http://membres-liglab.imag.fr/donsez

Java Message Service (JMS)

Merci à Didier DONSEZ

Université Joseph Fourier (Grenoble 1)

PolyTech Grenoble LIG ERODS

Didier.Donsez@imag.fr

Didier.Donsez@ieee.org

Motivation

- Messaging Oriented Middleware
- messagerie inter-applicative
- l 'envoi et la réception des messages entre applications est asynchrone
- les messages sont structurés et correspondent à des événements, des requêtes, des rapports, ...
- ne nécessite pas de connexion permanente comme pour le CI/Sv
- *ne pas confondre avec le courrier électronique (API JavaMail)*
- API Java `javax.jms` d 'un client à un serveur MOM
- 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 'abonnent un ou plusieurs *Subscriber*s
- support pour les transactions distribuées XA

JMS targets enterprise messaging; the API is chosen to abstract the programming of a wide variety of message-oriented-middleware (MOM) products in a vendor neutral and portable manner, using the Java programming language. A Destination refers to a named physical resource managed by the underlying MOM. It is administered and configured via vendor provided tools, and typically accessed by a user application via the Java Naming and Directory Interface (JNDI) APIs (external to JMS). A MessageProducer will send messages to a destination and a MessageConsumer can receive messages from a destination. The destination can be thought of a mini-message broker or a channel independent of the producers and consumers.

A Connection is a heavy-weight object representing the link between the application and the middleware. Its attributes include a clientID. It provides methods to start() and stop() communication and to close() a connection. MessageProducer is used to produce messages. A default destination may be specified when the producer is created; it can also be specified when sending messages. In addition, the delivery mode, priority, and expiration can be specified for the outgoing message headers.
A persistent delivery mode means that a message will be delivered once-and-only-once; the message is stored in permanent storage before the send() method returns. A non-persistent delivery mode means that the message will be delivered at most once; a message may be dropped if the JMS provider fails.

Static destinations are discovered via JNDI APIs, which bind logical destination names to destination objects. The static destinations accessible this way must have been previously configured in the JMS middleware (server) using vendor supplied administrative tool. Since JMS discovery is administered, the static destinations must be determined and configured before a client can use them.

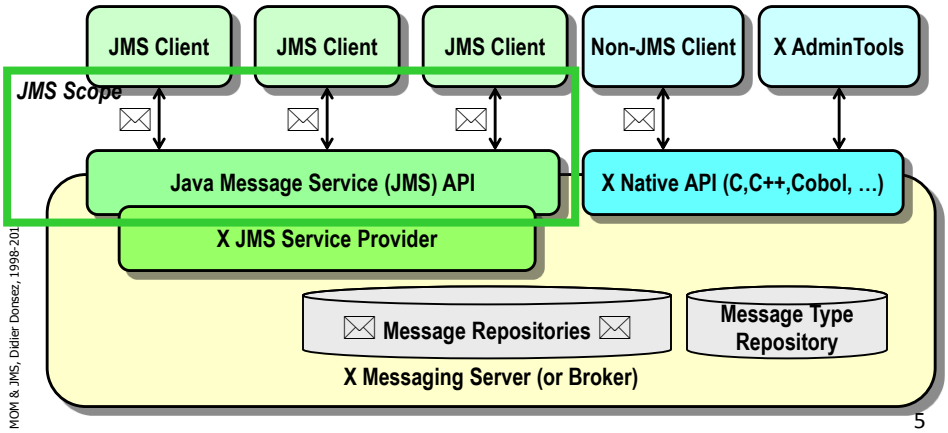
The following is a list of JMS providers:

