JMS ou Java Messaging System

Architectures Réparties en JAVA

- **Architecture répartie (distribuée)** : architecture où toutes les ressources ne se trouvent pas au même endroit ou sur la même machine. On parle également d'informatique distribuée.

Ce concept s'oppose à celui d'architecture centralisée.

- Principe : Les architectures distribuées reposent sur la possibilité d'utiliser des objets qui s'exécutent sur des machines réparties sur le réseau et communiquent par messages au travers du réseau.

Internet est un exemple de réseau distribué puisqu'il ne possède aucun nœud central.

- Plusieurs raisons d'utiliser les applications réparties:
  - Répartir les données géographiquement
  - Permettre la redondance d'une application afin de diminuer le taux de pannes et augmenter la fiabilité
  - Permettre la montée en charge
  - Intégrer des applications existantes qui cohabitent dans l'entreprise

Architectures Réparties en JAVA

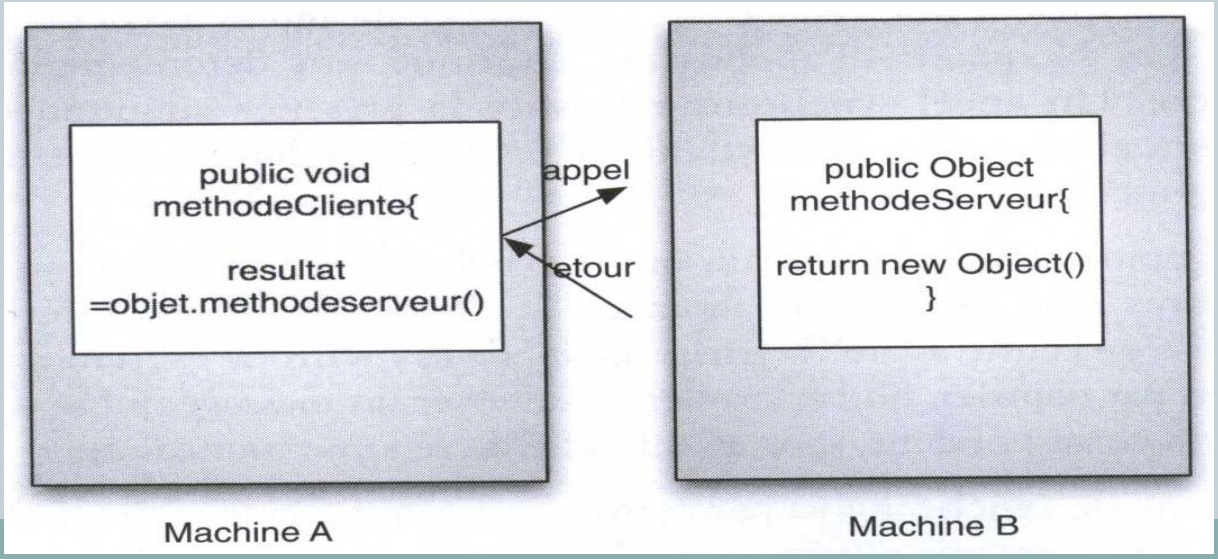
- Les erreurs de conception des architectures distribuées:
  - Réseau non fiable: équipements en pannes, coupures de connexion, transmission de données non fiable,…
  - Latence non nulle: temps de propagation sur le réseau est non négligeable, bande passante faible
  - Réseau non sécurisé: la sécurité globale est directement liée à celle du maillon le plus faible
  - Coût de transport non nul: coût de gestion des ressources
  - Réseau non homogène: protocole IP, mobile sur bluetooth… RMI, IIOP de CORBA, JMS…

Architectures Réparties en JAVA

- Problématique des systèmes distribués: la communication entre applications.

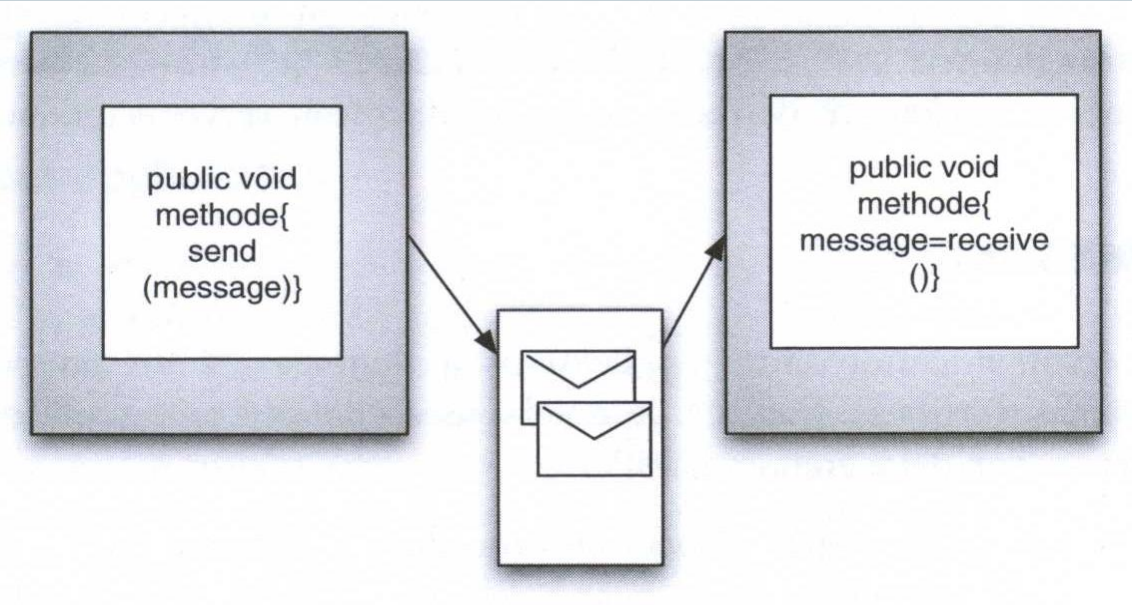
Deux grandes familles:

