Message-Driven Beans

Michel Buffa (buffa @unice.fr), UNSA 2002

Message-Driven Beans

- Nouveauté EJB 2.0,
- *Messaging* = moyen de communication léger, comparé à RMI-IIOP,
- Pratique dans de nombreux cas,
- Message-Driven beans = beans accessibles par *messaging* asynchrone.

Message-Driven Beans : motivation

- Performance
  - Un client RMI-IIOP *attend* pendant que le serveur effectue le traitement d'une requête,
- Fiabilité
  - Lorsqu'un client RMI-IIOP parle avec un serveur, ce dernier doit être en train de fonctionner. S'il crashe, ou si le réseau crashe, le client est coincé.
- Pas de broadcasting !
  - RMI-IIOP limite les liaisons 1 client vers 1 serveur

Messaging

- C'est comme le mail ! Ou comme si on avait une troisième personne entre le client et le serveur !

Remote method invocations:

Messaging:

Messaging

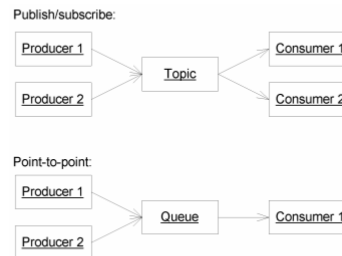
- A cause de ce "troisième homme" les performances ne sont pas toujours au rendez-vous !
- Message Oriented Middleware (MOM) est le nom donné aux middlewares qui supportent le *messaging*.
  - Tibco Rendezvous, IBM MQSeries, BEA Tuxedo/Q, Microsoft MSMQ, Talarian SmartSockets, Progress SonicMQ, Fiorano FioranoMQ, …
  - Ces produits fournissent : messages avec garantie de livraison, tolérance aux fautes, load-balancing des destinations, etc…

The Java Message Service (JMS)

- Les serveurs MOM sont pour la plupart propriétaires : pas de portabilité des applications !
- JMS = un standard pour normaliser les échanges entre composant et serveur MOM,
  - Une API pour le développeur,
  - Un Service Provider Interface (SPI), pour rendre connecter l'API et les serveurs MOM, via les drivers JMS

JMS : Messaging Domains

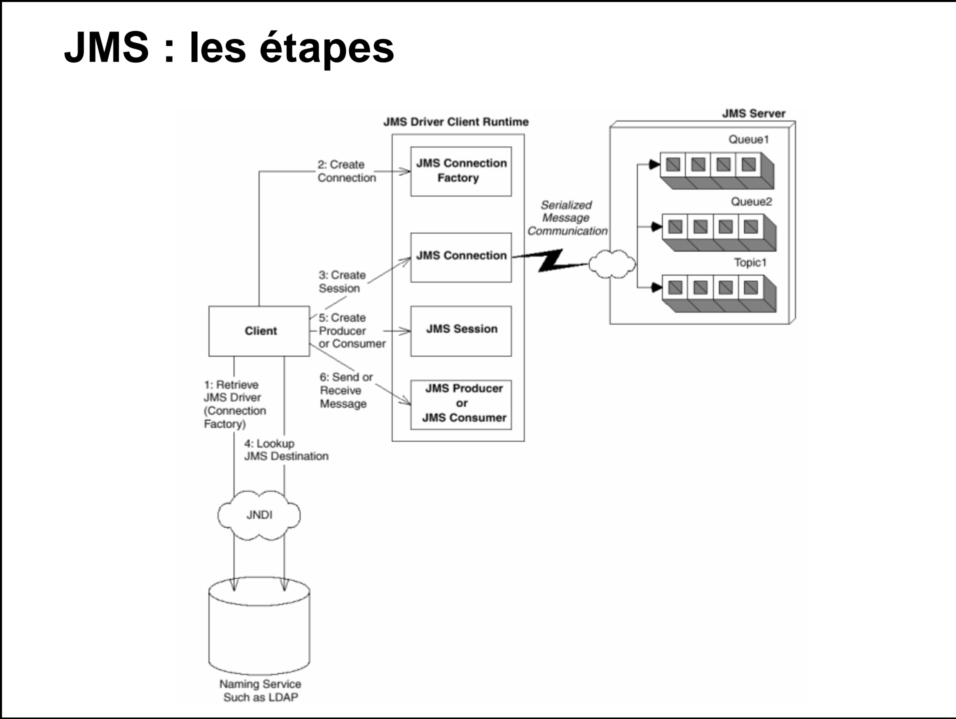
- Avant de faire du mesaging, il faut choisir un *domaine*
  - Domaine = type de messaging
- Domaines possibles
  - Publish/Subscribe (pub/sub) : n producteurs, n consommateurs (tv)
  - Point To Point (PTP) : n producteurs, 1 consommateur