- **Apache ActiveMQ**
- Redis pub/sub
- Apache Qpid, using AMQP[5]
- Oracle Weblogic (part of the Fusion Middleware suite) and Oracle AQ from Oracle
- EMS from TIBCO
- FFMQ, GNU LGPL licensed
- JBoss Messaging and HornetQ from JBoss
- JORAM, from the OW2 Consortium
- Open Message Queue, from Oracle
- OpenJMS, from The OpenJMS Group
- Solace JMS from Solace Systems
- RabbitMQ by Rabbit Technologies Ltd., acquired by SpringSource
- SAP Process Integration ESB
- SonicMQ from Progress Software
- SwiftMQ
- Tervela
- Ultra Messaging from 29 West (acquired by Informatica)
- webMethods from Software AG
- WebSphere Application Server from IBM, which provides an inbuilt default messaging provider known as the Service Integration Bus (SIBus), or which can connect to WebSphere MQ as a JMS provider[6]
- WebSphere MQ (formerly MQSeries) from IBM
- FioranoMQ

etc
Generally, all Java Enterperise Edition servers offer a JMS provider

Architecture JMS

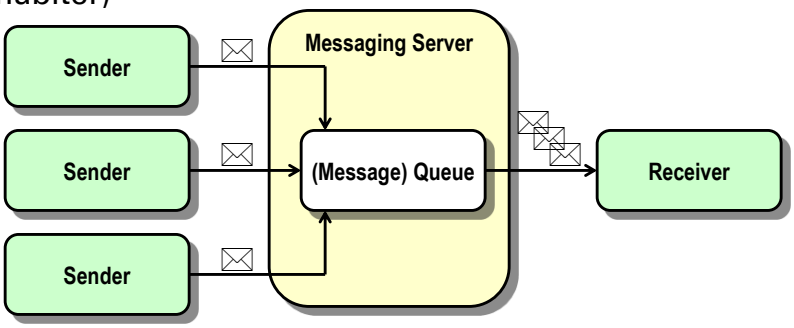
- le client utilise les classes du package `javax.jmx` pour l 'envoi et la réception de messages
- le serveur implante un JMS Service Provider qui l 'interface à son noyau



5

Modèle Point à Point
Point-to-Point

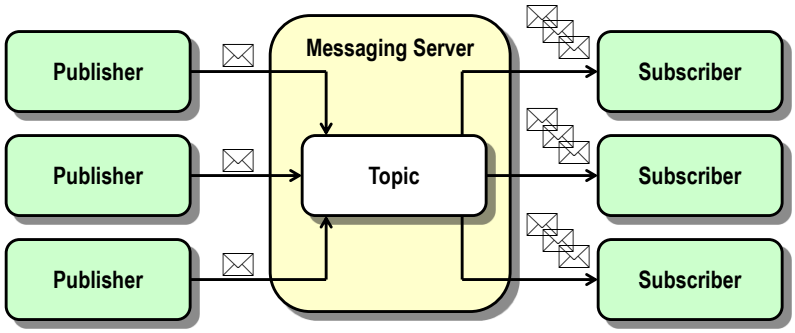
- Concept de Queue
- Un (ou plusieurs) *Sender* envoie (*produit*) un message à une *Queue* (i.e. file d 'attente de messages)
- **Chaque** message de la *Queue* **reçu que par un seul** *receiver* (*consommateur*) (plusieurs receivers peuvent cohabiter)



6

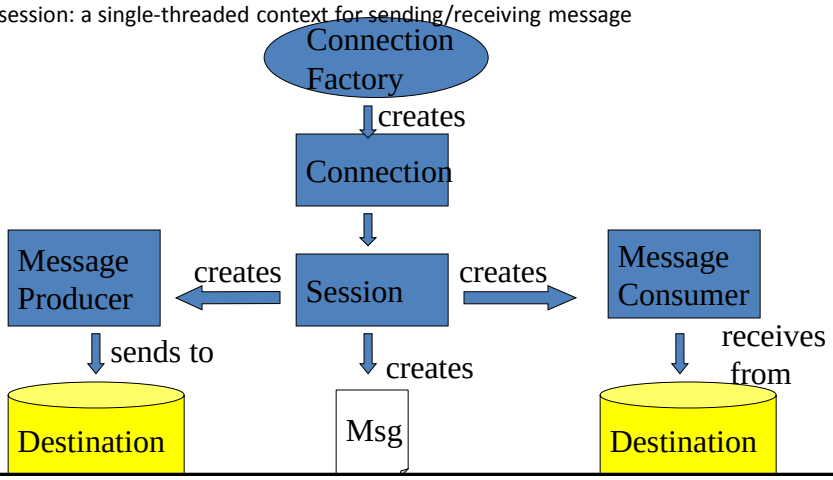
Modèle Publication-Souscription
Publish-Subscribe