1. **Technologies d'appel de procédure à distance** (RPC, Remote Procedure Call): le principe est l'invocation d'un service (procédure ou méthode d'un objet) situé sur une machine distante. Les standards sont tels que CORBA, RMI, DCOM (Microsoft), SOAP (Web Services)



Architectures Réparties en JAVA

2. Technologies d'échange de messages: les messages sont véhiculés par l'infrastructure MOM (Message Oriented Middleware). Ces messages sont génériques: ils peuvent représenter tout type de contenu (binaire: image, objets sérialisés. Texte:documents XML)



Architectures Réparties en JAVA

- Mode synchrone et asynchrone :
  - Synchrone: l'application cliente est bloquée en attente d'une réponse à une requête au serveur. Nécessite la présence simultanée du client et du serveur (téléphone).
  - Asynchrone: l'appel n'est pas bloquant; l'application cliente continue son déroulement sans attendre la réponse. Ne nécessite pas la présence simultanée du client et du serveur (courrier).

- Comparaison:
  - Le mode synchrone permet de s'assurer que le message a été envoyé et que la réponse est reçu : plus de fiabilité
  - Le mode asynchrone permet de libérer du temps pour d'autres actions afin de paralléliser les tâches: plus de performance

Architectures Réparties en JAVA

MOM ou *Message-Oriented Middleware :*

- Un MOM est une plateforme logicielle fournissant un moyen de communication entre divers applications

- Le principe est qu'une application **ne communique pas directement avec l'autre**, mais dépose son "message" dans le MOM. L'autre application de son coté, viendra simplement vérifier l'arrivée de messages sur le MOM

- Mode de communication :

Asynchrone

Comparaison MOM/RPC

Caractéristique	MOM	RPC
Métaphore	Courier	Téléphone (sans répondeur)
Relation temporelle entre client et serveur	**Asynchrone**	**Synchrone**
Nature du dialogue	File d'attente	Requête - Réponse
Etat opérationnel du serveur	Pas nécessaire	Obligatoire
Support des transactions	Dépend du produit	Dépend du produit (nécessité d'un RPC transactionnel)
Filtrage des messages	Possible	Non
Performances	Lent en cas de sécurisation des messages par écriture sur disque	Plus efficace que MOM car pas de sauvegarde

Architectures Réparties en JAVA

- **JMS** (pour Java Messaging System) est l'implémentation Java de ce qui est appelé MOM
- JMS défini un ensemble d'interfaces pour créer des applications de messageries en Java : **javax.jms.***

- **Caractéristiques de JMS :**
  - Faible couplage : Pour communiquer, la seule chose qui est nécessaire à deux applications, c'est d'avoir un message compréhensible par les deux cotés, et un serveur JMS commun. Pas besoin de connaitre l'adresse de l'autre, pas besoin de synchronisation.

  Intérêt : Si l'application réceptrice est modifiée, déplacée ou remplacée, cela reste complètement transparent du point de vue de l'émetteur

  - Asynchrone : JMS est un protocole asynchrone, ce qui veut dire que l'application émettrice émet son message, et continue son traitement sans attendre que le récepteur confirme l'arrivée du message. De son coté, le récepteur récupère les messages quand il le souhaite.

  Intérêt : Si l'application réceptrice effectue une tâche longue, l'émetteur n'est pas bloqué

Architectures Réparties en JAVA

- Persistant : JMS propose un mode persistant, c'est à dire que les messages envoyés par un émetteur et non encore consommé par un client sont stocké de manière persistance (disque, base de données, …). Ce qui assure une garantie de livraison du message, même si le serveur JMS devait être arrêté (maintenance, plantage, …).

Intérêt : Assurance que le message ne sera pas perdu et arrivera bien au client.

- Transactionnel : Une communication JMS peut être comprise dans une transaction (via JTA). Les messages ne sont effectifs qu'à la validation d'une transaction. Tous les messages produits sont envoyés à la validation; en cas d'abandon de la transaction les messages produits sont abandonnés.

Architectures Réparties en JAVA

- Dans une architecture de communication, il y a plusieurs intervenants et composants qui interviennent :
  - Le provider JMS
  - Le client JMS (émetteur et récepteur de messages)
  - Le messa

- JMS est l'A rète de celle-ci pour pou

- Un Provide

Voici une liste

Implémentation	Licence
Apache ActiveMQ	OpenSource - Apache 2.0 License
OpenJMS	OpenSource
JBoss Messaging	OpenSource
Joram	OpenSource - LGPL license
WebShere MQ - IBM	Commercial
Sonic MQ	Commercial
Oracle Advanced Queueing	Commercial
TIBCO Enterprise Message Service	Commercial
Sun Java System Message Queue	Commercial

Architectures Réparties en JAVA

- Les clients JMS sont les applications qui utilise JMS pour communiquer, elle sont de deux types :
  - **Producteur** : Client qui crée et envoie des messages
  - **Consommateur** : Client qui reçoit des messages

  Un client peut être producteur et consommateur à la fois.

