
# MOM -
# Message Oriented Middleware

&

# JMS -
# Java Message Service

Ada Diaconescu
ada.diaconescu@telecom-paristech.fr

# Plan

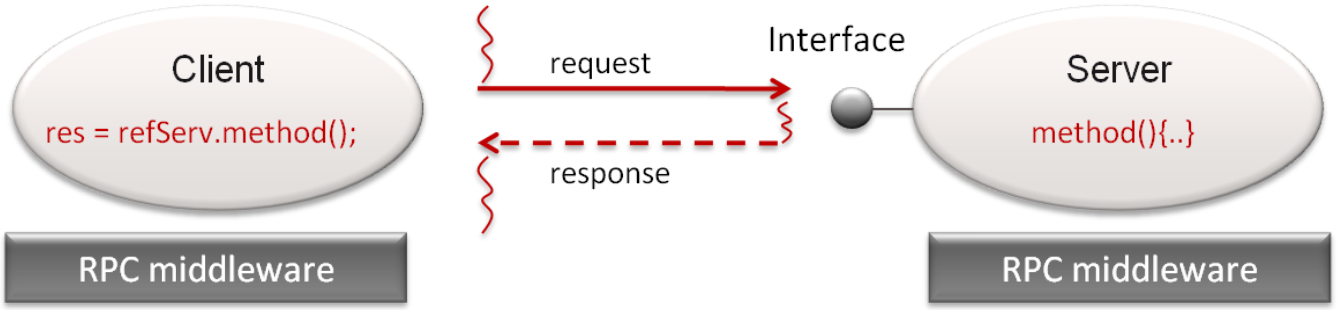
- Message Oriented Middleware – MOM
- Java Message Service – JMS
- Exemples de code (utilisant l'API JMS)


# Partie 1

MOM – Message Oriented Middleware
(Intergiciel Orienté Message)

# Motivation - Pour quoi les *Messages*? (1 /2)

## RPC / RMI – Rappel

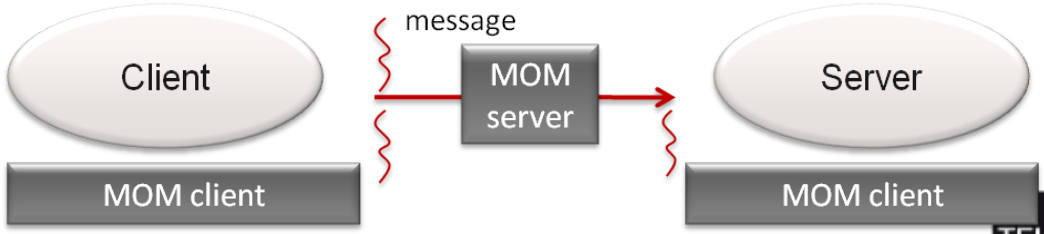



- Couplage fort
  - Le client dépend de l'interface du serveur
  - Le client doit « connaître » le serveur (Référence)
- Dépendance temporelle
  - Le client et le serveur doivent être simultanément disponibles
- Communication synchrone/bloquante (habituellement)
  - Le client est bloqué entre l'envoi de la requête et l'arrivée de la réponse

# Motivation - Pour quoi les *Messages*? (2 /2)

- **Différentes applications ont des besoins et contraintes différents**
  - Pas de disponibilité simultané
    - les différents composants de l'application ne sont pas toujours disponibles simultanément (surtout dans les applications réparties à grande échelle)
  - Possibilité de communication asynchrone / non-bloquante
    - La logique métier de l'application permet à un composant d'envoyer des informations à un autre composant et de continuer son exécution sans recevoir une réponse immédiate
  - Besoins d'un couplage faible
    - Le développeur veut éviter qu'un composant dépende de l'interface des autres composants ou même de « connaître » les autres composants (Références directes) => remplacement facile

=> Communication par Message



# Intergiciels Orientés Messages (MOM)

- Proposent un modèle simple et fiable pour l'échange de messages dans une application répartie
- Utilisent un des modèles de communication les plus anciens
- Sont utilisés dans les systèmes de grande dimension
  - Réseaux bancaires
  - Télécommunications
  - Systèmes de réservation, commerce
  - … etc.

