JMS

Chap #5

Introduction

Le but de ce chapitre est notamment :

- De rappeler les concepts clés de la communication par message au travers de l'API normalisée JMS (Java Messaging Service).
- De montrer les problèmes liés à la diffusion d'information non seulement en point à point mais aussi à un groupe de destinataires.

Ce chapitre abordera également les notions de programmation asynchrone et de qualité de service.

Java Messaging Service

Une API standard

JMS est une API standard de Java EE (Enterprise Edition). Elle permet de fournir à java une implémentation de MOM (Message Oriented Middleware) ou intergiciel de communication par messages, en quelque-sorte un service de mail pour applications.

Toutefois, pour pouvoir utiliser JMS, il faut disposer d'une implémentation externe en Java. Il existe de nombreuses implémentations, tant commerciales que Open Source.

Citons par exemple : WebSphere MQ d'IBM, Sonic MQ de Progress Software mais aussi Fiorano et Swift MQ pour les versions commerciales ; ActiveMQ, Joram, OpenJMS pour les versions Open Source.

Il existe également des utilitaires pour tester la conformité d'une implémentation donnée à l'API officielle : http://jmscts.sourceforge.net/

Java Messaging Service
Une API standard

Nb : notez bien que l'API JMS est définie dans la versions serveur Java EE de Java et non la version standard Java SE.

Il y a eu une remise à plat entre JMS 1.0 et JMS 1.1 afin d'unifier les noms des méthodes et les classes entre les deux modes de communication :

- Par exemple `publish` et `send` sont unifiés en `send`.
- **Ce cours s'appuie sur la version 1.1 de JMS.**

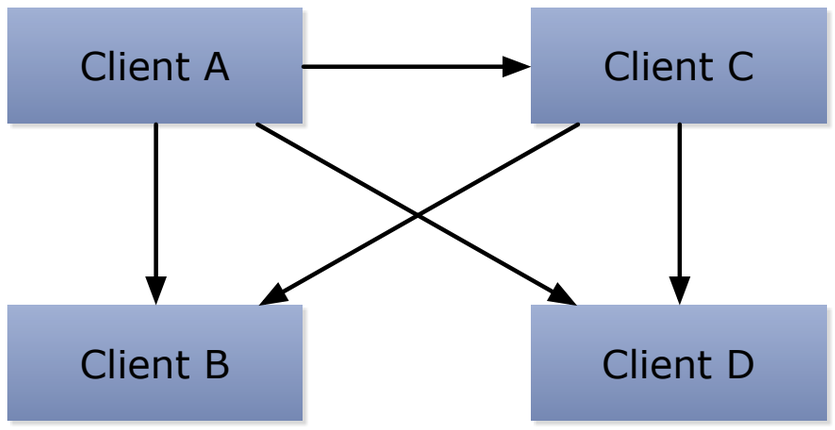
Java Messaging Service
Communication à plusieurs

Alors que dans les technologies RPC telles que RMI ou CORBA il y a une communication synchrone directe entre un objet client et un objet serveur, JMS introduit une couche de médiation entre un ou plusieurs producteurs et un ou plusieurs consommateurs.

- Un producteur de message va créer un message et utiliser une implémentation JMS pour l'envoyer.

- Un consommateur va en contrepartie lire le message (le consommer).

Typiquement, une communication RPC nécessitera de nombreux liens point à point entre clients et serveurs (cf. figure suivante) :



Java Messaging Service
Communication à plusieurs

La couche de médiation JMS permet par contre une communication de un ou plusieurs producteurs vers 0 ou plusieurs consommateurs.

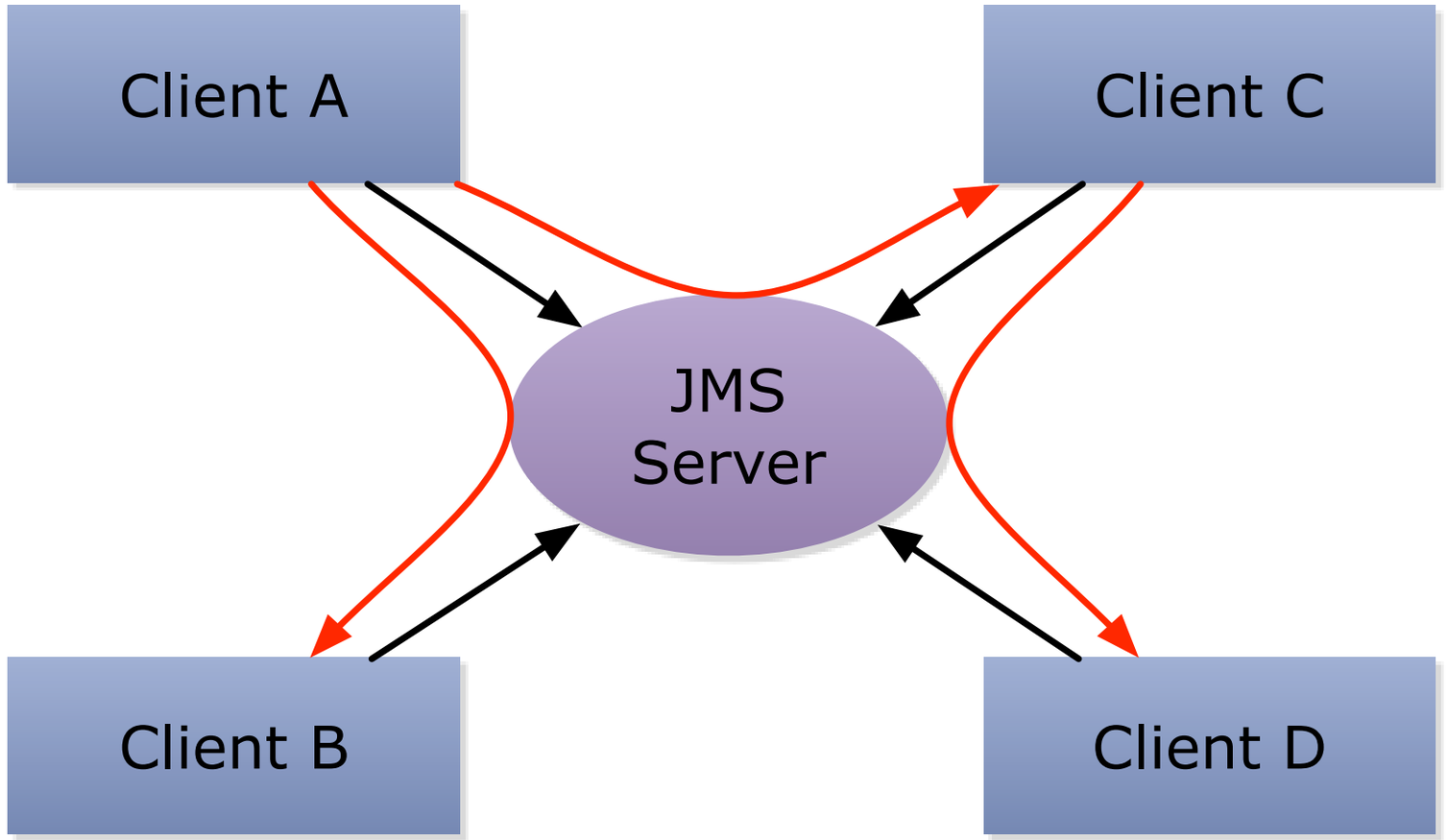
Attention à ne plus parler de clients et de serveurs pour les consommateurs et les producteurs, car en quelque sorte **tous les producteurs et consommateurs sont vus comme des clients par le courtier (“broker”) JMS** (cf. figure suivante) :

- La communication s’établira au travers de deux segments : par exemple entre A et B, A établira une communication avec le courtier JMS, ensuite B fera de même, **à un instant pouvant être différé**.
- Le courtier peut en effet stocker des messages de façon persistante, afin d’assurer une communication **asynchrone** : tout comme lorsque vous envoyez du courrier à un correspondant, celui-ci pourra le lire à une date ultérieure depuis sa boîte aux lettres (BAL).
- Sa BAL sera identifiée par une `Destination` de type `Queue` ou `Topic`, selon que l’on est en mode point à point ou en diffusion.

Java Messaging Service

Communication à plusieurs

Interconnexion entre plusieurs interlocuteurs au travers d'un courtier JMS :



Java Messaging Service
Destinations

Le modèle JMS provient de l'unification de 2 modes de communication :

- **Un mode de communication en point à point** avec un stockage temporaire dans une boîte aux lettre appelée Queue, où un producteur va déposer un message lu ultérieurement par un consommateur.

- **Un mode communication par événements**, ou en mode publication/abonnement ("publish/subscribe"), où 0 ou plusieurs consommateurs écoutent sur un même canal de diffusion appelé `Topic`, alimenté par un ou plusieurs producteurs.

Les destinations de type `Queue` ou `Topic` évitent au producteur et au consommateur de connaître leurs adresses respectives, et elles assurent la transparence de localisation, contrairement aux sockets où l'on doit connaître le port et le nom des machines pour communiquer.

Java Messaging Service
Destinations

Les destinations peuvent être ou non administrés :

- Certaines implémentations nécessitent la création de Queue dans un outil d'administration et non dynamiquement dans un programme.