JMS : les étapes

1. *Localiser le driver JMS*
   - lookup JNDI. Le driver est une *connection factory*
2. *Créer une connection JMS*
   - obtenir une *connection* à partir de la *connection factory*
3. *Créer une session JMS*
   - Il s'agit d'un objet qui va servir à recevoir et envoyer des messages. On l'obtient à partir de la *connection.*
4. *Localiser la destination JMS*
   - Il s'agit du canal, de la chaîne télé ! Normalement, c'est réglé par le déployeur. On obtient la *destination* via JNDI.
5. *Créer un producteur ou un consommateur JMS*
   - Utilis és pour écrire ou lire un message. On les obtient à partir de la *destination* ou de la *session.*
6. Envoyer ou recevoir un message

JMS : les étapes



JMS : les interfaces

PARENT INTERFACE	POINT-TO-POINT	PUB/SUB
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

JMS : exemple de code (1)

```
import javax.naming.*;

import javax.jms.*;
import java.util.*;

public class Client {
 public static void main (String[] args) throws Exception {
  // Initialize JNDI
  Context ctx = new InitialContext(System.getProperties());

  // 1: Lookup ConnectionFactory via JNDI
  TopicConnectionFactory factory =
     (TopicConnectionFactory)
      ctx.lookup("javax.jms.TopicConnectionFactory");

  // 2: Use ConnectionFactory to create JMS connection
  TopicConnection connection =
  factory.createTopicConnection();
```

JMS : exemple de code (2)

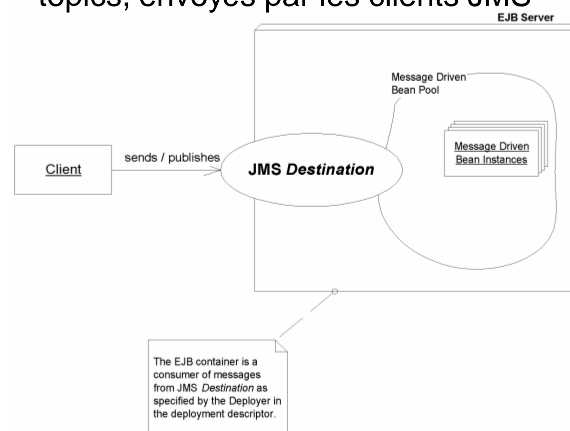
```
  // 3: Use Connection to create session
  TopicSession session = connection.createTopicSession(
   false, Session.AUTO_ACKNOWLEDGE);

  // 4: Lookup Destination (topic) via JNDI
  Topic topic = (Topic) ctx.lookup("testtopic");

  // 5: Create a Message Producer
  TopicPublisher publisher = session.createPublisher(topic);

  // 6: Create a text message, and publish it
  TextMessage msg = session.createTextMessage();
  msg.setText("This is a test message.");
  publisher.publish(msg);
 }
}
```

Intégrer JMS et les EJB

- Pourquoi créer un nouveau type d'EJB ?
- Pourquoi ne pas avoir délégué le travail à un objet spécialisé ?
- Pourquoi ne pas avoir augmenté les caractéristiques des session beans ?
- Parce que ainsi on peut bénéficier de tous les avantages déjà rencontrés : cycle de vie, *pooling*, descripteurs spécialisés, code simple…

Qu'est-ce qu'un Message-Driven Bean ?

- Un EJB qui peut recevoir des messages
  - Il consomme des messages depuis les queues ou topics, envoyés par les clients JMS



Qu'est-ce qu'un Message-Driven Bean ?

- Un client n'accède pas à un MDB via une interface, il utilise l'API JMS,
- Un MDB n'a pas d'interface Home, Local Home, Remote ou Local,
- Les MDB possèdent une seule méthode, faiblement typée : `onMessage()`
  - **Elle accepte un message JMS (BytesMessage, ObjectMessage, TextMessage, StreamMessage ou MapMessage)**
  - **Pas de vérification de types à la compilation.**
  - **Utiliser `instanceof` au run-time pour connaître le type du message.**
- **Les MDB n'ont pas de valeur de retour**
  - **Ils sont découplés des *producteurs* de messages.**

Qu'est-ce qu'un Message-Driven Bean ?