- Concept de Topic (centre d intérêt)
- Un (ou plusieurs) *publisher*s envoie un message à un *Topic*
- **Tous les** *subscriber*s de ce *Topic* reçoivent le message



JMS API Programming Model

JMS Interfaces:
connectionfactory: administrative object used by client to create a connection
connection: an active connection for JMS provider
destination: administrative object that encapsulates the identity of a message
destination –queue or topic
session: a single-threaded context for sending/receiving message



API JMS

- Superclasses communes à PtP et PubSub

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

Fonctionnement d 'un client typique JMS

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

Les objets administrés par le Provider

- Les objets administrés par le Provider
  - ConnectionFactory
    - point d 'accès à un serveur MOM
    - accessible par JNDI et enregistré par l 'administrateur du serveur MOM
  - Destination
    - Queue ou Topic administré par le serveur MOM
    - accessible par JNDI et enregistré par l 'administrateur du serveur MOM
- Les objets
  - 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
    - supporte (optionnel) l ' ExceptionListener
  - Session
    - encapsule la liaison TCP/IP avec le Provider JMS

javax.jms.Connection

- Rôle
  - encapsule la liaison TCP/IP avec le Provider JMS
  - authentifie le client et spécifie un identifiant unique de client
- Cycle de vie
  - initialisation/setup
    - crée les Sessions, MessageProducers et les MessageConsumers.
    - fournit le ConnectionMetaData
    - supporte (optionnel) l ' ExceptionListener
  - 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

javax.jms.Session

- Rôle
  - contexte mono-threadé
  - fabrique les MessageProducers et les MessageConsumers
  - fabrique les destinations temporaires TemporaryDestination
  - définit les numéros de série des messages (consommés et produits)
    - un ordre par Session et par Destination. Pas d 'ordre entre Destinations
  - sérialise l 'exécution des MessageListeners enregistrés
    - méthode onMessage()
  - définit une chaîne de transactions atomiques
    - la portée est définie par commit() ou rollback()
  - Acquittement des messages

Tolérance aux pannes et Acquittement des messages

- Principe
  - le client reçoit un message et peut ne le traiter que partiellement avant une erreur (Crash, Exception, ...)
  - En cas d 'erreur dans une Session, le provider JMS renvoie (*redeliver*) ce message au client tant que le client n 'a pas acquitté ce message.
- Modes d 'acquittement
  - mode automatique
  - mode manuel

Tolérance aux pannes et Acquittement des messages

- Mode automatique d 'acquitement
- AUTO_ACKNOWLEDGE : le message est acquité à la réception réussie par le client (receive(), subscribe(), onMessage())
- Transactionnel : l 'acquitement des messages est automatique dans une session transactionnelle
- Mode manuel d 'acquitement
- DUPS_OK_ACKNOWLEDGE : acquittement faible de la livraison des messages. En cas de panne, le client peut recevoir les duplicas. Le client doit le tolérer !
- CLIENT_ACKNOWLEDGE : le message est acquitté par le client en utilisant la méthode Message.acknowledge()
- Remarque: les messages non acquittés précédents un message acquitté avec Message.acknowledge() sont également acquittés
- La méthode Session.recover() arrête la distribution des messages et reprend cette distribution à partir du message non acquitté le plus ancien. Les messages redistribués sont marqués 'redelivered'

15

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

16

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)

Le modèle Point à Point
Point-To-Point PTP

- Classes et interface

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 répartit les messages 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())

Le modèle Publication-Souscription
Publish-Subscribe PubSub

- Classes et interface

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 »

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

- Motivation
  - 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*
- Structure d 'un Message
  - Entête (*header*)
    - 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*)
    - contient les données applicatives