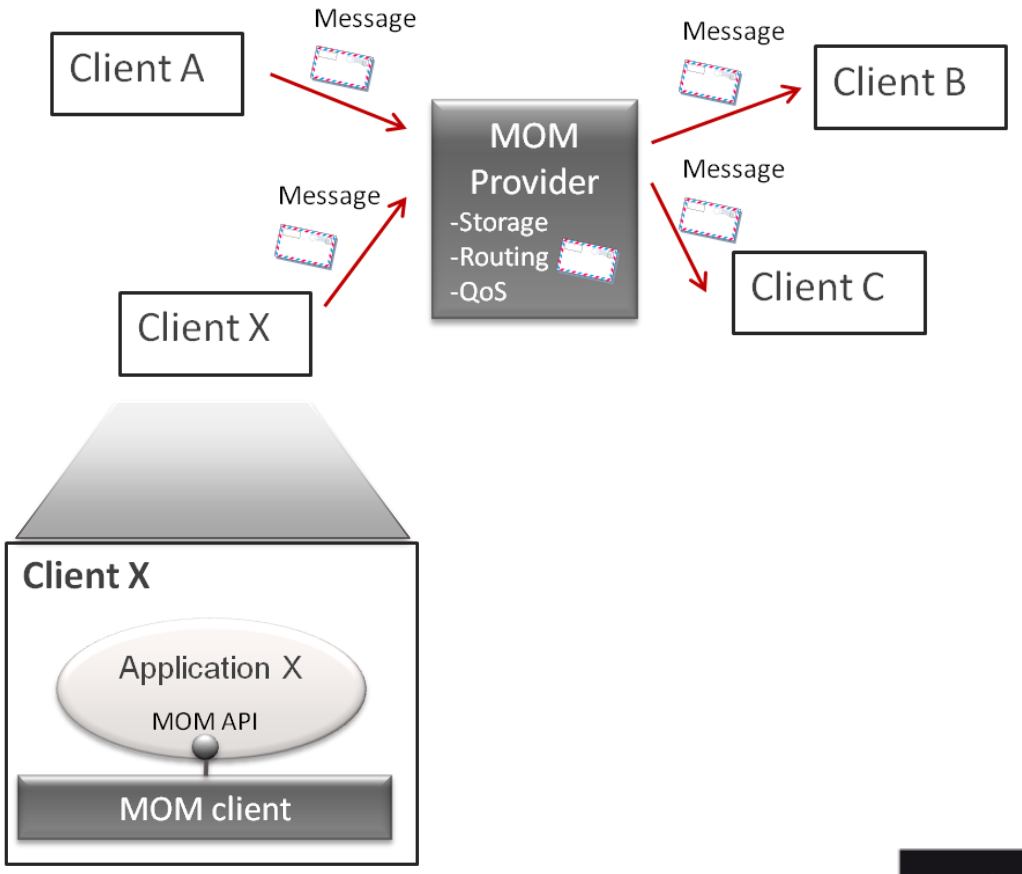
# MOM - Caractéristiques

- Garantie de délivrance de messages
- Support des transactions
- Gestion du routage
- Passage à l'échelle
- Support pour la configuration (politiques de QoS)
- Composants faiblement couplés (généralement !)
  - Pas de dépendance d'interface
  - Pas de référence directe
  - Pas de dépendance temporelle – l'exécution de l'émetteur n'est pas interrompue si le destinataire n'est pas disponible
  - Communication asynchrone / non-bloquante (pas de réponse implicite, sauf ack.)

# MOM – Architecture générale

## Entités de base :

- Client – utilisateur du MOM
  - Émission et réception de messages
- Fournisseur – « provider »
  - support du MOM : routage, stockage des messages
- Message - support des données transmises
  - Type MIME - formats texte, son, image, ..
  - QoS - priorité, date de remise, ..



# Communication par message

## Vue d'ensemble - caractéristiques

- Synchrone vs. Asynchrone
- Persistant vs. Transitoire
- Unicast vs. Multicast
- Mode « push » vs. « pull »
- Modèles de communication :
  Requête/Réponse, Point à Point, Publication / Abonnement, …
- Systèmes de routage des messages
- Bibliothèque de traitement de messages
- ...

**=> Plusieurs modèles de communication par message**

- Implantant diverses combinaisons des aspects indiqués auparavant

# Communication par message

## Communication synchrone vs. asynchrone

- Synchrone / bloquante
  - L'émetteur reste bloqué jusqu'à ce que le destinataire acquitte la réception du message (« Acknowledgement »)
- Asynchrone / non-bloquante
  - L'émetteur continue de s'exécuter après avoir soumis le message pour la transmission

# Communication par message

## Données persistantes vs. transitoires

- Communication persistante (« persistent »)
  - Le serveur MOM conserve le message jusqu'à ce qu'il soit transmis au récepteur; le message n'est jamais perdu ou effacé
  - => pas de dépendance temporelle : l'émetteur et le récepteur *ne* sont *pas* obligés d'être présents en même temps

- Communication transitoire (« transient »)
  - Le serveur MOM conserve le message seulement pendant l'exécution simultanée de l'émetteur et du récepteur
  - => dépendance temporelle : l'émetteur et le(s) récepteur(s) doivent être présents en même temps

## Unicast vs. multicast

- « Unicast »
  - Le message est envoyé à un seul destinataire
- « Multicast » - Diffusion (ou group)
  - Le message est distribué à plusieurs destinataires

# Communication par message

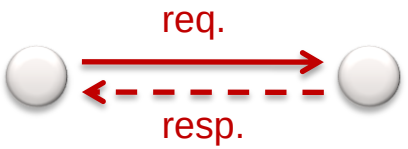
## Mode « push » vs. « pull »

- Mode « push »
  - L'émetteur envoie le message au récepteur
  - Ex. en appelant une méthode de « callback » du récepteur
- Mode « pull »
  - Le récepteur va chercher le message chez l'émetteur
  - Ex. :
    - périodiquement;
    - en restant bloqué jusqu'à ce qu'un message devient disponible;
    - sur notification de l'émetteur (« push » et « pull »)

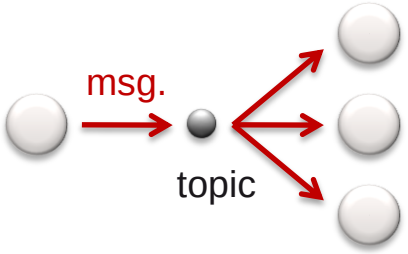
# Modèles de communication par message

- Point à Point (« Point to Point »)


- Requête / Réponse



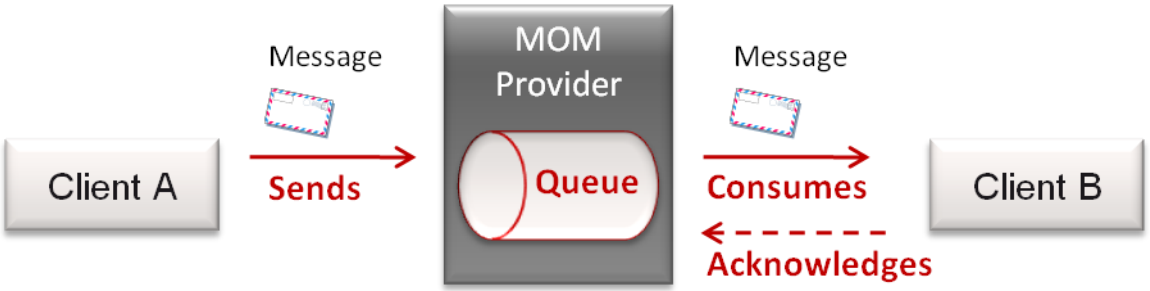

- Publication/Abonnement (« Publish/Subscribe »)




- …

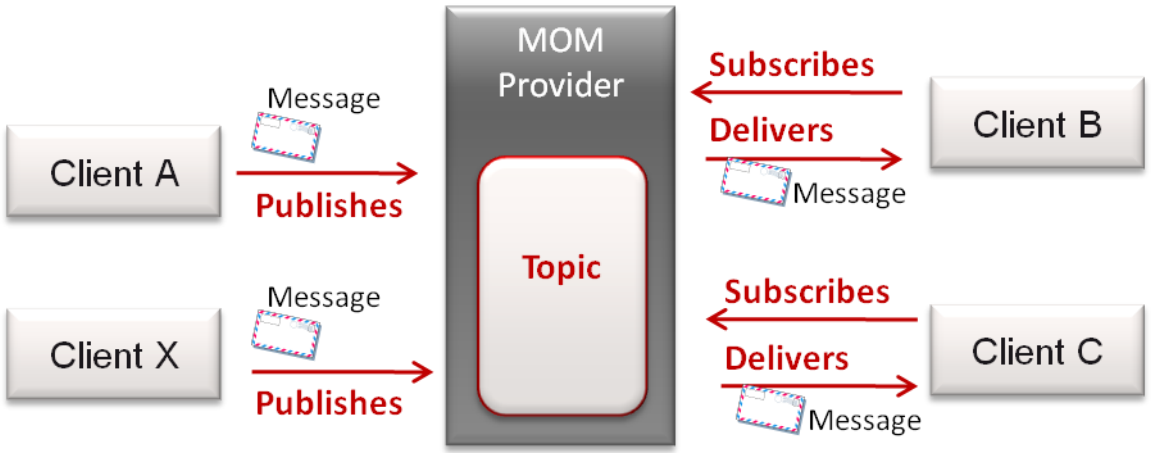
# Modèles de communication par message (ex 1)