Architectures Réparties en JAVA

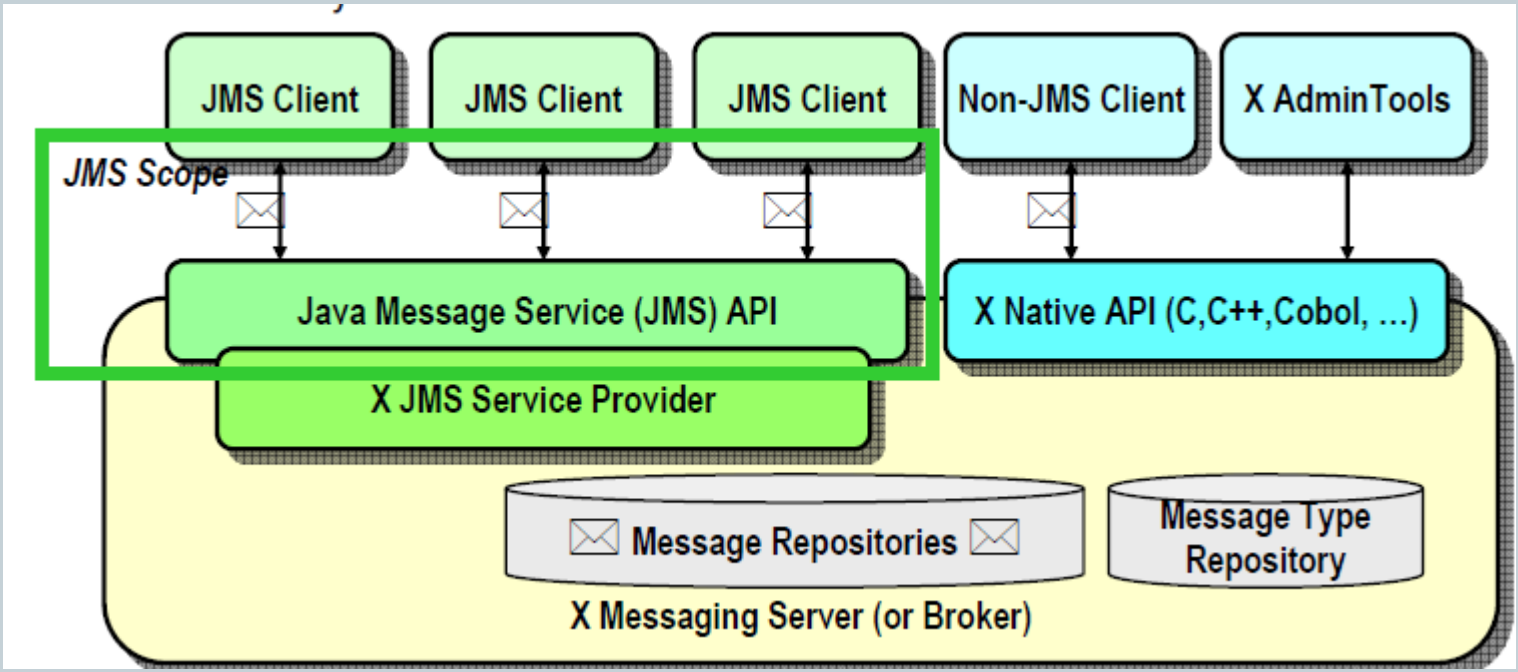
Modèles de messagerie (*messaging*) :

- **Point à Point (*Point-to-Point*)** : concept de ***Queue***, *une file d'attente de messages*
- **Publication-Souscription (*Publish-Subscribe*)** : concept de ***Topic***, *un sujet auquel s'abonne un ou plusieurs Subscribers*

Architecture JMS:

- le client utilise les classes du package **javax.jmx** pour l'envoi et la réception de messages
- le serveur implémante un JMS Service Provider

Architectures Réparties en JAVA

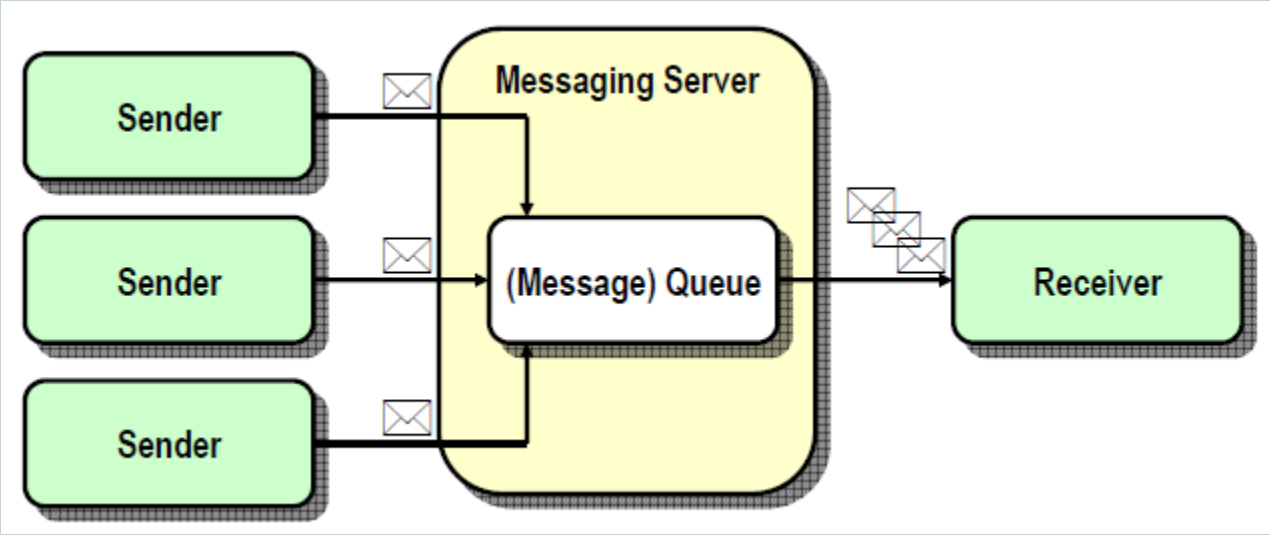


Un *client non-JMS* est une application envoyant et/ou recevant des messages en communiquant avec le JMS Provider selon son protocole particulier, soit en direct, soit par l'intermédiaire des fonctions d'une API. Cette application n'est pas écrite en Java.