- Pour envoyer une réponse à l'expéditeur : plusieurs *design patterns…*
- Les MDB ne renvoient pas d'exceptions au client (mais au container),
- Les MDB sont stateless…
- Les MDB peuvent être des abonnés durables ou non-durables (*durable or nondurable subscribers*) à un *topic*
  - *Durable = reçoit tous les messages, même si l'abonné est inactif,*
  - *Dans ce cas, le message est rendu persistant et sera délivré lorsque l'abonné sera de nouveau actif.*
  - *Nondurable = messages perdus lorsque abonné inactif.*

Qu'est-ce qu'un Message-Driven Bean ?

- Le consommateur (celui qui peut les détruire) des messages est en général le Container
  - C'est lui qui choisit d'être durable ou non-durable,
  - S'il est durable, les messages résistent au crash du serveur d'application.

Développer un Message-Driven Bean

- Les MDBs doivent implémenter

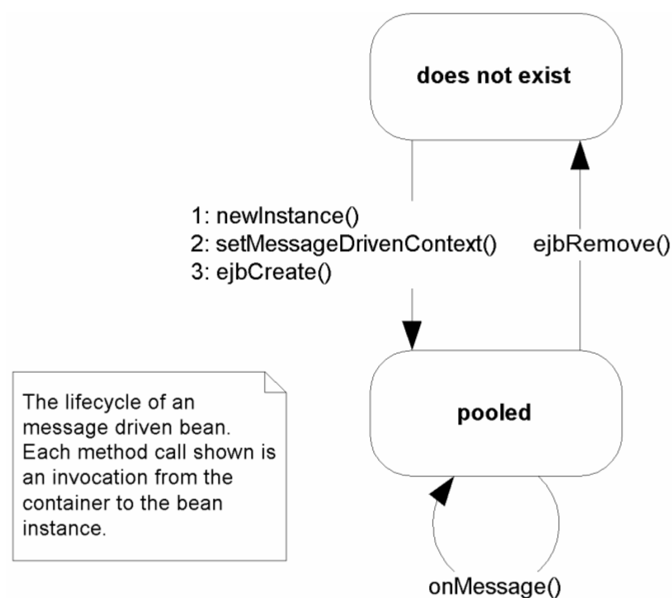
```
public interface javax.jms.MessageListener {
    public void onMessage(Message message);
}
public interface javax.ejb.MessageDrivenBean
 extends EnterpriseBean {

 public void ejbRemove()
  throws EJBException;

 public void setMessageDrivenContext(MessageDrivenContext ctx)
  throws EJBException;
}
```

- La classe d'implémentation doit fournir une méthode **`ejbCreate()`** qui renvoit **`void`** et qui n'a pas d'arguments.

Développer un Message-Driven Bean

- Méthodes qui doivent être implémentées
  - **`onMessage(Message)`**
    - Invoquée à chaque consommation de message
    - Un message par instance de MBD, pooling assuré par le container
  - **`ejbCreate()`**
    - Récupèrer les références aux ressources, initialiser des attributs…
  - **`ejbRemove()`**
    - Libérer les ressources…
  - **`setMessageDrivenContext(MessageDrivenContext)`**
    - Appelée avant ejbCreate, sert à récupèrer le contexte.
    - Ne contient que des méthodes liées aux transactions…

Développer un Message-Driven Bean



Un exemple simple

- Un MDB qui fait du logging, c'est à dire affiche des messages de textes à l'écran chaque fois qu'il consomme un message.
  - Utile pour débugger….
- Rappel : pas d'interfaces !

La classe du bean

```
package examples;

import javax.ejb.*;
import javax.jms.*;

/** Sample Message-Driven Bean */
public class LogBean implements MessageDrivenBean, MessageListener {
 protected MessageDrivenContext ctx;

 /** Associates this Bean instance with a particular context. */
 public void setMessageDrivenContext(MessageDrivenContext ctx) {
  this.ctx = ctx;
 }

  /** Initializes the bean */
 public void ejbCreate() {
  System.err.println("ejbCreate()");
 }
 ...
```

```
...
/** Our one business method */
 public void onMessage(Message msg) {
   TextMessage tm = (TextMessage) msg;
   try {
     String text = tm.getText();
     System.err.println("Received new message : " + text);
   } catch(JMSException e) {
      e.printStackTrace();
   }
 }
  /** Destroys the bean */
 public void ejbRemove() {
  System.err.println("ejbRemove()");
 }
}
```