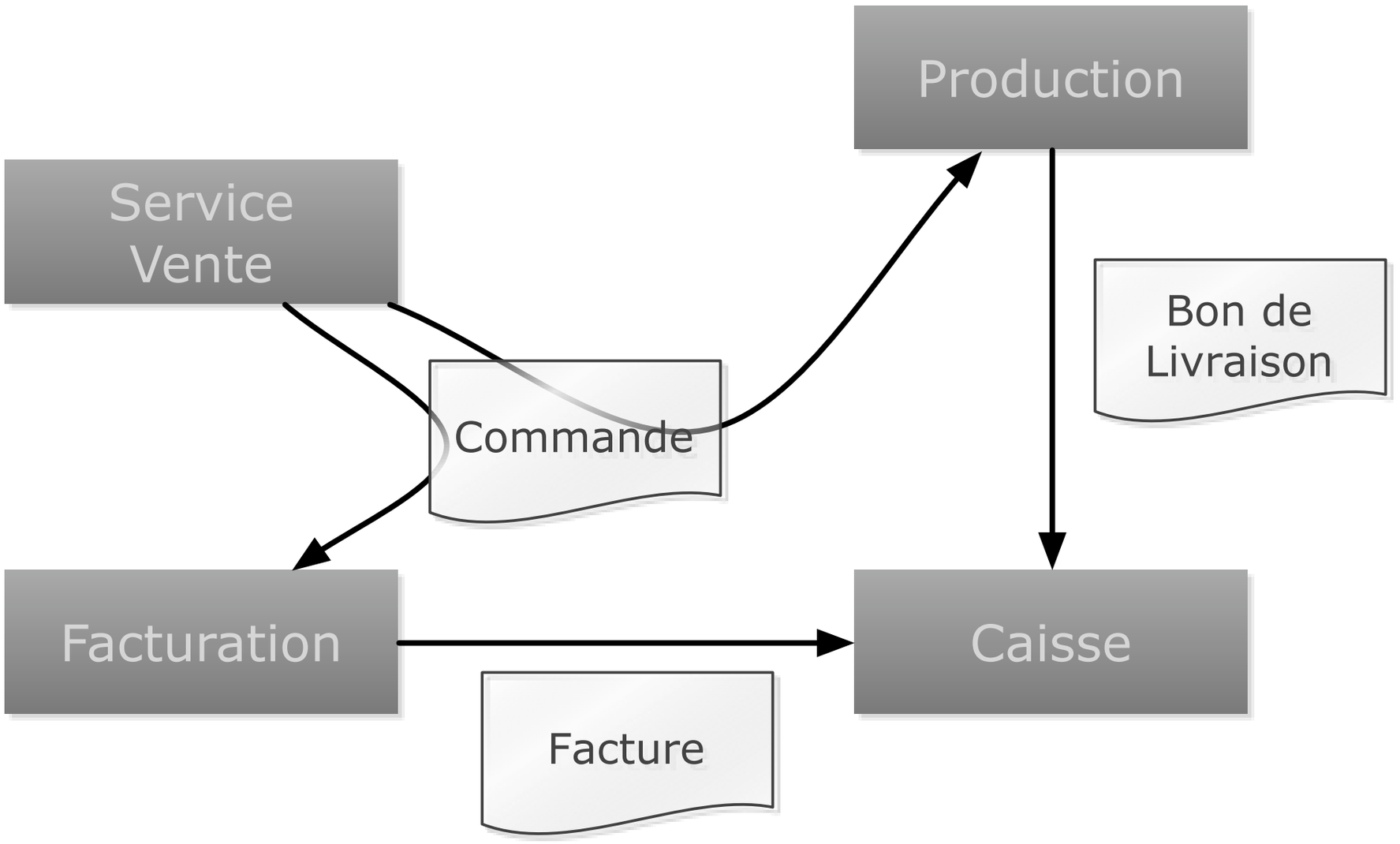
- En effet, les Queue sont souvent liées à une capacité de stockage physique. L'outil d'administration permet alors de consulter l'état de la Queue, le nombre de messages en attente.

- Les Topics peuvent également parfois être administrés.

Le mode de communication en point à point est bien adapté à un workflow d'entreprise où les documents circulent avec un temps de traitement spécifique (cf. figure suivante).

Java Messaging Service
Destinations

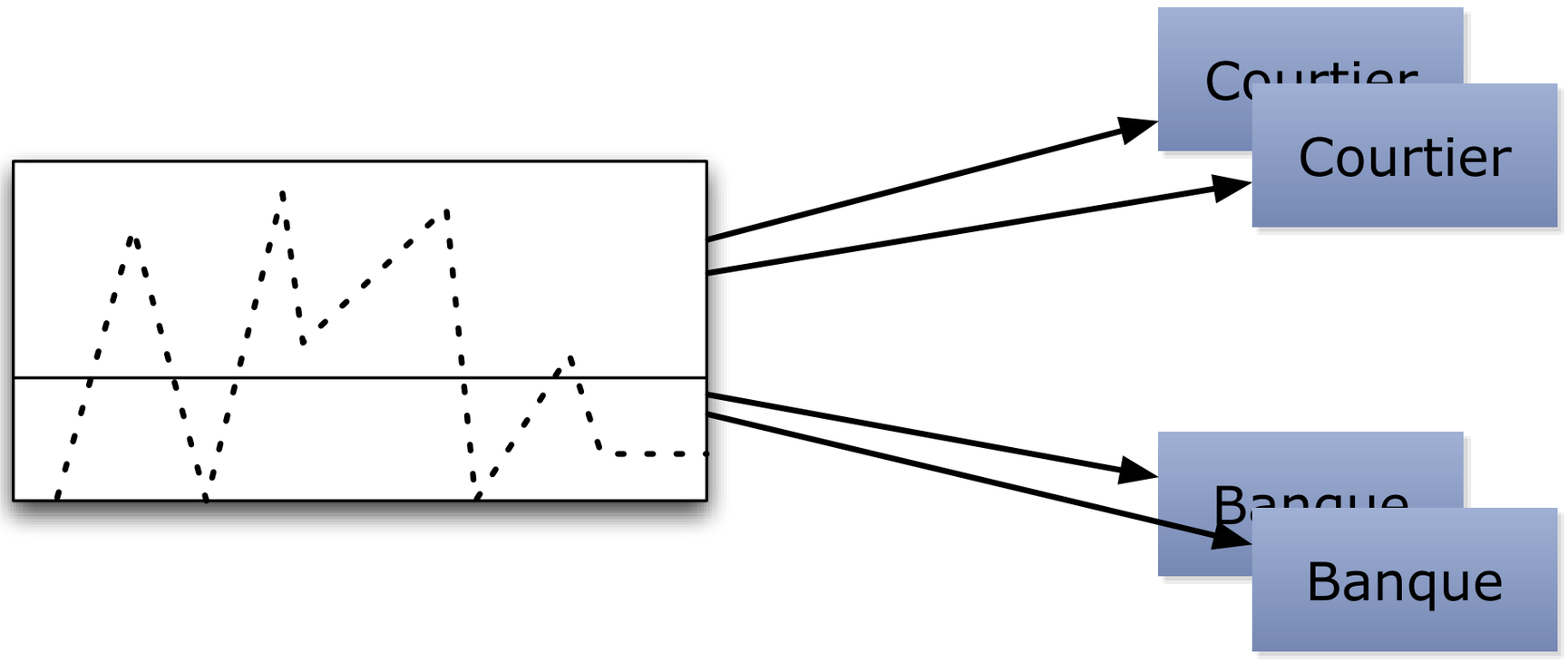
Mode point à point de circulation des documents :



Java Messaging Service
Destinations

Le mode de communication par événement convient au contraire pour diffuser des alertes à plusieurs destinataires (cf. figure suivante), par exemple des informations de bourse à des courtiers, ou bien des anomalies réseau en télécommunications, des points de synchronisation en vidéo, etc...

Diffusion d'alertes de cours de change à la Bourse :

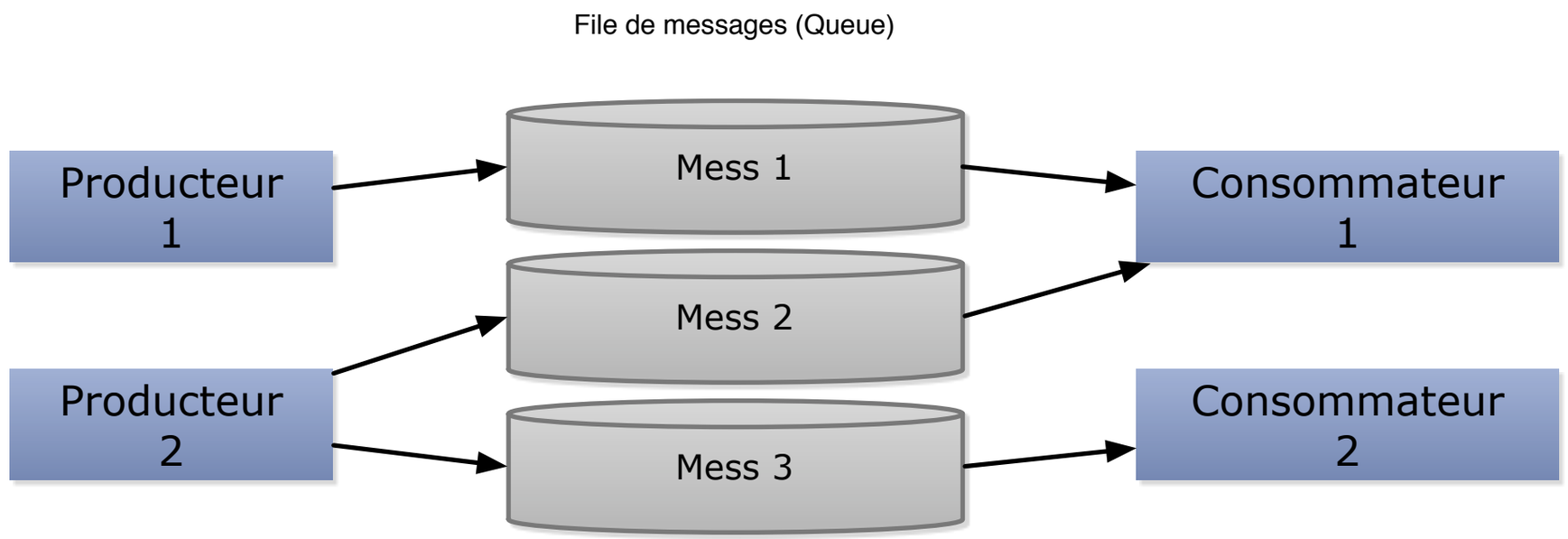


Java Messaging Service
Destinations

La plupart des implémentations de JMS permettent la communication sous les 2 modes, toutefois certaines n'implémentent parfois que le mode `Topic` : elles ne sont alors pas conformes à la spécification JMS.