Architectures Réparties en JAVA

Modèle Point à Point (*Point-to-Point)*

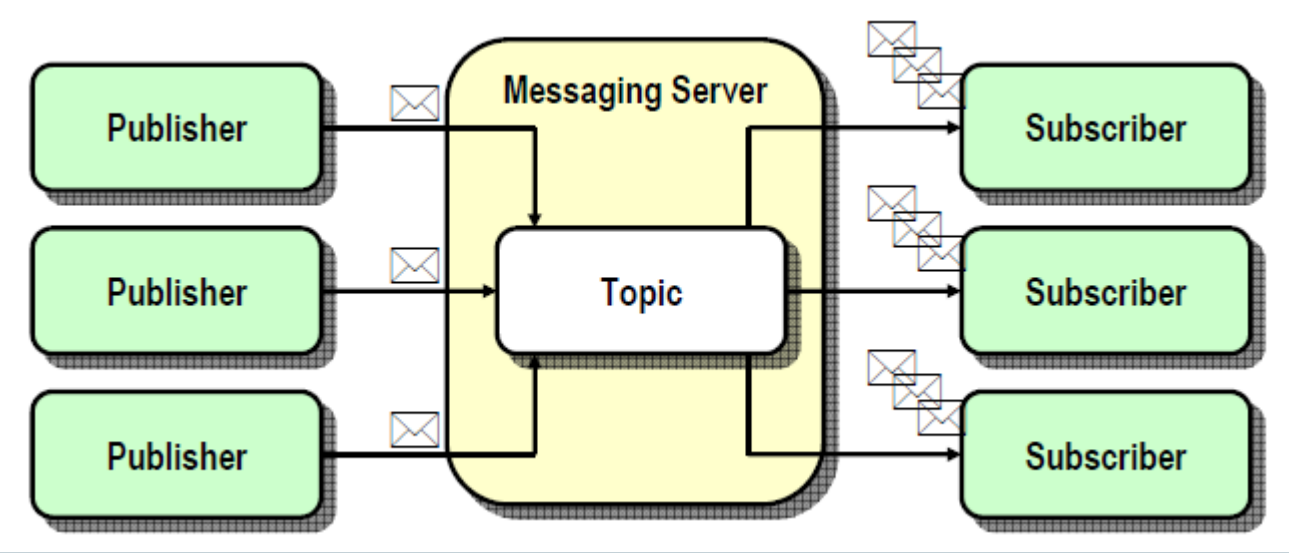
- Concept de **Queue** :
  - Un (ou plusieurs) ***Sender*** *envoie (produit) un message à* une *Queue spécifique (i.e. file d'attente de messages)*
  - Un et un seul ***receiver*** *reçoit (consomme) les messages* de la *Queue (un message à un seul consommateur)*
  - *La queue retient les messages jusqu'à ce qu'ils soient consommés ou expirés*



Architectures Réparties en JAVA

Modèle Publication-Souscription (*Publish-Subscribe)*

- Concept de **Topic** (centre d intérêt) :
  - Un (ou plusieurs) ***publishers*** *envoie un message à un Topic (chaque message peut avoir plusieurs consommateurs)*
  - Tous les ***subscribers*** *(abonnés) de ce Topic reçoivent le message*
  - *Un client qui s'abonne à un topic ne peut recevoir que les messages envoyés après son abonnement*



Architectures Réparties en JAVA

- Classes de l'API JMS :

JMS Parent	Description	Modèle PTP	Modèle Pub/Sub
ConnectionFactory	*un objet administré par le provider pour créer une Connection*	QueueConnectionFactory	TopicConnectionFactory
Connection	*une connexion active vers un JMS provider*	QueueConnection	TopicConnection
Destination	*encapsule l'identité d'une destination*	Queue	Topic
Session	*contexte (mono-thread) pour l'émission et la réception de messages*	QueueSession	TopicSession
MessageProducer	*objet pour la production (l'envoi) de messages vers une Destination (créé par la Session)*	QueueSender	TopicPublisher
MessageConsumer	*objet pour la consommation (la réception) de messages d'une destination (créé par la Session)*	QueueReceiver, QueueBrowser	TopicSubscriber

Architectures Réparties en JAVA

Fonctionnement d ’un client JMS :

- Phase d ’initialisation (setup) :
  - trouver l ’objet ConnectionFactory par JNDI
  - créer une Connexion JMS
  - trouver un (ou plusieurs) objet Destination par JNDI
  - créer une (ou plusieurs) Session avec la Connexion JMS
  - créer le(s) MessageProducer ou/et MessageConsumer avec la Session et la (les) Destination
  - Créer l’objet récepteur/envoyeur de messages
  - demander à la Connexion de démarrer la livraison des messages
- Phase de consommation/production de messages :
  - créer des messages et les envoyer
  - recevoir des messages et les traiter

Architectures Réparties en JAVA

Les objets administrés par le Provider :

- ConnectionFactory : javax.jms.ConnectionFactory
  - point d 'accès à un serveur MOM
  - accessible par JNDI et enregistré par l 'administrateur du serveur MOM : sous OpenJMS le nom est *"**ConnectionFactory**"*.