Le descripteur de déploiement

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
    2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">
<ejb-jar>
 <enterprise-beans>
  <!--
   For each message-driven bean that is located in an ejb-jar file, you have to
    define a <message-driven> entry in the deployment descriptor.
  -->
  <message-driven>
   <!-- The nickname for the bean could be used later in DD -->
   <ejb-name>Log</ejb-name>
   <!-- The fully qualified package name of the bean class -->
   <ejb-class>examples.LogBean</ejb-class>
   <!-- The type of transaction supported (see Chapter 10) -->
   <transaction-type>Container</transaction-type>
   <!-- Whether I'm listening to a topic or a queue -->
   <message-driven-destination>
    <destination-type>javax.jms.Topic</destination-type>
   </message-driven-destination>
  </message-driven>
 </enterprise-beans>
</ejb-jar>
```

Question ?

- Comment sait-on quelle queue ou quel topic de messages le bean consomme ?
  - Cela n'apparaît pas dans le descripteur !
- C'est fait exprès pour rendre les MDB portables et réutilisables.
- L'information se trouve dans le descripteur spécifique

Exemple de descripteur spécifique, tiré d'un autre exemple (Borland)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Borland Software Corporation//DTD Enterprise JavaBeans 2.0//EN"
    "http://www.borland.com/devsupport/appserver/dtds/ejb-jar_2_0-borland.dtd">
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>HelloEJBQueue</ejb-name>
      <message-driven-destination-name>serial://jms/q</message-driven-destination-name>
      <connection-factory-name>serial://jms/xaqcf</connection-factory-name>
      <pool>
        <max-size>20</max-size>
        <init-size>2</init-size>
      </pool>
    </message-driven>
    <message-driven>
      <ejb-name>HelloEJBTopic</ejb-name>
      <message-driven-destination-name>serial://jms/t</message-driven-destination-name>
      <connection-factory-name>serial://jms/tcf</connection-factory-name>
      <pool>
        <max-size>20</max-size>
        <init-size>2</init-size>
      </pool>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Le client (1)

```
import javax.naming.*;

import javax.jms.*;
import java.util.*;

public class Client {
 public static void main (String[] args) throws Exception {
  // Initialize JNDI
  Context ctx = new InitialContext(System.getProperties());

  // 1: Lookup ConnectionFactory via JNDI
  TopicConnectionFactory factory =
     (TopicConnectionFactory)
      ctx.lookup("javax.jms.TopicConnectionFactory");

  // 2: Use ConnectionFactory to create JMS connection
  TopicConnection connection =
  factory.createTopicConnection();
```

Le client (2)

```
  // 3: Use Connection to create session
  TopicSession session = connection.createTopicSession(
   false, Session.AUTO_ACKNOWLEDGE);

  // 4: Lookup Destination (topic) via JNDI
  Topic topic = (Topic) ctx.lookup("testtopic");

  // 5: Create a Message Producer
  TopicPublisher publisher = session.createPublisher(topic);