Le modèle de Messages JMS
Les champs d 'entête

- Nom des champs d 'entête

JMSDestination : destination du message
JMSDeliveryMode : sûreté de distribution
NON_PERSISTENT : faible
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 founi 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 : identifiant du type de message défini par l'émetteur
JMSRedelivered : n'a du sens que si le récepteur n 'acquitte pas immédiatement la réception (Transaction)
JMSPriority : priorité (de 0 à 9) du message

- Consultation/Modification
- méthodes setXXX() / getXXX() où XXX est le nom du champ d 'entête

Le modèle de Messages JMS
Les champs de propriétés

- 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 : autre
- servent au filtrage
- servent comme données complémentaires au corps
- exemple : la date d 'expiration d 'un document XML véhiculé par un StringMessage

Le modèle de Messages JMS
Les champs de propriétés

- Enumération des propriétés

  Enumeration getPropertyNames() / Enumeration ConnectionMetaData.getJMSXPropertyNames()
  énumère les propriétés (JMSX) du message

  boolean propertyExists(String) *teste l 'existence d 'une propriété*

- 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 reçus sont en lecture seule sinon l 'exception MessageNotWriteableException est levée
  - Type Objet correspondant : Boolean,Byte,Short,Int,Long,Float,Double,String
  - méthode void setObjectProperty(String,Object)
  - méthode Object getObjectProperty(String)
  - Réinitialisation des propriétés : void clearProperties()

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

- Sous interfaces de javax.jms.Message :

javax.jms.TextMessage ▪contient une String qui peut être un document XML, un message SOAP... ▪**String getText()/ setText(String)**
javax.jms.MapMessage ▪contient un ensemble de paires name-value ▪**int getInt(String name) / setInt(String name, int value), ...** ▪**Object getObject(String)/ setObject(String,Object), Enumeration getMapNames()**
javax.jms.ObjectMessage ▪contient un objet Java sérialisé : **Object getObject()/ setObject(Object)**
javax.jms.BytesMessage ▪contient un tableau de bytes (non interprétés) ▪**int readInt()/writeInt(int), ..., int readBytes(Byte[])/writeBytes(Byte[]),** ▪**Object readObject()/ writeObject(Object), reset()**
javax.jms.StreamMessage ▪contient un flot de données (*Java primitives*) qui s 'écrit et se lit séquentiellement ▪**int readInt()/writeInt(int), ..., String readString()/writeString(String)**

Remarque: Message SOAP sur JMS

- *In a recent "Strategic Planning" research note, Gartner issued a prediction that "by 2004, more than 25 percent of all standard Web services traffic will be asynchronous...." and "by 2006, more than 40 percent of the standard Web services traffic will be asynchronous."*

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

- Méthodes
  - la méthode clearBody() réinitialise le corps
- Remarque
  - le corps d 'un message reçu est en lecture seule

Le modèle de Messages JMS
Le corps (body) du 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);
```


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

```
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);
```

Le modèle de Messages JMS
La sélection (filtrage) des messages

- Motivation
  - le MessageProducer catégorise les messages à envoyer avec les propriétés
  - le MessageConsumer peut filtrer les messages reçus
  - l 'expression de sélection est spécifiée à la fabrication du MessageConsumer
  - les messages reçus sont ceux qui vérifient l 'expression de sélection
- Expressions de sélection
  - sous-ensemble de SQL-92

    identifiants, littéraux (Chaînes, Numériques, TRUE/FALSE)

    connecteurs OR, AND, NOT, ( )

    >, <, >=, <=, <>, =, BETWEEN, LIKE, IN, IS [NOT] NULL
  - Exemple

```
String selector="JMSType='car' AND ( color='blue' OR color='red' ) AND price IS NOT NULL";
QueueReceiver receiver= session.createReceiver(queue, selector);
```


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();
}}
```

ConnectionFactory

dynamicQueues/
queueExo2

-Djava.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory
-Djava.naming.provider.url=tcp://localhost:61616

