka

- Kafka est un projet Apache initialement développé par LinkedIn, utilisé par exemple par Airbnb et d'autres grandes entreprises
- Aborde le problème de la communication et de l'intégration de composants dans des systèmes à grande échelle
- Permet d'interconnecter les applications avec un environnement d'analyse de données massives
- Conçu en autre, pour traiter des flux de données d'activités en temps réel (logs, collections d'indicateurs, etc.), haut débit, partitionné
- Kafka est écrit en Scala, et il n'est pas conforme aux spécifications JMS

Spécificités de Kafka

- Supporte un grand nombre de consommateurs (scalability issues)
- Supporte des consommateurs temporaires (ad hoc)
- Supporte des consommateurs en mode batch (par exemple 1 fois par jour en demandant un gros volume de données)
- Haute disponibilité (reprise automatique si un broker disparait)
- Haute performance : plus d'1 million événements par seconde sur un petit cluster
- Polymorphe : messaging system (events), activity tracking (analyse de click sur les applications web), collecte de mesures sur des systèmes avec alertes, audit, stream processing)
- Philosophie : le cluster ne s'occupe pas des clients, il stocke et réparti les messages, pas de transformation de données automatique, pas de cryptage, d'autorisation, d'authentification

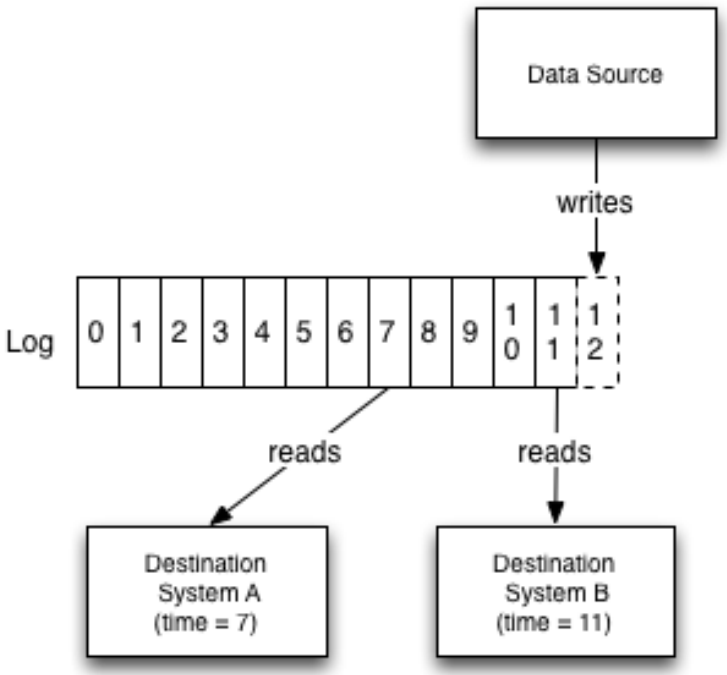
Élements de base

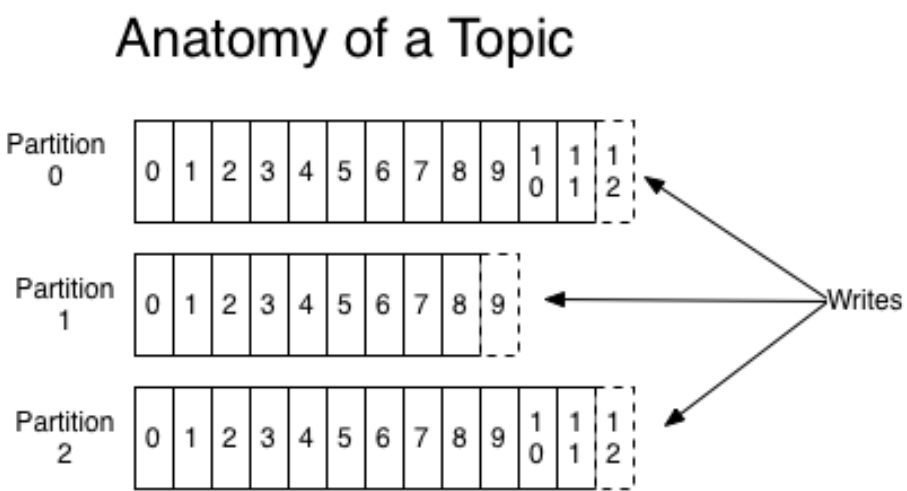
- Les messages sont organisés en Topics
- Mode d'interaction est de type producteurs/consommateurs
- Fonctionne en cluster de brokers (nœuds)
- Les topics volumineux sont répartis dans des partitions sur différents nœuds
- Chaque message à un id, un offset permet de consommer les messages à partir d'un id donné (reprise de données)
- Les partitions sont dupliquées : l'une est le leader (mais les clients n'interagissent pas directement)
- Les figures suivantes sont extraites de la documentation officielle (`http://kafka.apache.org`)