- Destination : javax.jms. Destination
  - Queue ou Topic administré par le serveur MOM
  - accessible par JNDI et enregistré par l 'administrateur du serveur MOM : sous OpenJMS le nom est "**queue1**" ou "**topic1**"

Architectures Réparties en JAVA

Les objets administrés par le Provider (suite):

- Connection : javax.jms.Connection
  - encapsule la liaison TCP/IP avec le Provider JMS
  - authentifie le client et spécifie un identifiant unique de client
  - crée les Sessions et fournit le ConnectionMetaData (qui fournit des informations concernant l'objet connexion: version du provider JMS, nom du provider JMS,version de l'API JMS,liste des propriétés du message)
    - QueueConnectionFactory. (**createQueueConnection**() / **createQueueConnection**(String userName, String password))
    - TopicConnectionFactory.(**createTopicConnection**()/ **createTopicConnection**(String userName, String password))

    Créer une connexion avec l'utilisateur par défaut/utilisateur spécifié.

  - Cycle de vie :
    - démarrage start() : procède à la livraison des Messages
    - pause stop() : interrompt la livraison des Messages entrants, n'affecte pas l 'envoi
    - reprise start()
    - fermeture close() : libération des ressources

Architectures Réparties en JAVA

Les objets administrés par le Provider (suite):

- Session : javax.jms.Session
  - contexte (mono-thread) pour l ’émission et la réception de messages
  - définit les numéros de série des messages (consommés et produits)
    - QueueSession **QueueConnection.createQueueSession**(boolean transacted, int acknowledgeMode)
    - TopicSession **TopicConnection.createTopicSession**(boolean transacted, int acknowledgeMode)

Paramètres:

- transacted : indique si la session est transactionnelle
- acknowledgeMode : indique si le consomateur ou le client va confirmer la réception des messages. Valeurs : Session.AUTO_ACKNOWLEDGE, Session.CLIENT_ACKNOWLEDGE, et Session.DUPS_OK_ACKNOWLEDGE (envoyer plusieurs fois le message à une même destination).

Pour confirmer la réception, le client appèle la méthode : Message. **acknowledge**()

Architectures Réparties en JAVA

javax.jms.MessagerConsumer :

- représente l 'objet consommateur des messages
  - fabriqué par la Session
    - QueueReceiver QueueSession.**createReceiver**(Queue queue, String messageSelector)
    - TopicSubscriber TopicSession.**createSubscriber**(Topic topic, String messageSelector, boolean)
  - Remarque : il peut y avoir plusieurs MessageConsumers sur une Session pour une même Destination
  - Sous-interfaces : QueueReceiver, TopicSubscriber

- Réception **Synchrone** des messages
  - le client se met en attente du prochain message (méthodes de **MessageConsumer**)
    - Message **receive()**, Message **receive(long timeout)** (msg arrivant dans une durée fixée), Message **receiveNoWait()** (si un msg est immédiatement disponible)

- Réception **Asynchrone** des messages
  - le client crée un objet qui implémente l 'interface **MessageListener** et positionne cet objet à l'écoute des messages (MessageConsumer. **setMessageListener**())
    - void setMessageListener(MessageListener) / MessageListener getMessageListener()
  - quand un message arrive, la méthode void **onMessage(Message)** de l 'objet MessageListener est invoqué par le serveur
  - La session n'utilise qu'une seule thread pour exécuter séquentiellement tous les MessageListeners

Architectures Réparties en JAVA

javax.jms. MessagerProducer :

- représente l ’objet producteur des messages
  - fabriqué par la Session
    - QueueSender QueueSession.**createSender**(Queue queue)
    - TopicPublisher TopicSession.**createPublisher**(Topic topic)
  - Remarque : il peut y avoir plusieurs MessageProducers sur une Session pour une même Destination
  - Sous-interfaces : QueueSender, TopicPublisher
- Production des messages
  - void QueueSender.**send**(Message)
  - void TopicPublisher.**publish**(Message)

Architectures Réparties en JAVA

Le modèle Point à Point *Point-To-Point PTP :*

JMS Parent	Modèle PTP
ConnectionFactory	QueueConnectionFactory
Connection	QueueConnection
Destination	Queue, TemporaryQueue
Session	QueueSession
MessageProducer	QueueSender
MessageConsumer	QueueReceiver
--	QueueBrowser

Remarques :

- plusieurs sessions peuvent se partager une même queue : JMS ne spécifie pas comment le provider réparti la même queue entre les QueueReceiver de ces sessions.
- Les messages non sélectionnés restent dans la queue : l 'ordre de réception n 'est plus alors l'ordre de production
- **QueueBrowser** permet de consulter les messages de la Queue sans les retirer en les énumérant (méthode Enumeration getEnumeration())

Architectures Réparties en JAVA

Le modèle Publication-Souscription *Publish-Subscribe PubSub*

JMS Parent	Modèle Pub/Sub
ConnectionFactory	TopicConnectionFactory
Connection	TopicConnection
Destination	Topic
Session	TopicSession
MessageProducer	TopicPublisher
MessageConsumer	TopicSubscriber

Remarque :

- le terme « publier » correspond à « produire »
- le terme « souscrire » correspond à « consommer »

Propriétés JNDI :

- Fichier **jndi.properties** localisé dans le classpath de l'application :

```
java.naming.factory.initial=org.exolab.jms.jndi.InitialContextFactory
java.naming.provider.url=tcp://localhost:3035
```

- Le code pour construire le JNDI **InitialContext** est :

```
import javax.naming.Context;
import javax.naming.InitialContext;
// ...
Context context = new InitialContext();
```

Architectures Réparties en JAVA

- **Exemple : Point à Point. La partie commune :**

```
class JMSQueueTest {
static Context messaging;                    static QueueConnectionFactory queueConnectionFactory; static Queue queue;
static QueueConnection queueConnection;              static QueueSession queueSession;
static QueueSender queueSender;                      static QueueReceiver queueReceiver;
static void setup(String queueConnectionFactoryName, String queueName){
  messaging = new InitialContext();
  queueConnectionFactory = (QueueConnectionFactory)messaging.lookup(queueConnectionFactoryName);
  queue = (Queue) messaging.lookup(queueName);
  queueConnection = queueConnectionFactory.createQueueConnection();
  queueSession = queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
}
static void setupSender(String queueConnectionFactoryName, String queueName){
  setup(queueConnectionFactoryName, queueName);
  queueSender = queueSession.createSender(queue);
  queueConnection.start();
}
static void setupReceiver(String queueConnectionFactoryName, String queueName){
  setup(queueConnectionFactoryName, queueName);
  queueReceiver = queueSession.createReceiver(queue);
  queueConnection.start();
} }
```