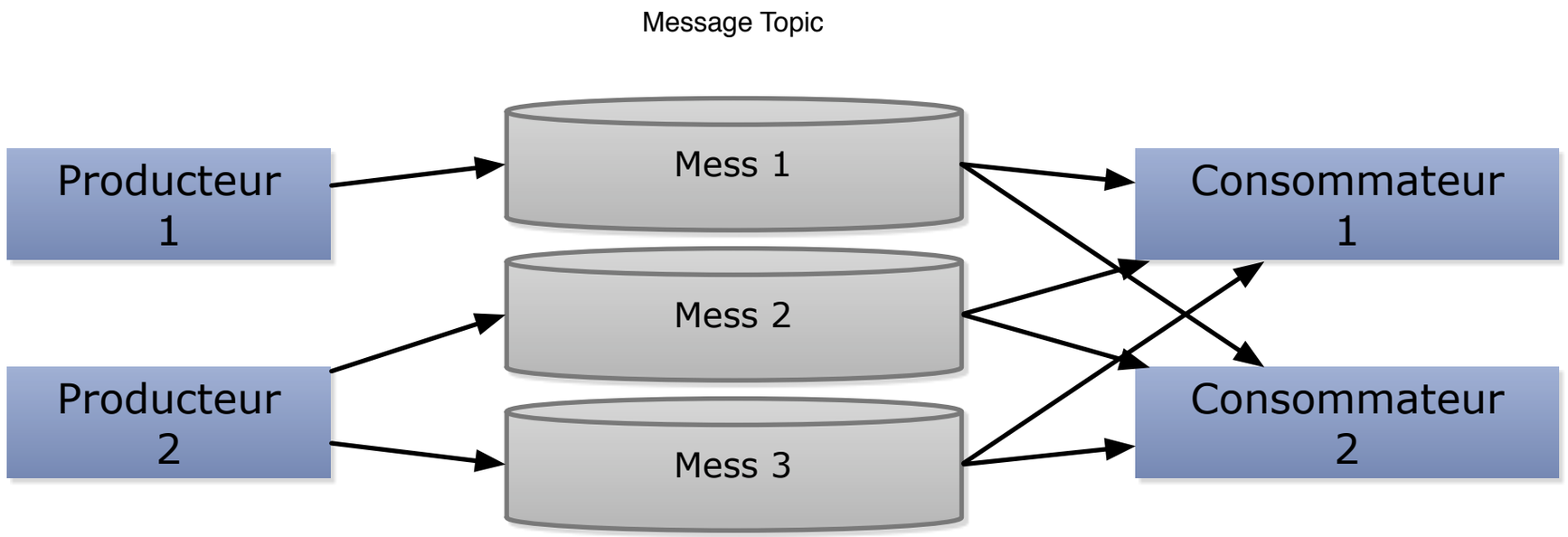
Chaque message individuel n'est produit que par un seul producteur, même si globalement une destination donnée peut être alimentée par plusieurs producteurs.

En **mode point à point**, un message individuel n'est lu que par un seul destinataire consommateur (cf. figure suivante).



Le message peut être stocké pour un certain temps par le courtier, et la lecture du message peut être différée par rapport à sa production. **C'est en ce sens que l'on dira que JMS est un mode de communication asynchrone**. Après lecture, le message est détruit de son lieu de stockage après accusé de réception du consommateur.

Dans un **mode publication/abonnement**, plusieurs destinataires consommateurs écoutent sur le même canal (`Topic`) et reçoivent collectivement le même message individuel (cf. figure suivante).



Java Messaging Service
Destinations

Comme les consommateurs peuvent s'abonner et se désabonner dynamiquement, il n'est pas possible de connaître le nombre d'accusés de réception à attendre. **Le message est donc détruit immédiatement**, sauf dans le cas particulier des abonnés durables que nous verrons par la suite.

- En mode publication/abonnement, un consommateur absent au moment de l'envoi du message ne le recevra donc pas. C'est pour cela qu'il est considéré moins fiable que le mode point à point : c'est la même analogie qu'entre le mode TCP et le mode UDP.

A quel moment exact le message est-il détruit ? C'est ce que nous allons voir avec les concepts de session et de connexion.

Java Messaging Service

Structure des messages

Contrairement aux modèles RPC (CORBA, RMI, SOAP, ...), les consommateurs et le producteurs ne partagent pas d'interfaces et s'échangent uniquement des informations sous forme de messages.

Un message est par définition la composition d'un en-tête, de propriétés et d'un contenu, ce contenu pouvant être de plusieurs natures : texte, binaire ou autre, comme nous le verrons par la suite. C'est pour cette raison que JMS est souvent utilisé pour l'intégration d'applications puisqu'il est moins contraignant sur les définitions des interfaces et s'adapte à l'existant sans modification.

Le message à un rôle ambivalent puisqu'il sert :

- De transport d'information.
- De point de synchronisation entre producteur(s) et consommateur(s) lorsque le consommateur utilise une réception synchrone avec une méthode `receive()`.

Java Messaging Service

Connexion et session

Il existe deux abstractions en JMS qui n'ont pas d'équivalent en mode RPC : la **connexion** et la **session** :

- La connexion est connue des utilisateurs de bases de données et de l'interface JDBC (Java Database Communication), elle permet notamment de gérer la sécurité des accès et le déroulement des transactions vers le gestionnaire de bases de données.

- La session, on le verra, joue un rôle dans la qualité de service propre à JMS.

Java Messaging Service

Connexion = Fiabilité

La connexion assure la fiabilité :

- Lorsque l'on utilise des sockets, un mode connecté précise que le client et le serveur doivent échanger une poignée de main (handshake) avant de rentrer en communication. Ce protocole assure que les deux parties sont disponibles au moment de la communication.

- Si l'un des deux raccroche, la communication est interrompue. C'est ainsi que le protocole TCP qui est en mode connecté est considéré comme plus fiable qu'UDP, où le message peut se perdre si le correspondant a raccroché.

- **La connexion JMS joue exactement ce rôle** : elle assure la fiabilité entre un client JMS et son courtier, et l'on peut récupérer les exceptions si la connexion est défaillante.

Java Messaging Service

Session = Qualité de Service

La connexion assure la Qualité de service :

- Une connexion peut ouvrir plusieurs sessions (cf. figure suivante). Une session est définie à l'intérieur d'une connexion comme une délimitation temporelle permettant de grouper des envois ou des réceptions de messages avec des propriétés spécifiques :

  - mode transactionnel ou non, priorités des messages, séquence des messages, gestion des accusés de réception.

  - C'est ce que l'on appellera la Qualité de Service !

- Naturellement, deux sessions consécutives peuvent définir des modes différents.

En conclusion, dans JMS, la sécurité est gérée par la connexion, les transactions par la session.

Exemple de code
Envoi d'un message

Un message est tout simplement un objet de la classe `Message`. Nous prendrons ici le type de message le plus simple, `TextMessage`, dont le contenu est une chaîne de caractères.

- On peut accéder au contenu d'un `TextMessage` en utilisant les méthodes `getText` et `setText(String)`.

Nb : Les autres types de message seront détaillés plus loin dans ce chapitre.

Pour envoyer un message, il faut disposer d'une `ConnectionFactory`, et connaître d'autre part le nom d'une destination, `Queue` ou `Topic`, (ici `MyQueue`).

- La `ConnectionFactory` utilise le design pattern de la fabrique et permet plusieurs connexions concurrentes. Elle encapsule tous les paramètres d'initialisation de la connexion.

Exemple de code
Envoi d'un message

- La `ConnectionFactory` peut provenir d'un annuaire JNDI ou être crée à partir de paramètres de configuration.

- La destination peut également être stockée dans un annuaire JNDI. Ceci est en particulier utilisé pour les destinations administrées et créées par l'outil d'administration propre à l'implémentation JMS.

La création d'une `ConnectionFactory` est spécifique à une implémentation, ici il s'agit de Sonic MQ. Le paramètre nécessaire est l'adresse du serveur et son numéro de port :

```
tcp://localhost:2506
```

On n'a besoin que d'un seul package pour JMS en plus des bibliothèques liées à l'implémentation :

```
import java.jms.*;
```

Exemple de code
Envoi d'un message

On crée alors la `ConnectionFactory` :

```
String broker = "tcp://localhost:2506";
ConnectionFactory qcf = (ConnectionFactory) new
    progress.message.jclient.ConnectionFactory(broker);
```

La connexion nécessite un nom d'utilisateur et un mot de passe, mails il est possible d'utiliser le login par défaut :

```
Connection connexion = qcf.createConnection("Utilisateur", "MotDePasse");
```

Après la création de la connexion, il faut une session. Ici la session ne comporte pas de transaction (paramètre `false`), et l'acquittement des messages est automatique (`AUTO_ACKNOWLEDGE`).

```
Session session = connexion.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Exemple de code
Envoi d'un message

La Queue peut avoir été créée dans le courtier avant utilisation, cela dépend des implémentations. On peut également avoir des destinations temporaires liées à la session. **L'instruction suivante donne seulement une référence sur cet espace de stockage créé à l'avance par un outil d'administration propre à l'implémentation**. Ici la destination a pour nom `MyQueue`.

```
Queue queue = session.createQueue("MyQueue");
```

On crée ensuite le producteur et le message.
Nb : Le producteur et le message n'existent que dans la session qui les crée.

```
MessageProducer sender = session.createProducer(queue);
TextMessage mess = session.createTextMessage("Hello");
sender.send(mess);
```

Il ne faut ensuite pas oublier de fermer la connexion. Ceci ferme également la session et déclenche tous les messages de gestion d'accusés réception :

```
session.close();
connexion.close();
```

Exemple de code
Envoi d'un message

Le code est exactement le même en mode publication/abonnement. La seule différence est le type de destination : ici un `Topic`.

Les Topic sont le plus souvent créés dynamiquement, contrairement aux Queue, car dans le mode usuel il n'y a pas d'acquittement des Topic, et donc pas de persistance des messages et de réservation de stockage.

```
Topic topic = session.createTopic("MonTopic");
```

La symétrie entre `Topic` et `Queue` est identique dans le code de réception de la section suivante.

Exemple de code

Réception synchrone d'un message

Pour la réception d'un message synchrone, la création d'une connexion et d'une session sont identiques, de même que la création de la référence à la Destination.

Toutefois, la session va cette fois créer un consommateur (`Consumer`) au lieu d'un producteur (`Producer`).

Nb : On peut créer dans la même session des producteurs et des consommateurs.

```
Queue queue = session.createQueue("MyQueue");
//Pour un Topic, on aurait de même createTopic(...)