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); // mis à vrai ou faux
 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();
} }
```


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();
  } }
```

Exemple : Point à Point *Receiver* asynchrone

```
// java JMSQueueAsyncReceiverTest testServer testQueue public class MyListener implements
      javax.jms.MessageListener {
 void onMessage(Message message) { JMSTest.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();
 } }
```


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(); // new InitialContext(properties)
 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();
} }
```

ConnectionFactory

dynamicTopics/ topicExo2

```
Hashtable properties = new Hashtable();
properties.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
properties.put(Context.PROVIDER_URL, "tcp://localhost:61616");
```

Exemple : Publication Souscription *Publisher*

```
// java JMSTopicPublisherTest testServer testTopic Hello 10.0 10
public class JMSTopicPublisherTest  extend JMSTopicTest {

static void publish(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);
  topicPublisher.publish(message);
}

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


Exemple : Publication Souscription *Subscriber* synchrone

```
// java JMSTopicSyncSubscriberTest testServer testTopic
public class JMSTopicSyncSubscriberTest  extend JMSTopicTest {
  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 syncSubscribe() {
    MapMessage message;
    while(!exit) {
      message= (MapMessage)topicSubcriber.receive();
      handleMessage(message);
    }
  }
  public static void main(String args[]) {
    setupSubscriber(argv[0],argv[1]);
    syncSubscriber();
    close();
  } }
```

Exemple : Publication Souscription *Subscriber* asynchrone

```
// java JMSTopicAsyncSubscriberTest testServer testTopic
 public class MyListener implements javax.jms.MessageListener {
  void onMessage(Message message) { JMSTest.handleMessage((MapMessage)message); }
}
public class JMSTopicAsyncSubscriberTest  extend JMSTopicTest {
  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 asyncSubscribe() {
    topicSubscriber.setMessageListener(new MyListener());
    while(!exit) { /* doSomething */ }
   }
public static void main(String args[]) {
    setupSubscriber(argv[0],argv[1]);
    asyncSubscriber();
    close();
  } }
```


Le Transactionnel dans JMS

- Motivation
  - inclure la production et consommation de messages d 'un client dans la portée d 'une transaction distribuée (JTA)
  - une Queue ou un Topic sont considérés comme des ressources XA
  - en cas d 'abandon, les messages ne sont pas postés ou retirés
  - le JMS Provider doit garantir la livraison « une et une seule fois »
  - Remarque : un messsage ne peut pas être posté et retiré dans la même transaction
- API

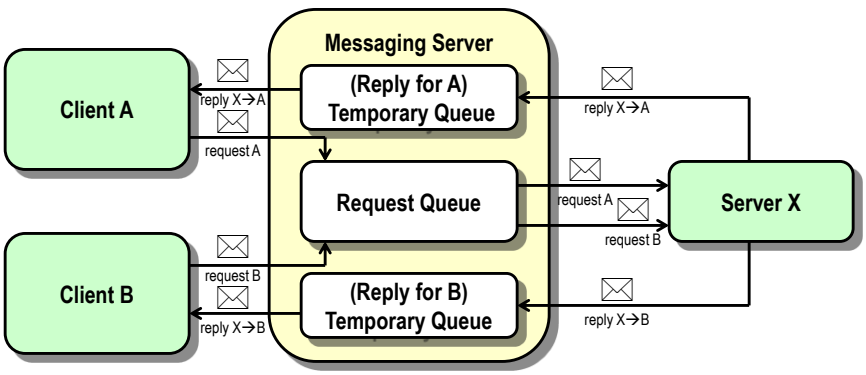
JMS Root	PTP Interface	Pub/Sub Interface
ServerSessionPool		
ServerSession		
ConnectionConsumer		
XAConnectionFactory	XAQueueConnectionFactory	XATopicConnectionFactory
XAConnection	XAQueueConnection	XATopicConnection
XASession	XAQueueSession	XATopicSession