Élements de base

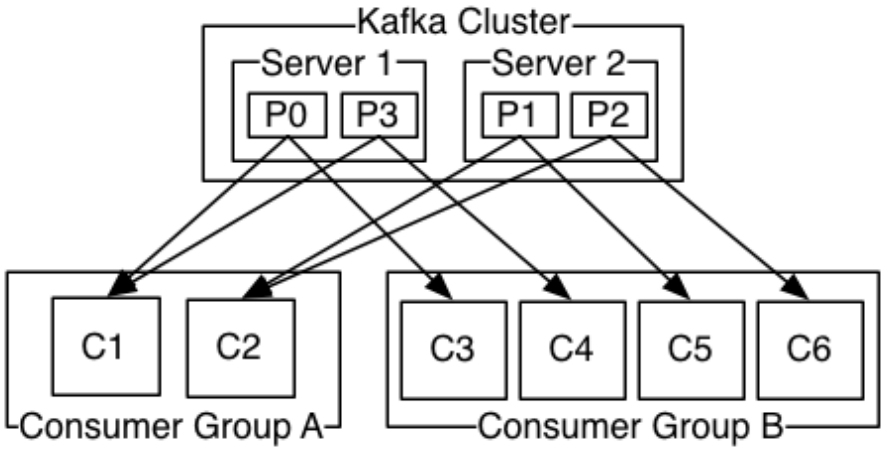
- Un message m est désigné par trois composantes ($Topic$, $Partition$, $offset$)
- Le créateur des topics donne une durée de rétention
- C'est aux consommateurs de se tenir à jour et de gérer leurs lectures
- Les figures suivantes sont extraites de la documentation officielle (`http://kafka.apache.org`)

Élements de base



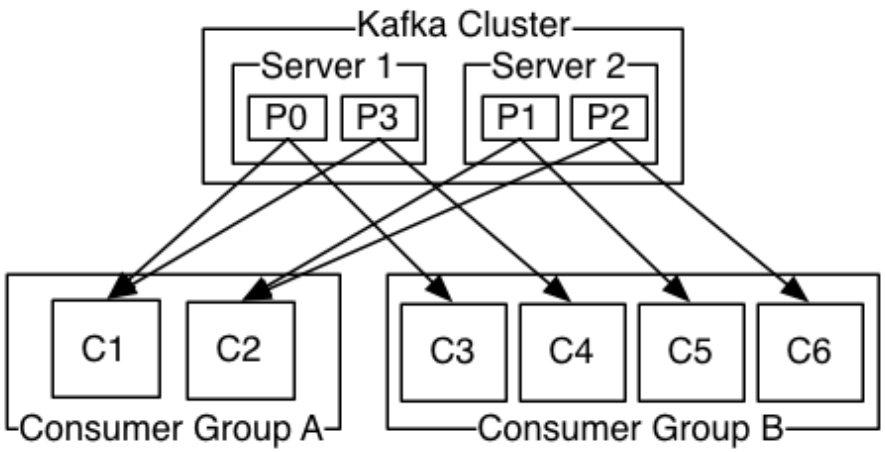


Élements de base et scaling



Élements de base : réplication

Le leader réplique vers les followers. Load balancing entre les partitions Le client détermine à quelle partition il s'adresse



Configuration 2 noeuds, 2 brokers

- Version 0.8.2.2 pour Linux, Scala 2.10, binary version, `http://kafka.apache.org/downloads.html`
- `$ mkdir mykafka`
- `$ cd mykafka`
- `$ cp $HOME/Downloads/kafka_2.10-0.8.2.2.tgz .`
- `$ tar xzfv kafka_2.10-0.8.2.2.tgz`
- Topics et partitions sont écrits dans des *log directories*
  - `$ mkdir kafka-log-1`
  - `$ mkdir kafka-log-2`
- Configurer le serveur kafka
  - `$ cd kafka_2.10-0.8.2.2`
  - `vi config/server.properties`