MessageConsumer receiver = session.createConsumer(queue);
//Ne pas oublier de demarrer la connexion pour l'attete de reception de messages
connexion.start();

//Reception d'un message de type texte
TextMessage mess = (TextMessage) receiver.receive();
System.out.println("Message recu : " + mess.getText());}
//Fermeture de la connexion et, par là même, de la session
connexion.close();
```

Exemple de code

Réception synchrone d'un message

Ne pas oublier de démarrer la connexion (start()) pour se mettre en attente active de réception des messages.

Ne pas oublier de fermer la connexion, afin de libérer des ressources. Le programme restera en attente tant que la connexion n'a pas été fermée. Certaines implémentations nécessitent de fermer la session avant la connexion.

La commande de réception receive est bloquante : si il n'y a pas de message, le consommateur sera bloqué" en attente indéfiniment. Si l'on souhaite éviter ce blocage, on peut utiliser :

- `ReceiveNoWait()` : renvoie le premier message dans la Queue si il existe, sinon renvoie null.

- `Receive(long timeout)` : attend un message pendant l'intervalle de temps `timeout`, sinon renvoi `null`.

Exemple de code

Réception asynchrone d'un message

Pour une réception de message utilisant `receive`, le consommateur est bloqué en attente de réception (cf. figure suivante). En ce sens, on parle de réception synchrone.

Le terme est ambivalent, car on qualifiera JMS de mode de communication asynchrone entre producteurs et consommateurs, car le moment d'émission du message peut être totalement découplé du moment de réception du message, tout comme pour une lettre traditionnelle.

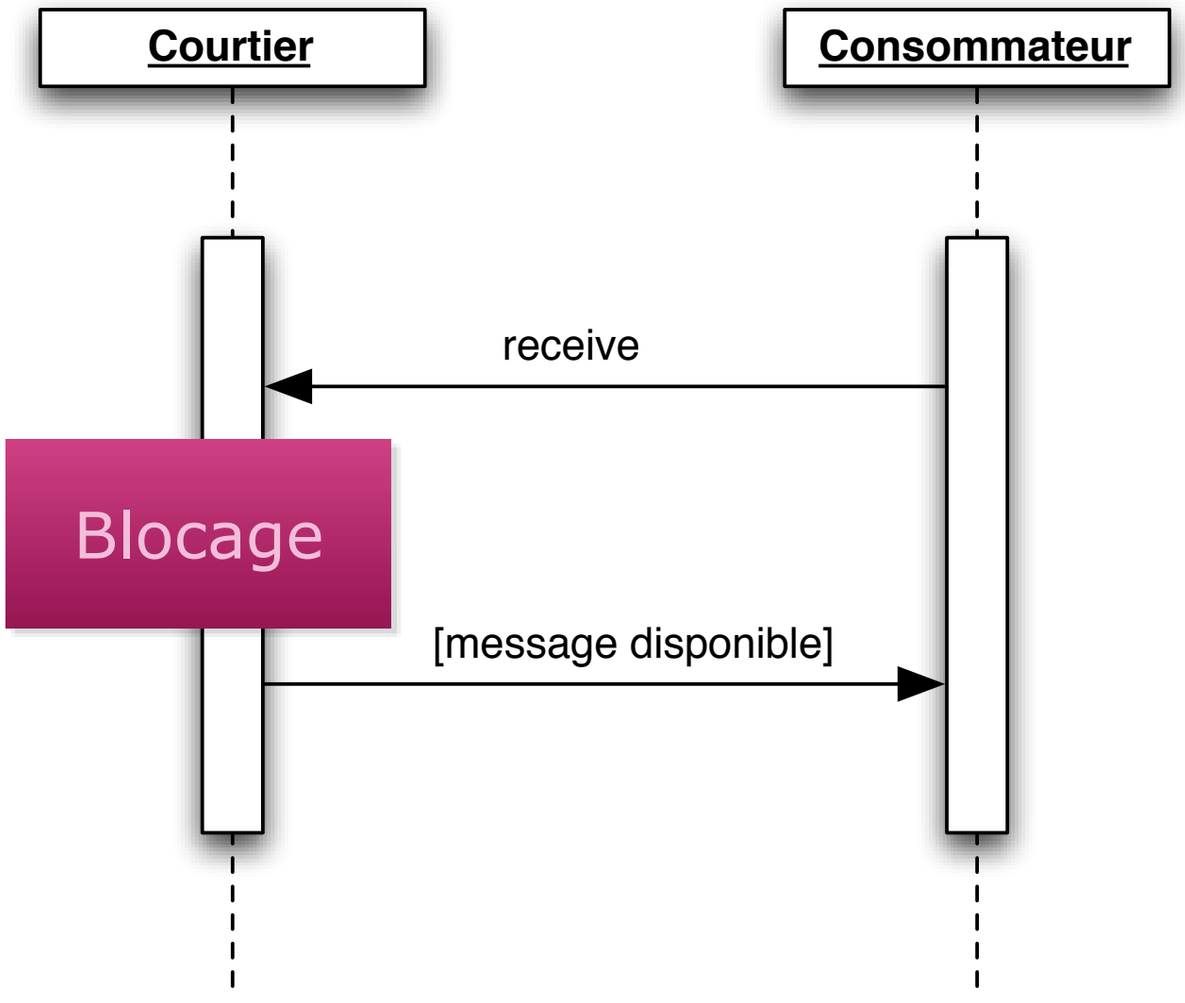
La confusion provient du fait qu'une communication par JMS comporte 2 segments distincts : l'un entre l'émetteur et le courtier, l'autre entre le courtier et le récepteur.

➡ **Globalement la communication est asynchrone dans JMS, mais le segment du récepteur peut être synchrone ou asynchrone.**

Exemple de code

Réception asynchrone d'un message

Le consommateur est bloqué en attente sur un `receive` tant qu'il n'y a pas de message :



Exemple de code

Réception asynchrone d'un message

L'attente bloquante peut détériorer les performances, ou générer des interblocages :

- Un interblocage survient lorsque 2 interlocuteurs sont chacun en attente d'un message de l'autre.
- Par ex. si A attend un message de B pour lui envoyer une réponse, et B attend également un message de A pour lui envoyer une réponse.

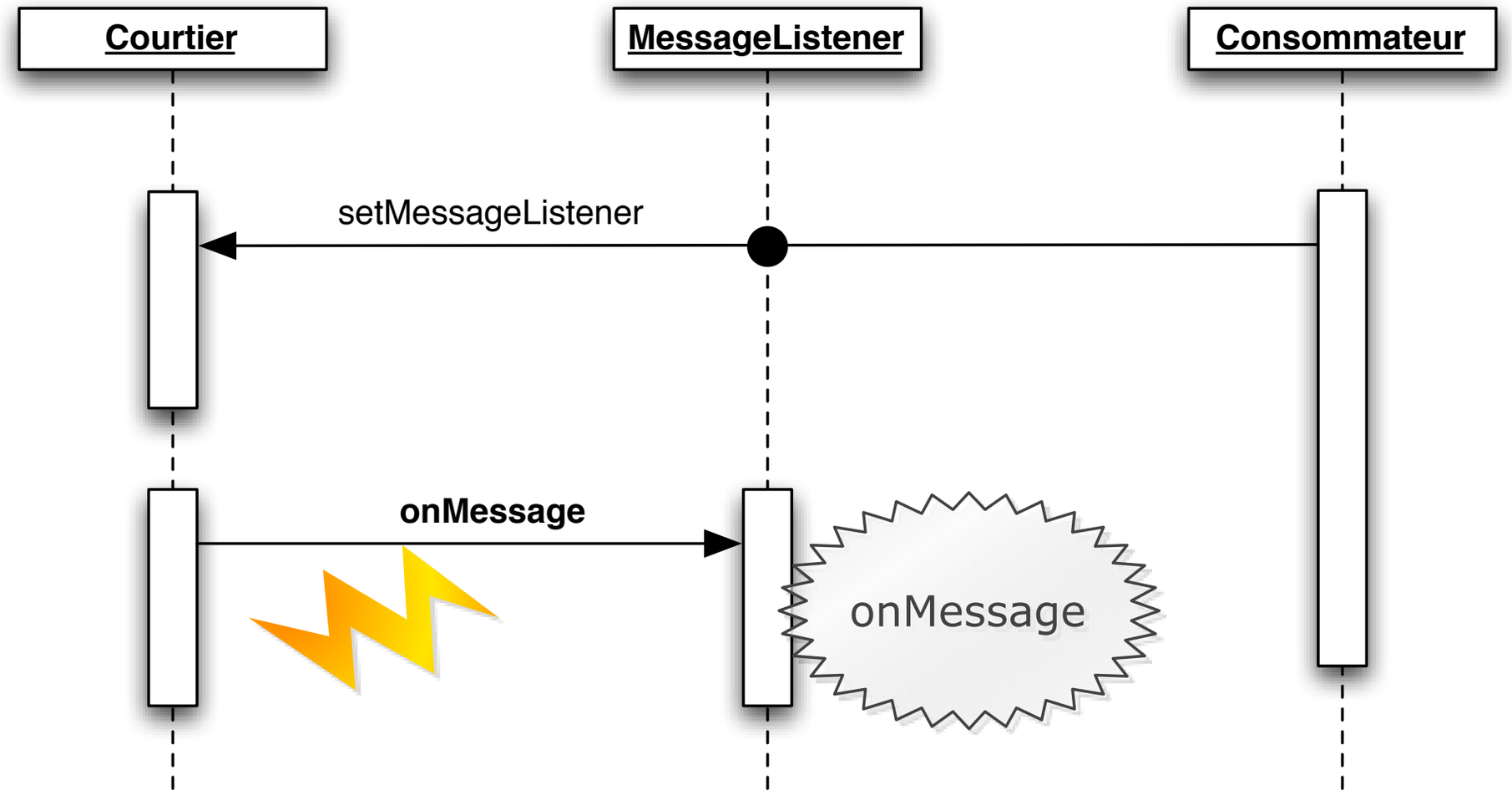
De ce fait, JMS permet aussi de faire de la réception asynchrone de message en utilisant le mécanisme classique en Java des `Listeners` (design pattern Observateur).

- Un `MessageListener` créera une thread permettant de gérer la réception du message sur disponibilité de celui-ci (cf. figure suivante).
- Le `MessageListener` sera attaché au consommateur mais peut ou non être une classe différente de celle qui abrite le consommateur. Il doit implémenter la méthode `onMessage(Message)`.

Exemple de code

Réception asynchrone d'un message

L'arrivée d'un message en mode de réception asynchrone déclenche l'appel de la méthode `onMessage` :



Exemple de code

Réception asynchrone d'un message

Cette technique évite le *polling* et améliore les performances :

- Le *polling* signifie que l'on va régulièrement scruter l'arrivée d'une donnée.
- C'est un mécanisme classique sur les drivers par exemple, mais très gourmand en temps :
  - soit l'intervalle de scrutation est faible et l'on perd du temps à vérifier une donnée absente,
  - soit l'intervalle est long et on perd la donnée.
- Par contre, cette technique asynchrone génère une thread de réception pour la méthode `onMessage` et nécessite de faire attention à l'accès des variables partagées au thread de réception et au thread principal (cf. cours sur les problématiques de synchronisation).

Exemple de code

Réception asynchrone d'un message

Dans le code suivant, on suppose que la classe qui lance la réception implémente également la méthode `onMessage` définie dans l'interface `MessageListener` :