- Point-à-Point (“Point-to-Point” - 1 : 1)
  - Chaque message est stocké dans une fille (« Queue ») jusqu’à ce que le destinataire le lise
  - Chaque message est consommé une seule fois (par un seul destinataire)
  - Pas de dépendance temporelle entre émetteur et destinataire du message
  - Le destinataire peut acquitter les messages reçus



# Modèles de communication par message (ex 2)

- Publication/Abonnement (“Publish/Subscribe” - * : * )
  - Le sujet (« Topic ») gère l’envoi de messages pour un ensemble de lecteurs abonnés
  - Découplage entre les « publishers » et les « subscribers »
    - Les fournisseurs n’ont pas besoins de connaître les consommateurs
  - Dépendance temporelle
    - Un client ne peut lire un message qu’après s’être abonné à un topic,
    - Un client abonné à un topic doit continuer à être actif pour recevoir les message du topic.

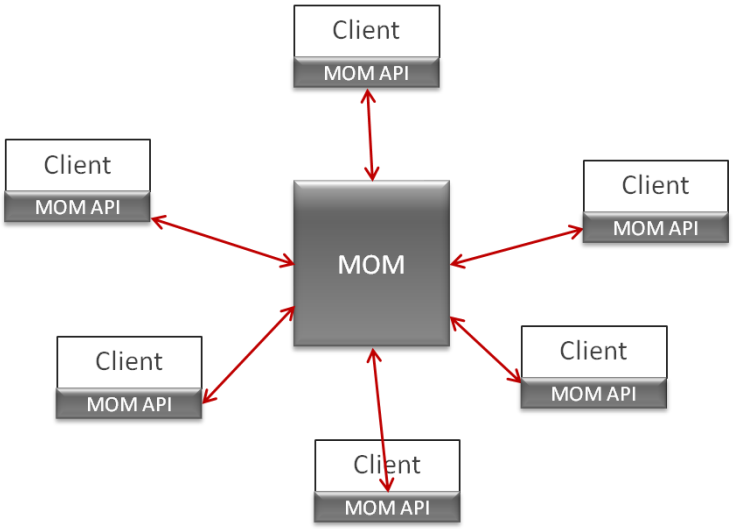


# Topologies pour le Fournisseur de MOM

- Différentes topologies possibles pour le Message Broker
  - Centralisée
  - Décentralisée
  - Hybride
- Chaque produit MOM peut utiliser une ou plusieurs topologies

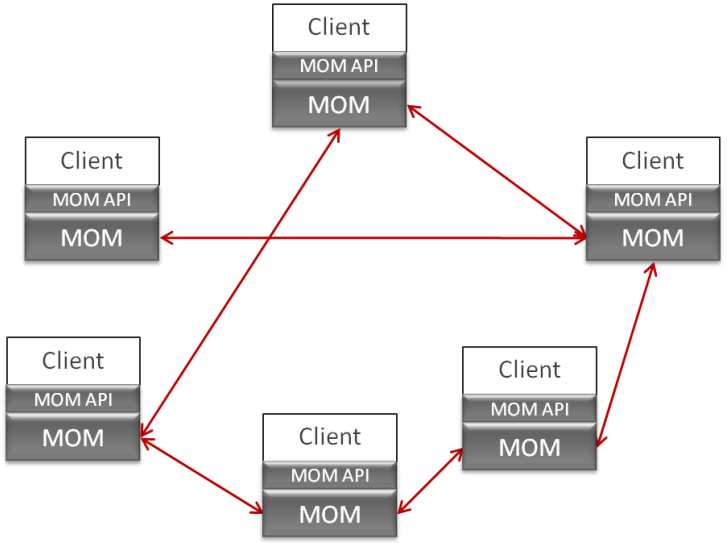
# MOM – topologie centralisée (« hub & spoke »)

- Serveur central de gestion de messages
  - Routage, stockage, transactions, ..
- Avantages
  - Découple les clients, qui ne voient que le serveur - l'ajout ou l'enlèvement d'un client n'a pas d'impact sur les autres clients
- Inconvénients
  - Engendre un trafic réseau important
  - Crée un goulot d'étranglement:
    - Point unique d'échec (« unique point of failure »)
    - Problèmes de passage à l'échelle
    - Possibilité de performance réduite (latence)