Configuration du serveur

```
1  # Licensed to the Apache Software Foundation (ASF) under one or more
2  # contributor license agreements.  See the NOTICE file distributed with
3  # this work for additional information regarding copyright ownership.
4  # The ASF licenses this file to You under the Apache License, Version 2.0
5  # (the "License"); you may not use this file except in compliance with
6  # the License.  You may obtain a copy of the License at
7  #
8  #    http://www.apache.org/licenses/LICENSE-2.0
9  #
10 # Unless required by applicable law or agreed to in writing, software
11 # distributed under the License is distributed on an "AS IS" BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13 # See the License for the specific language governing permissions and
14 # limitations under the License.
15 # see kafka.server.KafkaConfig for additional details and defaults
16
17 ############################## Server Basics ##############################
18
19 # The id of the broker. This must be set to a unique integer for each broker.
20 broker.id=0
21
22 ############################## Socket Server Settings
       ##############################
23
24 # The port the socket server listens on
25 port=9092
26
27 # Hostname the broker will bind to. If not set, the server will bind to all
       interfaces
28 #host.name=localhost
29 ...
```

Configuration du serveur

```
...
############################# Log Basics #############################
# A comma seperated list of directories under which to store log files
log.dirs=~/mykafka/kafka-log-1

# The default number of log partitions per topic. More partitions allow greater
# parallelism for consumption, but this will also result in more files across
# the brokers.
num.partitions=1

# The number of threads per data directory to be used for log recovery at
     startup and flushing at shutdown.
# This value is recommended to be increased for installations with data dirs
     located in RAID array.
num.recovery.threads.per.data.dir=1
...
############################# Zookeeper #############################
# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=localhost:2181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000
```

Configuration du serveur

Rôle de ZooKeeper, Kakfa dépend de zookeeper pour :

- maintenir l'état de brokers
- savoir qui est le contrôleur
- connaître le leader
- savoir quels sont les topics etc.
- tous les nœuds doivent référencer le même serveur Zookeeper
- `$ bin/zookeeper-server-start.sh config/zookeeper.properties &`
- `$ bin/kafka-server-start.sh config/server.properties &`
- `$ cp config/server.properties config/server2.properties`
- `$ vi config/server2.properties`
- changer `broker.id`, `port` et `log.dirs`
- `$ bin/kafka-server-start.sh config/server2.properties &`

Création de topic, producer, consumer

- `$ bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic test1 --partitions 2 --replication-factor 2`
- `$ bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic test1`
- `$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test1`
- `$ bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test1`
- les consommateurs font référence à Zookeeper pour déterminer les brokers et y enregistrer leurs offsets temporaires

Producteur consommateur en Java

Producteurs ancienne forme :

- Synchrones : sûrs mais pas forcement rapide
- Asynchrones : rapide, sans notification d'erreurs

Nouvelles version des producteurs :

- utilisables de manière synchrone ou asynchrone avec gestion des erreurs incluent des *futures* et des *call_back*
- multi-thread
- l'usage de la mémoire peut être limité/contrôlé

Dans cet exemple le producteur compte et émet les nombres à un intervalle régulier, spécifié.

Producteur

Exemples de codes données sont ceux de Gwen Shapira (https://github.com/gwenshap/kafka-examples).

```
1  import java.util.concurrent.ExecutionException;
2
3  public interface DemoProducer {
4
5      /**
6       * create configuration for the producer
7       * consult Kafka documentation for exact meaning of each configuration
            parameter
8       */
9      public void configure(String brokerList, String sync);
10
11     /* start the producer */
12     public void start();
13
14     /**
15      * create record and send to Kafka
16      * because the key is null, data will be sent to a random partition.
17      * exact behavior will be different depending on producer implementation
18      */
19     public void produce(String s) throws ExecutionException,
           InterruptedException;
20
21     public void close();
22 }
```