```
Queue queue = session.createQueue("MyQueue");
MessageConsumer receiver = session.createConsummer(queue);

//On doit attacher le MessageListener avant de démarrer la connexion
receiver.setMessageListener(this);
connexion.start();

//Le code suivant permet de vérifier quelle thread est utilisée
System.out.println("Thread lancement" + Thread.currentThread().getId());

//Code pour attente bloquante permettant de ne pas fermer la connexion
```

Le code suivant sera déclenché dans le `MessageListener` sur réception du message :

```
public void onMessage(Message message) {
   //Affiche le contenu du message
   System.out.println(((TextMessage) message).getText());
   //Vérifie que la thread utilisée est bien différente de celle de lancement
   System.out.println("Thread onMessage" + Thread.currentThread().getId());
}
```

Exemple de code

Réception asynchrone d'un message

Attention :

- Une session ne peut comporter que des consommateurs synchrones ou que des consommateurs asynchrones dotés de `MessageListener`.

- Il n'est pas possible de mélanger deux modes de réception dans la même session : on créera alors deux sessions.

Exemple de code

Retour sur les connexions et sessions

Le mécanisme de création de connexion et de session est parfaitement identique pour la création et la réception de messages (cf. figure suivante). Il suppose deux points d'entrée :

- La fabrique de connexions (`ConnectionFactory`).
- Le nom de la destination (`Queue` ou `Topic`).

La `ConnectionFactory` crée une connexion qui crée à son tour une ou plusieurs sessions.

Un `MessageProducer` ou un `MessageConsumer` n'a d'existence qu'à l'intérieur d'une session. Il est spécifique à une destination. On voit simplement que pour plusieurs `Topic` d'intérêt on doit créer plusieurs consommateurs (resp. producteurs), car les consommateurs sont liés à une seule destination.

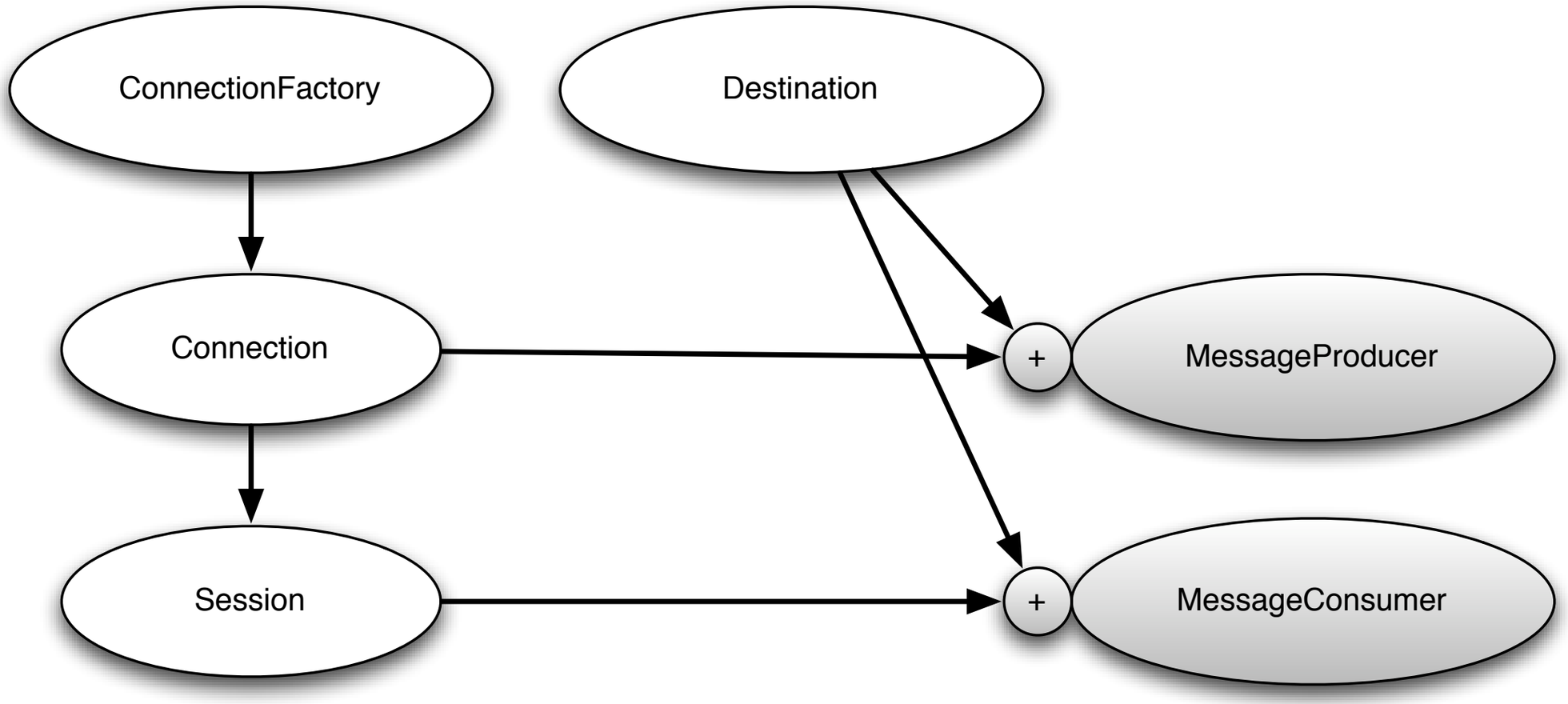
Un consommateur peut naturellement recevoir plusieurs messages, de même qu'un producteur peut en produire plusieurs dans la même session.

Une session peut créer plusieurs consommateurs et plusieurs producteurs.

Exemple de code

Retour sur les connexions et sessions

Deux points d'entrée dans le système JMS : les destinations et les fabriques de connexions :



Exemple de code

Retour sur les connexions et sessions

Pourquoi utiliser plusieurs connexions ?

- Une connexion est liée à une machine et un port sur lequel écoute le courtier. L'utilisation de plusieurs connexions permet de se connecter à plusieurs courtiers, par exemple pour faire de la redondance.

- Une connexion est également attachée à une identification, typiquement utilisateur et mot de passe. Les implémentations de JMS les plus complètes permettent de restreindre l'accès à certaines ressources telles que des `Topic` ou des `Queue` pour certains utilisateurs. On utilisera éventuellement plusieurs connexions pour différents utilisateurs.

Exemple de code

Retour sur les connexions et sessions

Pourquoi utiliser une ConnectionFactory ?

- Le concept de fabrique (`Factory`) est usuel en Java. Il permet dans notre cas de créer des connexions personnalisées, avec des paramètres prédéfinis tels que l'adresse du courtier ou l'utilisateur à utiliser.

- De plus, on stocke cette fabrique d'ordinaire dans un annuaire d'entreprise ou dans un annuaire applicatif au travers de l'interface JNDI qui a été présentée dans ce cours. La fabrique permet alors aux clients JMS de disposer d'une configuration pré-établie puisque la fabrique contient les paramètres d'initialisation.

- Voici un exemple de code utilisant l'implémentation JMS Joram pour obtenir une connexion préconfigurée en accédant à l'annuaire propriétaire de Joram sur le port `16400` en local :

Exemple de code

Retour sur les connexions et sessions

Pourquoi utiliser une ConnectionFactory ?

```
Properties properties = System.getProperties();
properties.put(“java.naming.factory.initial”,
“fr.dyade.aaa.jndi2.client.NamingContextFactory”);
properties.put(“java.naming.factory.host”, “localhost”);
properties.put(“java.naming.factory.port”, “16400”);

InitialContext jndi = new InitialContext();
ConnectionFactory fabriqueDeConnexion = (ConnectionFactory) jndi.lookup(“fabrique”);
Connection connexion = fabriqueDeConnexion.createConnection();
```

Exemple de code

Retour sur les connexions et sessions

Pourquoi utiliser une ConnectionFactory ?

- L'API JMS précise que les `ConnectionFactory` doivent être `Referenceable` et `Serializable` afin de pouvoir les stocker dans un annuaire JNDI. Ce ne sont toutefois pas les seuls objets à figurer dans l'annuaire : les `Queue` et `Topic` peuvent également être découverts ou configurés dans un annuaire. Ainsi lors du déploiement d'une application, un client pourra obtenir, dans un annuaire, la liste des destinations qu'il doit adresser, sans configuration locale et statique.

- L'utilisation de JNDI est également utile pour permettre d'interchanger les implémentations de JMS : en effet, comme la `ConnectionFactory` comporte tous les paramètres d'initialisation, l'obtention d'un tel objet permet un code identique quelle que soit l'implémentation :