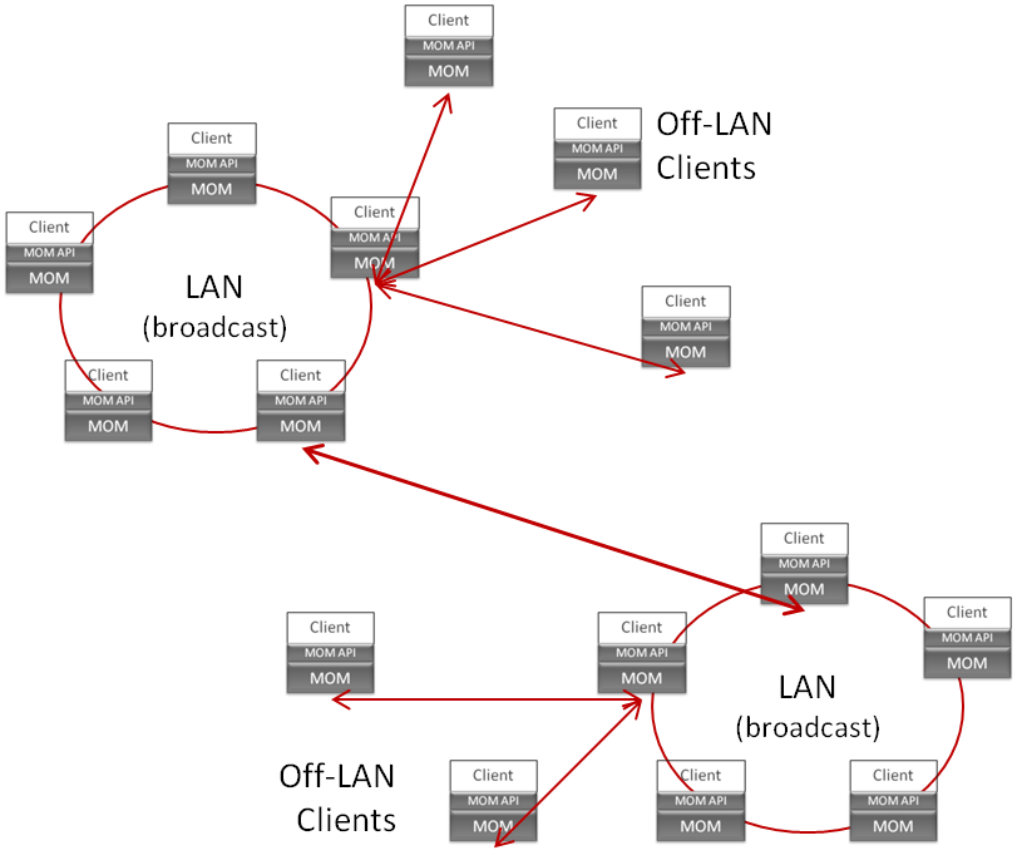
# MOM – topologie décentralisée

- Une instance de MOM est installé chez chaque Client - pas de serveur central
  - Stockage, transactions, sécurité, …
  - Routage de messages
    - Ad-hoc – entre les MOMs
    - Basé sur le protocole réseaux existant - ex. IP-Multicast
- Avantages
  - Distribution des fonctions du MOM entre les serveurs – ex. persistance, sécurité, …
- Inconvénients
  - Problèmes potentiels d'interopérabilité – ex. différents vendeurs ou versions de MOM
  - Clients plus lourds, duplication des fonctions



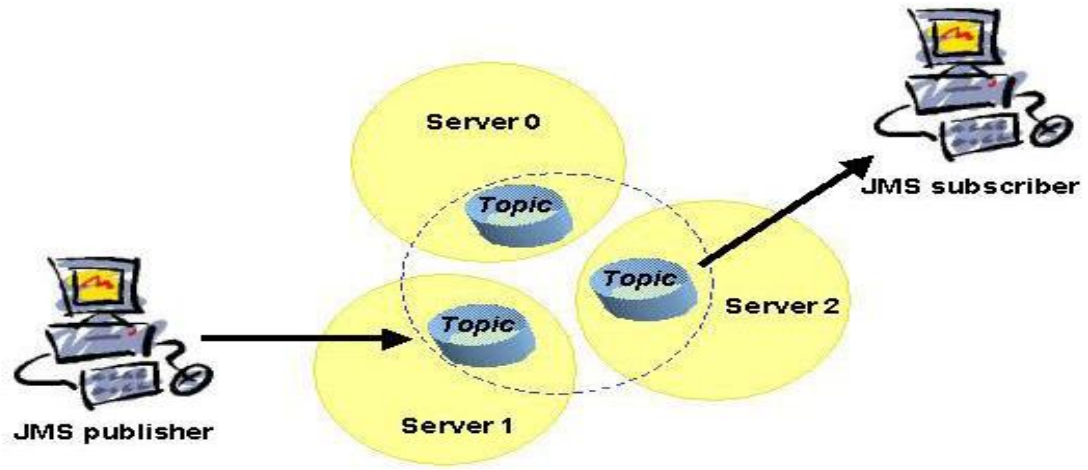
# MOM – topologie hybride

- Combine les deux architectures précédentes
- Combine les avantages et les inconvénients



# MOM – exemple de topologie pour Joram

- Plusieurs Serveurs
- Chaque Client se connecte à un des Serveurs
- Chaque Client peut communiquer avec tout autre Client, quel que soit le Serveur auquel ils se sont connectés



# Paramètres de configuration du MOM

## Administration du MOM

- Déploiement
  - Position des nœuds
  - Allocation des ressources
- Configuration des files de messages
  - Taille des files
  - Persistance
  - Filtrage des messages
- Outils d'aide à l'administration

## Configuration des clients

- Point d'accès au MOM
  - Authentification
  - Établissement de connexions
- Mode d'émission/réception des messages
  - Type de connexion
  - Priorités d'accès au MOM
  - Filtrage en réception


# Partie 2

JMS – Java Message Service

API

# MOM – besoins de standardisation

- Un modèle de répartition - Une définition unique
  - « *modèle de répartition fondé sur l'échange de messages entre les nœuds d'une application répartie* »
- Multiples solutions d'implantation
  - Différentes sémantiques et services offerts
    - Envoi « à l'aveugle » des messages ou support de l'acquittement; support des transactions; …
    - Gestion de la mobilité, support pour la reconfiguration dynamique, …
  - Différentes architectures et implantations
    - Communication à base de TCP/IP, IP multicast, SSL, HTTP, une sous-couche RPC, …
    - Diverses implantations pour les files et les topics
    - Diverses topologie supportées – centralisée, décentralisée, hybride
  - => Besoins de standardisation
    - Ex: Java Message Service (JMS), OMG COS Events/Notification, WS-Reliable Messaging, AMQP, …

# MOM - efforts de standardisation

- Jusqu'à ~2001, peu ou pas d'efforts de normalisation
  - Une API par vendeur
  - Conceptions différentes (ex. utilisation des ressources)
  - Fonctionnalités différentes

- Difficultés
  - Limitation de l'interopérabilité (critique)
  - Problèmes de maintenance et d'évolutivité

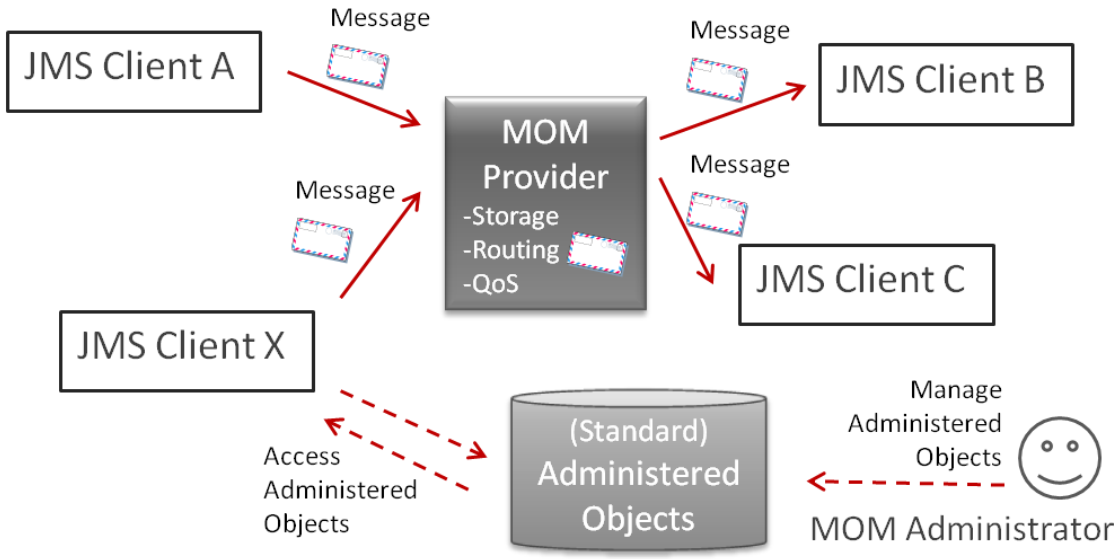
- Évolutions
  - Java Messaging Service (JMS) - une API standard pour le client
  - CORBA COS Notification - une API pour le client, description de l'infrastructure (objets et API)
  - Advanced Message Queueing Protocol (AMQP) – un standard ouvert pour l'interopérabilité d'intérgiciels à message écrits en différents langages et pour différentes plates-formes
  - …