Modèle Requête-Réponse (i)
Request-Reply

- Principe
  - Client-Serveur en mode Asynchrone
  - mais nombreux styles de Requête-Réponse
    - 1- un message de requête donne un message de réponse
    - 2- un message de requête donne un flot de messages de réponse de la part de plusieurs serveurs (*respondents*)
    - ...
- Non explicitement supporté par JMS
  - mais JMS fournit des facilités pour le construire
    - la création de TemporaryQueue et de TemporaryTopic temporaires utilisées pour les réponses (à Destination unique)
    - le champ d 'entête de message JMSReplyTo
      - qui spécifie la Destination pour la réponse
    - le champ d 'entête de message JMSCorrelationID
      - qui peut être utilisé comme référence de la requête originelle

Modèle Requête-Réponse (ii)

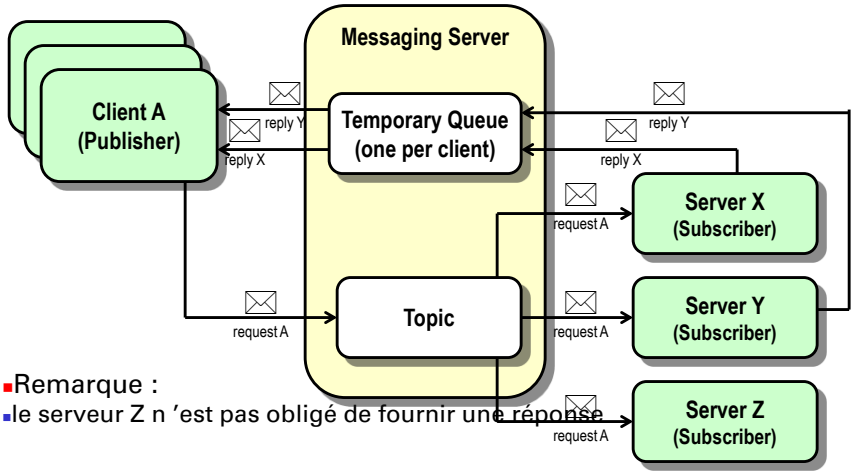
- Proposition d 'implantation de 1 (avec les Queues de JMS)



- Question :
  - A quoi peut servir le partage de la Queue par plusieurs serveurs ?

Modèle Requête-Réponse (iii)

- Proposition d 'implantation de 2 (avec les Topics de JMS)



- Remarque :
- le serveur Z n 'est pas obligé de fournir une réponse

Modèle Requête-Réponse (iii)

- QueueRequestor/TopicRequestor
- classes utilitaires
qui implantent de la modèle synchrone de Requête-Réponse
- la Session ne peut être transactionnelle
- Cycle de vie
- création avec le constructeur
  QueueRequestor(QueueSession session, Queue queue)
  TopicRequestor(TopicSession session, Topic topic)
- envoi d 'une requête et attente de la réponse
  Message request(Message msg)
- fermeture pour libérer les ressources void close()

- Remarque
- la Session ne peut être transactionnelle

Les Destinations Temporaires

- objet unique, qui ne peut être copié
- fabriqué par la Session dont la durée de vie n 'excède pas la Connection
- peut être détruit (delete()) après usage pour libérer des ressources
- ne peut être consommée que par la Connection
- typiquement utilisé pour le modèle Requête-Réponse (propriété *JMSReplyTo)*

- TemporaryQueue
  - objet unique, sous interface de Queue
  - fabriquée par QueueSession.createTemporaryQueue()
  - n 'existe que durant une QueueConnection

Ce que JMS n 'adresse pas

- En tant qu 'API Client, JMS n 'adresse pas
  - Administration du produit MOM
  - Équilibrage de charge et Tolérance aux Pannes
  - Dépôt des types de message
    - JMS ne fournit pas un dépôt des formats (types) de message ni un langage de définition
  - Notification d 'erreur et d 'avertissement
    - chaque produit peut envoyer aux clients des messages systèmes
    - JMS ne normalise pas ces messages.
  - Sécurité
    - JMS ne contrôle pas la confidentialité et l 'intégrité des messages
  - Protocole de transport
  - Déploiement
- Extensions proposées par des éditeurs de MOM
  - Topics hiérarchiques, Dead queue, XMLMessage …