Architectures Réparties en JAVA

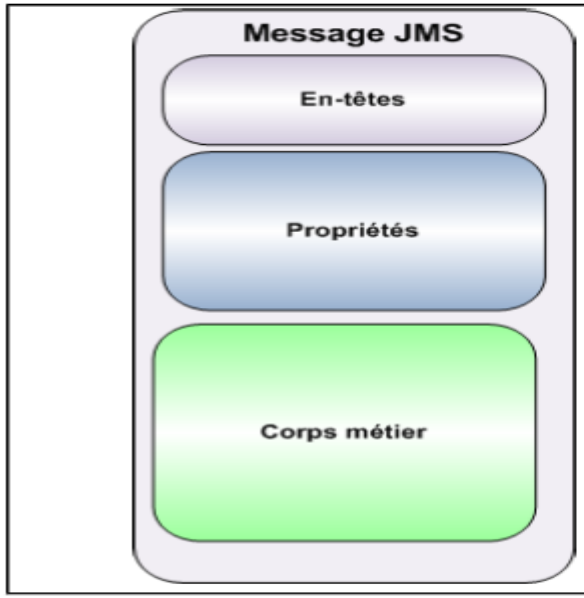
- **Exemple : Publication Souscription. La partie commune :**

```
class JMSTopicTest {
static Context messaging;                    static TopicConnectionFactory topicConnectionFactory; static Topic topic;
static TopicConnection  topicConnection;              static TopicSession topicSession;
static TopicPublisher topicPublisher ;                static TopicSubscriber topicSubscriber ;
static void setup(String  topicConnectionFactoryName, String  topicName){
  messaging = new InitialContext();
  topicConnectionFactory = (TopicConnectionFactory)messaging.lookup(topicConnectionFactoryName);
  topic = (Topic) messaging.lookup(topicName);
  topicConnection = topicConnectionFactory.createTopicConnection();
  topicSession = topicConnection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
}
static void setupPublisher(String topicConnectionFactoryName, String topicName){
  setup(topicConnectionFactoryName, topicName);
  topicPublisher = topicSession.createPublisher(topic);
  topicConnection.start();
}
static void setupSubscriber(String topicConnectionFactoryName, String topicName){
  setup(topicConnectionFactoryName, topicName);
  topicSubscriber = topicSession.createSubscriber(topic);
  topicConnection.start();
} }
```

Architectures Réparties en JAVA

Le modèle de Messages JMS *L 'interface javax.jms.Message*

- Un message JMS est la structure qui transite dans une communication JMS.
- Modèle commun à tous les produits
  - Supporte l'hétérogénéïté des clients (non JMS, langage, plate-forme, …)
  - Supporte les objets Java et les documents XML
- L 'interface **javax.jms.Message**
  - sous-interfaces: BytesMessage, MapMessage, ObjectMessage, StreamMessage, TextMessage
- Il est composé de plusieurs parties :



En-têtes : Divers informations d'ordre technique du message : identifiant, identifiant de correlation, priorité, durée de validité
Propriétés : Information de type "clé -> valeurs" qui peuvent être ajouter par l'application pour apporter des précision du le message
Corps du message : Données métiers du message

Architectures Réparties en JAVA

- Les en-têtes des messages : champs communs aux Providers (Produit) et obligatoires, destinés au routage et l'identification du message

- Propriété (*property)*
  - champs supplémentaires
    - propriétés standards : équivalent aux champs d'entête optionnels
    - propriétés spécifiques au (produit) provider
    - propriétés applicatives : peuvent répliquer le contenu d'une valeur du corps
- Corps (*body)*

Architectures Réparties en JAVA

Nom des champs d 'entête :

JMSDestination : destination du message **Queue ou Topic**
JMSDeliveryMode : sûreté de distribution
NON_PERSISTENT : faible **PERSISTANT par défaut**
PERSISTENT : garantie supplémentaire qu 'un message ne soit pas perdu
JMSExpiration : calculé à partir du TTL (Time To Live) depuis la prise en charge
JMSMessageID : identifiant du Message founit par le provider
JMSTimestamp : date de prise en charge du message par le provider
JMSCorrelationID : identifiant d 'un lien avec un autre Message (Request/Reply)
JMSReplyTo : identifiant de la Destination pour les réponses
JMSType : identifient du type de message dans le Message Type Repository
JMSRedelivered : le récepteur n 'acquitte pas immédiatement la réception (Transaction)
JMSPriority : priorité (de 0 à 9) du message

le message est stocké, et restauré en cas de panne du JMS provider

4 par défaut

- Time-to-live : MessageProducer. **setTimeToLive**(millisecondes). 0 c'est illimité. Par défaut c'est 4.
- Consultation/Modification : méthodes setXXX() / getXXX() où XXX est le nom du champs d'entête

Architectures Réparties en JAVA

Nom des propriétés :

- propriétés standards : nom préfixé par JMSX
  - correspondent à des champs d 'entête optionnels

JMSXUserID, JMSXAppID : identité de l 'utilisateur et de l 'application
JMSXGroupID, JMSXGroupSeq : groupe de messages et son numéro de séq
JMSXProducerTXID, JMSXConsumerTXID : identifiants de transaction
JMSXRcvTimestamp : date de réception
JMSXState : 1(waiting), 2(ready), 3(expired), 4(retained)
JMSXDeliveryCount : The number of message delivery attemps

- propriétés spécifiques : nom préfixé par JMS_VendorName
- propriétés applicatives :
  - servent au filtrage
  - servent comme données complémentaires au corps
    - exemple : la date

Architectures Réparties en JAVA

- Enumération des propriétés
  - Enumeration Message.**getPropertyNames()** /
    Enumeration ConnectionMetaData.**getJMSXPropertyNames()** : énumère les propriétés (JMSX) du message
  - boolean **propertyExists**(String) : teste l 'existence d 'une propriété
  - Ces méthodes lancent des **JMSException** si le provider JMS provider échoue.