Producteur (old style)

```
1  import kafka.javaapi.producer.Producer;
2  import kafka.producer.KeyedMessage;
3  import kafka.producer.ProducerConfig;
4
5  import java.util.Properties;
6
7  // Simple wrapper to the old scala producer, to make the counting code cleaner
8  public class DemoProducerOld implements DemoProducer{
9      private Properties kafkaProps = new Properties();
10     private Producer<String, String> producer;
11     private ProducerConfig config;
12
13     private String topic;
14
15     public DemoProducerOld(String topic) {
16         this.topic = topic;
17     }
18
19     @Override
20     public void configure(String brokerList, String sync) {
21         kafkaProps.put("metadata.broker.list", brokerList);
22         kafkaProps.put("serializer.class", "kafka.serializer.StringEncoder");
23         kafkaProps.put("request.required.acks", "1");
24         kafkaProps.put("producer.type", sync);
25         kafkaProps.put("send.buffer.bytes","550000");
26         kafkaProps.put("receive.buffer.bytes","550000");
27
28         config = new ProducerConfig(kafkaProps);
29     }
```

Producteur (old style)

```
1  @Override
2      public void start() {
3          producer = new Producer<String, String>(config);
4      }
5
6      @Override
7      public void produce(String s) {
8          KeyedMessage<String, String> message = new KeyedMessage<String, String>(
               topic, null, s);
9          producer.send(message);
10     }
11
12     @Override
13     public void close() {
14         producer.close();
15     }
```

Producteur (old style)

```
1  import java.util.concurrent.ExecutionException;
2
3  public class SimpleCounter {
4
5      private static DemoProducer producer;
6
7      public static void main(String[] args) throws InterruptedException,
           ExecutionException {
8
9          if (args.length == 0) {
10             System.out.println("SimpleCounterOldProducer {broker-list} {topic} {
                   type old/new} {type sync/async} {delay (ms)} {count}");
11             return;
12         }
13
14         /* get arguments */
15         String brokerList = args[0];
16         String topic = args[1];
17         String age = args[2];
18         String sync = args[3];
19         int delay = Integer.parseInt(args[4]);
20         int count = Integer.parseInt(args[5]);
21
22         if (age.equals("old"))
23             producer = new DemoProducerOld(topic);
24         else if (age.equals("new"))
25             producer = new DemoProducerNewJava(topic);
26         else
27             System.out.println("Third argument should be old or new, got " + age
                   );
```

Producteur (old style)

```
1   /* start a producer */
2           producer.configure(brokerList, sync);
3           producer.start();
4
5           long startTime = System.currentTimeMillis();
6           System.out.println("Starting...");
7           producer.produce("Starting...");
8
9           /* produce the numbers */
10          for (int i=0; i < count; i++ ) {
11              producer.produce(Integer.toString(i));
12              Thread.sleep(delay);
13          }
14
15          long endTime = System.currentTimeMillis();
16          System.out.println("... and we are done. This took " + (endTime -
                startTime) + " ms.");
17          producer.produce("... and we are done. This took " + (endTime -
                startTime) + " ms.");
18
19          /* close shop and leave */
20          producer.close();
21          System.exit(0);
22      }
```

Deux mots sur Maven

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
                apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>ProducerExample</groupId>
8      <artifactId>SimpleCounter</artifactId>
9      <version>1.0-SNAPSHOT</version>
10     <dependencies>
11         <dependency>
12             <groupId>org.apache.kafka</groupId>
13             <artifactId>kafka_2.10</artifactId>
14             <version>0.8.2.2</version>
15         </dependency>
16         <dependency>
17             <groupId>org.apache.zookeeper</groupId>
18             <artifactId>zookeeper</artifactId>
19             <version>3.4.6</version>
20         </dependency>
21     </dependencies>
```

Deux mots sur Maven

```
<build>
  <plugins>
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-compiler-plugin</artifactId>
     <configuration>
       <compilerVersion>1.5</compilerVersion>
       <source>1.5</source>
       <target>1.5</target>
     </configuration>
   </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <finalName>uber-${artifactId}-${version}</finalName>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Producteur

Lancer le producteur, ne pas oublier d'avoir lancer un consommateur

- `./run_params.sh localhost:9092 test1 new sync 500 10`

Consommateur

Plusieurs types de consommateurs peuvent être instanciés :

- **High level Consumer :** garder les trace dans Zookeeper de offset lus dans les topics Kafka, loadbalancing automatique
- **Simple Consumer :** API de bas niveau, permet le contrôle de ce qui est lu et des offsets, requiert une expertise
- **New Consumer :** prise en charge automatique des erreurs et du load balancing, gestion manuelle ou automatique des offsets (pas encore dans la version courante)