```
ConnectionFactory factory = contexte.lookup(“reference”);
(...)
```

Exemple de code

Retour sur les connexions et sessions

Pourquoi utiliser plusieurs sessions ?

- La session gère les paramètres de qualité de service des messages, ainsi que les Queues temporaires, les transactions, l'ordre chronologique des messages, la réception synchrone ou asynchrone.

- Pour les transactions, la session servira de délimitation, elle permettra d'assurer l'ordre des messages à l'intérieur d'une même session. Elle assurera également la gestion des accusés de réception des messages.

- Des sessions trop longues peuvent typiquement générer des attentes d'accusés de réception des messages et détériorer les performances. Par contre il est important de comprendre qu'une session peut comporter à la fois des producteurs et des consommateurs de messages, qui peuvent envoyer un nombre quelconque de messages.

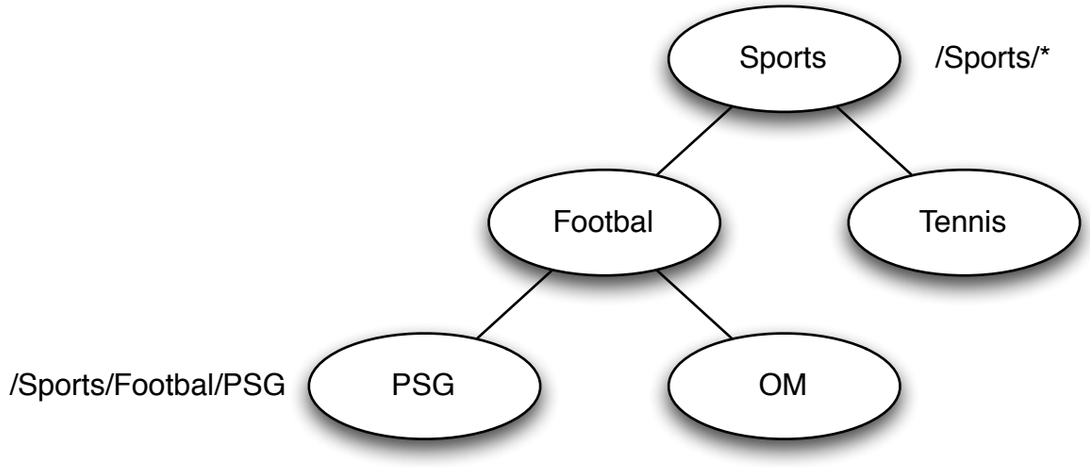
Exemple de code

Utilisation de hiérarchies de Topic

La plupart des implémentations JMS offrent des hiérarchies de Topic comme le montre la figure suivante. Toutefois, la syntaxe de filtrage des Topic ou la création de l'arborescence diffère.

En supposant que le filtrage se fait par des expressions régulières, il est ainsi possible de s'abonner à /Sports/* pour recevoir les Topic fils : Sports, Footbal, Tennis, PSG, OM.

Ceci peut être utile pour faire des diffusions plus ou moins ciblées, ou des communications de groupes : si chaque application écoute sur un Topic particulier, on peut les regrouper pour faire une diffusion générale.



Nature des messages

Un message JMS est constitué de plusieurs parties :

- **Le corps du message** qui véhicule les informations.
- **Les propriétés du message** (une liste de paires attribut/valeur).
- **L'en-tête**, utilisé pour les informations de qualité de service et le routage, ainsi que le transport des propriétés associés au message.

Nature des messages

Le corps du message

JMS fournit plusieurs natures de messages, permettant différents codages des informations (marshalling) : `TextMessage`, `BytesMessage`, `MapMessage`, `StreamMessage`, `ObjectMessage`.

`TextMessage` :

- Un `TextMessage` transporte du texte et peut utiliser les nouvelles classes d'encodage de caractère `java.nio` dans toutes les langues. Un message de texte peut également transporter un contenu au format XML. Ceci permet par exemple à JMS de servir de protocole sous-jacent à SOAP. Ceci permet également d'accorder la syntaxe du message entre les deux correspondants en utilisant par exemple un schéma XML pour vérifier la conformité de la syntaxe du texte.

- Création d'un message par la session - la chaînes de caractères est passée en argument :

```
TextMessage messageTexte = session.createTextMessage(“hello”);
```

Nature des messages

Le corps du message

- Réception d'un message

```
TextMessage messageTexte = (TextMessage) receiver.receive();
System.out.println(messageTexte.getText());
```

- On remarquera qu'on ignore la nature d'un message à réception, et que l'on ne peut filtrer sur la nature du message. Il faut donc concevoir l'application afin d'utiliser toujours le même type de message pour une destination donnée ou prévoir un aiguillage à l'aide de `instanceof`.

`BytesMessage` :

- Un `BytesMessage` transporte des données binaires. C'est utile pour les images, les fichiers binaires, et de manière générale tous les attachements que l'on connaît pour les emails.

- En effet, JMS peut être **considéré** comme un système d'échange d'emails entre applications et non entre personnes.

Nature des messages

Le corps du message

- On peut également transporter des messages encodés dans tout système de format binaire, tels que ASN1 très utilisés en télécommunications et en cryptographie (il est plus compact que XML).

- Exemple d'utilisation de `BytesMessage` :

```
BytesMessage outMessage = session.createBytesMessage();
byte[] buffer = readFile("photo.jpg");
outMessage.writeBytes(buffer);
```

- Le problème ici est que l'on ne connaît pas forcément la taille du buffer envoyé ! On peut y remédier en mettant une propriété dans l'en-tête du message. Les propriétés seront détaillées au paragraphe suivant.

```
outMessage.setIntProperty("size", buffer.length);
```

- Le récepteur du message pourra alors allouer le tampon de la taille voulue :

Nature des messages

Le corps du message


```
Message message = consumer.receive();
//On teste le type de message
if(message instanceof BytesMessage) {

   //On récupère la taille dans une propriété
   int size = inMessage.getIntProperty(“size”);

   //On alloue un tampon adapté
   byte[] replyBytes = new byte[size];

   ((BytesMessage) message).readBytes(replyBytes)
   (...)
}
```

`MapMessage` :

- Un `MapMessage` est utilisé pour envoyer un dictionnaire de clés et de valeurs. Il se marie facilement avec un `ResultSet` provenant d’un base de données, et fournit les mêmes types natifs (`int`, `float`, etc...).

```
MapMessage message = session.createMapMessage();
message.setInt(“Quantity”, 4);
message.setStringProperty(“itemType”, “Screen”);
```

Nature des messages

Le corps du message

En conclusion, JMS permet une grande variété de formats d'échange et ne préjuge en rien sur la manière dont ces données seront exploitées.

Chaque client JMS peut modifier aisément le contenu sans recompilation, et sans adhérer à une syntaxe stricte d'interface. C'est un des succès de JMS puisqu'il permet à deux applications de communiquer avec un minimum de modifications.

En contrepartie, il reste à la charge du développeur de s'assurer que les types de messages échangés sont cohérents, ainsi que la syntaxe de leur contenu : aucune erreur ne pourra être détectée à la compilation. On peut imaginer d'automatiser ce processus en définissant des schémas XML, par exemple pour valider le contenu texte d'un message.

On notera également que l'analyse syntaxique (parsing) d'un message trop long peut être très pénalisante en terme de performances. Enfin, l'encodage du texte en chaînes de caractères sur 16 bits peut être pénalisent en terme de bande passante, tout comme le protocol SOAP (cours suivant).

Nature des messages

Propriétés : filtrage et routage

L'en-tête d'un message peut contenir des propriétés qui peuvent être utilisées pour filtrer les messages ou les re-router vers d'autres destinataires.

Les propriétés sont des couples (nom, valeur) utilisables par le système ou l'application pour sélectionner les messages ou les destinataires. Ceci convient naturellement à la fois aux `Queues` et aux `Topic`.

Les propriétés peuvent être des types natifs Java comme `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` et `String`.

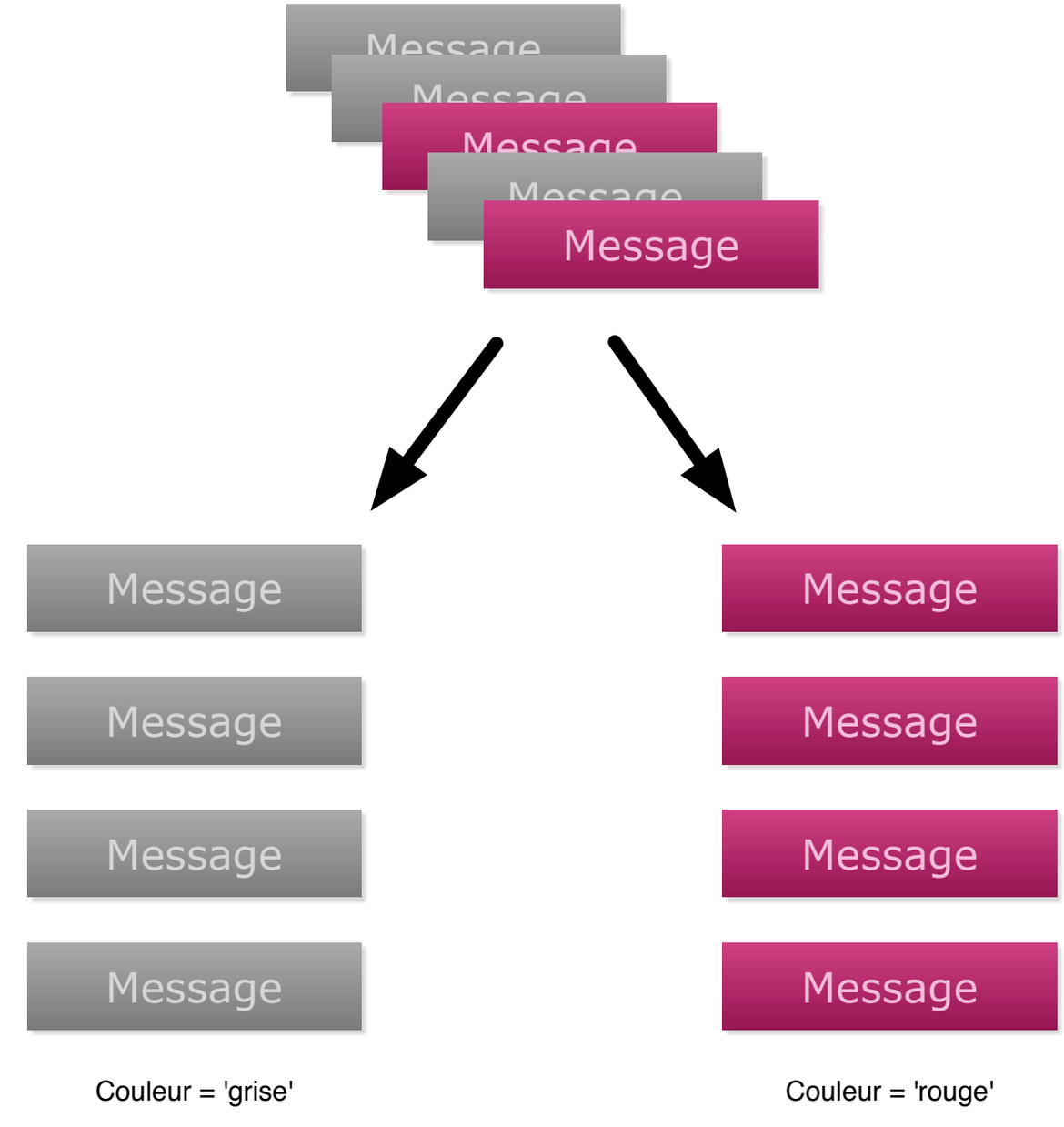
Le filtrage des messages par propriété (cf. figure suivante) permet de ne pas avoir à ouvrir le contenu d'un message avant de le router, et ainsi de ne pas faire jouer les accusés de réception. Il s'appuie sur une syntaxe proche de SQL 92 afin de définir des expressions booléennes sur les propriétés :