JMS dans J2EE

- Partie intégrante de J2EE/EJB
- JNDI : recherche des objets administrés
- JTA : XASession
- EJB : Message-Driven Bean

Exemple de Chat avec JMS (1/5)

d'après Richard Monson-Haefel & David Chappell

```
package chap2.chat;

import javax.jms.*;
import javax.naming.*;
import java.io.*;
import java.io.InputStreamReader;
import java.util.Properties;

public class Chat implements javax.jms.MessageListener{
   private TopicSession pubSession;
   private TopicSession subSession;
   private TopicPublisher publisher;
   private TopicConnection connection;
   private String userName;
```

Exemple de Chat avec JMS (2/5)

d'après Richard Monson-Haefel & David Chappell

```
/* Constructor. Establish JMS publisher and subscriber */
   public Chat(String topicName, String username, String password) throws Exception {
      // Obtain a JNDI connection
      Properties env = new Properties();
      // ... specify the JNDI properties specific to the vendor
      env.put("BROKER", "localhost");       InitialContext jndi = new InitialContext(env);
      // Lookup a JMS connection factory
      TopicConnectionFactory conFactory =(TopicConnectionFactory)jndi.lookup("TopicConnectionFactory");
      // Create a JMS connection
      TopicConnection connection = conFactory.createTopicConnection(username,password);
      // Create a JMS session object
      TopicSession pubSession = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
      TopicSession subSession = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
      // Lookup a JMS topic
      Topic chatTopic = (Topic)jndi.lookup(topicName);
      // Create a JMS publisher and subscriber
      TopicPublisher publisher =  pubSession.createPublisher(chatTopic);
      TopicSubscriber subscriber = subSession.createSubscriber(chatTopic);
      // Set a JMS message listener
      subscriber.setMessageListener(this);
      // Intialize the Chat application
      set(connection, pubSession, subSession, publisher, username);
      // Start the JMS connection; allows messages to be delivered
      connection.start(); }
```

52

Exemple de Chat avec JMS (3/5)

d'après Richard Monson-Haefel & David Chappell

```
/* Initializes the instance variables */
   public void set(TopicConnection con, TopicSession pubSess,
                   TopicSession subSess, TopicPublisher pub,
                   String username) {
      this.connection = con;
      this.pubSession = pubSess;
      this.subSession = subSess;
      this.publisher = pub;
      this.userName = username;
   }

   /* Receive message from topic subscriber */
   public void onMessage(Message message) {
      try {
         TextMessage textMessage = (TextMessage) message;
         String text = textMessage.getText();
         System.out.println(text);
      } catch(JMSException jmse){ jmse.printStackTrace(); }
   }
```

53

Exemple de Chat avec JMS (4/5)

d'après Richard Monson-Haefel & David Chappell

```
/* Create and send message using topic publisher */
   protected void writeMessage(String text)throws JMSException {
      TextMessage message = pubSession.createTextMessage();
      message.setText(userName+" : "+text);
      publisher.publish(message);
   }
   /* Close the JMS connection */
   public void close() throws JMSException {
      connection.close();
   }
```


Exemple de Chat avec JMS (5/5)

d'après Richard Monson-Haefel & David Chappell

```
/* Run the Chat client */
   public static void main(String [] args){
      try{
         if(args.length!=3) System.out.println("Topic or username missing");
         // args[0]=topicName; args[1]=username; args[2]=password
         Chat chat = new Chat(args[0],args[1],args[2]);
         // read from command line
         BufferedReader commandLine = new
           java.io.BufferedReader(new InputStreamReader(System.in));
         // loop until the word "exit" is typed
         while(true){
            String s = commandLine.readLine();
            if(s.equalsIgnoreCase("exit")){
               chat.close(); // close down connection
               System.exit(0);// exit program
            }else
               chat.writeMessage(s);
         }
      }catch(Exception e){ e.printStackTrace(); }
   }}
```