# (Sun/Oracle's) JMS: Java Message Service

- Spécification d'un API pour les MOM
- Intégré à J2EE 1.3 ++, couplage avec les EJB (Message-Driven Bean)
- Première spécification d'un MOM publiquement accessible
  - Implémentée par les principaux MOM
  - Adaptable à d'autres langages (C++, Ada)
  - Peu restrictive: synthèse des MOM existants => autorise plutôt qu'interdit
- JMS : spécification 1.1
  - Spécification pour le client
    - P-t-P, Pub/Sub, call-backs,
    - Filtrage (syntaxe à la SQL) et transactions
    - Type de messages
  - Ne spécifie pas l'infrastructure
    - Protocole, représentation, transport
    - Processus de configuration
    - Gestion des erreurs, de la reprise sur pannes
    - Interfaces d'administration
    - Sécurité

# JMS Architecture

- **JMS Client** – utilise l'API JMS
  - Pour envoyer/recevoir des messages
  - Pour les « Objets Administrés »
- **JMS Provider** – impl. l'API JMS
  - Gestion des files et topics
  - Transmission des messages
  - Configuration globale du MOM
  - Source d'hétérogénéité –
    ex. mécanismes d'administration, protocole utilisé, gestion des erreurs, …
- **JMS Message** – objets contenant l'information communiquée
  - Un en-tête
  - Des propriétés (optionnel)
  - Un corps (optionnel)




- **Objets Administrés**
  - Fabriques de connections (« Connection Factories »)
  - Destinations: files et sujets d'intérêt (« queues » et « topics »)

# Message JMS

- **Trois parties: entête, propriétés, corps**
- **En-tête** du message - identification et routage du message
  - Paires (nom, valeur):

    `JMSDestination, JMSDeliveryMode, JMSMessageID, JMSTimestamp, JMSExpiration, JMSRedelivered, JMSPriority, JMSReplyTo, JMSCorrelationID, JMSType`

- **Propriétés** (optionnel) – spécifiques à l'application, utilisées pour filtrer les messages

  Ex – sender : `message.setStringProperty("Username","John Doe");`

  Ex – receiver :
  ```
  TopicSubscriber sub =
  session.createSubscriber(chatTopic,"Username !='William'");//avec filtre!
  ```

- **Corps** (optionnel) - contient les données de l'application

  Ex: `message.setText(MSG_TEXT + " " + (i + 1));`

  - TextMessage: chaîne de caractères
  - MapMessage: ensemble de paires (nom, valeur)
  - ByteMessage: flux d'octets
  - StreamMessage: flux de valeurs
  - ObjectMessage: objet sérialisable

# Réception de messages JMS

- *Attention à l'utilisation des termes « synchrone » & « asynchrone » !*

- Réception **Synchrone** - mode « pull », bloquant
  - Le consommateur récupère explicitement le message depuis la Destination en appelant la méthode **`receive`**.
  - Cette méthode bloque jusqu'à ce qu'un message soit disponible, ou qu'un délai expire.

- Réception **Asynchrone** – mode « push », avec dépendance temporelle
  - Le consommateur enregistre un « Message Listener » auprès de la Destination ciblée
  - Lorsqu'un message arrive, le fournisseur JMS délivre le message au « Message Listener » en appelant la méthode **`onMessage`**.

# Types de Destinations JMS

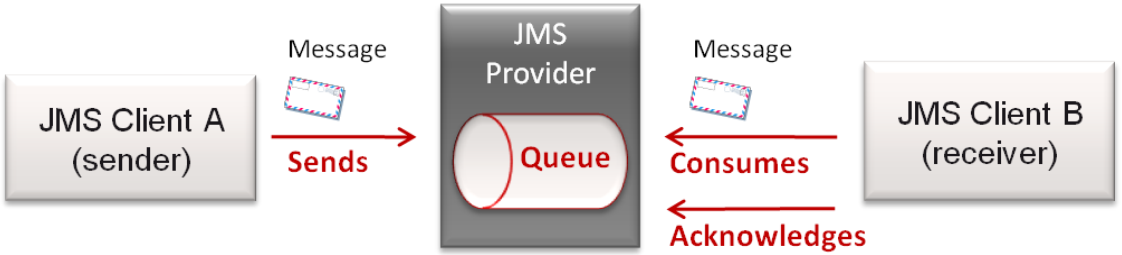
- File - « Queue »
  - Persistance des messages
  - Découplage temporel entre le producteur et le consommateur des messages
  - Habituellement utilisé pour la communication Point-à-Point

- Sujet - « Topic »
  - Non-persistance des messages
  - Couplage temporel entre  le producteur et le consommateur des messages (sauf utilisation de l'option « **durable subscription** »)
  - Habituellement utilisé pour la communication Publication/Abonnement

- *Note: JMS permet l'utilisation des deux types de destinations - queues et topics, avec les deux modes de réception - synchrone et asynchrone.*