```
(Country = 'UK') OR (Country = 'US') OR (Country = 'FR')
age >= 15 AND age <= 19
```

Nature des messages

Propriétés : filtrage et routage

Filtrage des messages par propriétés :



Nature des messages

Propriétés : filtrage et routage

Attention : les chaînes sont notées en SQL avec de simples quotes : `'chaine'`.

Envoi du message :

```
TextMessage message = session.createText("bonjour");
message.setStringProperty("couleur", "grise");
publisher.send(message);
```

A l'arrivée, le filtrage est attaché au consommateur, et non sur la méthode de réception :

```
MessageConsumer receiver = session.createConsumer(topic, "couleur='grise'");
connexion.start();
TextMessage mess = (TextMessage) receiver.receive();
```

On peut également utiliser la réception asynchrone en attachant un `MessageListener` au consommateur filtré. Par contre dans tous les cas, le filtre n'est plus modifiable après création du consommateur.

De nombreux usages découlent du filtrage : le routage dans un centre d'appels en fonction du motif d'appel, le routage d'information de bourse en fonction de l'importance du message, etc...

Nature des messages

En-tête des messages et QoS

L'en-tête du message véhicule des informations de Qualité de Service (QoS). Toutefois rien n'oblige une implémentation à mettre en oeuvre tous ces aspects.

De plus, selon les implémentations les différents niveaux de définition seront implémentés ou non :

- lors de l'envoi du message :

```
producer.send( Message message,
               int deliveryMode, //persistant ou non
               int priority,     //priorité
               long timeToLive   //durée de vie
```

- lors de la création du message :

```
message.setDeliveryMode(DeliveryMode.PERSISTENT);
```

- ou attachés au producteur pour tous ses messages :

```
producer.setTimeToLive(10000);
```

Nature des messages

En-tête des messages et QoS

On trouve plusieurs attributs dans l'en-tête du message (méthode `getXXX` du message) :

Champ d'en-tête	Fonction	Défini
JMSDestination	La Queue ou le Topic.	Automatiquement lors du send.
JMSDeliveryMode	Définit si le message est persistant ou non.	Renseigné par code.
JMSExpiration	Définit le temps de rétention d'un message persistant.	Renseigné par code lors du send.
JMSMessageID	Identifiant unique du message.	Généré par le courtier.
JMSPriority	Priorité du message.	Permet de traiter certains messages en priorité. N'a qu'une portée sur la session.
JMSTimestamp	Estampille.	Généré par le courtier.
JMSCorrelationID	Corrélation entre deux messages. Utilisé pour les requêtes-réponses.	Par code.
JMSReplyTo	Champ optionnel permettant de donner une adresse de retour.	Par code.
JMSType	Nature du contenu du message (Text, Bytes, etc.)	Courtier.
JMSRedelivered	Indique s'il s'agit d'une nouvelle tentative d'envoi sur échec d'acquittement.	Courtier.

Nature des messages

En-tête des messages et QoS

Le mode de livraison (`deliveryMode`) peut être persistant ou non : `DeliveryMode.PERSISTANT` ou `NON_PERSISTANT`. En pratique, les messages seront quasiment toujours persistants, par contre on pourra jouer sur leur durée de vie (`TimeToLive`) exprimée en milliseconde (ms). Une durée de vie de `0` indique par convention que le message n'expirera jamais.

Si l'envoi d'un message persistant échoue, celui-ci sera renvoyé dans le mode point à point ou le mode publication/abonnement avec abonnement durable. Ainsi on peut être sur qu'un message persistant ne sera supprimé que sur accusé de réception du consommateur.

Le renvoi du message peut être détecté en inspectant le champ `JMSRedelivered`. En effet, JMS assure la livraison du message en réitérant celle-ci sur erreur.

Fiabilité de la communication

Une des motivations de l'utilisation de JMS va être la fiabilité du transport des informations.

Fiabilité de la communication

Abonnés durables

Lors de la réception d'un message **en mode publication/abonnement**, le client doit être présent, il ne sera pas livré de nouveau. En effet, comme le nombre de destinataires est dynamique et non connu à l'avance, il est impossible de savoir si tout le monde a reçu le message et de renvoyer celui-ci si cela n'est pas le cas.

Pour fiabiliser la réception des messages et être sûr que tous les clients qui le souhaitent on reçu le message, ceux-ci doivent souscrire un abonnement durable (durable subscribers). Cet abonnement permettra au courtier de s'assurer qu'ils ont reçu le message par acquittement, et de leur renvoyer si ce n'est pas le cas.

Pour cela, il suffit d'une part de donner un identifiant à la connexion, d'autre part de créer le consommateur en spécifiant un nom d'abonnement :

Fiabilité de la communication

Abonnés durables


```
ConnectionFactory qcf = ... // selon implémentation
Connection qc = qcf.createConnection(“Utilisateur”, “MotDePasse”);
qc.setClientID(“MonAppliJMS”);
Session s = qc.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic topic = s.createTopic(“Developpement.Durable”);
MessageConsumer receiver = s.createDurableSubscriber(topic,”Abonnement1”);
receiver.setMessageListener(new Receiver());
qc.start();
```

On vérifiera alors que si le message est persistant et envoyé en mode publication/abonnement, le client pourra le recevoir ultérieurement sur reconnexion.

Pour supprimer l’abonnement, il faut se connecter avec le même `clientID`, et utiliser la méthode `unsubscribe` avec le nom d’abonnement :

```
qc.setClientID(“MonAppliJMS”);
(...)
s.unsubscribe(“Abonnement1”);
```

Fiabilité de la communication

Séquence des messages

Les messages transitent par un nombre quelconque de courtiers. De plus, il peuvent être associés à des priorités diverses, et le courtier peut être plus ou moins chargé.

Les implémentations vont autant que possible respecter la chronologie des messages, mais certains ordinateurs peuvent avoir des horloges non synchronisées, etc... Il est donc possible qu'en bout de chaîne il y ait une inversion de l'ordre des messages. Il est important dans la conception d'envisager le cas d'inversion des messages et soit de s'assurer que cela n'a pas d'impact soit d'y remédier.

L'inversion des messages ne peut arriver dans un mode RPC pour un même producteur, car il doit attendre le retour de son appel avant d'effectuer le suivant.

Fiabilité de la communication

Acquittement des messages

Envoi d'un message une fois et une seule :

- Dans certaines applications il est très important que le message soit envoyé une fois et une seule, par exemple pour un retrait bancaire.

  - Si lors d'un retrait d'argent, le serveur de banque renvoie une erreur de connexion, il peut être impossible de savoir si elle a eu lieu avant ou après la prise en compte de la transaction.

  - Si on renvoi le message de débit, cela peut être au détriment de l'utilisateur qui sera furieux d'être débité deux fois, et si on ne le renvoie pas, cela peut porter préjudice à la banque.

  - Obtenir que le message soit envoyé une, et une seule, fois peut être très important. **Pour cela on utilisera des mécanismes sophistiqués d'acquittement. (cf. ci-après).**

Fiabilité de la communication

Acquittement des messages

Acquittement gérés par le système :

- Une des manières d'assurer la fiabilité est de pouvoir gérer les accusés de réception. **Ils ne sont gérés que pour les Queue ou pour les Topic avec des abonnés durables**. Par défaut on utilise l'attribut `Session.AUTO_ACKNOWLEDGE`, mais il existe deux autres possibilités.

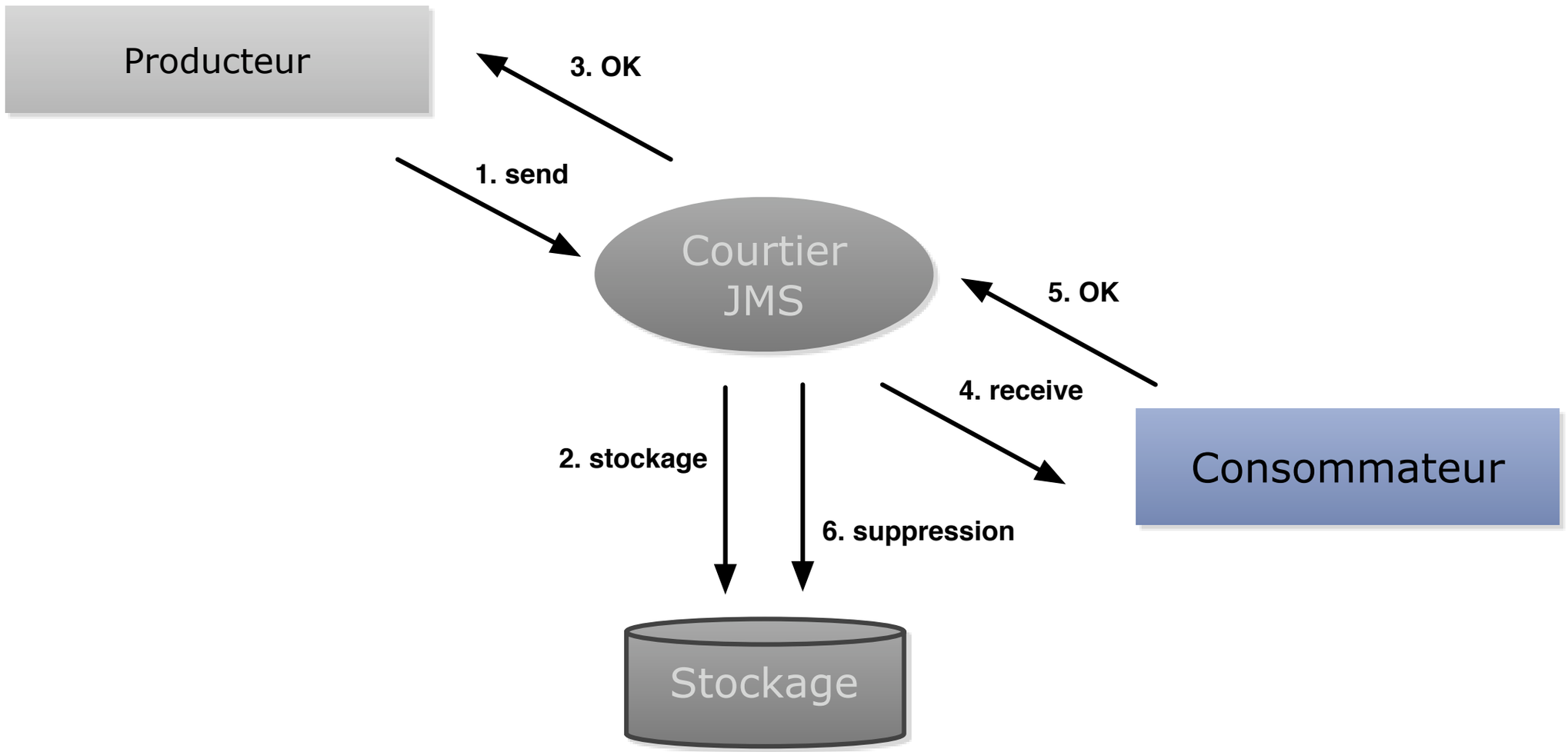
```
connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