- Valeur des propriétés
  - Type : boolean, byte, short, int, long, float, double, String
    - méthodes void **setIntProperty(String,int)**,
      void **setBooleanProperty(String,boolean)**, …
    - méthodes int **getIntProperty(String)**, boolean **getBooleanProperty(String)**, …
      - L 'appel à getXXXProperty() retourne null si la propriété n 'existe pas
      - Les propriétés d 'un message reçu sont en lecture seule sinon l 'exception MessageNotWriteableException est levée
  - Type Objet correspondant : Boolean,Byte,Short,Integer,Long,Float,Double,String
    - méthode void setObjectProperty(String,Object)
    - méthode Object getObjectProperty(String)
  - Réinitialisation des propriétés : void **clearProperties()**
  - Ces méthodes appartiennent à l'interface **javax.jms.Message**

Architectures Réparties en JAVA

Le modèle de Messages JMS *Le corps (body) du message*

- Sous interfaces de javax.jms.Message qui ajoute un élément "body"
  - **javax.jms.TextMessage**
    - contient un String qui peut être un document XML, un message SOAP…
      - String getText()/ setText(String)
  - **javax.jms.MapMessage**
    - contient un ensemble de paires name-value (le nom est de type String)
      - int getInt(String name) / setInt(String name, int value), …
      - Object getObject(String)/ setObject(String,Object) : les objets doivent être de type primitif ou un objet tableau de bytes (ex. à partir d'un fichier)
      - Enumeration getMapNames()
  - **javax.jms.ObjectMessage**
    - contient un objet Java sérialisé : Serializable getObject()/ void setObject(Object)
    - L'objet qui doit être envoyé par le message doit supporter l'interface **Serializable**.

- **Le modèle de Messages JMS *Le corps (body) du message (suite)***

  - **javax.jms.BytesMessage**
    - contient un tableau de bytes
      - int readInt()(4bytes)/writeInt(int), …, int readBytes(Byte[])/writeBytes(Byte[]),
      - Object readObject()/ writeObject(Object), reset()

  - **javax.jms.StreamMessage**
    - contient un flux de données (Objets *Java primitives) qui s'écrit et se lit* séquentiellement
      - int readInt()/writeInt(int), …, String readString()/writeString(String)
      - Un appel de méthode non adaptée au type cible lancera l'exception **MessageFormatException**

Architectures Réparties en JAVA

- Méthodes :
  - Message.clearBody() réinitialise le corps (les entêtes et propriétés ne sont pas réinitialisés)
  - clearProperties() réinitialise les propriétés
  - getJMSMessageID() : donne l'ID du message.

- Remarque :
  - le corps d 'un message reçu est en lecture seule : appeler une des méthodes setXXX() lancera l'exception **MessageNotWriteableException**.

- Pour la création de message :

  Méthodes de la classe Session (classe mère de QueueSession et TopicSession)
  - Message **createMessage**()
  - BytesMessage **createBytesMessage**()
  - MapMessage **createMapMessage**()
  - ObjectMessage **createObjectMessage**()
  - ObjectMessage **createObjectMessage**(java.io.Serializable object)
  - StreamMessage **createStreamMessage**()
  - TextMessage **createTextMessage**()
  - TextMessage **createTextMessage**(java.lang.String text)
  - MessageListener **getMessageListener**()

Architectures Réparties en JAVA

- Exemple de corps de message : envoi de message

```
String textstr = "<?xml version=\"1.0\"?><!DOCTYPE person SYSTEM \"person.dtd\">"
                +"<person><firstname>Joe</firstname><lastname>Smith</lastname>"
                +"<salary value=\"10000.0\"><age value=\"38\"></person>";
// l 'API javax.xml est préférable pour construire le document XML
TextMessage textmsg = session.createTextMessage();
textmsg.setText(textstr);

MapMessage mapmsg = session.createMapMessage();
mapmsg.setString("Firstname", "Joe");      mapmsg.setString("Lastname", "Smith");
mapmsg.setDouble("Salary", 10000.0);       mapmsg.setLong("Age", 38);

Person object = new Person("Joe","Smith", 37); object.setAge(38); object.setSalary(10000.0);
ObjectMessage objectmsg = session.createObjectMessage();
objectmsg.setObject(object); // Person doit implémeter java.io.Serializable

Byte[] bytesarray = { 'J','o','e',' ','S','m','i','t','h'};
BytesMessage bytesmsg = session.createByteMessage();
bytesmsg.writeBytes(bytesarray); bytesmsg.writeInt(38); bytesmsg.writeDouble(10000.0);

StreamMessage streammsg = session.createStreamMessage();
streammsg.writeString("Joe");              streammsg.writeString("Smith");
streammsg.writeLong(38);                   streammsg.writeFloat(10000.0);
```

Architectures Réparties en JAVA

- Exemple de corps de message : réception de message

  Message receivedmsg = QueueReceiver .**receive()**;