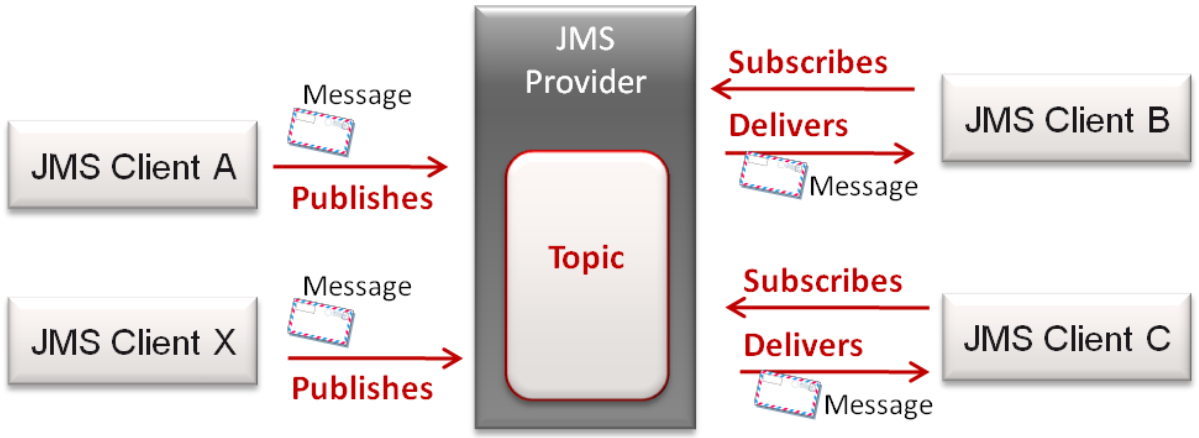
# Modèles de communication JMS (1 /2)

- Point-à-Point
  - Chaque message est stocké dans une File jusqu'à ce qu'un destinataire le lise (ou jusqu'à ce que le message expire)
  - Chaque message est consommé une seule fois, par un seul destinataire
  - Pas de dépendance temporelle entre l'émetteur et le destinataire du message
  - Le destinataire acquitte les messages reçus



# Modèles de communication JMS (2 /2)

- Publication / Abonnement
  - Chaque message peut avoir plusieurs consommateurs
  - Dépendance temporelle entre le producteur et le consommateur d'un message
    - Les consommateurs reçoivent seulement les messages produits après leur souscription
    - Les consommateurs doivent rester disponibles afin de recevoir des messages
      (les consommateurs peuvent créer des souscriptions « durables » afin de relâcher cette contrainte)



# JMS Client – Comment initialiser la communication ?

- Une Usine de Connexions – « Connection Factory »
  - Prend en charge la connexion avec le fournisseur JMS (MOM).
  - Encapsule les paramètres de connexion mis en place par l'administrateur.
- Une Session
  - Est un contexte mono-tache
  - Utilisé pour l'émission et la réception de messages
  - Gère plusieurs consommateurs et producteurs de message

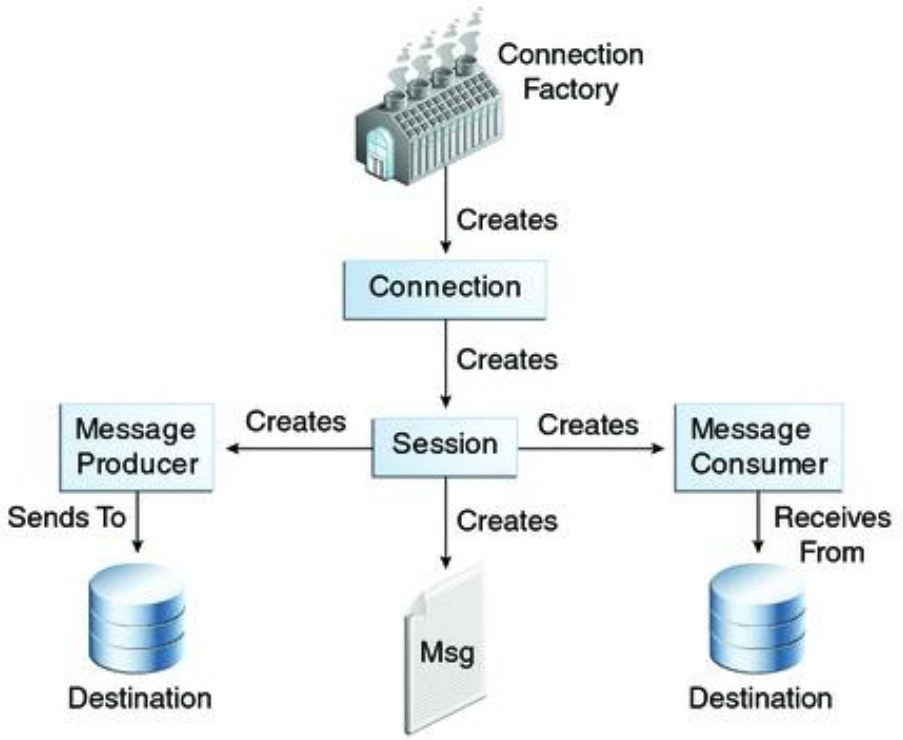



Image from Oracle's JMS tutorial
http://download.oracle.com/javaee/6/tutorial/doc/bnceh.html

# JMS Client – Mais comment obtenir la Fabrique de Connexions et les Destinations?

- L'administrateur du MOM:
  - Crée les Objets Administrés : Usines de Connections (FC) et Destinations (D)
  - Ajoute les Objets Administrés dans l'annuaire JNDI (opération Bind)

  à l'aide d'un Outil Administratif

- Les Clients JMS:
  - Utilisent l'annuaire JNDI afin d'obtenir les Objets Administrés – FC, D
  - Utilisent les Fabriques afin d'obtenir des connections logiques au MOM
  - Utilisent les Destinations afin d'envoyer/recevoir des Messages

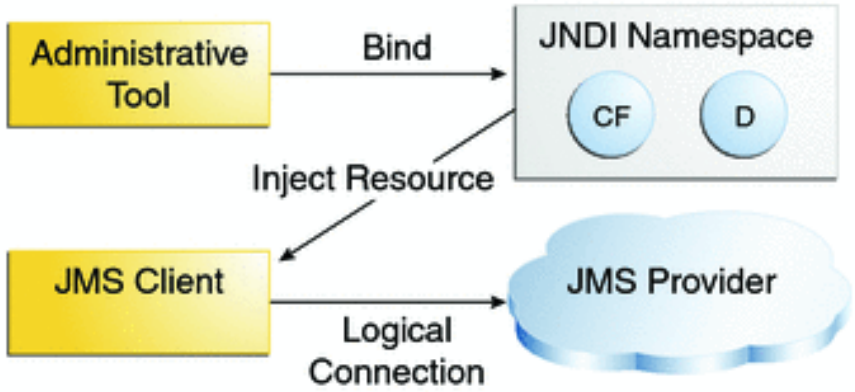



Image from Oracle's JMS tutorial
http://download.oracle.com/javaee/6/tutorial/doc/bncdx.html