- `AUTO_ACKNOWLEDGE` signifie que respectivement lors du send le client JMS sera en attente d'un accusé de réception du courtier, et inversement lors du message `receive`. Si un incident survient, le message n'est pas considéré comme reçu et peut être renvoyé à nouveau pour le courtier. La figure suivante illustre ce concept.

Fiabilité de la communication

Acquittement des messages

Le message est supprimé après accusé de réception :



Fiabilité de la communication

Acquittement des messages

Acquittement gérés par le système (suite) :

- En mode automatique, l'acquittement est envoyé après chaque message. Si le message est renvoyé à nouveau, il portera un drapeau `JMSRedelivered` dans son en-tête. Le courtier utilise l'attribut `JMSMessageID` pour vérifier que les messages sont envoyés une fois et une seule.
- En mode `CLIENT_ACKNOWLEDGE`, le consommateur a la responsabilité d'acquitter le message à tout moment de la session :

```
message.acknowledge();
```

- Il peut ainsi acquitter un paquet de messages d'un coup, ce qui peut optimiser les performances. Si jamais il y a une exception avant l'acquittement global, tous les messages seront renvoyés au prochain `receive`, même ceux qui avaient déjà été reçus.

Fiabilité de la communication

Acquittement des messages

Acquittement applicatif de messages :

- Le modèle de communication JMS est asynchrone, par envoi de requêtes sans réponses. Il peut être souhaitable de recevoir une réponse de manière différée, mais comment alors l'apparier avec la requête initiale ?
- Il existe plusieurs éléments de réponse à ce problème :
  - Utiliser l'attribut `JMSCorrelationID` dans l'en-tête du message.
  - Utiliser des objets `QueueRequestor`;
  - Utiliser des Queues temporaires de réponse (`TemporaryQueue`).
- L'utilisation des `QueueRequestor` est simple : cet objet se substitue au `MessageProducer` et au `MessageConsumer`, et va rester bloqué en attente sur la méthode `request` :

Fiabilité de la communication

Acquittement des messages

Acquittement applicatif de messages :

```
requestor = new QueueRequestor(session, queue);
connection.start();
TextMessage requete = session.createTextMessage();
msg.setText("Hello World!");
Message reponse = requestor.request(requete);
```

- Pendant ce temps, son interlocuteur va devoir répondre de manière particulière. Il va extraire de l'en-tête du message l'adresse de retour et également apparier les numéros de corrélation de la requête et de la réponse.

```
String requete = messageRequete.getText();
Queue replyQueue = (Queue) messageRequete.getJMSReplyTo();
if( replyQueue != null ) {
   //renvoi la réponse
   TextMessage reponse = session.createTextMessage("Reponse");
   reponse.setJMSCorrelationID(aMessage.getJMSMessageID());
   MessageProducer producteur...
   producteur.send(replyQueue, reponse);
}
```

Fiabilité de la communication

Acquittement des messages

Acquittement applicatif de messages :

- Tout se passe comme si vous demandiez un document officiel à une administration en incluant dans votre courrier une enveloppe de retour à votre adresse.
- Le correspondant peut récupérer la destination de retour dans le champ `JMSReplyTo` du message.

Fiabilité de la communication

Utilisation des transactions

Les transactions ne sont pas réservées aux bases de données. La motivation d'une transaction en JMS est de permettre de rendre des messages solidaires au sein d'une session **afin de s'assurer qu'ils sont tous envoyés ou reçus, ou aucun d'entre eux**.

Dans l'exemple de requête-réponse précédent, deux opérations interviennent sur l'interlocuteur :

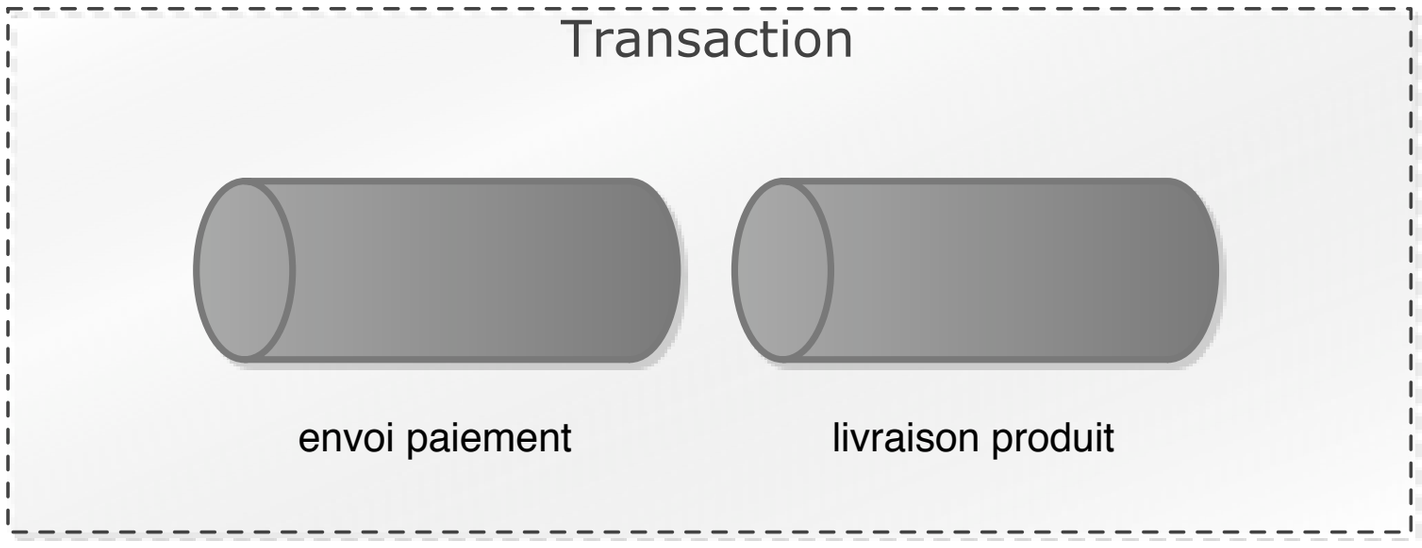
- La réception de la requête.
- L'envoi de la réponse.

Il est souhaitable que la réponse n'ait lieu que si la requête a été bien reçue, et de même que la requête bien reçue induise une réponse. On peut donc lier ces deux opérations dans une même transaction (cf. figure suivante). La délimitation de la transaction sera le début et la fin de la session.

Fiabilité de la communication

Utilisation des transactions

Concrètement, dans le flux de circulation d'information, on voudra par exemple s'assurer de déclencher une livraison de produit uniquement si l'on a reçu le paiement, et inversement mettre obligatoirement en livraison toutes les commandes payées.



Fiabilité de la communication

Utilisation des transactions

La transaction et attachée à la session lors de sa création (booléen `true` pour indiquer la présence d'une transaction) :

```
connexion.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

La session peu alors procéder à des émissions et réceptions de messages. La transaction sera validée lors du commit et invalidée lors du rollback.

```
1. try {
2.   messagePaiement = consommateur.receive();
3.   producteur.send(messageLivraison);
4.   session.commit();
5. } catch(Exception e) {
6.   session.rollback();
7. }
```

Si une exception survient entre 2. et 3., la session ne sera pas validée, **le message de paiement n'aura pas été reçu et l'ordre de livraison n'aura pas été envoyé.**

Conclusion

JMS est une API définie dans Java EE qui permet deux modes de communication : par messages ou par événements.

Le mode par événement permet une diffusion fiable d'une information vers plusieurs interlocuteurs.

L'API JMS comporte des mécanismes sophistiqués de gestion de la qualité de service (transaction, acquittement, priorités), mais cela ne dispense pas de valider chaque implémentation pour connaître le réel niveau de qualité de service mis en oeuvre.

Les communications transitent en général par un stockage intermédiaire qui peut induire des aléas de performance en fonction de la charge du courtier.

Certains éditeurs introduisent des solutions hautement disponibles pour augmenter la fiabilité en couplant plusieurs courtiers.

Fin du cours #6