  (ou bien : Message receivedmsg = TopicSubscriber.**receive()**;)

```
TextMessage textmsg = (TextMessage)receivedmsg;
String textstr=textmsg.getText(textstr);  System.out.println(textstr);
            // le document XML peut être parsé

MapMessage mapmsg = (MapMessage)receivedmsg;
String firstname= mapmsg.getString("Firstname");
String lastname= mapmsg.getString("Lastname");
long age= mapmsg.getLong("Age"); double salary= mapmsg.getDouble("Salary");
System.out.println(firstname +  " " + lastname +  " " + age +  " " + salary);

ObjectMessage objectmsg = (ObjectMessage)receivedmsg;
Person object = (Person)objectmsg.getObject();
System.out.println(object.toString());

BytesMessage bytesmsg = (BytesMessage)receivedmsg;
Byte[] bytesarray ;
int length=bytesmsg.readBytes(bytesarray);
long age= bytesmsg.readLong(); double salary= bytesmsg.readDouble();

StreamMessage streammsg = (StreamMessage)receivedmsg;
String firstname= streammsg.readString(); String lastname= streammsg.readString();
long age= streammsg.readLong(); double salary= streammsg.readDouble();
System.out.println(firstname +  " " + lastname +  " " + age +  " " + salary);
```

Architectures Réparties en JAVA

- **Exemple : Point à Point. La partie commune :**

```
class JMSQueueTest {
static Context messaging;                    static QueueConnectionFactory queueConnectionFactory; static Queue queue;
static QueueConnection queueConnection;              static QueueSession queueSession;
static QueueSender queueSender;                      static QueueReceiver queueReceiver;
static void setup(String queueConnectionFactoryName, String queueName){
  messaging = new InitialContext();
  queueConnectionFactory = (QueueConnectionFactory)messaging.lookup(queueConnectionFactoryName);
  queue = (Queue) messaging.lookup(queueName);
  queueConnection = queueConnectionFactory.createQueueConnection();
  queueSession = queueConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
}
static void setupSender(String queueConnectionFactoryName, String queueName){
  setup(queueConnectionFactoryName, queueName);
  queueSender = queueSession.createSender(queue);
  queueConnection.start();
}
static void setupReceiver(String queueConnectionFactoryName, String queueName){
  setup(queueConnectionFactoryName, queueName);
  queueReceiver = queueSession.createReceiver(queue);
  queueConnection.start();
} }
```

Architectures Réparties en JAVA

- **Exemple : Point à Point. *Sender :***

```
// java JMSQueueSenderTest testServer testQueue Hello 10.0 10
public class JMSQueueSenderTest  extend JMSQueueTest {

static void send(String stringValue,double doubleValue, long doubleValue, boolean booleanProperty) {
  MapMessage message = session.createMapMessage();
  message.setObject("stringValue", stringValue);
  message.setDouble("doubleValue", doubleValue);
  message.setLong("longValue", longValue);
  message.setBooleanProperty("exit", booleanProperty);
 queueSender.send(message);
}

public static void main(String args[]) {
  setupSender(argv[0],argv[1]);
  long cpt=Long.parseLong(argv[4]);
  while(--cpt>0) {
       send(argv[2]+cpt,Double.parseDouble(agv[3]).valueDouble()+cpt,cpt, false);
  }
  send(argv[2]+cpt,agv[3]+cpt,cpt, true);
  close();
}}
```

Architectures Réparties en JAVA

- **Exemple : Point à Point. *Receiver synchrone:***

```
// java JMSQueueSyncReceiverTest testServer testQueue
public class JMSQueueSyncReceiverTest  extend JMSQueueTest {
  static boolean exit=false;
  static void handleMessage(MapMessage message) {
    String stringValue=message.getString("stringValue");          System.out.print(" stringValue="+stringValue);
    double doubleValue=message.getDouble("doubleValue");          System.out.print(" doubleValue="+doubleValue);
    long longValue=((Long)message.getObject("longValue")).longValue(); System.out.println(" longValue="+longValue);
    exit = message.getBooleanProperty("exit");
  }
  static void syncReceive() {
    MapMessage message;
    while(!exit) {
      message= (MapMessage)queueReceiver.receive();
      handleMessage(message);
    }
  }
  public static void main(String args[]) {
    setupReceiver(argv[0],argv[1]);
    syncReceiver();
    close();
  } }
```

Architectures Réparties en JAVA

- **Exemple : Point à Point. *Receiver asynchrone:***

```
// java JMSQueueAsyncReceiverTest testServer testQueue
public class MyListener implements javax.jms.MessageListener {
void onMessage(Message message) { JMSQueueAsyncReceiverTest.handleMessage((MapMessage)message); }
}
public class JMSQueueAsyncReceiverTest  extend JMSQueueTest {
 static boolean exit=false;
 static void handleMessage(MapMessage message) {
  String stringValue=message.getString("stringValue");          System.out.print(" stringValue="+stringValue);
  double doubleValue=message.getDouble("doubleValue");          System.out.print(" doubleValue="+doubleValue);
  long longValue=((Long)message.getObject("longValue")).longValue(); System.out.println(" longValue="+longValue);
  exit = message.getBooleanProperty("exit");
 }
 static void asyncReceive() {
  queueReceiver.setMessageListener(new MyListener());
  while(!exit) { /* doSomething */ }
 }
 public static void main(String args[]) {
  setupReceiver(argv[0],argv[1]);
  asyncReceiver();
  close();
 } }
```

Architectures Réparties en JAVA

- JMS ne définit pas les fonctionnalités suivantes qui restent dépendanets du provider :
  - une sécurisation des messages ;
  - le balancement de charges ;
  - la tolérances aux fautes ;
  - une administration du fonctionnement du provider ;
  - la possibilité de définir de nouveaux types de messages ;
  - la possibilité de "réveiller un client" quand des messages arrivent.
  - ....