# JMS - - Lien avec d'autres API Java

- JNDI: annuaire
- JTA: transaction

# Fournisseurs JMS

- **Sun Java System Message Queue** (Sun → Oracle)
  - JMS intégré au serveur d'entreprise de Sun (GlassFish)
  - http://www.oracle.com/us/products/middleware/application-server/oracle-glassfish-server/index.html
- **MQ JMS** (IBM)
  - Un des leaders sur ce marché
  - http://www7b.software.ibm.com/wsdd/library/techtip/0112_cox.html
- **WebLogic JMS** (BEA → Oracle)
  - Serveur d'entreprise, peut supporter JMS
  - http://download.oracle.com/docs/cd/E13222_01/wls/docs92/messaging.html
- **JMSCourier** (Codemesh)
  - Application C++ et JMS
  - http://www.codemesh.com/en/AlignTechnologyCaseStudy.html
- **TIBCO** http://www.tibco.com/products/soa/messaging
- **Fiorano Software** http://www.fiorano.com
- **JRUN Server** http://www.allaire.com
- **GemStone** http://www.gemstone.com
- **Nirvana** http://www.pcbsys.com
- **Oracle** http://www.oracle.com
- **…**
- **Vendor lists**
  - http://www.techspot.co.in/2006/06/jms-vendors.html
  - http://adtmag.com/articles/2003/01/31/jms-vendors.aspx

- **Joram**
  - **http://joram.ow2.org/**
  - **Logiciel Libre (licence LGPL)**
  - **Développé par ScalAgent (depuis 1999)**

***Utilisé en TP***

# Références JMS

- JMS homepage (Sun → Oracle)
  - http://java.sun.com/products/jms → http://www.oracle.com/technetwork/java/jms/index.html
- JMS specifications
  - http://www.oracle.com/technetwork/java/jms-101-spec-150080.pdf
- JMS Tutorial
  - http://download.oracle.com/javaee/1.3/jms/tutorial
- JMS API FAQs
  - http://www.oracle.com/technetwork/java/faq-140431.html
- Gopalan Suresh Raj, September, 1999
  - http://www.execpc.com/~gopalan/jms/jms.html
- Richard Monson-Haefel & David A. Chappell, “Java Message Service”, O’Reilly, January 2001


# Partie 3

Exemples de code

avec l'API JMS
(http://download.oracle.com/javaee/6/api)

# Les Exemples

- # 1: JMS Sender
- # 2: JMS Receiver
- # 3: JMS Publisher
- # 4: JMS Listener

# Exemple JMS: Point-à-Point #1 (1/4)
## *Sender*

```
import javax.jms.*;
import javax.naming.*;

public class SimpleQueueSender {
     public static void main(String[] args) {
          String queueName = null;
          Context jndiContext = null;
          QueueConnectionFactory queueConnectionFactory = null;
          QueueConnection queueConnection = null;
          QueueSession queueSession = null;
          Queue queue = null;
          QueueSender queueSender = null;
          TextMessage message = null;
          final int NUM_MSGS;
          final String MSG_TEXT = new String("Here is a
                                                  message");
          Properties env = new Properties();
          // Specify the vendor-specific JDNI properties
```

# Exemple JMS: Point-à-Point #1 (2/4)
## *Sender*

```
queueName = "Test Queue";

//Create a JNDI InitialContext object
try {

     jndiContext = new InitialContext(env);

}
catch (NamingException e) {
     System.out.println("Could not create JNDI " + "context: " +
                                                    e.toString());
     System.exit(1);
}
```

# Exemple JMS: Point-à-Point #1 (3/4)
## *Sender*

```
//Look up the connection factory and the queue
//If either does not exist Then exit
try {

        queueConnectionFactory = (QueueConnectionFactory)
                jndiContext.lookup("QueueConnectionFactory");

        queue = (Queue)jndiContext.lookup(queueName);

}
catch (NamingException e) {
        System.out.println("JNDI lookup failed: " + e.toString());
        System.exit(1);
}
```

# Exemple JMS: Point-à-Point #1 (4/4)
## *Sender*

```
try {
      queueConnection =
            queueConnectionFactory.createQueueConnection();

      queueSession = queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

      queueSender = queueSession.createSender(queue);

      message = queueSession.createTextMessage();

      for (int i = 0; i < NUM_MSGS; i++) {
            message.setText(MSG_TEXT + " " + (i + 1));
            System.out.println("Sending message: "
                  + message.getText());
            queueSender.send(message);
      }
}
catch (JMSException e) {
      System.out.println("Exception occurred: " + e.toString());
}
```

# Exemple JMS: Point-à-Point #2 (1/4)
## *Receiver*

```
import javax.jms.*;
import javax.naming.*;

public class SimpleQueueReceiver {
     public static void main(String[] args) {
            String queueName = null;
            Context jndiContext = null;
            QueueConnectionFactory queueConnectionFactory = null;
            QueueConnection queueConnection = null;
            QueueSession queueSession = null;
            Queue queue = null;
            QueueReceiver queueReceiver = null;
            TextMessage message = null;

            Properties env = new Properties();
            // Specify the vendor-specific JDNI properties
```

# Exemple JMS: Point-à-Point #2 (2/4)
## *Receiver*

```
  queueName = "Test Queue";

  //Create an JNDI InitialContext object
  try {

       jndiContext = new InitialContext(env);

}
  catch (NamingException e) {
       System.out.println("Could not create JNDI " + "context: " +
                                                    e.toString());
       System.exit(1);
  }
```

# Exemple JMS: Point-à-Point #2 (3/4)
## *Receiver*

```
//Look up connection factory and queue
//If either does not exist Then exit
  try {
       queueConnectionFactory = (QueueConnectionFactory)
             jndiContext.lookup("QueueConnectionFactory");

       queue = (Queue) jndiContext.lookup(queueName);
  }
  catch (NamingException e) {
       System.out.println("JNDI lookup failed: " + e.toString());
       System.exit(1);
  }
```

# Exemple JMS: Point-à-Point #2 (4/4)
## *Receiver*

```
try {
      queueConnection =
            queueConnectionFactory.createQueueConnection();

      queueSession = queueConnection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

      queueReceiver = queueSession.createReceiver(queue);

      queueConnection.start(); //!!!

      while (true) {
            Message m = queueReceiver.receive(1);//timeout = 1ms
            if (m != null) {
                  if (m instanceof TextMessage) {
                        message = (TextMessage) m;
                        System.out.println("Reading message: " +
                                          message.getText());
      }
}
catch (JMSException e) {
      System.out.println("Exception occurred: " + e.toString()
}
```