  // 6: Create a text message, and publish it
  TextMessage msg = session.createTextMessage();
  msg.setText("This is a test message.");
  publisher.publish(msg);
 }
}
```

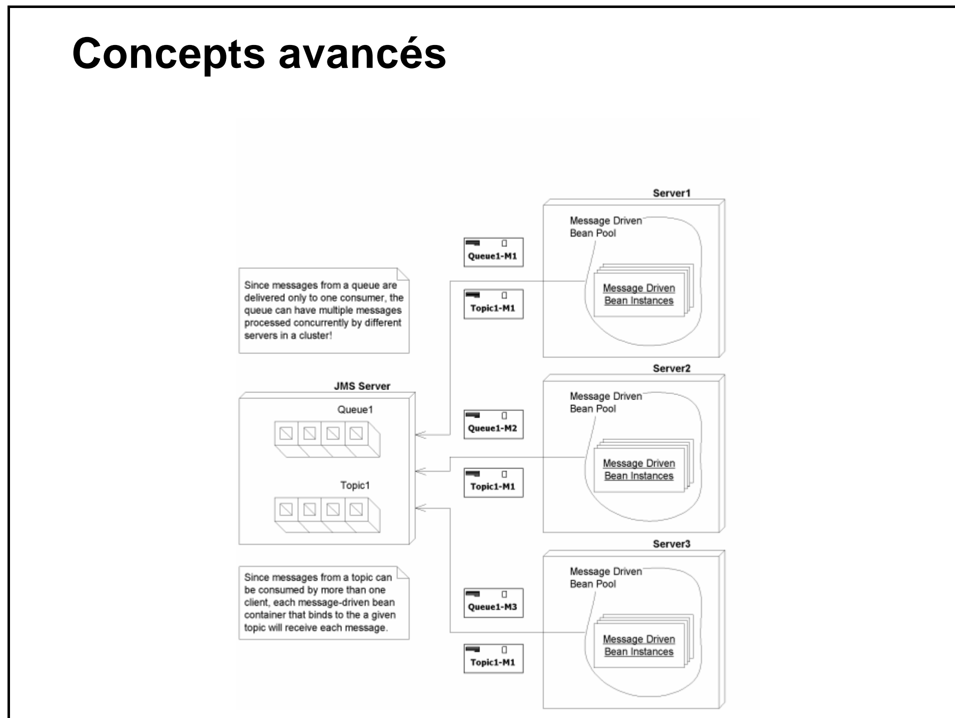
Concepts avancés

- Transactions et MBD,
  - La production et la consommation du message sont dans deux transactions séparées…
- Sécurité,
  - Les MDB ne reçoivent pas les informations de sécurité du producteur avec le message. On ne peut pas effectuer les opérations classiques de sécurité sur les EJB.
- Load-Balancing,
  - Modèle idéal : les messages sont dans une queue et ce sont les MDB qui consomment, d'où qu'ils proviennent.
  - Comparer avec les appels RMI-IIOP pour les session et entity beans, ou on ne peut que faire des statistiques…

Concepts avancés

- Consommation dupliquée dans les architectures en clusters : utiliser une queue au lieu d'un topic si on veut que le message ne soit consommé qu'une fois !
- Chaque container est un consommateur !

Concepts avancés

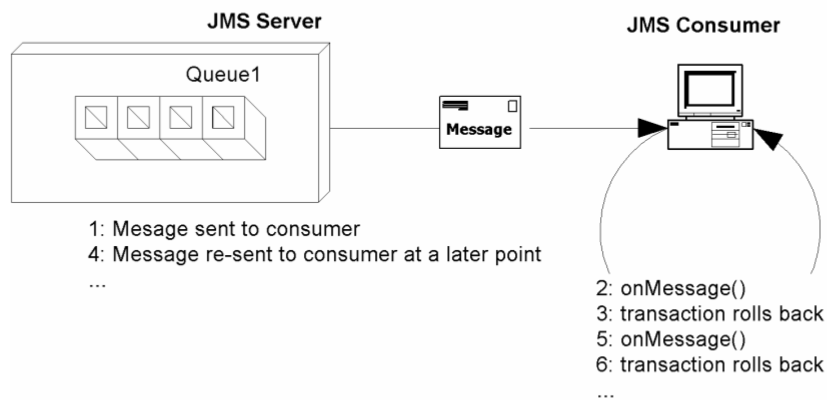


Pièges !

- Ordre des messages
  - Le serveur JMS ne garantit pas l'ordre de livraison des messages.
- L'appel à ejbRemove() n'est pas garanti, comme pour les session beans stateless…
  - A cause du pooling,
  - En cas de crash.
- Messages empoisonnés (*poison messages*)
  - A cause des transactions un message peut ne jamais être consommé

Pièges !

- Messages empoisonnés (*poison messages*)
  - A cause des transactions un message peut ne jamais être consommé



MDB empoisonné !

```
package examples;

import javax.ejb.*;
import javax.jms.*;

public class PoisonBean
 implements MessageDrivenBean, MessageListener {

 private MessageDrivenContext ctx;

 public void setMessageDrivenContext(MessageDrivenContext ctx) {
  this.ctx = ctx;
 }
 public void ejbCreate() {}
 public void ejbRemove() {}
 ...
```

MDB empoisonné !

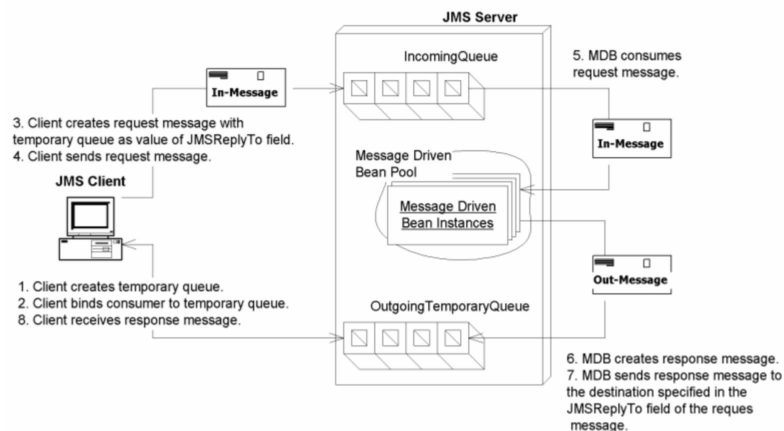
```
...
public void onMessage(Message msg)  {
  try {
   System.out.println("Received msg " + msg.getJMSMessageID());
   // Let's sleep a little bit so that we don't see rapid fire re-sends of the message.
   Thread.sleep(3000);