Consommateur simple

```
1  import kafka.consumer.*;
2  import kafka.javaapi.consumer.ConsumerConnector;
3  import kafka.message.MessageAndMetadata;
4  import kafka.serializer.StringDecoder;
5  import kafka.utils.VerifiableProperties;
6  import org.apache.commons.collections.buffer.CircularFifoBuffer;
7
8  import java.util.HashMap;
9  import java.util.List;
10 import java.util.Map;
11 import java.util.Properties;
12
13 public class SimpleMovingAvgZkConsumer {
14
15     private Properties kafkaProps = new Properties();
16     private ConsumerConnector consumer;
17     private ConsumerConfig config;
18     private KafkaStream<String, String> stream;
19     private String waitTime;
20
21     public static void main(String[] args) {
22         if (args.length == 0) {
23             System.out.println("SimpleMovingAvgZkConsumer␣{zookeeper}␣{group.id}
                   ␣{topic}␣{window-size}␣{wait-time}");
24             return;
25         }
```

Consommateur simple

```
1   String next;
2   int num;
3   SimpleMovingAvgZkConsumer movingAvg = new SimpleMovingAvgZkConsumer();
4   String zkUrl = args[0];
5   String groupId = args[1];
6   String topic = args[2];
7   int window = Integer.parseInt(args[3]);
8   movingAvg.waitTime = args[4];
9   CircularFifoBuffer buffer = new CircularFifoBuffer(window);
10  movingAvg.configure(zkUrl,groupId);
11  movingAvg.start(topic);
12        while ((next = movingAvg.getNextMessage()) != null) {
13            int sum = 0;
14            try {
15                num = Integer.parseInt(next);
16                buffer.add(num);
17            } catch (NumberFormatException e) {
18                // just ignore strings
19            }
20            for (Object o: buffer) {
21                sum += (Integer) o;
22            }
23            if (buffer.size() > 0) {
24                System.out.println("Moving avg is: " + (sum / buffer.size()));
25            }
26            // uncomment if you wish to commit offsets on every message
27            // movingAvg.consumer.commitOffsets();
28        }
29        movingAvg.consumer.shutdown();
30        System.exit(0);
31    }
```

Consommateur simple

```
1   private void configure(String zkUrl, String groupId) {
2         kafkaProps.put("zookeeper.connect", zkUrl);
3         kafkaProps.put("group.id",groupId);
4         kafkaProps.put("auto.commit.interval.ms","1000");
5         kafkaProps.put("auto.offset.reset","largest");
6
7         // un-comment this if you want to commit offsets manually
8         //kafkaProps.put("auto.commit.enable","false");
9
10        // un-comment this if you don't want to wait for data indefinitely
11        kafkaProps.put("consumer.timeout.ms",waitTime);
12
13        config = new ConsumerConfig(kafkaProps);
14    }
```

Consommateur simple

```
1  private void start(String topic) {
2          consumer = Consumer.createJavaConsumerConnector(config);
3          /* We tell Kafka how many threads will read each topic. We have one
                topic and one thread */
4          Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
5          topicCountMap.put(topic,new Integer(1));
6
7          /* We will use a decoder to get Kafka to convert messages to Strings
8          * valid property will be deserializer.encoding with the charset to use.
9          * default is UTF8 which works for us */
10         StringDecoder decoder = new StringDecoder(new VerifiableProperties());
11
12         /* Kafka will give us a list of streams of messages for each topic.
13         In this case, its just one topic with a list of a single stream */
14         stream = consumer.createMessageStreams(topicCountMap, decoder, decoder).
               get(topic).get(0);
15     }
16
17     private String getNextMessage() {
18         ConsumerIterator<String, String> it = stream.iterator();
19         try {
20             return it.next().message();
21         } catch (ConsumerTimeoutException e) {
22             System.out.println("waited␣" + waitTime + "␣and␣no␣messages␣arrived.
                   ");
23             return null;
24         }
25     }
26 }
```

Conclusion : l'eco-système de Kafka

Stream Processing :

- Storm, Samza : stream-processing
- Storm Spout : consomme des messages Kafka et émet des tuples pour Storm
- SparkStreaming : consommateur Kafka pour Spark

Intégration avec Hadoop :

- Flume : collecter et agréger des flux (Kafka Source consumer / Sink producer)
- Camus : initié par LinkedIn, c'est une passerelle Kafka/HDFS pipeline
- Kafka Hadoop Loader

Lambda architecture