# Exemple JMS: Pub/Sub #1 (1/4)
## *Publisher*

```
import javax.jms.*;
import javax.naming.*;

public class SimpleTopicPublisher{
     public static void main(String[] args) {

               String topicName = null;
               Context jndiContext = null;
               TopicConnectionFactory topicConnectionFactory = null;
               TopicConnection topicConnection = null;
               TopicSession topicSession = null;
               Topic topic = null;
               TopicPublisher topicPublisher = null;
               TextMessage message = null;
               final int NUM_MSGS;
               final String MSG_TEXT = new String("Here is a
                                                   message");

               Properties env = new Properties();
               // Specify the vendor-specific JDNI properties
```

# Exemple JMS: Pub/Sub #1 (2/4)
## *Publisher*

```
  topicName = "Test Topic";

  //Create a JNDI InitialContext
  try {

       jndiContext = new InitialContext(env);

}
  catch (NamingException e) {
       System.out.println("Could not create JNDI " + "context: " +
                                                    e.toString());
       System.exit(1);
  }
```

# Exemple JMS: Pub/Sub #1 (3/4)
## *Publisher*

```
//Look up connection factory and topic
//If either does not exist Then exit
  try {

       topicConnectionFactory = (TopicConnectionFactory)
              jndiContext.lookup("TopicConnectionFactory");


       topic = (Topic) jndiContext.lookup(topicName);

  } catch (NamingException e) {
       System.out.println("JNDI lookup failed: " + e.toString());
       System.exit(1);
  }
```

# Exemple JMS: Pub/Sub #1 (4/4)
## *Publisher*

```
try {
      topicConnection =
            topicConnectionFactory.createTopicConnection(user,pswd);

      topicSession = topicConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);

      topicPublisher = topicSession.createPublisher(topic);

      message = topicSession.createTextMessage();

      for (int i = 0; i < NUM_MSGS; i++) {
            message.setText(MSG_TEXT + " " + (i + 1));
            System.out.println("Publishing message: "
                  + message.getText());
            topicPublisher.publish(message);
      }
} catch (JMSException e) {
      System.out.println("Exception occurred: " + e.toString());
}
```

# Exemple JMS: Pub/Sub #2 (1/5)
## *Subscriber*

```
import javax.jms.*;
import javax.naming.*;

public class SimpleTopicSubscriber{
     public static void main(String[] args) {

               String topicName = null;
               Context jndiContext = null;
               TopicConnectionFactory topicConnectionFactory = null;
               TopicConnection topicConnection = null;
               TopicSession topicSession = null;
               Topic topic = null;
               TopicSubscriber topicSubscriber = null;
               TextMessage message = null;
               final int NUM_MSGS;
               final String MSG_TEXT = new String("Here is a
                                                   message");

               Properties env = new Properties();
               // Specify the vendor-specific JDNI properties
```

# Exemple JMS: Pub/Sub #2 (2/5)
## *Subscriber*

```
topicName = "Test Topic";

//Create a JNDI InitialContext
try {

     jndiContext = new InitialContext(env);

}
catch (NamingException e) {
     System.out.println("Could not create JNDI " + "context: " +
                                                    e.toString());
     System.exit(1);
}
```

# Exemple JMS: Pub/Sub #2 (3/5)
## *Subscriber*

```
//Look up connection factory and topic
//If either does not exist Then exit
try {

     topicConnectionFactory = (TopicConnectionFactory)
           jndiContext.lookup("TopicConnectionFactory");

     topic = (Topic) jndiContext.lookup(topicName);

} catch (NamingException e) {
     System.out.println("JNDI lookup failed: " + e.toString());
     System.exit(1);
}
```

# Exemple JMS: Pub/Sub #2 (4/5)
## *Subscriber*

```
try {
    topicConnection =
      topicConnectionFactory.createTopicConnection(user, psswd);

    topicSession = topicConnection.createTopicSession(false,
          Session.AUTO_ACKNOWLEDGE);

    topicSubscriber = topicSession.createSubscriber(topic);

    topicListener = new TextListener();
    topicSubscriber.setMessageListener(topicListener);

    topicConnection.start(); //!!!

    System.out.println("To end program, enter Q or q, " + "then
          <return>");
    //Read user input, exit when Ctrl+C entered
```

# Exemple JMS: Pub/Sub #2 (5/5)
## *Subscriber - TextListener*

```
import javax.jms.*;

public class TextListener implements MessageListener {

    public void onMessage(Message message) {
        TextMessage msg = null;
        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Reading message: " +
                    msg.getText());
            }
            else { System.out.println("Message of wrong
                type: "
                + message.getClass().getName());
            }
        }
        catch (JMSException e) {
            System.out.println("JMSException in
                onMessage(): " + e.toString());
        }
    }
}
```


# Conclusion

MOM & JMS

# Acteurs et utilisation

## Nombreux produits

- IBM WebsphereMQ (MQSeries)
- SonicMQ (→ Progress Software)
- BEA WebLogic (→ Oracle)
- Microsoft Message Queuing (MSMQ)
- Amazon Simple Queue Service (SQS)
- Joram (→ ScalAgent)
- ~ZeroMQ (pas un MOM)
- …

## Utilisation

- Grands systèmes d'information - utilisateurs 'historiques'
  - ex: banque, assurance
  - Passage à l'échelle nécessaire
- EAI (Enterprise Application Integration)
  - Intégration de systèmes existants
  - Connexion à des bases de données
  - Gestion aisée de plusieurs protocoles
- …

# Conclusion

- Les MOM sont
  - ~ Faciles à implémenter
  - ~ Faciles à déployer
  - ~ Très configurables
  - ~ Passent à l'échelle

- MAIS
  - Restent des boîtes à outils
  - Complexes à utiliser


# Annexes

...

# Autres exemples d'outils pour la communication par message

- Sockets
- MOM
- Autres approches – ex. ZMQ (Zero MQ), Akka, …

# Implémentation minimaliste

- *nc* (netcat) : « TCP/IP swiss army knife »
  - Outil de manipulation des sockets
  - Permet de mettre en place une communication par message en 2 lignes de shell
  - Utilisation des pipes, sockets + langage shell

Sur un terminal :

```
nana.enst.fr$ nc -l 2222
```

Sur un autre :

```
yse.enst.fr$ echo bla | nc nana.enst.fr 2222
```

- Puis perl, awk, .. pour construire/analyser des messages

# ZeroMQ

- Solution intermédiaire – entre:
  - La manipulation directe de sockets
    => flexibilité
  - L'utilisation d'un MOM
    => facilité d'usage mais difficultés de mise en place et coûts en performance
- Librairie qui facilite la manipulation de sockets pour la mise en place d'un système de communication par message
- http://www.zeromq.org

- ...