   // We could either throw a system exception here or
   // manually force a rollback of the transaction.
   ctx.setRollbackOnly();
  }
  catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

MDB empoisonné !

- Solutions
  - Ne pas lever d'exception,
  - Utiliser des transactions gérées par le bean, non par le container,
  - Certains serveurs peuvent configurer une "poison message queue" ou posséder un paramètre "nb max retries"
  - …

Comment renvoyer des résultats à l'expéditeur du message ?

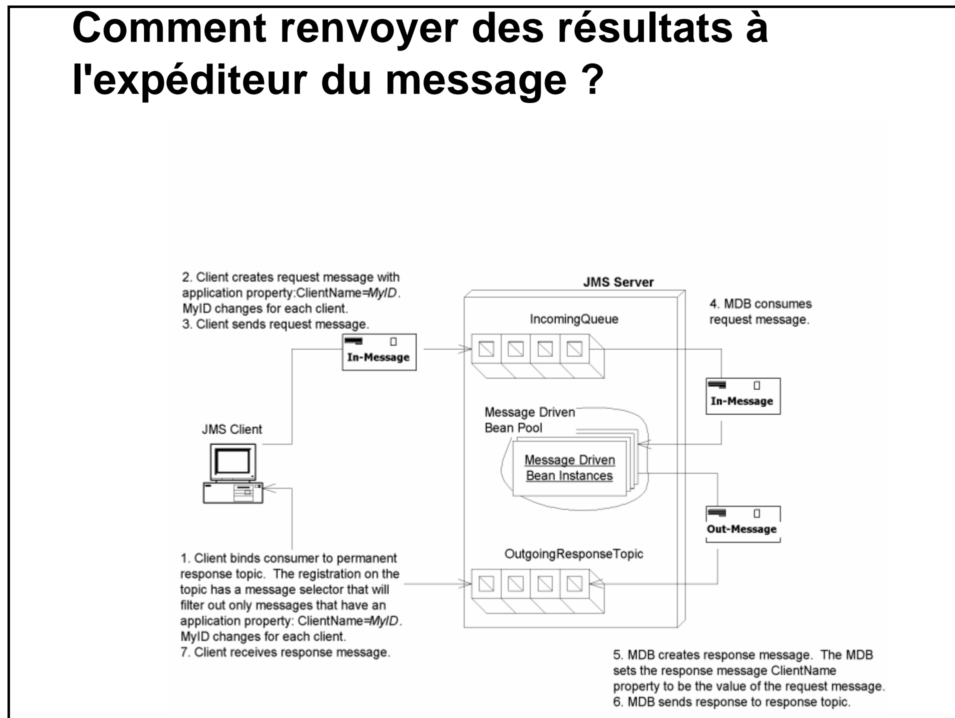
- A faire à la main ! Rien n'est prévu !



Comment renvoyer des résultats à l'expéditeur du message ?

- Néanmoins, des problèmes se posent si le client est lui-même un EJB de type stateful session bean
  - Que se passe-t-il en cas de *passivation* ?
  - Perte de la connexion à la destination temporaire!
- Solution : ne pas utiliser d'EJB SSB comme client! Utiliser une Servlet ou un JSP
- Autre solution : configurer un topic permanent pour les réponses, au niveau du serveur JMS.

Comment renvoyer des résultats à l'expéditeur du message ?



Comment renvoyer des résultats à l'expéditeur du message ?

- D'autres solutions existent…
- JMS propose deux classes **`javax.jms.QueueRequestor`** et **`javax.jms.TopicRequestor`** qui implémentent une pattern simple question/réponse
- Solution bloquante, pas de gestion de transactions…
- Le futur : invocation de méthode asynchrone